Алексей Поляков
Виталий Брусенцев

bhv®

Методы и алгоритмы компьютерной графики

в примерах на Visual C++

2-е издание

+ CD-ROM

- Основы компьютерной графики
- Работа с векторной и растровой графикой
- Приемы программирования
- Обзор основных графических библиотек
- Использование возможностей библиотеки GDI+

УДК 681.3.068+800.92 Visual C++
ББК 32.973.26-018.1
П54

Поляков А. Ю., Брусенцев В. А.

П54 Методы и алгоритмы компьютерной графики в примерах на Visual C++, 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2003. — 560 с.: ил.

ISBN 5-94157-377-4

В книге последовательно, "от простого к сложному", рассматриваются понятия, алгоритмы и методы компьютерной графики, а также средства программирования. Описаны особенности платформ Windows и .NET, разработка программ в среде Visual C++ с использованием объектно-ориентированного стиля программирования, возможности, предоставляемые библиотекой MFC и архитектурой Document-View, создание многопоточных приложений с MDI-интерфейсом. Подробно анализируются возможности библиотеки нового поколения GDI+: рисование векторных примитивов сложной формы с градиентной заливкой, управление прозрачностью векторных и растровых объектов, поддержка форматов графических файлов (BMP, GIF, TIFF, JPEG и др.), отрисовка растров с наложением альфа-канала, масштабированием, растяжением, искажением и поворотом. К книге прилагается компакт-диск с примерами программ и изображений.

Для студентов и программистов

УДК 681.3.068+800.92 Visual C++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Анатолий Адаменко*
Зав. редакцией	*Григорий Добин*
Редактор	*Дарья Масленникова*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Елена Самсонович*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

**Лицензия ИД № 02429 от 24.07.00. Подписано в печать 14.08.03.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 45,15.
Тираж 3 000 экз. Заказ №1072
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.**

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02 от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

**Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.**

ISBN 5-94157-377-4

Содержание

Структура книги

Книга состоит из 15 глав и тематически может быть разделена на пять частей. Первая часть — изучение основных понятий компьютерной графики и средств программирования (*главы 1–3*). Вторая часть — векторная графика (*главы 4–8*). Третья часть — растровая графика (*главы 9–11*). Четвертая часть — использование библиотеки GDI+ (*главы 12–14*). Пятая часть — назначение графических библиотек (*глава 15*).

Глава 1 посвящена рассмотрению задач, при решении которых используется компьютерная графика. В ней также приводятся основные понятия и определения.

В *главе 2* большое внимание уделено основным средствам, которые будут использованы далее при программировании всех примеров книги. Среди них: особенности программирования "под Windows"; основные понятия и принципы объектно-ориентированного программирования; назначение библиотеки MFC; процесс создания приложения в MS Visual C++.

В *главе 3* рассматривается создание "графического" приложения Painter, в котором пользователь сможет нарисовать свой первый рисунок, сохранить его в файл и вывести на печать.

В *главе 4* более детально рассматриваются используемые программные средства, добавляется новая функциональность в программу Painter, проектируется иерархия классов графических объектов.

Глава 5 посвящена рассмотрению математических основ преобразований на плоскости.

В *главе 6* материал пятой главы находит свое практическое применение, а также рассматривается реализация в программе Painter некоторых функций редактирования создаваемых изображений.

Глава 7 посвящена теории и практике преобразований в трехмерном пространстве. В программе Painter реализуется построение трехмерной поверхности $z = f(x, y)$ и линий уровня на поверхности.

В *главе 8* рассматриваются математические основы и программная реализация построения сплайновых кривых.

Глава 9 посвящена использованию растровых ресурсов (пиктограмм, курсоров, растровых картинок) в приложениях.

В *главе 10* рассматривается растровый формат BMP, в программе Painter реализуется экспорт изображений в BMP-файл.

Глава 11 целиком посвящена работе с растровыми изображениями. В ней рассматривается создание и применение графических фильтров для обработки изображений. Разрабатывается программа, позволяющая загрузить, обработать с помощью цифровых фильтров и сохранить растровые изображения. Программа иллюстрирует разработку многодокументных, многопоточных приложений. Эту программу вы сможете использовать на практике для улучшения качества своего электронного фотоархива, а расширив ее возможности — добиться неповторимых эффектов.

В *главе 12* читателю сообщаются необходимые сведения о назначении и структуре библиотеки GDI+, новых возможностях для работы с графикой, создании программ на Visual C++ и на Visual C# с применением GDI+.

Глава 13 дает читателю достаточно полное представление о работе с растрами средствами GDI+, классе Bitmap и поддержке стандартных графических форматов, выводе растров на экран, создании анимации и специальных эффектов.

Глава 14 посвящена возможностям GDI+ при работе с векторной графикой. Рассматриваются темы: векторные примитивы и геометрия GDI+; графические объекты и их состояние; вывод различных векторных примитивов — программа Clock; регионы и траектории; использование метафайлов; системы координат.

В *главе 15* рассматривается назначение библиотек OpenGL и DirectX, приводится пример использования OpenGL для визуализации трехмерной сцены и вывода на экран растрового изображения.

Моему брату

Предисловие к первому изданию

Эта книга представляет собой курс основ компьютерной графики. Здесь изложены алгоритмы и методы решения многих задач, возникающих при работе с векторными и растровыми изображениями. Основное внимание уделено вопросам прикладного программирования с использованием языка Microsoft Visual C++ и MFC. Материал книги накоплен за несколько лет чтения курса лекций "Компьютерная графика" в Томском государственном университете систем управления и радиоэлектроники (ТУСУР), а также в результате практической работы над научными и коммерческими проектами.

Основная цель книги — показать, как можно запрограммировать ту или иную задачу компьютерной графики: построить сплайновую кривую, нарисовать поверхность, отобразить на этой поверхности линии уровня, управлять несколькими объектами-фигурами в программе, сохранить изображение, считать растровое изображение с диска, обработать и записать обратно на диск, распечатать изображение на принтере.

Другая задача — продемонстрировать различные пути и способы достижения одинаковых результатов. Поэтому при изучении материала книги старайтесь находить новые, более удачные варианты решения рассмотренных задач.

Весь теоретический материал книги иллюстрируется рабочими примерами программ. В качестве языка программирования выбран C++, поскольку он представляется наиболее подходящим для решения серьезных задач компьютерной графики. В примерах применяется объектно-ориентированный стиль программирования (ООП). Используя стиль ООП, легко представить программу в виде отдельных частей (модулей, деталей), взаимодействующих между собой, если же вы склонны к философствованию, то можно мыслить о программе, как о сообществе неких индивидуумов, которые могут рождаться, обретать какие-то свойства, общаться между собой и умирать.

Каждая программа, описанная в книге, представляет собой законченное приложение. Законченное в том смысле, что его можно откомпилировать,

запустить, посмотреть, что оно делает, сохранить результаты работы. Для работы с примерами вам потребуется компьютер с операционной системой MS Windows 95 (или более поздней версией) и среда разработки MS Visual C++ 6.0.

Поскольку данная книга является учебником по компьютерной графике, а не справочным пособием по Windows API или GDI, то в ней, как правило, не приводится подробного описания параметров вызова API-функций или полного описания методов классов MFC. Эти сведения могут быть легко получены из электронной библиотеки Microsoft Developer Network (MSDN) Library или специальной литературы.

На кого рассчитана эта книга

Книга предназначена студентам и преподавателям, специализирующимся в области информатики и вычислительной техники.

Предполагается, что читатель уже имеет определенный опыт программирования. Однако изложение в книге ведется "от простого к сложному", основные этапы создания программ описаны достаточно подробно. Для понимания теоретического материала требуются начальные знания из области линейной алгебры и тригонометрии. Необходимым условием для успешного усвоения практической части книги является знание языка программирования C++ и знакомство с основными идеями объектно-ориентированного программирования.

Благодарности

Я благодарен преподавателям и студентам кафедры компьютерных систем управления и проектирования ТУСУР за участие в обсуждении многих рассмотренных в книге тем. Хочу выразить отдельную признательность моему научному руководителю, кандидату технических наук Леониду Ивановичу Бабаку. Спасибо студентам Елене Завадской и Сергею Цибенко за участие в деле построения поверхностей и линий уровня.

Я очень признателен руководству компании Элекард **http://www.elecard.com** за предоставленное в мое распоряжение необходимое оборудование, а своим коллегам по фирме Moonlight Russia — за ценное общение.

Выражаю глубокую благодарность за поддержку своим родителям Анне Афанасьевне и Юрию Антониновичу.

Особое спасибо жене Марине за терпение, заботу и вдохновение.

Алексей Поляков

Отзывы читателей на первое издание книги "Методы и алгоритмы компьютерной графики на Visual C++"

"Мне очень понравилась ваша книга, все так здорово и доступно описано, простым человеческим языком (если можно так выразиться :)"
Николай Кириченко

n-Y-c@nwgsm.ru
ООО "Русская коллекция", Оператор ФНА, Предпечатная подготовка
Санкт-Петербург, Россия

"С большим удовольствием прочитал Вашу книгу "Методы и алгоритмы компьютерной графики...". Книга на самом деле замечательная. ... У меня такое субъективное впечатление, что наконец пошла новая волна в отечественном (в какой-то степени отечество у нас общее) творчестве.
...Я думаю категориями конкретных задач. У меня не академический интерес. Я работаю для промышленности и... хочу видеть в книге живые ситуации и думать — ага, а в моем случае это будет так-то и так-то. Пишите дальше, у Вас здорово получается".

Дмитрий Удовицкий
info@techno-sys.com
ООО "Технос", директор.
Разработка систем автоматизированного проектирования и технологической подготовки производства.
г. Николаев, Украина

"Я просто хотел сказать спасибо за книжку "Комп. граф. в примерах на VC++". Очень помогло. Было просто не найти материала, MSDN слишком велика, чтобы там что-то отыскать :)"

Дмитрий Алексеевич Лубенский,
Department of Computer Science, программист, проект "Music Recognition"
Joensuu, Finland.

Предисловие ко второму изданию

Выход второго издания книги означает, что книги первого издания нашли своего читателя, а это радует. Со времени первого издания книги прошел всего лишь год, а базовые принципы мироустройства меняются не так быстро, поэтому материал, изложенный в книге, не утратил своей актуальности. Положительные отзывы читателей свидельствуют о том, что выбранный подход, заключающийся в опробировании всего теоритического материала на практике, оправдывает себя.

В то же время прогресс не стоит на месте. Это нашло отражение в том, что в книгу добавлены три главы, в которых рассмотрены перспективные новинки в технологии программирования: графическая библиотека **GDI+**, платформа **.NET Framework** и язык разработки **C#**. Поскольку книга посвящена компьютерной графике, то основное внимание уделено изучению библиотеки **GDI+**. Надо отметить, что **GDI+** — это действительно библиотека нового поколения с существенно расширенными возможностями, которые позволяют рядовым программистам достичь уровня графики и функциональности, сравнимого с продуктами класса CorelDRAW! и Adobe Photoshop. Вот лишь некоторые новшества и достоинства новой библиотеки:

- **Поддержка популярных форматов графических файлов:** необычайно приятное новшество для всех программистов, имеющих дело с разными графическими форматами. Поддерживаются форматы **BMP**, **GIF**, **TIFF**, **JPEG**, **Exi** (расширение **TIFF** и **JPEG** для цифровых фотокамер), **PNG**, **ICO**, **WMF** и **EMF**. Декодеры различных форматов выполнены с учетом их специфики, так что возможно, например, отобразить анимационный **GIF** или добавить комментарий к **TIFF**-файлу. Загруженный, созданный или модифицированный файл может быть сохранен на диск в одном из подходящих форматов.
- **Работа с растрами:** теперь можно практически все! Поддерживается отрисовка растров с наложением внешнего альфа-канала, масштабированием, растяжением, искажением и поворотом растров. При этом можно установить

режимы отображения отдельных пикселов — от простого переноса до префильтрации (наилучшее качество изображения). Стало возможным рисовать векторные примитивы, залитые текстурами.

- **Градиентная закраска:** позволяет заливать сложные фигуры оттенками с различными законами распределения цвета, рисовать векторные примитивы (например, линии) с градиентной окраской.
- **Поддержка прозрачности:** можно создавать кисти и растры с прозрачными и полупрозрачными областями, заливать области полупрозрачным цветом, назначать Color Key для растрового изображения и работать с его альфа-каналом, а также рисовать полупрозрачные (!) векторные примитивы и текст.

Причем благодаря объектно-ориентированному интерфейсу и новому дизайну графических функций/объектов использовать **GDI+** просто и удобно. На момент написания этих строк авторам не известно о наличии других книг на русском языке, в которых бы рассматривалась тема **GDI+**.

Материал книги первого издания также подвергся некоторым изменениям. Во-первых, благодаря откликам читателей удалось исправить ряд допущенных ошибок. Во-вторых, существенной переработке подверглась *глава 11* — материал дополнился расмотрением фильтров для удаления шума с растровых изображений, структура примера переработана так, чтобы обеспечить многопоточное выполнение программы. Использование дополнительных потоков для преобразований позволяет одновременно выполнять продолжительные операции в нескольких изображениях и при этом не терять контроль над выполнением программы.

Хочется еще раз подчеркнуть, что основная идея книги — дать читателю широкое представление о возможностях компьютерной графики и показать практичесие приемы решения ряда задач. При таком подходе, конечно, трудно претендовать на полноту изложения, в частности, такие интересные библиотеки как Open GL и Direct X в книге рассмотрены весьма кратко. Тем не менее мы уверены, что книга получилась интересной и полезной.

Обратная связь

Если у вас возникнут вопросы, замечания, пожелания, критика и предложения, пожалуйста, направляйте их в "книгу жалоб и предложений" по адресу **graphics@elecard.net.ru**. Мы будем рады, если вы поделитесь с нами своими впечатлениями. Ответы на часто задаваемые вопросы будут публиковаться по адресу **http://www.elecard.com/graphics**.

Благодарности

Прежде всего, хочу поблагодарить всех читателей, приславших свои отзывы на первое издание книги. Добрые слова и поддержка послужили стимулом для дальнейшего развития, а замечания — совершенствования. Особенно я хотел бы поблагодарить за отзывы и комментарии Дмитрия Удовицкого (г. Николаев, Украина), Николая Кириченко (Санкт-Петербург), Дмитрия Лубенского (Joensuu, Finland), Романа Худеева (компания Alparysoft, Томск).

Большое спасибо Алексею Соколову (компания Darim, Томск) за то, что он взял на себя труд прочитать книгу и сделал ценные замечания.

Хочу еще раз выразить свою признательность коллегам по фирме Moonlight Cordless Ltd. (Russian developer center), а также сотрудникам компании Elecard за творческую и дружескую атмосферу (чему в немалой степени способствует зеленый чай нашего доктора Ромы Позднякова), ну и, конечно, за обмен опытом и просвещение сослуживцев, в чем особо преуспели Андрей Поздняков, Петр Губанов и Александр Иванов.

В значительной степени второе издание книги состоялось благодаря Виталию Брусенцеву, который написал три замечательных главы про GDI+.

Хочется также поблагодарить создателей и авторов сайта **www.rsdn.ru** — Russian Software Developer Network за разработку и поддержку прекрасного ресурса, в котором можно найти статьи и форумы по широкому кругу вопросов программирования, в том числе посвященных темам компьютерной графики: GDI, GDI+, OpenGL, DirectX. Большое спасибо работникам издательства "БХВ-Петербург" и нашему редактору Дарье Масленниковой (отловившей массу ошибок) — именно их нелегкий труд превращает электронный текст в книжки с глянцевой обложкой.

Но Самое Большое Спасибо — моим жене и сыну, благодаря им я вообще что-то делаю :-).

Алексей Поляков

Часть I

ОСНОВЫ КОМПЬЮТЕРНОЙ ГРАФИКИ И СРЕДСТВ ПРОГРАММИРОВАНИЯ

Глава 1

Основные понятия компьютерной графики

В данной главе рассматриваются:

- цели и задачи компьютерной графики;
- основные понятия и определения.

1.1. Цели и задачи компьютерной графики

Понятие "компьютерная графика" объединяет довольно широкий круг операций по обработке графической информации с помощью компьютера. Причем наблюдается явная тенденция "компьютеризации" изображений, циркулирующих в обществе. Стали обыденностью термины "цифровое фото" и "видео". Западные кинорежиссеры давно уже пытаются испугать нас ужасами будущего, захваченного компьютерными монстрами, подсовывающими людям виртуальный суррогат вместо прекрасной реальности. В виртуальных буднях грядущего компьютерной графике отводится огромная роль. Это связано с тем, что, по мнению ученых, исследующих проблемы мозга, зрительная система в иерархии мозговых структур человека занимает особое место. С восприятием и обработкой визуальной информации непосредственно связано примерно 20% мозга человека. Благодаря зрению мы получаем по разным оценкам от 70 до 90% сведений об окружающем мире. Следовательно, образный мир компьютерной графики является одним из глубинных проявлений человеческой природы.

В компьютерной графике можно выделить несколько основных направлений.

- Визуализация научных (расчетных или экспериментальных) данных. Большинство современных математических программных пакетов (например, Maple, MatLab, MathCAD) имеют средства для отображения графиков, поверхностей и трехмерных тел, построенных на основе каких-либо расчетов. Кроме того, графическая информация может активно

использоваться в самом процессе вычислений. Например, в системе Image, разработанной на кафедре компьютерных систем управления и проектирования Томского университета систем управления и радиоэлектроники, визуальные образы, выводимые на экран, являются основой для решения математических и проектных задач. Визуализация позволяет представить большой объем данных в удобной для анализа форме и широко используется при обработке результатов различных измерений и вычислений.

- Геометрическое проектирование и моделирование. Это направление компьютерной графики связано с решением задач начертательной геометрии — построением чертежей, эскизов, объемных изображений с помощью программных систем, получивших название CAD-системы (от английского Computer-Aided Design), например AutoCAD.

  Существует большое количество специализированных CAD-систем в машиностроении, архитектуре и т. д.

- Распознавание образов. Способность распознавать абстрактные образы считают одним из важнейших факторов, определившим развитие мыслительных способностей человека, выделив его из животного мира[1]. Задача распознавания и классификации графической информации является одной из ключевых и при создании искусственного интеллекта. Уже в наши дни компьютеры распознают образы повсеместно (системы идентификации футбольных хулиганов у входа на стадион; анализ аэро- и космических фотоснимков; системы сортировки, наведения и т. д.). Возможно, самый известный пример распознавания образов — сканирование и перевод "фотографии" текста в набор отдельных символов, формирующих слова. Такую операцию позволяет выполнить программное обеспечение многих современных сканеров. Кроме того, существуют специализированные программы распознавания текста, например FineReader.

- Изобразительное искусство. К этому направлению можно отнести разнообразную графическую рекламу: от текстовых транспарантов и фирменных знаков до компьютерных видеофильмов, обработку фотографий, создание рисунков, мультипликацию и т. д. В качестве примера популярных

---

[1] "Символические изображения, будь то животные, нарисованные на стенах пещер, или лунные календари, вырезанные на кости животных, ассоциируются исключительно с нашим видом. Возможно, в результате произошедшего небольшого щелчка в устройстве нашего мозга вкупе с некоторым "переключением проводов" наши предки обрели тот тип мышления, который был недоступен неандертальцам, — а именно способность к распознаванию абстрактных символов. И как только этот скрытый интеллектуальный потенциал начинал использоваться вследствие изобретения способов символьной коммуникации — произошедшего, скажем, 35 000 лет назад, его обладатели должны были быстро выйти на совершенно новый уровень технологий". — Майкл Л. Ротсчайльд. Биономика.

программ из этой области компьютерной графики можно назвать Adobe Photoshop (обработка растровых изображений), CorelDRAW (создание векторной графики), 3DS Max (трехмерное моделирование).

- Виртуальная реальность. Реальность, даже виртуальная, подразумевает воздействия на всю совокупность органов чувств человека, в первую очередь на его зрение. К компьютерной графике можно отнести задачи моделирования внешнего мира в различных приложениях: от компьютерных игр до тренажеров. Кроме того, не стоит забывать о компьютерах-злодеях, которые используют виртуальную реальность для захвата мира. Поэтому надо изучать компьютерную графику, чтобы не дать себя провести :-).
- Цифровое видео. Все более широкое распространение получают анимированные изображения, записанные в цифровом формате. Это прежде всего фильмы, передаваемые через компьютерные сети, а также видеодиски Digital Video Disk (DVD), цифровое, кабельное и спутниковое телевидение.

Приведенная классификация сфер применения компьютерной графики является во многом условной. Возможно, найдутся задачи, которые нельзя отнести ни к одному из обозначенных направлений.

1.2. Основные понятия и определения

1.2.1. Графический формат

Графическим форматом называют порядок (структуру), согласно которому данные, описывающие изображение, записаны в файле.

Графические данные обычно разделяются на два класса: *векторные* и *растровые*. Изображения, в зависимости от типа описывающих их данных, называются векторными или растровыми.

Векторные данные используются для представления прямых, многоугольников, кривых и т. д. с помощью определенных в числовом виде *базовых* (опорных, контрольных, ключевых) точек. Программа, обрабатывающая векторные данные, воспроизводит линии посредством соединения базовых точек. Вместе с информацией о базовых точках хранятся атрибуты (цвет, толщина и другие параметры линий) и набор правил (соглашений) вывода (рисования). Пример векторного изображения приведен на рис. 1.1.

Растровые данные представляют собой набор числовых значений, определяющих яркость и цвет отдельных *пикселов*. Пикселами (или пикселями — от английского *pixel*) называются минимальные элементы (цветные точки), из которых формируется растровое изображение. Термин "растр" в компьютерной графике и полиграфии имеет несколько отличающиеся значения.

Далее под растром будем понимать массив пикселов (массив числовых значений). Для обозначения массива пикселов часто используется термин *bitmap* (битовая карта). В *bitmap* каждому пикселу отводится определенное число битов (одинаковое для всех пикселов изображения). Это число называется *битовой глубиной* пиксела или *цветовой глубиной* изображения, т. к. от количества битов, отводимых на один пиксел, зависит количество цветов изображения. Наиболее часто используется цветовая глубина 1, 2, 4, 8, 15, 16, 24 и 32 бита.

Рис. 1.1. Векторный рисунок

Источниками растровых данных могут быть программы, формирующие изображение на растровом экране и различного рода устройства для ввода изображений (сканеры, цифровые камеры и др.).

Пример растрового изображения приведен на рис. 1.2.

Рис. 1.2. Растровый рисунок

Типы форматов графических файлов определяются способом хранения и типом графических данных. Наиболее широко используются растровый, векторный и метафайловый форматы.

Векторный формат наиболее удобен для хранения изображений, которые можно разложить на простые геометрические фигуры (например, чертежи или текст). Векторные файлы содержат математические описания элементов изображения. Наиболее распространенные векторные форматы: AutoCAD DXF и Microsoft SYLK.

Растровый формат используется для хранения растровых данных. Файлы такого типа особенно хорошо подходят для хранения изображений реального мира, например оцифрованных фотографий. Растровые файлы содержат битовую карту изображения и ее спецификацию. Наиболее распространенные растровые форматы: BMP, TIFF, GIF, PCX, JPEG.

Метафайловый формат позволяет хранить в одном файле и векторные, и растровые данные. Примером такого формата являются файлы CorelDRAW — CDR.

Кроме того, существуют файловые форматы для хранения мультипликации (видеоинформации), мультимедиа-форматы (одновременно хранят звуковую, видео- и графическую информацию), гипертекстовые (позволяют хранить не только текст, но и связи-переходы внутри него) и гипермедиа (гипертекст плюс графическая и видеоинформация) форматы, форматы трехмерных сцен, форматы шрифтов и т. д.

1.2.2. Элементы графического файла

Графические файлы состоят из последовательности данных или структур данных, называемых *файловыми элементами* или *элементами данных*. Эти элементы подразделяются на три категории: поля, теги и потоки.

- *Поле* — это структура данных в графическом файле, имеющая фиксированный размер. Для определения положения поля в файле обычно задают либо абсолютное смещение от начала или конца файла, либо смещение относительно другого поля.
- *Тег* — это структура данных, размер и позиция которой изменяются от файла к файлу. Позиция тега может задаваться абсолютно, либо относительно другого файлового элемента. Теги могут содержать в себе другие теги или наборы связанных полей.
- *Поток* — набор данных, предназначенный для последовательного чтения. В отличие от полей и тегов поток не обеспечивает быстрого доступа к нужным данным, т. к. их положение в файле определяется в процессе чтения.

Как правило, в графических файлах применяются комбинации этих элементов данных.

1.2.3. Преобразование форматов

Часто возникает задача преобразования формата данных, связанная с необходимостью обмена изображениями между различными программами. Для преобразования данных существуют специализированные программы. Кроме того, распространенные графические редакторы позволяют читать и сохранять изображения в различных форматах. Поэтому преобразование растровых данных из одного формата в другой обычно не представляет сложности. Другое дело — преобразование векторных изображений в растровые. При таком преобразовании неизбежно теряется часть информации, т. к. происходит переход от "идеального" математического описания рисунка к его дискретному (растровому) представлению. Обратное преобразование из растрового формата в векторный вообще является нетривиальной задачей, связанной с распознаванием образов.

1.2.4. Сжатие данных

Сжатие — процесс уменьшения физического размера блока данных. Так как изображения, как правило, описываются большим количеством данных, файлы изображений имеют большой размер. Поэтому графические данные часто подвергаются сжатию. Обычно каждый формат графического файла поддерживает какой-либо из методов (алгоритмов) сжатия. В большинстве случаев сжатие заключается в замене избыточной информации на ее более компактную форму. Сжатие бывает физическое и логическое. Различие между физическим и логическим сжатием заключается в методе получения более компактной формы данных. Физическое сжатие данных выполняется без учета содержащейся в них информации. Логическое сжатие, напротив, основано на логическом анализе информации. Примером логического сжатия может служить преобразование строки "Союз Советских Социалистических Республик" в аббревиатуру "СССР". Для графических данных логическое сжатие не применяется.

Методы сжатия бывают *с потерями* и *без потерь*. Когда данные сжимаются, а затем восстанавливаются (распаковываются), причем полученные данные полностью соответствуют исходной информации, то говорят, что имело место сжатие без потерь. То есть при методе сжатия без потерь не должно происходить какого-либо изменения данных.

Методы сжатия с потерями предусматривают отбрасывание некоторой части данных изображения для достижения большей степени сжатия.

Некоторые наиболее распространенные методы сжатия.

- Упаковка пикселов. Метод заключается в компактной записи пикселов с глубиной 1, 2 и 4 бита компактно в 8-битовые байты соответственно по 8, 4 и 2 штуки.

- Групповое кодирование (Run-Length Encoding, RLE) — является общим алгоритмом кодирования и применяется в таких растровых форматах, как BMP, TIFF, PCX.
- Алгоритм Lempel-Ziv-Welch (LZW) — применяется в форматах GIF, TIFF.
- Алгоритм JPEG, разработанный объединенной экспертной группой по фотографии, включает в себя целый набор методов сжатия. Базовая реализация JPEG применяет схему преобразования изображения по алгоритму дискретных косинус-преобразований с последующим кодированием методом Хаффмана.
- Фрактальное сжатие — математический процесс, используемый для кодирования растровых изображений в совокупность математических данных, которые описывают фрактальные (похожие, повторяющиеся) свойства изображения.

Сжатие в основном применяется к данным растровых изображений. В растровых файлах сжимаются только данные изображения, другие же данные (заголовок файла, таблица цветов и т. п.) всегда остаются несжатыми.

Векторные файлы сжимаются редко. Это связано с тем, что векторные форматы сами по себе хранят данные в очень компактной форме и сжатие не дает ощутимого эффекта.

Важно уяснить, что алгоритмы сжатия не задают какой-либо файловый формат, а определяют только способ кодирования данных.

1.2.5. Пикселы и цвет

Различают физические и логические пикселы.

Физические пикселы — реальные точки, отображаемые на устройстве вывода — наименьшие элементы на поверхности отображения, которыми можно манипулировать. При выводе на экран или принтер один физический пиксел обычно формируется из нескольких более мелких цветовых точек. Например, один цветной пиксел на мониторе формируется из трех более мелких точек красного, зеленого и синего цветов, яркость которых и определяет цвет пиксела.

Логические пикселы подобны чертям на кончике иглы[1] — они имеют местоположение и цвет, но не занимают физического пространства (то есть нельзя вычислить высоту и ширину такого пиксела). По сути логический пиксел — это всего лишь некоторое число, которое задает его цвет. Положение же логического пиксела (его координаты) определяется его местом в карте изображения и количеством пикселов на единицу измерения (разрешением).

[1] "Сколько чертей уместится на кончике иглы" — известный схоластический спор.

При отображении логических пикселов на физическом экране происходит преобразование численных данных, характеризующих яркость и цвет логических пикселов, в интенсивность свечения физических пикселов. При этом никак не обойтись без учета размера и расположения физических пикселов.

Количество возможных цветов пиксела напрямую связано с отводимым для него количеством битов и равно 2^n, где n — количество битов. Изображения, каждому пикселу которого отводится один бит, называют монохромными — двухцветными. Такие изображения вполне подходят для чертежей и текста. Изображения с глубиной цвета 24 бита и более называют *truecolor* (истинные цвета). Каждый пиксел такого изображения может принимать один из более 16 миллионов цветов. Считается, что этого вполне достаточно, чтобы приемлемо отобразить окружающую нас действительность.

При отображении цветов, заданных для логических пикселов на устройстве визуализации, может возникнуть проблема согласования цветов. Например, если устройство вывода способно отобразить до 16 миллионов цветов, а изображение имеет глубину цвета 8 бит (256 цветов), то проблем с его отображением не будет. Если же все наоборот, то программе визуализации придется потрудиться, выполняя преобразование цветов изображения соответственно возможностям устройства вывода. В последнем случае неизбежна потеря части данных и, как следствие, снижение качества изображения. Кроме того, возможно возникновение всякого рода побочных эффектов (муар, вторичные контуры — артефакты, одним словом).

1.2.6. Палитры цветов

Палитра цветов, называемая также *картой* или *таблицей цветов*, представляет собой одномерный массив цветовых величин. С помощью палитры цвета задаются косвенно, посредством указания их позиции в массиве. При использовании палитры цветов сведения о цветах пикселов записаны в файле в виде последовательности индексов. Использование палитр во многих случаях позволяет значительно сократить объем растровых данных.

Наиболее часто используются палитры из 16 и 256 цветов, соответственно глубина цвета составляет 4 и 8 битов, но могут быть и палитры других размеров.

Например, каждый пиксел изображения с глубиной цвета в 4 бита может иметь 16 цветов (2^4). Эти 16 цветов определены в палитре, которая обычно включается в один файл с растровыми данными. Каждое пикселное значение рассматривается как индекс в этой палитре и содержит одно из значений от 0 до 15. Значения цветов в палитре задаются с максимально возможной точностью. Обычно элемент палитры занимает 24 бита (3 байта). Таким образом, элемент палитры может задавать один из 16 777 216 цветов (2^{24}). Программа, осуществляющая визуализацию изображения, читает из файла растровые данные — индексы в таблице цветов и использует соответствующие

им цвета для окрашивания пикселов экрана (рис. 1.3). Палитры разных изображений чаще всего различаются.

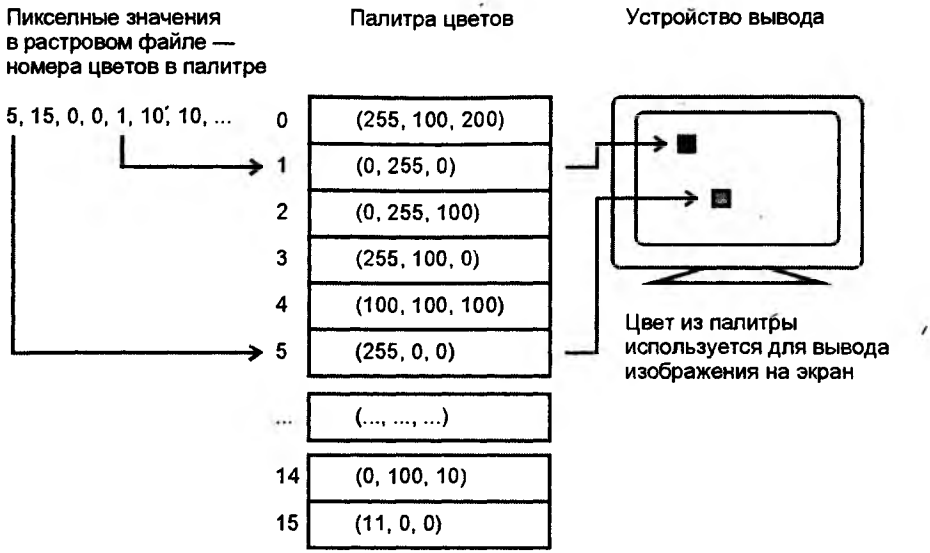


Рис. 1.3. Определение цвета с помощью палитры

1.2.7. Цвет. Цветовые модели

Для описания цветов применяют несколько различных математических систем (моделей). Ни одна из существующих систем представления цвета не является наилучшей. Для разных целей и задач служат различные системы.

В графических файлах обычно используют цветовые модели, основанные на трех основных цветах, которые не могут быть получены смешиванием других цветов. Все множество цветов, которые могут быть получены путем смешивания основных цветов, представляет собой *цветовое пространство*, или *цветовую гамму*.

Цветовые модели могут быть разделены на две категории: *аддитивные* и *субтрактивные*. В аддитивных моделях новые цвета получают путем сложения основных цветов различной интенсивности с черным цветом. Чем больше интенсивность добавляемых цветов, тем ближе к белому результирующий цвет. Белый цвет получается при максимальных значениях интенсивности всех трех основных цветов, черный — при минимальных. Аддитивные модели формирования цвета применяются в самосветящихся устройствах (например, мониторах).

В субтрактивных моделях основные цвета вычитаются из белого. Чем больше интенсивность вычитаемых цветов, тем ближе результат к черному.

Субтрактивные модели применяются при формировании цветных изображений на отражающих носителях, например бумаге.

Наиболее распространенные цветовые модели.

- **Модель RGB** (Red-Green-Blue — красный-зеленый-синий) — модель, наиболее широко используемая в графических форматах. RGB — аддитивная модель. Каждый пиксел представляется в виде трех числовых величин — интенсивностей красного, зеленого и синего цветов. Каждому цвету обычно отводится 8 битов, в которых может быть записано 256 уровней интенсивности. Таким образом, значение (0, 0, 0) представляет черный цвет, а (255, 255, 255) — белый.
- **Модель CMY** (Cyan-Magenta-Yellow — голубой-пурпурный-желтый) — субтрактивная модель, которая применяется для получения цветных изображений на белой поверхности. При освещении изображения, полученного с помощью модели CMY, каждый из основных цветов поглощает дополняющий его цвет: голубой поглощает красный; пурпурный — зеленый; желтый — синий. Теоретически наивысшая интенсивность всех трех основных цветов субтрактивной модели должна обеспечить черный цвет. На практике этого не происходит (так как реальные красители далеки от математических идеалов), поэтому в модель вводят четвертый компонент — черный цвет (Black), обозначаемый буквой "K". В результате получается модель CMYK, широко распространенная в полиграфии. При использовании субтрактивной модели изображение каждого пиксела (цветной точки) изображения состоит из четырех пятен основных цветов.

Существуют и другие модели, не основанные на смешении цветов, например HSV (Hue-Saturation-Value — оттенок-насыщенность-величина). Оттенок — это, по сути, цветовой тон, например, красный, оранжевый, синий и т. д. Насыщенность определяет количество белого в оттенке. Если в полностью насыщенном (100%) оттенке красного не содержится белого, такой оттенок считается чистым. Насыщенность 50% задает более светлый цвет, в нашем примере он будет соответствовать розовому цвету. Величина (яркость) задает интенсивность свечения цвета.

1.3. Заключение

Итак, в этой главе мы коротко рассмотрели основные задачи, при решении которых используются средства компьютерной графики, и дали определения используемым терминам. В следующих главах на практике рассмотрим работу с векторной и растровой графикой. В качестве дополнительной литературы можно порекомендовать [6, 18].

Глава 2

Особенности программирования "под Windows"

В данной главе рассматриваются:

- особенности программирования "под Windows";
- основные понятия и принципы объектно-ориентированного программирования;
- создание приложения в MS Visual C++.

Сегодня, когда операционная система Win32 (имеются в виду все 32-разрядные разновидности Windows) получила широчайшее распространение, и о работе под DOS и Win16 уже стали забывать, программирование под Windows стало обыденностью. Однако не лишним будет подчеркнуть те концепции Windows, благодаря которым она стала столь популярной и которые, наверняка, получат развитие в Win64. Win32 предоставляет программисту такие возможности, о которых лет 10 назад можно было только мечтать. Пожалуй, основными отличиями операционных систем Win32 от их предшественниц является вытесняющая многозадачность и возможность прямой адресации до 4 Гбайт.

Вытесняющая многозадачность означает, что операционная система сама решает, какой из программ предоставить в распоряжение процессор. Каждая программа, отработав некоторое время, автоматически выгружается системой, и управление передается другой задаче. Возможно, кому-то не понравится такой "авторитарный" стиль управления. Зато подобная организация позволяет, во-первых, создать иллюзию одновременного выполнения нескольких программ, и, во-вторых, не допускает возможности полного захвата ресурсов компьютера какой-нибудь сбойной задачей, защищая таким образом систему от "зависания" (но мы-то знаем, что на практике это не всегда так). На мой взгляд, о вытесняющей многозадачности лучше всех сказал Омар Хайям:

> Напрасно ты винишь в непостоянстве рок,
> Что не в накладе ты, тебе и невдомек.
> Когда б он в милостях своих был постоянен,
> Ты б очереди ждать своей до смерти мог.

Вторая особенность Win32 — 32-разрядная адресация в полностью виртуальном адресном пространстве, т. е. любое приложение[1], независимо от того, сколько их одновременно выполняется, может распоряжаться 4 Гбайтами памяти. Причем они не могут случайно повредить данные друг друга, т. к. каждому приложению выделяется свое адресное пространство. Получив в свое распоряжение столь большой ресурс памяти, программисты смогли наконец-то вздохнуть свободно. Интересно, далеко ли то время, когда 4 Гбайт не будет хватать для решения обычных задач (может, оно уже настало?).

Кроме перечисленных особенностей Win32, можно отметить также следующие:

- поддержка многозадачности на уровне процессов и потоков;
- возможность синхронизации потоков;
- существование файлов, проецируемых в память;
- наличие механизмов обмена данными между процессами.

В Win32 процессом называется каждая выполняющаяся программа. Каждый процесс имеет, как минимум, один поток выполнения. Термин "поток" (thread) можно определить как логическое направление выполнения программы. Если программу-процесс сравнить с рекой, то потоки можно сравнить с ее рукавами. Река может разделиться на несколько потоков-рукавов, которые будут течь параллельно друг другу. Например, когда программа должна выполнить какую-нибудь продолжительную операцию, ее целесообразно выделить в отдельный поток, основной же поток программы в это время может продолжать общаться с пользователем. Все потоки, принадлежащие одному процессу, выполняются в одном адресном пространстве и имеют общие с этим процессом код, ресурсы и глобальные переменные.

Для того чтобы несколько потоков слаженно решали поставленные задачи и не мешали друг другу при использовании каких-то общих ресурсов, применяется синхронизация потоков. Синхронизация может потребоваться, например, в случае, когда один из потоков должен дождаться завершения какой-то операции, выполняемой другим потоком, или когда потоки работают с ресурсом, способным одновременно обслуживать лишь один из них. Поскольку потоки выполняются в условиях вытесняющей многозадачности, функции их синхронизации берет на себя операционная система. Для управления потоками в Windows используются специальные флаги, на которых основано действие нескольких механизмов синхронизации: *семафоры* (semaphores), *исключающие* (mutex) семафоры, *события* (event), *критические секции* (critical section).

Каждый процесс в Win32 имеет возможность прямой адресации до 4 Гбайт. Однако 4 Гбайт оперативной памяти пока не очень типичны для большинства персональных компьютеров, да и предоставить каждому выполняемому при-

[1] Так называют свои программы настоящие Win32-программисты ;-)

ложению по 4 Гбайт нереально. Поэтому программы работают с *виртуальными* адресами, которые отличаются от физических адресов памяти. Для того чтобы предоставить в распоряжение программ такой большой объем памяти, Windows использует специальный механизм подкачки памяти с жесткого диска. Этот механизм называется страничной организацией памяти (paging). При такой организации логическое адресное пространство каждого процесса разбито на отдельные блоки, называемые *страницами*. Страницы могут располагаться как в оперативной памяти, так и на жестком диске. Операционная система оптимизирует выделение оперативной памяти таким образом, чтобы процессы могли получить быстрый доступ к часто используемым данным. Диспетчер виртуальной памяти отслеживает давно неиспользуемые страницы памяти и отправляет их в страничный файл на диск.

Видимо, похожий механизм управления памятью используется при создании *файлов, проецируемых в память* (memory-mapped files), называемых еще *отображаемыми файлами*. Проецируемый файл как бы отображает файл на диске в диапазон адресов в памяти. Это позволяет выполнять операции с файлом точно так же, как если бы работа осуществлялась в памяти. Операционная система сама организует обмен данных между памятью и диском. Использование файлов, проецируемых в память, позволяет значительно упростить работу с файлами, а также дает возможность разделять данные между процессами. Разделение данных происходит следующим образом: два или более процессов создают в своей виртуальной памяти проекции одной и той же физической области памяти — объекта "проецируемый файл". Когда один процесс записывает данные в свою "проекцию" файла, изменения немедленно отражаются и в "проекциях", созданных в других процессах. Для организации такого взаимодействия все процессы должны использовать одинаковое имя для объекта "проецируемый файл".

В Win32 каждый процесс имеет свое адресное пространство, поэтому одновременно выполняемые приложения не могут случайно повредить данные друг друга. Однако часто возникает необходимость в обмене информацией между процессами. Для этой цели в Win32 предусмотрены специальные способы, позволяющие приложениям получать совместный доступ к каким-либо данным. Все они основаны на механизме проецирования файлов, описанном выше.

Один из способов заключается в создании "Почтового ящика" (mailslot) — специальной структуры в памяти, имитирующей обычный файл на жестком диске. Приложения могут помещать данные в "ящик" и считывать их из него. Когда все приложения, работающие с ящиком, завершаются, "почтовый" ящик и данные, находящиеся в нем, удаляются из памяти.

Другой способ заключается в организации коммуникационной магистрали — "канала" (pipe), соединяющего процессы. Приложение, имеющее доступ к каналу с одного его "конца", может связаться с приложением, имеющим доступ к каналу с другого его "конца", и передать ему данные. Канал

может выделять приложениям односторонний или двусторонний доступ. При одностороннем доступе одно приложение может записывать данные в канал, а другое — считывать; при двустороннем — оба приложения могут выполнять операции чтения/записи. Существует возможность организовать канал, коммутирующий сразу несколько процессов.

Далее, при реализации алгоритмов компьютерной графики, мы постараемся использовать эти возможности, предоставляемые Windows.

2.1. Типы данных в Windows

В Windows-программах вместо стандартных для C и C++ типов данных (таких, как `int` или `char*`) применяются типы данных, определенные в библиотечных файлах (например, в WINDOWS.H). Часто используются следующие типы:

- HANDLE — 32-разрядное целое в качестве дескриптора (числового идентификатора какого-либо ресурса);
- HWND — дескриптор окна (32- разрядное длинное целое);
- BYTE — 8-разрядное беззнаковое символьное значение;
- WORD — 16-разрядное беззнаковое короткое целое;
- DWORD, UINT — 32-разрядное беззнаковое длинное целое;
- LONG — 32-разрядное длинное целое со знаком;
- BOOL — целое, используется для обозначения истинности (1 — TRUE) или ложности (0 — FALSE);
- LPSTR — 32-разрядный указатель на строку символов.

Кроме перечисленных, существует еще много других типов, обозначающих дескрипторы различных ресурсов, указатели, структуры и т. д. Различия многих из них заключаются лишь в том, что они обозначают. В Windows-программах можно также использовать и все традиционные типы данных.

2.2. Структура Windows-приложения

Работа Windows-программ основана на обработке сообщений, которые поступают от пользователя, операционной системы и других программ. Структура приложений обязательно включает в себя главную функцию `WinMain`, одинаково устроенную для всех приложений. С нее начинается выполнение всех Windows-программ. `WinMain` должна выполнить следующие основные действия:

1. Определить и зарегистрировать класс окна в Windows.
2. Создать и отобразить окно, определяемое данным классом.
3. Запустить цикл обработки сообщений.

При определении класса окна указывается специальная *функция окна*, которая должна реагировать на поступающие окну сообщения. Каждое приложение имеет свою очередь сообщений — ее создает Windows и помещает туда все сообщения, адресованные окну приложения. После создания и отображения окна запускается основной цикл обработки сообщений, в котором сообщения проходят предварительную обработку и возвращаются Windows обратно. Затем Windows вызывает функцию окна программы с очередным сообщением в качестве аргумента. Анализируя сообщение, функция окна инициирует соответствующие операции. Сообщения, обработка которых не предусмотрена функцией окна, передаются Windows, которая выполняет обработку "по умолчанию". Ниже приведена минимальная программа для Windows (листинг 2.1).

Листинг 2.1. Минимальная программа для Windows

```
// Минимальное приложение для Win32
#include <windows.h>
// Прототип функции окна
LRESULT CALLBACK WinFunction (
HWND hwnd,        // Указатель на окно
UINT message,     // Идентификатор сообщения
WPARAM wparam,    // Параметры
LPARAM lparam);   // сообщения

// Имя класса окна
char szWindowName[]="MyClass";

// Функция WinMain
int WINAPI WinMain(
   HINSTANCE hThisInst,  // Дескриптор экземпляра приложения
   HINSTANCE hPrevInst,  // В Win32 всегда NULL
   LPSTR lpszArgs,       // Указатель на строку с аргументами
   int nWinWode)         // Способ визуализации окна при запуске
{
   // Будет содержать указатель главного окна программы
   HWND hwnd;
   // Структура-буфер для хранения сообщений
   MSG msg;
   // Структура для определения класса окна
   WNDCLASSEX wcl;
   wcl.hInstance=hThisInst;          // Дескриптор приложения
```

```
wcl.lpszClassName = szWindowName;  // Имя класса окна
wcl.lpfnWndProc = WinFunction;     // Функция окна
wcl.style=0;                       // Стиль по умолчанию
wcl.hIcon=LoadIcon(NULL, IDI_APPLICATION);   // Пиктограмма
wcl.hCursor=LoadCursor(NULL, IDC_ARROW);     // Курсор
wcl.lpszMenuName=NULL;             // Без меню
wcl.cbClsExtra=0;                  // Без доп. информации
wcl.cbWndExtra=0;   // Без доп. информации
wcl.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);   // Фон

// Регистрация класса окна
if(!RegisterClassEx(&wcl)) return 0;

// Создание окна
hwnd=CreateWindow(
szWindowName,   // Имя класса окна
"Минимальная программа для Windows",   // Заголовок
WS_OVERLAPPEDWINDOW,   // Стиль
CW_USEDEFAULT,         // Координаты x и y
CW_USEDEFAULT,         // верхнего левого угла
CW_USEDEFAULT,         // Ширина
CW_USEDEFAULT,         // Высота окна
HWND_DESKTOP,          // Родительское окно
NULL,                  // Дескриптор меню
hThisInst,             // Дескриптор приложения
NULL);                 // Без доп. информации

// Отображение окна
ShowWindow(hwnd, nWinWode);
// Перерисовка окна
UpdateWindow(hwnd);
// Цикл обработки сообщений
while(GetMessage(
          &msg,  // Буфер для сообщения
          NULL,  // Получать сообщения от всех окон приложения
          0, 0)) // Диапазон получаемых сообщений — все
{
    // Возвращает управление Windows
    DispatchMessage(&msg);
```

```
    }
    return msg.wParam;     // Значение, возвращаемое программой
};
// Функция окна вызывается Windows для обработки сообщений
LRESULT CALLBACK WinFunction (
    HWND hwnd,          // Дескриптор окна
    UINT message,       // Идентификатор сообщения
    WPARAM wparam,      // Параметры
    LPARAM lparam)
{
    switch(message)
    {
        case WM_DESTROY:     // Завершение программы
                PostQuitMessage(0);
                break;
        default:             // Передает необработанные сообщения Windows
                return DefWindowProc(hwnd, message, wparam, lparam);
    }
    return 0;
};
```

Рассмотрим выполнение данной программы. Схематично процесс обработки сообщений показан на рис. 2.1.

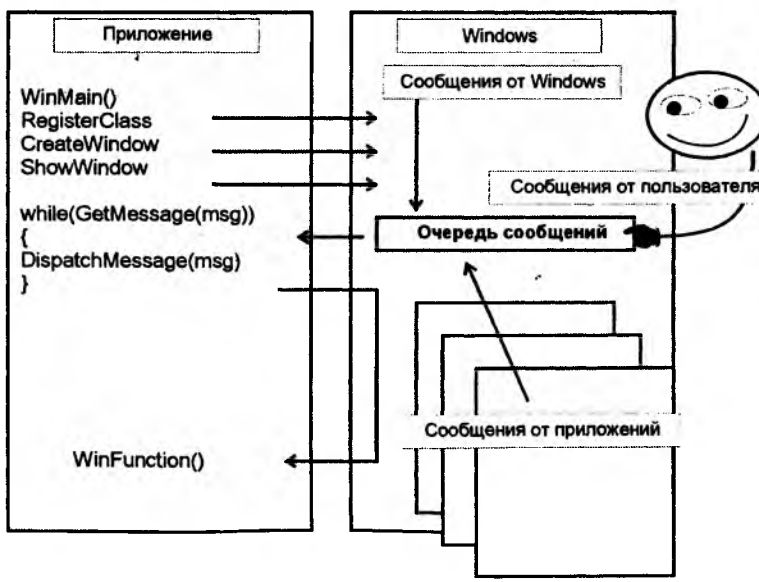


Рис. 2.1. Схема обработки сообщений приложением

Все программы, написанные для Windows, должны включать библиотечный файл WINDOWS.H. Этот файл содержит прототипы API-функций (о них будет рассказано дальше), а также определения различных типов данных, макросов и констант, используемых в Windows. Функция окна, используемая в программе, называется `WinFunction()`. Она объявлена как функция обратного вызова, т. к. вызывается Windows для обработки сообщений. Программа начинает выполнение с вызова функции `WinMain()`. Первое, что выполняет функция `WinMain()`, — определяет класс окна путем заполнения полей структуры `WNDCLASSEX`. С помощью данной структуры программа указывает Windows функцию и стиль окна. Регистрация класса окна выполняется с помощью API-функции `RegisterClassEx()`. После этого создается окно с помощью API-функции `CreateWindow()`, параметры которой определяют внешний вид окна. В конце функции `WinMain()` расположен цикл сообщений. Он является обязательной частью всех приложений Windows. Пока приложение выполняется, оно непрерывно получает сообщения и извлекает их из очереди с помощью API-функции `GetMessage()`. Сообщения поступают через параметр функции типа структура `MSG`. Если очередь сообщений пуста, то вызов `GetMessage()` передает управление Windows. При завершении работы с программой функция `GetMessage()` возвращает ноль. После того как сообщение прочитано, оно передается назад в Windows API-функцией `DispatchMessage()`. Windows хранит сообщение до тех пор, пока не передаст его программе в качестве параметра функции окна. Когда цикл сообщений завершается, функция `WinMain()` также завершает свое выполнение, возвращая Windows значение кода возврата.

Такая схема работы приложения выглядит чрезмерно сложной. Возникает вопрос: для чего приложению извлекать сообщения из очереди сообщений, чтобы затем снова возвратить их Windows? Возможно, такая схема осталась как рудимент от ОС Windows 3.1, в которой многозадачность была не вытесняющей. Тем не менее при подобной организации обработки сообщений приложение получает возможность выполнить предварительную обработку сообщений, а именно:

- фильтровать получаемые сообщения, задавая нужные параметры функции `GetMessage()`;
- транслировать виртуальные коды клавиш в клавиатурные сообщения с помощью функции `TranslateMessage()`;
- транслировать коды нажатия "горячих" клавиш в сообщения команд меню функцией `TranslateAccelerator()`.

С использованием этих функций цикл обработки сообщений может выглядеть следующим образом (листинг 2.2).

Листинг 2.2. Пример стандартного цикла обработки сообщений

```
// Цикл обработки сообщений
while(GetMessage(&msg, NULL, 0, 0))
{
   // Транслировать горячие клавиши
   if(!TranslateAccelerator(hwnd, hAccel, &msg))
   {
      // Транслировать виртуальные клавиши
      TranslateMessage(&msg);
      // Возвращает управление Windows
      DispatchMessage(&msg);
   }
}
```

Любое приложение взаимодействует с Windows через Application Programming Interface (API). API содержит несколько сотен функций, которые реализуют все необходимые системные действия, такие как выделение памяти, создание окон, вывод на экран и т. д. Функции API содержатся в библиотеках динамической загрузки (Dynamic Link Libraries (DLL)), которые загружаются в память только в тот момент, когда к ним происходит обращение.

Одним из подмножеств API является GDI (Graphic Device Interface — интерфейс графических устройств). Задача GDI — обеспечение аппаратно-независимой графики. Благодаря функциям GDI Windows-приложение может выполняться на самых различных компьютерах.

2.3. Создание приложения в MS Visual C++

Среда разработки программ Microsoft Visual C++ предназначена для создания приложений — программ, снабженных всем необходимым для их работы: файлами ресурсов, библиотеками и т. д. В Visual C++ в основном разрабатываются приложения на основе Microsoft Foundation Class Library (MFC) — библиотеки базовых классов объектов фирмы Microsoft, хотя есть возможность построения и других типов программ. Такие приложения представляют собой совокупность объектов, которыми является само приложение и все его компоненты: документы, облики документов, диалоги и т. д. Интерфейс Visual C++ версии 6.0 показан на рис. 2.2. Visual C++ включает удобные средства создания и редактирования всех типов ресурсов, имеет мощные инструменты для генерации каркасов приложений (AppWizard),

а также для создания и управления классами приложения (ClassWizard). Работу с этими инструментами рассмотрим далее на примере разработки конкретных приложений.

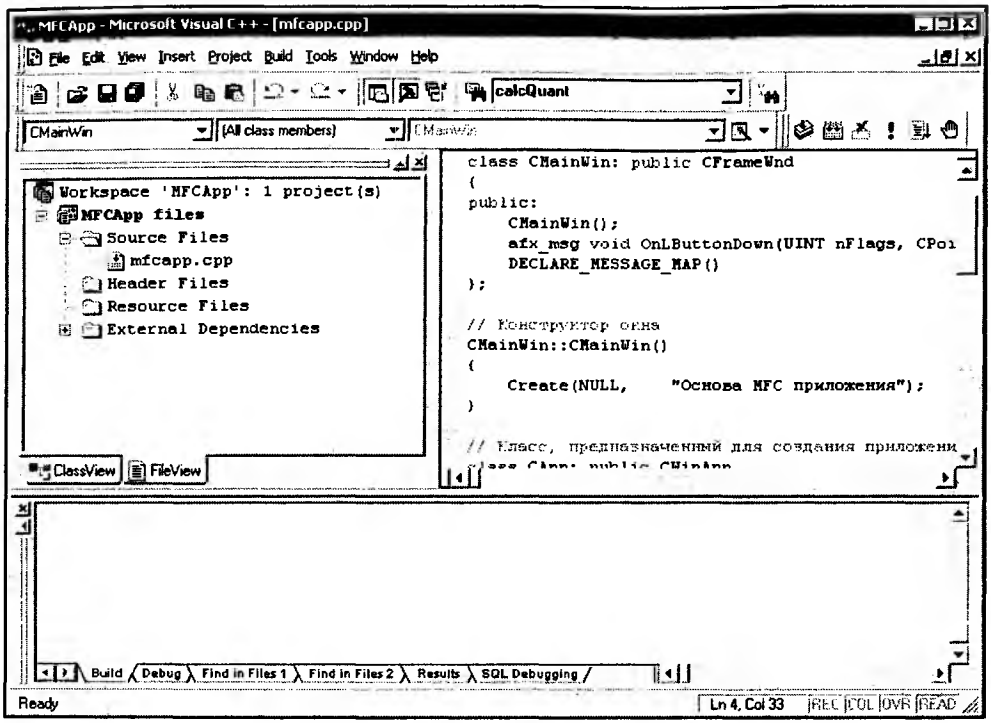


Рис. 2.2. Среда разработки программ Visual C++

Надо отметить, что профессионализм и успех современного программиста состоят не только (часто даже не столько) в умении разрабатывать и реализовывать хитроумные алгоритмы, но и в способности использовать труд огромной армии коллег по всему миру. Неоценимую помощь при разработке программ может оказать библиотека знаний Microsoft Developer Network (MSDN) — это более гигабайта ценной информации. В MSDN можно найти не только справочную информацию по функциям API или классам MFC, но и массу статей, технических описаний, примеров программ, в которых разработчики делятся опытом. В арсенале профессионального программиста эта библиотека просто необходима. Кроме того, очень много полезной информации и практических советов можно найти на Интернет-сайтах, посвященных программированию (адреса некоторых из них приведены в конце книги в перечне Интернет-ресурсов).

Прежде чем продолжить повествование, кратко повторим основные принципы объектно-ориентированного стиля программирования.

2.4. Основные понятия и принципы объектно-ориентированного программирования

Осязаемая реальность,
проявляющая четко выделяемое поведение.
(Определение объекта)
Гарди Буч.
"Объектно-ориентированный анализ и проектирование"

Основные понятия и принципы объектно-ориентированного программирования (ООП): *объект*, *класс*, *инкапсуляция*, *наследование*, *полиморфизм*. Класс является обобщением понятия типа данных и задает свойства и поведение объектов класса, называемых также экземплярами класса. Каждый объект принадлежит некоторому классу. Класс можно представлять как объединение данных и процедур, предназначенных для их обработки. Данные класса называются также *переменными класса*, а процедуры — методами класса. В C++ переменная класса называется *данное-член* класса (data member), а метод — *функция-член* класса (member function). Переменные определяют свойства или состояние объекта. Методы определяют поведение объекта.

Пусть определен класс A, тогда можно определить новый класс B, наследующий свойства и поведение объектов класса A. Это означает, что в классе B будут определены переменные и методы класса A. Класс B в данном случае является производным (порожденным) от класса A. Класс A является родительским (базовым) по отношению к B. В производном классе можно задать новые свойства и поведение, определив новые переменные и методы. Можно также переопределить существующие методы базового класса. *Переопределение* (перегрузка — overloading) метода класса A — это определение в классе B метода с именем, уже являющимся именем какого-либо метода класса A. Метод базового класса можно объявить *виртуальным* (с атрибутом `virtual`). Тогда при переопределении метода в производном классе должно сохраняться число параметров метода и их типы. Виртуальность обеспечивает возможность написания полиморфных функций, аргумент которых может иметь разные типы (классы) при разных обращениях к функции и при этом будут выполняться действия, специфичные для типа фактически переданного аргумента. Например, пусть в классе A определен виртуальный метод `VirtualMethod()`, который выдает сообщение: "Я метод класса A". Переопределим этот метод в классе B так, чтобы он выдавал сообщение: "Я метод-член класса B". Теперь рассмотрим такой фрагмент псевдокода (листинг 2.3).

Листинг 2.3. Пример переопределения метода класса

```
//Определяем класс A
class A
{
  public:
   // Виртуальный метод класса A
   virtual void VirtualMethod() {вывод сообщения: "Я метод класса A"};
}

//Определяем класс B, производный от класса A
class B: public A
{
   public:
   // Виртуальный метод класса B
    void VirtualMethod() {вывод сообщения: "Я метод класса B"};
}

// Полиморфная функция. Обратите внимание: в качестве аргумента
// она принимает указатель на класс A, базовый по отношению к
// классу B.
void ShowMessage(A* x)
{
   // Вызов метода VirtualMethod()объекта x
   x->VirtualMethod();
}

// Создаем объект ObjectA класса A
A ObjectA;
// Создаем объект ObjectB класса B
B ObjectB;

// Вызываем полиморфную функцию с аргументом - указателем
// на объект класса A
// Будет выдано сообщение "Я метод класса A"
ShowMessage(&ObjectA);

// Вызываем полиморфную функцию с аргументом - указателем
// на объект класса B
// Будет выдано сообщение "Я метод класса B"
ShowMessage(&ObjectB);
```

Класс B, производный от класса A, может быть базовым для класса C и т. д. Все порожденные классы называются наследниками, а классы, от которых они порождены, — предками.

Надо отметить, что определения классов (называемые также интерфейсами классов) помещаются обычно в заголовочные файлы с расширением h, а реализация функций классов помещается в одноименные файлы с расширением cpp. Такая организация структурирует исходный текст, облегчает управление файлами с исходными текстами и позволяет выполнять раздельную компиляцию модулей программы.

2.5. Библиотека Microsoft Foundation Class Library

Библиотека MFC — это базовый набор классов, написанных на языке C++ и предназначенных для упрощения и ускорения процесса программирования под Windows. Библиотека представляет собой многоуровневую иерархию классов (около 200 членов), обеспечивающих возможность создания Windows-приложений на основе объектно-ориентированного подхода. Достоинством MFC является то, что использование ее классов позволяет избежать большей части рутинной работы и примерно в три раза сокращает объем текста программ. Инкапсулируя функции API, классы MFC значительно облегчают работу с ними. Интерфейс, обеспечиваемый библиотекой, практически не зависит от деталей, его реализующих. Поэтому программы, написанные на основе MFC, могут быть легко адаптированы к новым версиям Windows.

MFC — каркас, на основе которого можно создавать программы под Windows. Как уже отмечалось, у всех Windows-приложений фиксированная структура, определяемая функцией `WinMain()`. Структура приложений, построенных на основе объектов классов библиотеки MFC, является также жестко заданной.

Имена классов библиотеки MFC начинаются с префикса "C", а имена переменных класса с "m_"; мы тоже далее будем придерживаться этого правила.

В простейшем случае программа, написанная с использованием MFC, содержит два класса, порождаемые от классов библиотеки: класс, предназначенный для создания приложения, и класс, предназначенный для создания окна. Первый порождается от класса `CWinApp` библиотеки MFC, второй — от `CFrameWnd`. Назовем их для определенности `CApp` и `CMainWin` соответственно. Эти два класса обязательны для любой программы. Ядро MFC-приложения — глобальный объект `theApp` класса `CApp` — отвечает за создание всех остальных объектов и обработку очереди сообщений.

Для создания MFC-программы необходимо выполнить следующие действия:

1. От `CFrameWnd` породить класс, определяющий окно, — класс `CMainWin`.

2. От CWinApp породить класс, определяющий приложение — CApp.
3. Создать карту сообщений.
4. Переопределить функцию InitInstance() класса CWinApp.
5. Создать экземпляр класса, определяющего приложение, — объект theApp.

Далее приведена минимальная программа, написанная на основе классов библиотеки MFC (листинг 2.4).

Листинг 2.4. Минимальная программа на основе классов библиотеки MFC

```
#include <afxwin.h>
// Определение основных классов
// Класс, предназначенный для создания главного окна
class CMainWin: public CFrameWnd
{
public:
   CMainWin();
   DECLARE_MESSAGE_MAP()
};

// Конструктор окна
CMainWin::CMainWin()
{
   Create(NULL, "Основа MFC-приложения");
}

// Класс, предназначенный для создания приложения
class CApp: public CWinApp
{
public:
   BOOL InitInstance();
};

// Функция инициализации приложения
BOOL CApp:: InitInstance()
{
   // Создание объекта - главного окна
   m_pMainWnd=new CMainWin;
   // Отобразить окно
```

```
    m_pMainWnd->ShowWindow(m_nCmdShow);
    // Перерисовать окно
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

// Карта сообщений приложения
BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
END_MESSAGE_MAP()

//////////////////////////////////////////////////////
// Начало выполнения приложения
// Создание глобального объекта – приложения
CApp theApp;
//////////////////////////////////////////////////////
```

Результат выполнения программы показан на рис. 2.3. Прежде всего в текст MFC-программы включается файл AFXWIN.H, который содержит описание классов библиотеки MFC и подключает большинство других библиотечных файлов, а также файл WINDOWS.H. Данное приложение начинает выполняться, когда создается экземпляр класса `CApp` — объект `theApp`. Порождаемый в примере класс `CMainWin` содержит два члена: конструктор `CMainWin()` и макрос `DECLARE_MESSAGE_MAP()`, в котором объявляется карта обработки сообщений для класса `CMainWin`. Пара команд `BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)` и `END_MESSAGE_MAP()` обрамляет карту сообщений главного окна; между ними должны помещаться вызовы функций обработки поступающих сообщений. В приведенном примере (листинг 2.4) никакие сообщения не обрабатываются.

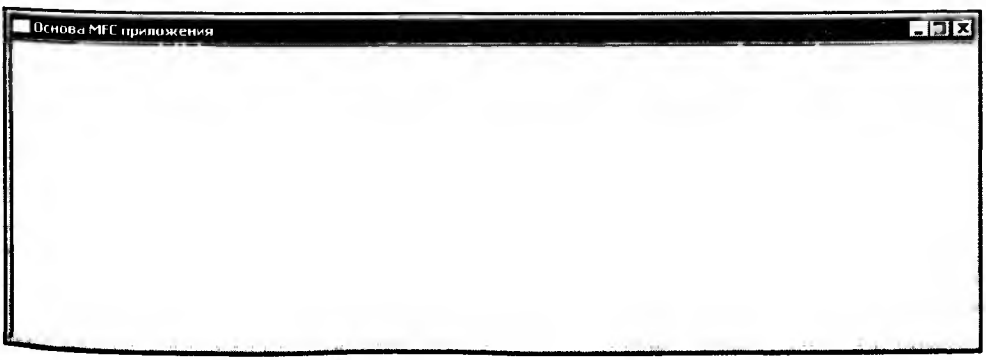


Рис. 2.3. Окно, создаваемое минимальной MFC-программой

2.6. Обработка сообщений

Сообщение — это некоторое уникальное 32-битное целое значение. В файле WINDOWS.H для всех сообщений определены стандартные имена, например: WM_CHAR, WM_PAINT, WM_MOVE, WM_CLOSE, WM_LBUTTONUP, WM_LBUTTONDOWN, WM_SIZE. Часто сообщения сопровождаются параметрами, несущими дополнительную информацию (координаты курсора, код нажатой клавиши и др.). Каждый раз, когда происходят события, касающиеся программы, Windows посылает ей соответствующее сообщение. Программисту вовсе не обязательно описывать в приложении реакции на все сообщения. Сообщения, для которых нет специального обработчика, в MFC-программе обрабатываются стандартным образом.

MFC содержит предопределенные функции-обработчики сообщений, которые можно использовать в программе. Для организации обработки сообщения необходимо выполнить следующие действия:

1. Включить макрокоманду сообщения в карту сообщений программы.
2. Включить в описание класса прототип функции-обработчика сообщения.
3. Включить в программу реализацию функции-обработчика.

Макрокоманды обеспечивают вызов функций-обработчиков стандартных сообщений Windows. Названия макрокоманд состоят из префикса ON_, названия сообщения и пары круглых скобок в конце, например ON_WM_LBUTTONDOWN(), ON_WM_LBUTTONUP, ON_WM_PAINT. Предусмотрена также специальная макрокоманда ON_COMMAND(), предназначенная для сообщений, поступающих при вызове пунктов меню, и сообщений, определенных программистом. Эта макрокоманда имеет два аргумента: ON_COMMAND(ID, METHODNAME), первый аргумент — идентификатор пункта меню или сообщения, определенного программистом; второй — имя функции-обработчика сообщения.

Макрокоманда сообщения помещается в карту сообщений окна. Карта сообщений — это специальный механизм MFC, который связывает идентификатор сообщения с соответствующим обработчиком. У одного приложения может быть несколько карт сообщений. Принадлежность карты к какому-либо окну определяют параметры макроса BEGIN_MESSAGE_MAP():

```
BEGIN_MESSAGE_MAP(класс окна-владельца сообщения, базовый класс)
   // макрокоманды сообщений
END_MESSAGE_MAP()
```

Имя функции-обработчика сообщения состоит из префикса On и названия сообщения (без WM_), например OnLButtonDown(). Прототип функции-обработчика сообщения должен быть помещен в описание класса окна перед

макросом DECLARE_MESSAGE_MAP(). При описании прототипа используется спецификатор типа afx_msg.

Рассмотрим практический пример — включим в предыдущую программу обработку сообщения WM_LBUTTONDOWN. После того как пользователь нажмет левую кнопку мыши в пределах рабочей области окна приложения, ему будет послано сообщение WM_LBUTTONDOWN. При получении данного сообщения программа будет выводить на экран сообщение о случившемся событии и координаты точки. Для этого выполним следующие действия:

1. Модифицируем карту обработки сообщений.

```
// Карта сообщений приложения
BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
// макрокоманда сообщения "нажата левая кнопка мыши"
   ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

2. Включаем в описание класса окна прототип функции-обработчика.

```
class CMainWin: public CFrameWnd
{
public:
   CMainWin();
   // прототип функции-обработчика сообщения
   // "нажата левая кнопка мыши"
   afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
   DECLARE_MESSAGE_MAP()
};
```

3. Включаем в программу реализацию функции-обработчика.

```
void CMainWin::OnLButtonDown(UINT nFlags, CPoint point)
{
   // CString – класс MFC, упрощающий работу со строками
   CString Str;
   Str.Format("Нажата левая кнопка мыши в точке x=%d, y=%d", point.x,
   point.y);
   // MessageBox – метод класса CWnd
   // (базового для CFrameWnd и, соответственно, CMainWin)
   // инкапсулирует одноименную API-функцию вывода сообщений
   // Первый аргумент (Str) – текст сообщения,
   // второй – заголовок окна,
   // третий – флаги, определяющие стиль окна
```

```
    MessageBox(Str, "Пришло сообщение", MB_OK| MB_ICONINFORMATION);
    // Стандартный обработчик
    CFrameWnd::OnLButtonDown(nFlags, point);
}
```

Работа программы показана на рис. 2.4.

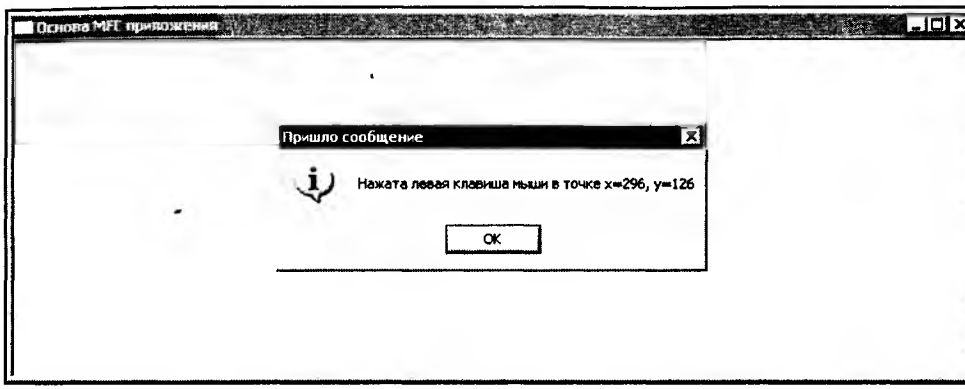


Рис. 2.4. Работа приложения с обработкой сообщения WM_LBUTTONDOWN

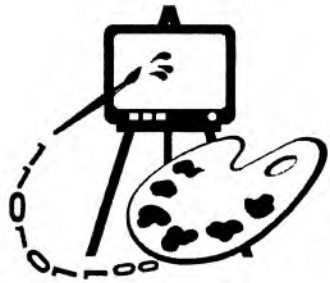
2.7. Заключение

В данной главе кратко рассмотрены принципы работы приложения в операционной системе Windows. В следующей главе будет рассказано о том, как MS Visual C++ позволяет автоматизировать процесс создания приложений.

Текст "минимальной MFC-программы" приведен на прилагаемом компакт-диске в каталоге \Sources\MFCApp.

Для более подробного изучения возможностей ОС Windows можно рекомендовать книгу [9]. Использование MFC хорошо описано в [7, 15]. Процесс создания приложений в MS Visual C++ подробно рассмотрен в [4].

Глава 3



Создаем первое "графическое" приложение

В данной главе рассматриваются:

- создание приложения с помощью генератора AppWizard;
- программа Painter 1.

3.1. Генератор приложений AppWizard. Создание приложения "Painter"

Наконец-то мы приступаем к программированию компьютерной графики. Начнем с создания простого приложения, которое, однако, будет содержать полноценный Windows-интерфейс пользователя. Данное приложение будет создавать Windows-окно со стандартными элементами управления. Особенностью нашего приложения станет то, что в нем можно будет рисовать. Пользователю достаточно будет щелкнуть левой клавишей мыши в рабочей области окна, и на экране волшебным образом появится линия. Линия соединит точку, в которой произошел щелчок, с точкой предыдущего щелчка. Указывая таким образом последовательность точек, можно будет сотворить рисунок в стиле "не отрывая руки". К концу этой главы мы создадим простейший графический редактор для работы с векторными изображениями. В ряде следующих глав программа будет совершенствоваться и послужит основой для изучения многих алгоритмов компьютерной графики.

Нашу работу по созданию программы существенно облегчит генератор приложений AppWizard — инструментальное средство Visual C++, позволяющее значительно упростить процесс создания Windows-приложений на основе MFC. При работе с этим инструментом появляется последовательность диалоговых окон, в которых AppWizard задает программисту вопросы. В процессе диалога пользователь определяет тип и характеристики проекта, который он хочет разработать. Определив, какие классы из библиотеки MFC необходимы для этого проекта, AppWizard создает проект приложения

и строит остовы всех нужных производных классов. Дальнейшая работа программиста заключается в определении свойств и поведения объектов этих классов. Проект приложения объединяет в себе описания классов, описания ресурсов и т. д.

Итак, создание приложения начинается с выбора команды **New** меню **File**. После этого на экране появляется диалоговое окно **New** (рис. 3.1) создания нового документа.

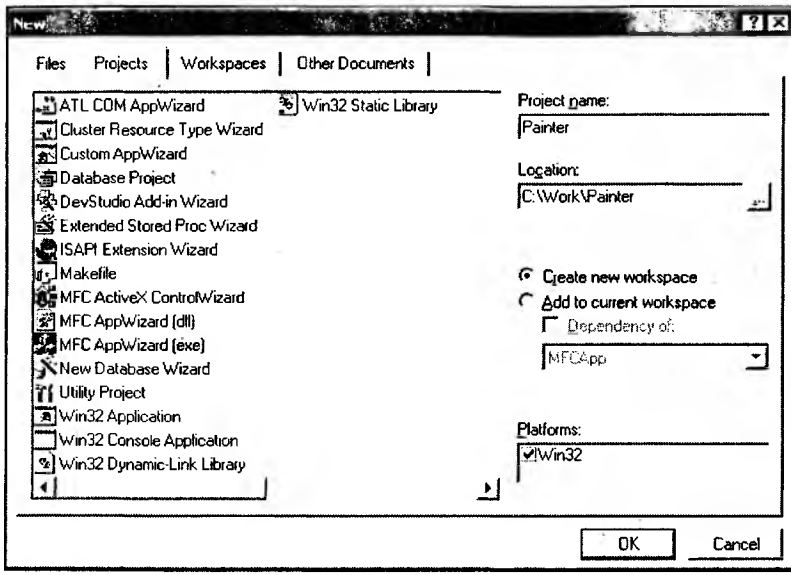


Рис. 3.1. Создание нового проекта

Так как Visual C++ позволяет создавать большое количество разнообразных файлов и приложений, то прежде всего нам потребуется уточнить тип создаваемого проекта. Для этого сначала выберем вкладку **Projects**, а затем выделим в списке типов проектов строку **MFC AppWizard (exe)**. Это даст понять Visual C++, что мы хотим воспользоваться услугами AppWizard для создания выполняемой программы. В полях ввода, расположенных в правой верхней части окна, укажем имя проекта и место на диске, где он должен быть расположен. Назовем проект "Painter" (что в переводе с английского означает "художник"). В каталоге Work на диске C будет создан подкаталог Painter, где в дальнейшем разместится одноименный проект. Нажмем **OK**.

Далее в работу включается AppWizard. На первом шаге он предлагает конкретизировать тип приложения (рис. 3.2). Укажем, что мы хотим получить приложение с архитектурой Document-View. Достоинства этой архитектуры описаны в следующей главе. При определении типа проекта выберем "однодокументное приложение" (**Single document**).

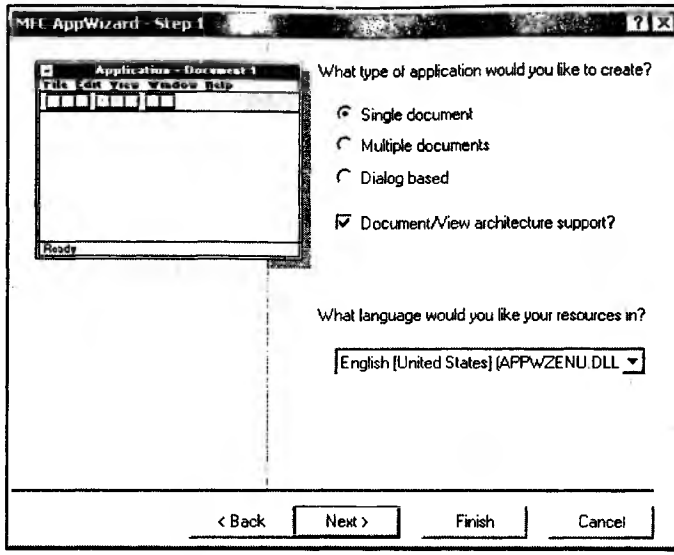


Рис. 3.2. Шаг 1. Выбор типа приложения

На следующем шаге (рис. 3.3) нам будет предложена поддержка баз данных. В данном проекте эта функция нам не потребуется. "Не поддерживается" (**None**) — эта установка является значением по умолчанию, ничего не меняем и идем дальше (кнопка **Next**).

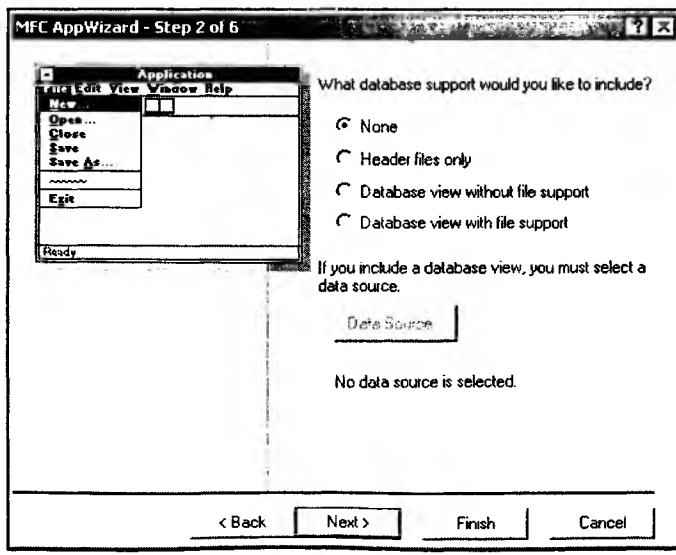


Рис. 3.3. Шаг 2. Предлагается поддержка баз данных

На третьем шаге (рис. 3.4) нам предложат поддержку средств для создания и управления составными документами — пока обойдемся без нее — кнопка **Next**.

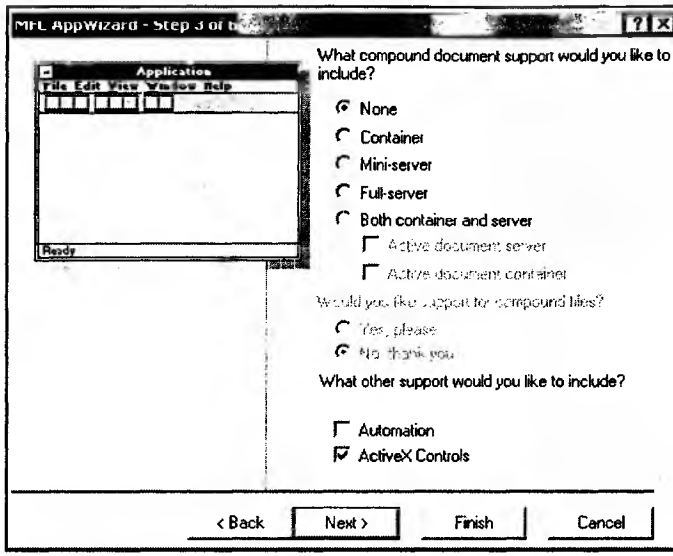


Рис. 3.4. Шаг 3. Поддержка составных документов

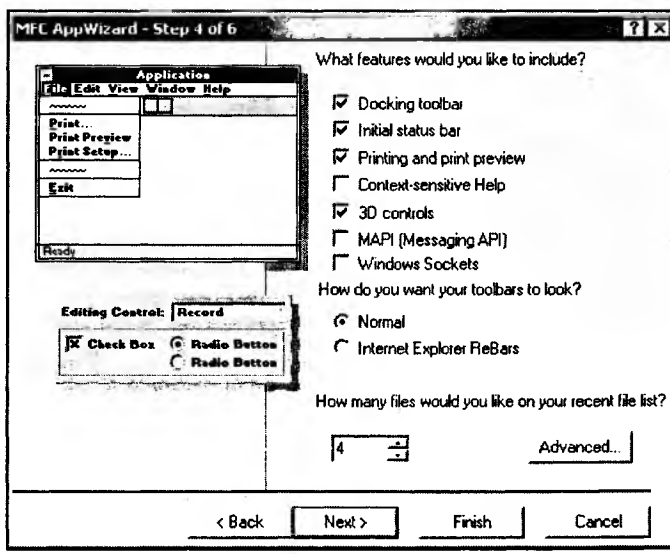


Рис. 3.5. Шаг 4. Возможности программы

На следующем шаге (рис. 3.5) AppWizard спрашивает: "Какими возможностями вы хотели бы наделить свою программу?" По умолчанию отмечены:

"пристыковываемые панели инструментов" (**Docking toolbar**); "строка состояния" (**Initial status bar**); "печать и предварительный просмотр" (**Printing and print preview**); "объемный вид элементов управления" (**3D controls**) и "нормальный" вид панелей инструментов (**Normal**). Такой набор вполне нам подходит, поэтому ничего не трогаем, жмем кнопку **Next**.

На пятом шаге (рис. 3.6) нам предложат выбрать стиль нашего приложения, комментарии в исходных текстах программы и варианты использования MFC-библиотек. По умолчанию установлено: "Стандартное MFC-приложение" (**MFC standard**), "Включать комментарии" (**Generate source files comments**), "Использовать MFC-библиотеки, как динамически подключаемые" (**As a shared DLL**). Использование динамически подключаемых MFC-библиотек выгодно тем, что размер приложения получается на несколько сотен килобайтов меньше за счет совместного использования несколькими программами одной и той же DLL-библиотеки. Статическое же связывание библиотек обеспечивает выполнение программы даже в случае, если на компьютере отсутствуют MFC DLL. Выбор пункта "стандартное MFC-приложение" означает, что мы получим заготовку приложения со стандартным окном. Альтернативой здесь является выбор пункта "В стиле проводника Windows" (**Windows Explorer**). В этом случае мы получили бы приложение, окно которого разделено на две части: слева дерево, а справа список. Генерация комментариев означает, что исходный сгенерированный текст приложения будет сопровождаться подсказками, что следует сделать при его модификации. Например, в функцию `CPainterView::OnDraw()` будет добавлена строка:

```
// TODO: add draw code for native data here,
```

что в переводе означает "добавьте здесь свой код рисования".

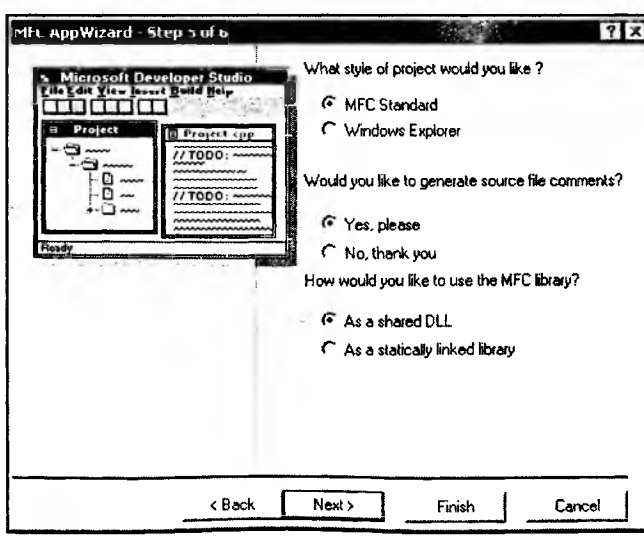


Рис. 3.6. Шаг 5. Выбор стиля приложения

На заключительном шаге AppWizard сообщает, какие классы библиотеки MFC будут использованы для создания нашего приложения (рис. 3.7). Здесь можно поменять базовый класс для класса-облика нашего приложения, имена классов и файлов. Сейчас нас все устраивает — нажимаем кнопку **Finish**.

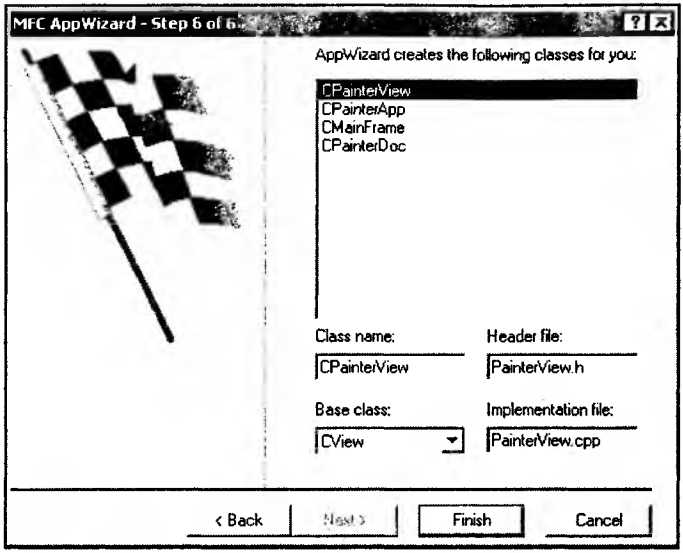


Рис. 3.7. Шаг 6. AppWizard сообщает, какие классы библиотеки MFC будут использованы

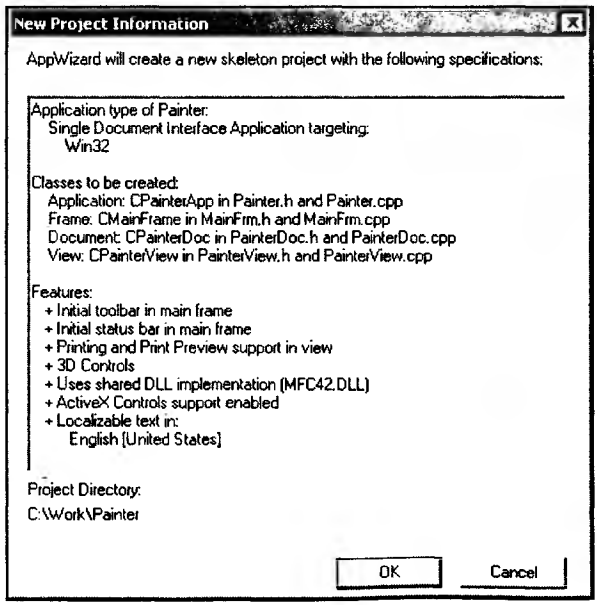


Рис. 3.8. Информация о создаваемом приложении

После этого AppWizard выводит сводную информацию (рис. 3.8) о создаваемом приложении: его типе, классах, файлах и возможностях. Для подтверждения создания такого приложения нажимаем **OK**.

Наконец-то будет создан проект нашего приложения и новое одноименное *рабочее пространство* (workspace). На экране появится окно рабочего пространства (рис. 3.9). "Рабочее пространство" — это специальный механизм управления файлами проекта, призванный облегчить программисту работу с большим количеством исходных текстов и ресурсов. "Рабочее пространство" можно сохранять на диске в виде файла с расширением dsw. При открытии этого файла (команда **File | Open Workspace**) будет восстановлено то состояние Visual C++, в котором был завершен прошлый сеанс работы с ним. В рабочее пространство могут быть включены один и более проектов, файлами которых можно управлять через окно рабочего пространства (будем называть его для краткости **Workspace**).

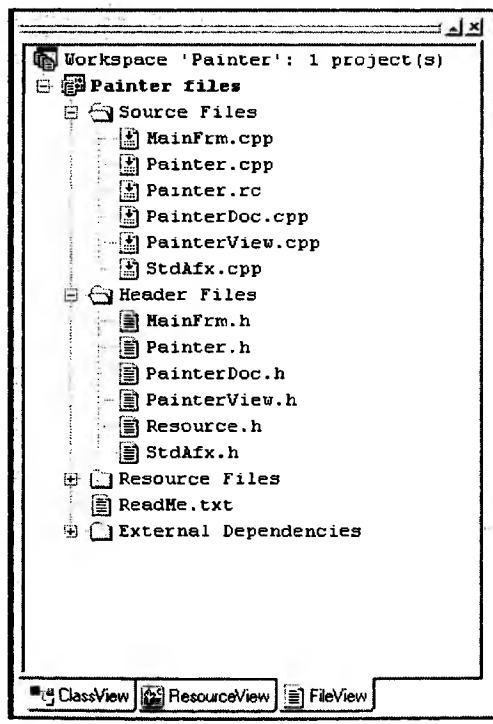


Рис. 3.9. Окно **Workspace**

Окно **Workspace** имеет три вкладки (рис. 3.9), каждая из которых открывает окно со списком:

- **FileView** — для просмотра и управления файлами проекта с исходным текстом приложения;

- **ResourceView** — для управления ресурсами приложения;
- **ClassView** — для работы с классами приложения.

В наше рабочее пространство включен один проект — созданный нами Painter. При активизации вкладки **FileView** в окне **Workspace** (см. рис. 3.9) будет показан список файлов с текстом программы, включенных в проект. Как и в Проводнике Windows, двойной щелчок мышью на изображении папки раскрывает ее содержимое, а на имени файла — открывает файл.

Через вкладку **ResourceView** можно получить доступ к ресурсам проекта. Папки в этом списке соответствуют ресурсам разного вида (рис. 3.10). Двойной щелчок на строке IDD_ABOUTBOX из папки **Dialog** откроет для редактирования шаблон диалогового окна **About**. С помощью появившейся панели инструментов **Controls** (элементы управления) можно модифицировать содержимое шаблона. Не будем откладывать это важное дело "на потом" и сразу увековечим свой копирайт. Для этого нам потребуется всего лишь щелкнуть правой клавишей мыши на тексте **Copyright (C) 2001** и выбрать из появившегося контекстного меню команду **Properties** (Свойства).

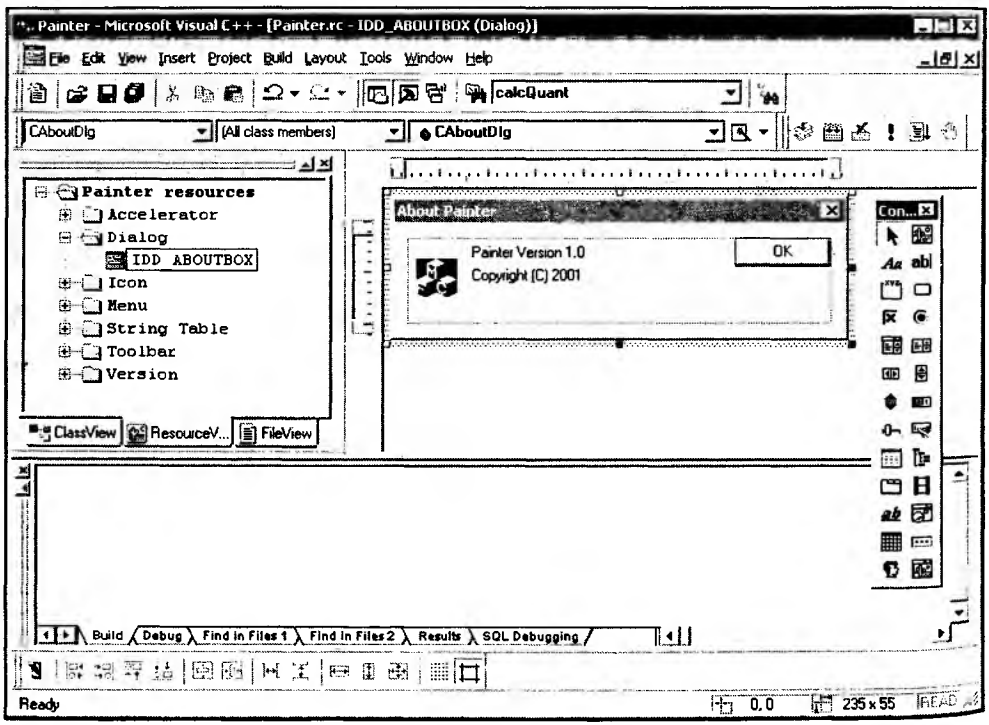


Рис. 3.10. Ресурсы проекта

Команда **Properties** (Свойства) вызовет диалоговое окно **Text Properties** (Свойства текста) (рис. 3.11). В правой части окна расположено поле **Caption**

(Заголовок), в котором можно ввести нужный текст. Я введу свою фамилию, а вы пишите что захотите. В окне **Text Properties** есть еще несколько вкладок, выбирая которые можно определять стиль отображения текста. Поэкспериментируйте с этими опциями самостоятельно. Кстати, для того чтобы русский текст правильно отображался в диалоговых окнах, щелкните правой кнопкой мыши на идентификаторе ресурса в окне **Workspace**, из контекстного меню выберите **Properties** и установите для свойства **Language** значение **Russian**.

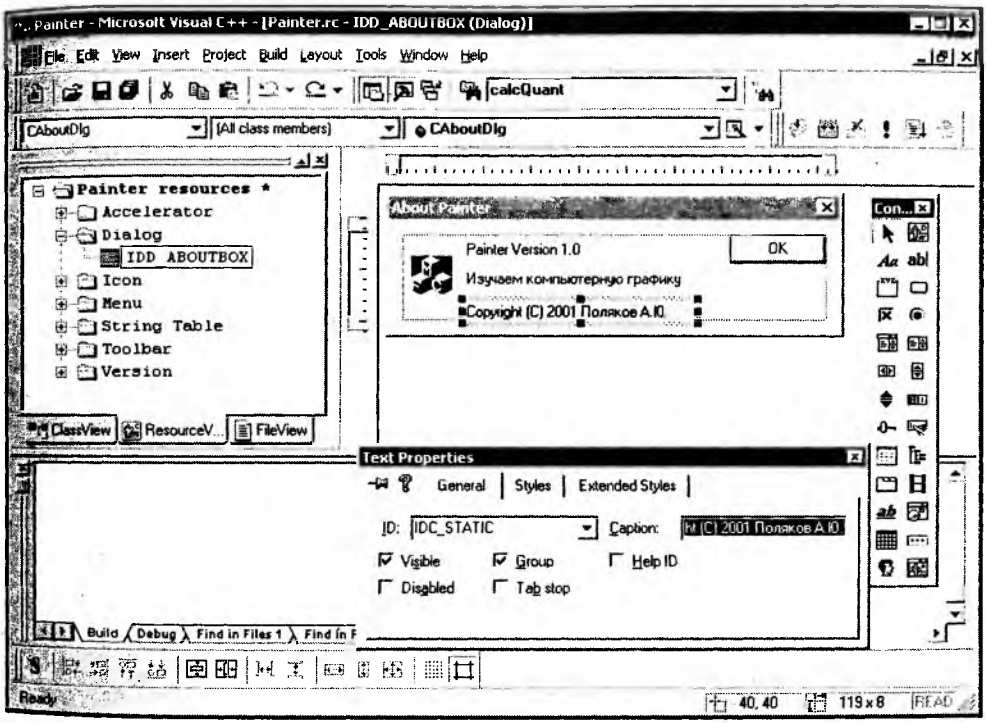


Рис. 3.11. Редактирование текста диалога **About**

Новый элемент управления "статический текст" можно создать, если сначала щелкнуть мышью на панели инструментов **Controls** кнопку **Static Text** (вторая сверху в первом столбце), а затем — на шаблоне диалогового окна. Размер и положение текста можно задать с помощью активных зон (темных квадратиков), расположенных по периметру текста.

В следующих главах, по мере необходимости, мы будем рассматривать работу и с другими ресурсами приложения.

Аналогично вкладка **ClassView** открывает список, в котором перечислены используемые в приложении классы. Через этот список можно быстро открывать текст реализации методов любого класса, добавлять в классы методы и переменные. Двойной щелчок на названии класса открывает описание

класса в заголовочном H-файле, двойной щелчок на названии метода — его реализацию в CPP-файле. Правой клавишей мыши можно вызвать контекстное меню (рис. 3.12), через которое можно добавить в класс методы и переменные, а также выполнить целый ряд других операций.

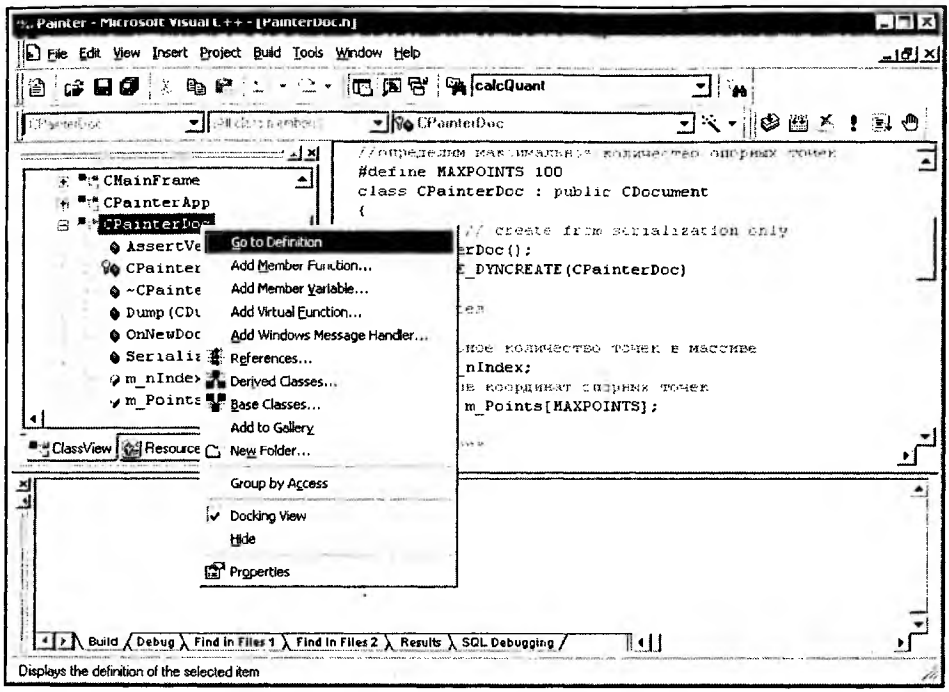


Рис. 3.12. Классы проекта

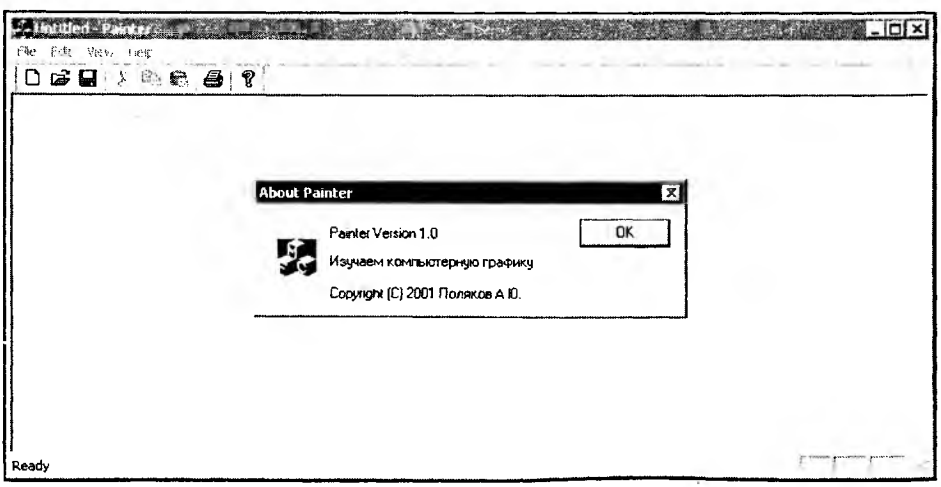


Рис. 3.13. Первый запуск программы Painter

Теперь проект можно откомпилировать (команда **Build | Build Painter.exe**) и запустить на выполнение (**Build | Execute Painter.exe**). Программа уже имеет все атрибуты Windows-приложения: меню, панель инструментов, строку состояния, стандартные диалоги открытия/сохранения файлов и даже режим предварительного просмотра печати. Более того, при выполнении команды **Help | About** мы увидим диалоговое окно с нашим текстом (рис. 3.13). Однако рисовать наша программа пока еще не умеет, но научить ее этому будет не так уж и сложно.

3.2. Добавление функций рисования

Придадим теперь приложению новые графические свойства. Мы собираемся создать программу, которая позволила бы нам рисовать на экране рисунки прямыми линиями, сохранять их в файл, загружать и выводить на печать.

Прежде всего, надо решить, что мы хотим от программы, и как с ней будет работать пользователь. Поскольку это наш первый проект, для начала потребуется что-нибудь простое и ясное. Пусть пользователь имеет возможность указывать на экране точки (назовем их "опорными"), которые должны будут последовательно соединяться прямыми линиями. Для задания точек на экране удобно воспользоваться мышью. Пусть щелчок левой клавишей мыши означает ввод опорной точки. Для обработки сообщения о щелчке включим в программу специальную функцию, которая будет получать значения координат точки.

Далее надо определиться, каким образом будут храниться данные — координаты опорных точек. Для хранения координат точки в Windows предназначена структура `POINT`, определенная в заголовочном файле Windef.h:

```
typedef struct tagPOINT
{
   LONG  x;
   LONG  y;
} POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;
```

Как видно, эта структура объединяет в себе координаты `x` и `y` точки. В библиотеке MFC структура `POINT` абстрагируется классом `CPoint`, который предоставляет в распоряжение программиста методы и переопределенные операторы для удобной работы с координатами точек. Класс `CPoint` определен в файле Afxwin.h следующим образом (листинг 3.1).

Листинг 3.1. Определение класса `CPoint`

```
//////////////////////////////////////////////////////////////
// CPoint - 2D-точка, подобно Windows-структуре POINT.

class CPoint : public tagPOINT
```

```
{
public:
// Конструкторы

   // Создает неинициализированный объект
   CPoint();
   // Значения объекта инициализируются аргументами
   CPoint(int initX, int initY);
   // Конструктор
   CPoint(POINT initPt);
   // Создает на основе структуры SIZE
   CPoint(SIZE initSize);
   // Создает на основе значения DWORD:
   // x = LOWORD(dw) y = HIWORD(dw)
   CPoint(DWORD dwPoint);

// Операции

// Смещает точку на заданное расстояние
   void Offset(int xOffset, int yOffset);
   void Offset(POINT point);
   void Offset(SIZE size);
   void operator+=(SIZE size);
   void operator-=(SIZE size);
   void operator+=(POINT point);
   void operator-=(POINT point);
// Логические операторы
   BOOL operator==(POINT point) const;
   BOOL operator!=(POINT point) const;

// Операторы, возвращающие значение CPoint
   CPoint operator+(SIZE size) const;
   CPoint operator-(SIZE size) const;
   CPoint operator-() const;
   CPoint operator+(POINT point) const;

// Операторы, возвращающие значение CSize
   CSize operator-(POINT point) const;

// Операторы, возвращающие значение CRect
```

```
    CRect operator+(const RECT* lpRect) const;
    CRect operator-(const RECT* lpRect) const;
};
```

В файле Afxwin.h определено также большое количество других полезных классов, которые мы будем далее использовать.

Используем класс `CPoint` для хранения координат опорной точки. Так как рисунок может состоять из большого количества опорных точек, надо предусмотреть какой-то вариант их хранения. Простейшим вариантом будет статический массив объектов `CPoint` достаточно большого размера.

Пришла пора решить, где будет храниться весь массив точек. При создании шаблона нашего приложения мы выбрали архитектуру Document-View. В соответствии с этой архитектурой в нашем приложении были созданы два класса:

- `CPainterDoc`, производный от класса `CDocument` библиотеки MFC;
- `CPainterView`, производный от класса `CView` библиотеки MFC.

Подробно архитектуру Document-View мы рассмотрим в следующей главе, а пока отметим, что данные обычно хранятся в классе документа.

Поэтому будем хранить массив точек в классе `CPainterDoc`.

Для этого откроем файл PainterDoc.h и включим в описание класса `CPainterDoc` следующие строки, выделенные полужирным шрифтом (листинг 3.2). Напомним, что файл PainterDoc.h можно открыть, дважды щелкнув левой кнопкой мыши на его имени в окне вкладки **FileView**.

Листинг 3.2. Модификация класса `CPainterDoc`. Файл PainterDoc.h

```
// PainterDoc.h : interface of the CPainterDoc class
//
/////////////////////////////////////////////////////////////////////////
// Определим максимальное количество опорных точек
#define MAXPOINTS 100
class CPainterDoc : public CDocument
{
protected: // create from serialization only
    CPainterDoc();
    DECLARE_DYNCREATE(CPainterDoc)
// Attributes
public:
    // Реальное количество точек в массиве
    WORD m_nIndex;
```

```
    // Массив координат опорных точек
    CPoint m_Points[MAXPOINTS];
...
```

Остальной код файла оставим без изменения.

В файле PainterDoc.cpp дополним конструктор класса `CPainterDoc` инициализацией переменной `m_nIndex` (листинг 3.3).

Листинг 3.3. Модификация конструктора класса `CPainterDoc`. Файл PainterDoc.cpp

```
////////////////////////////////////////////////////////////////
// CPainterDoc construction/destruction

CPainterDoc::CPainterDoc()
{
    // Сначала в массиве нет точек
    m_nIndex=0;
}
```

3.3. Использование генератора классов ClassWizard

Далее используем средство ClassWizard, предоставляемое Visual C++ для автоматизации создания классов и обработчиков сообщений. Выполним команду **View | ClassWizard** (быстрый вызов — комбинация клавиш <Ctrl>+<W>). На экране должно появиться диалоговое окно (рис. 3.14) с открытой вкладкой **Message Maps** (Карты сообщений). Выберем из списка **Class Name** (Имя класса) класс `CPainterView`, а из списка сообщений (**Messages**) — сообщение `WM_LBUTTONDOWN` и нажмем кнопку **AddFunction**, а затем кнопку **OK**.

После этого ClassWizard вставит в карту сообщений класса `CPainterView` макрокоманду `ON_WM_LBUTTONDOWN()` (листинг 3.4).

Листинг 3.4. Карта сообщений класса `CPainterView`. Файл PainterView.cpp

```
BEGIN_MESSAGE_MAP(CPainterView, CView)
    //{{AFX_MSG_MAP(CPainterView)
    ON_WM_LBUTTONDOWN()
    //}}AFX_MSG_MAP
    // Standard printing commands
```

```
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

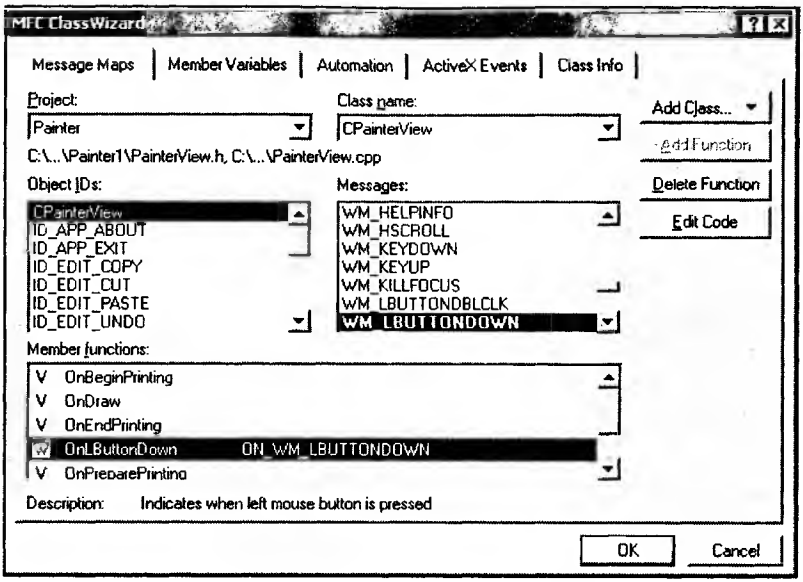


Рис. 3.14. Диалоговое окно **MFC ClassWizard**

В описание класса `CPainterView` (файл PainterView.h) будет вставлено имя функции-обработчика данного сообщения — `OnLButtonDown()` (листинг 3.5).

Листинг 3.5. Определение функции `OnLButtonDown()` в классе `CPainterView`

```
// Generated message map functions
protected:
    //{{AFX_MSG(CPainterView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Нам остается только отредактировать реализацию данной функции в файле PainterView.cpp. Для этого в списке классов окна вкладки **ClassView** в рабочей области нашего проекта дважды щелкнем на имени функции-члена `OnLButtonDown()` класса `CPainterView`. Visual C++ откроет нам тело данной функции в файле PainterView.cpp.

Добавим в него строки, которые выполняют сохранение координат указателя мыши в момент нажатия левой кнопки мыши. В листинге 3.6 приведен код функции `OnLButtonDown()` (полужирным шрифтом выделены добавленные строки).

Листинг 3.6. Модификация функции `OnLButtonDown()`. Файл PainterView.cpp

```
//////////////////////////////////////////////////////////////////////
// CPainterView message handlers

void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
   // Проверим, не исчерпали ли ресурс
   if(pDoc->m_nIndex==MAXPOINTS)
      AfxMessageBox("Слишком много точек");
   else
   {
      // Запоминаем точку
      pDoc->m_Points[pDoc->m_nIndex++]=point;
      // Указываем, что окно надо перерисовать
      Invalidate();
      // Указываем, что документ изменен
     pDoc->SetModifiedFlag();
   }
   // Даем возможность стандартному обработчику
   // тоже поработать над этим сообщением
   CView::OnLButtonDown(nFlags, point);
}
```

Добавим в метод `OnDraw()` класса `CPainterView` строки, которые будут выполнять вывод на экран линий, соединяющих опорные точки. В листинге 3.7 приведен код этой функции (полужирным шрифтом выделены добавленные строки).

Листинг 3.7. Модификация функции `OnDraw()`. Файл PainterView.cpp

```
//////////////////////////////////////////////////////////////////////
// CPainterView drawing

void CPainterView::OnDraw(CDC* pDC)
```

```
{
   CPainterDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);
   // TODO: add draw code for native data here
   // Если имеются опорные точки
   if(pDoc->m_nIndex>0)
      // Поместим перо в первую из них
      pDC->MoveTo(pDoc->m_Points[0]);
   // Пока опорные точки не закончатся, будем их соединять
   for(int i=1; i<pDoc->m_nIndex; i++)
      pDC->LineTo(pDoc->m_Points[i]);
}
```

После этого откомпилируем программу и запустим ее на выполнение. Теперь можно заняться изобразительным искусством.

3.4. Сохранение рисунков в файл

Для того чтобы наши бессмертные творения стали воистину таковыми, модифицируем функцию `Serialize()` класса `CPainterDoc` в файле PainterDoc.cpp (листинг 3.8). Данная функция является членом класса `CDocument` и унаследована классом `CPainterDoc`. Она отвечает за сохранение и загрузку данных. Об особенностях ее работы мы поговорим позднее, а сейчас просто добавим в нее выделенные полужирным шрифтом строки. (Обратите внимание, что AppWizard добавил в тело функции комментарии "`// TODO:`", которые подсказывают нам, код выполнения каких операций следует добавить в этом месте.)

Листинг 3.8. Модификация метода `Serialize()` класса `CPainterDoc`. Файл PainterDoc.cpp

```
/////////////////////////////////////////////////////////////////////////
// CPainterDoc serialization

void CPainterDoc::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
   {
      // TODO: add storing code here
      // Сохраняем количество точек
      ar << m_nIndex;
```

```
        // Сохраняем значения координат точек
        for(int i=0; i<m_nIndex; i++)
            ar << m_Points[i];
    }
    else
    {
        // TODO: add loading code here
        // Загружаем количество точек
        ar >> m_nIndex;
        // Загружаем значения координат точек
        for(int i=0; i<m_nIndex; i++)
            ar >> m_Points[i];
    }
}
```

Теперь можно сохранять созданные рисунки в файлах на диске и считывать их вновь.

Надо отметить, что рисунки будут сохраняться в формате программы Painter. Это не какой-нибудь широко известный формат (например, BMP или CDR), поэтому популярные графические программы его не прочитают. Для того чтобы отличать файлы рисунков программы Painter, мы можем ввести свое расширение, например pr1, где цифра 1 обозначает версию формата.

3.5. Создание нового рисунка

Для того чтобы наша программа стала полноценной, необходимо еще наделить ее функцией создания нового рисунка.

Так как в однодокументных приложениях используется один объект-документ, то просто вставим код реинициализации его переменных и обновим его представление на экране (об особенностях разных типов приложений и о взаимодействии объектов-документов с их обликами на экране пойдет речь в следующих главах). Для этого добавим в метод `OnNewDocument()` класса `CPainterDoc` строки, выделенные полужирным шрифтом (листинг 3.9).

Листинг 3.9. Модификация метода `OnNewDocument()` класса `CPainterDoc`. Файл PainterDoc.cpp

```
BOOL CPainterDoc:: OnNewDocument()
{
    // Сначала вызывается метод базового класса
if (!CDocument::OnNewDocument()) return FALSE;
```

```
    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
    // Сбросили счетчик
    m_nIndex=0;
    // Перерисовали
    UpdateAllViews(NULL);
    return TRUE;
}
```

Работа созданной программы показана на рис. 3.15.

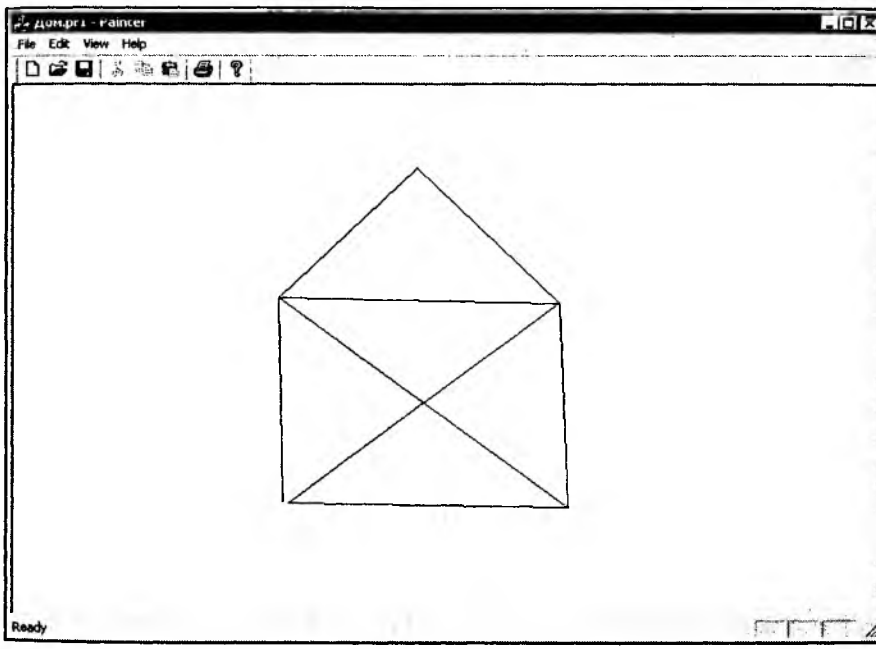


Рис. 3.15. Программа Painter в действии

3.6. Вывод рисунков на печать и предварительный просмотр

Возможно, вы будете приятно удивлены, узнав, что наша программа Painter способна не только выводить изображение на экран, но и отправлять его на принтер. Поддержка печати и предварительного просмотра реализуется классами библиотеки MFC. Вспомните, на 4-м шаге создания нашего приложения мы попросили генератор AppWizard добавить поддержку печати и предварительного просмотра. Поэтому эти функции оказались уже реализованными в нашей программе.

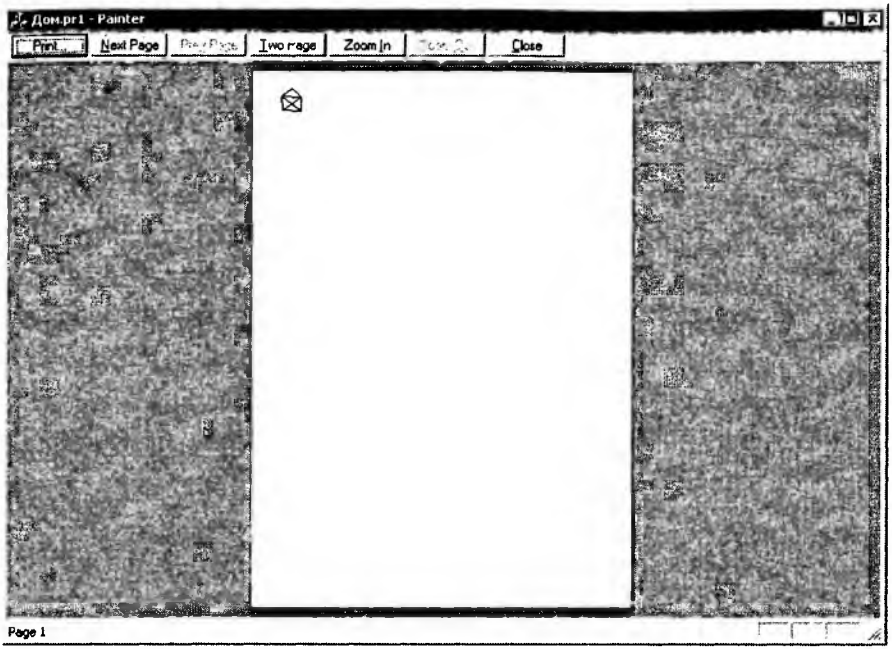


Рис. 3.16. Предварительный просмотр

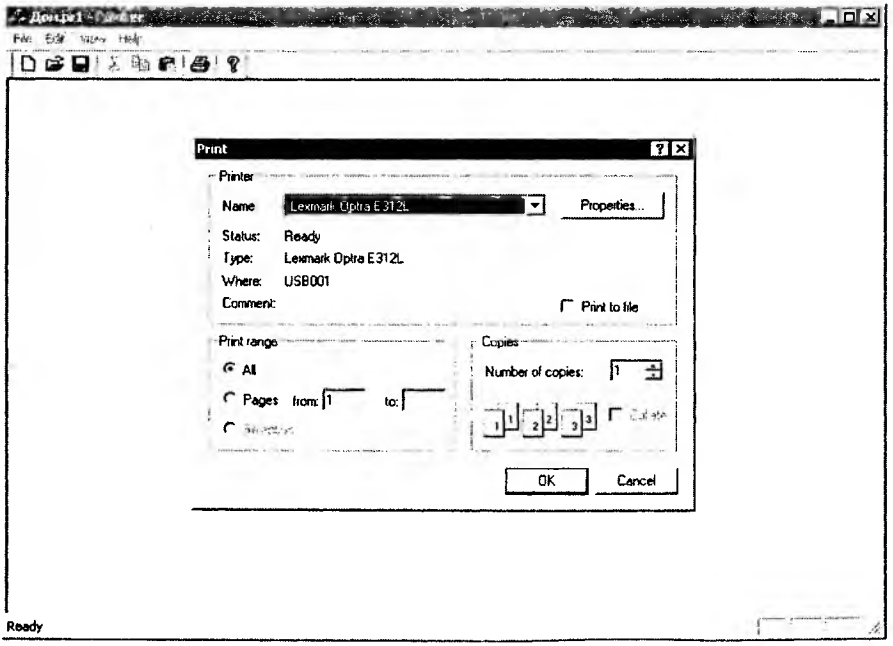


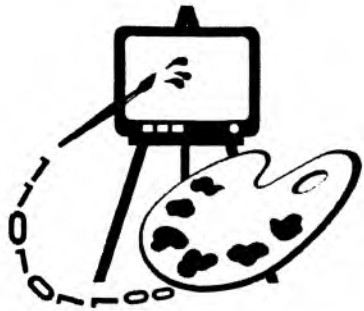
Рис. 3.17. Диалоговое окно **Print**

Выполнив команду **File | Print Preview** (Предварительный просмотр), можно увидеть, как будет выглядеть созданный в программе Painter рисунок на листе бумаги (рис. 3.16).

Нажатие кнопки **Print** (Печать) вызовет на экран стандартный диалог печати, в котором можно задать свойства принтера и дать задание распечатать рисунок (рис. 3.17). Может показаться странным, что такой большой на экране рисунок оказался таким маленьким на листе. Этот эффект связан с тем, что у экрана разрешение, как правило, гораздо меньше, чем у принтера. О способах решения этой проблемы мы поговорим в *главе 4*. А пока можно отдохнуть и порисовать.

3.7. Заключение

В этой главе мы рассмотрели средства автоматизации процесса создания приложений. Мы проделали большую работу — создали графическое приложение "под Windows", которое умеет уже довольно много: рисует, сохраняет и печатает созданные рисунки. Текст программы приведен на прилагаемом компакт-диске в каталоге \Sources\Painter1.



Часть II

РАБОТА С ВЕКТОРНОЙ ГРАФИКОЙ

Глава 4

Архитектура приложений Document-View

В данной главе рассматриваются:

- архитектура приложений Document-View;
- контекст устройства и графические методы класса CDC;
- описание геометрических тел с использованием объектно-ориентированного подхода, применение наследования, виртуальных методов и полиморфных функций;
- программа Painter 2.

4.1. Архитектура приложений Document-View

При построении проектов в Visual C++ основной (но не единственной) является архитектура приложений Document-View (документ-облик). Главная идея такой структуры программы — разделение данных и их представления на экране. Реализация архитектуры Document-View заключается в том, что помимо классов главного окна и приложения создаются еще два класса: класс *документа* и класс *облика*. Классы документа и облика являются производными от классов библиотеки MFC. Базовым для класса документа является класс `CDocument`; для класса облика — `CView`. Под документом понимается любая совокупность данных: текст, звук, изображение и т. д. Приложения, построенные с использованием архитектуры Document-View, могут быть двух типов: *однодокументные* (Single Document Interface, SDI) и *многодокументные* (Multiple Document Interface, MDI). Однодокументные SDI-приложения позволяют редактировать одновременно лишь один документ. Примерами однодокументных приложений являются программы Notepad и Paint, входящие в набор стандартных программ Windows. Многодокументные MDI-приложения, как следует из их названия, позволяют редактировать сразу несколько документов (пример — MS Word), кроме того, MDI-приложения могут одновременно поддерживать несколько разных типов

документов. Например, среда MS Visual C++ позволяет одновременно редактировать большое количество документов разного типа: тексты программ, ресурсы (меню, диалоговые окна, пиктограммы, растровые изображения).

Достоинство архитектуры Document-View в том, что она позволяет отделить данные от их визуального представления. В таких приложениях одни и те же данные, хранящиеся в объекте-документе, могут быть представлены одновременно в различной форме с помощью разных объектов-обликов. Например, некоторый график (один и тот же набор данных) мог бы в одном окне программы выглядеть как таблица значений, в другом окне — как кривая, в третьем — как диаграмма.

Центральными объектами в такой архитектуре являются один или несколько объектов-документов. Каждый документ сопровождается, по крайней мере, одним объектом-обликом (их может быть несколько для одного и того же документа, как в приведенном выше примере с графиком). Облик является объектом, предназначенным для отображения данных документа на экране и обеспечения взаимодействия с пользователем. Пользователь наблюдает данные, визуализированные объектом-обликом, и выполняет их редактирование. Объект-облик принимает управляющие действия пользователя, связывается с объектом-документом и изменяет данные, используя методы документа (рис. 4.1). Логически облики привязаны к документу.

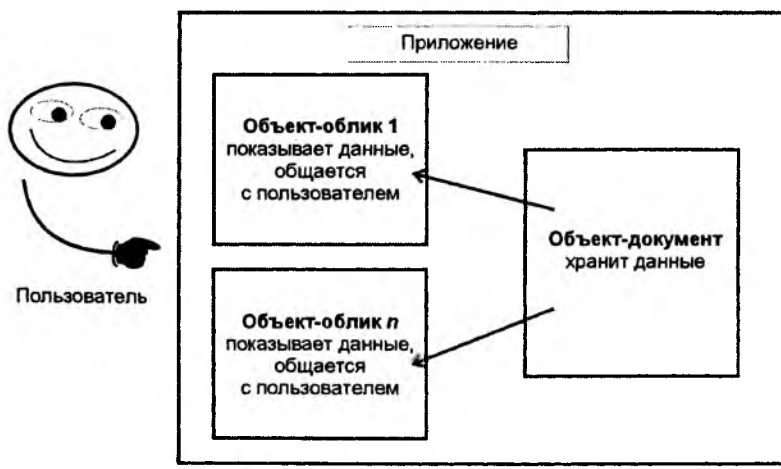


Рис. 4.1. Архитектура Document-View

Документы создаются как экземпляры классов, производных от класса `CDocument` библиотеки MFC. Класс `CDocument` имеет хорошо развитые методы загрузки, сохранения и управления данными. Облики создаются как объекты классов, производных от класса `CView` из MFC, и имеют набор методов для отображения данных.

Классы CDocument и CView имеют средства для общения между собой. Объект-документ содержит список всех объектов-обликов, представляющих его данные на экране. Метод UpdateAllViews() класса CDocument позволяет послать всем объектам-обликам уведомление о необходимости перерисовать свое содержимое. Метод имеет несколько параметров, но чаще всего его вызов выглядит следующим образом: UpdateAllViews(NULL);

Метод GetDocument() класса CView позволяет облику получить указатель на объект-документ, данные которого он представляет. Этот метод очень важен, он дает возможность облику получить доступ к данным и методам документа. Еще один важный метод класса CView носит название OnDraw() и предназначен для того, чтобы в производном классе его переопределили таким образом, чтобы на экран выводились необходимые данные. Этот метод всегда вызывается при обработке сообщения Windows о необходимости перерисовки окна. Вызывать этот метод напрямую из тела программы не надо! При необходимости перерисовать окно вызывается функция Invalidate() класса CView.

С использованием AppWizard построение приложения выполняется в два этапа. Сначала строится основа приложения, затем программист наделяет методы производных классов новыми свойствами. Вернемся к рассмотрению проекта Painter, который мы начали создавать в предыдущей главе, и разберемся, где у нас документ, где облик и как они взаимодействуют.

Первое, что мы сделали при разработке программы, — это создали с помощью генератора AppWizard основу однодокументного приложения. Были созданы класс документа CPainterDoc, производный от CDocument, и класс облика CPainterView, производный от CView.

Затем мы модифицировали наш класс документа. В определение класса CPainterDoc мы добавили переменные, в которых сохраняются данные:

```
// Реальное количество точек в массиве
WORD m_nIndex;
// Массив координат опорных точек
CPoint m_Points[MAXPOINTS];
```

В конструкторе класса CPainterDoc мы произвели инициализацию переменной m_nIndex. В однодокументных приложениях имеется только один объект-документ. При запуске приложения создается объект-документ, который используется на протяжении всего сеанса работы приложения. В момент создания объекта вызывается конструктор и выполняется инициализация переменной m_nIndex. При выполнении же команды **File | New** новый объект-документ не создается и, соответственно, конструктор второй раз не вызывается. Вместо конструктора при выборе этой команды автоматически вызывается метод OnNewDocument() класса CDocument. Мы переопределили этот метод таким образом, чтобы в нем обнулялась пере-

менная m_nIndex — это то же самое, что удалить все опорные точки из массива (листинг 4.1).

Листинг 4.1. Функция CPainterDoc:: OnNewDocument(). Файл Paintdoc.cpp

```
BOOL CPainterDoc:: OnNewDocument()
{
   // Сначала вызывается метод базового класса
   if (!CDocument::OnNewDocument()) return FALSE;
   // Сбросили счетчик
   m_nIndex=0;
   // Перерисовали
   UpdateAllViews(NULL);
   return TRUE;
}
```

Для обновления содержимого окна программы в методе OnNewDocument() мы использовали функцию UpdateAllViews(NULL) (можно ее закомментировать и посмотреть, что получится).

Основная операция — рисование линий — в нашей программе выполняется через нажатие левой кнопки мыши в клиентской области окна. Все, что при этом надо сделать, — это лишь запомнить в массиве m_Points координаты точки, в которой произошел щелчок, и увеличить счетчик m_nIndex на единицу. Чтобы линия отобразилась на экране, мы инициируем перерисовку содержимого окна программы с помощью функции Invalidate() класса облика.

Для того чтобы добраться до данных объекта-документа в функции-обработчике нажатия кнопки мыши OnLButtonDown() и в функции OnDraw(), использован метод GetDocument(), возвращающий указатель на объект-документ (листинги 4.2 и 4.3).

Листинг 4.2. Функция CPainterView::OnLButtonDown(). Файл PainterView.cpp

```
void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
   // Проверим, не исчерпали ли ресурс
   if(pDoc->m_nIndex==MAXPOINTS)
   AfxMessageBox("Слишком много точек");
   else
```

```
    {
        // Запоминаем точку
        pDoc->m_Points[pDoc->m_nIndex++]=point;
        // Указываем, что окно надо перерисовать
        Invalidate();
        // Указываем, что документ изменен
        // (это приведет к тому, что при закрытии документа
        // пользователя спросят: "Не хочет ли он сохранить изменения?")
        pDoc->SetModifiedFlag();
    }
    // Даем возможность стандартному обработчику
    // тоже поработать над этим сообщением
    CView::OnLButtonDown(nFlags, point);
}
```

Кроме классов документа и облика, AppWizard создал еще один класс — класс приложения `CPainterApp`, производный от `CWinApp`. Именно в этом классе содержится механизм создания и управления объектами-документами. Таким образом, упрощенно схему взаимодействия объектов нашего приложения можно описать следующим образом.

- Сначала создается объект-приложение (экземпляр класса `CPainterApp`), который инициирует создание объекта-документа и объекта-облика.
- Объект-приложение хранит данные, относящиеся ко всему приложению, и организует работу документов и обликов.
- Объект-документ хранит данные, для обработки которых предназначена программа, и имеет методы для их модификации.
- Объект-облик визуализирует данные, общается с пользователем и изменяет данные.

4.2. Контекст устройства, графические методы класса *CDC*

Контекстом устройства (Device Context — DC) в Windows называется логическое представление какого-либо физического устройства, например монитора, принтера или плоттера. Контекст устройства представляет собой структуру данных, которая определяет состояние драйвера устройства и способ вывода информации. Windows-приложения не взаимодействуют напрямую с физическими устройствами, а для того, чтобы нарисовать что-нибудь, обращаются к функциям Graphic Device Interface — GDI, которые в качестве

аргумента принимают контекст устройства. Такая схема работы позволяет абстрагироваться от аппаратной части компьютера и облегчить переносимость программ.

Библиотека MFC содержит ряд классов, инкапсулирующих различные типы контекстов устройств Windows. Класс CDC является базовым для остальных классов контекстов устройств. Этот класс используется для организации вывода информации на экран и на другие устройства (принтер) и имеет несколько десятков различных методов. Часть методов класса CDC предназначена для задания различных атрибутов контекста устройства (параметров, влияющих на вывод информации, например: цвет, режим отображения и др.), часть — для вывода текста, рисования линий и фигур. Класс CDC содержит методы для работы с векторной и растровой графикой.

Мы не будем здесь приводить описание всех методов класса CDC. При желании информацию о любом из классов MFC, в том числе и о классе CDC, можно получить в библиотеке MSDN. Далее же, по мере необходимости, мы будем рассматривать только те методы класса CDC, которые пригодятся для совершенствования наших программ.

Например, метод `CDC::SelectObject()` предназначен для выбора в контексте устройства объекта, с использованием которого будет выполняться рисование. В качестве аргумента этот метод может принимать указатели на объекты: *перья* (pens), *кисти* (brushes), *шрифты* (fonts), *битовые изображения* (bitmaps) и *регионы* (regions). Соответственно, метод имеет следующие формы:

- `CPen* SelectObject( CPen* pPen );`
- `CBrush* SelectObject( CBrush* pBrush );`
- `virtual CFont* SelectObject( CFont* pFont );`
- `CBitmap* SelectObject( CBitmap* pBitmap );`
- `int SelectObject( CRgn* pRgn ).`

Метод возвращает указатель на объект, который был замещен, или NULL в случае ошибки.

Метод `CDC::LineTo()` предназначен для рисования линии из текущей позиции в точку, координаты которой передаются как параметр. Метод имеет две переопределенные формы:

```
BOOL LineTo( int x, int y );
BOOL LineTo( POINT point );
```

Возвращает TRUE (1), если линия нарисовалась, и FALSE (0) в противном случае. Рисование выполняется текущим пером. Перо требуемого цвета и стиля можно установить методом `CDC::SelectObject()`.

Метод CDC::Rectangle() рисует прямоугольник, размеры которого заданы параметрами x1, y1, x2, y2:

```
BOOL Rectangle( int x1, int y1, int x2, int y2 );
```

Метод CDC::Ellipse() рисует эллипс в пределах прямоугольника, заданного параметрами x1, y1, x2, y2:

```
BOOL Ellipse( int x1, int y1, int x2, int y2 );
```

При использовании архитектуры Document-View вывод информации осуществляется объектом-обликом в функции OnDraw(). В качестве параметра в функцию OnDraw() передается указатель на объект класса CDC:

```
OnDraw(CDC* pDC)
```

Объект, адрес которого передается через указатель pDC, заранее подготовлен каркасом приложения и содержит контекст того устройства, на которое осуществляется вывод. Нам требуется лишь воспользоваться методами класса CDC для вывода наших данных. Что мы и сделали при выводе нашего рисунка в методе CPainterView::OnDraw().

Листинг 4.3. Функция CPainterView::OnDraw(). Файл PainterView.cpp

```
void CPainterView::OnDraw(CDC* pDC)
{
   CPainterDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);
   // TODO: add draw code for native data here
   // Если имеются опорные точки
   if(pDoc->m_nIndex>0)
      // Поместим перо в первую из них
      pDC->MoveTo(pDoc->m_Points[0]);
      // Пока не закончатся опорные точки, будем их соединять
      for(int i=1; i<pDoc->m_nIndex; i++)
      pDC->LineTo(pDoc->m_Points[i]);
}
```

4.3. Модификация программы Painter

Настала пора продолжить развитие нашей программы Painter. Начнем, пожалуй, с решения проблемы несоответствия между размером рисунка, видимого на экране, и отпечатанного на принтере. Затем придадим программе новые возможности рисования простых фигур.

4.3.1. Решение проблемы вывода на принтер

Как уже было отмечено в предыдущей главе, различие размеров рисунка на экране и на листе бумаги объясняется тем, что экран и принтер имеют разную разрешающую способность (разрешение). Разрешение устройства, отображающего какую-либо информацию, измеряется в количестве пикселов на единицу размера. Обычно разрешение измеряется в пикселах на дюйм (dot per inch — dpi) или в пикселах на метр.

Представим себе, что мы выводим на экран прямую линию длиной 100 пикселов, а разрешение экрана 100 dpi (точек на дюйм). Тогда видимая на экране длина прямой составит один дюйм (25,4 мм). Теперь, если мы захотим вывести ту же прямую на принтер с разрешением 600 dpi, то ее длина на отпечатанном рисунке будет составлять одну шестую дюйма (25,4/6 мм), т. е. будет в шесть раз короче, чем на экране.

В нашем случае разрешение экрана было 120 dpi, а разрешение принтера — 600 dpi. Поэтому рисунок на предварительном просмотре (см. рис. 3.16) и выглядел таким маленьким.

Узнать разрешение устройства вывода можно с помощью метода `GetDeviceCaps()` класса `CDC`. В качестве аргумента этой функции передается идентификатор того значения, которое требуется получить.

Например, вызов этого метода может выглядеть следующим образом:

```
// pDC-указатель на контекст устройства
// Узнаем размер рабочей области в пикселах по горизонтали
int XSize=pDC->GetDeviceCaps (HORZRES);
// Узнаем размер рабочей области в пикселах по вертикали
int YSize=pDC->GetDeviceCaps(VERTRES);
// Узнаем разрешение в пикселах на дюйм по горизонтали
int XRes=pDC->GetDeviceCaps(LOGPIXELSX);
// Узнаем разрешение в пикселах на дюйм по вертикали
int YRes=pDC->GetDeviceCaps(LOGPIXELSY);
```

Рассматриваемая проблема соотношения размеров на экране и принтере может быть решена с использованием следующего алгоритма:

1. Узнали разрешение экрана.
2. При выводе на печать узнали разрешение принтера.
3. Вычислили соотношение разрешений и масштабировали рисунок.

При программировании некоторых графических задач так и поступают. Однако мы можем воспользоваться средствами классов библиотеки MFC, которые упрощают решение этой проблемы. Можно установить такой режим отображения нашего объекта-облика, который позволит работать в пространстве логических координат.

4.3.2. Установка режима отображения

Прежде всего, заменим у нашего объекта-облика базовый класс `CView` на класс `CScrollView`, который позволит прокручивать рисунки в окне программы, если они будут превышать его размеры (класс `CView` нам таких вольностей не разрешит). Конечно, если бы мы знали, что все так обернется, то могли поменять базовый класс облика еще на 6-м шаге генератора приложений (см. рис. 3.7 — в нижней части окна диалога есть выпадающий список, из которого можно выбирать базовые классы). Однако можно сделать замену и вручную. Для этого нам потребуется всего-то открыть файл PainterView.h и дописать в названии базового класса слово "Scroll". Должно получиться так:

```
class CPainterView : public CScrollView
```

В файле же PainterView.cpp следует поменять базовый класс в карте сообщений. Должно получиться так, как в листинге 4.4.

Листинг 4.4. Карта сообщений с базовым классом `CScrollView`. Файл PainterView.cpp

```
IMPLEMENT_DYNCREATE(CPainterView, CScrollView)

BEGIN_MESSAGE_MAP(CPainterView, CScrollView)
   //{{AFX_MSG_MAP(CPainterView)
      ON_WM_LBUTTONDOWN()
   //}}AFX_MSG_MAP
   // Standard printing commands
   ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
   ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
   ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

Класс `CScrollView`, который теперь отвечает в программе Painter за вывод рисунков, позволяет нам установить так называемый *режим отображения.*

Режим отображения определяет параметры работы объекта-облика и влияет на то, как будет происходить вывод информации. Устанавливается режим отображения с помощью метода `CScrollView::SetScrollSizes()`:

```
void SetScrollSizes( int nMapMode, SIZE sizeTotal, const SIZE& sizePage =
sizeDefault, const SIZE& sizeLine = sizeDefault );
```

Параметры метода:

- `nMapMode` — режим отображения. Этот параметр может принимать значения, приведенные в табл. 4.1.

Таблица 4.1. *Режимы отображения*

Режим отображения	Логические единицы	Направление оси Y
MM_TEXT	1 пиксел	Вниз

Таблица 4.1 *(окончание)*

Режим отображения	Логические единицы	Направление оси Y
MM_HIMETRIC	0,01 мм	Вверх
MM_TWIPS	1/1440 дюйма	Вверх
MM_HIENGLISH	0,001 дюйма	Вверх
MM_LOMETRIC	0,1 мм	Вверх
MM_LOENGLISH	0,01 дюйма	Вверх

Все эти значения определены в Windows.

- `sizeTotal` — размер области просмотра в логических единицах. Член `cx` структуры `SIZE` содержит горизонтальный размер; `cy` — вертикальный. Оба значения должны быть не меньше нуля.
- `sizePage` — "размер страницы" — определяет, на сколько будет "прокручена" область просмотра при щелчке на полосе прокрутки. Член `cx` структуры `SIZE` содержит горизонтальный размер шага прокрутки; `cy` — вертикальный. По умолчанию имеет значение `sizeDefault`, которое говорит Windows, что размер страницы равен 1/10 от общего размера области просмотра, т. е. `sizePage.cx=sizeTotal.cx/10`, а `sizePage.cy=sizeTotal.cy/10`.
- `sizeLine` — "размер строки" — определяет, на сколько будет "прокручена" область просмотра при щелчке на стрелочке полосы прокрутки. Член `cx` структуры `SIZE` содержит горизонтальный размер шага прокрутки; `cy` — вертикальный. По умолчанию имеет значение `sizeDefault`, равное 1/10 от размера страницы, т. е. `sizeLine.cx=sizePage.cx/10`, а `sizeLine.cy=sizePage.cy/10`.

После того как мы установили какой-либо режим отображения, вывод информации в функции `OnDraw()` выполняется в логических единицах, а контекст устройства сам пересчитывает (масштабирует) изображение таким образом, чтобы его размеры соблюдались. По умолчанию при создании объекта-облика устанавливается режим отображения `MM_TEXT`. Логические единицы в этом случае совпадают с физическими пикселами на экране или принтере и нам потребуются дополнительные усилия, если мы захотим, чтобы размеры везде совпадали.

Давайте теперь посмотрим, как эти усилия можно уменьшить с помощью установки подходящего режима. Установим для объекта-облика в нашей

программе режим отображения MM_HIMETRIC. Теперь, если мы захотим нарисовать прямую линию длиной 100 мм, нам достаточно примерно такого фрагмента кода:

```
// Поместим перо в начальную точку
pDC->MoveTo(0, 0);
// Нарисуем прямую длиной 100 мм,
// что равняется 10000 логических единиц режима отображения MM_HIMETRIC
pDC->LineTo(10000, 0);
```

Наш облик сообщит контексту устройства, в каком режиме отображения мы работаем, а контекст устройства самостоятельно пересчитает логические единицы в физические и нарисует прямую линию длиной 100 мм.

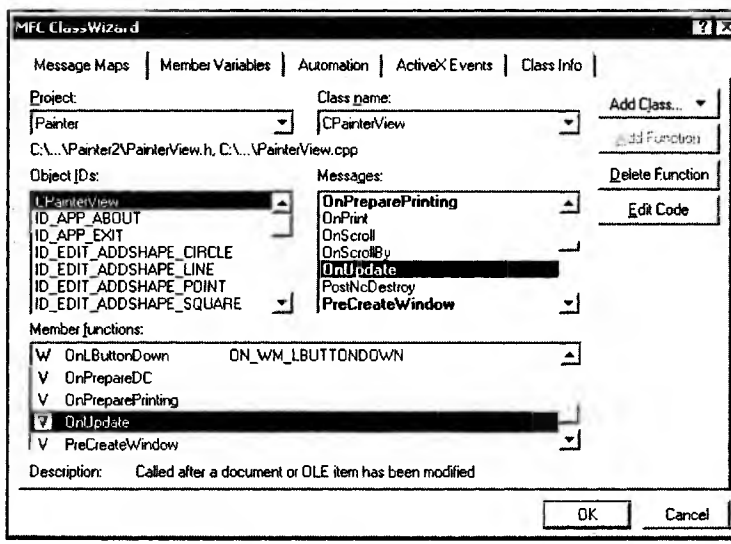


Рис. 4.2. Переопределение метода OnUpdate() с помощью инструмента ClassWizard

Режим отображения должен быть установлен после того, как объект-облик создан. Хорошее место для такой операции — метод CView::OnInitialUpdate(). Метод OnInitialUpdate() вызывается один раз после того, как объект-облик создан, но перед тем, как окно облика впервые покажется на экране. Поэтому в этой функции мы можем задать параметры, которые будут действовать на протяжении всего существования окна облика. Наряду с режимом отображения сразу должен быть указан и размер облика. Далее мы позволим пользователю изменять размер листа во время редактирования рисунка. Поэтому лучше выполнить установку режима отображения и размеров области вывода в методе CView::OnUpdate(). Отличие этого метода от OnInitialUpdate() в том, что он вызывается каждый раз, когда возникает необходимость обновить окно облика. Так как класс CView базовый для

нашего класса CPainterView, то все, что нам требуется сделать, — это переопределить метод OnUpdate(). Для того чтобы выполнить эту операцию автоматизированно, вызовем ClassWizard, выберем в списке **Class name** название класса CPainterView, а в списке сообщений — сообщение OnUpdate (рис. 4.2). Затем нажмем кнопку **Add Function**, которая добавит функцию в текст нашей программы, и кнопку **Edit Code**, которая перенесет нас в файл PainterView.cpp для редактирования метода OnUpdate().

Добавим в метод OnUpdate() установку режима отображения (листинг 4.5).

Листинг 4.5. Установка режима отображения в методе OnUpdate(). Файл PainterView.cpp

```
void CPainterView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
// TODO: Add your specialized code here and/or call the base class
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
   CSize sizeTotal;
   // Ширина
   sizeTotal.cx = pDoc->m_wSheet_Width;
   // Высота
   sizeTotal.cy = pDoc->m_wSheet_Height;
   // Установим режим и размер листа
   SetScrollSizes(pDoc->m_wMap_Mode, sizeTotal);
   // Вызываем метод базового класса
   CScrollView::OnUpdate(pSender, lHint, pHint);
}
```

Обратите внимание, размер области, в которую будет производиться вывод рисунков, мы инициализируем значениями, хранящимися в переменных m_wSheet_Width и m_wSheet_Height. Эти переменные мы введем в класс CPainterDoc специально для задания ширины и высоты листа. Кроме того, потребуется еще одна переменная для хранения режима отображения. Определение класса CPainterDoc после введения переменных приведено в листинге 4.6.

Листинг 4.6. Определение класса CPainterDoc. Файл PainterDoc.h

```
// Максимальное количество опорных точек
#define MAXPOINTS 100class CPainterDoc : public Cdocument
{
```

```
protected: // create from serialization only
   CPainterDoc();
   DECLARE_DYNCREATE(CPainterDoc)

// Данные
public:
   // Реальное количество точек в массиве
   WORD m_nIndex;
   // Массив координат опорных точек
   CPoint m_Points[MAXPOINTS];
   // Ширина листа
   WORD m_wSheet_Width;
   // Высота листа
   WORD m_wSheet_Height;
   // Режим отображения
   WORD m_wMap_Mode;

// Методы
public:

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CPainterDoc)
   public:
   virtual BOOL OnNewDocument();
   virtual void Serialize(CArchive& ar);
   //}}AFX_VIRTUAL
   // Implementation
public:
   virtual ~CPainterDoc();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
   //{{AFX_MSG(CPainterDoc)    //}}AFX_MSG    DECLARE_MESSAGE_MAP()};
```

Теперь нам потребуется инициализировать введенные в класс CPainterDoc новые переменные какими-то начальными значениями (в дальнейшем мы позволим пользователю самому задавать параметры листа). Обычно начальная инициализация выполняется в конструкторе класса. Дополним функцию-конструктор CPainterDoc() строками, в которых присвоим нашим переменным значения, по умолчанию задающие режим отображения MM_HIMETRIC и размеры листа, соответствующие формату A4 (листинг 4.7).

Листинг 4.7. Конструктор класса CPainterDoc. Файл PainterDoc.cpp

```
CPainterDoc::CPainterDoc()
{
   // Сначала в массиве нет точек
   m_nIndex=0;
   // Режим отображения 1 лог. ед. = 0,01 мм
   m_wMap_Mode = MM_HIMETRIC;
   // Размер листа формата A4
   m_wSheet_Width = 21000;
   m_wSheet_Height = 29700;
}
```

Формат листа A4 мы выбрали из соображений, что он отпечатается нормально практически на любом принтере. Установи мы размер больше, у нас бы возникли сложности с выводом на печать — наш рисунок мог бы не уместиться на одну страницу. В принципе, все проблемы можно решить, предусмотрев, например, разбивку большого листа на несколько страниц при печати. Однако подробное рассмотрение этого вопроса уведет нас несколько в сторону от темы книги. Дополнительную информацию по организации печати и предварительного просмотра можно найти в разделах *"Printing"* и *"Technical Note 30"* документации *"Visual C++ Programmer's Guide"*.

Режим отображения мы установили, но для того чтобы наша программа правильно обрабатывала "управляющие действия пользователя по созданию рисунка", потребуется внести некоторые изменения в метод CPainterView::OnLButtonDown(). Метод с изменениями приведен в листинге 4.8. Полужирным шрифтом выделен фрагмент кода, который мы ввели для того чтобы преобразовать точку из физических координат курсора на экране в логические координаты нашего рисунка. Сначала мы получаем указатель на контекст устройства, затем с помощью функции OnPrepareDC() сообщаем ему параметры нашего рисунка, в том числе и режим отображения, и используем метод CDC::DPtoLP() для перевода физических координат

точки в логические координаты. Кстати, в функции OnDraw() такая подготовка контекста устройства нам не потребуется, т. к. она выполняется автоматически.

Листинг 4.8. Измененный метод CPainterView::OnLButtonDown().Файл PainterView.cpp

```
void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();

   CPoint   LogPoint=point;

   // Получим контекст устройства, на котором рисуем
   CDC *pDC=GetDC();
   // Подготовим контекст устройства
   OnPrepareDC(pDC);
   // Переведем физические координаты точки в логические
   pDC->DPtoLP(&LogPoint);
   // Освободим контекст устройства
   ReleaseDC(pDC);

   // Проверим, не исчерпали ли ресурс
   if(pDoc->m_nIndex==MAXPOINTS)
      AfxMessageBox("Слишком много точек");
   else
   {
      // Запоминаем точку
      pDoc->m_Points[pDoc->m_nIndex++]=LogPoint;
      // Указываем, что окно надо перерисовать
      Invalidate();
      // Указываем, что документ изменен
      pDoc->SetModifiedFlag();
   }
   // Даем возможность стандартному обработчику
```

```
    // тоже поработать над этим сообщением
    CScrollView::OnLButtonDown(nFlags, point);
}
```

Функция `OnPrepareDC()` присутствует в базовом классе нашего облика. Она вызывается в тех случаях, когда надо выполнить какую-либо специальную подготовку контекста устройства перед выводом на него изображения. Например, перед тем, как указатель на контекст устройства поступает в функцию `OnDraw()`, базовый класс облика, зная, какой режим отображения мы установили, вызывает метод `OnPrepareDC()` и соответствующим образом подготавливает контекст устройства. Мы можем переопределить этот метод в классе `CPainterView`. Зачем нам это может потребоваться? Ну, например, нас не совсем устраивает, что поскольку в режиме `MM_HIMETRIC` ось *Y* направлена вверх, а точка начала координат физического устройства по умолчанию расположена в верхнем левом углу, нам придется работать в отрицательном диапазоне логических значений *Y*. Для изменения этой ситуации с помощью ClassWizard переопределим функцию `OnPrepareDC()` и добавим в нее код, приведенный в листинге 4.9. Напомним, что для переопределения метода базового класса или назначения функции-обработчика какого-либо сообщения нужно в поле **Class Name** выбрать имя класса, а в поле **Message** — имя сообщения или метода, затем нажать кнопку **Add Function**.

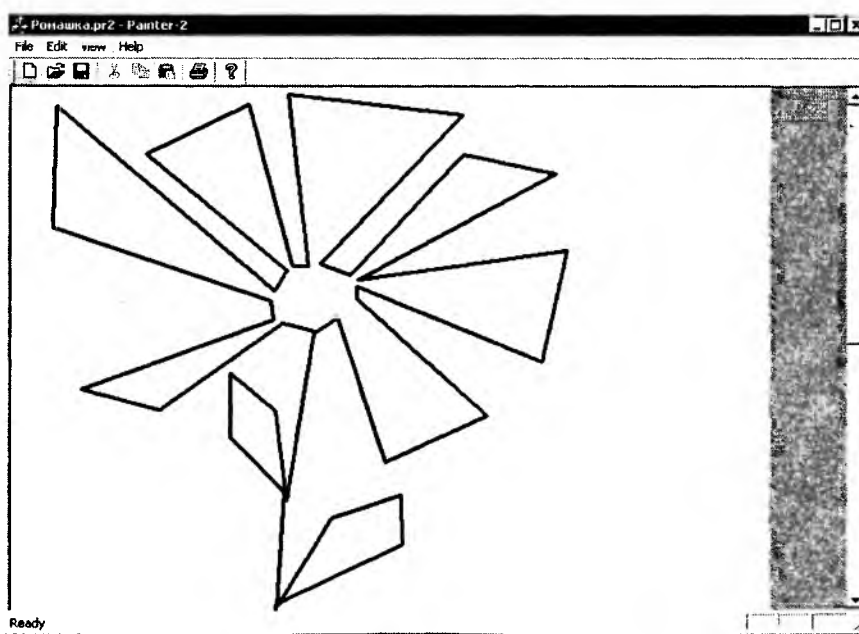


Рис. 4.3. Программа Painter с рисунком на весь лист

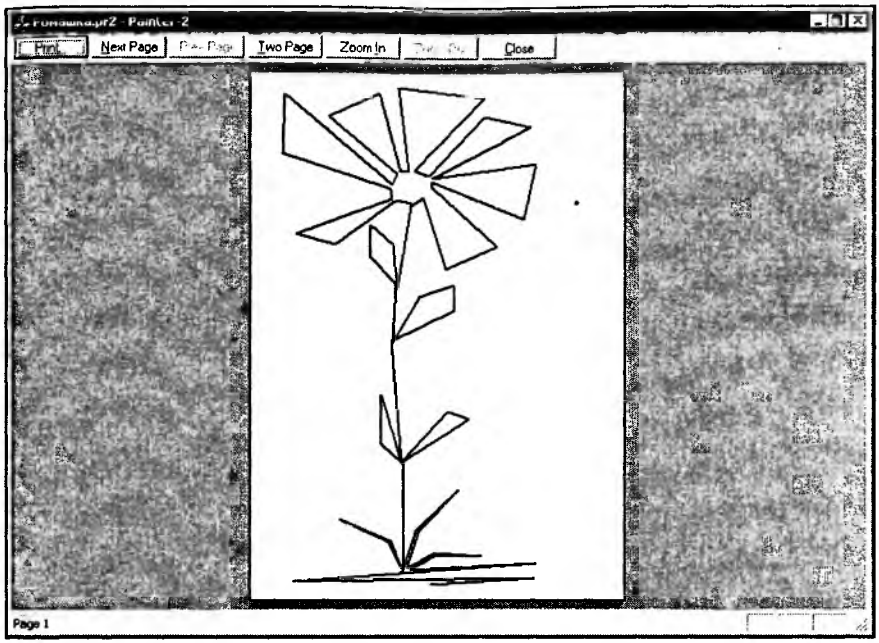


Рис. 4.4. Программа Painter с рисунком на весь лист в режиме предварительного просмотра

Листинг 4.9. Функция `CPainterView::OnPrepareDC()`. Файл PainterView.cpp

```
void CPainterView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
   // Вызов метода базового класса
   CScrollView::OnPrepareDC(pDC, pInfo);
   // Получим указатель на документ
   CPainterDoc *pDoc=GetDocument();
   // Создадим точку в левом нижнем углу листа
   CPoint OriginPoint(0, -pDoc->m_wSheet_Height);
   // Переведем точку в координаты физического устройства
   pDC->LPtoDP(&OriginPoint);
   // Установим эту точку в качестве
   // начала координат физического устройства
   pDC->SetViewportOrg(OriginPoint);
}
```

Все, теперь можно посмотреть, как наша программа заработает. Создадим рисунок размером на весь лист (рис. 4.3) и отобразим его в режиме предварительного просмотра (рис. 4.4).

4.3.3. Установка размеров листа

Дадим теперь возможность пользователю изменять размеры листа. Схема работы будет следующая:

1. Пользователь вызывает команду **File | Page property**.
2. Появляется диалоговое окно, в котором пользователь задает размеры листа.
3. Программа принимает эти размеры к сведению.

Сделать все это очень просто — достаточно добавить в программу диалог, в котором будут вводиться ширина и высота листа. Для этого откроем вкладку **ResourceView** в окне **Workspace** *(см. разд. 3.1)* и добавим шаблон нового диалога (щелкнуть правой кнопкой мыши на слове **Dialog** и выбрать из контекстного списка **Insert Dialog**). Вставим в шаблон диалога поле редактирования **Edit Box** (вторая сверху кнопка во втором столбце на панели инструментов **Controls**) и установим для него идентификатор `IDC_EDIT_WIDTH` (рис. 4.5).

Для того чтобы значения в поле редактирования можно было "прокручивать", добавим к полю редактирования элемент **Spin** (седьмой в первом столбце), установим для него идентификатор `IDC_SPIN_WIDTH` и свойства **Auto Buddy** и **Set buddy integer** (рис. 4.6). Эти элементы управления будут использоваться для задания ширины листа.

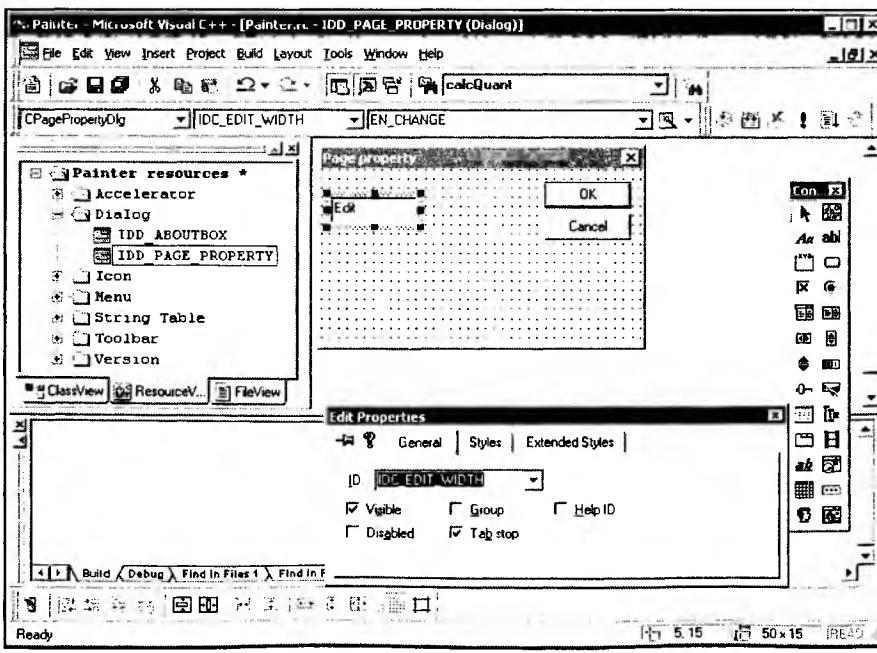


Рис. 4.5. Шаблон нового диалогового окна

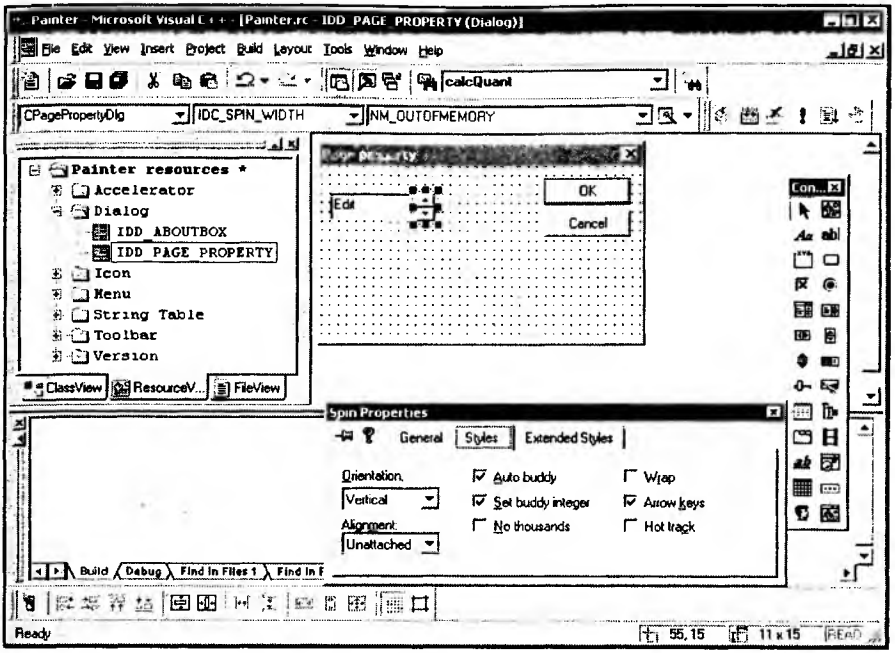


Рис. 4.6. Добавление в шаблон элемента **Spin**

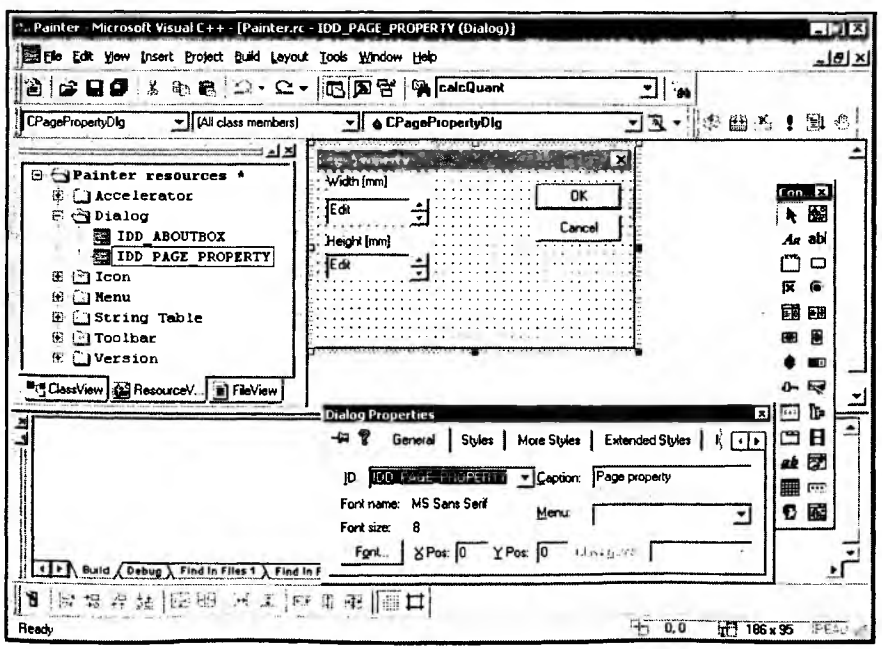


Рис. 4.7. Шаблон диалогового окна после редактирования

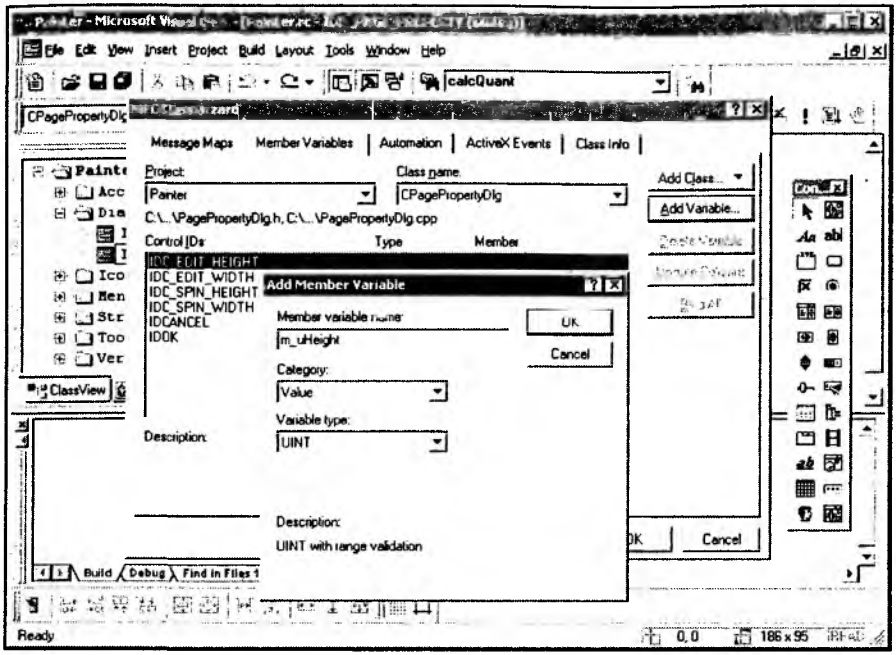


Рис. 4.8. Добавление переменной в класс `CPagePropertyDlg`

Создадим такие же элементы управления с идентификаторами `IDC_EDIT_HEIGHT` и `IDC_SPIN_HEIGHT` для задания высоты листа. Напоследок сделаем подписи Width и Height к полям ввода с помощью элементов **Static Text**, озаглавим диалог **Page property** и назовем его идентификатор `IDD_PAGE_PROPERTY`. В результате должно получиться что-то похожее на рис. 4.7.

Вызовем генератор классов ClassWizard с помощью комбинации клавиш <Ctrl>+<W>. Он предложит создать нам новый класс для нашего диалога. Согласимся, и назовем класс `CPagePropertyDlg`. Создадим теперь переменные, в которых будут храниться размеры листа. Для этого перейдем на вкладку **Member Variables** нашего нового класса `CPagePropertyDlg`. Выделим идентификатор `IDC_EDIT_HEIGHT` поля редактирования высоты листа и нажмем кнопку **Add Variable**. Появится диалоговое окно **Add Member Variable**, в котором назовем новую переменную `m_uHeight` и зададим ее тип UINT (рис. 4.8).

Аналогично создадим переменную `m_uWidth` для поля редактирования ширины.

После создания этих переменных для них можно задать диапазон доступных значений (при выделении имени переменной в нижней части диалогового окна **MFC ClassWizard** появляются поля ввода, в которых можно задать диапазон). Зададим для ширины `m_uWidth` диапазон от 0 до 210, а для высоты `m_uHeight` от 0 до 297 (размеры мы будем указывать в мм, так что допустимый диапазон будет соответствовать формату листа A4). Соблюдение этого диапазона значений будет проверяться автоматически.

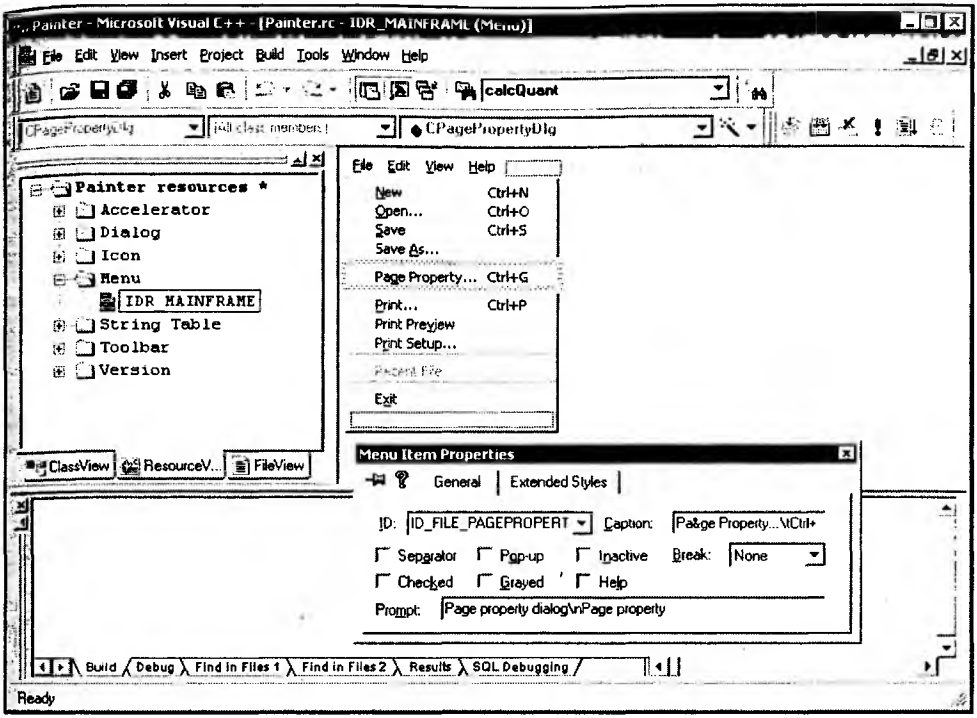


Рис. 4.9. Редактирование меню

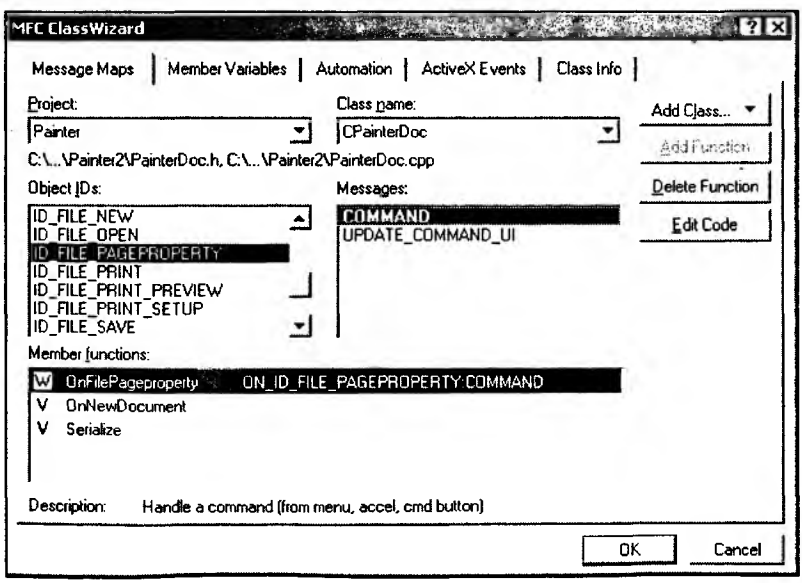


Рис. 4.10. Назначение функции-обработчика для команды `ID_FILE_PAGEPROPERTY`

Для элементов управления IDC_SPIN_WIDTH и IDC_SPIN_HEIGHT создадим переменные m_ctlSpinHeight и m_ctlSpinWidth типа CSpinButtonCtrl — они потребуются нам для управления этими Spin-элементами.

Добавим теперь в класс CPagePropertyDlg функцию-обработчик сообщения WM_INITDIALOG, которое поступает перед тем, как диалоговое окно будет показано на экране. В функции-обработчике зададим диапазон прокрутки для Spin-элементов (листинг 4.10).

Листинг 4.10. Функция CPagePropertyDlg::OnInitDialog(). Файл PagePropertyDlg.cpp

```
BOOL CPagePropertyDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
   // TODO: Add extra initialization here
   // Установим диапазон прокрутки Spin-элементов
   m_ctlSpinWidth.SetRange(0, 210);
   m_ctlSpinHeight.SetRange(0, 297);
   return TRUE;  // return TRUE unless you set the focus to a control
   // EXCEPTION: OCX Property Pages should return FALSE
}
```

Теперь надо добавить команду **Page property** в меню **File**. Для этого откроем вкладку **Resource View** и папку **Menu** и дважды щелкнем левой кнопкой мыши на ресурсе IDR_MAINFRAME. Откроется шаблон меню программы. Раскроем меню **File**, а затем отредактируем содержимое пустого пункта (внизу меню) и перетащим его на нужно место. Должно получиться что-то похожее на рис. 4.9.

Все, что осталось сделать, — это добавить обработчик этой команды в класс документа. Вызовем ClassWizard и для идентификатора команды ID_FILE_PAGEPROPERTY назначим метод-обработчик OnFilePageproperty() (рис. 4.10). Да, и не забудьте включить заголовочный файл PagePropertyDlg.h в PainterDoc.cpp.

Отредактируем метод OnFilePageproperty(), как показано в листинге 4.11.

Листинг 4.11. Метод CPainterDoc::OnFilePageproperty(). Файл PainterDoc.cpp

```
void CPainterDoc::OnFilePageproperty()
{
   // TODO: Add your command handler code here
   // Создаем объект – диалог свойств листа
```

```
    CPagePropertyDlg PPDlg;
    // Инициализируем параметры диалога текущими значениями
    // делим на 100, т. к. в диалоге размеры в мм
    PPDlg.m_uWidth=m_wSheet_Width/100;
    PPDlg.m_uHeight=m_wSheet_Height/100;
    // Вызываем диалог
    if(PPDlg.DoModal()==IDOK)
    {   // Запоминаем новые значения
        // умножаем на 100, т. к. 1 лог. ед. = 0,01 мм
        m_wSheet_Width=PPDlg.m_uWidth*100;
        m_wSheet_Height=PPDlg.m_uHeight*100;
        // Обновляем облик
        UpdateAllViews(NULL);
    }
}
```

Осталось лишь подвести логическую черту под нашими действиями по установке размеров листа. Во-первых, если установлены размеры листа, рисовать за его пределами не положено. Реализовать такое ограничение легко — мы просто-напросто при подготовке контекста устройства в методе `OnPrepareDC()` ограничим область на контексте устройства, в которой возможно рисование (листинг 4.12).

Листинг 4.12. Метод `CPainterView::OnPrepareDC()`. Файл PainterView.cpp

```
void CPainterView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // Вызов метода базового класса
    CScrollView::OnPrepareDC(pDC, pInfo);
    // Получим указатель на документ
    CPainterDoc *pDoc=GetDocument();
    // Создадим точку в левом нижнем углу листа
    CPoint OriginPoint(0, -pDoc->m_wSheet_Height);
    // Переведем точку в координаты физического устройства
    pDC->LPtoDP(&OriginPoint);
    // Установим эту точку в качестве
    // начала координат физического устройства
    pDC->SetViewportOrg(OriginPoint);
    // Ограничим область, доступную для рисования
    pDC->IntersectClipRect( 0,0, pDoc->m_wSheet_Width,
                                 pDoc->m_wSheet_Height);
}
```

Во-вторых, пользователь должен видеть границы листа. Для реализации этого требования будем заполнять серым цветом недоступную для рисования область окна. Заливку будем выполнять в методе-обработчике `CPainterView::OnEraseBkgnd()` сообщения `WM_ERASEBKGND`, добавленного с помощью ClassWizard (листинг 4.13).

Листинг 4.13. Метод `CPainterView::OnEraseBkgnd()`. Файл PainterView.cpp

```
BOOL CPainterView::OnEraseBkgnd(CDC* pDC)
{
   // Вызвали метод базового класса
   BOOL Res=CScrollView::OnEraseBkgnd(pDC);
   // Создали кисть серого цвета
   CBrush br( GetSysColor( COLOR_GRAYTEXT ) );
   // Выполнили заливку неиспользуемой области окна
   FillOutsideRect( pDC, &br );
   return Res;
}
```

Работа программы с новыми возможностями показана на рис. 4.11 и 4.12.

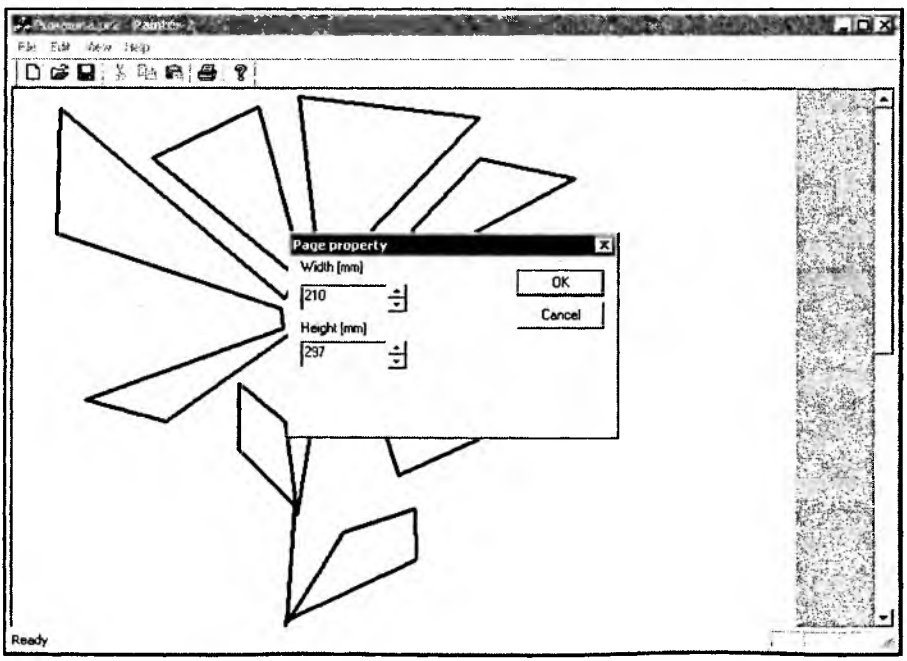


Рис. 4.11. Диалог установки размеров листа

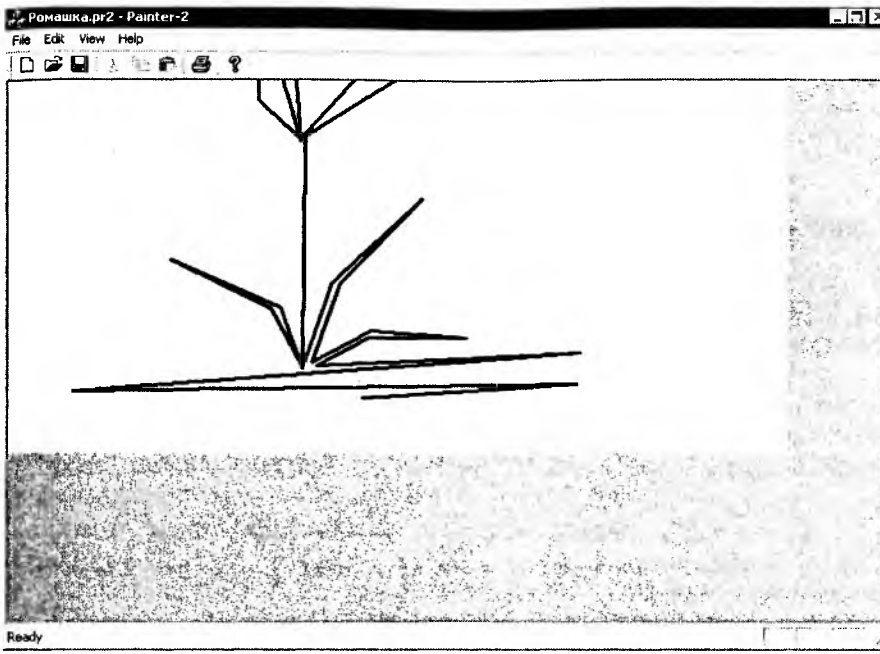


Рис. 4.12. Рисунок на листе с новыми размерами

Поскольку точка начала координат находится в левом нижнем углу (см. листинг 4.9), то после уменьшения размеров листа (см. рис. 4.11) верхняя часть рисунка не отображается (см. рис. 4.12). Можно придумать и другие схемы обрезки изображений.

4.3.4. Реализация функций рисования примитивов

Добавим в новую версию программы Painter возможность рисовать не только прямые, но и примитивные фигуры, например круг и квадрат. Заодно вспомним, как реализуются виртуальные функции.

Краткий план нашей работы.

1. Начнем с того, что создадим иерархию классов графических объектов — фигур. В основу иерархии поместим базовый класс, который будет отражать свойства и методы, общие для любой фигуры.
2. От базового класса породим производные классы, которые будут отвечать за рисование различных фигур. В дальнейшем каждая простая фигура рисунка будет представлена объектом соответствующего класса.
3. Введем в класс документа `CPainterDoc` механизм хранения объектов-фигур.

4. Дополним метод сохранения и загрузки изображений из файла класса `CPainterDoc` средствами для корректной работы с версиями форматов файлов.
5. Дополним интерфейс программы Painter командами для создания различных фигур.

Создание базового класса иерархии фигур

Представим, какие свойства являются общими для простых фигур типа круг, прямоугольник и т. п. Каждая такая фигура может характеризоваться:

- положением на экране;
- размером;
- цветом и толщиной линий;
- цветом и стилем заливки.

Эти общие для большей части фигур свойства целесообразно реализовать в базовом классе.

- Теперь подумаем, что объект-фигура должна уметь. Ну, как минимум, от нее можно потребовать, чтобы она могла:
- нарисовать себя на указанном контексте устройства;
- сообщить (в случае, если кто поинтересуется) область на экране, которую занимает ее изображение.

Формирование иерархии классов очень похоже на построение генеалогического древа. Как и родственные отношения между людьми, отношения между классами могут быть описаны в терминах предок-потомок. Предки значат очень много, поскольку их генетика во многом определяет, какими станут потомки. В отличие от родственных отношений, при написании классов у нас гораздо больше свободы воли, мы можем выбирать предков, перенимать от них самое лучшее и определять, что достанется нашим потомкам. Конечно, для того чтобы выбрать достойного предка, надо знать о них. Например, Эдди Кей[1] "мог перечислить более шестисот своих предков и о каждом рассказать хотя бы один анекдот". Нам же стоит присмотреться к классам MFC и подумать, найдется ли подходящий предок для нашего "фигурного" класса.

Положение на экране задается парой координат — базовой точкой, относительно которой рисуется вся фигура. Для хранения координат точек в библиотеке MFC есть специальный класс `CPoint`, который мы рассмотрели в *разд. 3.2.* Класс `CPoint` имеет удобные средства для хранения и управления точками, поэтому мы можем применить его при создании нашего класса фигур. В дальнейшем нам также потребуются средства для динамического создания объектов

[1] Персонаж романа Курта Воннегута "Завтрак для чемпионов, или Прощай, черный понедельник".

и хранения их в памяти, а также для сохранения в файл объектов-фигур. Поддержка этих возможностей уже реализована в классе CObject библиотеки MFC. Класс CObject является базовым для большинства классов библиотеки MFC (но не для CPoint). Его методы нам тоже пригодятся. Итак, решено, мы порождаем наш класс, назовем его CBasePoint, от классов CPoint и CObject.

Для хранения информации о размере фигуры заведем переменную m_wSize типа WORD. Этот параметр для разных фигур может иметь разный смысл, например, для круга это может быть радиус, а для квадрата — длина стороны.

Для хранения параметров отображения фигур добавим в класс переменные, которые задают толщину и цвет линий, способ закраски (заливки) внутренних частей фигур (см. листинг 4.14). В классе также присутствуют два объекта: перо — m_Pen класса CPen и кисть — m_Brush класса CBrush. Они будут использоваться при рисовании фигуры. Позже мы рассмотрим, как кисть может быть создана на основе шаблона — растрового изображения (bitmap). Для хранения идентификатора растрового изображения в класс введена специальная переменная m_dwPattern_ID. Благодаря этому, информация о том, какой шаблон используется кистью, может быть записана в файл при сохранении изображения и считана из него. Для хранения информации о цвете используется тип COLORREF, который определяет 32-битное число (DWORD), специально предназначенное для хранения цвета модели RGB. Значение COLORREF формируется из трех компонентов с помощью макроса RGB():

```
COLORREF RGB(
   BYTE byRed,     // красный компонент цвета
   BYTE byGreen,   // зеленый компонент цвета
   BYTE byBlue     // синий компонент цвета
);
```

Например, создадим светло-серый цвет:

```
COLORREF LightGrayColor = RGB(200, 200, 200);
```

Объявление класса поместим в специально созданный файл Shaps.h (листинг 4.14).

Листинг 4.14. Определение класса CBasePoint. Файл Shaps.h

```
class CBasePoint: public CPoint, public CObject
{
   DECLARE_SERIAL(CBasePoint)
   CPen   m_Pen;      // перо
   CBrush   m_Brush; // кисть
protected:
   // Метод сериализации
   virtual void Serialize(CArchive& ar);
```

```
    // Подготавливает контекст устройства
    virtual BOOL PrepareDC(CDC *pDC);
    // Восстанавливает контекст устройства
    virtual BOOL RestoreDC(CDC *pDC);
public:
    // Данные
    WORD    m_wSize;                    // размер фигуры
    int     m_iPenStyle;                // стиль линий
    int     m_iPenWidth;                // ширина линий
    COLORREF    m_rgbPenColor;          // цвет линий
    int     m_iBrushStyle;              // стиль заливки
    COLORREF    m_rgbBrushColor;        // цвет заливки
    DWORD   m_dwPattern_ID;             // идентификатор шаблона заливки
public:
    // Конструкторы
    CBasePoint();                           // конструктор без параметров
    CBasePoint(int x, int y, WORD s);       // конструктор с параметрами
    ~CBasePoint(){};                        // деструктор
    // Методы
    // Отображает фигуру на экране
    virtual void Show(CDC *pDC);
    // Сообщает область захвата
    virtual void GetRegion(CRgn &Rgn);
    // Устанавливает параметры линий
    virtual BOOL SetPen(COLORREF color, int width =1,
                                        int style=PS_SOLID);
    // Устанавливает параметры заливки
    virtual BOOL SetBrush(COLORREF color, DWORD pattern =0,
                                        int style=-1);
};
```

Прежде всего в определении класса мы задали прототипы двух конструкторов. Конструктор без параметров обязателен для выполнения операций сериализации. Конструктор с параметрами позволяет нам задать свойства объекта-фигуры при его создании.

Макрокоманды

```
DECLARE_SERIAL(имя_класса) и
IMPLEMENT_SERIAL(имя_класса, имя_базового_класса,
                             версия_формата_файла)
```

в совокупности с функцией Serialize() обеспечивают возможность удобного сохранения и считывания (сериализации) объекта. Макрос DECLARE_SERIAL должен присутствовать в заголовочном H-файле с объявлением класса, а макрос IMPLEMENT_SERIAL помещается в CPP-файл с реализацией методов класса. Метод Serialize() класс CBasePoint унаследовал от своего базового класса CObject. Мы переопределим этот метод так, чтобы он сохранял данные класса CBasePoint. Метод Serialize() при необходимости можно переопределять в классах, производных от CBasePoint.

Параметры рисования линий и внутренней заливки фигур будут определять свойства объектов: перо — m_Pen и кисть — m_Brush. Свойства кисти и пера задаются с помощью функций SetPen() и SetBrush(). Для каждого объекта-фигуры, например при его создании, могут быть заданы свои атрибуты рисования. Если свойства объектов перо и кисть не заданы, то при рисовании фигуры будут использоваться атрибуты рисования, принятые в системе по умолчанию.

В объявление класса CBasePoint также включены прототипы двух виртуальных функций: Show() и GetRegion(). Функция Show() будет рисовать фигуру на указанном контексте устройства, а функция GetRegion() будет сообщать область, занимаемую фигурой. Эта функция может пригодиться, например в случае, когда мы хотим выполнить операцию выбора какой-то фигуры в рисунке. Функции объявлены виртуальными и будут переопределяться в производных классах. Виртуальность этих функций позволит по-своему рисовать каждую из фигур нашей будущей иерархии классов фигур.

Реализацию методов класса CBasePoint поместим в файл Shapes.cpp (листинг 4.15).

Листинг 4.15. Реализация методов класса CBasePoint. Файл Shapes.cpp

```
#include "stdafx.h"
#include "shapes.h"

////////////////////////////////////
// Реализация методов класса CBasePoint
CBasePoint::CBasePoint(): CPoint(0, 0)
{
   m_wSize=1;

   m_iPenStyle=PS_SOLID;
   m_iPenWidth=1;
   m_rgbPenColor=RGB(0,0,0);
```

```
   m_iBrushStyle=-1; // не используем штриховку
   m_rgbBrushColor=RGB(0,0,0);
   m_dwPattern_ID=0; // нет шаблона заливки
};

CBasePoint::CBasePoint(int x, int y, WORD s):CPoint(x, y)
{
   m_wSize=s;

   m_iPenStyle=PS_SOLID;
   m_iPenWidth=1;
   m_rgbPenColor=RGB(0,0,0);
   m_iBrushStyle=-1; // не используем штриховку
   m_rgbBrushColor=RGB(0,0,0);
   m_dwPattern_ID=0; // нет шаблона заливки
};

IMPLEMENT_SERIAL(CBasePoint, CObject , VERSIONABLE_SCHEMA|1)
void CBasePoint::Serialize(CArchive &ar)
{
   if(ar.IsStoring()) // сохранение
   {
      // Сохраняем параметры объекта
      ar<<x;
      ar<<y;
      ar<<m_wSize;
      ar<<m_iPenStyle;
      ar<<m_iPenWidth;
      ar<<m_rgbPenColor;
      ar<<m_iBrushStyle;
      ar<<m_rgbBrushColor;
      ar<<m_dwPattern_ID;
   }
   else   // чтение
   {
      // Получили версию формата
      int Version=ar.GetObjectSchema();
      // В зависимости от версии
      // можно выполнить различные варианты загрузки
      // Загружаем параметры объекта
      ar>>x;
```

```
      ar>>y;
      ar>>m_wSize;
      ar>>m_iPenStyle;
      ar>>m_iPenWidth;
      ar>>m_rgbPenColor;
      ar>>m_iBrushStyle;
      ar>>m_rgbBrushColor;
      ar>>m_dwPattern_ID;

      SetPen(m_rgbPenColor, m_iPenWidth, m_iPenStyle);
      SetBrush(m_rgbBrushColor, m_dwPattern_ID, m_iBrushStyle );
   }
};
BOOL CBasePoint::SetPen(COLORREF color, int width /*=1*/,
                                        int style/*=PS_SOLID*/)
{
   m_iPenStyle=style;
   m_iPenWidth=width;
   m_rgbPenColor=color;
   if(HPEN(m_Pen)!=NULL)    // Если перо уже существует
   if(!m_Pen.DeleteObject()) return FALSE; // удалили старое перо
   // Создаем новое перо и возвращаем результат
   return m_Pen.CreatePen( m_iPenStyle, m_iPenWidth, m_rgbPenColor);
};
BOOL CBasePoint::SetBrush(COLORREF color, DWORD pattern /*=0*/,
                                          int style/*=-1*/)
{
   m_iBrushStyle=style;
   m_dwPattern_ID=pattern;
   m_rgbBrushColor=color;
   int res=1;
   if(HBRUSH(m_Brush)!=NULL)    // Если кисть уже существует
   if(!m_Brush.DeleteObject()) return FALSE; // удалили старую кисть
   if(m_dwPattern_ID>0)    // есть шаблон заливки
   {
      CBitmap Pattern;
      if(!Pattern.LoadBitmap(m_dwPattern_ID)) return FALSE;
      return m_Brush.CreatePatternBrush(&Pattern);
   }
```

```
   if(m_iBrushStyle>=0)    // указан стиль штриховки
      return m_Brush.CreateHatchBrush( m_iBrushStyle, m_rgbBrushColor);
   // Создаем сплошную кисть и возвращаем результат
   return m_Brush.CreateSolidBrush(m_rgbBrushColor);
};

BOOL CBasePoint::PrepareDC(CDC *pDC)
{
   // Сохраняем состояние контекста устройства
   if(!pDC->SaveDC()) return FALSE;
   // Устанавливаем перо и кисть
   if(HPEN(m_Pen)!=NULL)
      pDC->SelectObject(&m_Pen);
   if(HBRUSH(m_Brush)!=NULL)
      pDC->SelectObject(&m_Brush);
   return TRUE;
};

BOOL CBasePoint::RestoreDC(CDC *pDC)
{
   // Восстанавливаем состояние контекста устройства
   return pDC->RestoreDC(-1);
};

void CBasePoint::Show(CDC* pDC)
{
   // Устанавливаем перо и кисть
   PrepareDC(pDC);
   // Рисуем кружок, обозначающий точку
   pDC->Ellipse(x-m_wSize, y-m_wSize, x+m_wSize, y+m_wSize);
   // Восстанавливаем контекст
   RestoreDC(pDC);
}

void CBasePoint::GetRegion(CRgn &Rgn)
{
   Rgn.CreateEllipticRgn(x-m_wSize, y-m_wSize, x+m_wSize, y+m_wSize);
}
```

Коротко рассмотрим, что выполняется в каждой из функций класса CBasePoint:

- `CBasePoint::CBasePoint(): CPoint(0, 0)`

  Конструктор без параметров. Инициализируются переменные класса.

- `CBasePoint::CBasePoint(int x, int y, WORD s):CPoint(x, y)`

  Конструктор с параметрами. Переменные класса инициализируются переданными в функцию значениями.

- `void CBasePoint::Serialize(CArchive &ar)`

  Метод сериализации. В качестве аргумента принимает ссылку на уже подготовленный объект класса CArchive, который обеспечивает обмен данными с файлом. В этом методе выполняется сериализация координат и размера фигуры. К сожалению, классы CPen и CBrush не имеют своих методов сериализации, хотя и являются производными от класса CObject. Поэтому мы сохраняем атрибуты рисования фигуры, а при загрузке на основе атрибутов устанавливаем параметры пера и кисти. Надо отметить, что в методе сериализации можно выполнить проверку формата данных объекта и, при необходимости, по-разному выполнять загрузку. Формат объекта может быть указан в третьем параметре макроса IMPLEMENT_SERIAL. В листинге 4.15 мы указали VERSIONABLE_SCHEMA|1, что означает формат №1 (имеется в виду формат данных объекта, а не файла). Флаг VERSIONABLE_SCHEMA говорит о том, что мы допускаем чтение различных форматов.

- ```
  BOOL CBasePoint::SetPen(COLORREF color, int width /*=1*/,
                                          int style/*=PS_SOLID*/)
  ```

  Этот метод используется для установки параметров пера. Первый параметр задает цвет, второй — толщину, а третий — стиль пера. Перо может иметь разные стили, идентификаторы которых определены в файле wingdi.h. Не каждый стиль допускает ширину пера более одной физической единицы, однако некоторые стили, например PS_GEOMETRIC, позволяют указать ширину в логических единицах.

- ```
  BOOL CBasePoint::SetBrush(COLORREF color, DWORD pattern /*=0*/,
                                            int style/*=-1*/)
  ```

  Этот метод используется для установки параметров кисти. Первый параметр задает цвет, второй — идентификатор шаблона, третий — стиль штриховки. Если указан идентификатор шаблона, то кисть создается на основе шаблона, а параметры цвета и стиля игнорируются. Для того чтобы создать штриховую кисть, требуется указать цвет и стиль штриховки, а параметр pattern должен быть равен нулю. Идентификаторы различных стилей кисти определены в файле wingdi.h. Сплошная кисть создается в том случае, если при вызове этой функции указан только параметр цвета.

- `BOOL CBasePoint::PrepareDC(CDC *pDC)`

  Метод предназначен для подготовки контекста устройства. В данной реализации устанавливает перо и кисть.

- `BOOL CBasePoint::RestoreDC(CDC *pDC)`

  Восстанавливает состояние контекста устройства, предшествующее вызову `PrepareDC()`.

- `void CBasePoint::Show(CDC* pDC)`

  Этот метод получает указатель на контекст устройства, подготавливает контекст и использует его методы для вывода изображения. Затем атрибуты контекста восстанавливаются.

- `void CBasePoint::GetRegion(CRgn &Rgn)`

  Метод используется для определения области, занимаемой фигурой. В качестве параметра метод получает ссылку на объект класса `CRgn`. `CRgn` — специальный класс библиотеки MFC, предназначенный для описания областей (регионов). Этот класс имеет методы для создания областей различной формы, а также метод для определения попадания точки внутрь области. Поскольку в методе `Show()` объект-точка изображается как круг, в этом методе определяется эллиптическая область. Метод `GetRegion()` будет далее использован в операции выбора фигур на рисунке.

Создание производных классов иерархии фигур

После того как базовый класс определен, можно перейти к созданию производных от него классов, которые будут описывать различные фигуры. Для начала создадим класс `CSquare` для рисования квадратов. Наша иерархия классов будет выглядеть, как показано на рис. 4.13. Класс `CNoname` на этой схеме символизирует будущие классы фигур.

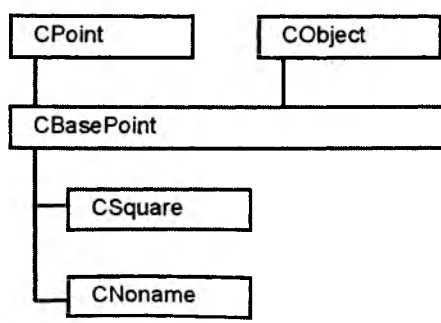


Рис. 4.13. Иерархия классов фигур

Описание класса CSquare приведено в листинге 4.16. Все что потребовалось сделать, — это переопределить несколько функций. Столь мало усилий при создании класса CSquare нам потребовалось потому, что "сквозь него на мир глядят его предки" — класс CBasePoint и умелые классы библиотеки MFC.

Листинг 4.16. Описание класса CSquare. Файл Shapes.h

```
class CSquare: public CBasePoint
{
   DECLARE_SERIAL(CSquare)
protected:
   // Метод сериализации
   void Serialize(CArchive& ar);
public:
   // Конструкторы
   CSquare(int x, int y, WORD s);
   CSquare();
   ~CSquare(){};
   // Методы
   // Отображает фигуру на экране
   void Show(CDC *pDC);
   // Сообщает область захвата
   void GetRegion(CRgn &Rgn);
};
```

Реализация методов класса CSquare приведена в листинге 4.17. Функция Show() теперь рисует квадрат, а функция GetRegion(), соответственно, определяет квадратную область. Метод же Serialize() только и делает, что вызывает одноименный метод базового класса.

Листинг 4.17. Реализация методов класса CSquare. Файл Shapes.cpp

```
CSquare::CSquare(int x, int y, WORD s): CBasePoint(x, y, s)
{
   m_wSize=s;
}

CSquare::CSquare(): CBasePoint()
{
```

```
    m_wSize=40;
}

IMPLEMENT_SERIAL(CSquare, CObject, 1)
void CSquare::Serialize(CArchive &ar)
{
    CBasePoint::Serialize(ar);
}
void CSquare::Show(CDC* pDC)
{
    int s=m_wSize/2;
    // Устанавливаем перо и кисть
    PrepareDC(pDC);
    // Рисуем квадрат
    pDC->Rectangle(x-s, y-s, x+s, y+s);
    // Восстанавливаем контекст
    RestoreDC(pDC);
}

void CSquare::GetRegion(CRgn &Rgn)
{
    int s=m_wSize/2;
    Rgn.CreateRectRgn(x-s, y-s, x+s, y+s);
}
```

Теперь осталось только включить файл Shapes.cpp в проект Painter. Для этого выполним команду **Project | Add to project-Files**, укажем имя файла в поле **File name**, нажмем кнопку **OK**.

Модификация класса документа *CPainterDoc*

В процессе работы с программой Painter пользователь сможет создавать большое число различных геометрических фигур. Внутри программы это будет выглядеть как динамическое создание объектов. Хранение создаваемых в программе объектов-фигур удобно организовать в виде списка указателей. В библиотеке MFC имеется параметризованный класс `CTypedPtrList`, который мы используем для создания специального объекта-списка. Объект-список будет выполнять всю работу по хранению и управлению указателями на объекты-фигуры. Объект-список определим с помощью шаблона параметризованного класса `CTypedPtrList`, где параметром типа будет `CObList` — класс, реализующий работу со списком объектов, а параметром аргумента — указатель на класс `CBasePoint`. Таким образом, в списке будут храниться указатели на класс `CBasePoint`. Это позволит обеспечить полиморфизм списка, т. к. в нем можно будет хранить указатели на любые объекты-фигуры, производных от `CBasePoint`

классов. Объект-список сделаем членом класса CPainterDoc. В описание класса CPainterDoc добавим следующие строки:

```
// Список указателей на объекты-фигуры
   CTypedPtrList<CObList, CBasePoint*> m_ShapesList;
```

Для поддержки работы с классами шаблонов требуется подключить файл afxtempl.h. В проект Painter включен файл StdAfx.h. Этот файл создан генератором приложений и предназначен для того, чтобы в нем подключались системные и часто используемые заголовочные файлы. Для подключения файла afxtempl.h добавим в файл StdAfx.h следующую строку:

```
#include <afxtempl.h>    // Работа с шаблонами
```

Включение в меню команд добавления фигур

Теперь настала пора пополнить меню программы Painter командами добавления в рисунок простых фигур. Как добавлять в меню команду, мы уже рассмотрели в *разд. 4.3.3.* Поэтому не будем на этом подробно останавливаться, нам достаточно лишь взглянуть на рис. 4.14, и уже все понятно.

При использовании архитектуры приложений Document-View за взаимодействие с пользователем и модификацию данных документа отвечает объект-облик. Поэтому воспользуемся средствами ClassWizard и добавим в класс CPainterView функции-обработчики для каждой из этих команд.

Традиционно схема действий при добавлении новой фигуры в рисунок примерно следующая:

1. Пользователь выбирает, какую фигуру он хочет добавить в рисунок.
2. Пользователь указывает щелчком мыши место фигуры на рисунке.

Такая схема подразумевает, что наша программа должна различать щелчки мыши. Для того чтобы научить этому программу, введем в класс CPainterView специальную переменную m_CurOper типа int и определим с помощью директивы #define ряд значений:

```
// Определение операций
#define OP_NOOPER   0
#define OP_LINE     1
#define OP_POINT    2
#define OP_CIRCLE   3
#define OP_SQUARE   4
```

Мы использовали #define для определения кода операций, однако для этих же целей можно использовать и перечисление enum, что в ряде случаев более удобно.

В функциях-обработчиках команд создания фигур будем присваивать этой переменной m_CurOper соответствующие значения (листинг 4.18).

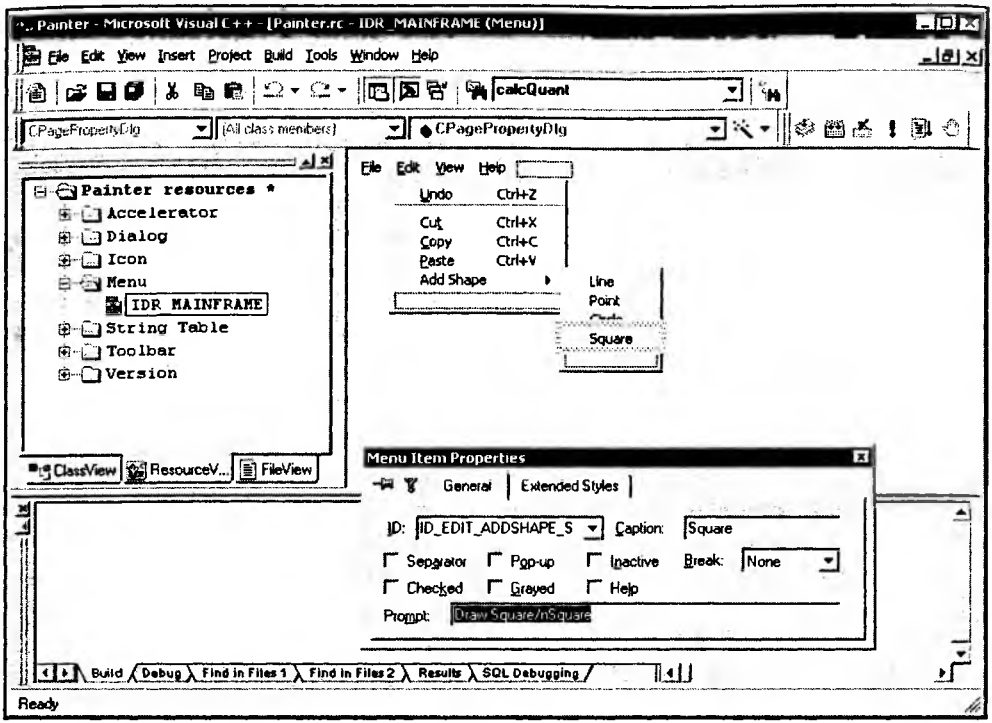


Рис. 4.14. Добавление в меню команд рисования фигур

Листинг 4.18. Функции-обработчики команд создания фигур. Файл PainterView.cpp

```
void CPainterView::OnEditAddshapeLine()
{
   // TODO: Add your command handler code here
   m_CurOper=OP_LINE;
}

void CPainterView::OnEditAddshapePoint()
{
   // TODO: Add your command handler code here
   m_CurOper=OP_POINT;
}

void CPainterView::OnEditAddshapeCircle()
{
```

```
   // TODO: Add your command handler code here
   m_CurOper=OP_CIRCLE;
}

void CPainterView::OnEditAddshapeSquare()
{
   // TODO: Add your command handler code here
   m_CurOper=OP_SQUARE;
}
```

В функции-обработчике нажатия левой клавиши мыши `OnLButtonDown()` с помощью оператора `switch()` будем производить выбор реагирования программы на нажатие (листинг 4.19).

Листинг 4.19. Функция `CPainterView::OnLButtonDown`. Файл PainterView.h

```
void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
   CPoint   LogPoint=point;

   // Получим контекст устройства, на котором рисуем
   CDC *pDC=GetDC();
   // Подготовим контекст устройства
   OnPrepareDC(pDC);

   // Переведем физические координаты точки в логические
   pDC->DPtoLP(&LogPoint);
   // Освободим контекст устройства
   ReleaseDC(pDC);
   switch(m_CurOper)
   {
      case OP_LINE:
         // Проверим, не исчерпали ли ресурс
         if(pDoc->m_nIndex==MAXPOINTS)
         AfxMessageBox("Слишком много точек");
         else
         {
             // Запоминаем точку
             pDoc->m_Points[pDoc->m_nIndex++]=LogPoint;
             // Указываем, что окно надо перерисовать
             Invalidate();
```

```
            // Указываем, что документ изменен
            pDoc->SetModifiedFlag();
        }
    break;
    case OP_POINT:
    case OP_CIRCLE:
    case OP_SQUARE:
    AddShape(m_CurOper, LogPoint);
    break;
  }

  // Даем возможность стандартному обработчику
  // тоже поработать над этим сообщением

  CScrollView::OnLButtonDown(nFlags, point);
}
```

Обратите внимание, при операции OP_LINE у нас выполняется уже знакомый нам фрагмент кода — рисование линий, а в случае, если значением m_CurOper является OP_POINT, OP_CIRCLE или OP_SQUARE, вызывается функция AddShape(). Как сказал Борхес: "Каждый рождается, где может", — наши же объекты-фигуры создаются в AddShape(). Эту функцию мы тоже добавили в класс CPainterView. AddShape(), который имеет два параметра: код операции (фигуры) и координаты точки, в которую фигура должна быть вставлена (листинг 4.20).

Листинг 4.20. Функция AddShape(). Файл PainterView.cpp

```
void CPainterView::AddShape(int shape, CPoint point)
{
  CPainterDoc *pDoc=GetDocument();
  CBasePoint *pShape=NULL;
  switch(shape)
  {
    case OP_POINT:
      // Создаем объект-точку
      pShape=new CBasePoint(point.x, point.y, 100);
      // Светло-серая заливка
      pShape->SetBrush(RGB(200,200,200));
    break;
    case OP_CIRCLE:
      // Создаем объект-круг
```

```
            pShape=new CBasePoint(point.x, point.y, 1000);
            // Черная линия шириной 2 мм
            pShape->SetPen(RGB(0,0,0), 200, PS_GEOMETRIC);
            // Темно-серая заливка
            pShape->SetBrush(RGB(100,100,100));
         break;
         case OP_SQUARE:
            // Создаем объект-квадрат
            pShape=new CSquare(point.x, point.y, 2000);
            // Красная линия шириной 1 мм
            pShape->SetPen(RGB(200,0,0), 100, PS_GEOMETRIC);
            // Темно-серая диагональная штриховка
            pShape->SetBrush(RGB(100,100,100),0,HS_DIAGCROSS);
            break;
      }
      if(pShape!=NULL) // Создали фигуру
      {
         // Добавляем в конец списка
         pDoc->m_ShapesList.AddTail(pShape);
         // Указываем, что окно надо перерисовать
         Invalidate();
         // Указываем, что документ изменен
         pDoc->SetModifiedFlag();
      }
   }
```

Функция `AddShape()` создает фигуру и добавляет ее в список объектов документа. Для того чтобы созданные объекты-фигуры были нарисованы на экране, модифицируем функцию `OnDraw()` так, как показано в листинге 4.21.

Листинг 4.21. Функция `OnDraw()`. Файл PainterView.cpp

```
void CPainterView::OnDraw(CDC* pDC)
{
   CPainterDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);
   // TODO: add draw code for native data here
   // Поставим перо позаметнее
   CPen Pen(PS_GEOMETRIC, 100, RGB(0,0,0));
   CPen *pOldPen=pDC->SelectObject(&Pen);
   // Если имеются опорные точки
```

```
    if(pDoc->m_nIndex>0)
       // Поместим перо в первую из них
       pDC->MoveTo(pDoc->m_Points[0]);
    // Пока не закончатся опорные точки, будем их соединять
    for(int i=1; i<pDoc->m_nIndex; i++)
       pDC->LineTo(pDoc->m_Points[i]);
    // Восстановим старое перо
    pDC->SelectObject(pOldPen);

    // Выводим все фигуры, хранящиеся в списке
    POSITION pos=NULL;
    CBasePoint* pShape=NULL;
    // Если в списке есть объекты
    if(pDoc->m_ShapesList.GetCount()>0)
       // Получим позицию первого объекта
       pos=pDoc->m_ShapesList.GetHeadPosition();
    while(pos!=NULL)
    {
       // Получим указатель на первый объект
       pShape=pDoc->m_ShapesList.GetNext(pos);
       // Нарисуем объект
       if(pShape!=NULL) pShape->Show(pDC);
    }
}
```

В функцию `OnDraw()` добавлен код, который вызывает метод `Show()` для всех объектов из списка. Заметьте, в списке у нас хранятся указатели на объекты класса `CBasePoint`, однако функция `Show()` рисует кружочки и квадратики. Такое замечательное явление называется *полиморфизмом* и объясняется виртуальностью функции `Show()`.

В принципе, уже можно компилировать программу и наслаждаться рисованием. Однако у нас остался нерешенным еще один важный вопрос — сохранение рисунков.

4.3.5. Сохранение рисунков

За сохранение данных в программе Painter отвечает объект-документ. Метод `CPainterDoc::Serialize()` осуществляет эту важную процедуру. Возможно, кто-то удивится, узнав, как мало нам понадобится сделать, чтобы научить программу Painter сохранять рисунки с кружочками, квадратами и прочими фигурами — добавить всего одну строку `m_ShapesList.Serialize(ar)` (листинг 4.22).

Листинг 4.22. Метод `CPainterDoc::Serialize()`. Файл PainterDoc.cpp

```
void CPainterDoc::Serialize(CArchive& ar)
{
   CString version;
   WORD i=0, version_n=0;
   if (ar.IsStoring()) // Сохраняем
   {
       // версию формата наших рисунков
       version_n=2;
       version.Format("pr%d", version_n);
       ar << version;
       // Количество точек
       ar << m_nIndex;
       // Значения координат точек
       for(int i=0; i<m_nIndex; i++) ar << m_Points[i];

       // Режим отображения
       ar << m_wMap_Mode;
       // Размер листа
       ar << m_wSheet_Width;
       ar << m_wSheet_Height;
   }
   else // Загружаем
   {
       // версию формата
       ar >> version;
       version_n=atoi((LPCTSTR)version.Right(1));
       switch(version_n) // в зависимости от версии формата
       {
           case 2:
              // Количество точек
              ar >> m_nIndex;
              // Загружаем значения координат точек
              for(i=0; i<m_nIndex; i++) ar >> m_Points[i];
              // Режим отображения
              ar >> m_wMap_Mode;
              // Размер листа
              ar >> m_wSheet_Width;
              ar >> m_wSheet_Height;
              break;
           default:
```

```
            AfxMessageBox("Неизвестный формат", MB_OK);
            return;
        }//switch(version)
    }
    // Выполняем сериализацию списка фигур
    m_ShapesList.Serialize(ar);
}
```

Помимо сохранения списка фигур, в код метода `Serialize()` также добавлена сериализация параметров листа и режима отображения. Кроме того, в этой версии функции первым параметром мы сохраняем/загружаем версию формата нашего рисунка. Дело в том, что при прямом вызове метода `Serialize()`, как это происходит для объекта-документа, автоматическая проверка версий форматов файлов не производится. Поэтому подход, который мы использовали в методе `CBasePoint::Serialize()` для проверки версий в этом случае не подойдет. Однако метод `CPainterDoc::Serialize()` может выдать сообщение о неправильном формате данных в случае, если мы попытаемся прочитать из архива данные, которых там быть не должно. Поэтому мы поступаем просто и прямо:

- при сохранении сначала записываем идентификатор нашего формата, а затем данные;
- при загрузке сначала читаем идентификатор, и в зависимости от его значения выбираем схему загрузки.
- Идентификатором формата может быть любое значение, например версия программы (если в каждой версии программы есть изменения в формате файлов). Если мы захотим в дальнейшем, чтобы наша программа правильно читала файлы разных форматов, то должны предусмотреть правильную загрузку данных, обусловленную идентификатором формата. Для того чтобы отличать файлы рисунков, созданные в программе Painter версии 2, введем новое расширение pr2.
- Надо отметить, что использование единственного числового значения в качестве идентификатора формата — не очень хороший подход. Дело в том, что первым прочитанным значением может оказаться число, которое вовсе не обозначает формат, а это повлечет за собой проблемы. Используем более сложный идентификатор, случайное появление которого менее вероятно. В качестве идентификатора будем записывать строку "pr#", где символ # заменим в дальнейшем номером формата. При чтении мы извлекаем из строки номер формата и выполняем соответствующий вариант загрузки.
- Для того чтобы автоматизировать присвоение расширения к имени файла, при сохранении можно использовать строковый ресурс с идентификатором `IDR_MAINFRAME`. Откроем этот ресурс и отредактируем его так, как показано на рис. 4.15.

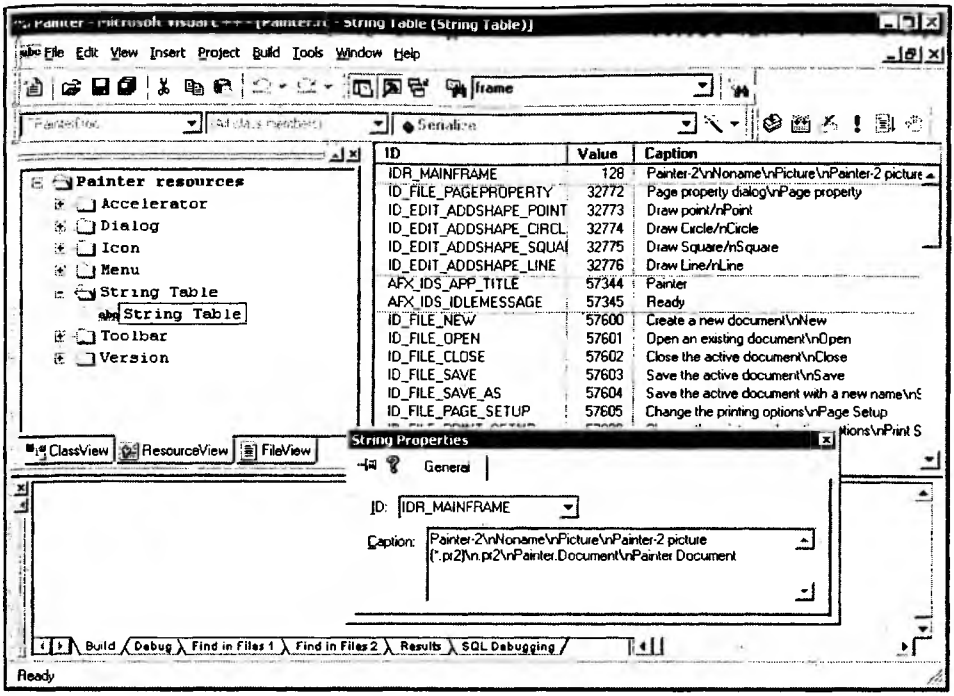


Рис. 4.15. Редактирование строкового ресурса

В строке

```
Painter-2\nNoname\nPicture\nPainter-2 picture (*.pr2)\n.pr2\
nPainter.Document\nPainter Document
```

подстроки, разделенные символом `\n`, имеют следующий смысл:

- `Painter-2` — имя, являющееся частью заголовка программы с SDI-интерфейсом;
- `Noname` — имя, присваиваемое по умолчанию новому документу (рисунку);
- `Picture` — название типа документа (имеет смысл в программах с MDI-интерфейсом);
- `Painter-2 picture (*.pr2)` — название типа файла, оно будет появляться в диалоговом окне **Открыть**;
- `.pr2` — фильтр для отбора файлов по расширению в диалоговом окне **Открыть**;
- `Painter.Document` — идентификатор типа документа для хранения в реестре Windows, он используется для регистрации диспетчером файлов Windows;
- `Painter Document` — название типа документа для хранения в реестре Windows.

4.3.6. Очистка памяти

Ну вот, мы создали в нашей программе достаточно много объектов. Надо не забыть своевременно их убрать. Для этого введем в класс `CPainterDoc` специальный метод `ClearShapesList()` (листинг 4.23).

Листинг 4.23. Метод `ClearShapesList()`. Файл PainterDoc.cpp

```
void CPainterDoc::ClearShapesList()
{
   // Очистили список объектов
   POSITION pos=NULL;
   // Пока в списке есть фигуры
   while(m_ShapesList.GetCount()>0)
      // Удаляем первую из них
      delete m_ShapesList.RemoveHead();
}
```

Будем вызывать `ClearShapesList()` в тех местах, где нам может потребоваться очистка списка, например, в деструкторе объекта-документа и в функции создания нового документа (листинг 4.24).

Листинг 4.24. Методы, в которых очищается список объектов. Файл PainterDoc.cpp

```
CPainterDoc::~CPainterDoc()
{
   // Очистили список фигур
   ClearShapesList();
}

BOOL CPainterDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
   return FALSE;

   // TODO: add reinitialization code here
   // (SDI documents will reuse this document)

   // Сбросили счетчик
   m_nIndex=0;
   // Очистили список фигур
   ClearShapesList();
```

```
    // Перерисовали
    UpdateAllViews(NULL);

    return TRUE;
}
```

4.4. Заключение

Кажется, наконец, все. Теперь приступим к рисованию. Созданный шедевр можно увидеть на рис. 4.16, а также найти на компакт-диске в каталоге Pics\Painter файл Грузовик.pr2. Текст программы приведен на диске в каталоге \Sources\Painter2.

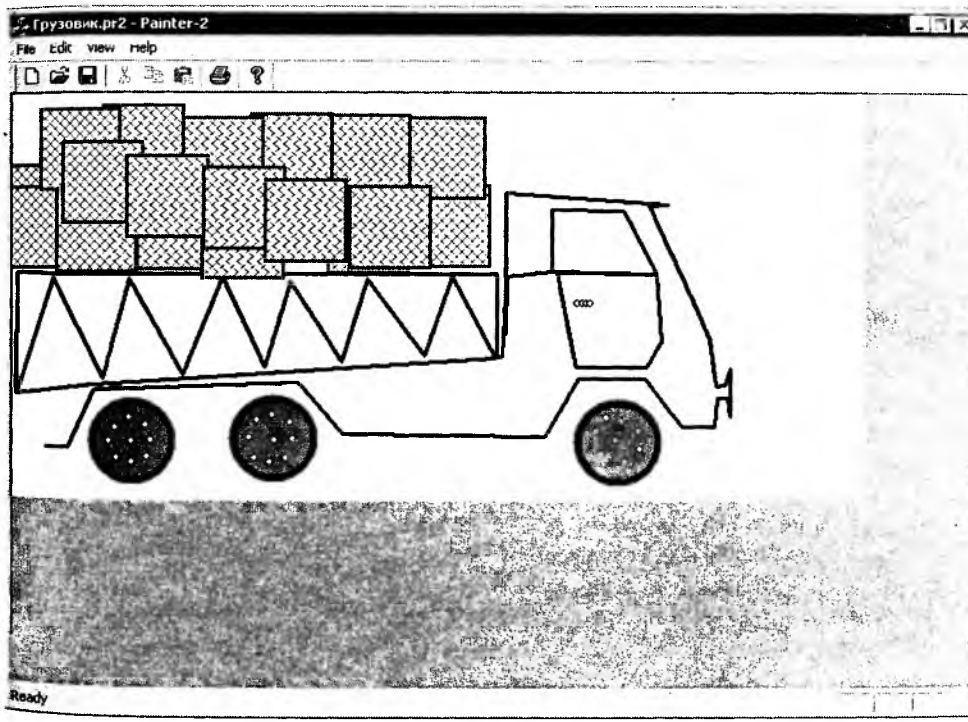


Рис. 4.16. Бессмертное творение в Painter 2

В следующих главах мы несколько расширим изобразительные возможности программы Painter и добавим функции редактирования рисунков.

Дополнительные сведения по работе с одно- и многодокументными приложениями можно найти в книге [4].

Глава 5

Математический аппарат алгоритмов компьютерной графики

В данной главе рассматриваются:

- некоторые определения аналитической геометрии;
- математический аппарат преобразований на плоскости.

Существенным недостатком текущей версии программы Painter является то, что она не позволяет редактировать рисунки. В настоящий момент мы можем только добавлять объекты в рисунок, но не имеем возможности открыть ранее созданный рисунок и изменить расположение объектов, их форму и цвет, порядок, в котором фигуры накладываются друг на друга, удалить лишние детали. Для того чтобы добавить в программу некоторые средства редактирования, нам потребуются несложные математические преобразования, рассмотрению которых и посвящается данная глава.

5.1. Векторы

Вектор — направленный отрезок прямой.

Ниже будут использованы обозначения:

- P, Q — концевые точки отрезка;
- $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ — векторы;
- $\mathbf{0}$ — вектор с нулевой длиной;
- $-\boldsymbol{a}$ — вектор длиной $|\boldsymbol{a}|$, направленный в сторону, противоположную $\boldsymbol{a}$;
- p, m — вещественные числа;
- $|\boldsymbol{a}|$ — длина вектора, равная расстоянию между концевыми точками.

5.1.1. Свойства векторов

- При параллельном переносе вектор не изменяется (рис. 5.1).
- Сумма векторов тоже является вектором: $\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{c}$ (рис. 5.2).
- Произведение $p\boldsymbol{a}$ — вектор длиной, равной $|p||\boldsymbol{a}|$, если $p = 0$ или $\boldsymbol{a} = \boldsymbol{0}$, то $p\boldsymbol{a} = \boldsymbol{0}$ (рис. 5.3);
  - если $p > 0$, результирующий вектор совпадает по направлению с $\boldsymbol{a}$;
  - если $p < 0$, результирующий вектор имеет направление, противоположное $\boldsymbol{a}$.

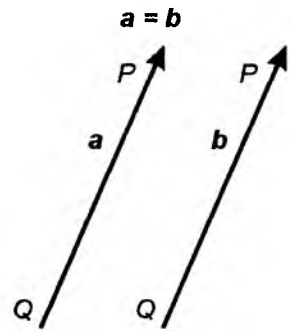


Рис. 5.1. Параллельный перенос векторов

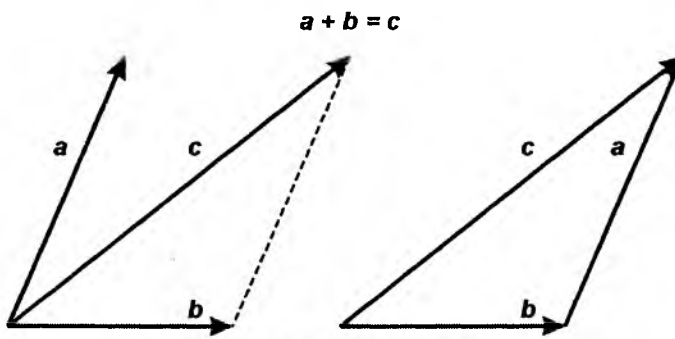


Рис. 5.2. Сумма векторов

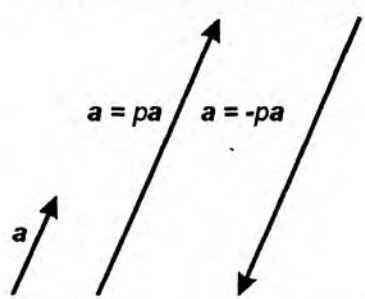


Рис. 5.3. Произведение векторов

Для векторов выполняются следующие правила:

- $\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{b} + \boldsymbol{a}$;
- $(\boldsymbol{a} + \boldsymbol{b}) + \boldsymbol{c} = \boldsymbol{a} + (\boldsymbol{b} + \boldsymbol{c})$;
- $\boldsymbol{a} + \boldsymbol{0} = \boldsymbol{a}$;
- $\boldsymbol{a} + (-\boldsymbol{a}) = \boldsymbol{0}$;
- $p(\boldsymbol{a} + \boldsymbol{b}) = p\boldsymbol{a} + p\boldsymbol{b}$;
- $(p + m)\boldsymbol{a} = p\boldsymbol{a} + m\boldsymbol{a}$;
- $1\boldsymbol{a} = \boldsymbol{a}$;
- $0\boldsymbol{a} = \boldsymbol{0}$.

В прямоугольной системе координат направление осей задается тройкой перпендикулярных единичных векторов.

Система координат называется *правой*, если при повороте от вектора $\boldsymbol{i}$ к вектору $\boldsymbol{j}$ на 90° направление вектора $\boldsymbol{k}$ совпадает с поступательным движением винта с правой резьбой (рис. 5.4). Начальная точка векторов обозначается буквой *O*.

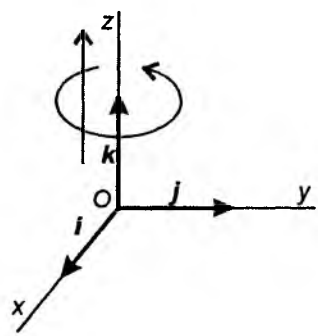


Рис. 5.4. Правая система координат

Любой вектор $\boldsymbol{V}$ может быть записан в виде линейной комбинации $\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$: $\boldsymbol{V} = x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$, где x, y, z — координаты конечной точки P вектора $\boldsymbol{V} = \boldsymbol{OP}$. Вектор $\boldsymbol{V}$ можно записать также в матричном виде

$$\boldsymbol{V} = [x, y, z] \text{ или } \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

5.1.2. Скалярное произведение векторов

Скалярное произведение векторов $\boldsymbol{a}$ и $\boldsymbol{b}$ обозначается $\boldsymbol{a} \cdot \boldsymbol{b}$ и определяется так:

$$\boldsymbol{a} \cdot \boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}| \cos \gamma, \qquad (5.1)$$

где γ — угол между $\boldsymbol{a}$ и $\boldsymbol{b}$. Скалярное произведение — это число $p = \boldsymbol{a} \cdot \boldsymbol{b}$. Применяя выражение (5.1) к единичным векторам $\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$, находим:

$$\boldsymbol{i} \cdot \boldsymbol{i} = \boldsymbol{j} \cdot \boldsymbol{j} = \boldsymbol{k} \cdot \boldsymbol{k} = 1; \quad \boldsymbol{i} \cdot \boldsymbol{j} = \boldsymbol{j} \cdot \boldsymbol{i} = \boldsymbol{j} \cdot \boldsymbol{k} = \boldsymbol{k} \cdot \boldsymbol{j} = \boldsymbol{k} \cdot \boldsymbol{i} = \boldsymbol{i} \cdot \boldsymbol{k} = 0. \tag{5.2}$$

Свойства скалярного произведения:

- $p(m\boldsymbol{a} \cdot \boldsymbol{b}) = pm(\boldsymbol{a} \cdot \boldsymbol{b})$;
- $(p\boldsymbol{a} + m\boldsymbol{b}) \cdot \boldsymbol{c} = p\boldsymbol{a} \cdot \boldsymbol{c} + m\boldsymbol{b} \cdot \boldsymbol{c}$;
- $\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{b} \cdot \boldsymbol{a}$;
- $\boldsymbol{a} \cdot \boldsymbol{a} = \boldsymbol{0}$, если $\boldsymbol{a} = 0$.

Скалярное произведение векторов $\boldsymbol{a} = [a_1, a_2, a_3]$ и $\boldsymbol{b} = [b_1, b_2, b_3]$:

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 b_1 + a_2 b_2 + a_3 b_3. \tag{5.3}$$

Соотношение (5.3) следует из $\boldsymbol{a} \cdot \boldsymbol{b} = (a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}) \cdot (b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k})$ с учетом свойств скалярного произведения и соотношения (5.2).

5.1.3. Векторное произведение векторов

Векторное произведение векторов $\boldsymbol{a}$ и $\boldsymbol{b}$ обозначается $\boldsymbol{a} \times \boldsymbol{b}$. Результатом векторного произведения является вектор: $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$.

Свойства векторного произведения:

- Если $\boldsymbol{a} = p\boldsymbol{b}$, где p — скаляр, то $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b} = 0$, иначе длина вектора $\boldsymbol{c}$ равна:

$$|\boldsymbol{c}| = |\boldsymbol{a}||\boldsymbol{b}| \sin \gamma, \tag{5.4}$$

где γ — угол между векторами $\boldsymbol{a}$ и $\boldsymbol{b}$. Направление вектора $\boldsymbol{c}$ перпендикулярно $\boldsymbol{a}$ и $\boldsymbol{b}$ и таково, что $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ именно в таком порядке образуют правостороннюю тройку. Это означает, что если $\boldsymbol{a}$ поворачивается на угол меньше 180° по направлению к вектору $\boldsymbol{b}$, то вектор $\boldsymbol{c}$ имеет направление, совпадающее с направлением поступательного движения винта с правой нарезкой при таком повороте.

- Пусть p — некоторая константа, тогда справедливы соотношения:

  $(p\boldsymbol{a}) \times \boldsymbol{b} = p(\boldsymbol{a} \times \boldsymbol{b})$;

  $\boldsymbol{a} \times (\boldsymbol{b} + \boldsymbol{c}) = \boldsymbol{a} \times \boldsymbol{b} + \boldsymbol{a} \times \boldsymbol{c}$;

  $\boldsymbol{a} \times \boldsymbol{b} = -\boldsymbol{b} \times \boldsymbol{a}$,

  в общем случае $\boldsymbol{a} \times (\boldsymbol{b} \times \boldsymbol{c}) \neq (\boldsymbol{a} \times \boldsymbol{b}) \times \boldsymbol{c}$.

- Для правой ортогональной системы координат, определяемой векторами $\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$, справедливы соотношения: $\boldsymbol{i} \times \boldsymbol{i} = \boldsymbol{j} \times \boldsymbol{j} = \boldsymbol{k} \times \boldsymbol{k} = \boldsymbol{0}$; $\boldsymbol{i} \times \boldsymbol{j} = \boldsymbol{k}$; $\boldsymbol{j} \times \boldsymbol{k} = \boldsymbol{i}$; $\boldsymbol{k} \times \boldsymbol{i} = \boldsymbol{j}$; $\boldsymbol{j} \times \boldsymbol{i} = -\boldsymbol{k}$; $\boldsymbol{k} \times \boldsymbol{j} = -\boldsymbol{i}$; $\boldsymbol{i} \times \boldsymbol{k} = -\boldsymbol{j}$. Учитывая эти соотношения для векторного произведения, имеем:

  $\boldsymbol{a} \times \boldsymbol{b} = (a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}) \times (b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k})$,

отсюда получаем:

$$\boldsymbol{a} \times \boldsymbol{b} = (a_2b_3 - a_3b_2)\boldsymbol{i} + (a_3b_1 - a_1b_3)\boldsymbol{j} + (a_1b_2 - a_2b_1)\boldsymbol{k}. \quad (5.5)$$

Правая часть уравнения (5.5) является выражением детерминанта третьего порядка (5.11). Отсюда выражение (5.5) может быть записано в матричной форме

$$\boldsymbol{a}\times\boldsymbol{b} = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}. \quad (5.6)$$

5.2. Детерминанты

Рассмотрим систему уравнений:

$$\begin{cases} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2 \end{cases}. \quad (5.7)$$

Чтобы решить систему (5.7), умножим первое уравнение на b_2, а второе — на $-b_1$ и сложим, получим: $(a_1b_2 - a_2b_1)x = b_2c_1 - b_1c_2$. Затем первое уравнение умножим на $-a_2$, а второе — на a_1 и сложим, в результате получим:

$$(a_1b_2 - a_2b_1)y = a_1c_2 - a_2c_1,$$

если $a_1b_2 - a_2b_1 \neq 0$, то:

$$x = \frac{b_2c_1 - b_1c_2}{a_1b_2 - a_2b_1}; \qquad y = \frac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}. \quad (5.8)$$

Выражение в делителе может быть записано:

$$D = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1b_2 - a_2b_1. \quad (5.9)$$

Выражение (5.9) называется *детерминантом* второго порядка.

С помощью детерминантов уравнение (5.7) может быть записано в виде

$$x = \frac{D_1}{D}, \quad y = \frac{D_2}{D}, \text{ при } D \neq 0, \quad (5.10)$$

где

$$D_1 = \begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}, \quad D_2 = \begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}.$$

D_i (i = 1, 2) получается заменой i-го столбца на правую часть системы (5.7). Такой способ пригоден для решения систем двух и более уравнений и называется "правилом Крамера".

Детерминант третьего порядка имеет вид

$$D=\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = a_1\begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix} - a_2\begin{vmatrix} b_1 & c_1 \\ b_3 & c_3 \end{vmatrix} + a_3\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}. \quad (5.11)$$

Аналогично записываются детерминанты более высоких порядков.

5.2.1. Свойства детерминантов

Рассмотрим основные свойства детерминантов.

- При транспонировании матрицы (если строки записать в столбцы) значение детерминанта не изменяется:

$$\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}.$$

- Перемена мест двух строк (столбцов) меняет знак детерминанта:

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = -\begin{vmatrix} a_1 & b_1 & c_1 \\ a_3 & b_3 & c_3 \\ a_2 & b_2 & c_2 \end{vmatrix}.$$

- Если любую строку (столбец) умножить на число, то значение детерминанта умножится на это же число:

$$\begin{vmatrix} ka_1 & kb_1 \\ a_2 & b_2 \end{vmatrix} = k\begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}.$$

- Если строка (столбец) изменяется путем добавления соответствующих элементов другой строки (столбца), умноженных на константу, то значение детерминанта не изменится:

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 + ka_1 & b_3 + kb_1 & c_3 + kc_1 \end{vmatrix} = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}.$$

- Если строка (столбец) является линейной комбинацией других строк (столбцов), то значение детерминанта равно нулю:

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ 3a_1 + ka_2 & 3b_1 + kb_2 & 3c_1 + kc_2 \end{vmatrix} = 0.$$

Использование детерминантов позволяет в удобной форме описывать разные геометрические объекты. Например, уравнение прямой в двумерном

пространстве (R^2), проходящей через точки $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, может быть записано следующим образом:

$$\begin{vmatrix} x & y & 1 \\ x_1 & y_1 & 1 \\ x_2 & x_2 & 1 \end{vmatrix} = 0. \tag{5.12}$$

Справедливость этой записи подтверждается следующим рассуждением: если, например, $x = x_1$, $y = y_1$, то первая строка является линейной комбинацией, следовательно, $D = 0$.

Плоскость в трехмерном пространстве (R^3), проходящая через точки $P_1(x_1, y_1, z_1)$, $P_2(x_2, y_2, z_2)$, $P_3(x_3, y_3, z_3)$ может быть описана следующим образом:

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = 0. \tag{5.13}$$

Свойства детерминантов могут быть использованы, например, при определении положения некоторой точки P_3 относительно прямой $P_1 P_2$ на плоскости. Запишем координаты этих точек в матричном виде (5.12) в порядке $P_1 P_3 P_2$. Если знак детерминанта такой матрицы будет положительным, точка P_3 лежит справа относительно прямой $P_1 P_2$ (если "смотреть" из точки P_1), отрицательным — слева.

5.3. Однородные координаты

Уравнение $aX + bY + c = 0$ описывает прямую в пространстве R^2. Заменим X на x/w, Y на y/w, получим уравнение $a(x/w) + b(y/w) + c = 0$. Запишем его в такой форме:

$$ax + by + cw = 0. \tag{5.14}$$

Уравнения типа (5.14) называют однородными, т. к. они имеют одинаковую структуру в терминах ax, by, cw — отсюда x, y, w называются однородными координатами точки (X, Y).

Если $w = 1$ (двумерное пространство располагается в плоскости $w = 1$ в системе x, y, w), то уравнение (5.14) описывает плоскость, проходящую через начало координат, и заданную прямую линию.

Если считать, что (x, y, w) — это иная форма записи $(x/w, y/w)$, то тогда w не должно быть равно нулю. Однако некоторые полезные свойства однородных координат проявляются именно при отсутствии такого требования.

Рассмотрим систему:

$$\begin{cases} 2x + 3y - 6 = 0, \\ 4x + 6y - 24 = 0. \end{cases} \quad (5.15)$$

Система (5.15) задает две параллельные линии и не имеет решения.

При замене координат на однородные, система (5.15) преобразуется к виду

$$\begin{cases} 2x + 3y - 6w = 0, \\ 4x + 6y - 24w = 0. \end{cases} \quad (5.16)$$

Система (5.16) имеет, по крайней мере, одно решение ($x = 0$, $y = 0$, $w = 0$).

Система

$$\begin{cases} 2x + 3y = 0, \\ w = 0 \end{cases}$$

эквивалентна системе (5.16), следовательно, верно соотношение:

$$\frac{x}{y} = \frac{-2}{3}. \quad (5.17)$$

Таким образом, решение системы (5.16) состоит из всех точек ($3k$, $-2k$, 0), где k — любое число. В пространстве x, y, w эти точки образуют прямую, проходящую через точки O (0, 0, 0) и (3, −2, 0). Данная линия бесконечно удалена, ее точки можно рассматривать как предельные точки ($3k$, $-2k$, w), при $w \to 0$.

5.4. Использование однородных координат

В обычных двумерных координатах линейное преобразование на плоскости может быть записано следующим образом:

$$[x'\ y'\ ] = [x\ y]\boldsymbol{A},$$

где $\boldsymbol{A}$ — матрица преобразования:

$$\boldsymbol{A} = \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix}.$$

Точки [1 0] и [0 1] с помощью матрицы $\boldsymbol{A}$ отображаются в точки [a_1 a_2] и [b_1 b_2]. Однако, независимо от вида матрицы $\boldsymbol{A}$, точка [0 0] отобразится в точку [0 0], поэтому таким способом нельзя выполнить операцию переноса точки в новую позицию. В однородных координатах точка в двумерном пространстве задается тройкой (x, y, w), и преобразование записывается в виде $[x'\ y'\ z'] = [x\ y\ z]\boldsymbol{A}$, где

$$\mathbf{A} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}.$$

В этом случае имеют место следующие преобразования:

$$[1\ 0\ 0]A = [a_1\ a_2\ a_3];$$
$$[0\ 1\ 0]A = [b_1\ b_2\ b_3];$$
$$[0\ 0\ 1]A = [c_1\ c_2\ c_3].$$

Это означает, что однородные координаты позволяют выразить любые преобразования путем матричного перемножения.

5.5. Преобразования на плоскости

Рассмотрим систему уравнений:

$$\begin{cases} x' = x + a, \\ y' = y. \end{cases} \tag{5.18}$$

Система (5.18) может означать:

- перемещение всех точек в плоскости x — y вправо на расстояние a (рис. 5.5);
- смещение координатных осей влево на расстояние a (рис. 5.6).

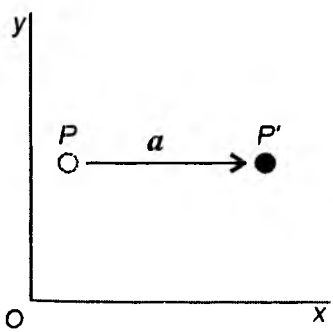


Рис. 5.5. Перенос точки

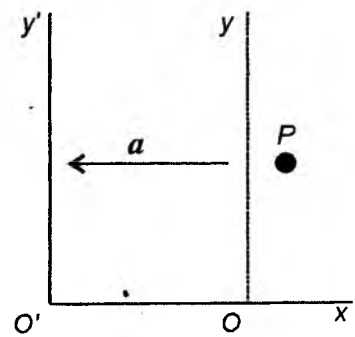


Рис. 5.6. Перенос системы координат

Аналогично и в более сложных ситуациях одно и то же преобразование можно рассматривать как изменение координат точки либо как изменение самой системы координат.

Общий случай операции переноса:

$$\begin{cases} x' = x + a; \\ y' = y + b. \end{cases} \tag{5.19}$$

Рассмотрим далее операцию поворота точки $P(x, y)$ вокруг начала координат O на угол φ в точку $P'(x', y')$ (рис. 5.7). Новые координаты точки рассчитываются с помощью системы уравнений:

$$\begin{cases} x' = ax + by, \\ y' = cx + dy. \end{cases} \tag{5.20}$$

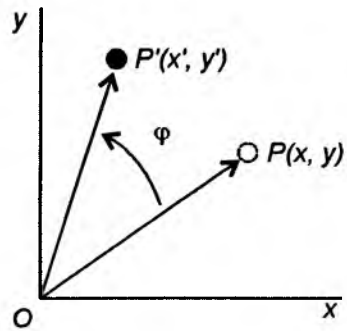


Рис. 5.7. Поворот точки $P(x, y)$ вокруг начала координат O на угол φ в точку $P'(x', y')$

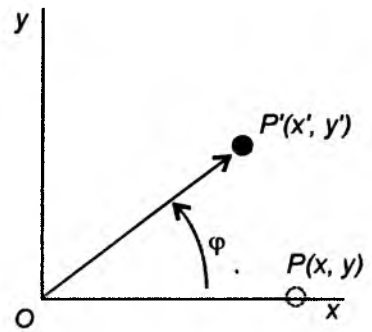


Рис. 5.8. Поворот точки $P(x, y)$ вокруг начала координат O на угол φ в точку $P'(x', y')$, если $P = (1, 0)$

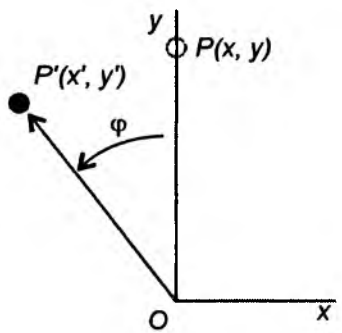


Рис. 5.9. Поворот точки $P(x, y)$ вокруг начала координат O на угол φ в точку $P'(x', y')$, если $P = (0, 1)$

Рассмотрим рис. 5.8. В случае если $P = (1, 0)$, то $x' = a$, $y' = c$. Из рис. 5.8 найдем: $a = \cos\varphi$, $c = \sin\varphi$.

Аналогично, если $P = (0, 1)$: $b = -\sin\varphi$, $d = \cos\varphi$ (рис. 5.9).

Таким образом, поворот вокруг начала координат O можно выразить в виде

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi, \\ y' = x\sin\varphi + y\cos\varphi. \end{cases} \tag{5.21}$$

часто требуется выполнить поворот вокруг какой-то произвольной точки (x_0, y_0). Схема выполнения такого преобразования следующая:

1. Точка начала координат O переносится в точку с координатами (x_0, y_0). Для выполнения этой операции достаточно отнять от координат точки $P(x, y)$ координаты точки (x_0, y_0).

2. Выполняется поворот вокруг новой точки начала координат согласно формулам:

$$\begin{cases} x' = (x - x_0)\cos\varphi - (y - y_0)\sin\varphi, \\ y' = (x - x_0)\sin\varphi + (y - y_0)\cos\varphi. \end{cases} \quad (5.22)$$

3. Возвращение системы координат в первоначальный вид. Прибавляем к координатам точки $P'(x', y')$ значения координат точки (x_0, y_0):

$$\begin{cases} x' = x' + x_0, \\ y' = y' + y_0. \end{cases} \quad (5.23)$$

Таким образом, поворот на угол φ вокруг произвольной точки (x_0, y_0) можно записать следующим образом:

$$\begin{cases} x' = x_0 + (x - x_0)\cos\varphi - (y - y_0)\sin\varphi, \\ y' = y_0 + (x - x_0)\sin\varphi + (y - y_0)\cos\varphi. \end{cases} \quad (5.24)$$

5.6. Матричная форма записи двумерных преобразований

Запись операции переноса в однородных координатах:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}.$$

Запись операции поворота на угол φ вокруг точки O в однородных координатах:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Поворот на угол φ вокруг точки (x_0, y_0):

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\boldsymbol{R},$$

где $\boldsymbol{R}$ — некоторая матрица 3×3.

Для нахождения $\boldsymbol{R}$ выполним следующие шаги:

1. Перенос точки (x_0, y_0) в начало координат — точку O:

$$[u_1 \quad v_1 \quad 1] = [x \quad y \quad 1]\boldsymbol{T}', \text{ где } \boldsymbol{T}' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{bmatrix}.$$

2. Поворот на угол φ относительно O:

$$[u_2 \quad v_2 \quad 1] = [u_1 \quad v_1 \quad 1]\boldsymbol{R}_0, \text{ где } \boldsymbol{R}_0 = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

3. Перенос из начала координат в точку (x_0, y_0):

$$[x' \quad y' \quad 1] = [u_2 \quad v_2 \quad 1]\boldsymbol{T}, \text{ где } \boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_0 & y_0 & 1 \end{bmatrix}.$$

Учитывая ассоциативность матричного умножения, т. е. $(\boldsymbol{AB})\boldsymbol{C} = \boldsymbol{A}(\boldsymbol{BC}) = \boldsymbol{ABC}$, найдем:

$$[x' \quad y' \quad 1] = [u_2 \quad v_2 \quad 1]\boldsymbol{T} = \{[u_1 \quad v_1 \quad 1]\boldsymbol{R}_0\}\boldsymbol{T} = \{[x \quad y \quad 1]\boldsymbol{T}'\}\boldsymbol{R}_0\,\boldsymbol{T} =$$
$$= [x \quad y \quad 1]\boldsymbol{T}'\boldsymbol{R}_0\,\boldsymbol{T},$$

следовательно

$$\boldsymbol{R} = \boldsymbol{T}'\boldsymbol{R}_0\boldsymbol{T} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ c_1 & c_2 & 1 \end{bmatrix};$$

где

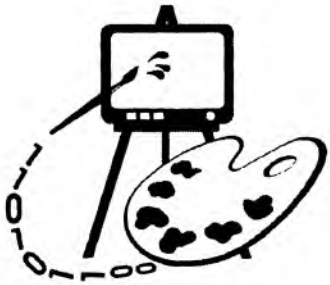
$$c_1 = x_0 - x_0\cos\varphi + y_0\sin\varphi,$$
$$c_2 = y_0 - x_0\sin\varphi - y_0\cos\varphi.$$

5.7. Заключение

В этой главе мы рассмотрели достаточный объем теоретических сведений? для того чтобы реализовать в программе Painter преобразования переноса и поворота на плоскости.

Дополнительно можно изучить литературу [2, 16].

Глава 6



Реализация функций редактирования рисунков

В данной главе рассматриваются:

- программная реализация функций редактирования рисунков (программа Painter 3);
- операция выбора фигуры;
- рисование полигональных фигур;
- рисование фигур произвольных размеров;
- рисование инверсным цветом;
- реализация преобразований на плоскости;
- определение реакций на нажатие клавиш;
- изменение порядка наложения фигур;
- удаление фигур;
- преобразование формата.

6.1. Выбор фигуры

Большинство операций редактирования начинаются с команды выбора фигур. Со стороны пользователя это выглядит примерно следующим образом:

1. Пользователь инициирует операцию выбора.
2. Пользователь указывает на фигуру.
3. Выбранная фигура каким-то образом помечается.
4. Пользователь выполняет преобразование фигуры.

В программе же все будет происходить, например, так:

1. Программа переходит в режим выбора. Ожидается щелчок мыши.

2. Обрабатывается щелчок мыши, проверяется, в какую фигуру попал пользователь.
3. В случае если есть попадание, выбранная фигура запоминается и помечается на рисунке.
4. Поступающие от пользователя команды преобразования применяются к выбранной фигуре.

Для индикации режима выбора определим в файле PainterView.h новый идентификатор операции:

```
#define OP_SELECT   10
```

В классе `CPainterDoc` (файл PainterDoc.h) заведем переменную, в которой будем сохранять адрес выбранного объекта-фигуры:

```
// Указатель на выбранную фигуру
CBasePoint* m_pSelShape;
```

А также добавим в классе `CPainterDoc` новый метод:

```
// Выбор активной фигуры
CBasePoint* SelectShape(CPoint point);
```

Реализация метода приведена ниже (листинг 6.1).

Листинг 6.1. Метод `CPainterDoc::SelectShape()`. Файл PainterDoc.cpp

```
CBasePoint* CPainterDoc::SelectShape(CPoint point)
{
   // Объект-область
   CRgn Rgn;
   // Указатель на элемент списка
   POSITION pos=NULL;
   // Начиная с "хвоста" списка
   if(m_ShapesList.GetCount()>0) pos=m_ShapesList.GetTailPosition();
   // Проверим, попадает ли точка point в какую-либо из фигур
   while(pos!=NULL)
   {
      m_pSelShape =m_ShapesList.GetPrev(pos);
      // Очистим объект-область
      Rgn.DeleteObject();
      m_pSelShape->GetRegion(Rgn);
      // Точка попадает в фигуру – возвращаем указатель на фигуру
      if(Rgn.PtInRegion(point)) return m_pSelShape;
   }
```

```
   // Если добрались до этого места, значит, не попали ни в какую фигуру
   return (m_pSelShape=NULL);
};
```

В качестве параметра в этот метод передаются координаты щелчка мыши. Далее для всех объектов списка вызывается метод `GetRegion()`, которому в качестве аргумента передается ссылка на объект класса `CRgn`. Каждый объект-фигура в своем методе `GetRegion()` конструирует область, соответствующую размеру и очертаниям фигуры на экране. Затем вызывается метод `PtInRegion()` класса `CRgn`, позволяющий определить, принадлежит ли точка области. Поиск происходит до тех пор, пока не найдется фигура, в которую попадает точка, или пока не кончится список объектов. Если произошло попадание в фигуру, то адрес объекта запоминается в переменной `m_pSelShape` и поиск прекращается. Если же точка не попадает ни в одну фигуру, переменной `m_pSelShape` присваивается `NULL`. Обратите внимание, поиск происходит с конца списка. Это связано с тем, что в методе `CPainterView::OnDraw()` объекты-фигуры выводятся на экран с начала списка, поэтому в случае перекрытия фигур "верхней" фигурой будет тот объект, который ближе к концу списка.

Для того чтобы пользователь мог перевести программу в режим выбора, добавим команду **Select** в меню **Edit**, а в класс `CPainterView` добавим функцию-обработчик этой команды (листинг 6.2).

Листинг 6.2. Функция-обработчик команды `Edit-Select`. Файл PainterView.cpp

```
void CPainterView::OnEditSelect()
{
   m_CurOper=OP_SELECT;
}
```

Все, что происходит в этой функции, — это присвоение переменной `m_CurOper` значения `OP_SELECT`. В конструкцию `switch-case` в методе-обработчике щелчка мыши добавим соответствующую реакцию (см. также листинг 6.8):

```
switch(m_CurOper)
{
...
   case OP_SELECT:
      pDoc->SelectShape(LogPoint)
      // Указываем, что окно надо перерисовать
      Invalidate();
   break;
}
```

Вот и все, что потребовалось для реализации операции выбора фигуры. По умолчанию последняя нарисованная фигура будет считаться выделенной, т. е. при добавлении объекта-фигуры в список фигур переменной `m_pSelShape` будем присваивать адрес этого объекта.

6.2. Маркировка активной фигуры

После того как пользователь сделал свой выбор, программа должна каким-то образом продемонстрировать свою готовность к дальнейшим действиям. Обычно выбранная фигура как-то помечается. Например, появляется рамка вокруг фигуры, или обводятся ее контуры. В нашей программе для выделения активной фигуры будем инвертировать цвета изображения в области, занимаемой фигурой. Для этого создадим специальный метод `MarkSelectedShape()` в классе `CPainterView` (листинг 6.3).

Листинг 6.3. Метод `CPainterView::MarkSelectedShape()`. Файл PainterView.cpp

```
void CPainterView::MarkSelectedShape(CDC *pDC){   CPainterDoc
*pDoc=GetDocument();
   CRgn Rgn;
   if(pDoc->m_pSelShape==NULL) return;
   pDoc->m_pSelShape->GetRegion(Rgn);
   // Пробуем получить прямоугольник, описывающий фигуру
   CRect Rect;
   int res=Rgn.GetRgnBox(&Rect);
   if(res!=ERROR && res!=NULLREGION)
   pDC->InvertRect(&Rect);
}
```

В этом методе сначала проверяется наличие активной фигуры. Затем определяется область, занимаемая фигурой. С помощью метода `GetRgnBox()` класса `CRgn` определяются координаты прямоугольника, охватывающего фигуру. Полученный прямоугольник передается в метод `InvertRect()` класса `CDC`, который инвертирует цвета указанной области экрана. Данный метод вызывается в функции `OnDraw()` в случае, если программа находится в режиме выбора:

```
// Выделяем активную фигуру
if(m_CurOper==OP_SELECT)   MarkSelectedShape(pDC);
```

Результат работы метода показан на рис. 6.1 и 6.2.

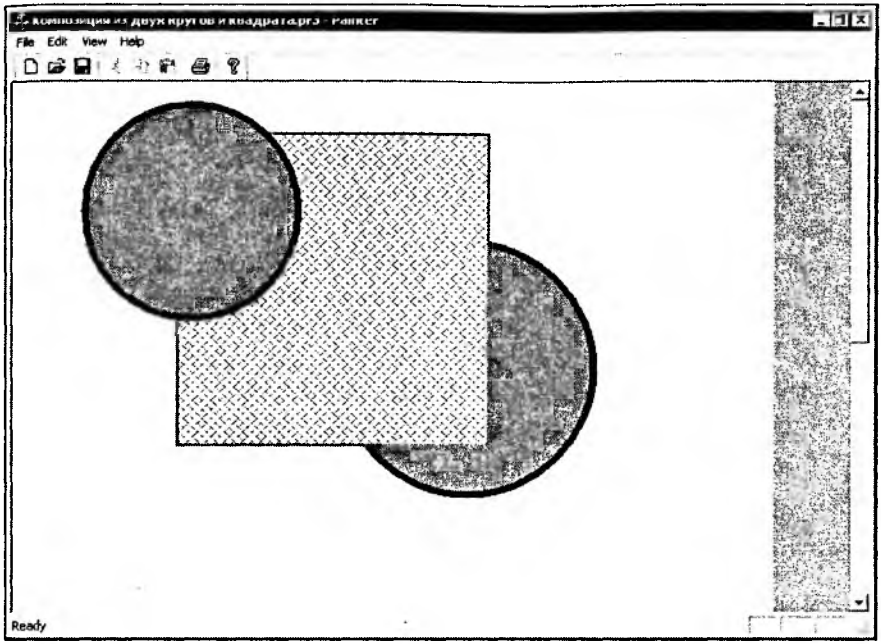


Рис. 6.1. Изображение до выполнения операции выбора

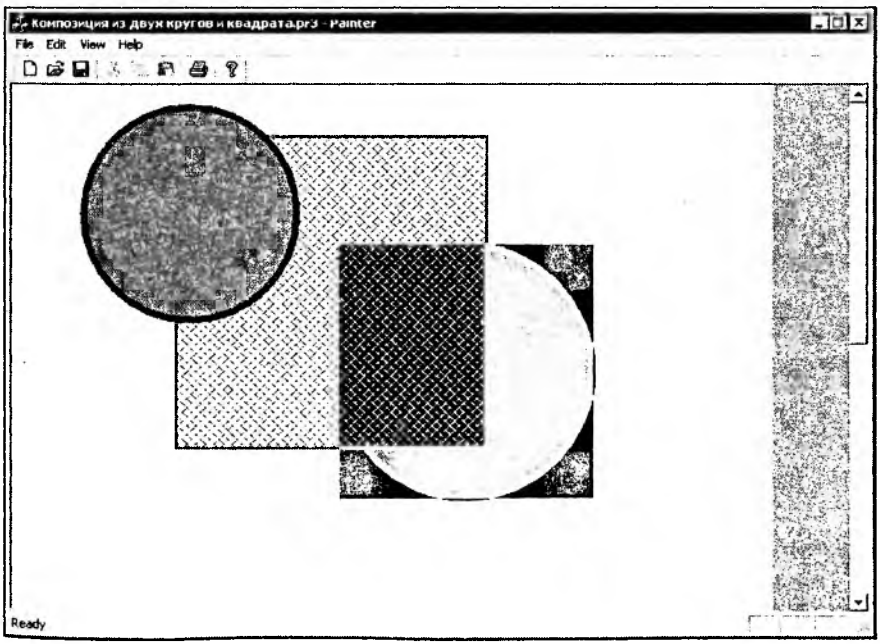


Рис. 6.2. Активная фигура на рисунке выделена инверсным цветом

6.3. Рисование полигональных фигур

Для рисования полигональных фигур заведем новый класс CPolygon, производный от CBasePoint. Особенностью этого класса является то, что в нем будут храниться координаты точек — вершин полигона. При этом координаты записаны в объектах класса CPoint, а сами объекты-точки хранятся в динамическом массиве — объекте класса CArray. Класс CArray принадлежит к группе классов MFC, обеспечивающих работу с *набором* (collection) элементов. Мы уже использовали ранее классы наборов CTypedPtrList и CObList для хранения указателей на объекты-фигуры в классе CPainterDoc. Класс CArray обеспечивает хранение элементов в виде массива. Замечательной особенностью такого массива является возможность динамического изменения его размера. Интерфейс класса CPolygon приведен в листинге 6.4.

Листинг 6.4. Интерфейс класса CPolygon. Файл Shapes.h

```
////////////////////////////////////////////////////
//Класс полигон
class CPolygon: public CBasePoint
{
   DECLARE_SERIAL(CPolygon)
// Режим рисования TRUE – заполненный полигон, FALSE – ломаная линия
   BOOL   m_bPolygon;
protected:
   // Метод сериализации
   void Serialize(CArchive& ar);
public:
   CArray <CPoint, CPoint> m_PointsArray;

   // Конструкторы
   CPolygon();
   ~CPolygon();
// Методы
   // Отображает фигуру на экране
   void Show(CDC *pDC);
   // Сообщает область захвата
   void GetRegion(CRgn &Rgn);
   // Устанавливает режим рисования полигона
   void SetPolygon(BOOL p) {m_bPolygon=p;};
   // Выполняет преобразование на плоскости
```

```
    void Transform(const CPoint &point0,
                          double ang, int a, int b);
};
```

Кроме массива точек в класс CPolygon введена переменная-флаг m_bPolygon типа BOOL, назначение которой — сигнализировать о том, надо ли заполнять полигон или нет. В зависимости от состояния этой переменной полигон будет рисоваться с помощью различных методов класса CDC. Таким образом, с помощью объектов класса CPolygon мы можем полностью заменить не очень "красивую" реализацию рисования ломаных в программе Painter версии 1. Реализация методов класса CPolygon приведена в листинге 6.5. Обратите внимание на функцию CPolygon::Serialize(): нам требуется обеспечить сохранение и загрузку лишь переменной m_bPolygon, сериализация же координат точек выполняется объектом-массивом — мы просто вызываем его метод Serialize().

Листинг 6.5. Реализация методов класса CPolygon. Файл Shapes.cpp

```
//////////////////////////////////////
// Реализация методов класса CPolygon

CPolygon::CPolygon(): CBasePoint()
{
   m_wSize=0;
   m_bPolygon=FALSE;
}

CPolygon::~CPolygon()
{
   m_PointsArray.RemoveAll( );
}

IMPLEMENT_SERIAL(CPolygon, CObject, 1)
void CPolygon::Serialize(CArchive &ar)
{
   if(ar.IsStoring()) // сохранение
   {
       // Сохраняем параметры объекта
       ar<<m_bPolygon;
   }
   else   // чтение
```

```
    {
        // Получили версию формата
        int Version=ar.GetObjectSchema();
        // В зависимости от версии
        // можно выполнить различные варианты загрузки
        // Загружаем параметры объекта
        ar>>m_bPolygon;
    }
    m_PointsArray.Serialize(ar);
    CBasePoint::Serialize(ar);
}
void CPolygon::Show(CDC* pDC)
{
    // Устанавливаем перо и кисть
    PrepareDC(pDC);
    // Рисуем
    if(m_bPolygon)
        pDC->Polygon(m_PointsArray.GetData(), m_PointsArray.GetSize());
    else
        pDC->Polyline( m_PointsArray.GetData(), m_PointsArray.GetSize());
    // Восстанавливаем контекст
    RestoreDC(pDC);
}

void CPolygon::GetRegion(CRgn &Rgn)
{
    Rgn. CreatePolygonRgn(m_PointsArray.GetData(),
m_PointsArray.GetSize(), ALTERNATE);
}

void CPolygon::Transform(const CPoint &point0, double ang, int a, int b)
{
    for(int i=0; i<m_PointsArray.GetSize(); i++)
        m_PointsArray[i]=::Transform(m_PointsArray[i],
        m_PointsArray[0], ang, a, b);
};
```

Для того чтобы пользователь мог инициировать операцию рисования полигона, в меню программы добавим команды **Edit | Add Shape | Polyline** и **Edit | Add Shape | Polygon**, а в класс `CPainterView` — обработчики этих команд

(листинг 6.6). Функции-обработчики отличаются лишь тем, что при создании заполненного полигона устанавливаются флаг m_bPolygon и цвет заливки.

Листинг 6.6. Обработчики команд создания фигур-полигонов

```
void CPainterView::OnEditAddshapePolyline()
{
   CBasePoint *pShape=new CPolygon;
   // Черная линия шириной 0,5 мм
   pShape->SetPen(RGB(0,0,0), 50, PS_GEOMETRIC);
   CPainterDoc *pDoc=GetDocument();
   // Добавляем в конец списка
   pDoc->m_ShapesList.AddTail(pShape);
   // Последняя фигура становится активной
   pDoc->m_pSelShape=pShape;
   // Указываем, что документ изменен
   pDoc->SetModifiedFlag();
   m_CurOper=OP_LINE;
}

void CPainterView::OnEditAddshapePolygon()
{
   CBasePoint *pShape=new CPolygon;
   // Темно-зеленая заливка
   pShape->SetBrush(RGB(0,100,0));
   // Черная линия шириной 0,5 мм
   pShape->SetPen(RGB(0,0,0), 50, PS_GEOMETRIC);
   // Так как pShape указатель на CBasePoint,
   // а метод SetPolygon() имеется только у класса CPolygon,
   // требуется преобразование типа указателя
   ((CPolygon*)pShape)->SetPolygon(TRUE);
   CPainterDoc *pDoc=GetDocument();
   // Добавляем в конец списка
   pDoc->m_ShapesList.AddTail(pShape);
   // Последняя фигура становится активной
   pDoc->m_pSelShape=pShape;
   // Указываем, что документ изменен
   pDoc->SetModifiedFlag();
   m_CurOper=OP_LINE;
}
```

В этих функциях динамически создаются и добавляются в список объектов-фигур объекты класса CPolygon и устанавливается текущая операция — рисование полигона (m_CurOper=OP_LINE). Точки-вершины будут добавляться в полигон при обработке нажатия левой кнопки мыши (см. листинг 6.8). Завершаться же операция рисования полигона будет двойным щелчком левой кнопки мыши. Для этого в класс CPainterView добавим обработчик сообщения WM_LBUTTONDBLCLK (листинг 6.7).

Листинг 6.7. Функция-обработчик сообщения WM_LBUTTONDBLCLK. Файл PainterView.cpp

```
void CPainterView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
   switch(m_CurOper)
   {
       case OP_LINE:
          m_CurOper=OP_NOOPER;
       break;
   }
   CScrollView::OnLButtonDblClk(nFlags, point);
}
```

6.4. Рисование фигур произвольных размеров

Введение класса CPolygon позволяет по-другому организовать обработку щелчка левой кнопкой мыши. В предыдущей версии программы вся обработка щелчка происходила в функции CPainterView::OnLButtonDown(), текст которой был приведен в листинге 4.19. Если текущей операцией являлось рисование прямых (case OP_LINE), то координаты точки сохранялись в статическом массиве объекта-документа, а в случае других операций в точке щелчка рисовалась фигура фиксированного размера. Изменим обработку нажатия щелчка мыши. Теперь будем различать события "нажата левая кнопка мыши" (WM_LBUTTONDOWN) и "отпущена левая кнопка мыши" (WM_LBUTTONUP). При нажатии мыши будем запоминать начальную точку операции. Если текущая операция "рисование полигона" (case OP_LINE), то нужно добавлять точку в массив вершин полигона. В других случаях больше никаких действий не предпринимать и ждать, когда пользователь отпустит мышь. Пользователь же сдвинет (перетащит) мышь в новую позицию при нажатой кнопке и только потом ее отпустит. Программа при обработке сообщения WM_LBUTTONUP запомнит конечную точку операции и в случае рисования фигур вызовет функцию CPainterView::AddShape() с начальной и конечной точками в качестве параметров. Функцию AddShape()

тоже немножко изменим. Как вы уже, наверное, догадались, теперь функция AddShape() будет устанавливать размер фигур в зависимости от значений координат начальной и конечной точек (листинг 6.8).

Листинг 6.8. Функции обработки сообщений мыши, обеспечивающие рисование фигур произвольных размеров. Файл PainterView.cpp

```
void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
   CPoint       LogPoint=point;
   // Получим контекст устройства, на котором рисуем
   CDC *pDC=GetDC();
   // Подготовим контекст устройства
   OnPrepareDC(pDC);
   // Переведем физические координаты точки в логические
   pDC->DPtoLP(&LogPoint);
   // Освободим контекст устройства
   ReleaseDC(pDC);
   // Запоминаем точку
   m_CurMovePoint=m_FirstPoint=LogPoint;
   switch(m_CurOper)
   {
       case OP_LINE:
       // Последним в списке должен быть полигон
       ((CPolygon*)pDoc->m_ShapesList.GetTail())↵
        ->m_PointsArray.Add(LogPoint);
       // Указываем, что окно надо перерисовать
       Invalidate();
       break;
   }
   // Даем возможность стандартному обработчику
   // тоже поработать над этим сообщением
   CScrollView::OnLButtonDown(nFlags, point);
}
void CPainterView::OnLButtonUp(UINT nFlags, CPoint point)
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
```

```
    CPoint      LogPoint=point;
    // Получим контекст устройства, на котором рисуем
    CDC *pDC=GetDC();
    // Подготовим контекст устройства метод базового класса
    OnPrepareDC(pDC);
    // Переведем физические координаты точки в логические
    pDC->DPtoLP(&LogPoint);
    // Освободим контекст устройства
    ReleaseDC(pDC);
    switch(m_CurOper)
    {
        case OP_POINT:
        case OP_CIRCLE:
        case OP_SQUARE:
           AddShape(m_CurOper, m_FirstPoint, LogPoint);
           // Указываем, что окно надо перерисовать
           Invalidate();
        break;
        case OP_SELECT:
           pDoc->SelectShape(LogPoint);
           // Указываем, что окно надо перерисовать
        Invalidate();
        break;
    }

    // Даем возможность стандартному обработчику
    // тоже поработать над этим сообщением
    CScrollView::OnLButtonUp(nFlags, point);
}

void CPainterView::AddShape(int shape, CPoint first_point, CPoint sec-
ond_point)
{
    CPainterDoc *pDoc=GetDocument();
    CBasePoint *pShape=NULL;
    // Расчет размера
    int size=0;
    size=(int) floor( sqrt((second_point.x-first_point.x)*
                           (second_point.x-first_point.x)+
```

```
                        (second_point.y-first_point.y)*
                        (second_point.y-first_point.y)) +0.5);
    switch(shape)
    {
        case OP_LINE:
        break;
        case OP_POINT:
           // Создаем объект-точку
           pShape=new CBasePoint(second_point.x, second_point.y, 100);
           // Светло-серая заливка
           pShape->SetBrush(RGB(200,200,200));
        break;
        case OP_CIRCLE:
           // Создаем объект-круг
           pShape=new CBasePoint(first_point.x, first_point.y, size);
           // Черная линия шириной 2 мм
           pShape->SetPen(RGB(0,0,0), 200, PS_GEOMETRIC);
           // Темно-серая заливка
           pShape->SetBrush(RGB(100,100,100));
        break;
        case OP_SQUARE:
           // Создаем объект-квадрат
           pShape=new CSquare(first_point.x, first_point.y, size*2);
           // Красная линия шириной 1 мм
           pShape->SetPen(RGB(200,0,0), 100, PS_GEOMETRIC);
           // Темно-серая диагональная штриховка
           pShape->SetBrush(RGB(100,100,100),0,HS_DIAGCROSS);
        break;
    }
    if(pShape!=NULL) // Создали фигуру
    {
        // Добавляем в конец списка
        pDoc->m_ShapesList.AddTail(pShape);
        // Последняя фигура становится активной
        pDoc->m_pSelShape=pShape;
        // Указываем, что документ изменен
        pDoc->SetModifiedFlag();
    }
}
```

Используя новые возможности, мы теперь запросто сможем рисовать замечательные по своей художественной ценности произведения (рис. 6.3).

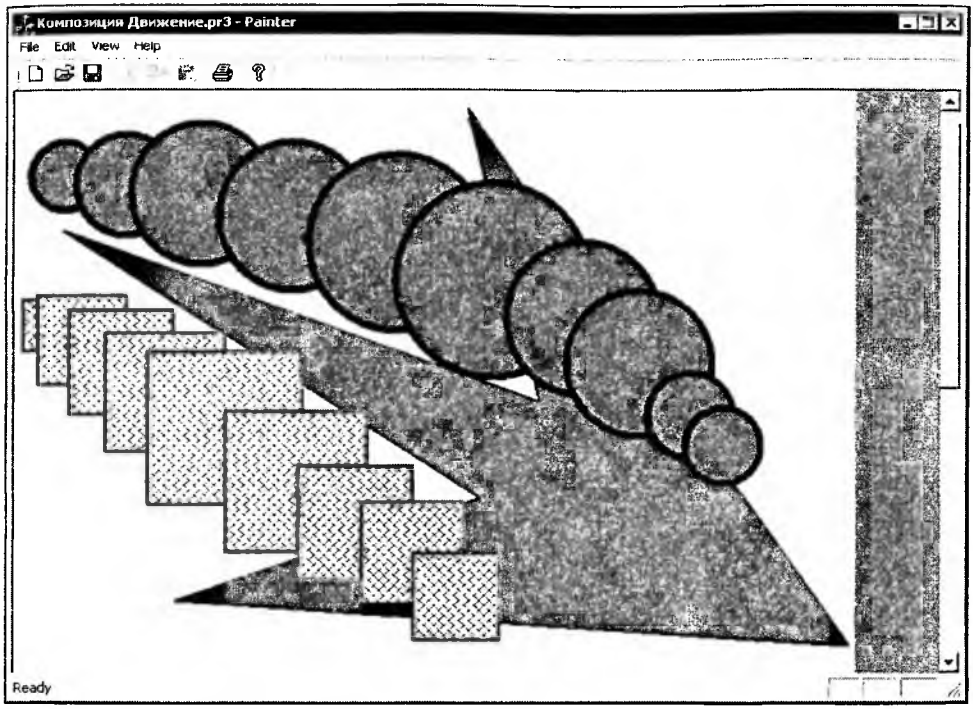


Рис. 6.3. Рисование фигур произвольных размеров

6.5. Рисование инверсным цветом

Многие операции по изменению изображений требуют вывода поверх картинки разного рода вспомогательных линий. Например, в процессе выполнения выбора части рисунка, во многих графических редакторах выделяемая область обозначается рамкой (рис. 6.4). Для того чтобы линия, нарисованная поверх какого-то другого изображения, была видна на любом фоне, применяют прием, называемый инверсией цвета. Прием заключается в том, что рисование производится цветом, обратным цвету пикселов экрана. Например, если пиксел экрана имеет цвет RGB (15, 233, 112), то новый цвет будет RBG (255-15, 255-233, 255-112). Таким образом, по белому фону будет нарисована черная линия, по черному — белая.

Рисование инверсным цветом имеет два основных достоинства:

- обеспечивает видимость выводимых рисунков в большинстве случаев, исключение — серый фон RGB (127, 127, 127);

- повторный вывод рисунка на то же место полностью восстанавливает исходное изображение.

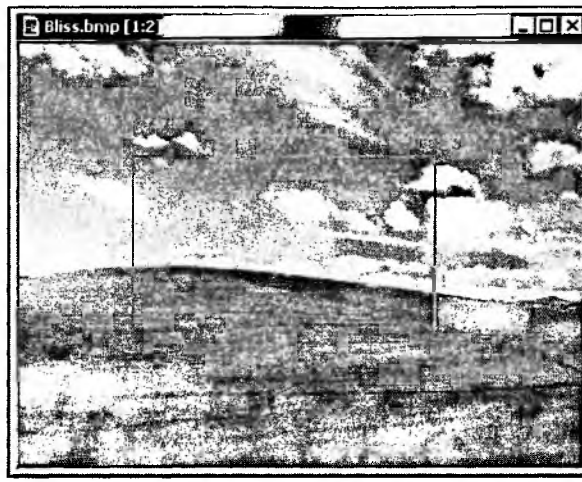


Рис. 6.4. Выделение области с помощью рамки

В нашем случае данный прием можно использовать для того, чтобы показать, на какое расстояние пользователь перетащил мышь при нажатой левой кнопке. Будем просто соединять прямой линией начальную точку операции и текущую точку, в которой находится мышь. Для этого заведем специальную функцию (листинг 6.9).

Листинг 6.9. Рисование отрезка прямой инверсным цветом

```
void CPainterView::DrawMoveLine(CPoint first_point, CPoint second_point)
{
   // Получим доступ к контексту устройства
   CClientDC dc(this);
   // Подготовим контекст устройства
   OnPrepareDC(&dc);
   // Установим режим рисования инверсным цветом
   int OldMode=dc.SetROP2(R2_NOT);
   // Рисуем прямую между двумя точками
   dc.MoveTo(first_point);   dc.LineTo(second_point);
   // Восстанавливаем прежний режим рисования
   dc.SetROP2(OldMode);
}
```

В функции DrawMoveLine() режим рисования инверсным цветом устанавливается методом SetROP2() класса CDC. Кроме инверсного режима рисования, с помощью этого метода можно установить и большое число других способов взаимодействия цвета выводимых рисунков с уже "закрашенными" пикселами экрана.

Различные режимы рисования представляют собой бинарные растровые операции, являющиеся всевозможными булевыми комбинациями двух переменных с использованием бинарных операторов AND, OR, XOR и NOT. Про режимы рисования можно почитать в MSDN, они хорошо описаны также в книгах [7, 16].

Обратите внимание, в этой функции использован другой способ получения доступа к контексту устройства. Раньше в случае такой надобности мы вызывали функцию GetDC(), а затем ReleaseDC(), например в функции OnLButtonUp(). С помощью же класса CClientDC мы можем получить доступ к дескриптору клиентской части окна, связанного с объектом-обликом, указатель на который передается конструктору в качестве аргумента. Конструктор объекта класса CClientDC вызывает GetDC(), а деструктор — ReleaseDC(). Поэтому вызывать их явно не надо.

Для того чтобы линия у нас постоянно отслеживала перемещения курсора, добавим в класс CPainterView функцию обработки сообщения WM_MOUSEMOVE (листинг 6.10).

Листинг 6.10. Метод CPainterView::OnMouseMove(). Файл PainterView.cpp

```
void CPainterView::OnMouseMove(UINT nFlags, CPoint point)
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
   CPoint   LogPoint=point;
   // Получим контекст устройства, на котором рисуем
   CDC *pDC=GetDC();
   // Подготовка контекста устройства – метод базового класса
   OnPrepareDC(pDC);
   // Переведем физические координаты точки в логические
   pDC->DPtoLP(&LogPoint);
   // Освободим контекст устройства
   ReleaseDC(pDC);
   switch(m_CurOper)
   {
       case OP_LINE:
```

```
            if(((CPolygon*)pDoc->m_ShapesList.GetTail())->
                                    m_PointsArray.GetSize()<=0) break;
            DrawMoveLine(m_FirstPoint, m_CurMovePoint);
            m_CurMovePoint=LogPoint;
            DrawMoveLine(m_FirstPoint, m_CurMovePoint);
         break;
         case OP_POINT:
         case OP_CIRCLE:
         case OP_SQUARE:
             if(nFlags==MK_LBUTTON)
                DrawMoveLine(m_FirstPoint, m_CurMovePoint);
             m_CurMovePoint=LogPoint;
             if(nFlags==MK_LBUTTON)
                DrawMoveLine(m_FirstPoint, m_CurMovePoint);
             break;
   }
   CScrollView::OnMouseMove(nFlags, point);
}
```

Эта функция отрабатывает сообщения о перемещении мыши. Внутри функции мы отрабатываем различные ситуации. Если пользователь рисует полигон, соединяем последнюю вершину полигона с точкой нахождения мыши. Если пользователь рисует какую-то фигуру, то соединяем точку начала операции с текущей точкой.

Обратите внимание, что рисование отрезка выполняется в два этапа. Сначала отрезок выводится на то место, где он был нарисован в прошлый раз. Затем рисуется на новом месте.

6.6. Реализация преобразований на плоскости

Использование систем уравнений (5.19) и (5.24), рассмотренных в *главе 5*, позволяет нам запрограммировать преобразования фигур на плоскости. Это будет коротенькая функция, которую целесообразно сделать глобальной. Назовем ее `Transform()`, прототип поместим в файл Global.h, а реализацию — в файл Global.cpp. Функция приведена в листинге 6.11.

Листинг 6.11. Функция `Transform()`. Файл Global.cpp

```
CPoint Transform(const CPoint &point,
                 const CPoint &point0, double ang, int a, int b)
```

```
{
   CPoint res;
   // Перевод в радианы
   ang=ang*atan(1.0)/45.0;
   res.x=(int)floor(point0.x+(point.x-point0.x)*cos(ang)-
                    (point.y-point0.y)*sin(ang)+a+0.5);
   res.y=(int)floor(point0.y+(point.x-point0.x)*sin(ang)+
                    (point.y-point0.y)*cos(ang)+b+0.5);
   return res;
};
```

Параметры функции:

- `point` — точка, над которой выполняется преобразование;
- `point0` — точка, вокруг которой выполняется операция поворота;
- `ang` — угол поворота в градусах;
- `a` — расстояние переноса по оси *X*;
- `b` — расстояние переноса по оси *Y*.

Возвращаемое значение — координаты преобразованной точки.

Для того чтобы объекты-фигуры определенных нами классов могли сами над собой выполнять заданное преобразование, включим в класс `CBasePoint` метод `Transform()`. Положение таких фигур, как "Точка", "Круг" и "Квадрат" задается единственной точкой, поэтому метод `Transform()` в классе `CBasePoint` отрабатывает только операцию переноса (листинг 6.12).

Листинг 6.12. Метод `CBasePoint::Transform()`. Файл Shapes.cpp

```
void CBasePoint::Transform(const CPoint &point0, double ang,
                           int a, int b)
{
   CPoint res=::Transform(CPoint(x, y), CPoint(0,0), 0, a, b);
   x=res.x; y=res.y;
};
```

Форма и положение фигуры "Полигон" определяются значениями координат ее вершин. Поэтому для класса `CPolygon` метод `Transform()` переопределен таким образом, чтобы преобразование выполнялось над всеми его вершинами (листинг 6.13).

Листинг 6.13. Метод `CPolygon::Transform()`. Файл Shapes.cpp

```
void CPolygon::Transform(const CPoint &point0, double ang, int a, int b)
{
   for(int i=0; i<m_PointsArray.GetSize(); i++)
      m_PointsArray[i]=::Transform(m_PointsArray[i],
                                   m_PointsArray[0], ang, a, b);
};
```

Обратите внимание, в качестве точки, относительно которой выполняется поворот, передаются координаты начальной вершины полигона, а переданная как параметр точка `point0` игнорируется. В принципе можно использовать и точку `point0`, только тогда нужно предусмотреть в программе средства для ее задания.

6.7. Определение реакций на нажатие клавиш

Как мы уже отмечали, обычно редактирование изображений выполняется в два этапа:

1. Выделение фигуры или области рисунка (группы фигур).
2. Преобразования над выделенной частью рисунка.

Будем полагать, что у нас уже имеется выделенная фигура и мы хотим выполнить над ней какое-либо двумерное преобразование. В "продвинутых" графических редакторах все множество преобразований, как правило, можно выполнить, виртуозно щелкая мышью, при этом для ускорения многих операций широко используются сочетания "горячих" клавиш. У нас же редактор начинающий, поэтому мы не станем забивать себе голову хитрой обработкой щелчков мыши, а ограничимся определением реакций на нажатие клавиш. Пусть клавиши со стрелками <↑>, <↓>, <←>, <→> будут использоваться для выполнения операции переноса, а <←>, <→> при нажатой клавише <Shift> — для операции поворота.

Как вы уже, наверно, заметили, в программировании, в противовес китайской пословице "тысячу способов узнать легко, одного результата добиться — трудно", часто множество путей ведет к достижению одной цели.

Обработку нажатий клавиш можно выполнять разными способами, отличающимися уровнем автоматизации.

Первый способ назначения "горячих" клавиш заключается в использовании таблицы акселераторов. Он применим для ускорения вызова команд меню. Таблица акселераторов создается генератором приложений и уже содержит

описание "горячих" клавиш для ряда стандартных команд. Чтобы определить комбинацию клавиш для какой-либо новой команды, достаточно дважды щелкнуть мышью на пустой строке таблицы, выбрать в появившемся диалоге идентификатор команды и назначить клавиши, которые станут "горячими" (рис. 6.5). В принципе, можно назначить "горячие" клавиши для идентификатора и не связанного с командой меню.

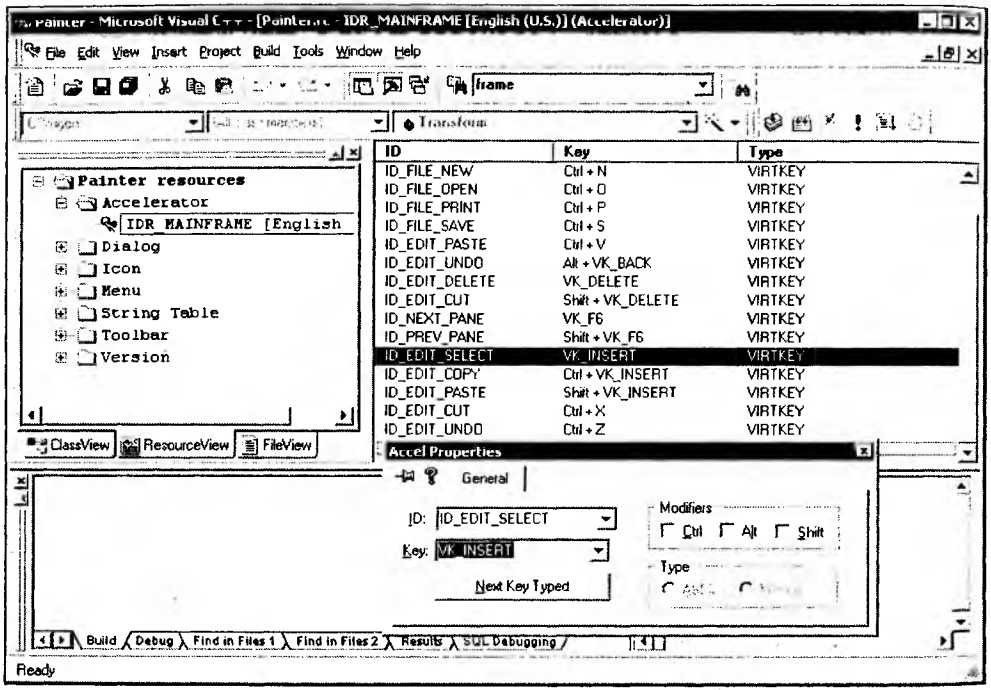


Рис. 6.5. Назначение клавиши <Insert> в качестве "горячей" для вызова команды **Edit | Select**

Другой способ заключается в определении функций-обработчиков сообщений `WM_KEYDOWN` и `WM_KEYUP` (напомним, что сделать это можно с помощью ClassWizard). Поскольку для преобразований мы не завели специальные команды меню, используем этот способ для определения реакций на нажатие клавиш. Обрабатывать сообщения поручим классу `CPainterView`, а реагировать на сообщения — классам самих фигур. Для этого добавим в класс `CPainterView` функции-обработчики сообщений: "нажали клавишу", "отпустили клавишу" (листинг 6.14). Задача обработчика в классе `CPainterView` — получить код нажатой клавиши и передать его методу `OnKeyDown` (листинг 6.15) выделенного объекта-фигуры.

Листинг 6.14. Функции-обработчики сообщений от клавиатуры. Файл CPainterView.cpp

```
void CPainterView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
   CPainterDoc *pDoc=GetDocument();
   BOOL modified=FALSE;
   if(pDoc->m_pSelShape!=NULL)
   modified=pDoc->m_pSelShape->OnKeyDown(nChar, nRepCnt, nFlags,
                                                        m_nMyFlags);
   switch(nChar)
   {
       case 16: m_nMyFlags=m_nMyFlags|SHIFT_HOLD; break; // Shift
       case 17: m_nMyFlags=m_nMyFlags|CTRL_HOLD; break; // Ctrl
   }
   if(modified)
   {
       // Указываем, что документ изменен
       pDoc->SetModifiedFlag();
       // Указываем, что окно надо перерисовать
       Invalidate();
   }
   CScrollView::OnKeyDown(nChar, nRepCnt, nFlags);
}

void CPainterView::OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags)
{
   switch(nChar)
   {
       case 16: m_nMyFlags=m_nMyFlags^SHIFT_HOLD; break; // Shift
       case 17: m_nMyFlags=m_nMyFlags^CTRL_HOLD;  break; // Ctrl
 }
   CScrollView::OnKeyUp(nChar, nRepCnt, nFlags);
}
```

В методе `CPainterView::OnKeyDown()` выполняются следующие действия:

- вызывается метод `OnKeyDown()` для активной фигуры;
- обрабатывается код нажатой клавиши: если нажата клавиша <Shift> или <Ctrl>, устанавливаются соответствующие биты в переменной `CPainterView::m_nMyFlags`;
- сообщается о необходимости перерисовки окна.

В методе `CPainterView::OnKeyUp()` при отпускании клавиш <Shift> и <Ctrl> обнуляются соответствующие биты переменной `CPainterView::m_nMyFlags`.

Клавиша <Ctrl> у нас пока не задействована, но, возможно, она пригодится нам далее, поэтому ее состояние тоже отслеживаем.

Так как мы поручаем объекту-фигуре самому реагировать на нажатие клавиш, то должны определить соответствующий метод. Поскольку все наши классы фигур имеют общего предка — класс `CBasePoint`, то новый метод (назовем его `OnKeyDown`) целесообразно ввести в этот базовый класс (листинг 6.15). Таким образом, все наши фигуры автоматически "научатся" реагировать на нажатие клавиш.

Листинг 6.15. Метод `CBasePoint::OnKeyDown()`. Файл Shapes.cpp

```
BOOL CBasePoint::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags,
                           UINT nMyFlags)
{
   BOOL res=TRUE;
   if(nMyFlags & SHIFT_HOLD) //поворот
   switch(nChar)
   {
       case 37:
          Transform(CPoint(0,0), -ROTATE_STEP, 0, 0);
          break;
       case 39:
          Transform(CPoint(0,0), ROTATE_STEP, 0, 0);
          break;
       default:
          res=FALSE;
   }
   else //перенос
   switch(nChar)
   {
       case 38: // вверх
          Transform(CPoint(0,0), 0, 0, MOVE_STEP);
          break;
       case 40: // вниз
          Transform(CPoint(0,0), 0, 0, -MOVE_STEP);
          break;
       case 37: // влево
          Transform(CPoint(0,0), 0, -MOVE_STEP, 0);
```

```
            break;
        case 39: // вправо
            Transform(CPoint(0,0), 0, MOVE_STEP, 0);
            break;
        default:
            res=FALSE;
    }
    return res;
}
```

В функции `OnKeyDown()` в зависимости от кода нажатой клавиши выбираются параметры, и вызывается метод `Transform()` выделенного объекта.

Обратите внимание, в метод `Transform()` передаются фиксированные значения переноса (`MOVE_STEP`) и поворота (`ROTATE_STEP`), определенные с помощью директивы `#define` в файле Shapes.h:

```
#define MOVE_STEP   100
#define ROTATE_STEP 5
```

Таким образом, каждое нажатие "горячей" клавиши влечет за собой либо перемещение объекта на 1 мм, либо поворот на 5 градусов.

Метод `CBasePoint::OnKeyDown()` возвращает значение `TRUE`, если состояние фигуры изменилось (нажание было обработано), или `FALSE`, если ничего не изменилось. Соответственно метод `CPainterView::OnKeyDown()` узнает, надо ли выполнять перерисовку.

6.8. Изменение порядка наложения фигур

Часто бывает полезным изменить порядок наложения фигур на рисунке. При нашей организации хранения объектов-фигур добавить такую возможность довольно легко. Достаточно менять местами адреса объектов в списке. Как правило, при изменении порядка наложения фигур ("уровня") на рисунке используют следующие команды.

- Переместить на самый верхний уровень. Так как вывод фигур происходит с начала списка, для выполнения этой команды нам надо переместить элемент списка с указателем на выделенную фигуру в самый конец списка. Тогда она будет выводиться последней поверх всех остальных фигур.
- Переместить на самый нижний уровень — достаточно переместить элемент списка с указателем на выделенную фигуру в самое начало списка.

- Переместить на уровень вверх — достаточно поменять местами элемент списка с указателем на выделенную фигуру со следующим элементом списка.
- Переместить на уровень вниз — достаточно поменять местами элемент списка с указателем на выделенную фигуру с предыдущим элементом списка.

Все эти операции реализуем в функции `ChangeOrder()`, которую сделаем методом класса `CPainterDoc` (листинг 6.16). В качестве параметра она принимает код команд, которые определим в файле PainterDoc.h:

```
#define TOP       0
#define BOTTOM    1
#define STEPUP    2
#define STEPDOWN  3
```

Листинг 6.16. Функция `CPainterDoc::ChangeOrder()`. Файл PainterDoc.cpp

```
BOOL CPainterDoc::ChangeOrder(int code)
{
   if(m_pSelShape==NULL) return FALSE;
   CBasePoint *pShape=NULL;
   // Указатель на элемент списка
   POSITION pos=NULL, pastpos=NULL;
   // Начнем поиск с "хвоста" списка
   if(m_ShapesList.GetCount()>0) pos=m_ShapesList.GetTailPosition();
   // Найдем позицию объекта в списке
   while(pos!=NULL)
   {
       if(m_pSelShape==m_ShapesList.GetAt(pos)) break;
       m_ShapesList.GetPrev(pos);
   }
   if(pos!=NULL) // Нашли элемент с указателем на выделенный объект
   switch(code)
   {
      case TOP:
         m_ShapesList.RemoveAt(pos);
         m_ShapesList.AddTail(m_pSelShape);
         break;
      case BOTTOM:
         m_ShapesList.RemoveAt(pos);
```

```
            m_ShapesList.AddHead(m_pSelShape);
            break;
        case STEPUP:
            pastpos=pos;
            m_ShapesList.GetNext(pos);
            if(pos!=NULL)
            {
                m_ShapesList.RemoveAt(pastpos);
                m_ShapesList.InsertAfter(pos, m_pSelShape);
            }
            break;
        case STEPDOWN:
            pastpos=pos;
            m_ShapesList.GetPrev(pos);
            if(pos!=NULL)
            {
                m_ShapesList.RemoveAt(pastpos);
                m_ShapesList.InsertBefore(pos, m_pSelShape);
            }
            break;
    }
    return TRUE;
};
```

Для того чтобы пользователь мог выполнить операции по изменению "уровня" выбранной фигуры, добавим в меню соответствующие команды (рис. 6.6).

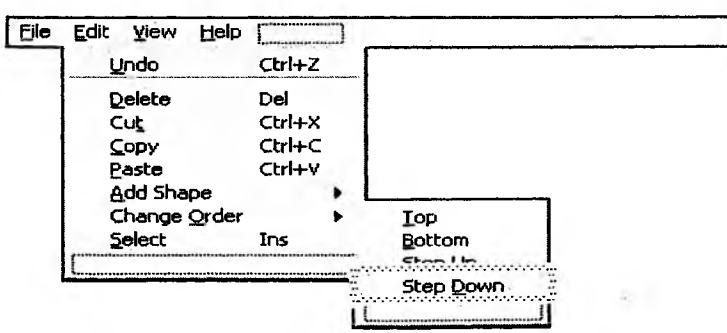


Рис. 6.6. Меню программы с добавленными командами изменения "уровня" выбранной фигуры

Функции-обработчики этих команд добавим в класс CPainterView (листинг 6.17).

Листинг 6.17. Функции-обработчики команд изменения "уровня" выбранной фигуры

```
void CPainterView::OnEditChangeorderTop()
{
   CPainterDoc *pDoc=GetDocument();
   if(pDoc->ChangeOrder(TOP))
   {
       // Указываем, что документ изменен
       pDoc->SetModifiedFlag();
       // Указываем, что окно надо перерисовать
       Invalidate();
   }
}

void CPainterView::OnEditChangeorderStepup()
{
   CPainterDoc *pDoc=GetDocument();
   if(pDoc->ChangeOrder(STEPUP))
   {
       // Указываем, что документ изменен
       pDoc->SetModifiedFlag();
       // Указываем, что окно надо перерисовать
       Invalidate();
   }
}

void CPainterView::OnEditChangeorderStepdown()
{
   CPainterDoc *pDoc=GetDocument();
   if(pDoc->ChangeOrder(STEPDOWN))
   {
       // Указываем, что документ изменен
       pDoc->SetModifiedFlag();
       // Указываем, что окно надо перерисовать
       Invalidate();
```

```
    }
}

void CPainterView::OnEditChangeorderBottom()
{
    CPainterDoc *pDoc=GetDocument();
    if(pDoc->ChangeOrder(BOTTOM))
    {
        // Указываем, что документ изменен
        pDoc->SetModifiedFlag();
        // Указываем, что окно надо перерисовать
        Invalidate();
    }
}
```

Результат применения команды "на уровень вверх" к выделенному кругу (см. рис. 6.2) приведен на рис. 6.7 — теперь круг рисуется поверх квадрата.

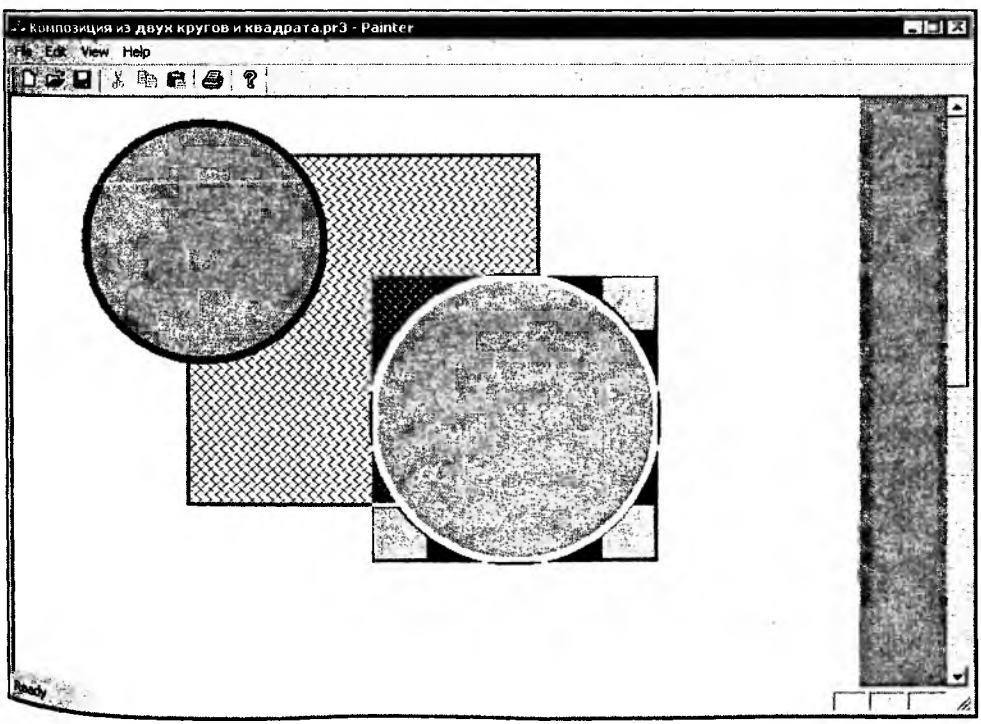


Рис. 6.7. Изменение уровня выделенного круга

6.9. Удаление фигур

Набор команд по редактированию должен обязательно включать команду удаления. Для реализации этой команды добавим в класс `CPainterDoc` метод `DeleteSelected()`, который будет удалять из списка выбранный объект-фигуру (листинг 6.18).

Листинг 6.18. Метод `CPainterDoc::DeleteSelected()`. Файл PainterDoc.cpp

```
BOOL CPainterDoc::DeleteSelected()
{
   if(m_pSelShape==NULL) return FALSE;
   CBasePoint *pShape=NULL;
   // Указатель на элемент списка
   POSITION pos=NULL, pastpos=NULL;
   // Начиная с "хвоста" списка
   if(m_ShapesList.GetCount()>0) pos=m_ShapesList.GetTailPosition();
   // Найдем позицию объекта в списке
   while(pos!=NULL)
   {
       if(m_pSelShape==m_ShapesList.GetAt(pos)) break;
       m_ShapesList.GetPrev(pos);
   }
   if(pos!=NULL) // Нашли его позицию
   {
       m_ShapesList.RemoveAt(pos);
       delete m_pSelShape;
       m_pSelShape=NULL;
       return TRUE;
   }
   return FALSE;
};
```

Вызов этого метода будет осуществляться из функции-обработчика `CPainterView::OnEditDelete()` команды **Delete**, которую добавим в меню **Edit** программы (листинг 6.19).

Листинг 6.19. Функция `CPainterView::OnEditDelete()`. Файл PainterView.cpp

```
void CPainterView::OnEditDelete()
{
```

```
    // TODO: Add your command handler code here
    CPainterDoc *pDoc=GetDocument();
    if(pDoc->DeleteSelected())
    {
        // Указываем, что документ изменен
        pDoc->SetModifiedFlag();
        // Указываем, что окно надо перерисовать
        Invalidate();
    }
}
```

Обратите внимание, удаление из списка фигур элемента, хранящего указатель на объект-фигуру, конечно, приведет к тому, что фигура не будет присутствовать на рисунке, однако чтобы предотвратить утечку памяти, мы также удаляем и сам объект-фигуру (см. листинг 6.18).

6.10. Преобразование формата

Так как мы ликвидировали в нашем объекте-документе массив точек, который ранее использовался для хранения полигонов, процедура сохранения рисунков будет выглядеть иначе, чем в предыдущей версии программы. Поэтому мы должны организовать загрузку таким образом, чтобы ранее созданные рисунки читались верно. Для того чтобы различать версии программ, у нас имеется идентификатор формата. Теперь, когда у нас есть специальный класс для описания полигональных фигур, мы можем использовать его для того, чтобы загрузить в него данные, ранее хранившиеся в статическом массиве (листинг 6.20).

Листинг 6.20. Функция `CPainterDoc::Serialize()`. Файл PainterDoc.cpp

```
void CPainterDoc::Serialize(CArchive& ar)
{
    CString version;
    // Количество точек
    WORD Index=0, i=0, version_n=0;
    CPoint point;

    CBasePoint *pPolygon=NULL;
    if (ar.IsStoring()) // Сохраняем
    {
        // версию формата наших рисунков
```

```
    version_n=3;
    version.Format("pr%d", version_n);
    ar << version;
    // Режим отображения
    ar << m_wMap_Mode;
    // Размер листа
    ar << m_wSheet_Width;
    ar << m_wSheet_Height;
}
else // Загружаем
{
    // версию формата
    ar >> version;
    version_n=atoi((LPCTSTR)version.Right(1));
    switch(version_n) // В зависимости от версии формата
    {
        case 2:
           pPolygon=new CPolygon;
           pPolygon->SetPen(RGB(0,0,0), 50, PS_GEOMETRIC);
           ar >> Index;
           // Загружаем значения координат точек
           for(i=0; i<Index; i++)
           {
               ar >> point;
               ((CPolygon*)pPolygon)->m_PointsArray.Add(point);
           }
        case 3:
           // Режим отображения
           ar >> m_wMap_Mode;
           // Размер листа
           ar >> m_wSheet_Width;
           ar >> m_wSheet_Height;
           break;
        default:
           AfxMessageBox("Неизвестный формат", MB_OK);
           return;
    }//switch(version)
}

// Выполняем сериализацию списка фигур
```

```
    m_ShapesList.Serialize(ar);
    // Добавляем объект-полигон в "голову" списка
    if(pPolygon!=NULL)
        m_ShapesList.AddHead(pPolygon);
}
```

6.11. Листинг программы

В программу было внесено большое количество изменений, поэтому не лишним будет привести текст основных файлов полностью:

- PainterDoc.h — содержит интерфейс класса документа приложения;
- PainterDoc.cpp — содержит реализацию методов класса документа приложения;
- PainterView.h — содержит интерфейс класса облика приложения;
- PainterView.cpp — содержит реализацию методов класса облика приложения;
- Shapes.h — содержит интерфейсы классов, описывающих различные фигуры;
- Shapes.cpp — содержит реализацию методов классов, интерфейсы которых описаны в Shapes.h;
- Global.h — содержит описание прототипов глобальных функций приложения;
- Global.cpp — содержит реализацию глобальных функций приложения.

6.11.1. Файл PainterDoc.h

```
// PainterDoc.h : interface of the CPainterDoc class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_PAINTERDOC_H__F8B9924B_79CF_11D5_BB4A↵
_20804AC10000__INCLUDED_)
#define AFX_PAINTERDOC_H__F8B9924B_79CF_11D5_BB4A_20804AC10000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


#define TOP        0
```

```
#define BOTTOM    1
#define STEPUP    2
#define STEPDOWN  3

class CBasePoint;
class CPainterDoc : public CDocument
{
protected:
   CPainterDoc();
   DECLARE_DYNCREATE(CPainterDoc)

// Данные
public:
   // Ширина листа
   WORD m_wSheet_Width;
   // Высота листа
   WORD m_wSheet_Height;
   // Режим отображения
   WORD m_wMap_Mode;

   // Список указателей на объекты-фигуры
   CTypedPtrList<CObList, CBasePoint*> m_ShapesList;
   // Указатель на выбранную фигуру
   CBasePoint* m_pSelShape;

// Методы
public:
   // Очистка списка объектов
   void ClearShapesList();
   // Выбор активной фигуры
   CBasePoint* SelectShape(CPoint point);
   // Изменение порядка следования активной фигуры в списке фигур
   BOOL ChangeOrder(int code);
   // Удалить выделенную фигуру
   BOOL DeleteSelected();

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CPainterDoc)
```

```
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CPainterDoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CPainterDoc)
    afx_msg void OnFilePageproperty();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.

#endif // !defined(AFX_PAINTERDOC_H__F8B9924B_79CF_11D5_BB4A↵
_20804AC10000__INCLUDED_)
```

6.11.2. Файл PainterDoc.cpp

```
// PainterDoc.cpp : implementation of the CPainterDoc class
//

#include "stdafx.h"
#include "Painter.h"

#include "PainterDoc.h"
```

```
#include "PainterView.h"
#include "PagePropertyDlg.h"
#include "Shapes.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CPainterDoc

IMPLEMENT_DYNCREATE(CPainterDoc, CDocument)

BEGIN_MESSAGE_MAP(CPainterDoc, CDocument)
   //{{AFX_MSG_MAP(CPainterDoc)
   ON_COMMAND(ID_FILE_PAGEPROPERTY, OnFilePageproperty)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CPainterDoc construction/destruction

CPainterDoc::CPainterDoc()
{
   // Режим отображения 1 лог. ед. = 0,01 мм
   m_wMap_Mode = MM_HIMETRIC;
   // Размер листа формата A4
   m_wSheet_Width = 21000;
   m_wSheet_Height = 29700;
   // Нет выбранной фигуры
   m_pSelShape=NULL;
}

CPainterDoc::~CPainterDoc()
{
   // Очистили список фигур
   ClearShapesList();
```

```
}

BOOL CPainterDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   // TODO: add reinitialization code here
   // (SDI documents will reuse this document)
   // Нет выбранной фигуры
    m_pSelShape=NULL;
   // Очистили список фигур
   ClearShapesList();
   // Перерисовали
   UpdateAllViews(NULL);
   return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CPainterDoc serialization

void CPainterDoc::Serialize(CArchive& ar)
{
   CString version;
   // Количество точек
   WORD Index=0, i=0, version_n=0;
   CPoint point;

   CBasePoint *pPolygon=NULL;
   if (ar.IsStoring()) // Сохраняем
   {
       // Версию формата наших рисунков
       version_n=3;
       version.Format("pr%d", version_n);
       ar << version;
       // Режим отображения
       ar << m_wMap_Mode;
       // Размер листа
       ar << m_wSheet_Width;
```

```
        ar << m_wSheet_Height;
    }
    else // Загружаем
    {
        // версию формата
        ar >> version;
        version_n=atoi((LPCTSTR)version.Right(1));
        switch(version_n) // В зависимости от версии формата
        {
            case 2:
               pPolygon=new CPolygon;
               pPolygon->SetPen(RGB(0,0,0), 50, PS_GEOMETRIC);
               ar >> Index;
               // Загружаем значения координат точек
               for(i=0; i<Index; i++)
               {
                   ar >> point;
                   ((CPolygon*)pPolygon)->m_PointsArray.Add(point);
               }
            case 3:
               // Режим отображения
               ar >> m_wMap_Mode;
               // Размер листа
               ar >> m_wSheet_Width;
               ar >> m_wSheet_Height;
               break;
            default:
               AfxMessageBox("Неизвестный формат", MB_OK);
               return;
        }//switch(version)
    }
    // Выполняем сериализацию списка фигур
    m_ShapesList.Serialize(ar);
    // Добавляем объект-полигон в "голову" списка
    if(pPolygon!=NULL)
       m_ShapesList.AddHead(pPolygon);
}

//////////////////////////////////////////////////////////////////////////
```

```
// CPainterDoc diagnostics

#ifdef _DEBUG
void CPainterDoc::AssertValid() const
{
   CDocument::AssertValid();
}

void CPainterDoc::Dump(CDumpContext& dc) const
{
   CDocument::Dump(dc);
}
#endif //_DEBUG

////////////////////////////////////////////////////////////////////////////
// CPainterDoc commands

void CPainterDoc::OnFilePageproperty()
{
   // TODO: Add your command handler code here
   // Создаем объект-диалог свойств листа
   CPagePropertyDlg PPDlg;
   // Инициализируем параметры диалога текущими значениями
   // Делим на 100, т. к. в диалоге размеры в мм
   PPDlg.m_uWidth=m_wSheet_Width/100;
   PPDlg.m_uHeight=m_wSheet_Height/100;
   // Вызываем диалог
   if(PPDlg.DoModal()==IDOK)
   {   // Запоминаем новые значения
      // Умножаем на 100, т. к. 1 лог. ед. = 0,01 мм
      m_wSheet_Width=PPDlg.m_uWidth*100;
      m_wSheet_Height=PPDlg.m_uHeight*100;
      // Обновляем облик
      UpdateAllViews(NULL);
   }
}

void CPainterDoc::ClearShapesList()
{
```

```
    // Очистили список объектов
    POSITION pos=NULL;
    while(m_ShapesList.GetCount()>0) // пока в списке есть фигуры
    delete m_ShapesList.RemoveHead();// удаляем первую из них
}

CBasePoint* CPainterDoc::SelectShape(CPoint point)
{
    // Объект-область
    CRgn Rgn;
   // Указатель на элемент списка
   POSITION pos=NULL;
   // Начиная с "хвоста" списка
   if(m_ShapesList.GetCount()>0) pos=m_ShapesList.GetTailPosition();
   // Проверим, попадает ли точка point в какую-либо из фигур
   while(pos!=NULL)
   {
       m_pSelShape=m_ShapesList.GetPrev(pos);
       // Очистим объект-область
       Rgn.DeleteObject();
       m_pSelShape->GetRegion(Rgn);
       // Точка попадает в фигуру — возвращаем указатель на фигуру
       if(Rgn.PtInRegion(point) ) return m_pSelShape;
   }
   // Если добрались до этого места, значит, не попали ни в какую фигуру
   return (m_pSelShape=NULL);
};

BOOL CPainterDoc::ChangeOrder(int code)
{
   if(m_pSelShape==NULL) return FALSE;
   CBasePoint *pShape=NULL;
   // Указатель на элемент списка
   POSITION pos=NULL, pastpos=NULL;
   // Начнем поиск с "хвоста" списка
   if(m_ShapesList.GetCount()>0) pos=m_ShapesList.GetTailPosition();
   // Найдем позицию объекта в списке
   while(pos!=NULL)
   {
```

```
        if(m_pSelShape==m_ShapesList.GetAt(pos)) break;
        m_ShapesList.GetPrev(pos);
    }
    if(pos==NULL) return FALSE;
        // Нашли элемент с указателем на выделенный объект
        // Меняем позицию элемента в списке
    switch(code)
    {
        case TOP:
            m_ShapesList.RemoveAt(pos);
            m_ShapesList.AddTail(m_pSelShape);
            break;
        case BOTTOM:
            m_ShapesList.RemoveAt(pos);
            m_ShapesList.AddHead(m_pSelShape);
            break;
        case STEPUP:
            pastpos=pos;
            m_ShapesList.GetNext(pos);
            if(pos!=NULL)
            {
                m_ShapesList.RemoveAt(pastpos);
                m_ShapesList.InsertAfter(pos, m_pSelShape);
            }
            break;
        case STEPDOWN:
            pastpos=pos;
            m_ShapesList.GetPrev(pos);
            if(pos!=NULL)
            {
                m_ShapesList.RemoveAt(pastpos);
                m_ShapesList.InsertBefore(pos, m_pSelShape);
            }
            break;
    }
    return TRUE;
};

BOOL CPainterDoc::DeleteSelected()
```

```
{
   if(m_pSelShape==NULL) return FALSE;
   CBasePoint *pShape=NULL;
   // Указатель на элемент списка
   POSITION pos=NULL, pastpos=NULL;
   // Начиная с "хвоста" списка
   if(m_ShapesList.GetCount()>0) pos=m_ShapesList.GetTailPosition();
   // Найдем позицию объекта в списке
   while(pos!=NULL)
   {
       if(m_pSelShape==m_ShapesList.GetAt(pos)) break;
       m_ShapesList.GetPrev(pos);
   }
   if(pos!=NULL) // Нашли его позицию
   {
      m_ShapesList.RemoveAt(pos);
      delete m_pSelShape;
      m_pSelShape=NULL;
      return TRUE;
   }
   return FALSE;
};
```

6.11.3. Файл PainterView.h

```
// PainterView.h : interface of the CPainterView class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_PAINTERVIEW_H__F8B9924D_79CF_11D5_BB4A↵
_20804AC10000__INCLUDED_)
#define
AFX_PAINTERVIEW_H__F8B9924D_79CF_11D5_BB4A_20804AC10000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// Определение операций
#define OP_NOOPER    0
```

```
#define OP_LINE      1
#define OP_POINT     2
#define OP_CIRCLE    3
#define OP_SQUARE    4
#define OP_SELECT    10
class CPainterView : public CScrollView
{
protected: // create from serialization only
   CPainterView();
   DECLARE_DYNCREATE(CPainterView)

// Данные
public:
   // Текущая операция
   int m_CurOper;
   // Начальная и текущая точки операции
   CPoint m_FirstPoint, m_CurMovePoint;
   // Индикатор нажатия клавиш Shift, Ctrl
   UINT m_nMyFlags;

// Методы
   CPainterDoc* GetDocument();
public:
  void AddShape(int shape, CPoint first_point, CPoint second_point);
   void MarkSelectedShape(CDC *pDC);
   void DrawMoveLine(CPoint first_point, CPoint second_point);

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CPainterView)
   public:
   virtual void OnDraw(CDC* pDC);  // overridden to draw this view
   virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
   virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo = NULL);
   protected:
   virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
   virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
   virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
   virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
```

```
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CPainterView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CPainterView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    afx_msg void OnEditAddshapePoint();
    afx_msg void OnEditAddshapeCircle();
    afx_msg void OnEditAddshapeSquare();
    afx_msg void OnEditSelect();
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnEditAddshapePolyline();
    afx_msg void OnEditAddshapePolygon();
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnEditChangeorderTop();
    afx_msg void OnEditChangeorderStepup();
    afx_msg void OnEditChangeorderStepdown();
    afx_msg void OnEditChangeorderBottom();
    afx_msg void OnEditDelete();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in PainterView.cpp
inline CPainterDoc* CPainterView::GetDocument()
```

```
    { return (CPainterDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.

#endif // !defined(AFX_PAINTERVIEW_H__F8B9924D_79CF_11D5_BB4A↵
_20804AC10000__INCLUDED_)
```

6.11.4. Файл PainterView.cpp

```
// PainterView.cpp : implementation of the CPainterView class
//

#include "stdafx.h"
#include "Painter.h"

#include "PainterDoc.h"
#include "PainterView.h"
#include "Shapes.h"
#include <math.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CPainterView

IMPLEMENT_DYNCREATE(CPainterView, CScrollView)

BEGIN_MESSAGE_MAP(CPainterView, CScrollView)
    //{{AFX_MSG_MAP(CPainterView)
    ON_WM_LBUTTONDOWN()
    ON_WM_ERASEBKGND()
```

```
    ON_COMMAND(ID_EDIT_ADDSHAPE_POINT, OnEditAddshapePoint)
    ON_COMMAND(ID_EDIT_ADDSHAPE_CIRCLE, OnEditAddshapeCircle)
    ON_COMMAND(ID_EDIT_ADDSHAPE_SQUARE, OnEditAddshapeSquare)
    ON_COMMAND(ID_EDIT_SELECT, OnEditSelect)
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
    ON_COMMAND(ID_EDIT_ADDSHAPE_POLYLINE, OnEditAddshapePolyline)
    ON_COMMAND(ID_EDIT_ADDSHAPE_POLYGON, OnEditAddshapePolygon)
    ON_WM_LBUTTONDBLCLK()
    ON_WM_KEYDOWN()
    ON_WM_KEYUP()
    ON_COMMAND(ID_EDIT_CHANGEORDER_TOP, OnEditChangeorderTop)
    ON_COMMAND(ID_EDIT_CHANGEORDER_STEPUP, OnEditChangeorderStepup)
    ON_COMMAND(ID_EDIT_CHANGEORDER_STEPDOWN, OnEditChangeorderStepdown)
    ON_COMMAND(ID_EDIT_CHANGEORDER_BOTTOM, OnEditChangeorderBottom)
    ON_COMMAND(ID_EDIT_DELETE, OnEditDelete)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CPainterView construction/destruction

CPainterView::CPainterView()
{
    // TODO: add construction code here
    m_CurOper=OP_NOOPER;
    m_bShifDown=FALSE;
}

CPainterView::~CPainterView()
{
}

BOOL CPainterView::PreCreateWindow(CREATESTRUCT& cs)
{
```

```
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs
    return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CPainterView drawing

void CPainterView::OnDraw(CDC* pDC)
{
    CPainterDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    // Выводим все фигуры, хранящиеся в списке
    POSITION pos=NULL;
    CBasePoint* pShape=NULL;
    // Если в списке есть объекты
    if(pDoc->m_ShapesList.GetCount()>0)
       // Получим позицию первого объекта
       pos=pDoc->m_ShapesList.GetHeadPosition();
     while(pos!=NULL)
     {
        // Получим указатель на первый объект
        pShape=pDoc->m_ShapesList.GetNext(pos);
        // Нарисуем объект
        if(pShape!=NULL) pShape->Show(pDC);
     }
     // Выделяем активную фигуру
     if(m_CurOper==OP_SELECT)    MarkSelectedShape(pDC);
}

void CPainterView::DrawMoveLine(CPoint first_point, CPoint second_point)
{
    // Получим доступ к контексту устройства
    CClientDC dc(this);
    // Подготовим контекст устройства
    OnPrepareDC(&dc);
    // Установим режим рисования инверсным цветом
    int OldMode=dc.SetROP2(R2_NOT);
```

```
    // Рисуем прямую между двумя точками
    dc.MoveTo(first_point);   dc.LineTo(second_point);
    // Восстанавливаем прежний режим рисования
    dc.SetROP2(OldMode);
}

////////////////////////////////////////////////////////////////////////
// CPainterView printing

BOOL CPainterView::OnPreparePrinting(CPrintInfo* pInfo)
{
    return DoPreparePrinting(pInfo);
}

void CPainterView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CPainterView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

////////////////////////////////////////////////////////////////////////
// CPainterView diagnostics

#ifdef _DEBUG
void CPainterView::AssertValid() const
{
    CView::AssertValid();
}

void CPainterView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CPainterDoc* CPainterView::GetDocument() // non-debug version is inline
```

```
{
   ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CPainterDoc)));
   return (CPainterDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CPainterView message handlers

void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
   CPoint   LogPoint=point;

   // Получим контекст устройства, на котором рисуем
   CDC *pDC=GetDC();
   // Подготовим контекст устройства
   OnPrepareDC(pDC);

   // Переведем физические координаты точки в логические
   pDC->DPtoLP(&LogPoint);
   // Освободим контекст устройства
   ReleaseDC(pDC);

   // Запоминаем точку
   m_CurMovePoint=m_FirstPoint=LogPoint;
   switch(m_CurOper)
   {
       case OP_LINE:
          // Последним в списке должен быть полигон
          ((CPolygon*)pDoc->m_ShapesList.GetTail())->
                              m_PointsArray.Add(LogPoint);
          // Указываем, что окно надо перерисовать
          Invalidate();
          break;
   }

   // Даем возможность стандартному обработчику
```

```
    // тоже поработать над этим сообщением
    CScrollView::OnLButtonDown(nFlags, point);
}

void CPainterView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // Получили указатель на объект-документ
    CPainterDoc *pDoc=GetDocument();
    CPoint   LogPoint=point;

    // Получим контекст устройства, на котором рисуем
    CDC *pDC=GetDC();
    // Подготовим контекст устройства метод базового класса
    OnPrepareDC(pDC);
    // Переведем физические координаты точки в логические
    pDC->DPtoLP(&LogPoint);
    // Освободим контекст устройства
    ReleaseDC(pDC);
    switch(m_CurOper)
    {
       case OP_POINT:
       case OP_CIRCLE:
       case OP_SQUARE:
          AddShape(m_CurOper, m_FirstPoint, LogPoint);
          // Указываем, что окно надо перерисовать
          Invalidate();
          break;
       case OP_SELECT:
          pDoc->SelectShape(LogPoint);
          // Указываем, что окно надо перерисовать
          Invalidate();
          break;
    }

    // Даем возможность стандартному обработчику
    // тоже поработать над этим сообщением
    CScrollView::OnLButtonUp(nFlags, point);
}

void CPainterView::OnMouseMove(UINT nFlags, CPoint point)
```

```
{
   // Получили указатель на объект-документ
   CPainterDoc *pDoc=GetDocument();
   CPoint   LogPoint=point;
   // Получим контекст устройства, на котором рисуем
   CDC *pDC=GetDC();
   // Подготовим контекст устройства
   OnPrepareDC(pDC);
   // Переведем физические координаты точки в логические
   pDC->DPtoLP(&LogPoint);
   // Освободим контекст устройства
   ReleaseDC(pDC);
   switch(m_CurOper)
   {
      case OP_LINE:
         if(((CPolygon*)pDoc->m_ShapesList.GetTail())->
            m_PointsArray.GetSize()<=0) break;
         DrawMoveLine(m_FirstPoint, m_CurMovePoint);
         m_CurMovePoint=LogPoint;
         DrawMoveLine(m_FirstPoint, m_CurMovePoint);
         break;
      case OP_POINT:
      case OP_CIRCLE:
      case OP_SQUARE:
         if(nFlags==MK_LBUTTON) DrawMoveLine(m_FirstPoint,
                                             m_CurMovePoint);
         m_CurMovePoint=LogPoint;
         if(nFlags==MK_LBUTTON) DrawMoveLine(m_FirstPoint,
                                             m_CurMovePoint);
         break;
   }
   CScrollView::OnMouseMove(nFlags, point);
}
void CPainterView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
   switch(m_CurOper)
   {
      case OP_LINE:
         m_CurOper=OP_NOOPER;
```

```
            break;
    }
    CScrollView::OnLButtonDblClk(nFlags, point);
}

void CPainterView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    // Получили указатель на объект-документ
    CPainterDoc *pDoc=GetDocument();
    CSize sizeTotal;
    // Ширина
    sizeTotal.cx = pDoc->m_wSheet_Width;
    // Высота
    sizeTotal.cy = pDoc->m_wSheet_Height;
    // Установим режим и размер листа
    SetScrollSizes(pDoc->m_wMap_Mode, sizeTotal);
    // Вызываем метод базового класса
    CScrollView::OnUpdate(pSender, lHint, pHint);
}

void CPainterView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // Вызов метода базового класса
    CScrollView::OnPrepareDC(pDC, pInfo);
    // Получим указатель на документ
    CPainterDoc *pDoc=GetDocument();
    // Создадим точку в левом нижнем углу листа
    CPoint OriginPoint(0, -pDoc->m_wSheet_Height);
    // Переведем точку в координаты физического устройства
    pDC->LPtoDP(&OriginPoint);
    // Установим эту точку
    // в качестве начала координат физического устройства
    pDC->SetViewportOrg(OriginPoint);
    // Ограничим область, доступную для рисования
    pDC->IntersectClipRect( 0,0, pDoc->m_wSheet_Width,
                                 pDoc->m_wSheet_Height);
}

BOOL CPainterView::OnEraseBkgnd(CDC* pDC)
```

```
{
   // Вызвали метод базового класса
    BOOL Res=CScrollView::OnEraseBkgnd(pDC);
   // Создали кисть серого цвета
   CBrush br( GetSysColor( COLOR_GRAYTEXT ) );
   // Выполнили заливку неиспользуемой области окна
   FillOutsideRect( pDC, &br );
   return Res;
}

void CPainterView::OnEditAddshapePoint()
{
   m_CurOper=OP_POINT;
}

void CPainterView::OnEditAddshapeCircle()
{
   m_CurOper=OP_CIRCLE;
}

void CPainterView::OnEditAddshapeSquare()
{
   m_CurOper=OP_SQUARE;
}

void CPainterView::AddShape(int shape, CPoint first_point,
                                       CPoint second_point)
{
   CPainterDoc *pDoc=GetDocument();
   CBasePoint *pShape=NULL;
   // Расчет размера
   int size=0;
   size=(int) floor( sqrt((second_point.x-first_point.x)*
                          (second_point.x-first_point.x)+
                          (second_point.y-first_point.y)*
                          (second_point.y-first_point.y)) +0.5);
   switch(shape)
   {
       case OP_LINE:
          break;
```

```
        case OP_POINT:
            // Создаем объект-точку
            pShape=new CBasePoint(second_point.x, second_point.y, 100);
            // Светло-серая заливка
            pShape->SetBrush(RGB(200,200,200));
            break;
        case OP_CIRCLE:
            // Создаем объект-круг
            pShape=new CBasePoint(first_point.x, first_point.y, size);
            // Черная линия шириной 2 мм
            pShape->SetPen(RGB(0,0,0), 200, PS_GEOMETRIC);
            // Темно-серая заливка
            pShape->SetBrush(RGB(100,100,100));
            break;
        case OP_SQUARE:
            // Создаем объект — квадрат
            pShape=new CSquare(first_point.x, first_point.y, size*2);
            // Красная линия шириной 1 мм
            pShape->SetPen(RGB(200,0,0), 100, PS_GEOMETRIC);
            // Темно-серая диагональная штриховка
            pShape->SetBrush(RGB(100,100,100),0,HS_DIAGCROSS);
            break;
    }
    if(pShape!=NULL) // Создали фигуру
    {
        // Добавляем в конец списка
        pDoc->m_ShapesList.AddTail(pShape);
        // Последняя фигура становится активной
        pDoc->m_pSelShape=pShape;
        // Указываем, что документ изменен
        pDoc->SetModifiedFlag();
    }
}

void CPainterView::OnEditSelect()
{
    m_CurOper=OP_SELECT;
}

void CPainterView::MarkSelectedShape(CDC *pDC)
```

```
{
   CPainterDoc *pDoc=GetDocument();
   CRgn Rgn;
   if(pDoc->m_pSelShape==NULL) return;
   pDoc->m_pSelShape->GetRegion(Rgn);
   // Пробуем получить прямоугольник, описывающий фигуру
   CRect Rect;
   int res=Rgn.GetRgnBox(&Rect);
   if(res!= ERROR && res!=NULLREGION)
   pDC-> InvertRect(&Rect);
}

void CPainterView::OnEditAddshapePolyline()
{
   CBasePoint *pShape=new CPolygon;
   // Черная линия шириной 0,5 мм
   pShape->SetPen(RGB(0,0,0), 50, PS_GEOMETRIC);
   CPainterDoc *pDoc=GetDocument();
   // Добавляем в конец списка
   pDoc->m_ShapesList.AddTail(pShape);
   // Последняя фигура становится активной
   pDoc->m_pSelShape=pShape;
   // Указываем, что документ изменен
   pDoc->SetModifiedFlag();
   m_CurOper=OP_LINE;
}

void CPainterView::OnEditAddshapePolygon()
{
   CBasePoint *pShape=new CPolygon;
   // Темно-зеленая заливка
   pShape->SetBrush(RGB(0,100,0));
   // Черная линия шириной 0,5 мм
   pShape->SetPen(RGB(0,0,0), 50, PS_GEOMETRIC);

   // Так как pShape указатель на CBasePoint,
   // а метод SetPolygon() имеется только у класса CPolygon,
   // требуется преобразование типа указателя
   ((CPolygon*)pShape)->SetPolygon(TRUE);

   CPainterDoc *pDoc=GetDocument();
```

```
    // Добавляем в конец списка
    pDoc->m_ShapesList.AddTail(pShape);
    // Последняя фигура становится активной
    pDoc->m_pSelShape=pShape;
    // Указываем, что документ изменен
    pDoc->SetModifiedFlag();
    m_CurOper=OP_LINE;
}
void CPainterView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    CPainterDoc *pDoc=GetDocument();
    BOOL modified=FALSE;
    UINT   nMyFlags=0;
    if(pDoc->m_pSelShape!=NULL)
    modified=pDoc->m_pSelShape->OnKeyDown(nChar, nRepCnt, nFlags,
                                                          m_nMyFlags);
    switch(nChar)
    {
        case 16: m_nMyFlags=m_nMyFlags|SHIFT_HOLD; break; // Shift
        case 17: m_nMyFlags=m_nMyFlags|CTRL_HOLD; break; // Ctrl
    }
    if(modified)
    {
        // Указываем, что документ изменен
        pDoc->SetModifiedFlag();
        // Указываем, что окно надо перерисовать
        Invalidate();
    }
    CScrollView::OnKeyDown(nChar, nRepCnt, nFlags);
}

void CPainterView::OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    switch(nChar)
    {
        case 16: m_nMyFlags=m_nMyFlags^SHIFT_HOLD; break; // Shift
        case 17: m_nMyFlags=m_nMyFlags^CTRL_HOLD;  break; // Ctrl
 }
    CScrollView::OnKeyUp(nChar, nRepCnt, nFlags);
```

```
}

void CPainterView::OnEditChangeorderTop()
{
   CPainterDoc *pDoc=GetDocument();
   if(pDoc->ChangeOrder(TOP))
   {
       // Указываем, что документ изменен
       pDoc->SetModifiedFlag();
       // Указываем, что окно надо перерисовать
       Invalidate();
   }
}

void CPainterView::OnEditChangeorderStepup()
{
   CPainterDoc *pDoc=GetDocument();
   if(pDoc->ChangeOrder(STEPUP))
   {
      // Указываем, что документ изменен
      pDoc->SetModifiedFlag();
      // Указываем, что окно надо перерисовать
      Invalidate();
   }
}

void CPainterView::OnEditChangeorderStepdown()
{
   CPainterDoc *pDoc=GetDocument();
   if(pDoc->ChangeOrder(STEPDOWN))
   {
       // Указываем, что документ изменен
       pDoc->SetModifiedFlag();
       // Указываем, что окно надо перерисовать
       Invalidate();
   }
}

void CPainterView::OnEditChangeorderBottom()
```

```
{
    CPainterDoc *pDoc=GetDocument();
    if(pDoc->ChangeOrder(BOTTOM))
    {
        // Указываем, что документ изменен
        pDoc->SetModifiedFlag();
        // Указываем, что окно надо перерисовать
        Invalidate();
    }
}

void CPainterView::OnEditDelete()
{
    // TODO: Add your command handler code here
    CPainterDoc *pDoc=GetDocument();
    if(pDoc->DeleteSelected())
    {
        // Указываем, что документ изменен
        pDoc->SetModifiedFlag();
        // Указываем, что окно надо перерисовать
        Invalidate();
    }
}
```

6.11.5. Файл Shapes.h

```
// файл Shapes.h
////////////////////////////////////////////////
// Класс базовая точка
#define MOVE_STEP 100
#define ROTATE_STEP 5
#define SHIFT_HOLD  1
#define CTRL_HOLD   2

class CBasePoint: public CPoint, public CObject
{
    DECLARE_SERIAL(CBasePoint)
    CPen   m_Pen;   // перо
    CBrush m_Brush; // кисть
```

```
protected:
   // Метод сериализации
   virtual void Serialize(CArchive& ar);
   // Подготавливает контекст устройства
   virtual BOOL PrepareDC(CDC *pDC);
   // Восстанавливает контекст устройства
   virtual BOOL RestoreDC(CDC *pDC);
public:
   // Данные
   WORD     m_wSize;      // размер фигуры
   int      m_iPenStyle; // стиль линий
   int          m_iPenWidth;      // ширина линий
   COLORREF     m_rgbPenColor;    // цвет линий
   int          m_iBrushStyle;    // стиль заливки
   COLORREF     m_rgbBrushColor; // цвет заливки
   DWORD        m_dwPattern_ID;   // идентификатор шаблона заливки
public:
   // Конструкторы
   CBasePoint();                          // конструктор без параметров
   CBasePoint(int x, int y, WORD s);  // конструктор с параметрами
   ~CBasePoint(){};                       // деструктор
   // Методы
   // Отображает фигуру на экране
   virtual void Show(CDC *pDC);
   // Сообщает область захвата
   virtual void GetRegion(CRgn &Rgn);
   // Устанавливает параметры линий
   virtual BOOL SetPen(COLORREF color, int width =1, int style=PS_SOLID);
   // Устанавливает параметры заливки
   virtual BOOL SetBrush(COLORREF color, DWORD pattern =0, int style=-1);
   // Выполняет преобразование на плоскости
   virtual void Transform(const CPoint &point0,
                          double ang, int a, int b);
   // Реакция на нажатие клавиши
   virtual BOOL OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags,
                          UINT nMyFlags);

};

//////////////////////////////////////////////////////
```

```
// Класс квадрат
class CSquare: public CBasePoint
{
    DECLARE_SERIAL(CSquare)
protected:
   // Метод сериализации
   void Serialize(CArchive& ar);
public:
   // Конструкторы
   CSquare(int x, int y, WORD s);
   CSquare();
   ~CSquare(){};
// Методы
   // Отображает фигуру на экране
   void Show(CDC *pDC);
   // Сообщает область захвата
   void GetRegion(CRgn &Rgn);
};

//////////////////////////////////////////////////
// Класс полигон
class CPolygon: public CBasePoint
{
   DECLARE_SERIAL(CPolygon)
// Режим рисования TRUE – заполненный полигон, FALSE – ломаная линия
   BOOL   m_bPolygon;
protected:
   // Метод сериализации
   void Serialize(CArchive& ar);
public:
   // Динамический массив точек-вершин
   CArray <CPoint, CPoint> m_PointsArray;
   // Конструкторы
   CPolygon();
   ~CPolygon();
// Методы
   // Отображает фигуру на экране
   void Show(CDC *pDC);
   // Сообщает область захвата
```

```
   void GetRegion(CRgn &Rgn);
   // Устанавливает режим рисования полигона
   void SetPolygon(BOOL p) {m_bPolygon=p;};
   // Выполняет преобразование на плоскости
   void Transform(const CPoint &point0,
                          double ang, int a, int b);
};
```

6.11.6. Файл Shapes.cpp

```
// файл Shapes.cpp
//////////////////////////////////////
// Реализация классов
#include "stdafx.h"
#include "shapes.h"
#include "global.h"

//////////////////////////////////////
// Реализация методов класса CBasePoint
CBasePoint::CBasePoint(): CPoint(0, 0)
{
   m_wSize=1;
   m_iPenStyle=PS_SOLID;
   m_iPenWidth=1;
   m_rgbPenColor=RGB(0,0,0);
   m_iBrushStyle=-1; // не используем штриховку
   m_rgbBrushColor=RGB(0,0,0);
   m_dwPattern_ID=0; // нет шаблона заливки
};

CBasePoint::CBasePoint(int x, int y, WORD s):CPoint(x, y)
{
   m_wSize=s;
   m_iPenStyle=PS_SOLID;
   m_iPenWidth=1;
   m_rgbPenColor=RGB(0,0,0);
   m_iBrushStyle=-1; // не используем штриховку
   m_rgbBrushColor=RGB(0,0,0);
   m_dwPattern_ID=0; // нет шаблона заливки
```

```
};

IMPLEMENT_SERIAL(CBasePoint, CObject , VERSIONABLE_SCHEMA|1)
void CBasePoint::Serialize(CArchive &ar)
{
   if(ar.IsStoring()) // сохранение
   {
       // Сохраняем параметры объекта
       ar<<x;
       ar<<y;
       ar<<m_wSize;
       ar<<m_iPenStyle;
       ar<<m_iPenWidth;
       ar<<m_rgbPenColor;
       ar<<m_iBrushStyle;
       ar<<m_rgbBrushColor;
       ar<<m_dwPattern_ID;
   }
   else   // чтение
   {
       // Получили версию формата
       int Version=ar.GetObjectSchema();
       // В зависимости от версии
       // можно выполнить различные варианты загрузки
       // Загружаем параметры объекта
       ar>>x;
       ar>>y;
       ar>>m_wSize;
       ar>>m_iPenStyle;
       ar>>m_iPenWidth;
       ar>>m_rgbPenColor;
       ar>>m_iBrushStyle;
       ar>>m_rgbBrushColor;
       ar>>m_dwPattern_ID;

       SetPen(m_rgbPenColor, m_iPenWidth, m_iPenStyle);
```

```
      SetBrush(m_rgbBrushColor, m_dwPattern_ID, m_iBrushStyle );
   }
};
BOOL CBasePoint::SetPen(COLORREF color, int width /*=1*/,
                        int style/*=PS_SOLID*/)
{
   m_iPenStyle=style;
   m_iPenWidth=width;
   m_rgbPenColor=color;
   if(HPEN(m_Pen)!=NULL)    // Если перо уже существует
   if(!m_Pen.DeleteObject()) return FALSE; // удалили старое перо
   // Создаем новое перо и возвращаем результат
   return m_Pen.CreatePen( m_iPenStyle, m_iPenWidth, m_rgbPenColor);
};
BOOL CBasePoint::SetBrush(COLORREF color, DWORD pattern /*=0*/,
                          int style/*=-1*/)
{
   m_iBrushStyle=style;
   m_dwPattern_ID=pattern;
   m_rgbBrushColor=color;
   int res=1;
   if(HBRUSH(m_Brush)!=NULL)    // Если кисть уже существует
   if(!m_Brush.DeleteObject()) return FALSE; // удалили старую кисть
   if(m_dwPattern_ID>0)    // есть шаблон заливки
   {
      CBitmap Pattern;
      if(!Pattern.LoadBitmap(m_dwPattern_ID)) return FALSE;
      return m_Brush.CreatePatternBrush(&Pattern);
   }
   if(m_iBrushStyle>=0)    // указан стиль штриховки
   return m_Brush.CreateHatchBrush( m_iBrushStyle, m_rgbBrushColor);
   // Создаем сплошную кисть и возвращаем результат
   return m_Brush.CreateSolidBrush(m_rgbBrushColor);
};

BOOL CBasePoint::PrepareDC(CDC *pDC)
```

```
{
   // Сохраняем состояние контекста устройства
   if(!pDC->SaveDC()) return FALSE;
   // Устанавливаем перо и кисть
   if(HPEN(m_Pen)!=NULL)
   pDC->SelectObject(&m_Pen);
   if(HBRUSH(m_Brush)!=NULL)
   pDC->SelectObject(&m_Brush);
   return TRUE;
};

BOOL CBasePoint::RestoreDC(CDC *pDC)
{
   // Восстанавливаем состояние контекста устройства
   return pDC->RestoreDC(-1);
};

void CBasePoint::Show(CDC* pDC)
{
   // Устанавливаем перо и кисть
   PrepareDC(pDC);
   // Рисуем кружок, обозначающий точку
   pDC->Ellipse(x-m_wSize, y-m_wSize, x+m_wSize, y+m_wSize);
   // Восстанавливаем контекст
   RestoreDC(pDC);
}

void CBasePoint::GetRegion(CRgn &Rgn)
{
   Rgn.CreateEllipticRgn(x-m_wSize, y-m_wSize, x+m_wSize, y+m_wSize);
}

void CBasePoint::Transform(const CPoint &point0,
                           double ang, int a, int b)
{
   CPoint res=::Transform(CPoint(x, y), CPoint(0,0), 0, a, b);
```

```
    x=res.x; y=res.y;
};

BOOL CBasePoint::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags,
                           UINT nMyFlags)
{
    BOOL res=TRUE;
    if(nMyFlags & SHIFT_HOLD) //поворот
    switch(nChar)
    {
        case 37:
            Transform(CPoint(0,0), -ROTATE_STEP, 0, 0);
            break;
        case 39:
            Transform(CPoint(0,0), ROTATE_STEP, 0, 0);
            break;
        default:
            res=FALSE;
    }
    else // перенос
    switch(nChar)
    {
        case 38: // вверх
            Transform(CPoint(0,0), 0, 0, MOVE_STEP);
            break;
        case 40: // вниз
            Transform(CPoint(0,0), 0, 0, -MOVE_STEP);
            break;
        case 37: // влево
            Transform(CPoint(0,0), 0, -MOVE_STEP, 0);
            break;
        case 39: // вправо
            Transform(CPoint(0,0), 0, MOVE_STEP, 0);
            break;
        default:
            res=FALSE;
```

```
    }
    return res;
}

////////////////////////////////////////
// Реализация методов класса CSquare

CSquare::CSquare(int x, int y, WORD s): CBasePoint(x, y, s)
{
    m_wSize=s;
}

CSquare::CSquare(): CBasePoint()
{
    m_wSize=40;
}

IMPLEMENT_SERIAL(CSquare, CObject, 1)
void CSquare::Serialize(CArchive &ar)
{
    CBasePoint::Serialize(ar);
}
void CSquare::Show(CDC* pDC)
{
    int s=m_wSize/2;
    // Устанавливаем перо и кисть
    PrepareDC(pDC);
    // Рисуем квадрат
    pDC->Rectangle(x-s, y-s, x+s, y+s);
    // Восстанавливаем контекст
    RestoreDC(pDC);
}

void CSquare::GetRegion(CRgn &Rgn)
{
    int s=m_wSize/2;
    Rgn.CreateRectRgn(x-s, y-s, x+s, y+s);
```

```
}

//////////////////////////////////////
// Реализация методов класса CPolygon

CPolygon::CPolygon(): CBasePoint()
{
   m_wSize=0;
   m_bPolygon=FALSE;
}

CPolygon::~CPolygon()
{
   m_PointsArray.RemoveAll( );
}

IMPLEMENT_SERIAL(CPolygon, CObject, 1)
void CPolygon::Serialize(CArchive &ar)
{
   if(ar.IsStoring()) // сохранение
   {
       // Сохраняем параметры объекта
       ar<<m_bPolygon;
   }
   else// чтение
   {
      // Получили версию формата
      int Version=ar.GetObjectSchema();
      // В зависимости от версии
      // можно выполнить различные варианты загрузки
      // Загружаем параметры объекта
      ar>>m_bPolygon;
   }
   m_PointsArray.Serialize(ar);
   CBasePoint::Serialize(ar);
}
Void CPolygon::Show(CDC* pDC)
```

```
{
   // Устанавливаем перо и кисть
   PrepareDC(pDC);
   // Рисуем
   if(m_bPolygon)
      pDC->Polygon(m_PointsArray.GetData(), m_PointsArray.GetSize());
   else
      pDC->Polyline( m_PointsArray.GetData(), m_PointsArray.GetSize());
   // Восстанавливаем контекст
   RestoreDC(pDC);
}

void CPolygon::GetRegion(CRgn &Rgn)
{
   Rgn. CreatePolygonRgn(m_PointsArray.GetData(),
                         m_PointsArray.GetSize(), ALTERNATE);
}

void CPolygon::Transform(const CPoint &point0, double ang, int a, int b)
{
   for(int i=0; i<m_PointsArray.GetSize(); i++)
      m_PointsArray[i]=::Transform(m_PointsArray[i],
                                   m_PointsArray[0], ang, a, b);
};
```

6.11.7. Файл Global.h

```
// файл Global.h
////////////////////////////////////////
// Прототипы глобальных функций
CPoint Transform(const CPoint &point, const CPoint &point0, double ang,
int a, int b);
```

6.11.8. Файл Global.cpp

```
// файл Global.cpp
////////////////////////////////////////
```

```
// Реализация глобальных функций
#include "stdafx.h"
#include "global.h"
#include <math.h>

CPoint Transform(const CPoint &point,
                 const CPoint &point0, double ang, int a, int b)
{
   CPoint res;
   // Перевод в радианы
   ang=ang*atan(1.0)/45.0;
   res.x=(int)floor(point0.x+(point.x-point0.x)*cos(ang)-
                              (point.y-point0.y)*sin(ang)+a+0.5);
   res.y=(int)floor(point0.y+(point.x-point0.x)*sin(ang)+
                    (point.y-point0.y)*cos(ang)+b+0.5);
   return res;
};
```

6.12. Заключение

Текст программы приведен на компакт-диске в каталоге \Sources\Painter3.

Теперь, когда у нас имеются средства редактирования, откроем рисунок \Pics\Painter\Грузовик.pr2, заменим груду ящиков в его кузове на новогодние елки, дополним рисунок деталями, в общем, насладимся новыми возможностями нашей программы (рис. 6.8).

Многие полезные команды редактирования остались нереализованными.

Что еще можно реализовать:

- пару команд Copy и Paste — для этого нужно описать в классах фигур конструкторы копирования;
- операции выбора цвета линий и заливки для фигур — специальный диалог или панель инструментов;
- операции "отменить" и "повторить" — каким-то образом надо запоминать состояние рисунка и (или) выполняемые операции.

Можно также развивать удобство интерфейса программы, например сделать контекстное меню, которое вызывалось бы при нажатии правой кнопки мыши.

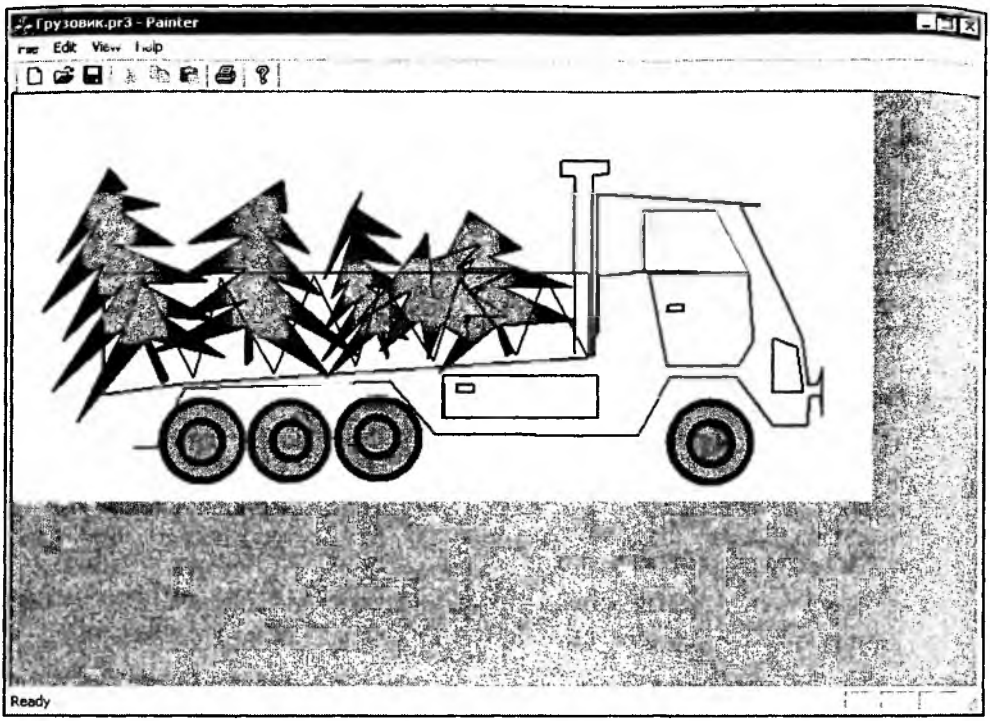
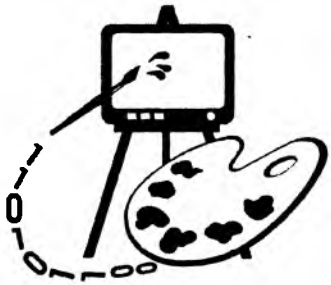


Рис. 6.8. Результат редактирования рисунка Грузовик.pr2

В общем, неограниченный простор для творчества. Однако, поскольку эти вопросы непосредственно не связаны с предметом книги, мы их рассматривать не будем. Вместо этого в следующей главе мы перейдем к рассмотрению преобразований в трехмерном пространстве.

Глава 7

Преобразования в трехмерном пространстве



Ну, это все довольно просто,
лишь бы осилить теорию...

Роберт Шекли "Обмен разумов"

В этой главе рассматриваются:

- преобразования в трехмерном пространстве;
- параллельная и перспективная проекции;
- основные подходы к решению задачи удаления невидимых элементов изображений;
- программная реализация преобразований в трехмерном пространстве (программа Painter 4);
- построение трехмерной поверхности $z = f(x, y)$;
- построение линий уровня на поверхности.

7.1. Перенос и поворот в трехмерном пространстве

Переносом в трехмерном пространстве называют преобразование точки $P(x, y, z)$ в точку $P'(x', y', z')$ в соответствии с системой уравнений:

$$\begin{cases} x' = x + a_1; \\ y' = y + a_2; \\ z' = z + a_3, \end{cases}$$

где a_1, a_2, a_3 — константы.

В матричном виде данная операция будет выглядеть следующим образом:

$$[x', y', z', 1] = [x, y, z, 1]T, \quad T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix}. \tag{7.1}$$

Поворот вокруг координатных осей может быть записан и без использования однородных координат. Для краткости записи так и поступим. В правой координатной системе (рис. 7.1) поворот вокруг оси z на угол α описывается следующей матрицей преобразования:

$$[x', y', z', 1] = [x, y, z, 1]R_z, \quad R_z = \begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{7.2}$$

где $c = \cos\alpha$ и $s = \sin\alpha$.

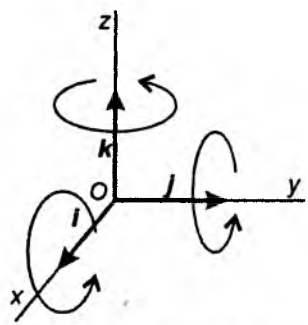


Рис. 7.1. Правая система координат

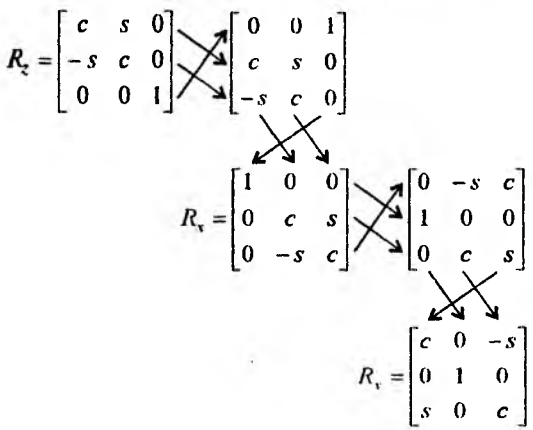


Рис. 7.2. Схема преобразования матриц

Для построения матриц поворота вокруг осей x и y (матриц R_x и R_y) можно использовать матрицу R_z. Данные матрицы получаются путем циклического переноса строк и столбцов по следующей схеме (рис. 7.2): $R_z \to R_x \to R_y \to R_z$.

При необходимости изменения координатной системы используют инвертированные матрицы:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a_1 & -a_2 & -a_3 & 1 \end{bmatrix}, \qquad R_z^{-1} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$R_x^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix}, \qquad R_y^{-1} = \begin{bmatrix} c & 0 & s \\ 0 & 1 & 0 \\ -s & 0 & c \end{bmatrix}. \qquad (7.3)$$

7.2. Параллельная проекция

Для выполнения преобразований необходимы точка наблюдения, объект и экран. Экран находится между наблюдателем и объектом (рис. 7.3). Если камера (глаз) находится в точке E, то для каждой точки P объекта прямая PE пересекает экран в точке P'. Систему координат, в которой определяется положение объекта, положение точки наблюдения и экрана, а также размеры экрана, будем называть *мировой*. Задача заключается в преобразовании мировых координат множества точек $P(x, y, z)$, принадлежащих объекту, в координаты точек изображения на экране $P'(X, Y)$, где (x, y, z) — мировые координаты точки P объекта, а (X, Y) — экранные координаты ее проекции (точки P'). Преобразование координат включает в себя этапы, схематично изображенные на рис. 7.4.

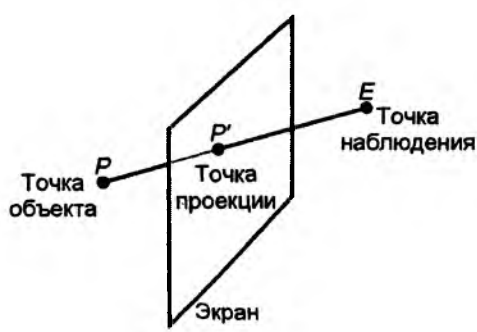


Рис. 7.3. Проекция точки объекта на экран

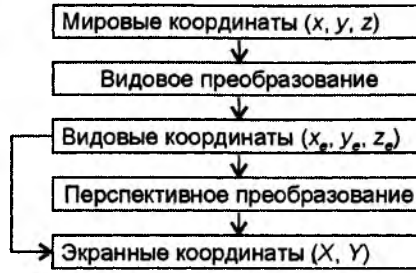


Рис. 7.4. Схема преобразования координат

Сначала мировые координаты преобразовываются в видовые координаты (с началом в точке E). Затем может быть выполнено перспективное преобразование, добавляющее эффект перспективы в зависимости от расстояния от объекта до экрана и расстояния от точки наблюдения до экрана. При построении параллельной проекции перспективное преобразование не выполняется, и видовые координаты (x_e, y_e) точки P используются в качестве экранных координат (X, Y) точки P'.

7.2.1. Видовое преобразование

Пусть система мировых координат правая, и ее начало, точка O, совпадает с центром объекта. Точка E задана в сферических координатах (ρ, θ, φ) относительно точки O:

$$x_E = \rho \sin\varphi \cos\theta,\ y_E = \rho \sin\varphi \sin\theta,\ z_E = \rho \cos\varphi. \qquad (7.4)$$

Вектор EO определяет направление наблюдения (рис. 7.5).

Система видовых координат показана на рис. 7.6. Кроме положения в пространстве, она отличается от мировых координат тем, что является левосторонней (мировая — правосторонняя). В матричном виде запись преобразования мировых координат в видовые координаты будет такой:

$$[x_e, y_e, z_e, 1] = [x, y, z, 1]V. \qquad (7.5)$$

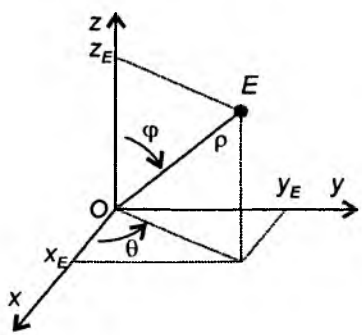


Рис. 7.5. Полярные координаты точки E

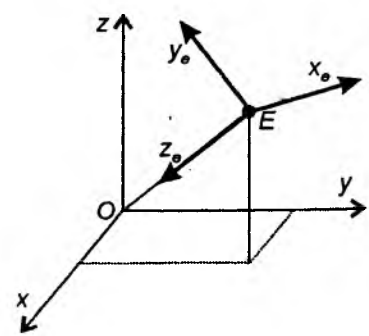


Рис. 7.6. Система видовых координат

Для получения матрицы V требуется перемножение матриц четырех элементарных преобразований. При этом выполняются следующие действия:

1. Перенос из точки O в точку E (рис. 7.7). Матрица данного преобразования:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_e & -y_e & -z_e & 1 \end{bmatrix}. \qquad (7.6)$$

2. Поворот координатной системы вокруг оси z на угол $\frac{\pi}{2} - \theta$ (рис. 7.7, 7.8).

В результате ось y совпадет по направлению с горизонтальной составляющей вектора OE, а ось x будет перпендикулярна этой составляющей. Так как поворот выполняется в *отрицательном* направлении, матрица данного преобразования будет совпадать с матрицей поворота *точки* на такой же угол в *положительном* направлении:

$$R_z = \begin{bmatrix} \cos(\pi/2-\theta) & \sin(\pi/2-\theta) & 0 \\ -\sin(\pi/2-\theta) & \cos(\pi/2-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \sin\theta & \cos\theta & 0 \\ -\cos\theta & \sin\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \qquad (7.7)$$

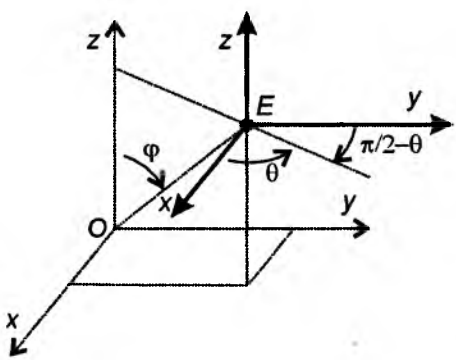


Рис. 7.7. Перенос в точку E

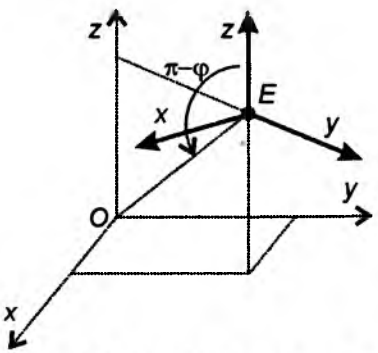


Рис. 7.8. Поворот вокруг оси z

3. Поворот системы вокруг оси x в положительном направлении на угол $\pi - \varphi$, что соответствует повороту точки на угол $-(\pi - \varphi) = \varphi - \pi$ (рис. 7.8, 7.9). Матрица данного преобразования будет такой:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi-\pi) & \sin(\varphi-\pi) \\ 0 & -\sin(\varphi-\pi) & \cos(\varphi-\pi) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & -\cos\varphi \end{bmatrix}. \qquad (7.8)$$

4. Изменение направления оси x: $x' = -x$ (рис. 7.10). Матрица данного преобразования:

$$M_{yz} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{7.9}$$

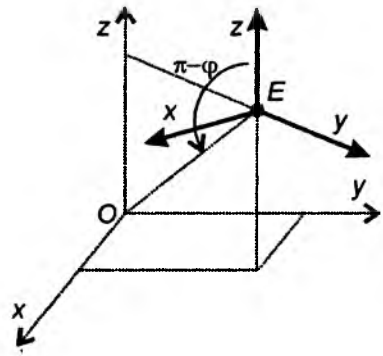


Рис. 7.9. Поворот вокруг оси x

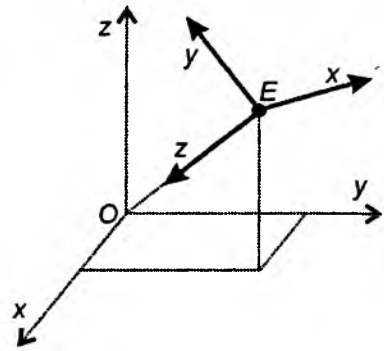


Рис. 7.10. Изменение направления оси x

Матрицу V найдем путем перемножения матриц (7.6) — (7.9):

$$V = TR_z^* R_x^* M_{yz}^* \tag{7.10}$$

(* — означает расширение матрицы 3 × 3 до размера 4 × 4 путем добавления строки и столбца [0, 0, 0, 1]).

Результат перемножения (7.10) — матрица преобразования мировых координат в видовые координаты:

$$V = \begin{bmatrix} -\sin\theta & -\cos\varphi\cos\theta & -\sin\varphi\cos\theta & 0 \\ \cos\theta & -\cos\varphi\sin\theta & -\sin\varphi\sin\theta & 0 \\ 0 & \sin\varphi & -\cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.11}$$

Видовые координаты точки находятся путем перемножения ее мировых координат (в однородной записи) на матрицу V (7.11).

Видовые координаты (x_e, y_e) — ортогональная (параллельная) проекция точки $P(x, y, z)$. Координаты (x_e, y_e) можно непосредственно использовать для формирования изображения на экране.

7.2.2. Перспективные преобразования

Рассмотрим рис. 7.11. Пусть координата y точки P равна нулю. Из подобия треугольников ΔEPO и $\Delta EP'Q$ следует:

$$\frac{P'Q}{EQ}=\frac{PO}{EO}\Rightarrow\frac{X}{d}=\frac{x}{z}\Rightarrow X=d\frac{x}{z}. \qquad (7.12,\ a)$$

Аналогично для Y найдем:

$$Y=d\frac{y}{z}. \qquad (7.12,\ б)$$

Так как ось z совпадает с EO — направлением взгляда на точку O — центр объекта, то начало системы экранных координат будет находиться в точке Q, в которой EO пересекает экран. Для помещения объекта в центр экрана можно дополнить выражения (7.12) соответствующим смещением:

$$X=d\frac{x}{z}+\frac{W}{2},\quad Y=d\frac{y}{z}+\frac{H}{2}, \qquad (7.13)$$

где W, H — ширина и высота экрана соответственно.

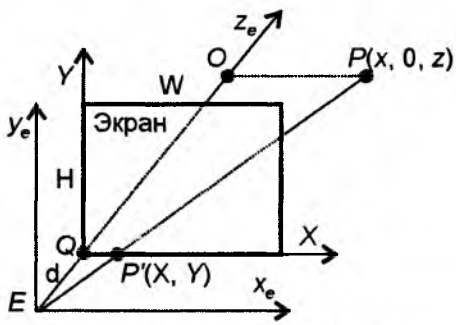


Рис. 7.11. Перспективное преобразование

7.3. Два основных подхода к удалению невидимых линий и поверхностей

Для построения более-менее реалистичного изображения трехмерных сцен необходимо уметь удалять невидимые части объектов (ребра и грани). Существует два основных подхода к решению данной задачи.

Первый подход заключается в непосредственном сравнении объектов друг с другом для выяснения того, какие части объектов являются видимыми. В данном случае работа ведется в пространстве объектов. Этот подход используется в алгоритмах, рассмотренных в *разд. 7.3.1* и *7.3.2*.

Второй подход заключается в определении для каждого пиксела экрана ближайшего к нему объекта (вдоль направления проецирования). При этом работа ведется в пространстве экранных координат. Этот подход используется в алгоритмах, рассмотренных в *разд. 7.3.3, 7.3.4* и *7.3.5.*

7.3.1. Алгоритм отсечения нелицевых граней

Пусть для каждой грани некоторой фигуры задан единичный вектор внешней нормали. Если вектор нормали грани составляет с направлением проецирования (направлением взгляда на объект) тупой угол, то такая грань не может быть видна и называется нелицевой. В случае, когда данный угол является острым, грань видна и называется лицевой (рис. 7.12).

В случае, когда трехмерная сцена представляет собой один выпуклый многогранник, удаление нелицевых граней полностью решает задачу удаления невидимых граней.

В общем случае описанная проверка не решает задачу полностью, но позволяет значительно сократить количество рассматриваемых граней.

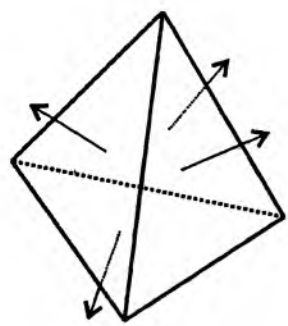


Рис. 7.12. Определение нелицевых граней

7.3.2. Алгоритм Робертса

Требуется, чтобы каждая грань была выпуклым многогранником. Поскольку такое условие наиболее просто обеспечивается для граней объекта в виде треугольников, то первым делом выполняется триангуляция — разбиение граней на треугольники. Далее составляется список граней (треугольников) и список ребер. Затем проверяется видимость ребер многогранника путем тестирования их на перекрытие гранями.

Сначала отбрасываются все ребра, обе определяющие грани которых являются нелицевыми. Оставшиеся ребра проверяются на перекрытие гранями. Возможны следующие случаи:

- грань не закрывает ребро (рис. 7.13) — переход к проверке перекрытия следующей гранью;

- грань полностью закрывает ребро (рис. 7.14) — на этом проверка видимости ребра заканчивается;

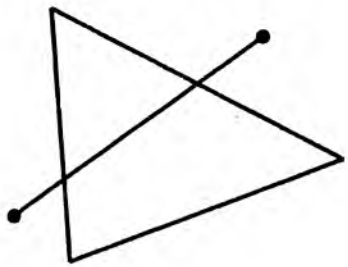


Рис. 7.13. Грань не закрывает ребро

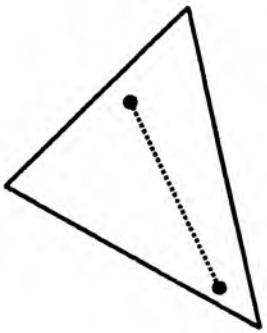


Рис. 7.14. Грань полностью закрывает ребро

- грань частично закрывает ребро (рис. 7.15) — в этом случае ребро разбивается на несколько частей, само ребро удаляется из списка, но в список добавляются те его части, которые не перекрываются гранью.

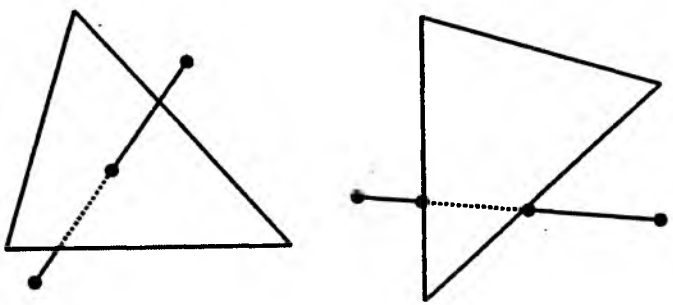


Рис. 7.15. Варианты частичного перекрытия ребра гранью

Количество проверок можно значительно сократить, если воспользоваться принципом "разделяй и властвуй". Идея заключается в следующем. Экран разделяется на несколько равных частей. Для каждой части составляется список граней, в который заносятся только те грани, проекции которых попадают в эту часть экрана. При определении видимости ребра сначала устанавливается, в какие части экрана попадает его проекция, и далее осуществляется проверка на перекрытие лишь гранями, содержащимися в списках данных частей.

7.3.3. Алгоритм *z*-буфера

Каждому пикселу экрана сопоставляется расстояние до проецируемого на него объекта (*z*-буфер). Для вывода на экран произвольной грани она сначала переводится в свое растровое представление на экране, и для каждого

пиксела определяется его "глубина". В случае, если это значение меньше значения, хранящегося в z-буфере (изначально $+\infty$), его значение заносится в z-буфер. В конце концов, рисуются лишь пикселы-проекции ближайших к экрану объектов.

В связи с простотой и одновременно значительной трудоемкостью данного алгоритма распространены его аппаратные реализации.

Этот алгоритм используется в библиотеке OpenGL для решения задачи загораживания при работе с невыпуклыми объектами. Алгоритм применяется в примере использования OpenGL, приведенном в *главе 12*.

7.3.4. Алгоритм Варнака

Алгоритм основан на разделении экрана на части (рис. 7.16). Экран делится сначала на 4 равные части. При этом возможны следующие ситуации:

1. Часть экрана полностью накрывается проекцией ближайшей грани.
2. Часть экрана не накрывается проекцией ни одной грани.
3. Ни первое, ни второе условия не выполняются.

В первом случае часть экрана полностью закрашивается цветом грани. Во втором случае — цветом фона. В третьем случае данная часть разбивается еще на 4 части, для каждой из которых вновь выполняется проверка. Разбиение можно продолжать, пока размер части не будет соответствовать одному пикселу. Если разбиение дошло до одного пиксела, то пиксел закрашивается цветом ближайшей к нему грани.

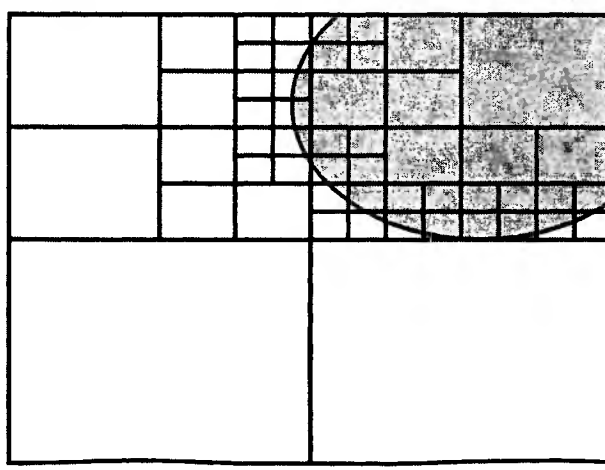


Рис. 7.16. Деление экрана на части

7.3.5. Алгоритм построчного сканирования

Допустим, что все изображение на экране представляет собой набор вертикальных линий (столбцов пикселов). Рассмотрим сечение трехмерной сцены плоскостью, проходящей через такую линию, и центр проекции (точку наблюдения). Результатом такого сечения будет набор отрезков, которые необходимо спроецировать на экран. Исходная задача свелась к удалению невидимых отрезков на каждой линии.

Данный алгоритм может быть использован при визуализации перемещения по лабиринту (рис. 7.17). В случае, если расстояние между полом и потолком одинаково по всей сцене, а стены вертикальны, задача может рассматриваться как двумерная.

Проведем через точку наблюдения и столбец пикселов экрана прямую линию. Видимым будет ближайшее пересечение со стенами лабиринта. Решая задачу в двумерном пространстве, определяем расстояние до ближайшей стены. Изображение в каждом столбце пикселов может состоять из трех частей: пола, стены и потолка. Часть линии закрашиваем цветом пола, часть — цветом стены, часть — цветом потолка. В зависимости от расстояния между стеной и точкой наблюдения можно менять интенсивность цветов.

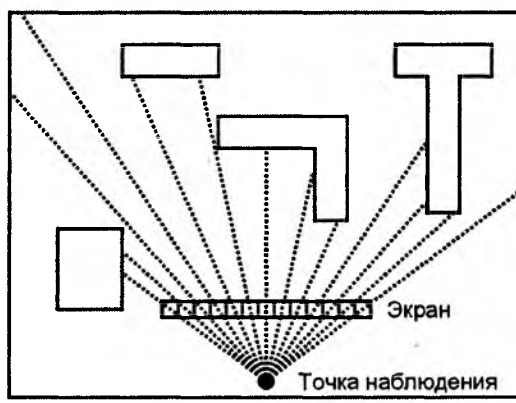


Рис. 7.17. Вид сверху на лабиринт

7.4. Программная реализация преобразований в трехмерном пространстве

Реализуем рассмотренный материал на практике — добавим в проект Painter возможности создания и манипуляции трехмерными объектами. Это будет версия 4 программы Painter.

Для задания точки в трехмерном пространстве определим структуру данных POINT3D, а для задания положения точки наблюдения и параметров построения проекции — структуру Perspective (листинг 7.1).

Листинг 7.1. Структуры данных для работы с трехмерными фигурами. Файл Shapes.h

```
struct POINT3D
{
   double x, y, z;
};

// Параметры трехмерной сцены
struct  Perspective
{
   POINT3D O; // Точка, вокруг которой выполняем поворот
   double rho, theta, phi, // Полярные координаты точки наблюдения (E)
          d;   // Расстояние от E до экрана
// 1 - вкл. перспективные преобразования; 0 - выкл.
   WORD with_perspective;
   LONGdx, dy; // Смещение проекции на экране
};
```

Будем представлять трехмерные фигуры в виде "проволочного каркаса". Каждая "проволочка" будет являться полигоном в трехмерном пространстве — объектом класса C3DPolygon. Класс C3DPolygon определим, как производный от класса CPolygon (листинг 7.2). Для хранения трехмерных координат фигуры в классе C3DPolygon определим динамический массив точек m_3DPointsArray. От класса CPolygon новый класс унаследует массив двумерных точек m_PointsArray и методы для работы с ним. В этом массиве будем сохранять экранные координаты фигуры, т. е. координаты проекции точек из массива m_3DPointsArray на экран. Для того чтобы рассчитать экранные координаты, введем метод MakeProjection(). Отображать же фигуру на экране будет унаследованный метод CPolygon::Show().

Листинг 7.2. Интерфейс класса C3DPolygon. Файл Shapes.h

```
class C3DPolygon: public CPolygon
{
   DECLARE_SERIAL(C3DPolygon)
 protected:    // Метод сериализации
```

```
    void Serialize(CArchive& ar);
  public:
    // Конструкторы
    C3DPolygon(){};
    ~C3DPolygon(){};
// Данные
    // Динамический массив точек-вершин в мировых координатах
    CArray <POINT3D, POINT3D> m_3DPointsArray;
// Методы
    // Добавить точку
    void AddPoint(POINT3D point) {m_3DPointsArray.Add(point);};
    // Расчет экранных координат
    void MakeProjection(Perspective P);
};
```

Для того чтобы собрать все "проволочки" — объекты класса `C3DPolygon` — в одну фигуру, определим класс `C3DShape`, в котором будем хранить список указателей на трехмерные полигоны (листинг 7.3).

Листинг 7.3. Интерфейс класса C3DShape. Файл Shapes.h

```
class C3DShape: public CBasePoint
{
    DECLARE_SERIAL(C3DShape)
 protected: // Виртуальный метод сериализации
   void Serialize(CArchive& ar);
 public:
   // Конструкторы
   C3DShape();
   ~C3DShape();
// Данные
   Perspective m_Percpective; // параметры обзора
   // список указателей на полигоны
   CTypedPtrList<CObList, C3DPolygon*> m_PtrPolygonList;
// Методы
   // Расчет проекции
   void MakeProjection();
   // Отображение фигуры на экране
   void Show(CDC *pDC);
```

```
    // Сообщает область захвата
    void GetRegion(CRgn &Rgn);
    // Добавить полигон
    void AddPolygon(C3DPolygon *pPolygon);
    // Реакция на нажатие клавиши
    BOOL OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags, UINT nMyFlags);
};
```

Кроме списка трехмерных полигонов, в классе `C3DShape` определены также параметры наблюдения фигуры — переменная `m_Percpective`. На основе этих параметров в методе `MakeProjection()` выполняется расчет экранных координат каждого полигона из списка `m_PtrPolygonList`. Поэтому если рисунок содержит несколько трехмерных фигур, на каждую из них можно смотреть по-разному. Если такая ситуация не устраивает, то можно определить единые параметры наблюдения для всего рисунка (трехмерной сцены) и передавать их в качестве параметра в метод `C3DShape::MakeProjection()` для каждой трехмерной фигуры сцены.

Рассмотрим теперь, как реализованы методы новых классов.

Все, что потребовалось сделать в функции сохранения/загрузки `C3DPolygon::Serialize()` трехмерного полигона, — это вызвать метод базового класса `CPolygon` (листинг 7.4).

Листинг 7.4. Метод `C3DPolygon::Serialize()`. Файл Shapes.cpp

```
IMPLEMENT_SERIAL(C3DPolygon, CPolygon, -1)
void C3DPolygon::Serialize(CArchive &ar)
{
    CPolygon::Serialize(ar);
    if(ar.IsStoring())
    {   }
    else
    {   }
    m_3DPointsArray.Serialize(ar);
};
```

Расчет экранных координат выполняется в методе `C3DPolygon::MakeProjection()` так: сначала рассчитываются коэффициенты матрицы преобразования (7.11), затем для всех точек из массива `m_3DPointsArray` находятся их экранные проекции и выполняется перспективное преобразование (листинг 7.5).

Листинг 7.5. Метод C3DPolygon::Serialize(). Файл Shapes.cpp

```
void C3DPolygon::MakeProjection(Perspective P)
{
   // Перевод в радианы
   P.theta=P.theta*atan(1.0)/45.0; P.phi=P.phi*atan(1.0)/45.0;
   // Расчет коэффициентов матрицы преобразования
   // Если установлен режим отображения MM_TEXT,
   // при котором начало координат в верхнем левом углу,
   // требуется лишь заменить знак у коэффициентов второго столбца
   // матрицы преобразования на противоположный
   // (перевернуть ось Y)
   double st=sin(P.theta), ct=cos(P.theta), sp=sin(P.phi), cp=cos(P.phi),
          v11=-st,   v12=-cp*ct,   v13=-sp*ct,
          v21=ct,    v22=-cp*st,   v23=-sp*st,
                     v32=sp,       v33=-cp,
          v41=P.dx,  v42=P.dy,     v43=P.rho;
   double x, y, z;
   double TempZ=0;
   // Расчет видовых координат точек
   m_PointsArray.SetSize(m_3DPointsArray.GetSize());
   for(int i=0; i<m_3DPointsArray.GetSize(); i++)
   {
       x=m_3DPointsArray[i].x-P.O.x;
       y=m_3DPointsArray[i].y-P.O.y;
       z=m_3DPointsArray[i].z-P.O.z;
       TempZ=v13*x+v23*y+v33*z+v43;
       m_PointsArray[i].x=(LONG)(v11*x+v21*y+v41+0.5);
       m_PointsArray[i].y=(LONG)(v12*x+v22*y+v32*z+v42+0.5);
       // Перспективные преобразования
       if(P.with_perspective)
       {
           m_PointsArray[i].x=(LONG)(P.d*m_PointsArray[i].x/TempZ +0.5);
           m_PointsArray[i].y=(LONG)(P.d*m_PointsArray[i].y/TempZ +0.5);
       }
       m_PointsArray[i].x+=(LONG)(P.O.x +0.5);
       m_PointsArray[i].y+=(LONG)(P.O.y +0.5);
   }
};
```

В конструкторе трехмерного объекта определяем положение точки наблюдения и расстояние до экрана (см. рис. 7.5, 7.11), а в деструкторе освобождаем память, занятую фигурой (листинг 7.6.).

Листинг 7.6. Конструктор класса C3DShape. Файл Shapes.cpp

```
C3DShape::C3DShape(): CBasePoint()
{
   m_Percpective.O.x=0;
   m_Percpective.O.y=0;
   m_Percpective.O.z=0;
   m_Percpective.rho=50000; // 50 см в режиме MM_HIMETRIC
   m_Percpective.theta=30;
   m_Percpective.phi=30;
   m_Percpective.d=25000; // 25 см в режиме MM_HIMETRIC
   m_Percpective.with_perspective=TRUE;
   m_Percpective.dx=0;
   m_Percpective.dy=0;
}

C3DShape::~C3DShape()
{
   while(m_PtrPolygonList.GetCount()>0)
       delete m_PtrPolygonList.RemoveHead();
};
```

Метод `C3DShape::Serialize()` трехмерной фигуры (листинг 7.7) выполняет сохранение/загрузку параметров наблюдения и вызывает метод `Serialize()` для списка трехмерных полигонов.

Листинг 7.7. Метод C3DShape::Serialize(). Файл Shapes.cpp

```
IMPLEMENT_SERIAL(C3DShape, CBasePoint, -1)
void C3DShape::Serialize(CArchive &ar)
{
   if(ar.IsStoring())
   {
       ar << m_Percpective.O.x;
       ar << m_Percpective.O.y;
       ar << m_Percpective.O.z;
```

```
        ar << m_Percpective.rho;
        ar << m_Percpective.theta;
        ar << m_Percpective.phi;
        ar << m_Percpective.d;
        ar << m_Percpective.with_perspective;
        ar << m_Percpective.dx;
        ar << m_Percpective.dy;
    }
    else
    {
        ar >> m_Percpective.O.x;
        ar >> m_Percpective.O.y;
        ar >> m_Percpective.O.z;
        ar >> m_Percpective.rho;
        ar >> m_Percpective.theta;
        ar >> m_Percpective.phi;
        ar >> m_Percpective.d;
        ar >> m_Percpective.with_perspective;
        ar >> m_Percpective.dx;
        ar >> m_Percpective.dy;
    }
    m_PtrPolygonList.Serialize(ar);
};
```

Метод `C3DShape::Show()` трехмерной фигуры (листинг 7.8) вызывает одноименный метод для всех полигонов, образующих фигуру.

Листинг 7.8. Метод `C3DShape::Show()`. Файл Shapes.cpp

```
void C3DShape::Show(CDC *pDC)
{
    // Вывод всех полигонов
    POSITION Pos=NULL;
    if(m_PtrPolygonList.GetCount()>0)
        Pos=m_PtrPolygonList.GetHeadPosition();
    while(Pos!=NULL)
        m_PtrPolygonList.GetNext(Pos)->Show(pDC);
};
```

Метод `C3DShape::GetRegion()` трехмерной фигуры (листинг 7.9) конструирует прямоугольный регион, охватывающий проекцию фигуры на экран.

Листинг 7.9. Метод C3DShape::GetRegion(). Файл Shapes.cpp

```
void C3DShape::GetRegion(CRgn &Rgn)
{
   // Конструируем область захвата C3DShape
   // в виде прямоугольника, охватывающего изображение
   // фигуры на экране
   CRect Frame; // охватывающий прямоугольник
   POSITION Pos=NULL;
   int i=0;
   CPolygon *pPolygon=NULL;
   if(m_PtrPolygonList.GetCount()>0)
      Pos=m_PtrPolygonList.GetHeadPosition();
   // Инициализируем прямоугольник значениями
   // первой точки первого полигона
   if(Pos!=NULL && (pPolygon=m_PtrPolygonList.GetAt(Pos))!=NULL &&
      pPolygon->m_PointsArray.GetSize()>0)
   {
       Frame.left=Frame.right=pPolygon->m_PointsArray[0].x;
       Frame.top=Frame.bottom=pPolygon->m_PointsArray[0].y;
   }
   else return;
   // Получаем габариты фигуры
   while(Pos!=NULL)
   {
       pPolygon=m_PtrPolygonList.GetNext(Pos);
       for(i=0; i<pPolygon->m_PointsArray.GetSize(); i++)
       {
           if(pPolygon->m_PointsArray[i].x<Frame.left)
             Frame.left=pPolygon->m_PointsArray[i].x;
           if(pPolygon->m_PointsArray[i].x>Frame.right)
              Frame.right=pPolygon->m_PointsArray[i].x;
           if(pPolygon->m_PointsArray[i].y>Frame.bottom)
              Frame.bottom=pPolygon->m_PointsArray[i].y;
           if(pPolygon->m_PointsArray[i].y<Frame.top)
              Frame.top=pPolygon->m_PointsArray[i].y;
       };
   }
   // Создаем область
```

```
    Rgn.CreateRectRgn(Frame.left, Frame.top, Frame.right, Frame.bottom);
}
```

Метод `C3DShape::AddPolygon()` предназначен для добавления новых "проволочек", образующих трехмерную фигуру, в список полигонов `m_PtrPolygonList`. В этом методе выполняется также расчет центра фигуры, вокруг которого затем будут производиться трехмерные преобразования (листинг 7.10).

Листинг 7.10. Метод `C3DShape::AddPolygon()`. Файл Shapes.cpp

```
void C3DShape::AddPolygon(C3DPolygon *pPolygon)
{
   m_PtrPolygonList.AddTail(pPolygon); // добавили в список
   // расчет центра
   POSITION Pos=NULL;
   C3DPolygon* pCurPolygon=NULL;
   WORD Count=0, i=0;
   if(m_PtrPolygonList.GetCount()>0)
      Pos=m_PtrPolygonList.GetHeadPosition();
   while(Pos!=NULL)
   {
       pCurPolygon=(C3DPolygon*)m_PtrPolygonList.GetNext(Pos);
       for(i=0; i<pCurPolygon->m_3DPointsArray.GetSize(); i++)
       {
           m_Percpective.O.x+=pCurPolygon->m_3DPointsArray[i].x;
           m_Percpective.O.y+=pCurPolygon->m_3DPointsArray[i].y;
           m_Percpective.O.z+=pCurPolygon->m_3DPointsArray[i].z;
       }
       Count+=i;
   }
   m_Percpective.O.x/=Count;
   m_Percpective.O.y/=Count;
   m_Percpective.O.z/=Count;
};
```

Метод `C3DShape::MakeProjection()` совсем прост, он только и делает, что вызывает одноименный метод для каждого трехмерного полигона, которому передает параметры наблюдения фигуры (листинг 7.11). Этот метод требуется вызывать, как только фигура создана (перед первым выводом фигуры на экран), а также каждый раз после того, как изменятся параметры наблюдения фигуры.

Листинг 7.11. Метод C3DShape::MakeProjection(). Файл Shapes.cpp

```
void C3DShape::MakeProjection()
{
   POSITION Pos=NULL;
   if(m_PtrPolygonList.GetCount()>0)
   Pos=m_PtrPolygonList.GetHeadPosition();
   while(Pos!=NULL)
      ((C3DPolygon*)m_PtrPolygonList.GetNext(Pos))->
                                          MakeProjection(m_Percpective);
};
```

Мы переопределили метод OnKeyDown() в классе C3DShape, для того чтобы иметь возможность изменять параметры наблюдения фигуры (листинг 7.12). В этом методе мы задействовали проверку состояния клавиши <Ctrl>: если она нажата, то клавишами <↑>, <↓> можно изменить расстояние до экрана, на который выполняется проецирование фигуры.

Листинг 7.12. Метод C3DShape::OnKeyDown(). Файл Shapes.cpp

```
BOOL C3DShape::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags, UINT nMyFlags)
{
   BOOL res=TRUE;
   if(nMyFlags & SHIFT_HOLD)
   switch(nChar)
   {
       // Up, точка наблюдения выше
       case 38: m_Percpective.phi-=ROTATE_STEP; break;
       // Down, точка наблюдения ниже
       case 40: m_Percpective.phi+=ROTATE_STEP; break;
       // Left, точка наблюдения левее
       case 37: m_Percpective.theta-=ROTATE_STEP; break;
       // Right, точка наблюдения правее
       case 39: m_Percpective.theta+=ROTATE_STEP; break;
       default: res=FALSE;
   }
   else
   if(nMyFlags & CTRL_HOLD)
   switch(nChar)
   {
```

```
        case 38: m_Percpective.d+=MOVE_STEP; break; // Up, экран дальше
        case 40: m_Percpective.d-=MOVE_STEP; break; // Down, экран ближе
        default: res=FALSE;
    }
    else // Перенос
    {
        switch(nChar)
        {
            case 38: m_Percpective.dy+= MOVE_STEP; break; // вверх
            case 40: m_Percpective.dy-= MOVE_STEP; break; // вниз
            case 37: m_Percpective.dx-= MOVE_STEP; break; // влево
            case 39: m_Percpective.dx+= MOVE_STEP; break; // вправо
            // Клавиша <P>, вкл/выкл перспективные преобразования
            case 80:
          m_Percpective.with_perspective=!m_Percpective.with_perspective;
                break;
            default:res=FALSE;
        }
    }
    if(res)
       // Расчет проекции
       MakeProjection();
    return res;
};
```

7.5. Рисуем трехмерную поверхность

Ну что же, нам осталось только добавить в интерфейс программы Painter команды рисования какой-нибудь трехмерной фигуры. В целом этот процесс ничем не отличается от внедрения команд рисования рассмотренных ранее фигур, поэтому подробно на нем останавливаться не станем. Отличие заключается в том, что нам требуется сконструировать трехмерную фигуру из отдельных полигонов. Можно, конечно, написать специальные классы для создания конкретных фигур (куб, шар и т. д.). Однако для иллюстрации достаточно ввести в класс `CPainterView` специальную функцию (листинг 7.13). Предлагаем нарисовать трехмерную поверхность. Поэтому функцию назовем `AddSurface()`. Эта функция вызывается из метода `CPainterView::AddShape()` (листинг 7.14). В качестве параметров функция `AddSurface()` принимает точку и размер. Поверхность рассчитывается как функция `ZFunction()` на сетке из `_GRID_DENSITY*_GRID_DENSITY` узлов

в квадрате со стороной size*2, с центром в точке first_point. Функция ZFunction() определена в файле Global.cpp (листинг 7.15). Поверхность строится из полигонов-"проволочек", параллельных оси X и оси Y и отображается на экране с учетом параметров наблюдения, заданных в конструкторе 3DShape.

Листинг 7.13. Метод CPainterView::AddSurface(). Файл PainterView.cpp

```
const int _GRID_DENSITY=30;
CBasePoint* CPainterView::AddSurface(CPoint first_point, int size)
{
   C3DShape *pShape=NULL;
   pShape=new C3DShape();
   // Рассчитываем поверхность в заданной области
   double dx=(double)size*2/_GRID_DENSITY,
   dy=(double)size*2/_GRID_DENSITY;
   // Создаем 3D-объект – поверхность, как набор 3D-полигонов
   POINT3D point3d;
   C3DPolygon *p3DPolygon=NULL;
   for(int i=0, j=0; i<_GRID_DENSITY; i++)
   {
       p3DPolygon=new C3DPolygon();
       for(j=0; j<_GRID_DENSITY; j++)
       {
           point3d.x=first_point.x+dx*i - size;
           point3d.y=first_point.y+dy*j - size;
           point3d.z=ZFunction(fabs(first_point.x-point3d.x),
                               fabs(first_point.y-point3d.y));
           p3DPolygon->AddPoint(point3d);
       }
       pShape->AddPolygon(p3DPolygon);
   }

   for(j=0; j<_GRID_DENSITY; j++)
   {
       p3DPolygon=new C3DPolygon();
       for(i=0; i<_GRID_DENSITY; i++)
       {
```

```
            point3d.x=first_point.x+dx*i - size;
            point3d.y=first_point.y+dy*j - size;
            point3d.z=ZFunction(fabs(first_point.x-point3d.x),
                        fabs(first_point.y-point3d.y));
            p3DPolygon->AddPoint(point3d);
        }
    pShape->AddPolygon(p3DPolygon);
    }
    pShape->MakeProjection();
    return pShape;
};
```

Листинг 7.14. Метод `CPainterView::AddShape()`. Файл PainterView.cpp

```
void CPainterView::AddShape(int shape, CPoint first_point, CPoint second_point)
{
    CPainterDoc *pDoc=GetDocument();
    CBasePoint *pShape=NULL;
    // Расчет размера
    int size=0;
    size=(int) floor( sqrt((second_point.x-first_point.x)*
                           (second_point.x-first_point.x)+
                           (second_point.y-first_point.y)*
                           (second_point.y-first_point.y)) +0.5);
    switch(shape)
    {
        case OP_LINE:
            break;
        case OP_POINT:
            // Создаем объект-точку
            pShape=new CBasePoint(second_point.x, second_point.y, 100);
            // Светло-серая заливка
            pShape->SetBrush(RGB(200,200,200));
            break;
        case OP_CIRCLE:
            // Создаем объект-круг
            pShape=new CBasePoint(first_point.x, first_point.y, size);
            // Черная линия шириной 2 мм
            pShape->SetPen(RGB(0,0,0), 200, PS_GEOMETRIC);
```

```
            // Темно-серая заливка
            pShape->SetBrush(RGB(100,100,100));
            break;
         case OP_SQUARE:
            // Создаем объект-квадрат
            pShape=new CSquare(first_point.x, first_point.y, size*2);
            // Красная линия шириной 1 мм
            pShape->SetPen(RGB(200,0,0), 100, PS_GEOMETRIC);
            // Темно-серая диагональная штриховка
            pShape->SetBrush(RGB(100,100,100),0,HS_DIAGCROSS);
            break;
         case OP_SURFACE:
            // Создаем объект-поверхность
            pShape=AddSurface(first_point, size);
            break;
      }
      if(pShape!=NULL) // Создали фигуру
      {
         // Добавляем в конец списка
         pDoc->m_ShapesList.AddTail(pShape);
        // Последняя фигура становится активной
        pDoc->m_pSelShape=pShape;
        // Указываем, что документ изменен
        pDoc->SetModifiedFlag();
      }
}
```

Листинг 7.15. Функция ZFunction(). Файл Global.cpp

```
double ZFunction(double x, double y)
{
   return (x*x+y*y)/10000;
};
```

Работа программы с функцией рисования поверхностей показана на рис. 7.18.

Для иллюстрации эффекта перспективы на рис. 7.19 и 7.20 показаны две трехмерные плоскости ($z = 10\,000$) с включенным и выключенным режимами расчета перспективных преобразований.

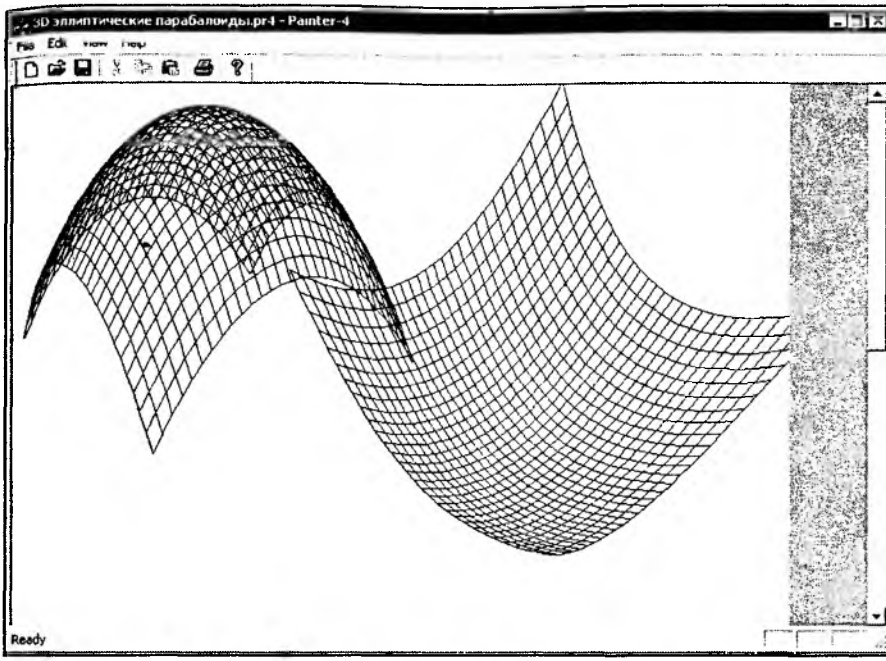


Рис. 7.18. Рисунок с трехмерными поверхностями

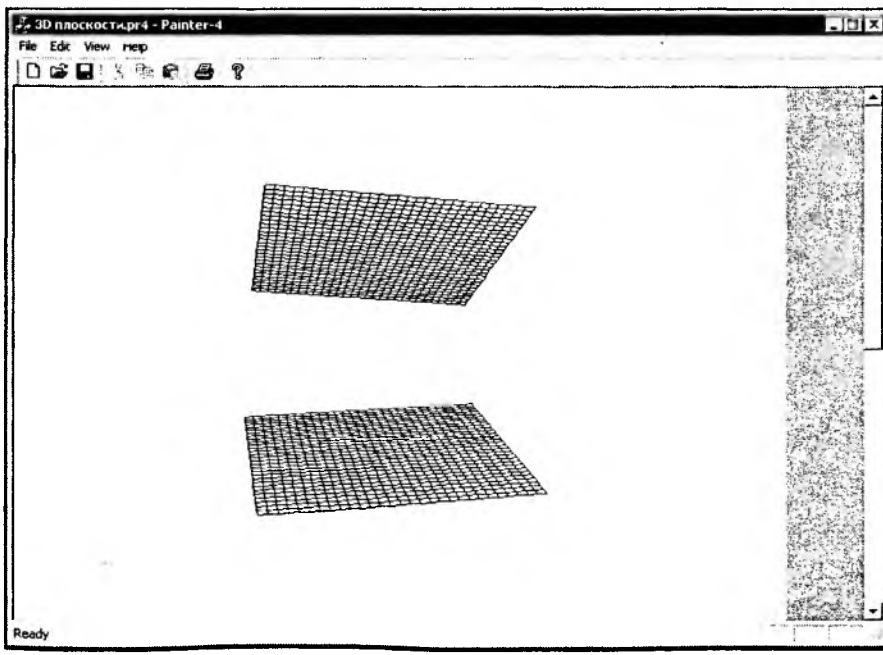


Рис. 7.19. Плоскости с включенным режимом расчета перспективных преобразований

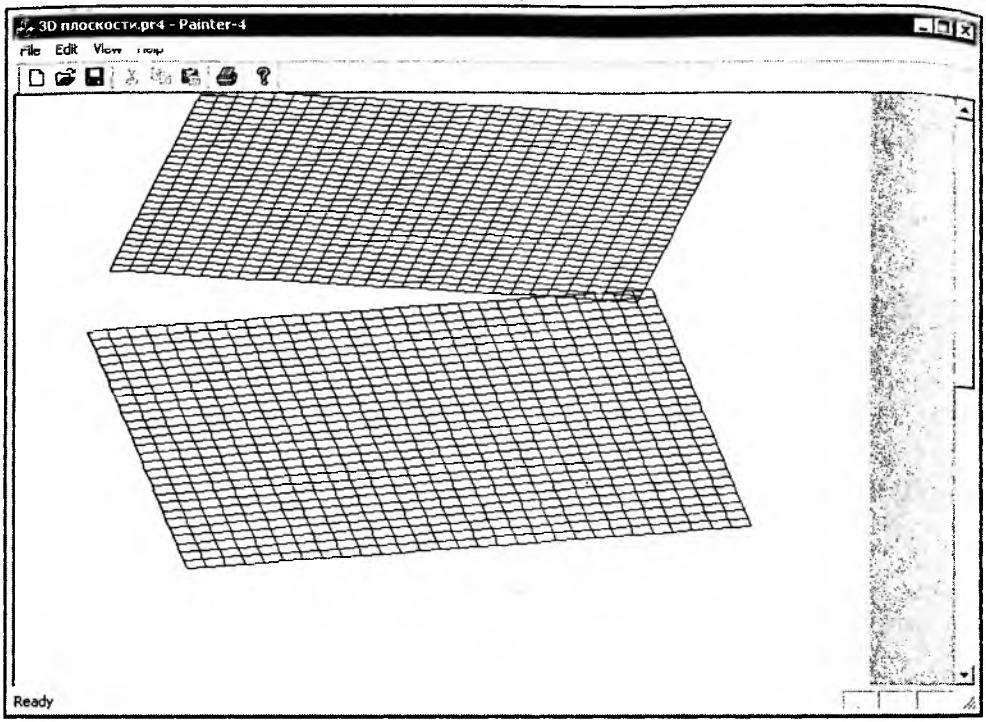


Рис. 7.20. Плоскости с выключенным режимом расчета перспективных преобразований

7.5.1. Построение линий уровня на поверхности

При визуализации данных, полученных экспериментальным или расчетным путем, часто встречается задача построения линий уровня на поверхностях. В частности, линии уровня знакомы нам по географическим картам, на которых они служат для обозначения высоты местности над уровнем моря.

Используя класс трехмерных фигур, мы тоже вполне можем решить данную задачу. Все, что для этого нам потребуется, — это завести еще несколько функций, и немного модифицировать метод `CPainterView::AddSurface()`.

Пусть поверхность некоторой функции $z = f(x, y)$ задана массивом значений $z(x, y)$, рассчитанных на сетке $x = (X_{min}, ..., X_{max})$, $y = (Y_{min}, ..., Y_{max})$ (рис. 7.21). Для нахождения линии уровня L требуется найти пересечение поверхности $z = f(x, y)$ с плоскостью $z = L$.

В этом случае общий подход к построению линии уровня заключается в следующем:

1. Выполняется триангуляция поверхности, каждая ячейка сетки разбивается на два треугольника (рис. 7.22).

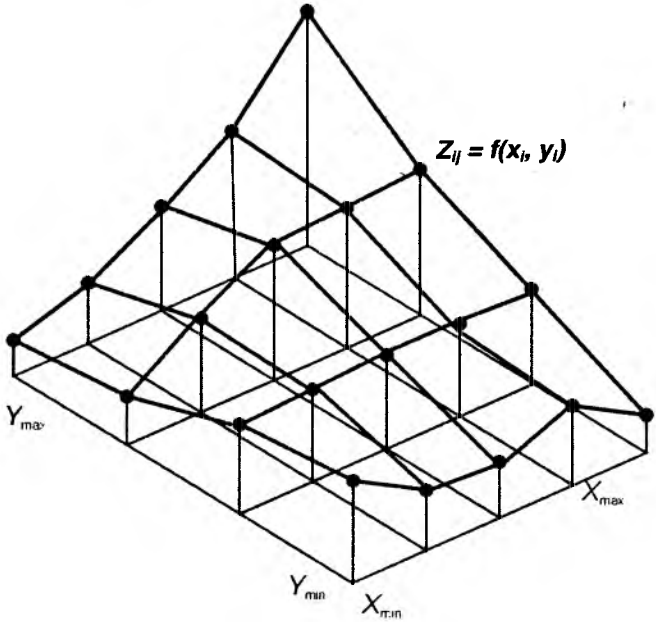


Рис. 7.21. Поверхность $z = f(x, y)$

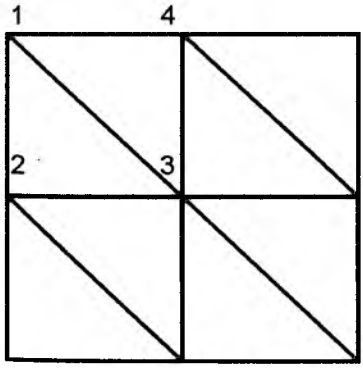


Рис. 7.22. Триангуляция поверхности

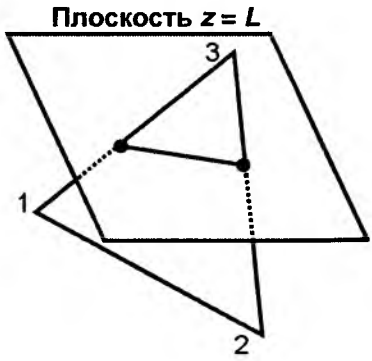


Рис. 7.23. Пересечение треугольника с плоскостью $z = L$

2. Для каждого треугольника находится пересечение с плоскостью L (рис. 7.23).

Этот метод достаточно прост и позволяет получить хорошее изображение линии уровня. Надо отметить, что линия уровня может быть с разрывами. Отдельные составные части линии будем называть *сегментами*. Общая схема построения линии уровня показана на рис. 7.24.

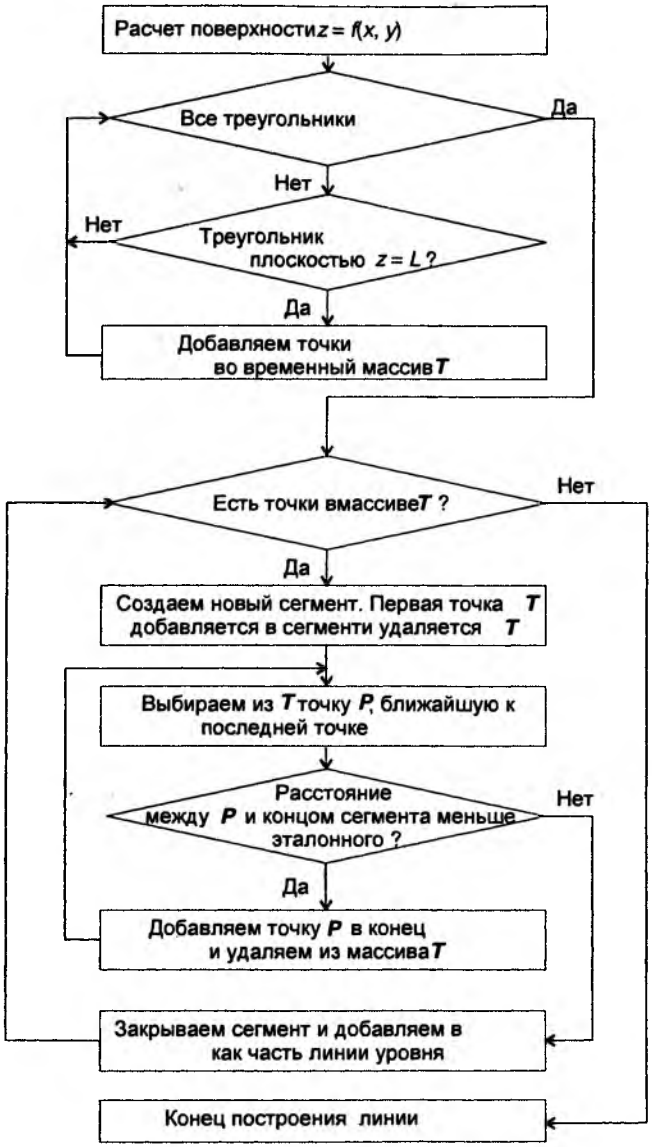


Рис. 7.24. Схема построения линии уровня

Для реализации описанного подхода несколько модифицируем метод `CPainterView::AddSurface()` таким образом, чтобы значения узлов сетки поверхности запоминались в массиве, который затем будем использовать для построения поверхности (листинг 7.16). Кроме того, будем определять минимальное и максимальное значения z, которые затем используем для расчета нескольких промежуточных уровней.

Листинг 7.16. Модифицированный метод CPainterView::AddSurface(). Файл PainterView.cpp

```
const int _GRID_DENSITY=30;
const int _LEVELS_DENSITY=5;
CBasePoint* CPainterView::AddSurface(CPoint first_point, int size)
{
  C3DShape *pShape=NULL;
  pShape=new C3DShape();

  // Рассчитываем поверхность в заданной области
  // dx, dy — шаг сетки
  double dx=(double)size*2/_GRID_DENSITY,
         dy=(double)size*2/_GRID_DENSITY;
  // Массив точек поверхности используется для временного хранения
  POINT3D point3d[_GRID_DENSITY*_GRID_DENSITY];
  // Создаем 3D-объект—поверхность, как набор 3D-полигонов
  C3DPolygon *p3DPolygon=NULL;
  // Добавляем "проволочки" вдоль оси Y
  for(int i=0, j=0; i<_GRID_DENSITY; i++)
  {
      p3DPolygon=new C3DPolygon();
      for(j=0; j<_GRID_DENSITY; j++)
      {
          point3d[j*_GRID_DENSITY+i].x=first_point.x+dx*i - size;
          point3d[j*_GRID_DENSITY+i].y=first_point.y+dy*j - size;
          point3d[j*_GRID_DENSITY+i].z=
          ZFunction(fabs(first_point.x-point3d[j*_GRID_DENSITY+i].x),
                    fabs(first_point.y-point3d[j*_GRID_DENSITY+i].y));
          p3DPolygon->AddPoint(point3d[j*_GRID_DENSITY+i]);
      }
      pShape->AddPolygon(p3DPolygon);
  }
  double minz=point3d[0].z, maxz=point3d[0].z;
  // Добавляем "проволочки" вдоль оси X
  for(j=0; j<_GRID_DENSITY; j++)
  {
      p3DPolygon=new C3DPolygon();
      for(i=0; i<_GRID_DENSITY; i++)
```

```
        {
            p3DPolygon->AddPoint(point3d[j*_GRID_DENSITY+i]);
            // Определяем пределы изменения Z
            if(point3d[j*_GRID_DENSITY+i].z<minz)
               minz=point3d[j*_GRID_DENSITY+i].z;
            if(point3d[j*_GRID_DENSITY+i].z>maxz)
               maxz=point3d[j*_GRID_DENSITY+i].z;
        }
        pShape->AddPolygon(p3DPolygon);
    }
    // Строим линии уровня
    double l_step=(maxz-minz)/_LEVELS_DENSITY;
    int color_step=200/_LEVELS_DENSITY; // изменение цвета
    for(i=0; i<_LEVELS_DENSITY; i++)
    AddRsection(pShape, point3d, _GRID_DENSITY, _GRID_DENSITY,
      minz + l_step/2 + l_step*i, RGB(0,0, 55+color_step*i));
    // Рассчитать проекцию на экран
    pShape->MakeProjection();
    return pShape;
};
```

В методе `CPainterView::AddSurface()` вызывается функция `AddRsection()`. Каждый вызов этой функции добавляет в объект класса `C3DShape` линию заданного уровня. В качестве аргументов в функцию передаются: указатель на объект, в который будет добавлена линия уровня, массив узлов сетки, задающих поверхность, количество узлов сетки, значение уровня и цвет. Внутри функции для каждого треугольника находится его пересечение с плоскостью $z = L$. Точки пересечения каждого треугольника с плоскостью добавляются в общий временный массив. Этот массив затем сортируется, и из него формируются один или несколько (так как линия уровня может быть прерывистая) трехмерных полигонов — объектов класса `C3DPolygon`. Функция `AddRsection()`, функция нахождения пересечения треугольника с плоскостью `AddTriangleSection()`, а также функции `CutCross()` для расчета координат пересечения отрезка с плоскостью и `Dist()` для расчета расстояния между двумя точками приведены в листинге 7.17. Данные функции являются глобальными, их прототипы определены в файле Shapes.h.

Листинг 7.17. Функции для построения линий уровня. Файл Shapes.cpp

```
int AddRsection(C3DShape *pShape, POINT3D *pSur, int x_size, int y_size,
double level, COLORREF color)
{
```

```
if(x_size<2 || y_size<2) return 0;
// Полигон для временного хранения точек линии уровня
C3DPolygon *pTempPolygon=new C3DPolygon();
if(pTempPolygon==NULL) return 0;
// Разбиваем поверхность на треугольники и пробуем найти пересечение
// для каждого треугольника и плоскости level.
// Точки пересечения добавляем в pTempPolygon
for(int x=0, y=0; y<y_size-1; y++)
   for(int x=0;  x<x_size-1; x++)
   {
       AddTriangleSection(pTempPolygon, &pSur[y*x_size+x],
              &pSur[(y+1)*x_size+x+1], &pSur[y*x_size+x+1], level);
       AddTriangleSection(pTempPolygon, &pSur[y*x_size+x],
              &pSur[(y+1)*x_size+x], &pSur[(y+1)*x_size+x+1], level);
   }
// Из полученного набора точек создаем аккуратные полигончики
// Для упрощения работы с точками получим ссылку на данные
// Это, конечно, не лучшая иллюстрация принципов ООП, зато удобно :)
CArray <POINT3D, POINT3D> &TempPointsArray=
                                    pTempPolygon->m_3DPointsArray;
int pos=0, posmin=0;
POINT3D EndSegPoint;
double D=0, dcur=0, dmin=0; // расстояние между точками
C3DPolygon *pSeg;
BOOL fContinueSeg=TRUE; // флаг "продолжить текущий сегмент"
// Вычисляем эталонное расстояние между точками -
// диагональ сетки на плоскости
POINT3D P1=pSur[0], P2=pSur[x_size+1];  P1.z=P2.z=0;
D=Dist(&P1, &P2);
// Пока во временном массиве осталась хотя бы пара точек,
// создаем из массива сегменты сечения
while(TempPointsArray.GetSize()-1>0)
{
    // Новый сегмент - полигон
    pSeg=new C3DPolygon(); fContinueSeg=TRUE;
    if(pSeg==NULL) return 0;
    // Установим цвет
    pSeg->SetPen(color);
    // Первая точка - начало и конец сегмента
    pSeg->AddPoint(TempPointsArray[0]);
```

```
EndSegPoint=TempPointsArray[0];
// Удаляем точку из общего массива точек
TempPointsArray.RemoveAt(0);
// Продолжаем полигон
while(fContinueSeg )
{
   posmin=0;
   dmin=D*2;
   // С начала массива
   // выбираем ближайшую к концу сегмента точку
   for(pos=0; pos<TempPointsArray.GetSize(); pos++)
   {
       dcur=Dist(&EndSegPoint, &TempPointsArray[pos]);
       if(dcur<dmin) // Запоминаем позицию(номер) ближайшей точки
           {dmin=dcur; posmin=pos;}
   }
   if(dmin<=D) // Расстояние до ближайшей точки меньше эталонного,
   {
       // но все-таки точка не совпадает с концом сегмента,
       // поэтому добавим ее в сегмент
       if(dmin>D/1000)
       {
           // Ближайшую точку в сегмент
           pSeg->AddPoint(TempPointsArray[posmin]);
           // Новая точка становится концом сегмента
           EndSegPoint=TempPointsArray[posmin];
       }
       // Удаляем эту точку
       TempPointsArray.RemoveAt(posmin);
   }
   else // не нашли близкой к концу точки - закрываем сегмент
      fContinueSeg=FALSE;
};
// Проверим, может стоит замкнуть сегмент
if(pSeg->m_3DPointsArray.GetSize()>2)
   if(Dist(&pSeg->m_3DPointsArray[0],
     &pSeg->m_3DPointsArray[pSeg->m_3DPointsArray.GetSize()-1])<D)
      pSeg->AddPoint(pSeg->m_3DPointsArray[0]);

// Добавляем полигон в фигуру
```

```
      pShape->AddPolygon(pSeg);
   }
   // Временный полигон нам больше не нужен
   delete pTempPolygon;
   return 1;
}

void AddTriangleSection(C3DPolygon *p3DPolygon, POINT3D *pP1,
                        POINT3D *pP2, POINT3D *pP3, double level)
{
   int  f1,f2,f3;
   double x1,x2,x3,y1,y2,y3;
   POINT3D P1, P2;
   if(!((pP1->z==level)&&(pP2->z==level)&&(pP3->z==level)) &&
      !((pP1->z>level)&&(pP2->z>level)&&(pP3->z>level)) &&
      !((pP1->z<level)&&(pP2->z<level)&&(pP3->z<level)) )
   if((pP1->z==level)&&(pP2->z==level)) // сторона в плоскости – добавляем
   {
      p3DPolygon->AddPoint(*pP1);
      p3DPolygon->AddPoint(*pP2);
   }
   else
   if((pP2->z==level)&&(pP3->z==level)) // сторона в плоскости – добавляем
   {
       p3DPolygon->AddPoint(*pP2);
       p3DPolygon->AddPoint(*pP3);
   }
   else
   if((pP3->z==level)&&(pP1->z==level)) // сторона в плоскости – добавляем
   {
       p3DPolygon->AddPoint(*pP3);
       p3DPolygon->AddPoint(*pP1);
   }
   else
   {   // Находим пересечение каждой стороны треугольника с плоскостью
       f1=CutCross(level,pP1, pP2, x1, y1);
       f2=CutCross(level,pP2, pP3, x2, y2);
       f3=CutCross(level,pP3, pP1, x3, y3);
       if(f1&&f2)
```

```
        {
            P1.x=x1; P1.y=y1; P1.z=level;
            P2.x=x2; P2.y=y2; P2.z=level;
            p3DPolygon->AddPoint(P1);
            p3DPolygon->AddPoint(P2);
        }
        if(f2&&f3)
        {
            P1.x=x2; P1.y=y2; P1.z=level;
            P2.x=x3; P2.y=y3; P2.z=level;
            p3DPolygon->AddPoint(P1);
            p3DPolygon->AddPoint(P2);
        }
        if(f1&&f3)
        {
            P1.x=x1; P1.y=y1; P1.z=level;
            P2.x=x3; P2.y=y3; P2.z=level;
            p3DPolygon->AddPoint(P1);
            p3DPolygon->AddPoint(P2);
        }
    }// ~else
}

int CutCross(double level, POINT3D *pP1, POINT3D *pP2,
             double &x, double &y)
{
    if( (pP1->z<level && pP2->z<level) || // отрезок под плоскостью level
       (pP1->z>level && pP2->z>level) ||  // отрезок над плоскостью level
       (pP1->z==pP2->z)  )                // отрезок в плоскости level
      {x=pP1->x; y=pP1->y; return 0;}
    else
    {
        x=pP2->x-(pP1->x-pP2->x)*(level-pP2->z)/(pP2->z-pP1->z);
        y=pP2->y-(pP1->y-pP2->y)*(level-pP2->z)/(pP2->z-pP1->z);
        return 1;
    }
}

double Dist(POINT3D *pP1, POINT3D* pP2)
```

```
{
   if(pP1==NULL||pP2==NULL) return 0;
   return sqrt(pow(pP1->x-pP2->x, 2)+
               pow(pP1->y-pP2->y, 2)+
               pow(pP1->z-pP2->z, 2));

};
```

7.6. Заключение

Текст программы находится на компакт-диске в каталоге Sources\Painter4.

Рисунки с поверхностями можно найти на диске в каталоге Pics\Painter файлы: "Этюд с поверхностями.pr4", "Поверхности из линий уровня.pr4" и др. Просмотреть и изменить эти рисунки можно с помощью программы Painter версии 4.

Надо отметить, что приведенная реализация построения поверхности далеко неоптимальна. Достаточно лишь того, что каждый узел поверхности дублируется — это, соответственно, удваивает все расчеты и расходуемый объем памяти. Поэтому если построение поверхностей — важная составляющая вашего приложения, целесообразно придумать более удачную реализацию. Можно начать хотя бы с того, что сконструировать свой класс для работы с поверхностями, глобальные функции, обеспечивающие построение линий уровня сделать членами этого класса. Для линий уровня добавить подписи значений уровней. В результате можно получить средство для визуализации результатов численного моделирования (рис. 7.25).

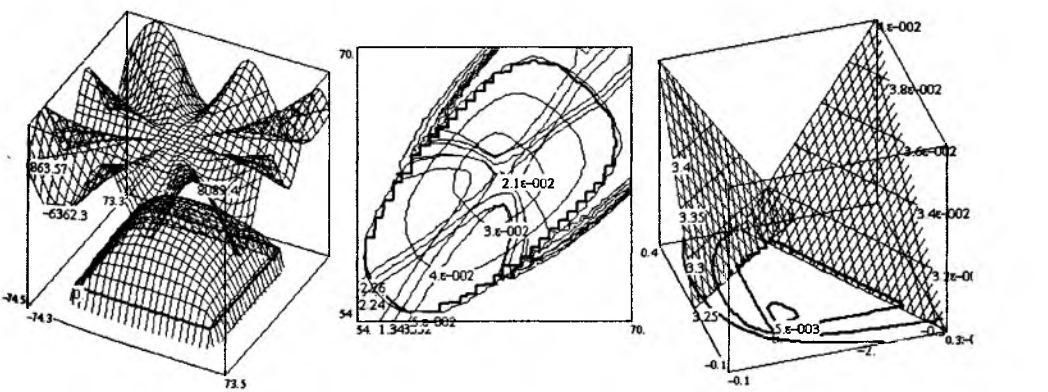


Рис. 7.25. Использование поверхностей и линий уровня для визуализации расчетных данных в программе проектирования технических устройств

Поэкспериментируйте с различными функциями $z = f(x, y)$ — это может оказаться интересным. Что хорошо, так это то, что в нашей программе уже

реализованы методы по сохранению, загрузке трехмерных объектов. Поэтому программа Painter 4 умеет сохранять рисунки с поверхностями. Один из таких шедевров приведен на рис. 7.26.

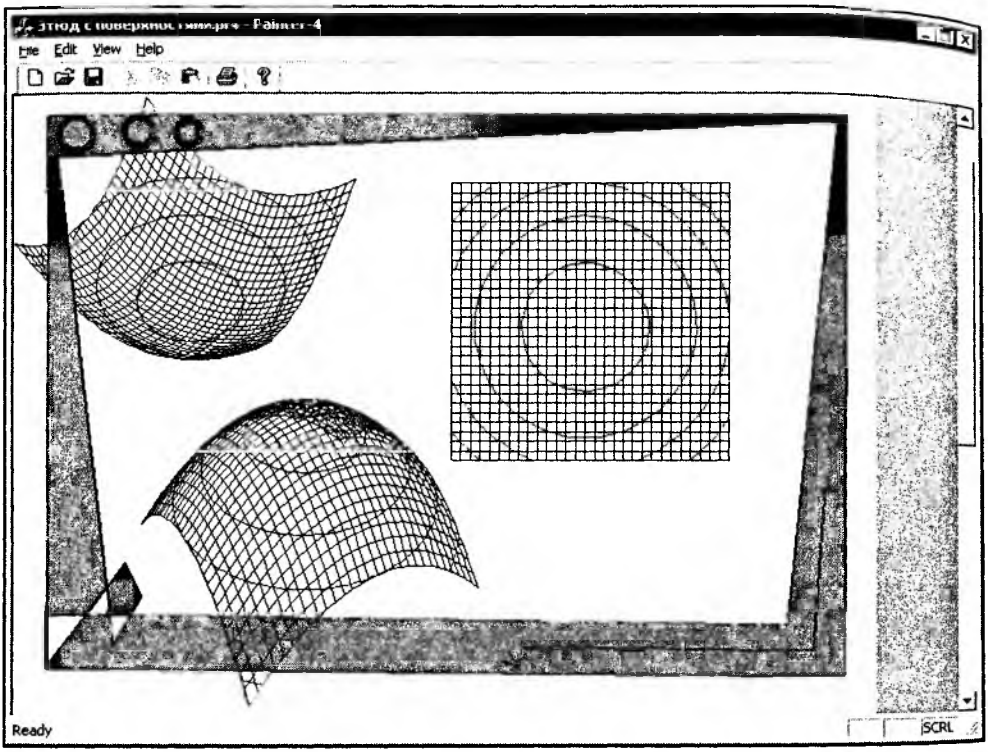


Рис. 7.26. Творение в Painter 4

Дополнительно можно порекомендовать изучить литературу [20], в этой книге в *главе 12* приводится несколько интересных алгоритмов визуализации данных.

Глава 8

Построение кривых

Ломаная кривая. Нет! Ломаная прямая. Нет! Ломаная линия. Во!

Студенческий фольклор

В этой главе рассматриваются:

- определения;
- параметрическое задание кривых;
- некоторые сплайновые кривые;
- программная реализация построения сплайновых кривых (программа Bezier).

Задачи построения кривой, соединяющей набор базовых точек, возникают во многих областях народного хозяйства. Например, бравые капитаны планируют на картах дальних морей маршруты своих судов (порты в данном случае вполне могут играть роль базовых точек). Отважные экспериментаторы проводят свои опыты и получают графики экспериментальных зависимостей, которые затем используют для доказательства своих теоретических предпосылок, а может и в иных, неизвестных нам целях. Талантливые дизайнеры, наметив точками контуры будущего изделия, придают своим эскизам законченные формы, соединяя точки линиями. Короче говоря, на своем жизненном пути кривые приходится рисовать практически всем. Более того, сам жизненный путь мало у кого бывает прямым. Поэтому рассмотрим некоторые практические приемы построения кривых.

В данной главе мы коснемся лишь прикладного аспекта данной проблемы. Для более подробного ознакомления с математическими основами построения кривых и поверхностей можно порекомендовать книги [13, 14].

8.1. Определения

Сначала дадим несколько общих определений.

- *Сплайн* — кривая, удовлетворяющая некоторым критериям гладкости.
- *Базовые (опорные) точки* — набор точек, на основе которых выполняется построение кривой.
- *Интерполяция* — построение кривой, точно проходящей через набор базовых точек.
- *Аппроксимация* — сглаживание, приближение, т. е. построение гладкой кривой, проходящей не через набор базовых точек, а вблизи них.
- *Экстраполяция* — построение линии за пределами интервала, заданного набором базовых точек.

В простейшем случае интерполяция может быть реализована путем соединения базовых точек отрезками прямых линий (рис. 8.1), этот способ называется *линейной интерполяцией*. Такая кривая точно проходит через набор базовых точек и отлично подходит, например, для иллюстрации динамики курса валют. Однако если этот набор базовых точек получен в результате некоторого эксперимента, то линейная интерполяция может не очень точно отражать поведение объекта эксперимента на интервалах между базовыми точками. Кроме того, такая интерполяция часто неудовлетворительна с эстетической точки зрения. Поэтому более приемлемой может быть интерполяция с помощью некоторой гладкой кривой.

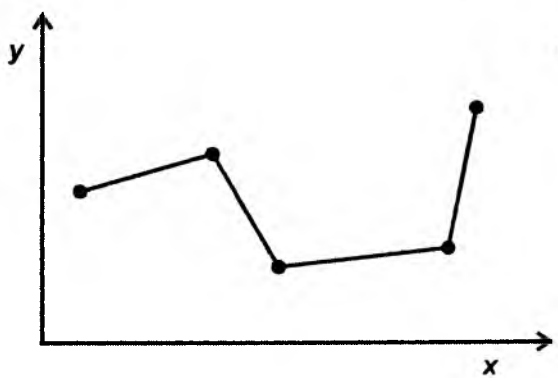


Рис. 8.1. Линейная интерполяция

Критерием гладкости является существование производных функции, описывающей кривую. Какого порядка существует производная — такого порядка и гладкость. Обычно достаточно гладкой считается функция, если она имеет производную первого или второго порядка.

Гладкая интерполяционная кривая на основе набора базовых точек из $n + 1$ штук может быть построена с помощью полинома степени n.

Полиномом называется функция вида

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ + a_1 x + a_0 = y. \qquad (8.1)$$

В формуле (8.1) неизвестными являются коэффициенты полинома a_i, $i = (0, 1, ..., n)$. Для того чтобы их найти, подставляем в уравнение $n + 1$ раз координаты из набора базовых точек. В результате получаем систему из $n + 1$ линейных относительно a_i уравнений.

Например, если набор точек состоит из трех штук, то степень полинома будет $n = 2$, а коэффициенты a_i можно получить из следующей системы уравнений:

$$\begin{cases} a_2 x_1^2 + a_1 x_1 + a_0 = y_1; \\ a_2 x_2^2 + a_1 x_2 + a_0 = y_2; \\ a_2 x_3^2 + a_1 x_3 + a_0 = y_3. \end{cases}$$

При этом важно, чтобы координаты одной и той же точки не присутствовали в наборе дважды, иначе система не будет иметь решения.

Недостатки такого подхода (когда весь набор базовых точек описывается одной функцией):

- графикам полиномов высоких степеней характерно сильное "волнение" в промежутках между базовыми точками;
- за пределами интервала базовых точек полиномы имеют тенденцию неограниченно возрастать или убывать;
- чем больше точек в наборе (выше степень полинома), тем больше уравнений для нахождения коэффициентов.

Чтобы избежать всех этих сложностей, при построении гладких кривых используют подход, заключающийся в формировании составной кривой из отдельных частей (сегментов).

Составную кривую второго порядка гладкости можно образовать из дуг обыкновенных полиномов третьей степени. Для расчета коэффициентов такого полинома требуется четыре базовых точки. Таким образом, каждый сегмент составной кривой строится на основе четырех точек. Чтобы обеспечить гладкость в местах стыковки сегментов, построение кривой осуществляется лишь между двумя "внутренними" точками каждой четверки, а сами четверки выбираются с "перекрытием", т. е. первой точкой очередной четверки выбирается вторая точка предыдущей четверки. Например, сегмент 1 (рис. 8.2) строится на основе точек 0, 1, 2, 3, а сегмент 2 — на основе точек 1, 2, 3, 4 и т. д. Чтобы построить кривую, начинающуюся в первой точке и заканчивающуюся в последней точке набора, концевые точки дублируются.

Выше предполагалось, что координаты базовых точек заданы в виде $y_i(x_i)$ и расположены в порядке возрастания значения их абсциссы. Например, случай, когда у разных точек набора абсциссы совпадают (рис. 8.3), не допускался. Поэтому сложные кривые (замкнутые или самопересекающиеся) удобно описывать при помощи параметрических уравнений.

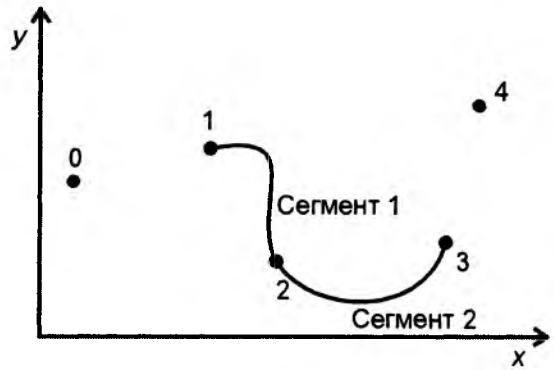


Рис. 8.2. Составная сплайновая кривая

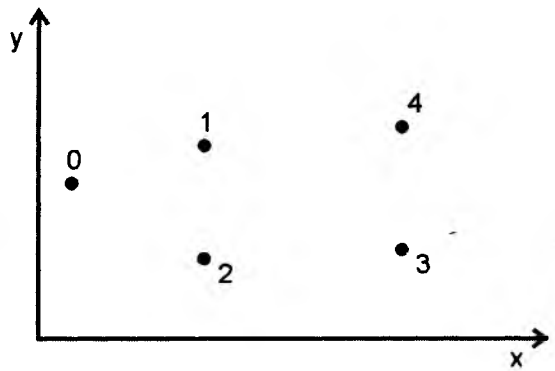


Рис. 8.3. Недопустимый набор точек

8.2. Параметрическое задание кривых

Уравнения вида:

$$x = x(t),\ y = y(t),\ z = z(t),\ \alpha \le t \le \beta \qquad (8.2)$$

называют параметрическим заданием кривой (в данном случае в трехмерном пространстве), при этом переменная t называется параметром.

Обозначим функции $x(t)$, $y(t)$, $z(t)$ вектором $\boldsymbol{R}$: $\boldsymbol{R}(t) = (x(t), y(t), z(t))$, а массив опорных точек вектором $\boldsymbol{P}$: $\boldsymbol{P} = \{P_i(x_i, y_i, z_i), i = 0, 1, 2, \ldots, n\}$.

Для каждой координаты рассчитывается своя независимая сплайновая кривая. Позиция точки, соответствующая некоторому заданному t, на каждой из параметрических кривых $x(t)$, $y(t)$, $z(t)$ и составляет положение точки (x, y, z) на сплайновой кривой. При этом значение $t = \alpha$ соответствует начальной точке сплайновой кривой, $t = \beta$ — конечной. "Пробегая" значения от α до β, параметр t задает положение каждой точки сплайновой кривой.

На практике для построения сплайновой кривой обычно используют метод составления линии из отдельных сегментов, описываемых элементарными уравнениями, как правило, третьей степени. При этом поступают следующим образом.

Для построения кривой на участке между точками с номерами i и $i + 1$ берут четверку точек с номерами $i - 1$, i, $i + 1$, $i + 2$.

Задают диапазон изменения параметра $0 \le t \le 1$. Значение параметра $t = 0$ соответствует начальной точке на участке кривой между точками с номерами i и $i + 1$. Значение параметра $t = 1$ соответствует конечной точке на участке кривой между точками с номерами i и $i + 1$. Значения $0 < t < 1$ соответствуют внутренним точкам данного участка.

Разбивают диапазон изменения параметра на m частей (например, $m = 10$).

На основе значений соответствующих координат четверки базовых точек и значения t_k, $k = (0, 1, ..., m)$ рассчитываются m промежуточных точек сплайновой кривой на участке между базовыми точками с номерами i и $i + 1$.

Рассчитанные на предыдущем шаге точки соединяются прямыми линиями. Таким образом, чем выше значение m, тем более точно будет аппроксимирована сплайновая кривая.

Достоинства такого подхода:

- упрощение расчетов;
- использование уравнений невысоких степеней;
- при добавлении точки в базовый набор необходимо пересчитать лишь четыре сегмента кривой.

При построении составной сплайновой кривой важно выполнение некоторых условий гладкости в точках их стыковки. Только в этом случае составная кривая будет обладать достаточно хорошими геометрическими характеристиками. Чтобы учесть это обстоятельство, удобно использовать класс так называемых геометрически непрерывных кривых.

Составная кривая называется *геометрически непрерывной*, если вдоль этой кривой единичный вектор ее касательной изменяется непрерывно, и *дважды геометрически непрерывной*, если и вектор кривизны также меняется непрерывно.

8.3. Сплайновые кривые

Существует большое количество разных вариантов сплайновых кривых, отличающихся своими свойствами. Приведем примеры некоторых из них.

8.3.1. Интерполяционная кривая Catmull — Rom

По заданному массиву точек P_0, P_1, P_2, P_3 сплайновая кривая Catmull — Rom определяется при помощи уравнения, имеющего следующий вид:

$$\boldsymbol{R}(t)=\frac{1}{2}\left(-t(1-t)^2 P_0+\left(2-5t^2+3t^3\right)P_1+t\left(1+4t-3t^2\right)P_2-t^2(1-t)P_3\right), \quad 0\le t\le 1. \tag{8.3}$$

Свойства составной сплайновой кривой Catmull — Rom:

- проходит точно через опорные точки;
- является геометрически непрерывной;
- набор базовых функций однозначно определяет кривую, т. е. нет возможности регулировать ее форму.

Поскольку сплайновая кривая Catmull — Rom является интерполяционной, то она проходит через каждую из базовых точек. Однако при построении составной сплайновой кривой каждый сегмент рассчитывается на участке между парой внутренних точек очередной четверки из набора базовых точек. Поэтому для построения составной сплайновой интерполяционной кривой, начинающейся в первой базовой точке и заканчивающейся в последней базовой точке, достаточно дополнить набор копиями первой и последней точек. Копия начальной точки при этом добавляется в начало набора, а копия последней точки — в конец набора.

Построить замкнутую интерполяционную сплайновую кривую можно, дополнив набор базовых точек из n штук точками: $P_{n+1} = P_0$, $P_{n+2} = P_1$, $P_{n+3} = P_2$.

8.3.2. Элементарная бета-сплайновая кривая

По заданному массиву точек P_0, P_1, P_2, P_3 бета-сплайновая кривая описывается уравнением:

$$\boldsymbol{R}(t) = b_0(t)P_0 + b_1(t)P_1 + b_2(t)P_2 + b_3(t)P_3, \quad 0\le t\le 1. \tag{8.4}$$

Функциональные коэффициенты $b_0(t)$, $b_1(t)$, $b_2(t)$, $b_3(t)$ задаются следующими формулами:

$$b_0(t)=\frac{2\beta_1^3}{\delta}(1-t)^3;$$

$$b_1(t) = \frac{1}{\delta}\left(2\beta_1^3 t\left(t^2 - 3t + 3\right) + 2\beta_1^2\left(t^3 - 3t + 2\right) + 2\beta_1\left(t^3 - 3t + 2\right) + \beta_2\left(2t^3 - 3t^2 + 1\right)\right);$$

$$b_2(t) = \frac{1}{\delta}\left(2\beta_1^2 t^2\left(-t + 3\right) + 2\beta_1 t\left(-t^2 + 3\right) + 2\beta_2 t^2\left(-2t + 3\right) + 2\left(-t^3 + 1\right)\right);$$

$$b_3(t) = \frac{2t^3}{\delta},$$

где $\beta_1 > 0$ и $\beta_2 \geq 0$ и $\delta = 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + \beta_2 + 2$.

Числовые параметры β_1 и β_2 называются параметрами формы бета-сплайновой кривой, параметр β_1 называется параметром скоса (смещения), а параметр β_2 — параметром натяжения.

Свойства составной бета-сплайновой кривой:

- проходит внутри выпуклой оболочки, заданной опорными точками;
- является дважды геометрически непрерывной кривой;
- параметры β_1 и β_2 позволяют регулировать ее форму.

Составная бета-сплайновая кривая, как правило, не проходит ни через одну из базовых точек. Однако известно, что начальная точка кривой обязательно лежит в треугольнике, образованном тремя начальными базовыми точками, а конечная точка кривой — в треугольнике, образованном тремя конечными базовыми точками. Поэтому для построения составной бета-сплайновой кривой, начинающейся в первой базовой точке и заканчивающейся в последней базовой точке, достаточно дополнить набор двумя копиями первой точки и двумя копиями последней точки.

Построить замкнутую бета-сплайновую кривую можно, дополнив набор базовых точек из n штук точками: $P_{n+1} = P_0$, $P_{n+2} = P_1$, $P_{n+3} = P_2$.

8.3.3. Сплайновая кривая Безье

По заданному массиву точек P_0, P_1, P_2, P_3 сплайновая кубическая элементарная кривая Безье описывается уравнением:

$$\boldsymbol{R}(t) = (((1 - t)P_0 + 3tP_1)(1 - t) + 3t^2P_2)(1 - t) + t^3P_3, \qquad (8.5)$$

$$0 \leq t \leq 1.$$

Элементарная кривая начинается в точке P_0 и заканчивается в точке P_3, касаясь при этом отрезков P_0P_1 и P_2P_3.

Свойства составной кривой Безье:

- проходит внутри выпуклой оболочки, заданной опорными точками;

- набор базовых функций однозначно определяет кривую, т. е. нет возможности регулировать ее форму.

Чтобы составная кривая Безье была геометрически непрерывной, необходимо чтобы каждые три точки в месте стыковки сегментов лежали на одной прямой. Например, пусть имеется шесть базовых точек P_0, ..., P_5. Для построения геометрически непрерывной составной кривой дополним этот набор вспомогательной точкой P_*, взятой на середине отрезка P_2P_3 (рис. 8.4). Составную кривую построим из двух сегментов элементарных кубических кривых Безье для четверок вершин P_0, P_1, P_2, P_* и P_*, P_3, P_4, P_5.

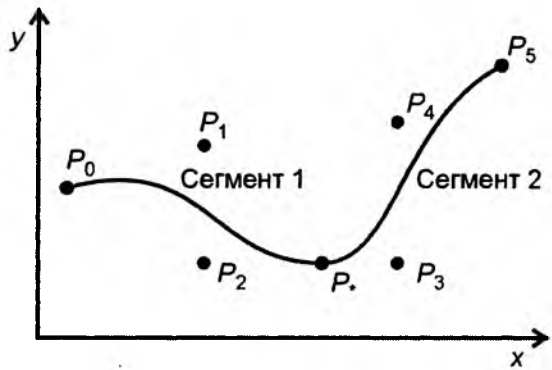


Рис. 8.4. Построение составной кривой Безье

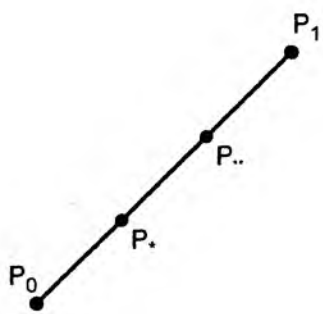


Рис. 8.5. Преобразование отрезка в кривую

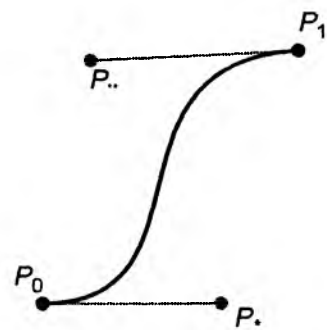


Рис. 8.6. Изменение формы кривой

Кубические кривые Безье довольно популярны в компьютерной графике. Например, в графическом редакторе CorelDRAW кривые Безье являются основой для создания всех типов линий. Для любого отрезка можно включить режим "Кривая". Для этого требуется выделить вторую точку отрезка (*узла* в терминологии CorelDRAW) и дать команду "преобразовать в кривую". В режиме "Кривая" отрезок дополняется парой *контрольных* точек

(P_* и P_{**}), равноудаленных от концов отрезка (рис. 8.5). Четверка точек P_0, P_*, P_{**}, P_1 определяет кривую Безье. Позицию точек можно менять, при этом CorelDRAW рассчитывает форму кривой (рис. 8.6).

8.4. Построение сплайновой кривой Безье с помощью средств MFC

Функция построения кривой Безье реализована в классе `CDC` библиотеки MFC. Прототип этой функции следующий:

```
BOOL PolyBezier(const POINT* lpPoints, int nCount);
```

- `lpPoints` — указатель на массив структур данных `POINT` (или объектов `CPoint`), который содержит базовые точки сплайнов;
- `nCount` — число точек в массиве `lpPoints`.

Число `nCount-1` должно быть кратно трем, т. е. при делении `(nCount-1)/3` должно получаться целое число больше нуля. Это связано с тем, что для построения элементарной кривой Безье требуется четыре точки, а при построении составной кривой последняя точка предыдущего сегмента считается начальной точкой следующего сегмента, т. е. для определения формы каждого сегмента требуются три точки (поэтому делим на три), конечная же точка последнего сегмента не дублируется (поэтому `nCount-1`).

Про кривые Безье, конечно же, не забыли и в новой библиотеке GDI+. Построение сплайновых кривых средствами GDI+ рассматривается в *разд. 14.2.2.*

8.5. Программная реализация построения сплайновых кривых

Для иллюстрации рассмотренного выше материала напишем маленькую программку. Это будет однодокументное приложение, которое легко создать с помощью генератора AppWizard (*см. разд. 3.1*). Назовем программу в честь кривых Безье "Bezier", однако это не помешает нам реализовать в ней кривые и других типов.

Основная идея программы состоит в следующем: случайным образом генерируется некоторое количество точек на плоскости, которые затем используются в качестве "базовых" для построения сплайновых кривых различных типов. Базовые точки и точки сплайновой кривой будем хранить в классе документа. На класс документа возложим также и функции по расчету сплайновых точек. Класс облика будет только рисовать базовые точки и сплайны, которые рассчитаны в классе документа.

Прежде всего модифицируем класс документа `CBezierDoc` (листинг 8.1). Добавим в него два динамических массива: `m_BasePointsArray` — для

хранения базовых точек кривой и `m_SplinePointsArray` — для хранения точек сплайновой кривой. Генерацию базовых точек поручим функции `GenerateBasePoints()`. Расчет сплайнов будут выполнять методы: `CreateBezier()` — сплайн Безье; `CreateCatmullRom()` — сплайн Catmull-Rom; `CreateBeta()` — бета-сплайн.

Рисование сплайна Безье будет выполняться методом `PolyBezier` класса `CDC`. Этот метод не заботится о геометрической непрерывности составной сплайновой кривой, которую он рисует. Поэтому для обеспечения гладкости стыковки сегментов требуется некоторым образом модифицировать набор базовых точек. Гладкой стыковки сегментов можно добиться двумя способами:

- изменить положение базовой точки на стыке сегментов так, чтобы она лежала на прямой между предыдущей и последующей точками набора;
- добавить в набор базовых точек вспомогательную "контрольную" точку *(см. разд. 8.3.3)*.

В файле Beziedoc.h также описаны прототипы функций:

- `GetMiddle()` — возвращает точку-середину отрезка, заданного точками-аргументами;
- `GetCatmullRomPoint()` — возвращает точку Catmull-Rom-сплайна, соответствующую параметру t на сегменте, заданном четверкой точек;
- `GetBetaPoint()` — возвращает точку бета-сплайна, соответствующую параметру t на сегменте, заданном четверкой точек. При расчете учитываются параметры формы β_1 и β_2.

Эти функции определены как глобальные из-за своей независимости, т. е. они никак не связаны со спецификой класса `CBezierDoc` и могут быть использованы для работы с любыми данными.

Реализация методов класса `CBezierDoc` и глобальных функций приведена в листинге 8.2.

Листинг 8.1. Класс документа `CBezierDoc`. Файл Beziedoc.h

```
// Количество генерируемых базовых точек
#define NBASEPOINTS 10
// Количество отрезков, аппроксимирующих сплайновую кривую
#define NAPPROXCUTS 10

// Тип сплайна
#define ANYSLPINE   0
#define BEZIER      1

class CBezierDoc : public CDocument
```

```
{
protected: // create from serialization only
   CBezierDoc();
   DECLARE_DYNCREATE(CBezierDoc)

// Attributes
public:
   // Динамический массив базовых точек кривой
   CArray <CPoint, CPoint> m_BasePointsArray;
   CArray <CPoint, CPoint> m_SplinePointsArray;
   // Тип сплайна. Так как кривая Безье у нас рисуется методом
   // класса CDC, надо дать знать, когда его использовать
   // Если m_nSplineType == BEZIER — кривая Безье;
   int m_nSplineType;
   // Operations
public:
   // Генерирует базовые точки
   void GenerateBasePoints();
   // На основе набора базовых точек создает
   // набор точек для построения кривой Безье
   // методом CDC::PolyBezier
   // Для обеспечения гладкости в местах стыковки сегментов кривой Безье:
   // если nMode == 0, ничего не делает;
   // если nMode == 1, дополняет набор базовых точек;
   // если nMode == 2, изменяет положение базовых точек в местах стыковки.
   void CreateBezier(int nMode=0);
   // На основе набора базовых точек рассчитывает Catmull — Rom-сплайн
   void CreateCatmullRom();
   // На основе набора базовых точек рассчитывает beta-сплайн
   void CreateBeta();

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CBezierDoc)
   //}}AFX_VIRTUAL

// Implementation
Public:
   Virtual ~CBezierDoc();
   Virtual void Serialize(CArchive& ar);   // overridden for document i/o
```

```
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
   //{{AFX_MSG(CBezierDoc)
   afx_msg void OnEditCreateBezier0();
   afx_msg void OnEditCreateBezier1();
   afx_msg void OnEditCreateBezier2();
   afx_msg void OnEditGeneratebaseline();
   afx_msg void OnEditCteatecatmullrom();
   afx_msg void OnEditCeratebeta();
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};

////////////////////////////////////////////////////////////////////////////
//Глобальная функция. Возвращает середину отрезка
CPoint GetMiddle(CPoint *pP1, CPoint *pP2);

// Возвращает точку Catmull-Rom-сплайна для параметра t=(0...1)
// на участке между точками 1 и 2 сегмента из четырех точек
// pSegment – указатель на начало сегмента
CPoint GetCatmullRomPoint(CPoint *pSegment, double t);

// Возвращает точку бета-сплайна для параметра t=(0...1)
// на участке между точками 1 и 2 сегмента из четырех точек
// pSegment – указатель на начало сегмента
// beta1, beta2 – параметры формы
CPoint GetBetaPoint(CPoint *pSegment, double t, double beta1, double beta2);
```

Листинг 8.2. Реализация методов класса CBezierDoc и глобальных функций. Файл Beziedoc.cpp

```
void CBezierDoc::GenerateBasePoints()
{
   // Выделим место под базовые точки
```

```
    // NBASEPOINTS определено в Beziedoc.h
    m_BasePointsArray.SetSize(NBASEPOINTS);
    // Зададим значения базовых точек случайным образом
    // в пределах клиентского окна программы
    CRect ClientRect;
    POSITION pos = GetFirstViewPosition();
    CView* pView =NULL;
    if(pos != NULL)
       pView = GetNextView(pos);
       if(pView!=NULL)
          pView->GetClientRect(ClientRect);
       else
          return;
    // Реинициализация генератора случайных чисел
    srand( (unsigned)time( NULL ) );
    int step=ClientRect.Width()/(m_BasePointsArray.GetSize()+1);
    // Задаем значения
    for(int i=0; i<m_BasePointsArray.GetSize(); i++)
    {
       m_BasePointsArray[i].x=step/2+step*i+rand()*step/RAND_MAX;
       m_BasePointsArray[i].y=rand()*ClientRect.Height()/RAND_MAX;
    }
    // Удаляем старый сплайн
    m_SplinePointsArray.RemoveAll();
};

void CBezierDoc::CreateBezier(int nMode/*=0*/)
{
    // Копируем базовые точки
    m_SplinePointsArray.SetSize(m_BasePointsArray.GetSize());
    for(int i=0; i<m_BasePointsArray.GetSize(); i++)
    m_SplinePointsArray[i]=m_BasePointsArray[i];

    switch (nMode)
    {
    case 1:
       // Дополняем массив
       for(i=2; i<m_SplinePointsArray.GetSize()-1; i+=3)
          m_SplinePointsArray.InsertAt(i+1,
```

```
            GetMiddle(&m_SplinePointsArray[i], &m_SplinePointsArray[i+1]));
      break;
      case 2:
         // Меняем позицию точек в местах стыковки сегментов
         for(i=2; i<m_SplinePointsArray.GetSize()-2; i+=3)
            m_SplinePointsArray[i+1]=
            GetMiddle(&m_SplinePointsArray[i], &m_SplinePointsArray[i+2]);
      break;
      }
      m_nSplineType=BEZIER;
};

void CBezierDoc::CreateCatmullRom()
{
   if(m_BasePointsArray.GetSize()<4) return;
   // Добавляем воображаемые базовые точки
   m_BasePointsArray.InsertAt(0, m_BasePointsArray[0]);
   m_BasePointsArray.Add(
            m_BasePointsArray[m_BasePointsArray.GetUpperBound()] );

   m_SplinePointsArray.SetSize(
       (m_BasePointsArray.GetSize()-3)*NAPPROXCUTS+1 );
   // Получим прямой доступ к данным массива базовых точек
   CPoint *pBasePoint=m_BasePointsArray.GetData();
   double t=0.,                  // параметр t
          dt=1.0/NAPPROXCUTS;    // шаг приращения параметра t
   int n=0,                      // локальный номер точки внутри сегмента
       nSegment=0;               // номер первой точки текущего сегмента
   for(int i=1; i<m_BasePointsArray.GetSize()-2; i++)
   {
      t=0.;
      for(n=0; n<=NAPPROXCUTS; n++, t+=dt)
         m_SplinePointsArray[nSegment+n]=
                                 GetCatmullRomPoint(pBasePoint+(i-1), t);
      nSegment+=NAPPROXCUTS;
   }
   // Удаляем воображаемые базовые точки
   m_BasePointsArray.RemoveAt(0);
   m_BasePointsArray.RemoveAt(m_BasePointsArray.GetUpperBound());
```

```
    m_nSplineType=ANYSLPINE;
};

void CBezierDoc::CreateBeta()
{
    if(m_BasePointsArray.GetSize()<4) return;
    double   Beta1=1.0,
             Beta2=0.0;
    // Добавляем воображаемые базовые точки
    m_BasePointsArray.InsertAt(0, m_BasePointsArray[0]);
    m_BasePointsArray.InsertAt(0, m_BasePointsArray[0]);
    m_BasePointsArray.Add(
       m_BasePointsArray[m_BasePointsArray.GetUpperBound()] );
    m_BasePointsArray.Add(
       m_BasePointsArray[m_BasePointsArray.GetUpperBound()] );

    m_SplinePointsArray.SetSize(
                        (m_BasePointsArray.GetSize()-3)*NAPPROXCUTS+1);
    // Получим прямой доступ к данным массива базовых точек
    CPoint *pBasePoint=m_BasePointsArray.GetData();
    double t=0.,                    // параметр t
           dt=1.0/NAPPROXCUTS;      // шаг приращения параметра t
    int n=0,                        // локальный номер точки внутри сегмента
        nSegment=0;                 // номер первой точки текущего сегмента
    for(int i=1; i<m_BasePointsArray.GetSize()-2; i++)
    {
       t=0.;
       for(n=0; n<=NAPPROXCUTS; n++, t+=dt)
          m_SplinePointsArray[nSegment+n]=
                   GetBetaPoint(pBasePoint+(i-1), t, Beta1, Beta2);
       nSegment+=NAPPROXCUTS;
    }
    // Удаляем воображаемые базовые точки
    m_BasePointsArray.RemoveAt(0,2);
    m_BasePointsArray.RemoveAt(m_BasePointsArray.GetUpperBound()-1,2);
    m_nSplineType=ANYSLPINE;
};
```

```
////////////////////////////////////////////////////////////////////////////
// Глобальная функция. Возвращает середину отрезка
CPoint GetMiddle(CPoint *pP1, CPoint *pP2)
{
   return CPoint(pP1->x+(pP2->x-pP1->x)/2, pP1->y+(pP2->y-pP1->y)/2);
}

// Возвращает точку Catmull-Rom-сплайна для параметра t=(0...1)
// на участке между точками 1 и 2 четверки точек
CPoint GetCatmullRomPoint(CPoint *pSegment, double t)
{
   double s=1.0-t, t2=t*t, t3=t2*t;
   CPoint Res;
   Res.x=(LONG)(0.5*(-t*s*s*pSegment[0].x+
         (2-5*t2+3*t3)*pSegment[1].x+
         t*(1+4*t-3*t2)*pSegment[2].x-t2*s*pSegment[3].x)+0.5);
   Res.y=(LONG)(0.5*(-t*s*s*pSegment[0].y+
         (2-5*t2+3*t3)*pSegment[1].y+
         t*(1+4*t-3*t2)*pSegment[2].y-t2*s*pSegment[3].y)+0.5);
   return Res;
};

CPoint GetBetaPoint(CPoint *pSegment, double t, double beta1, double beta2)
{
   double s=1.0-t,
         t2=t*t,
         t3=t2*t,
         b12=beta1*beta1,
         b13=b12*beta1,
         delta=2.0*b13+4.0*b12+4.0*beta1+beta2+2.0,
         d=1.0/delta,
         b0=2*b13*d*s*s*s,
         b3=2*t3*d,
         b1=d*(2*b13*t*(t2-3*t+3)+2*b12*(t3-3*t2+2)+
            2*beta1*(t3-3*t+2)+beta2*(2*t3-3*t2+1)),
         b2=d*(2*b12*t2*(-t+3)+2*beta1*t*(-t2+3)+
            beta2*t2*(-2*t+3)+2*(-t3+1) );
   CPoint Res;
   Res.x=(LONG)(b0*pSegment[0].x+b1*pSegment[1].x+
```

```
        b2*pSegment[2].x+b3*pSegment[3].x +0.5);
    Res.y=(LONG)(b0*pSegment[0].y+bl*pSegment[1].y+
        b2*pSegment[2].y+b3*pSegment[3].y +0.5);
    return Res;
};
```

Задача изображения на экране базовых точек и сплайновых кривых решается методами класса CBezierView. Для этих целей добавим в описание класса два метода (листинг 8.3): DrawBaseLine() — рисует базовые точки и соединяет их прямыми линиями; DrawSplineLine() — рисует сплайновую кривую. Реализация этих методов приведена в листинге 8.4.

Листинг 8.3. Интерфейс класса CBezierView с новыми методами. Файл Bezievw.h

```
class CBezierView : public CView
{
protected:
    CBezierView();
    DECLARE_DYNCREATE(CBezierView)

// Attributes
public:
    CBezierDoc* GetDocument();
    // Методы
    // Рисует базовую кривую
    void DrawBaseLine(CDC *pDC);
    // Рисует сплайновую кривую
    void DrawSplineLine(CDC *pDC);
...
```

Листинг 8.4. Реализация методов рисования сплайновой кривой. Файл Bezievw.cpp

```
void CBezierView::DrawBaseLine(CDC *pDC)
{
    CBezierDoc* pDoc = GetDocument();
    int nCount=pDoc->m_BasePointsArray.GetSize();
    // Рисуем точки
    for(int i=0; i<nCount; i++)
        pDC->Ellipse( pDoc->m_BasePointsArray[i].x-4,
```

```
                    pDoc->m_BasePointsArray[i].y-4,
                    pDoc->m_BasePointsArray[i].x+4,
                    pDoc->m_BasePointsArray[i].y+4);
   // Перо для рисования базовой кривой
   CPen PenBase; PenBase.CreatePen(PS_DOT, 1, RGB(0,0,255));
   CPen *pPenOld=pDC->SelectObject(&PenBase);
   // Соединим прямыми базовые точки
   pDC->Polyline(pDoc->m_BasePointsArray.GetData(), nCount);
   // Восстановим старое перо
   pDC->SelectObject(pPenOld);
};

void CBezierView::DrawSplineLine(CDC *pDC)
{
   CBezierDoc* pDoc = GetDocument();
   int nCount=pDoc->m_SplinePointsArray.GetSize();
   // Сплайн рисуем красным пером
   CPen PenSpline; PenSpline.CreatePen(PS_SOLID, 2, RGB(255,0,0));
   CPen *pPenOld=pDC->SelectObject(&PenSpline);
   if(pDoc->m_nSplineType==BEZIER)
   {
      // Рисуем кривую Безье методом класса CDC::PolyBezier
      pDC->PolyBezier(pDoc->m_SplinePointsArray.GetData(), nCount/3*3+1);
      // Покажем точки стыковки сегментов
      for(int i=3; i<nCount; i+=3)
      pDC->Ellipse( pDoc->m_SplinePointsArray[i].x-4,
                    pDoc->m_SplinePointsArray[i].y-4,
                    pDoc->m_SplinePointsArray[i].x+4,
                    pDoc->m_SplinePointsArray[i].y+4);
   }
   else
      // Соединим прямыми базовые точки
      pDC->Polyline(pDoc->m_SplinePointsArray.GetData(), nCount);
   // Восстановим старое перо
   pDC->SelectObject(pPenOld);
}
```

Методы `DrawBaseLine()` и `DrawSplineLine()` вызываются из метода `CBezierView::OnDraw()` при необходимости обновления изображения на экране (листинг 8.5).

Листинг 8.5. Метод `CBezierView::OnDraw()`. Файл Bezievw.cpp

```
void CBezierView::OnDraw(CDC* pDC)
{
   CBezierDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   DrawBaseLine(pDC);
   DrawSplineLine(pDC);
}
```

8.6. Заключение

Текст программы находится на компакт-диске в каталоге Sources\Bezier. Результаты работы программы показаны на рис. 8.7—8.11.

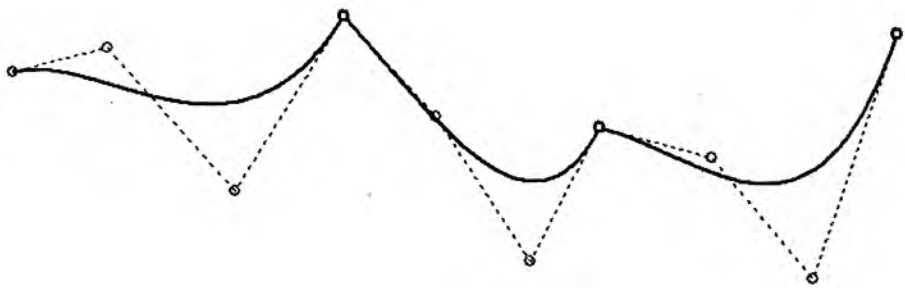


Рис. 8.7. Построение кривой функцией `CDC::PolyBezier()` без изменения набора базовых точек

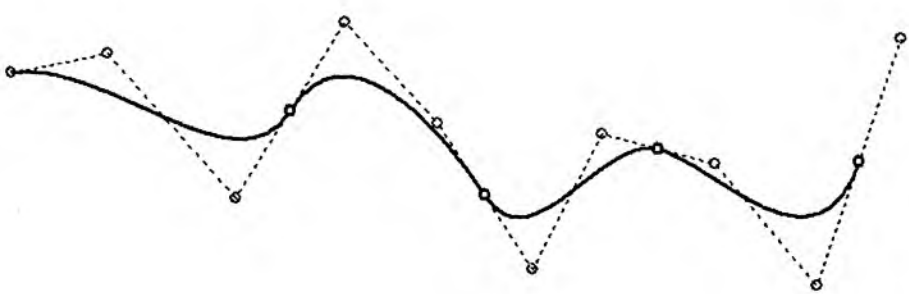


Рис. 8.8. Построение кривой функцией `CDC::PolyBezier()`. Набор базовых точек дополнен вспомогательными контрольными точками в местах стыковки сегментов. В результате количество сегментов увеличилось. Так как число вспомогательных точек оказалось не кратно трем, сплайновая кривая не доходит до последней базовой точки

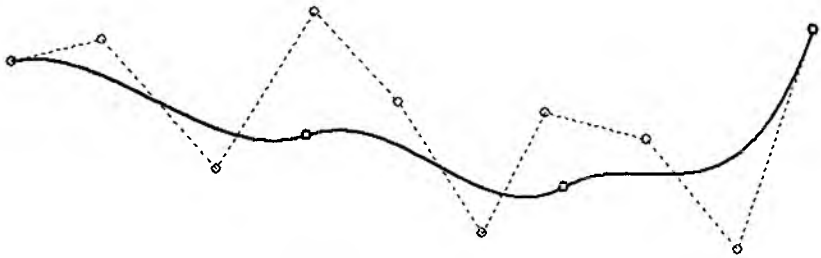


Рис. 8.9. Построение кривой функцией `CDC::PolyBezier()`. Для обеспечения гладкости кривой изменено положение базовых точек на стыках сегментов

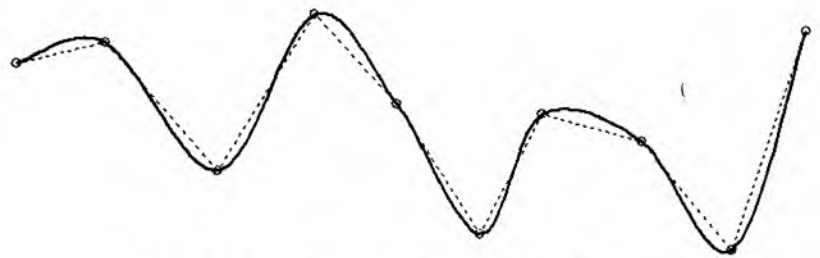


Рис. 8.10. Интерполяционная кривая Catmull-Rom

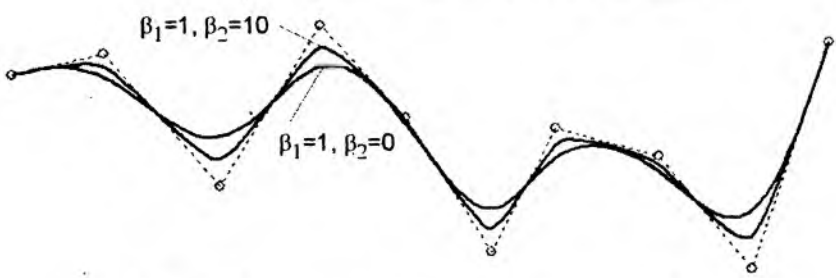
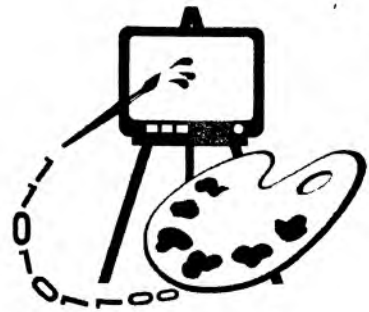


Рис. 8.11. Аппроксимационная бета-сплайновая кривая

На основе рассмотренных методов можно реализовать в программе Painter новый класс — сплайновая кривая. Такой класс можно породить от класса `CPoligon` и добавить в него методы расчета сплайновых кривых. Кроме того, рассмотренный подход можно использовать для рисования сплайновых кривых в трехмерном пространстве. Однако в случае построения сплайновой поверхности придется использовать другие расчетные формулы. Если у вас возникнет такая задача, рекомендуем почитать [17]. Кроме того, для построения поверхностей можно воспользоваться средствами библиотеки OpenGL [20].

Для изучения свойств бета-сплайновой кривой можно поэкспериментировать со значениями коэффициентов β_1 и β_2.

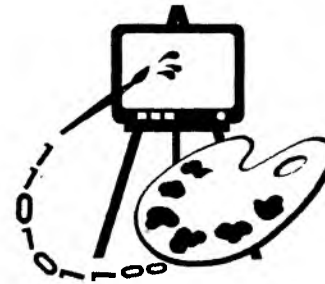


Часть III

РАБОТА С РАСТРОВОЙ ГРАФИКОЙ

Глава 9

Работа с растровыми ресурсами



В этой главе рассматривается:

- создание пиктограмм;
- создание собственных курсоров;
- использование растровых изображений (программа Painter 4.1).

В *главе 1* упоминалось о разных типах изображений, однако до сих пор мы работали только с векторной графикой. На самом же деле в компьютерном мире растровые изображения имеют, возможно, даже бóльшее распространение.

Рассмотрим далее, как использование растровых ресурсов позволит оживить аскетический облик программы Painter и добавит в это приложение новые изобразительные возможности.

9.1. Ресурсы

Ресурсы — это некоторые данные, которые присоединяются к концу выполняемого файла. Обычно ресурсы хранятся на диске и загружаются в память при необходимости. Стандартные ресурсы Windows-приложений представляют собой данные разных типов:

- ускорители (таблицы акселераторов);
- строки;
- диалоговые окна;
- меню;
- информация о версии программы;
- растровые изображения;
- курсоры;
- пиктограммы.

Среда разработки MS Visual C++ предоставляет удобные средства для создания и редактирования ресурсов всех типов. Работу с некоторыми из ресурсов мы уже коротко рассматривали. Остановимся теперь подробней на тех ресурсах, которые представляют собой графические данные.

При создании Windows-приложения AppWizard позволяет автоматически укомплектовать программу стандартным набором ресурсов. Эти ресурсы перечислены на вкладке **Ресурсы** (ResourceView) **Рабочего пространства** (Workspace). Программа Painter также уже содержит ресурсы (рис. 9.1).

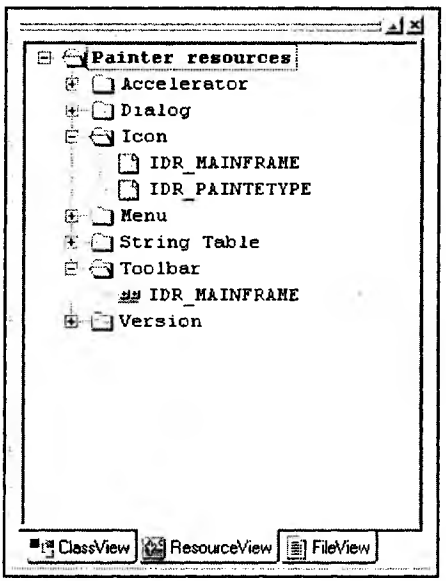


Рис. 9.1. Ресурсы программы Painter

9.2. Пиктограммы приложения

Windows-приложение обычно имеет свою пиктограмму (иконку, значок) — маленькое растровое изображение, которое сопровождает повсюду программу и помогает отличить одно приложение от другого. Стандартные размеры пиктограмм 16×16, 32×32 и 48×48 пикселов. Наиболее часто употребляются размеры 16×16 и 32×32. Пиктограммы различного размера применяются, например, в разных режимах просмотра содержимого каталога в Проводнике Windows. Если программа не содержит пиктограммы нужного размера, то Windows самостоятельно создает ее из той, что имеется. Если же программа вообще не имеет собственной пиктограммы, то Windows использует одну из стандартных пиктограмм в соответствии с типом файла.

Программа Painter уже содержит заготовки значков для самой программы (идентификатор `IDR_MAINFRAME`) и для создаваемых документов (идентифи-

катор IDR_PAINTETYPE). Двойной щелчок мыши на идентификаторе открывает пиктограмму во встроенном растровом редакторе MS Visual C++ (рис. 9.2). Используя инструменты редактирования, расположенные на правой панели, можно привести рисунок в соответствие со своим вкусом.

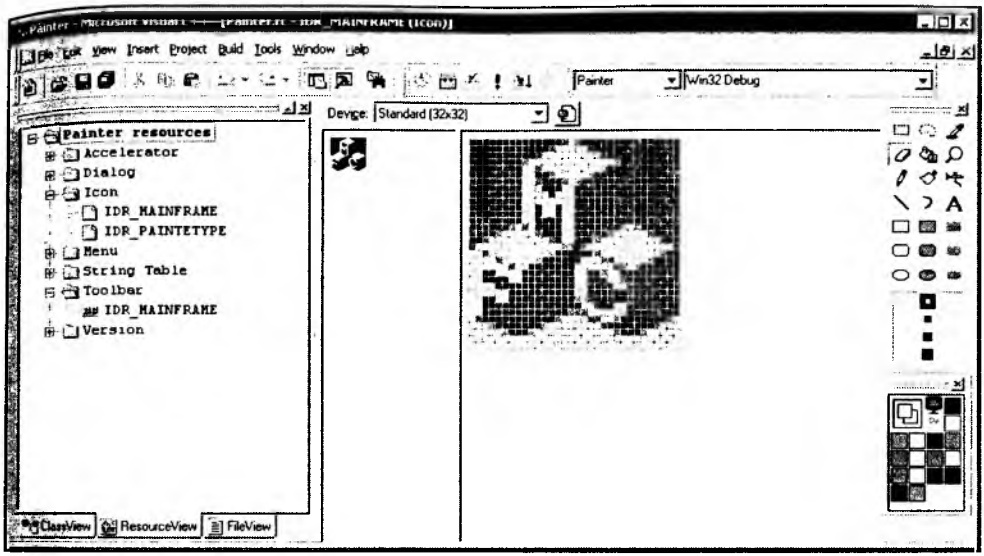


Рис. 9.2. Редактирование пиктограммы

Пиктограмма 16×16 программы помещается в левый верхний угол заголовка приложения. Пиктограмма 32×32 по умолчанию отображается в диалоговом окне **About**, с которым мы уже знакомы *(см. разд. 3.1)*. С новой пиктограммой это диалоговое окно может выглядеть, например, так, как показано на рис. 9.3.

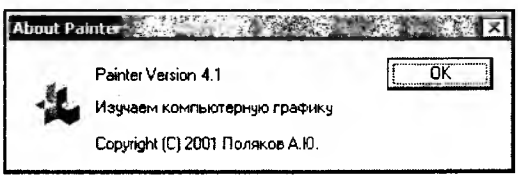


Рис. 9.3. Диалоговое окно **About** с новой пиктограммой

Пиктограмма, добавленная генератором приложения AppWizard в ресурсы программы Painter и помещенная в диалоговое окно **About**, загружается автоматически каркасом приложения. Рассмотрим далее, как можно выполнить загрузку пиктограммы "вручную". Сделаем так, чтобы при наведении курсора мыши на значок в диалоговом окне **About** изображение пиктограммы менялось.

Для этого выполним следующие действия.

1. Создадим копию пиктограммы нашего приложения. В окне **Workspace** выделим идентификатор пиктограммы `IDR_MAINFRAME`, затем выполним команды **Edit | Copy (Ctrl+C)** и **Edit | Paste (Ctrl+V)**. В наши ресурсы добавится еще одна пиктограмма с идентификатором `IDR_MAINFRAME1`.
2. Отредактируем пиктограмму. Например, изменим цвет фона (рис. 9.4).

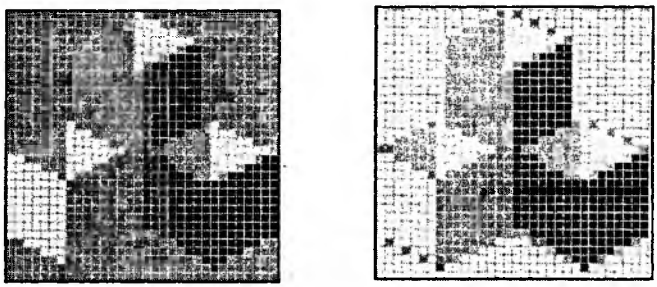


Рис. 9.4. Исходная и измененная пиктограммы

3. Присвоим пиктограмме в диалоговом окне **About** более осмысленный идентификатор `IDC_STATIC_ICON` (рис. 9.5).

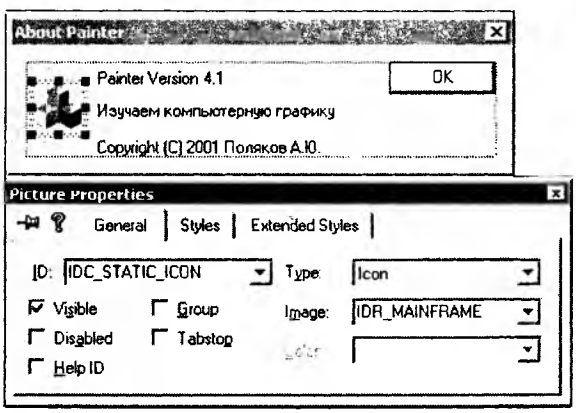


Рис. 9.5. Пиктограмма с новым идентификатором

4. Добавим с помощью ClassWizard в класс диалога метод обработки движения мыши `CAboutDlg::OnMouseMove()`.
5. Напишем в методе `CAboutDlg::OnMouseMove()` несколько строк, в которых будем проверять положение мыши, и, если курсор мыши находится над пиктограммой, будем показывать измененную копию пиктограммы (листинг 9.1).

Листинг 9.1. Обработка движения мыши в классе диалога About. Файл Painter.cpp

```
void CAboutDlg::OnMouseMove(UINT nFlags, CPoint point)
{
   HICON hIcon;
   CRect IconRect;
   // Координаты пиктограммы в единицах диалогового окна
   IconRect.left=11; IconRect.top=17;
   IconRect.right=IconRect.left+20; IconRect.bottom=IconRect.top+20;
   // Преобразовать координаты пиктограммы в экранные координаты
   MapDialogRect(&IconRect);
   // Если попали в пиктограмму
   if(IconRect.PtInRect(point)) // загрузить измененную копию
      hIcon=AfxGetApp()->LoadIcon(IDR_MAINFRAME1);
   else hIcon=AfxGetApp()->LoadIcon(IDR_MAINFRAME); // загрузить оригинал
   // Получим указатель на элемент интерфейса, показывающий пиктограмму
   CWnd* pIcinWnd = GetDlgItem(IDC_STATIC_ICON);
   ASSERT(pIcinWnd!=NULL);
   // Установим пиктограмму
   ((CStatic*)pIcinWnd)->SetIcon(hIcon);

   CDialog::OnMouseMove(nFlags, point);
}
```

Координаты пиктограммы можно узнать, выделив ее в редакторе шаблона диалога (они будут показаны в строке состояния). Однако эти координаты приводятся в специальных единицах, которые отличаются от экранных координат. К счастью, у диалога есть встроенный метод `MapDialogRect()`, который позволит нам преобразовать координаты пиктограммы из "диалоговых" координат в экранные. Метод `LoadIcon()` класса `CWinApp` загружает пиктограмму и возвращает ее в дескриптор. Причем если ресурс однажды уже был загружен, то повторная загрузка не выполняется. Этот метод позволяет загрузить пиктограмму, соответствующую по размерам параметрам `SM_CXICON` и `SM_CYICON` системных метрик Windows (обычно 32×32). Пиктограммы других размеров могут быть загружены API-функцией `LoadImage()`, которая будет рассмотрена в *разд. 9.6*.

На рис. 9.6 показано диалоговое окно **About** с двумя состояниями пиктограммы.

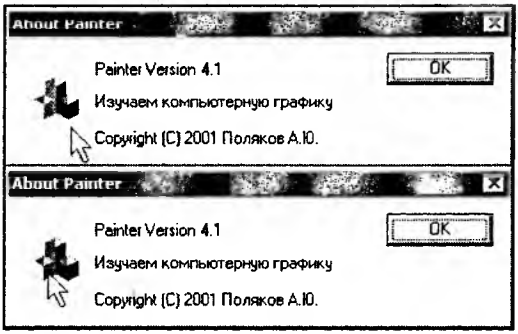


Рис. 9.6. Диалоговое окно **About** с "активной" пиктограммой

9.3. Изображение панели инструментов

Панель инструментов — элемент управления Windows-приложения, который обычно выглядит как ряд кнопок, ускоряющих выполнение команд. Генератором AppWizard в наше приложение Painter уже добавлен такой элемент. Внешний вид кнопок определяется растровой картинкой, изображение которой мы можем изменить. Так и поступим. На вкладке **RecourseView** окна **Workspace** откроем папку с надписью **Toolbar** и двойным щелчком мыши на идентификаторе `IDR_MAINFRAME` загрузим изображение панели в редактор. Щелкнув мышью на кнопке панели, можно получить ее увеличенное изображение и отредактировать его. Кроме того, редактор позволяет назначить любой кнопке идентификатор команды, которая будет вызываться данной кнопкой. Для этого надо выполнить двойной щелчок мышью на кнопке в панели инструментов (рис. 9.7). Как только кнопке будет присвоен идентификатор, в панели появится новая "пустая" кнопка. Для перемещения кнопки в требуемую позицию нужно перетащить ее мышью. Удалить кнопку можно просто, убрав ее мышью из панели инструментов.

Новая панель инструментов может выглядеть так, как показано на рис. 9.8.

9.4. Курсор

Курсор — это изображение размером 32×32 пиксела, которое показывает нам положение мыши на экране. Windows использует большое число различных курсоров для обозначения разных состояний (режимов) работы приложений (рис. 9.9).

Приложение может самостоятельно устанавливать вид курсора. Реализуем в программе Painter эту возможность. Будем изменять курсор в зависимости от того, какую фигуру собираемся нарисовать.

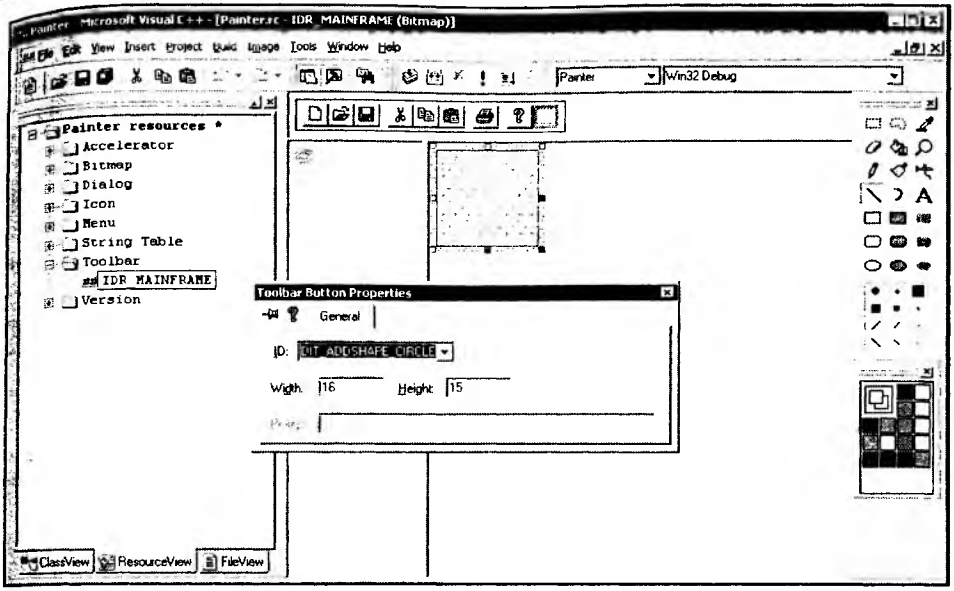


Рис. 9.7. Редактирование панели инструментов

Рис. 9.8. Новая панель инструментов

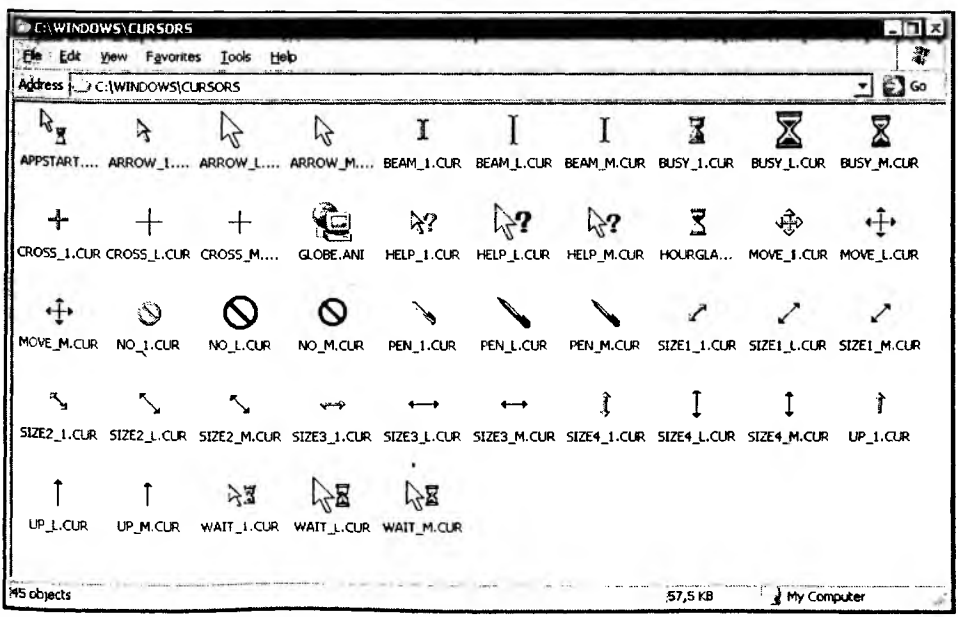


Рис. 9.9. Курсоры Windows

Прежде всего, добавим в ресурсы программы ресурс "курсор". Для этого выполним команду **Insert | Resource**, и в появившемся диалоге выберем **Cursor** (рис. 9.10).

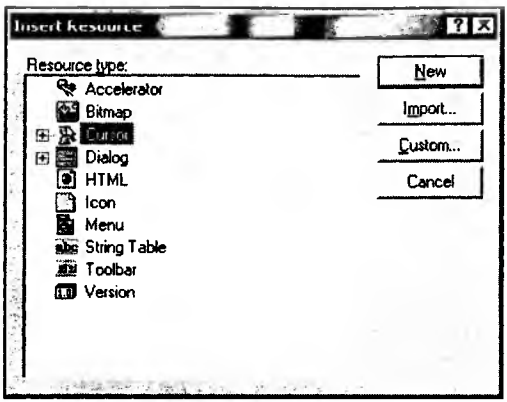


Рис. 9.10. Добавление ресурса "курсор"

В программу будет вставлен пустой шаблон курсора, изображение которого можно отредактировать так же, как и изображение пиктограммы. Отличие заключается в том, что шаблон курсора по умолчанию монохромный, поэтому для рисования доступны только два цвета: черный и белый. На самом деле курсоры могут быть и цветными и состоять из 256 цветов. Превратить курсор в цветной можно, загрузив палитру цветов (эта операция будет рассмотрена в следующем разделе). При создании курсора требуется также определить пиксел (hotspot), который будет указывать точное положение мыши. Для этого надо нажать кнопку **Set Hotspot** (находится на панели инструментов над изображением) и затем щелкнуть левой кнопкой мыши на любом пикселе внутри изображения курсора. Отмеченный пиксел и будет указывать точное положение курсора мыши.

Ну что же, создадим четыре разных курсора для индикации различных операций (рис. 9.11).

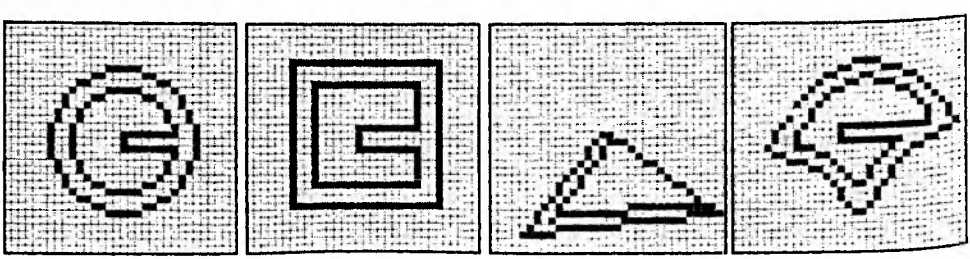


Рис. 9.11. Курсоры для разных операций

Для загрузки курсоров будем использовать метод `LoadCursor()` класса `CWinApp`. Этот метод принимает в качестве параметра идентификатор курсора и возвращает его в дескриптор.

Поскольку за тип текущей операции у нас отвечает класс `CPainterView`, сделаем дескрипторы курсоров данными этого класса (листинг 9.2), а курсоры загрузим в конструкторе `CPainterView::CPainterView()` (листинг 9.3).

Листинг 9.2. Фрагмент интерфейса класса `CPainterView` с объявлением дескрипторов для курсоров. Файл PainterView.h

```
class CPainterView : public CScrollView
{
...
// Данные
public:
   // Текущая операция
   int m_CurOper;
   // Курсоры различных операций
   HCURSOR m_hcurCircle;    // рисуем круг
   HCURSOR m_hcurSquare;    // рисуем квадрат
   HCURSOR m_hcurPolygon;    // рисуем полилинию или полигон
   HCURSOR m_hcurSurface;    // рисуем поверхность
   // Курсор "по умолчанию"
   HCURSOR m_hcurDefault;    // используем в операции выбора
...
```

Кроме курсоров, обозначающих различные операции, потребуется также дескриптор курсора по умолчанию (обычно стрелочка), будем устанавливать его при операции выбора фигуры. Курсор "по умолчанию" хранится во внутренней структуре `WNDCLASS` класса окна. Дескриптор этого курсора мы можем получить с помощью Windows API-функции `GetClassLong()` при создании окна облика. Для этого с помощью ClassWizard добавим функцию обработки сообщения `WM_CREATE` (листинг 9.3).

Листинг 9.3. Методы класса `CPainterView`, в которых происходит загрузка курсоров. Файл PainterView.cpp

```
CPainterView::CPainterView()
{
   // TODO: add construction code here
   m_CurOper=OP_NOOPER;
```

```
    m_nMyFlags=0;

    m_hcurCircle=AfxGetApp()->LoadCursor(IDC_CURSOR_CIRCLE);
    m_hcurSquare=AfxGetApp()->LoadCursor(IDC_CURSOR_SQUARE);
    m_hcurPolygon=AfxGetApp()->LoadCursor(IDC_CURSOR_POLYGON);
    m_hcurSurface=AfxGetApp()->LoadCursor(IDC_CURSOR_SURFACE);
}

int CPainterView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CScrollView::OnCreate(lpCreateStruct) == -1)
        return -1;

    m_hcurDefault=(HCURSOR)::GetClassLong(GetSafeHwnd(), GCL_HCURSOR);
    return 0;
}
```

Для установки курсора можно воспользоваться Windows API-функцией `SetCursor()`. Однако Windows обновляет изображение курсора каждый раз, когда происходит перемещение мыши. При этом изображение курсора заменяется на заданное при регистрации класса окна. Поэтому один из способов контроля формы курсора заключается в обработке приложением сообщения `WM_SETCURSOR`. Форму курсора можно устанавливать и в методе `CPainterView::OnMouseMove()`, дополнив его следующей конструкцией (листинг 9.4).

Листинг 9.4. Установка курсора в методе `CPainterView::OnMouseMove()`

```
// Изменение формы курсора
// Для того чтобы предотвратить мигание курсора,
// установим "пустой" курсор по умолчанию
::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, NULL);
// Установим курсор, соответствующий выполняемой операции
switch(m_CurOper)
    {
    case OP_CIRCLE:
        SetCursor(m_hcurCircle);
        break;
    case OP_SQUARE:
        SetCursor(m_hcurSquare);
```

```
    break;
  case OP_LINE:
    SetCursor(m_hcurPolygon);
    break;
  case OP_SURFACE:
    SetCursor(m_hcurSurface);
    break;
  default:
    ::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, (LONG)m_hcurDefault);
  }
```

Этот способ хорош в случае, если вы хотите динамически менять форму курсора, например для индикации попадания в некоторую область окна. Обратите внимание, что перед вызовом `SetCursor()` курсор окна по умолчанию устанавливается в `NULL` с помощью Windows API-функции `SetClassLong()`. Это делается для того, чтобы предотвратить мигание курсора при его движении.

В нашем случае, когда мы переключаем режимы с помощью команд, можно просто устанавливать соответствующий курсор "по умолчанию" для окна при включении той или иной операции (листинг 9.5). Такой подход значительно сократит количество вызовов функций при обработке перемещения мыши. Хотя при современных скоростях это может быть и не особо важно, но "мелочь, а приятно".

Листинг 9.5. Установка формы курсора при выборе типа операции. Файл PainterView.cpp

```
void CPainterView::OnEditAddshapePoint()
{
   m_CurOper=OP_POINT;
   ::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, (LONG)m_hcurCircle);
}

void CPainterView::OnEditAddshapeCircle()
{
   m_CurOper=OP_CIRCLE;
   ::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, (LONG)m_hcurCircle);
}

void CPainterView::OnEditAddshapeSquare()
{
```

```
    m_CurOper=OP_SQUARE;
    ::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, (LONG)m_hcurSquare);
}

void CPainterView::OnEditAddshapePolyline()
{
    CBasePoint *pShape=new CPolygon;
    // Черная линия шириной 0,5 мм
    pShape->SetPen(RGB(0,0,0), 50, PS_GEOMETRIC);

    CPainterDoc *pDoc=GetDocument();
    // Добавляем в конец списка
    pDoc->m_ShapesList.AddTail(pShape);
    // Последняя фигура становится активной
    pDoc->m_pSelShape=pShape;
    // Указываем, что документ изменен
    pDoc->SetModifiedFlag();

    m_CurOper=OP_LINE;
    ::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, (LONG)m_hcurPolygon);
}

void CPainterView::OnEditAddshapePolygon()
{
    CBasePoint *pShape=new CPolygon;
    // Темно-зеленая заливка
    pShape->SetBrush(RGB(0,100,0));
    // Черная линия шириной 0,5 мм
    pShape->SetPen(RGB(0,0,0), 50, PS_GEOMETRIC);

    // Так как pShape указатель на CBasePoint,
    // а метод SetPolygon() имеется только у класса CPolygon,
    // требуется преобразование типа указателя
    ((CPolygon*)pShape)->SetPolygon(TRUE);
    CPainterDoc *pDoc=GetDocument();
    // Добавляем в конец списка
    pDoc->m_ShapesList.AddTail(pShape);
    // Последняя фигура становится активной
    pDoc->m_pSelShape=pShape;
```

```
    // Указываем, что документ изменен
    pDoc->SetModifiedFlag();

    m_CurOper=OP_LINE;
    ::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, (LONG)m_hcurPolygon);
}

void CPainterView::OnEditAddshapeSurface()
{
    m_CurOper=OP_SURFACE;
    ::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, (LONG)m_hcurSurface);
}
```

При использовании способа, описанного в листинге 9.5, в методе `CPainterView::OnMouseMove()` достаточно оставить разбор одного случая:

```
default:
    ::SetClassLong(GetSafeHwnd(), GCL_HCURSOR, (LONG)m_hcurDefault);
```

Это позволит восстановить курсор по умолчанию для всех операций, которым не определен собственный курсор.

9.5. Растровое изображение Bitmap

Рассмотренные выше курсоры и пиктограммы являются, по сути, частными случаями ресурса "растровое изображение" (bitmap). Растровые изображения не ограничены в размерах и могут иметь до 256 цветов.

Для добавления растрового изображения в ресурсы программы существует несколько способов. Один из них выполняется с помощью команды **Insert | Resource | Bitmap | New**. Свойства изображения задаются в диалоговом окне **Bitmap Properties** (рис. 9.12), которое вызывается двойным щелчком мыши в окне редактирования за пределами изображения.

В диалоговом окне **Bitmap Properties** можно установить размер изображения и количество используемых цветов. Растровое изображение будет сохранено в файл, имя которого задано в поле **File name**, и получит идентификатор, заданный в поле **ID**. Саму картинку можно нарисовать, используя инструменты встроенного редактора MS Visual C++. Для рисования можно также воспользоваться и более мощным графическим редактором, например Adobe Photoshop. В этом случае изображение из одного редактора в другой можно перенести через буфер обмена. Каждое изображение имеет свою палитру, поэтому если изображение было вставлено через буфер обмена, для правильного отображения цветов требуется также загрузить соответствующую

палитру. Загрузка палитры выполняется с помощью команды **Image | Load Palette**. Конечно, палитра должна быть предварительно сохранена в файл в редакторе, в котором создается изображение. Просмотреть палитру изображения можно на вкладке **Palette** диалога **Bitmap Properties** (рис. 9.13).

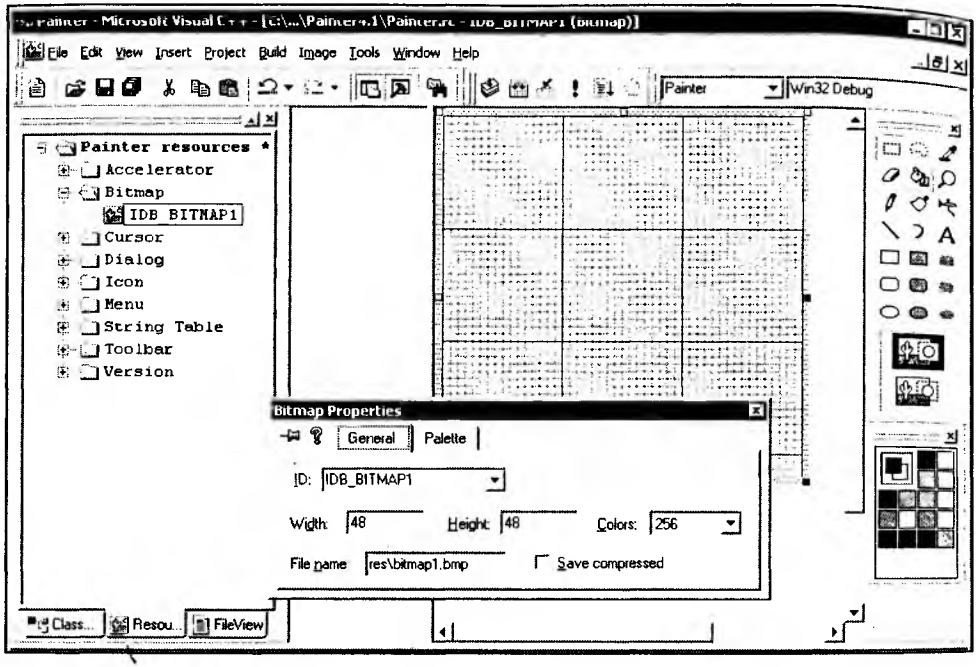


Рис. 9.12. Свойства растрового изображения

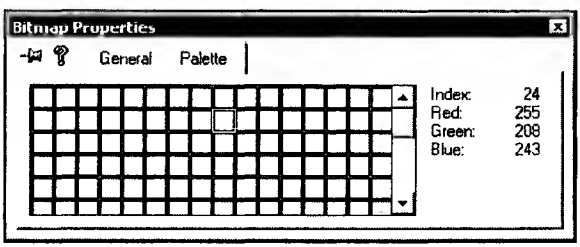


Рис. 9.13. Палитра изображения

Еще один способ вставки растрового изображения в ресурсы программы реализуется с помощью команды **Insert | Resource | Import**. Эта команда позволяет импортировать изображение из файла на диске. Конечно, для того чтобы что-то импортировать, нужно сначала создать изображение и сохранить в режиме индексированных цветов в формате BMP.

Растровые изображения в MFC описываются классом `CBitmap`. Класс `CBitmap` позволяет выполнять операции загрузки и вывода изображений,

являющихся ресурсами приложения. Для загрузки изображения в программе создается объект класса `CBitmap`, а затем вызывается метод `CBitmap::LoadBitmap(IDB_BITMAP)`, которому в качестве параметра передается идентификатор растрового ресурса.

Рассмотрим далее, как можно использовать растровые изображения в качестве шаблонов кисти. При создании базового класса `CBasePoint` иерархии фигур программы Painter была зарезервирована специальная переменная для хранения идентификатора шаблона кисти *(см. разд. 4.3.4)*. Таким образом, фигуры в программе Painter потенциально уже умеют создавать кисти на основе растровых шаблонов. Нам остается только добавить в программу сами растровые изображения и каким-то образом сообщить фигурам их идентификаторы. Пусть пользователь имеет возможность видеть, какие шаблоны он может использовать для заливки фигур. Для этого создадим в программе специальную диалоговую (инструментальную) панель, на которой будут представлены шаблоны. Щелчком мышки пользователь сможет выбрать понравившийся ему шаблон.

Диалоговая панель представляет собой разновидность немодального диалога, т. е. для продолжения работы с программой пользователю не обязательно завершать работу с этим диалогом. Поэтому такая панель может постоянно находиться на экране. Создание диалоговой панели выполняется в два этапа. Сначала создается объект MFC-класса `CDialogBar`. Затем с помощью метода `CDialogBar::Create()` создается Windows-окно диалоговой панели и связывается с объектом.

Для реализации этих новшеств выполним следующие действия.

1. Добавим в ресурсы программы 8 картинок размером 32×32 пиксела с количеством цветов 256, которые послужат нам в качестве шаблонов (рис. 9.14). Надо отметить, что шаблон кисти может иметь и больший размер.

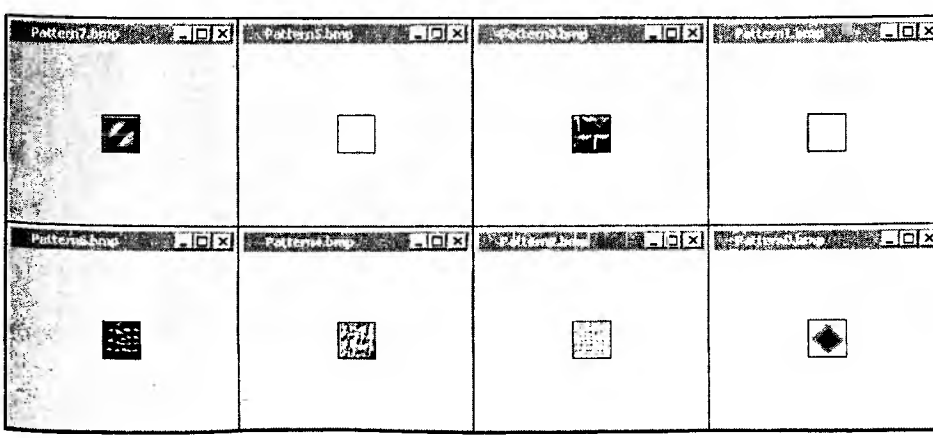


Рис. 9.14. Шаблоны кисти

2. Создадим шаблон диалога, который будет использован для создания инструментальной панели. Добавим в ресурсы программы новый диалог и присвоим ему идентификатор `IDD_IMAGES`. Разместим внутри нового диалога элемент управления — "Список" (List Control) с идентификатором `IDC_IMAGE_LIST` (рис. 9.15).

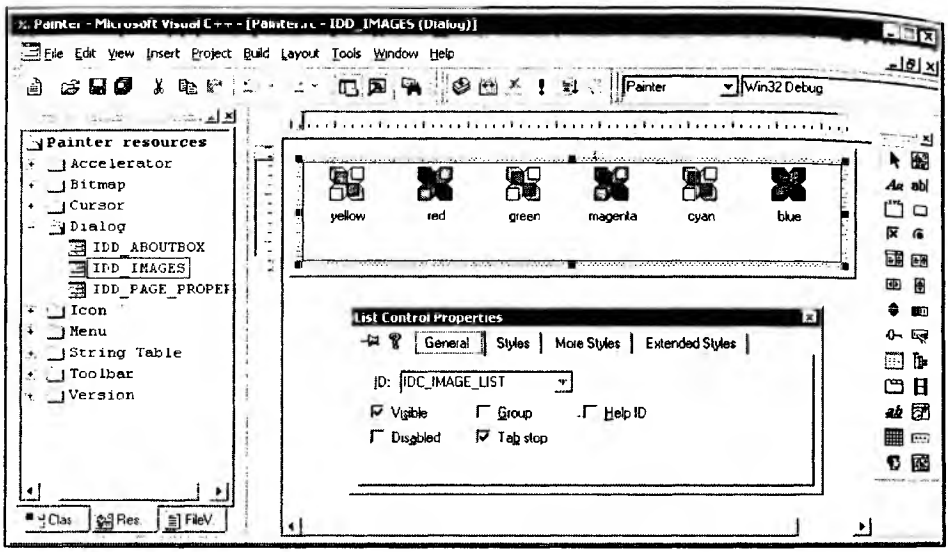


Рис. 9.15. Будущая диалоговая панель

Шаблон диалога должен иметь стиль `WS_CHILD` и не иметь стиля `WS_VISIBLE` (рис. 9.16).

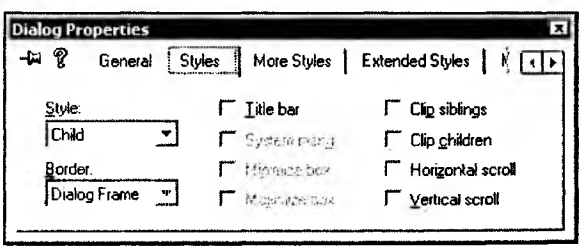


Рис. 9.16. Свойства шаблона диалога

3. Объект диалоговая панель сделаем членом класса `CMainFrame` (листинг 9.6). В класс `CMainFrame` также добавлен объект класса `CImageList`, который будет использован для хранения списка изображений и связан с элементом управления диалога. Кроме того, в интерфейс добавлены прототипы двух функций: `FillPatternsList()` и `SetSelectedPattern()`.

4. Добавим в метод `CMainFrame::OnCreate()` создание окна диалоговой панели (листинг 9.7). Окно создается с помощью метода `CDialogBar::Create()`. Прототип метода `Create()`:

```
BOOL Create(CWnd* pParentWnd, UINT nIDTemplate, UINT nStyle, UINT nID)
```

Здесь:

- `pParentWnd` — указатель на родительское окно;
- `nIDTemplate` — идентификатор шаблона диалога;
- `nStyle` — стиль и размещение панели;
- `nID` — идентификатор панели.

5. После создания окна диалоговой панели оно показывается на экране. Затем вызывается определенный нами метод `FillPatternsList()`, заполняющий список изображений. Реализация этого метода приведена в листинге 9.8. Изображениями шаблонов кистей заполняется объект `m_PatternsList`, который затем связывается с элементом управления "Список" — идентификатором `IDC_IMAGE_LIST`, который мы вставили в шаблон диалога. Класс MFC `CImageList`, объектом которого является `m_PatternsList`, позволяет хранить в виде списка набор одинаковых растровых картинок. В нашем случае такими картинками являются изображения шаблонов заливки. Объект класса `CImageList` можно связать с элементом управления "Список". Тогда каждому элементу (пункту) списка (названию шаблона) можно сопоставить изображение. Это и происходит при добавлении названий шаблонов `Caption` в список диалога:

```
pImageListCtrl->InsertItem( i, Caption, i )
```

Здесь:

- первый параметр метода `InsertItem()` — индекс добавляемого элемента;
- второй параметр — текст;
- третий параметр — индекс картинки в списке изображений.

6. Диалоговое окно будет посылать сообщения-уведомления родительскому окну-рамке. Для перехвата и обработки этих сообщений добавим с помощью ClassWizard в класс `CMainFrame` метод-обработчик сообщения `WM_NOTIFY` (листинг 9.9). В этом методе проверяется, от какого элемента управления пришло сообщение,. и если от списка шаблонов поступило сообщение "Ой, меня щелкнули мышью", вызывается метод `SetSelectedPattern()` (листинг 9.10).

7. В методе `SetSelectedPattern()` вызывается метод класса `CPainterDoc: SetPatternForSelected()`, которого на самом деле еще нет в этом классе. Для того чтобы этот метод появился в классе `CPainterDoc`, добавим его туда (листинг 9.11).

Листинг 9.6. Фрагмент интерфейса класса `CMainFrame`. Файл MainFrm.h

```
class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
   CMainFrame();
   DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:
   CDialogBar m_dlgBarImages; // панель изображений
   CImageList m_PatternsList; // список изображений шаблонов заливки
// Operations
public:
   // Заполняет список шаблонов заливки
   void FillPatternsList();
   // Устанавливает текущий шаблон для выделенной фигуры
   void SetSelectedPattern();
...
```

Листинг 9.7. Метод `CMainFrame::OnCreate()`. Файл MainFrm.cpp

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_TOP
      | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
      TRACE0("Failed to create toolbar\n");
      return -1;      // fail to create
```

```
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
         sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;      // fail to create
    }

    // TODO: Delete these three lines if you don't want the toolbar to
    //  be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);

    // панель изображений
    if (!m_dlgBarImages.Create(this, IDD_IMAGES, WS_CHILD | CBRS_BOTTOM,
         IDD_IMAGES))
    {
        TRACE0("Failed to create dialog bar\n");
        return -1;
    }
    m_dlgBarImages.ShowWindow(SW_SHOW);
    FillPatternsList();

    return 0;
}
```

Листинг 9.8. Метод `CMainFrame::FillPatternsList()`. Файл MainFrm.cpp

```
void CMainFrame::FillPatternsList()
{
    m_PatternsList.DeleteImageList();
    // Создаем список изображений
```

```
m_PatternsList.Create( 32, 32, ILC_COLOR24, 0, 1 );
// Загружаем все изображения и добавляем в список
CBitmap bm;
int index=0; // индекс добавленной картинки
bm.LoadBitmap(IDB_PATTERN0);
index=m_PatternsList.Add(&bm, RGB(0, 0, 0)); bm.DeleteObject( );

bm.LoadBitmap(IDB_PATTERN1);
index=m_PatternsList.Add(&bm, RGB(0, 0, 0)); bm.DeleteObject( );

bm.LoadBitmap(IDB_PATTERN2);
index=m_PatternsList.Add(&bm, RGB(0, 0, 0)); bm.DeleteObject( );

bm.LoadBitmap(IDB_PATTERN3);
index=m_PatternsList.Add(&bm, RGB(0, 0, 0)); bm.DeleteObject( );

bm.LoadBitmap(IDB_PATTERN4);
index=m_PatternsList.Add(&bm, RGB(0, 0, 0)); bm.DeleteObject( );

bm.LoadBitmap(IDB_PATTERN5);
index=m_PatternsList.Add(&bm, RGB(0, 0, 0)); bm.DeleteObject( );

bm.LoadBitmap(IDB_PATTERN6);
index=m_PatternsList.Add(&bm, RGB(0, 0, 0)); bm.DeleteObject( );

bm.LoadBitmap(IDB_PATTERN7);
index=m_PatternsList.Add(&bm, RGB(0, 0, 0)); bm.DeleteObject( );
// Получаем указатель на элемент управления "Список" диалоговой панели
CListCtrl* pImageListCtrl = (CListCtrl*)
    m_dlgBarImages.GetDlgItem(IDC_IMAGE_LIST);
ASSERT(pImageListCtrl!=NULL);
pImageListCtrl->DeleteAllItems( );
// Добавляем в "Список" названия шаблонов
CString Caption;
```

```
    for(int i=0; i<=index; i++)
    {
        Caption.Format("Pat %d",i+1);
        pImageListCtrl->InsertItem( i, Caption, i );
    }
    // Связываем элемент управления диалога "Список"
    // со списком изображений шаблонов
    pImageListCtrl->SetImageList(&m_PatternsList, LVSIL_NORMAL);
}
```

Листинг 9.9. Метод `CMainFrame::OnNotify()`. Файл MainFrm.cpp

```
BOOL CMainFrame::OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
    if(wParam==IDC_IMAGE_LIST && ((NMHDR*)lParam)->code == NM_CLICK)
        SetSelectedPattern();// установить выбранный шаблон
    return CFrameWnd::OnNotify(wParam, lParam, pResult);
```

Листинг 9.10. Метод `CMainFrame:: SetSelectedPattern ()`. Файл MainFrm.cpp

```
void CMainFrame::SetSelectedPattern()
{
    CListCtrl* pImageListCtrl = (CListCtrl*)
        m_dlgBarImages.GetDlgItem(IDC_IMAGE_LIST);
    ASSERT(pImageListCtrl!=NULL);
    POSITION pos = pImageListCtrl->GetFirstSelectedItemPosition();
    int nItem=-1;
    if (pos != NULL)
        nItem = pImageListCtrl->GetNextSelectedItem(pos);
    if (nItem<0) return; // ничего не выделено
    UINT Pattern_ID=0;
    switch(nItem)
    {
        case 0: Pattern_ID=IDB_PATTERN0; break;
        case 1: Pattern_ID=IDB_PATTERN1; break;
```

```
        case 2: Pattern_ID=IDB_PATTERN2; break;
        case 3: Pattern_ID=IDB_PATTERN3; break;
        case 4: Pattern_ID=IDB_PATTERN4; break;
        case 5: Pattern_ID=IDB_PATTERN5; break;
        case 6: Pattern_ID=IDB_PATTERN6; break;
        case 7: Pattern_ID=IDB_PATTERN7; break;
    }

    // Получим указатель на документ
    CPainterDoc *pDoc=(CPainterDoc*)GetActiveDocument();
    // Установили шаблон заливки
    pDoc->SetPatternForSelected(Pattern_ID);
    // Обновили изображение
    pDoc->UpdateAllViews(NULL);
}
```

Листинг 9.11. Метод `CPainterDoc::SetPatternForSelected()`. Файл PainterDoc.cpp

```
void CPainterDoc::SetPatternForSelected(UINT Pattern_ID)
{
    if(m_pSelShape==NULL) return;
    m_pSelShape->SetBrush(RGB(0,0,0), Pattern_ID);
};
```

9.6. Универсальная функция загрузки графических ресурсов

Выше мы рассмотрели, как загружать графические ресурсы с использованием методов различных классов MFC. Однако в наборе Windows API имеется функция `LoadImage()`, которую можно использовать для загрузки графических ресурсов всех типов. Причем данная функция позволяет загрузить изображения не только из ресурсов программы, но и из файлов. Прототип этой функции следующий:

```
HANDLE LoadImage(
    HINSTANCE hinst,    // дескриптор приложения, содержащего ресурс
    LPCTSTR lpszName,   // имя файла или идентификатор изображения
```

```
UINT uType,          // тип изображения
int cxDesired,       // ширина изображения
int cyDesired,       // высота изображения
UINT fuLoad          // флаги загрузки
);
```

Параметр `uType` обозначает тип загружаемого изображения:

- `IMAGE_BITMAP` — растровое изображение;
- `IMAGE_CURSOR` — курсор;
- `IMAGE_ICON` — пиктограмма.

Параметры `cxDesired`, `cyDesired` обозначают ширину и высоту загружаемого изображения в пикселах. Если эти параметры равны нулю, то для курсоров и пиктограмм используются системные метрики (`SM_CXICON`, `SM_CYICON`) или (`SM_CXCURSOR`, `SM_CYCURSOR`), соответственно. Если `uType` равен `IMAGE_BITMAP`, то эти параметры должны быть равны нулю.

Параметр `fuLoad` может быть комбинацией следующих флагов:

- `LR_DEFAULTCOLOR` — флаг по умолчанию, означает "не `LR_MONOCHROME`";
- `LR_CREATEDIBSECTION` — если `uType` равен `IMAGE_BITMAP`, то этот флаг указывает, что не нужно приводить цвета изображения к совместимым с контекстом устройства. Функция возвращает DIB-компонент растрового изображения;
- `LR_DEFAULTSIZE` — если `cxDesired`, `cyDesired` равны нулю, то следует использовать системные метрики для пиктограмм и курсоров. Если этот флаг не установлен и `cxDesired`, `cyDesired` равны нулю, то функция использует реальные размеры изображений;
- `LR_LOADFROMFILE` — загружает изображения из файла, имя которого указано параметром `lpszName`. Если флаг не установлен, функция считает, что `lpszName` — это идентификатор ресурса;
- `LR_LOADMAP3DCOLORS` — заменяет серые цвета изображения системными цветами "трехмерных" изображений Windows. Например, цвет `RGB(128,128,128)` будет заменен на `COLOR_3DSHADOW`;
- `LR_LOADTRANSPARENT` — все пикселы, имеющие цвет, совпадающий с цветом первого пиксела, будут окрашены в цвет окна по умолчанию (`COLOR_WINDOW`). Работает только при загрузке изображений с глубиной цвета не более 8 бит на пиксел (256 цветов);
- `LR_MONOCHROME` — загружает изображение в двух цветах: черном и белом;
- `LR_SHARED` — разделяет дескриптор изображения, если оно уже загружено. Если флаг не установлен, то каждый вызов функции для одного и того же ресурса загружает его снова и возвращает новый дескриптор. Этот флаг не должен быть установлен при загрузке изображений нестан-

дартных размеров или изображений, которые могут быть изменены после загрузки, а также при загрузке из файла. Данный флаг должен быть установлен при загрузке системных пиктограмм или курсоров;

- LR_VGACOLOR — использовать VGA-цвета.

С использованием функции LoadImage() мы могли бы загрузить изображения прямо из BMP-файлов (листинг 9.12).

Листинг 9.12. Модифицированный метод CMainFrame::FillPatternsList(). Файл MainFrm.cpp

```
#define N_PATTERNS 8
void CMainFrame::FillPatternsList()
{
   m_PatternsList.DeleteImageList();
   // Создаем список изображений
   m_PatternsList.Create( 32, 32, ILC_COLOR24, 0, 1 );

   // Загружаем все изображения и добавляем изображение в список
   CBitmap bm;
   int index=0; // индекс добавленной картинки
   // Загрузка шаблонов заливки из файлов BMP на диске
   CString Path, PatternName;
   // Получили название текущего каталога
   GetCurrentDirectory( MAX_PATH, Path.GetBuffer(MAX_PATH));
   // А так можно узнать, где располагается exe-файл программы,
   // если хотим указать положение каталога с шаблонами
   // относительно положения  exe-файла
   // GetModuleFileName(AfxGetInstanceHandle(),
   //    Path.GetBuffer(MAX_PATH), MAX_PATH);

   Path.ReleaseBuffer();
   for(int n=0; n<N_PATTERNS; n++)
   {
      // Формируем путь и имя файла шаблона
      PatternName.Format("\\Patterns\\Pattern%d.bmp", n);
      PatternName=Path+PatternName;
      HBITMAP hBMP=NULL;
```

```
        // Загружаем шаблон
        hBMP=(HBITMAP)LoadImage(NULL, PatternName, IMAGE_BITMAP, 0, 0,
             LR_LOADFROMFILE|LR_DEFAULTCOLOR);
        // Добавляем изображение шаблона в список шаблонов
        index=m_PatternsList.Add(bm.FromHandle(hBMP) , RGB(0, 0, 0));
        bm.DeleteObject( );
    }

    // Получаем указатель на элемент управления "Список" диалоговой панели
    CListCtrl* pImageListCtrl = (CListCtrl*)
        m_dlgBarImages.GetDlgItem(IDC_IMAGE_LIST);
    ASSERT(pImageListCtrl!=NULL);
    pImageListCtrl->DeleteAllItems( );
    // Добавляем в "Список" названия шаблонов
    CString Caption;
    for(int i=0; i<=index; i++)
    {
        Caption.Format("Pat %d",i+1);
        pImageListCtrl->InsertItem( i, Caption, i );
    }
    // Связываем список-элемент управления диалога
    // со списком изображений шаблонов
    pImageListCtrl->SetImageList(&m_PatternsList, LVSIL_NORMAL);
}
```

Однако в этом случае возникнут сложности при сохранении изображений, созданных в программе Painter. Если мы используем для рисования какой-нибудь фигуры кисть, шаблон которой загружается из файла, то при отсутствии файла с шаблоном рисунок уже не сможет быть правильно отображен. Поэтому придется каким-то образом сохранять шаблоны вместе с рисунком или распространять вместе с программой.

9.7. Заключение

В этой главе мы рассмотрели средства, которые позволяют улучшить интерфейс и расширить изобразительные возможности программы Painter (рис. 9.17). Исходный код программы содержится на компакт-диске в каталоге Sources\Painter4.1.

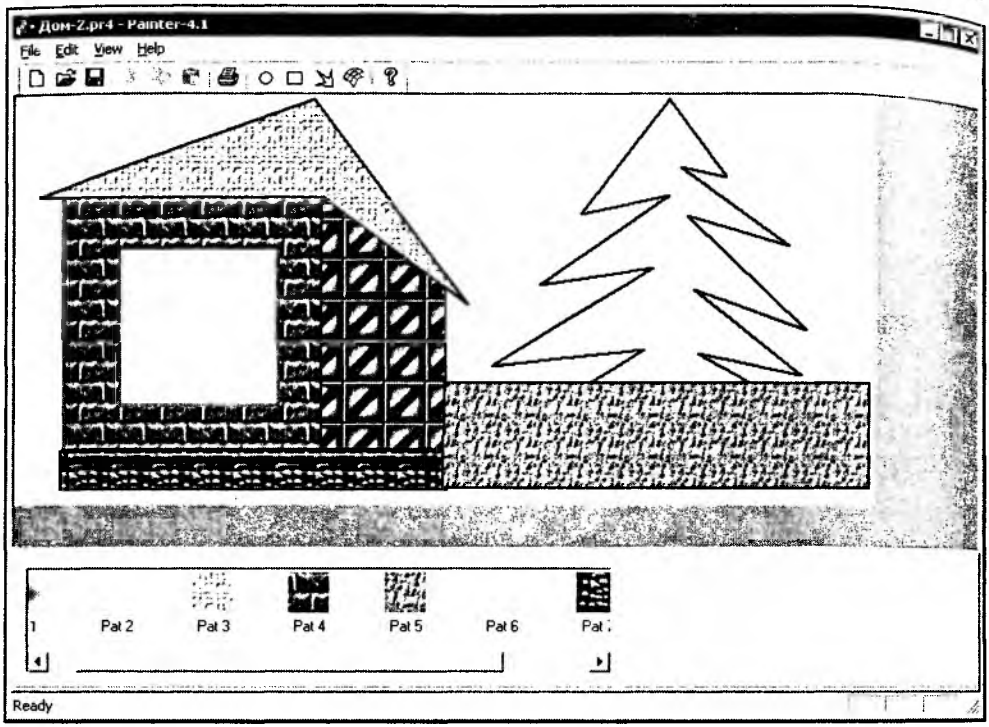


Рис. 9.17. Программа Painter с новым интерфейсом

В следующих главах продолжим работу с растровой графикой. Дополнительную информацию о работе с ресурсами приложения можно найти в [7].

Глава 10

Экспорт изображений в BMP-файл

В этой главе рассматриваются:

- растровый формат Microsoft Windows Bitmap (BMP);
- экспорт изображений из программы Painter в BMP-файл (программа Painter 4.2).

10.1. Общее описание формата BMP

Microsoft Windows Bitmap (BMP) — собственный растровый формат операционной системы Windows. Мы думаем, можно смело утверждать, что все Windows-приложения, предназначенные для работы с изображениями, поддерживают формат BMP. Формат основан на внутренних структурах представления растровых данных Windows, но, несмотря на это, поддерживается многими "не Windows"- и даже "не PC"-приложениями. Формат совершенствовался и развивался по мере появления новых версий Windows. Первоначально он был очень простым, содержал лишь растровые данные и не поддерживал сжатие. Растровые данные представляли собой индексы в цветовой палитре, которая была фиксированной и определялась графической платой. Поэтому этот формат называют аппаратно-зависимым (Device Dependent Bitmap, DDB), он был ориентирован на графические платы для IBM PC (CGA, EGA, Hercules) и другие.

Развитием формата BMP стало введение в него поддержки изменяемой цветовой палитры. Это позволило хранить информацию о цветах вместе с растровыми данными. Такое изменение формата позволило сделать хранимые изображения аппаратно-независимыми (Devise Independent Bitmap, DIB). Иногда аббревиатуру DIB используют как синоним BMP.

Уже существует по крайней мере четыре Windows-версии формата BMP и две версии формата для операционной системы OS/2. Возможно, при создании новых версий операционной системы Windows в формат будут и дальше

вноситься изменения. В каждой новой версии в заголовке растра появляется новая информация. Однако грамотно написанные программы, осуществляющие чтение и отображение изображений в новом формате, способны работать и со старыми форматами.

10.2. Структура файла

Файлы DDB исходного формата BMP содержали два раздела: заголовок файла и растровые данные (рис. 10.1).

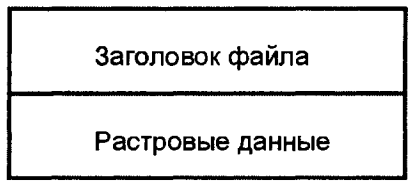


Рис. 10.1. Структура файла DDB исходного формата BMP

Файлы более поздних версий содержат четыре раздела: заголовок файла, информационный заголовок растра, палитру цветов и растровые данные (рис. 10.2).

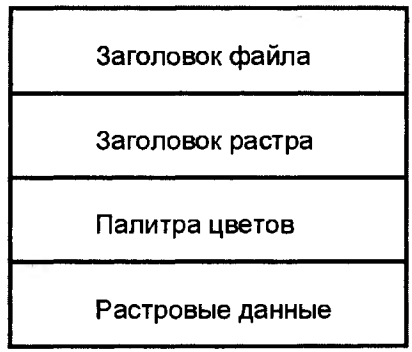


Рис. 10.2. Структура BMP-файла

Рассмотрим в деталях структуру данных файла формата BMP версии 3.*x*, появившегося с операционной системой Microsoft Windows 3.*x*. Этот формат поддерживается большинством существующих в настоящее время приложений.

Все версии формата BMP начинаются с 14-байтового заголовка-структуры BITMAPFILEHEADER:

```
typedef struct tagBITMAPFILEHEADER
{
   WORD  bfType;       // Тип файла, должен быть 4d42h ("BM")
```

```
  DWORD bfSize;       // Размер файла в байтах
  WORD  bfReserved1; // Зарезервировано, должен быть 0
  WORD  bfReserved2; // Зарезервировано, должен быть 0
  DWORD bfOffBits;    // Смещение в байтах до начала растровых данных
} BITMAPFILEHEADER;
```

Поле `bfType` содержит 2-байтовое число-идентификатор типа файла, его значение должно быть равно `4D42h` или `BM` (в формате ASCII).

Поле `bfSize` содержит общий размер файла BMP в байтах. В несжатых файлах это поле может быть равно нулю. Размер файла в этом случае можно узнать у операционной системы.

Поля `bfReserved1` и `bfReserved2` не содержат данных и обычно устанавливаются в нуль. Программа, работающая с файлами BMP, может использовать эти поля для своих целей.

В поле `bfOffBits` хранится смещение в байтах от начала файла до начала растровых данных. Эту информацию можно использовать для быстрого доступа к растровым данным.

За заголовком файла следует заголовок растра `BITMAPINFOHEADER`. Его длина составляет 40 байтов.

```
typedef struct tagBITMAPINFOHEADER
{
  DWORD  biSize;          // Размер этого заголовка в байтах
  LONG   biWidth;         // Ширина изображения в пикселах
  LONG   biHeight;        // Высота изображения в пикселах
  WORD   biPlanes;        // Количество цветовых плоскостей
  WORD   biBitCount       // Количество битов на пиксел
  DWORD  biCompression;   // Используемые методы сжатия
  DWORD  biSizeImage;     // Размер растра в байтах
  LONG   biXPelsPerMeter; // Горизонтальное разрешение
  LONG   biYPelsPerMeter; // Вертикальное разрешение
  DWORD  biClrUsed;       // Количество цветов в изображении
  DWORD  biClrImportant;  // Минимальное количество "важных" цветов
} BITMAPINFOHEADER;
```

Поле `biSize` указывает размер заголовка `BITMAPINFOHEADER` в байтах. Поля `biWidth` и `biHeight` определяют соответственно ширину и высоту изображения в пикселах. Если `biHeight` — положительное число, то изображение представляет собой растр с началом в левом нижнем углу. Если `biHeight` — отрицательное, то начало растра в левом верхнем углу.

Поле `biPlanes` — количество цветовых плоскостей, в BMP-файлах одна цветовая плоскость, поэтому значением этого поля всегда является единица.

В поле biBitCount указывается количество бит, отводимых под один пиксел. Допустимые значения 1, 4, 8, 24.

Поле biCompression содержит идентификатор используемого метода сжатия. Значение 0 этого поля указывает на то, что данные не сжаты, 1 — был применен 8-битовый алгоритм сжатия RLE; 2 — был применен 4-битовый алгоритм RLE.

Поле biSizeImage задает размер растровых данных в байтах. Бывает, что для несжатых растровых данных значение этого поля равно нулю. В этом случае их размер может быть вычислен на основе значений полей biHeight, biWidth и biBitCount.

Поля biXPelsPerMeter и biYPelsPerMeter содержат информацию соответственно о горизонтальном и вертикальном разрешении, выраженном в пикселах на метр. Эта информация позволяет определить физические размеры изображения при выводе на печать.

В поле biClrUsed указывается количество используемых цветов в палитре. Например, при biBitCount, равном 8, палитра может содержать до 256 цветов, если же реально в изображении задействовано меньшее количество цветов, то его можно указать в поле biClrUsed. Значение поля biClrUsed определяет, сколько места нужно отвести под хранение палитры. Если значение этого поля равно нулю, то количество цветов в палитре рассчитывается на основе значения поля biBitCount.

Поле biClrImportant содержит минимальное количество цветов, которые могут быть использованы для адекватного воспроизведения изображения. Такие цвета стараются поместить в начало палитры. Если устройство не способно отразить всю палитру цветов, то оно использует только начало палитры. Обычно значение этого поля равно нулю.

За заголовком растра может следовать палитра цветов. Палитра цветов состоит из последовательности 4-байтовых структур RGBQUAD:

```
typedef struct _RGBQUAD
{
   BYTE   rgbBlue;      // Синяя составляющая
   BYTE   rgbGreen;     // Зеленая составляющая
   BYTE   rgbRed;       // Красная составляющая
   BYTE   rgbReserved;  // Заполнитель (всегда 0)
} RGBQUAD;
```

Значения цветовых составляющих хранятся в полях rgbBlue, rgbGreen, rgbRed. Поле rgbReserved не используется.

Структура BITMAPINFOHEADER и структуры RGBQUAD собираются в структуре BITMAPINFO:

```
typedef struct tagBITMAPINFO {
```

```
    BITMAPINFOHEADER    bmiHeader;
    RGBQUAD             bmiColors[1];
} BITMAPINFO;
```

Количество элементов в массиве `bmiColors[]` соответствует количеству цветов в палитре изображения.

После структуры `BITMAPINFO` на расстоянии `bfOffBits` (поле структуры `BITMAPFILEHEADER`) от начала файла начинаются растровые данные. Растровые данные представляют собой индексы в палитре цветов (в случае если `biBitCount` равно 1, 4, 8) или реальные значения цветов пикселов (в случае если `biBitCount` равно 24). Если `biBitCount` равно 24, то каждый пиксел представляется тремя байтами: первый байт — интенсивность синего цвета, затем по байту на зеленый и красный цвет. Этот формат цвета называется RGB888 или RGB24.

Растровые данные, соответствующие одной строке пикселов изображения, вне зависимости от формата цвета должны быть выровнены на границу двойного слова `DWORD`, т. е. каждая строка пикселов должна описываться целым числом двойных слов. Например, строка из 5 пикселов по 24 бита (3 байта) на пиксел может быть описана 15 байтами, но длина строки растровых данных в формате BMP должна быть 16 байтов. Последний байт будет служить лишь для целей выравнивания.

Формат BMP версии 3.*x* имеет разновидность (для Windows NT), предназначенную для хранения растровых данных с пиксельной глубиной 16 и 32 битов. Этот формат имеет точно такую же структуру заголовка растра `BITMAPINFOHEADER`. Его длина составляет 40 байтов. Отличие заключается в том, что поле `biBitCount` может принимать значения 16 и 32.

При пиксельной глубине 16 битов для хранения цвета пиксела отводится два байта (слово — тип `WORD`), каждому компоненту цвета пиксела отводится по 5 битов (формат цвета RGB555). Младшие 5 битов задают интенсивность синего цвета, затем по 5 битов на зеленый и красный цвет, старший бит в слове не используется.

При пиксельной глубине 32 бита для хранения цвета пиксела отводится 4 байта (двойное слово — тип `DWORD`). При этом на каждый компонент цвета отводится по 8 бит, так же как и при 24-битной глубине, а старший байт в `DWORD` не используется (формат цвета RGB888).

Дополнительные возможности этой разновидности формата проявляются, если указать значение поля `biCompression`, равное 3. В этом случае вслед за структурой `BITMAPINFOHEADER` (на месте палитры цвета) следуют три поля `DWORD`: `RedMask`, `GreenMask`, `BlueMask`, которые задают битовые маски для компонентов цвета пиксела. Биты в этих масках обязательно должны быть смежными и не содержать перекрывающихся полей.

Для 16-битовых растровых данных часто применяют формат RGB565, который задается следующей маской.

```
RedMask   = 0xF8000000; // 1111 1000 0000 0000 0000 0000 0000 0000
GreenMask = 0x07E00000; // 0000 0111 1110 0000 0000 0000 0000 0000
BlueMask  = 0x001F0000; // 0000 0000 0001 1111 0000 0000 0000 0000
```

С помощью этой маски из значения WORD, задающего цвет пиксела, извлекается значение каждого цветового компонента. В формате RGB565 красному и синему цветам отводится по 5 битов, а зеленому — 6 битов. Такое неравноправие обосновывают тем, что человеческий глаз более восприимчив к зеленому цвету, поэтому тщательная запись его градаций позволяет повысить качество изображения.

Для 32-битовых растровых данных используют формат RGB101010, определяющий по 10 битов на каждый цвет, который задается следующей маской.

```
RedMask   = 0xFFC00000; // 1111 1111 1100 0000 0000 0000 0000 0000
GreenMask = 0x003FF000; // 0000 0000 0011 1111 1111 0000 0000 0000
BlueMask  = 0x00000FFC; // 0000 0000 0000 0000 0000 1111 1111 1100
```

По сравнению с форматом RGB888 такое представление позволяет описать большее количество цветов.

10.3. Экспорт рисунков в растровый файл формата BMP

Рисунки, созданные в программе Painter, могут быть сохранены в файл в некотором своем формате. Однако недостаток таких файлов в том, что они могут быть прочитаны лишь программой Painter. Для того чтобы можно было просматривать и редактировать изображения, созданные нашей программой, в других редакторах, добавим функцию сохранения изображения в файл BMP-формата. Рисунки, сохраненные в формате BMP, могут быть загружены практически любым растровым редактором, например Adobe Photoshop или программой Microsoft Paint, входящей в набор стандартных Windows-приложений.

Для сохранения изображения в файл BMP-формата нам потребуется:

1. Вывести рисунок на растр, совместимый с контекстом устройства отображения (экрана).
2. Записать рисунок в файл.
3. Добавить соответствующую команду в меню программы Painter.

Программа Painter создает векторные изображения и позволяет использовать разные растровые заливки фигур. Растрирование такого рисунка может оказаться сложной задачей. Но тут нам на помощь придут классы MFC

и API-функции Windows. Растрирование выполняется при выводе рисунка на экран или принтер. Точно так же мы можем записать изображение в некоторую область памяти. Для этого дополним наш класс `CPainterView` методом `SaveBMP`, в котором и выполним перевод нашего рисунка в растровое представление (листинг 10.1). После растрирования рисунка он записывается в файл с помощью глобальной функции `SaveBitmapToBMPFile`, прототип которой определен в файле Savebmp.h (листинг 10.2), а реализация — в файле Savebmp.cpp (листинг 10.3). Вот почти и все, осталось только добавить в меню команду **Save as BMP**, а в класс `CPainterView` — метод-обработчик `OnSaveBmp()`. В этом методе с помощью стандартного Windows-диалога будем получать имя файла и вызывать метод `SaveBMP()` (листинг 10.4).

Листинг 10.1. Метод `CPainterView::SaveBMP()`. Файл PainterView.cpp

```
void CPainterView::SaveBMP(CString &FileName)
{
  // Получим контекст устройства, на котором рисуем
  CDC *pwDC=GetDC();
  // Подготовим контекст устройства
  OnPrepareDC(pwDC);

  // Запомним прежнюю позицию прокрутки
  CPoint ScrollPos=GetScrollPosition();
  // Установим позицию прокрутки в начало,
  // чтобы нарисовалась вся картинка
  ScrollToPosition(CPoint(0,0));

  // Контекст устройства для памяти, в которую будет происходить вывод
  CDC MemDC;
  // Создадим контекст устройства, совместимый с экраном
  MemDC.CreateCompatibleDC(pwDC);
  // Подготовим контекст
  // Этот метод установит режим отображения, в котором мы работаем
  OnPrepareDC(&MemDC);

  // Узнаем логические и физические размеры рисунка
  // Получили указатель на объект-документ
  CPainterDoc *pDoc=GetDocument();
  CSize   SizeTotal,    // это логический размер
```

```
        SizeBMP;      // это физический размер
    SizeTotal.cx = pDoc->m_wSheet_Width; // ширина
    SizeTotal.cy = pDoc->m_wSheet_Height; // высота
    SizeBMP=SizeTotal;
    MemDC.LPtoDP(&SizeBMP); // Перевод логического размера в физический

    // Создадим растровую "заготовку", на которой будем рисовать
    CBitmap BMP;
    BMP.CreateCompatibleBitmap(pwDC, SizeBMP.cx, SizeBMP.cy);
    // Установим заготовку в контекст памяти
    MemDC.SelectObject(&BMP);

    // Белый фон
    CBrush brush;
    CBrush *pBrushOld;
    if(brush.CreateSolidBrush(RGB(255,255,255)))
    {
        pBrushOld=MemDC.SelectObject(&brush);
        MemDC.PatBlt(0, 0, SizeTotal.cx, SizeTotal.cy, PATCOPY);
        MemDC.SelectObject(pBrushOld);
    }

    // Вывод рисунка в контекст памяти
    OnDraw(&MemDC);

    // Теперь запишем полученную растровую картинку в файл
    SaveBitmapToBMPFile(FileName, BMP, MemDC);

    // Контекст экрана больше не нужен
    ReleaseDC(pwDC);
    // Вернем позицию прокрутки
    ScrollToPosition(ScrollPos);
}
```

Листинг 10.2. Прототипы функций записи растра в файл. Файл Savebmp.h

```
// Макрос для определения количества байтов в выровненной
// по DWORD строке пикселов в DIB
// Width — длина строки в пикселах; BPP — битов на пиксел
```

```
#define BYTESPERLINE(Width, BPP) ((WORD)((((DWORD)(Width) *
↳(DWORD)(BPP) + 31) >> 5)) << 2)

BOOL SaveBitmapToBMPFile(CString &FileName, CBitmap &BMP, CDC &DC);
PBITMAPINFO CreateBitmapInfoStruct(HBITMAP hBmp);
```

Листинг 10.3. Функции `CreateBitmapInfoStruct()` и `SaveBitmapToBMPFile()`. Файл Savebmp.cpp

```
#include "stdafx.h"
#include "SaveBMP.h"

PBITMAPINFO CreateBitmapInfoStruct(HBITMAP hBmp)
{
    BITMAP bmp;
    PBITMAPINFO pbmi;
    WORD    cClrBits;

    // Получаем размер картинки и количество битов на пиксел (формат цвета)
    if (!GetObject(hBmp, sizeof(BITMAP), (LPSTR)&bmp))
        return NULL;
    // Преобразование формата цвета к стандартному числу битов
   if(bmp.bmBitsPixel>24) bmp.bmBitsPixel=24;
      cClrBits = (WORD)(bmp.bmBitsPixel);
   if (cClrBits == 1)
      cClrBits = 1;
   else if (cClrBits <= 4)
        cClrBits = 4;
   else if (cClrBits <= 8)
        cClrBits = 8;
   else if (cClrBits <= 16)
        cClrBits = 16;
   else
        cClrBits = 24;

    // Выделяем память для структуры  BITMAPINFO
   // с учетом размера структуры BITMAPINFOHEADER
   if (cClrBits < 16)
```

```
        // учитываем также палитру цветов (массив структур RGBQUAD)
      pbmi = (PBITMAPINFO) new BYTE[sizeof(BITMAPINFOHEADER) +
                                    sizeof(RGBQUAD) * (1<<cClrBits)];
   else  // палитры нет
      pbmi = (PBITMAPINFO) new BYTE[ sizeof(BITMAPINFOHEADER)];

   // Заполняем поля структуры BITMAPINFO
   pbmi->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
   pbmi->bmiHeader.biWidth = bmp.bmWidth;
   pbmi->bmiHeader.biHeight = bmp.bmHeight;
   pbmi->bmiHeader.biPlanes = bmp.bmPlanes;
   pbmi->bmiHeader.biBitCount = bmp.bmBitsPixel;
   pbmi->bmiHeader.biXPelsPerMeter = 0;
   pbmi->bmiHeader.biYPelsPerMeter = 0;

   if (cClrBits < 16)
        pbmi->bmiHeader.biClrUsed = (1<<cClrBits);
     // Сжимать картинку не собираемся – ставим флаг BI_RGB
    pbmi->bmiHeader.biCompression = BI_RGB;
   // Вычисляем количество байтов, требуемых для хранения изображения
   // с учетом количества пикселов и битов на пиксел.
   // В файле BMP строки должны быть выровнены на границу 4 байт,
   // поэтому длина строки в байтах определяется с помощью
   // специального макроса (см. файл savebmp.h)
    pbmi->bmiHeader.biSizeImage =
                  BYTESPERLINE(pbmi->bmiHeader.biWidth, cClrBits)*
                  pbmi->bmiHeader.biHeight;
   // Считаем, что все цвета нашей картинки важны
   pbmi->bmiHeader.biClrImportant = 0;

   return pbmi;
}

BOOL SaveBitmapToBMPFile(CString &FileName, CBitmap &BMP, CDC &DC)
{
   HANDLE hf=NULL;              // указатель на файл
   BITMAPFILEHEADER hdr;        // заголовок BMP-файла
   PBITMAPINFOHEADER pbih;      // заголовок картинки
   BYTE * pBits=NULL;           // указатель на растровые данные
```

```
 DWORD dwWidthBytes=0;        // Длина строки растровых данных в байтах
 DWORD dwTmp=0;               // для временных потребностей
// Создаем структуру – заголовок растра
PBITMAPINFO pbmi=CreateBitmapInfoStruct((HBITMAP)BMP);

pbih = (PBITMAPINFOHEADER) pbmi;

// Выделяем память под растровые данные
pBits =  new BYTE[pbih->biSizeImage];
if (!pBits) return FALSE;

// Получаем растровые данные
// и таблицу цветов (массив структур RGBQUAD), если она есть
if (!GetDIBits(DC.m_hDC, (HBITMAP)BMP, 0, (WORD) pbih->biHeight,
                pBits, pbmi, DIB_RGB_COLORS))
   return FALSE;

// Создаем файл на диске
hf = CreateFile(FileName,
                GENERIC_READ | GENERIC_WRITE,
                (DWORD) 0,
                (LPSECURITY_ATTRIBUTES) NULL,
                CREATE_ALWAYS,
                FILE_ATTRIBUTE_NORMAL,
                (HANDLE) NULL);
if (hf == INVALID_HANDLE_VALUE) return FALSE;
// Идентификатор типа файла BMP: 0x42 = "B" 0x4d = "M"
hdr.bfType = 0x4d42;

// Размер всего файла вместе с заголовками и данными
hdr.bfSize = (DWORD) (sizeof(BITMAPFILEHEADER) +
             pbih->biSize + pbih->biClrUsed *
             sizeof(RGBQUAD) + pbih->biSizeImage);
hdr.bfReserved1 = 0;
hdr.bfReserved2 = 0;

// Вычисляем смещение до начала растровых данных
hdr.bfOffBits = (DWORD) sizeof(BITMAPFILEHEADER) +
                pbih->biSize + pbih->biClrUsed
```

```
                    * sizeof (RGBQUAD);

   // Записываем заголовок файла – структуру BITMAPFILEHEADER
   if (!WriteFile(hf, (LPVOID) &hdr, sizeof(BITMAPFILEHEADER),
      (LPDWORD) &dwTmp, (LPOVERLAPPED) NULL))
      return FALSE;

   // Записываем заголовок картинки – структуру BITMAPINFOHEADER
   // и палитру – массив RGBQUAD
   if (!WriteFile(hf, (LPVOID) pbih, sizeof(BITMAPINFOHEADER) +
                  pbih->biClrUsed * sizeof (RGBQUAD),
                  (LPDWORD) &dwTmp, (LPOVERLAPPED) NULL))
       return FALSE;

    // Записываем растровые данные
   dwWidthBytes = BYTESPERLINE(pbih->biWidth, pbih->biBitCount);
   LONG i=0, j=0;
   BYTE *pCurStr=NULL;    // указатель на текущую строку
   pCurStr=pBits;
   for(i=0; i<pbih->biHeight; i++) // записываем по строкам
   {
       if (!WriteFile(hf, (LPSTR) pCurStr, (int) dwWidthBytes,
                         (LPDWORD) &dwTmp, (LPOVERLAPPED) NULL))
          return FALSE;
      pCurStr+=dwWidthBytes;
   }

   // Закрываем файл
   if (!CloseHandle(hf)) return FALSE;

   // Освобождаем память
   if(pBits!=NULL) delete[] pBits;
   if(pbmi!=NULL) delete[] pbmi;
   return TRUE;
}
```

Несколько подозрительной выглядит в функции `CreateBitmapInfoStruct()` строка:

```
if(bmp.bmBitsPixel>24) bmp.bmBitsPixel=24;
```

Дело в том, что мы указываем значение поля biCompression=BI_RGB, т. е. 0, а если biCompression равно 0, то в случае bmBitsPixel, равного 32, данные поступят в формате RGB888, и нам нет смысла выделять место под лишний байт. Функция же GetDIBits(), с помощью которой извлекаются растровые данные, оказывается достаточно умна, чтобы преобразовать данные в соответствии со значением bmBitsPixel=24. Таким образом, данные будут представлены в том же формате RGB888, но занимать будут не 32 бита, а 24, как им и полагается.

Листинг 10.4. Метод CPainterView::OnSaveBmp(). Файл PainterView.cpp

```
void CPainterView::OnSaveBmp()
{
   CPainterDoc *pDoc=GetDocument();
   // Из названия картинки формируем имя файла
   CString FileName=pDoc->GetTitle();
   if(FileName.ReverseFind('.')>-1)
      FileName=FileName.Left(FileName.ReverseFind('.'));
   // Фильтр файлов
   CString Filter="BMP File (*.bmp)|*.bmp|All Files (*.*)|*.*||";
   CString DefExt="BMP";
   CFileDialog SaveDlg(FALSE, (LPCSTR)DefExt, (LPCSTR)FileName,
                  OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
                  (LPCSTR)Filter, this);
   if(SaveDlg.DoModal()==IDCANCEL) return;
   SaveBMP(SaveDlg.GetPathName());
}
```

10.4. Заключение

Ну вот, теперь программа Painter умеет экспортировать свои рисунки в растровый формат BMP. Рисунки сохраняются в том виде, в котором они представлены на экране. Причем сделано так, что экспортируется весь лист. В принципе можно ограничиться сохранением только той части рисунка, в которой имеется изображение. Если вы захотите задавать размер и разрешение растрового рисунка, как, например, это происходит при экспорте изображений из CorelDRAW, то, возможно, добиться этого удастся изменением режима отображения для контекста памяти в методе CPainterView::SaveBMP().

Исходные тексты программы приведены на компакт-диске в каталоге Sources\Painter4.2. На рис. 10.3 показана программа Painter с тестовым

рисунком. На рис. 10.4 — рисунок, экспортированный из программы Painter в BMP-файл (вы можете найти его в каталоге \Pics\Painter\ — файл Тест.BMP). На рис. 10.5 — тот же рисунок, но уже в растровом редакторе Microsoft Paint.

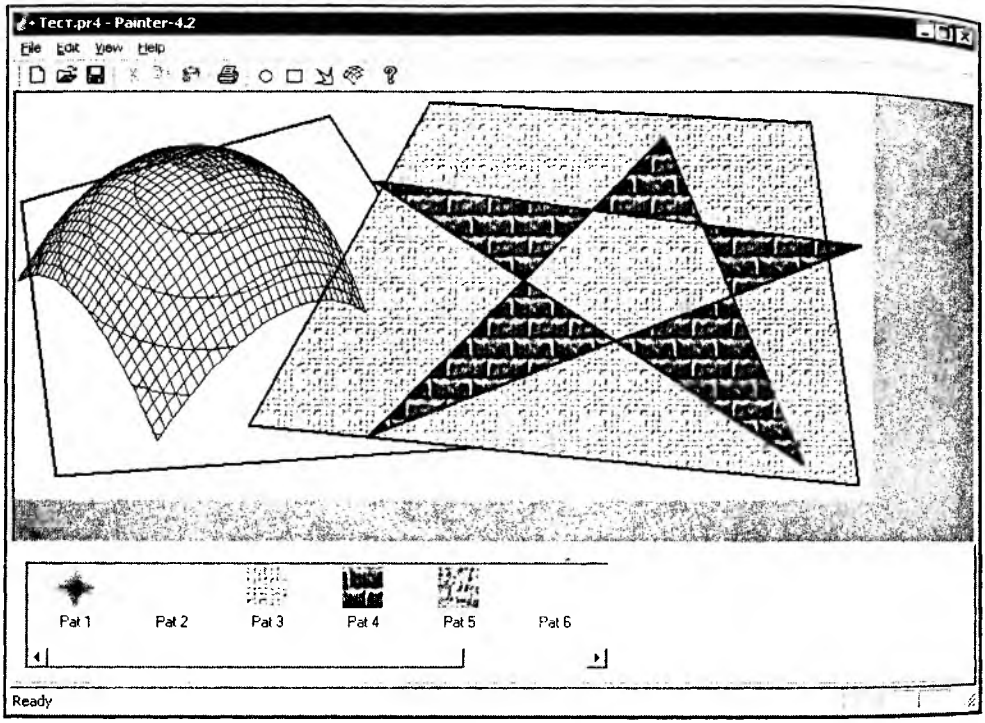


Рис. 10.3. Рисунок в программе Painter

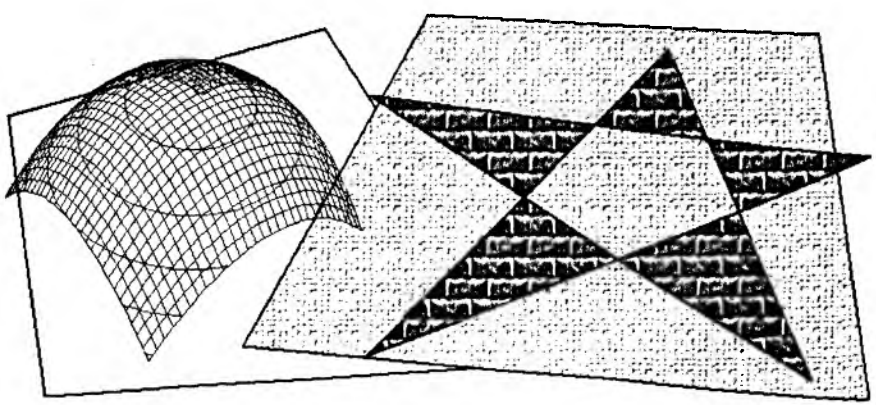


Рис. 10.4. Рисунок, экспортированный в BMP-формат

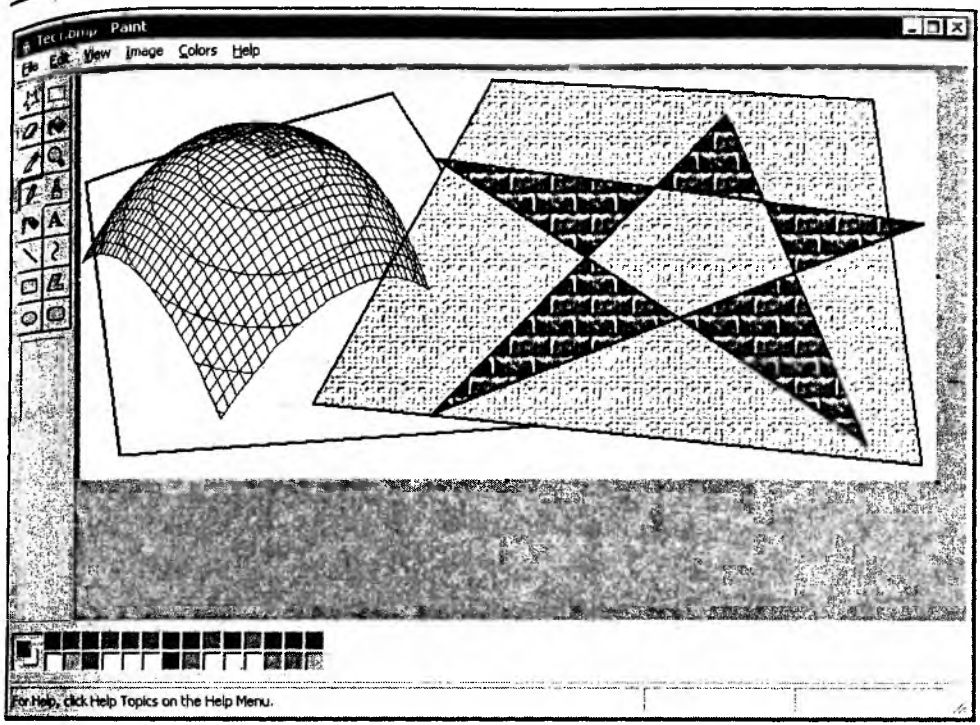


Рис. 10.5. Рисунок в растровом редакторе Microsoft Paint

Дополнительную информацию о формате BMP, а также описание большого числа других форматов графических файлов можно найти в книгах [6, 18, 19]. Кроме того, в *разд. 13.3.3* показано, насколько упрощается работа с файлами разных форматов (читай — жизнь программистов) при использовании средств библиотеки GDI+.

Глава 11

Просмотр и редактирование растровых изображений

В этой главе рассматриваются:

- создание многодокументного приложения;
- многопоточное выполнение программы;
- загрузка изображений из файлов формата BMP;
- вывод растровых изображений на экран;
- режимы масштабирования изображений;
- гистограмма изображений и ее использование;
- редактирование изображений с помощью точечных и пространственных фильтров;
- программная реализация преобразований (программа BMViewer).

11.1. Создание многодокументного приложения

До сих пор мы работали с программой, поддерживающей одновременную работу лишь с одним документом-изображением (SDI-интерфейс). Во многих случаях такая организация интерфейса вполне приемлема. Рассмотрим далее, как можно создать приложение с многодокументным интерфейсом (MDI-интерфейсом). Особых усилий от нас не потребуется. Всю основную работу сделает генератор приложений AppWizard.

Создадим каркас приложения для просмотра и редактирования рисунков:

1. С помощью команды **File | New | Projects | MFC AppWizard (exe)** начнем создание приложения. Назовем проект BMViewer.
2. На первом шаге выберем тип приложения Multiple documents.
3. До шестого шага можно принять все установки по умолчанию.

4. На шестом шаге генератора приложений изменим базовый класс облика с предложенного по умолчанию CView на CScrollView. Этот поступок не пройдет бесследно, а обеспечит нас впоследствии возможностью прокручивать изображения в окне облика, если они целиком в нем не поместятся. Здесь же можно изменить имена файлов, в которых будут размещены классы нашего приложения. Например, предложенное имя BMViewerView.h можно заменить на более лаконичное BMView.h.

Вот и все, каркас программы готов. Осталось теперь наделить его полезными качествами.

11.2. Класс *CRaster* для работы с растровыми изображениями

Структура файла с растровым изображением в формате Microsoft Windows Bitmap (BMP) была рассмотрена в прошлой главе, поэтому сразу перейдем к делу.

Создадим в программе специальный класс, который будет отвечать за загрузку и осуществлять поддержку операций по обработке растрового изображения. Назовем класс CRaster. Интерфейс класса CRaster приведен в листинге 11.1, а реализация методов — в листинге 11.2.

Листинг 11.1 Интерфейс класса CRaster. Файл Raster.h

```
// Raster.h : interface of CRaster class
// (C) Alexey Polyakov 2002-2003
////////////////////////////////////////////////////////////////////////
#ifndef _RASTER_INCLUDED
#define _RASTER_INCLUDED
// макрос для определения количества байт в выровненной по DWORD строке
пикселов в DIB
// Width - длина строки в пикселах; BPP - бит на пиксел
#define BYTESPERLINE(Width, BPP) ((WORD)((((DWORD)(Width) * \
(DWORD)(BPP) + 31) >> 5)) << 2)

class CRaster
{
   LPBITMAPINFO   m_pBMI;   //указатель на описание изображения
   PBYTE          m_pData;  //указатель на начало растровых данных
public:
   CRaster();
```

```
~CRaster();
void Clear(); //очистка памяти
// Возвращает:
//   указатель на заголовок растра
LPBITMAPINFO GetBMInfoPtr(){return m_pBMI;}
//   указатель на таблицу цветов
RGBQUAD* GetBMColorTabPtr();
//   ширину в пикселах;
LONG GetBMWidth();
//   высоту в пикселах
LONG GetBMHeight();
//   указатель на растровые данные
BYTE* GetBMDataPtr(){return m_pData;};
//   указатель на пиксел
BYTE* GetPixPtr(LONG x, LONG y);

// Загружает из файла
BOOL LoadBMP(CString FileName);
// Выводит DIB на контекст pDC
// x, y - позиция левого верхнего угла области назначения
// cx, cy - размер области назначения
// x0, y0 - позиция левого верхнего угла выводимой части изображения
// cx0, cy0 - размер выводимой части изображения
// str_mode - режим масштабирования
// rop - растровая операция, определяет способ наложения изображения
void DrawBitmap(CDC *pDC, LONG x=0, LONG y=0, LONG cx=0, LONG cy=0,
                LONG x0=0, LONG y0=0, LONG cx0=0, LONG cy0=0,
                int str_mode=COLORONCOLOR, DWORD rop=SRCCOPY);
 // Выводит DIB на контекст pDC с позиции (x,y) в масштабе scale
void DrawBitmap(CDC *pDC, LONG x, LONG y, double scale,
                int str_mode=COLORONCOLOR, DWORD rop=SRCCOPY);

// Записывает BMP в файл
BOOL SaveBMP(CString FileName);
// Создает копию
BOOL CreateCopy(CRaster *pOrg);
// Создает растр заданного размера,
// совместимый с параметрами BITMAPINFO
BOOL CreateCompatible(LPBITMAPINFO pBMI, LONG width=0, LONG height=0);
```

```
    // Возвращает гисторамму изображения
    BOOL GetHistogram(DWORD *pHist, int Range);
};
#endif
```

Листинг 11.2 Реализация методов класса CRaster. Файл Raster.cpp

```
// Raster.cpp : implementation of the CRaster class
// (C) Alexey Polyakov 2002-2003
////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Raster.h"

CRaster::CRaster()
{
   m_pData=NULL;
   m_pBMI=NULL;
}

CRaster::~CRaster()
{
   Clear();
};

void CRaster::Clear()
{

   if(m_pData!=NULL) delete[] m_pData;
   m_pData=NULL;

   if(m_pBMI!=NULL) delete[] m_pBMI;
   m_pBMI=NULL;

};

RGBQUAD* CRaster::GetBMColorTabPtr()
{
   return(LPRGBQUAD)(((BYTE*)(m_pBMI))+sizeof(BITMAPINFOHEADER));
```

```
};

LONG CRaster::GetBMWidth()
{
   if(m_pBMI==NULL) return 0;
   return m_pBMI->bmiHeader.biWidth;
};

LONG CRaster::GetBMHeight()
{
   if(m_pBMI==NULL) return 0;
   return m_pBMI->bmiHeader.biHeight;
};

BYTE* CRaster::GetPixPtr(LONG x, LONG y)
{
   if( x<0 || x>= m_pBMI->bmiHeader.biWidth ||
       y<0 || y>= m_pBMI->bmiHeader.biHeight ||
       m_pData == NULL)
      return NULL;
   return (m_pData+(BYTESPERLINE(m_pBMI->bmiHeader.biWidth,
   m_pBMI->bmiHeader.biBitCount)*y + x*m_pBMI->bmiHeader.biBitCount/8));
};

BOOL CRaster::LoadBMP(CString FileName)
{
   //Очистим
   Clear();
   //Открываем файл
   CFile File;
   if(!File.Open(FileName, CFile::modeRead)) return FALSE;

   ////////////////////////////////////////////////////////
   //Загружаем изображение
   //Читаем заголовок файла. Это дает его размер и положение
   //начала данных
   BITMAPFILEHEADER   FI;
   File.Read(&FI, sizeof(BITMAPFILEHEADER));
```

```
    //Проверяем, Windows Bitmap изображение ?
    if(FI.bfType!=0x4D42)
    { File.Close(); return FALSE;}

    //Смещаем позицию
    File.Seek(sizeof(BITMAPFILEHEADER), CFile::begin);

    //Считаем, что все от заголовка файла до начала растровых данных
    //есть BITMAPINFO
    //Выделяем память под заголовок
    m_pBMI=(LPBITMAPINFO)new BYTE[FI.bfOffBits-sizeof(BITMAPFILEHEADER)];
    if(m_pBMI==NULL) { File.Close(); return FALSE;}
    //Читаем BITMAPINFO
    File.Read(m_pBMI, FI.bfOffBits-sizeof(BITMAPFILEHEADER));

    //Умеем работать только с несжатыми данными
    if(m_pBMI->bmiHeader.biCompression!=0)
       { File.Close(); return FALSE;}

    //Переход к началу данных
    File.Seek(FI.bfOffBits, CFile::begin);
    //Выделяем память под данные
    //расчет размера
    if(m_pBMI->bmiHeader.biSizeImage==0)
       m_pBMI->bmiHeader.biSizeImage=
                BYTESPERLINE(m_pBMI->bmiHeader.biWidth,
                m_pBMI->bmiHeader.biBitCount)*m_pBMI->bmiHeader.biHeight;

    m_pData= new BYTE[m_pBMI->bmiHeader.biSizeImage];
    if(m_pData==NULL) { File.Close(); return FALSE;}
    //Читаем данные
    File.Read(m_pData, m_pBMI->bmiHeader.biSizeImage);

    File.Close();
    return TRUE;
};

void CRaster::DrawBitmap(CDC *pDC,
```

```
                        LONG x/*=0*/, LONG y/*=0*/,
                        LONG cx/*=0*/, LONG cy/*=0*/,
                        LONG x0/*=0*/, LONG y0/*=0*/,
                        LONG cx0/*=0*/, LONG cy0/*=0*/,
                        int str_mode/*=COLORONCOLOR*/,
                        DWORD rop /*=SRCCOPY*/)
{
   if(m_pBMI==NULL || m_pData==NULL) return;
   //размеры не заданы – габариты в пикселах
   if(cx==0) cx=GetBMWidth();
   if(cy==0) cy=GetBMHeight();

   if(cx0==0) cx0=GetBMWidth();
   if(cy0==0) cy0=GetBMHeight();
   HDC  hdc=pDC->GetSafeHdc();
   if(hdc==NULL) return;
   // Установка режима масштабирования
   int oldStretchMode=pDC->SetStretchBltMode(str_mode);
   ::StretchDIBits(hdc,       // дескриптор контекста устройства
                   x, y,      // позиция в области назначения
                   cx, cy,    // размеры обл. назначения
                   x0, y0,    //позиция в исходной области
                   cx0, cy0, //размеры исх. обл.
                   m_pData,   //данные
                   m_pBMI,    //заголовок растра
                   DIB_RGB_COLORS,    //опции
                   rop);      //код растровой операции
if(oldStretchMode!=0)
   pDC->SetStretchBltMode(oldStretchMode);
};

Void CRaster::DrawBitmap(CDC *pDC, LONG x, LONG y, double scale,
                        int str_mode/*=COLORONCOLOR*/,
                        DWORD rop /*=SRCCOPY*/)
{
   if(m_pBMI==NULL || m_pData==NULL) return;

   LONG x0=0, y0=0;
   LONG cx0=GetBMWidth();
```

```
   LONG cy0=GetBMHeight();
   LONG cx=static_cast<LONG>(scale*cx0+0.5);
   LONG cy=static_cast<LONG>(scale*cy0+0.5);

   DrawBitmap(pDC, x, y, cx, cy, x0, y0, cx0, cy0, str_mode, rop);

};

BOOL CRaster::SaveBMP(CString FileName)
{
   //Открываем файл
   CFile File;
   if(!File.Open(FileName, CFile::modeCreate|CFile::modeWrite))
      return FALSE;

   ////////////////////////////////////////////////////////
   //Записываем изображение
   //Вычислим размер заголовка растра вместе с таблицей цветов
   DWORD SizeOfBMI= (DWORD)m_pBMI->bmiHeader.biSize +
   m_pBMI->bmiHeader.biClrUsed*sizeof(RGBQUAD);

   // Заголовок файла
   BITMAPFILEHEADER   FI;
   // Идентификатор типа файла BMP: 0x42 = "B" 0x4d = "M"
   FI.bfType = 0x4d42;
   // Размер всего файла вместе с заголовками и данными
   FI.bfSize = (DWORD) sizeof(BITMAPFILEHEADER) + SizeOfBMI +
               m_pBMI->bmiHeader.biSizeImage;
   FI.bfReserved1 = 0;
   FI.bfReserved2 = 0;
   // Вычисляем смещение до начала растровых данных
   FI.bfOffBits = (DWORD) sizeof(BITMAPFILEHEADER) + SizeOfBMI;
   // Записываем заголовок файла
   File.Write(&FI, sizeof(BITMAPFILEHEADER));

   //Записываем BITMAPINFO вместе с таблицей цветов
   File.Write(m_pBMI, SizeOfBMI);

   //Данные
```

```
   File.Write(m_pData, m_pBMI->bmiHeader.biSizeImage);

   File.Close();
   return TRUE;
};

BOOL CRaster::CreateCopy(CRaster *pOrg)
{
   Clear();
   if(!pOrg) return FALSE;

   LPBITMAPINFO pOrgBMI=pOrg->GetBMInfoPtr();
   //Вычислим размер заголовка растра вместе с таблицей цветов
   DWORD SizeOfBMI= (DWORD)pOrgBMI->bmiHeader.biSize +
                    pOrgBMI->bmiHeader.biClrUsed*sizeof(RGBQUAD);

   // Выделим память под заголовок растра
   m_pBMI=(LPBITMAPINFO)new BYTE[SizeOfBMI];
   if(!m_pBMI) return FALSE;
   // Копируем заголовок растра
   memcpy(m_pBMI, pOrg->GetBMInfoPtr(), SizeOfBMI);

   // Расчет размера памяти под данные
   if(m_pBMI->bmiHeader.biSizeImage==0)
   m_pBMI->bmiHeader.biSizeImage=
   BYTESPERLINE(m_pBMI->bmiHeader.biWidth, m_pBMI->bmiHeader.biBitCount)*
   m_pBMI->bmiHeader.biHeight;
   // Выделяем память под данные
   m_pData= new BYTE[m_pBMI->bmiHeader.biSizeImage];
   if(!m_pData) return FALSE;

   // Копируем данные
   memcpy(m_pData, pOrg->GetBMDataPtr(), m_pBMI->bmiHeader.biSizeImage);

   return TRUE;
};

BOOL CRaster::CreateCompatible(LPBITMAPINFO pBMI, LONG width/*=0*/,
                                                  LONG height/*=0*/)
```

```
{
   if(!pBMI) return FALSE;

   if(width==0)    width=pBMI->bmiHeader.biWidth;
   if(height==0)   height=pBMI->bmiHeader.biHeight;

   // Проверяем, может существующий растр и так совместим
   if( m_pBMI!=NULL &&                          // существует
       m_pBMI->bmiHeader.biWidth==width &&    // такого же размера
       m_pBMI->bmiHeader.biHeight==height && // и глубина цвета совпадает
       m_pBMI->bmiHeader.biBitCount==pBMI->bmiHeader.biBitCount)
      return TRUE; // Растр и так совместим
   ////////////////////////////////////////////////////
   // Создаем совместимый растр
   Clear();
   //Вычислим размер заголовка растра вместе с таблицей цветов
   DWORD SizeOfBMI= (DWORD) pBMI->bmiHeader.biSize +
                  pBMI->bmiHeader.biClrUsed*sizeof(RGBQUAD);

   // Выделим память под заголовок растра
   m_pBMI=(LPBITMAPINFO)new BYTE[SizeOfBMI];
   if(!m_pBMI) return FALSE;

   // Копируем заголовок растра
   memcpy(m_pBMI, pBMI, SizeOfBMI);

   // Устанавливаем размер
   m_pBMI->bmiHeader.biWidth=width;
   m_pBMI->bmiHeader.biHeight=height;

   // Расчет размера памяти под данные
   m_pBMI->bmiHeader.biSizeImage=BYTESPERLINE(m_pBMI->bmiHeader.biWidth,
              m_pBMI->bmiHeader.biBitCount)*m_pBMI->bmiHeader.biHeight;
   // Выделяем память под данные
   m_pData= new BYTE[m_pBMI->bmiHeader.biSizeImage];
   if(!m_pData) return FALSE;

   return TRUE;
};
```

```
BOOL CRaster::GetHistogram(DWORD *pHist, int Range)
{
   // Умеет работать только с данными RGB888
   if(m_pBMI->bmiHeader.biBitCount!=24) return FALSE;

   // Обнулим таблицу
   for(int i=0; i<Range; i++)
      pHist[i]=0;

   LONG DataStrLength=
   BYTESPERLINE(m_pBMI->bmiHeader.biWidth, m_pBMI->bmiHeader.biBitCount);
   BYTE *pCurPix=NULL;
   BYTE Brightness=0;
   for(int y=0, x=0; y<m_pBMI->bmiHeader.biHeight; y++)
      for(x=0; x<m_pBMI->bmiHeader.biWidth; x++)
      {
         // Адрес пиксела
         pCurPix=m_pData+y*DataStrLength+x*3;

         // Яркость рассчитывается как 0,3*Red+0,59*Green+0,11*Blue,
         // но пикселные данные хранятся в файле BMP, в порядке BGR
         Brightness=(BYTE)(( 0.11*(*pCurPix) +
                             0.59*(*(pCurPix+1))+
                             0.3*(*(pCurPix+2)))*Range/256);
         pHist[Brightness]+=1;
      }
   return TRUE;
};
```

Назначение большинства методов класса `CRaster`, мы надеемся, понятно из их названий и комментариев. В реализации также нет каких-то особенностей, которые достойны были бы специального рассмотрения. Этот класс никак не связан со структурой приложения, и вы можете использовать его в любой программе "под Windows". Причем если заменить использование классов `CFile` и `CString` в методах `LoadBMP()` и `SaveBMP()` на реализацию операций с помощью API-функций, то можно обойтись и без MFC. Однако с MFC все же удобнее.

В методе `LoadBMP()` не реализована распаковка сжатых изображений, поэтому класс `CRaster` умеет работать только с данными, так сказать, в их натуральном виде.

Кстати, в случае, когда данные хранятся в несжатом виде, мы могли бы и не выделять динамически память под хранение заголовков и данных изображения. Вместо этого можно использовать механизм, называемый "файлы, проецируемые в память". Об этом механизме кратко упоминается в начале *главы 2*, для освоения же работы с ним можно рекомендовать изучить [9]. В остальном работа с данными осуществлялась бы точно также. Механизм проецирования файлов особенно удобен при работе с файлами большого размера.

Класс `CRaster` содержит два метода `DrawBitmap`, выполняющих вывод изображения на контекст устройства. Аргументы одного из методов позволяют задать положение и размеры выводимой области исходного изображения и определить область назначения. По умолчанию изображение выводится полностью в масштабе 1:1, однако с помощью аргументов этой функции можно и изменить масштаб. Второй метод позволяет просто указать позицию начала вывода и масштаб, в котором должно быть нарисовано изображение. Оба метода внутри используют мощную API-функцию `StretchDIBits()`. Начиная с Windows 98, реализация этой функции умеет выводить растровые данные в форматах JPEG и PNG. Класс CDC, который мы обычно используем для рисования, имеет похожий метод `CDC::StretchBlt()`, но он в качестве исходного изображения просит указать контекст устройства, а не указатель на данные.

Режим масштабирования выбирается `CDC`-методом `SetStretchBltMode()`:

```
int SetStretchBltMode( int nStretchMode );
```

Аргумент функции — `iStretchMode` — режим масштабирования.

Поддерживаются следующие режимы масштабирования:

- `BLACKONWHITE` — выполняет булеву операцию `AND` между цветом существующих и удаленных пикселов (при уменьшении размера изображения). Этот режим используется, если масштабируется рисунок "черным по белому", т. е. алгоритм масштабирования будет стараться сохранить черные пикселы;
- `COLORONCOLOR` — этот режим удаляет (добавляет) строки (столбцы) пикселов без каких-либо попыток сохранить содержащуюся в них информацию. Наиболее быстрый режим. Используется, когда необходимо сохранить цвета изображения неизменными;
- `WHITEONBLACK` — выполняет булеву операцию `OR`. Этот режим используется, если масштабируется рисунок "белым по черному";
- `HALFTONE` — преобразует изображение к заданному размеру и при этом трансформирует цвета так, чтобы средний цвет полученной картинки

приближался к исходному цвету. Наиболее медленный режим. Однако масштабированная картинка выглядит лучше за счет сглаживания "лестничного эффекта". Этот режим не работает в Windows 95/98, и похоже, заменяется режимом COLORONCOLOR.

При масштабировании фотографий и цветных рисунков в большинстве случаев наиболее подходящим является режимы COLORONCOLOR и HALFTONE. Далее мы рассмотрим на практике различия между этими режимами.

Метод GetHistogram() предназначен для получения гистограммы яркости изображения.О том, что это такое и зачем "оно" нужно, мы поговорим дальше.

В проект приложения добавим с помощью команды **Project | Add to project | Files** файлы Raster.h и Raster.cpp.

Поскольку данными (документом) в нашей программе будут изображения, на следующем шаге модифицируем класс документа так, чтобы он умел работать с изображениями.

11.3. Модификация класса документа для обеспечения работы с изображениями

Прежде всего надо решить, что является данными, которыми будет управлять класс документа. Данными в нашем случае будет изображение, а для изображений мы завели класс CRaster, значит, в классе документа надо определить данные как объекты класса CRaster. В принципе, для целей показа картинки на экране хватит и одного объекта CRaster. Однако мы собираемся в дальнейшем наделить программу некоторыми возможностями по редактированию изображений, поэтому нам потребуется не один, а, как минимум, два объекта: один для хранения исходной картинки, второй — буфер для приема преобразованной картинки.

Порядок работы с двумя объектами CRaster в этом случае будет выглядеть следующим образом.

1. Загружаем изображение в первый объект CRaster и показываем его на экране до тех пор, пока пользователь не даст команду выполнить какие-нибудь изменения изображения.
2. Помещаем измененное изображение во второй объект CRaster и начинаем показывать второй объект-картинку.
3. Может случиться так, что пользователю не понравится то, как мы изменили его картинку, тогда он отдает команду "Отменить преобразования". Легко — просто меняем объекты местами. Конечно, если мы хотим побаловать пользователя и предоставить ему возможность долго "капризничать", тогда нам придется завести бóльшее количество копий картинок,

которые отражали бы последовательность произведенных преобразований. Организовать хранение копий можно в виде стека LIFO (last in — first out), где на самом верху будет храниться последняя из копий.

Полностью программный код классов приложения будет приведен *в разд. 11.8*, в тексте же рассмотрим ключевые моменты, необходимые для понимания основных идей.

Итак, заведем в классе нашего документа `CBMDoc` пару объектов `CRaster`, которые и будут хранить изображения:

```
CRaster m_BM[2];      // два буфера для изображений
CRaster *m_pCurBM;    // указатель на активный буфер
```

Указатель `m_pCurBM` будет хранить адрес текущего изображения, его-то мы и будем показывать.

Для загрузки изображения переопределим метод `OnOpenDocument()` класса `CBMDoc` (листинг 11.3). Сделать это можно с помощью ClassWizard. Мы надеемся, вы уже научились пользоваться генератором классов, поэтому не будем на этом останавливаться. Надо отметить, что каркас приложения при запуске программы автоматически создает пустой новый документ. Чтобы этого не происходило, переопределим в классе `CBMApp` метод-обработчик сообщения команды `ID_FILE_NEW` и оставим тело этого метода пустым. Если же вы захотите наделить программу функцией создания нового документа, то придется как-то иначе обрабатывать эту команду.

Листинг 11.3. Метод `OnOpenDocument()` класса `CBMDoc`. Файл BMDoc.cpp

```
BOOL CBMDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
   if (!CDocument::OnOpenDocument(lpszPathName))
      return FALSE;

   // Загружаем в первый буфер
   if(m_BM[0].LoadBMP(lpszPathName))
   {
      m_pCurBM=&m_BM[0];
      //Умеем редактировать только RGB888 (RGB24) данные
      if(m_pCurBM->GetBMInfoPtr()->bmiHeader.biBitCount!=24)
         m_bEditable=FALSE;
      else
         m_bEditable=TRUE;

      return TRUE;
```

```
    }
    return FALSE;

}
```

Как вы можете видеть из листинга 11.3, изображение загружается в первый из объектов `CRaster`. Этот объект становится текущим, его адрес запоминаем в переменной `m_pCurBM`. Далее проверяем формат цвета изображения. Если он не равен RGB888, то ставим флажок `m_bEditable` в значение `FALSE`. Это вовсе не означает, что картинки с отличающимся от RGB888 форматом цвета не будут показываться нашей программой, просто те функции по редактированию изображений, которые мы добавим далее, будут ориентированы на работу с RGB888, а флаг `m_bEditable` будет предостерегать от их неправильного использования. Если у вас появится желание редактировать "не RGB888"-картинки, то вам придется либо переделать функции преобразований, либо (что, кажется, проще) конвертировать картинки в формат RGB888 при загрузке из файла.

11.4. Использование виртуального экрана

В программировании, когда приходится работать с выводом на экран, широко используется концепция "виртуального экрана", на котором происходит подготовка изображения к показу на мониторе. Подготовка может занимать относительно длительное время и включать в себя ряд операций, которые вовсе не обязательно показывать пользователю. Копирование же из виртуального экрана на "реальный" происходит в одно действие, гораздо быстрее, чем построение картинки.

В нашем случае изображение может быть выведено на контекст устройства с помощью метода `CRaster::DrawBitmap()`, внутри этого метода используется API-функция `StretchDIBits()`, которая всем хороша, однако работает сравнительно медленно. Поэтому поэксплуатируем идею виртуального экрана. Создадим в программе такой экран, и будем выводить изображение не очень часто, а только тогда, когда его изменим. Более часто перерисовывать изображение придется объекту-облику, например при обработке сообщений прокрутки, при перерисовке окна после перекрытия другими окнами или при изменении размеров окна, в этом случае будем просто копировать изображение из виртуального экрана на контекст устройства дисплея с помощью метода `CDC::BitBlt()`, который работает относительно быстро.

Виртуальный экран может быть реализован по-разному. Далее мы рассмотрим реализацию, при которой размер экрана равен размеру масштабированной картинки. *Достоинство* такого подхода в том, что на виртуальном экране мы можем разместить полностью готовую для вывода картинку. В случае, если картинка целиком не помещается в окно вывода, достаточно просто

сместиться на позицию прокрутки и отобразить в окне нужную часть изображения.

Недостаток — требуется большое количество памяти под растр виртуального экрана. Допустим, у нас имеется изображение 100 на 100 пикселов, при глубине цвета 32 бита растр виртуального экрана будет занимать 40 Кбайт, а при масштабировании рисунка в 10 раз и создании виртуального экрана такого же размера его растр будет занимать уже 4 Мбайта. Такими темпами могут легко возникнуть проблемы с выводом больших рисунков.

В принципе, можно и не держать большой виртуальный экран. Вполне достаточно экрана размером с максимально возможное окно вывода. В этом случае нам потребуется лишь отображать на этот экран нужную часть изображения.

Достоинство такого подхода в том, что мы экономим память и можем масштабировать изображения, не опасаясь нехватки памяти под виртуальный экран.

Недостаток — нам придется предпринять дополнительные действия по обработке сообщений о прокрутке изображения или изменении размеров окна, и каждый раз при поступлении таких сообщений выполнять вывод на виртуальный экран требуемой части изображения. Это может замедлить прокрутку изображений.

Поскольку за визуализацию изображений в нашей программе ответственен класс облика `CBMView`, возложим на него также и обязанности по обслуживанию виртуального экрана. Изменения, которые надо сделать, рассмотрены в следующем разделе.

11.5. Модификация класса облика

Для реализации виртуального окна заведем в объекте-облике пару переменных:

```
CBitmap    m_VirtScreenBitmap;
CDC        m_VirtScreenDC;
```

В объекте `m_VirtScreenBitmap` будем хранить растр виртуального экрана, а объект `m_virtScreenDC` будет контекстом виртуального экрана.

Контекст виртуального экрана должен быть совместим с контекстом окна, в который будет выполняться вывод. Добавим в класс `CBMView` обработчик сообщения `WM_CREATE`, в котором и создадим совместимый контекст (в листинг 11.4).

Листинг 11.4. Создание контекста для виртуального окна. Файл BMView.cpp

```
int CBMView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CScrollView::OnCreate(lpCreateStruct) == -1)
```

```
    return -1;

  // TODO: Add your specialized creation code here

  // Создадим совместимый контекст для виртуального экрана
  CPaintDC dc(this);
  m_VirtScreenDC.CreateCompatibleDC(&dc);

  return 0;
}
```

Теперь нам остается только организовать вывод изображения на виртуальный экран. Для решения этой задачи нам потребуется сделать следующее:

1. Добавить метод, в котором на виртуальный экран будет водиться изображение, содержащееся в объекте-документе. Назовем его `UpdateVirtualScreen()`. В этом же методе будем создавать растр для виртуального экрана, достаточный по размерам для вывода всего изображения.
2. В метод `CBMView::OnDraw()` добавим копирование виртуального экрана в клиентскую часть окна вывода. Копирование будем выполнять с учетом позиции прокрутки.
3. Переопределить (с помощью ClassWizard) виртуальный метод `OnUpdate()` так, чтобы в нем устанавливались размеры области прокрутки, соответствующие размеру изображения с учетом коэффициента масштабирования. Это требуется для того, чтобы пользователь мог, "прокручивая" изображение в окне, полюбоваться на любую его часть. Добавленный же AppWizard-ом (при создании каркаса приложения) в класс метод `OnInitialUpdate()` можно удалить (но тогда не забыть убрать и объявление этого метода в интерфейсе класса) или просто закомментировать в нем действия по установке размеров области прокрутки. Это можно сделать, так как реализация `OnInitialUpdate()` в базовом классе вызывает метод `OnUpdate()`, который мы уже модифицировали должным образом.
4. Определить (с помощью ClassWizard) метод-обработчик сообщения WM_ERASEBKGND, для того чтобы самим контролировать перерисовку (очистку) фона окна облика. Это позволит избежать мерцания при перерисовке окна облика.

Текст этих методов приведен в листинге 11.5.

Листинг 11.5. Модифицированные методы облика. Файл BMview.cpp

```
// Цвет для заливки фона
#define GRAY RGB(127, 127, 127)
BOOL CBMView::UpdateVirtualScreen()
{
  CBMDoc* pDoc = GetDocument();
```

```
ASSERT_VALID(pDoc);

// Получили указатель на активную картинку
CRaster* pCurBM=pDoc->GetCurrentBMPtr();
if(pCurBM==NULL) return FALSE;

// Вычисляем размеры картинки с учетом масштаба
LONG imgw=static_cast<LONG>(pCurBM->GetBMWidth()*m_dScale);
LONG imgh=static_cast<LONG>(pCurBM->GetBMHeight()*m_dScale);

// Если битмап уже существует, возьмем ее размер
BITMAP BMStruct; BMStruct.bmWidth=BMStruct.bmHeight=0;
if(m_VirtScreenBitmap.GetSafeHandle( ))
   m_VirtScreenBitmap.GetBitmap(&BMStruct);

// Если размеры виртуального экрана меньше размеров картинки,
// увеличим экран
if(BMStruct.bmWidth<imgw || BMStruct.bmHeight<imgh)
{
   CPaintDC dc(this);
   // Размеры дисплея в пикселах
   int scrw=dc.GetDeviceCaps(HORZRES);
   int scrh=dc.GetDeviceCaps(VERTRES);
   // Выберем временную битмап в контексте
   // это освободит m_VirtScreenBitmap
   // (если она была ранее выбрана в контексте)
   // и даст возможность удалить ее
   CBitmap TempBM; TempBM.CreateCompatibleBitmap(&dc,1,1);
   m_VirtScreenDC.SelectObject(&TempBM);

   // Разрушим ранее существовавшую битмап
   m_VirtScreenBitmap.DeleteObject();

   // и на ее месте построим новую по размерам изображения,
   // не меньше размеров дисплея
   if(!m_VirtScreenBitmap.CreateCompatibleBitmap(&dc,
                  (imgw<scrw?scrw:imgw), (imgh<scrh?scrh:imgh)))
      return FALSE;
      // Новую битмап выберем в контексте виртуального экрана
```

```
        m_VirtScreenDC.SelectObject(&m_VirtScreenBitmap);
    }
    // Очистим виртуальный экран
    CBrush FonBrush(GRAY); // кисть для заливки фона
    m_VirtScreenBitmap.GetBitmap(&BMStruct); // узнаем размеры экрана
    m_VirtScreenDC.FillRect(&CRect(0,0,
                    BMStruct.bmWidth, BMStruct.bmHeight), &FonBrush);

    // Выведем на виртуальный экран картинку
    pCurBM->DrawBitmap(&m_VirtScreenDC, 0, 0, m_dScale, m_nStretchMode);
    // Обновим изображение на экране
    Invalidate();
    return TRUE;
};

void CBMView::OnDraw(CDC* pDC)
{
    // TODO: Add your specialized code here and/or call the base class
    CBMDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // Получим размер клиентской части окна
    CRect ClientRect;
    GetClientRect(&ClientRect);

    // Копируем содержимое виртуального экрана
    // с учетом позиции прокрутки
    CPoint ScrollPos=GetScrollPosition();
    pDC->BitBlt(ScrollPos.x, ScrollPos.y,
                ClientRect.Width(), ClientRect.Height(),
                &m_VirtScreenDC, ScrollPos.x, ScrollPos.y, SRCCOPY);
}
void CBMView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    CBMDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // Обновим изображение на виртуальном экране
```

```
    if(UpdateVirtualScreen())
    {
       // Размер области прокрутки
       CSize ScrollSize;

       // Область прокрутки - весь размер картинки с учетом масштаба
       ScrollSize=pDoc->GetCurrentBMSize();
       ScrollSize.cx=static_cast<LONG>(ScrollSize.cx*m_dScale);
       ScrollSize.cy=static_cast<LONG>(ScrollSize.cy*m_dScale);
       SetScrollSizes(MM_TEXT, ScrollSize);
    }
    else
       AfxMessageBox("Ошибка при выводе на виртуальный экран");

    // Вызываем метод базового класса
    CScrollView::OnUpdate(pSender, lHint, pHint);
}
```

Рассмотрим, как же все это работает. Когда метод `OnOpenDocument()` объекта-документа возвращает `TRUE`, каркас приложения создает объект-облик и посылает ему уведомление о необходимости отобразить содержимое документа. Это уведомление в нашем случае обрабатывается методом `CBMView::OnUpdate()`, который, в свою очередь, вызывает `CBMView::UpdateVirtualScreen()`. Метод `UpdateVirtualScreen()` создает растр нужного размера, присоединяет его дескриптору виртуального экрана, запрашивает у объекта-документа текущее изображение и выводит его на виртуальный экран. Вывод осуществляется с использованием переменных `m_dScale`, `m_nStretchMode`, которые задают масштаб и режим масштабирования. Эти переменные мы добавим в класс `CBMView` как раз для того, чтобы поэкспериментировать с маштабированием. В интерфейсе класса данные переменные описаны следующим образом:

```
    double   m_dScale;
    int      m_nStretchMode;
```

В конструкторе же эти переменные инициализируются начальными значениями:

```
CBMView::CBMView()
{
    m_dScale=1.0;
    m_nStretchMode=HALFTONE;
}
```

После того как картинка была выведена на виртуальный экран, вызывается метод `Invalidate()`, который сообщает облику, что ему следует обновить изображение на экране. Облик, получив это сообщение, вызывает свой метод `OnDraw()`, в котором изображение с виртуального экрана копируется на "реальный" экран.

Все. Уже, кажется, можно загрузить картинки и посмотреть, как они выглядят. На рис. 11.1 показана программа с загруженными четырьмя рисунками (ведь это у нас многодокументное приложение).

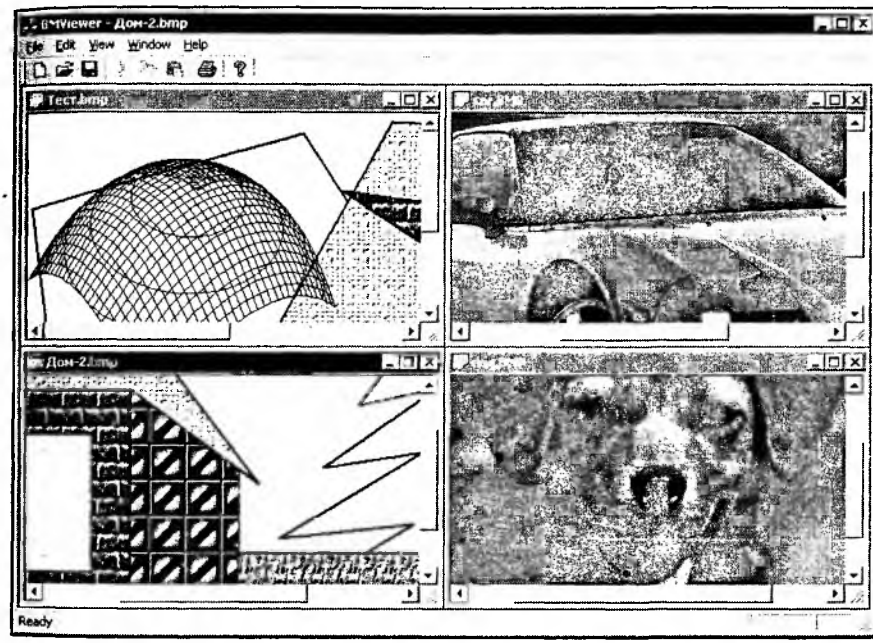


Рис. 11.1. Просмотр рисунков в программе BMViewer

Более того, в программах с MDI-интерфейсом у одного документа может быть несколько объектов-обликов, как на рис. 11.2. Создать еще один облик для документа можно командой **Window | New window**. Изменение данных объекта-документа будет отражаться во всех окнах (всеми объектами-обликами). Такую возможность можно использовать, например, для показа одного и того же изображения в разных окнах и в разных масштабах, далее мы посмотрим, как это выглядит на практике. Для изучения особенностей SDI- и MDI-приложений можно также порекомендовать книгу [4].

Чтобы можно было изменять масштаб вывода изображения, добавим в меню **View** программы команды **Zoom In** и **Zoom Out**, а для установки режима масштабирования — команды **Stretch HALFTONE** и **Stretch COLORONCOLOR** (рис. 11.3). Можно также назначить этим командам горячие клавиши *(см. разд. 6.7)*.

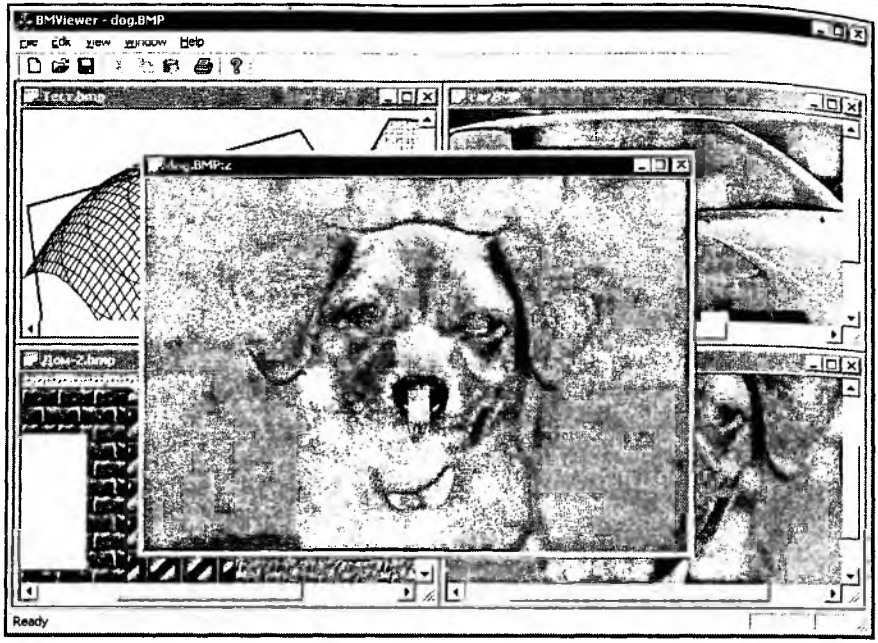


Рис. 11.2. Четвертая картинка показана сразу в двух окнах

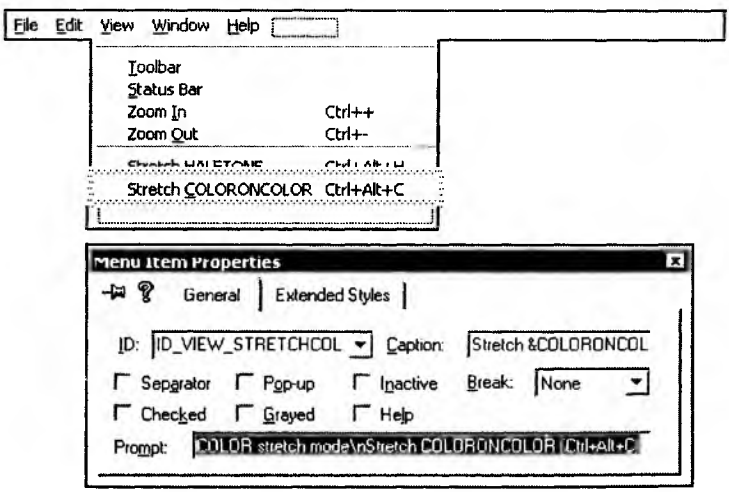


Рис. 11.3. Добавление команд масштабирования в меню программы

Обработчики этих команд добавим с помощью ClassWizard в класс облика (листинг 11.6). Эти функции изменяют состояние переменных `m_dScale` и `m_nStretchMode`. Для команд установки режимов масштабирования также добавлены методы `OnUpdateViewStretchhalftone()` и

OnUpdateViewStretchcoloroncolor() для обработки сообщения UPDATE_COMMAND_UI. В этих функциях можно управлять состоянием соответствующих команд в интерфейсе программы (например, можно делать недоступными команды в зависимости от состояния программы). В данном случае мы просто маркируем соответствующий режим масштабирования.

Листинг 11.6. Обработка команд масштабирования. Файл BMview.cpp

```
void CBMView::OnViewZoomin()
{
   // TODO: Add your command handler code here
   m_dScale*=2;
   OnUpdate(NULL, 0, NULL);
}

void CBMView::OnViewZoomout()
{
   // TODO: Add your command handler code here
   m_dScale/=2;
   OnUpdate(NULL, 0, NULL);
}

void CBMView::OnViewStretchhalftone()
{
   // TODO: Add your command handler code here
   m_nStretchMode=HALFTONE;
   OnUpdate(NULL, 0, NULL);
}

void CBMView::OnUpdateViewStretchhalftone(CCmdUI* pCmdUI)
{
   // TODO: Add your command update UI handler code here
   pCmdUI->SetCheck(m_nStretchMode==HALFTONE);
}

void CBMView::OnViewStretchcoloroncolor()
{
   // TODO: Add your command handler code here
   m_nStretchMode=COLORONCOLOR;
```

```
    OnUpdate(NULL, 0, NULL);
}

void CBMView::OnUpdateViewStretchcoloroncolor(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->SetCheck(m_nStretchMode==COLORONCOLOR);
}
```

В приложениях с MDI-интерфейсом возможно также использовать механизм, называемый Splitted window, который позволяет показать несколько обликов в одном окне-рамке. Для этого потребуется лишь научить рамку дочернего окна нашего приложения работать с несколькими обликами. Сделать это довольно просто, надо всего-навсего на этапе **4** создания каркаса приложения генератора **AppWizard** зайти по кнопке **Advanced** в диалог **Advanced Options** (расширенные установки) и на закладке **Window Style** поставить галочку напротив **Use split window**. Но если мы это забыли сделать — тоже не беда, — зайдем в интерфейс класса `CChildFrame` и руками добавим переменную-объект класса `CSplitterWnd`:

```
// Attributes
protected:
    CSplitterWnd m_wndSplitter;
```

Затем с помощью ClassWizard переопределим в классе `CChildFrame` виртуальный метод `OnCreateClient` и немного подправим его код, как показано в листинге 11.7.

Листинг 11.7. Подправленный метод CChildFrame::OnCreateClient. Файл ChildFrm.cpp

```
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext*
pContext)
{
    // TODO: Add your specialized code here and/or call the base class

//    return CMDIChildWnd::OnCreateClient(lpcs, pContext);
    return m_wndSplitter.Create( this,
        2, 2,                 // Максимальное количество строк и столбцов
        CSize( 10, 10 ),      // Минимальный размер окна
        pContext );
}
```

В результате — о чудо! В одном окне-рамке мы можем увидеть четыре различных окна-облика с нашей картинкой в разных масштабах и разных режимах масштабирования (рис. 11.4). Для того чтобы получить такой эффект, надо ухватиться мышью за ползунок в левой (верхней) части нижней (правой) полосы прокрутки и перетащить его в нужную позицию.

Хотя метод CSplitterWnd::Create() имеет параметры для задания максимального количества строк и столбцов, в документации сказано, что эти значения почему-то не должны превышать 2. Хотя если попробовать увеличить это значение "на свой страх и риск", то можно разбить рамку и на 9 различных обликов.

Активным считается тот облик, в котором последний раз щелкнули мышью. К нему и применяются команды масштабирования. В верхнем ряду (рис. 11.4) показаны увеличенные фрагменты изображения: слева в режиме HALFTONE, справа — COLORONCOLOR. Видно, что режим COLORONCOLOR не пытается бороться с "лестничным эффектом".

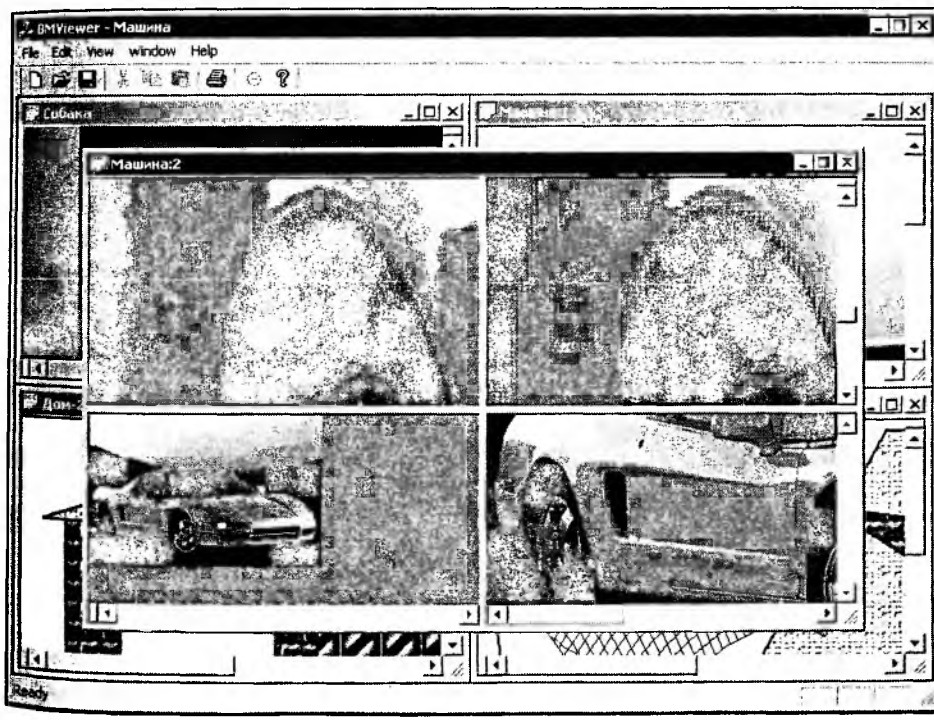


Рис. 11.4. В одном окне-рамке отображаются четыре различных облика одного объекта-документа

11.6. Редактирование изображений

Ну вот, наконец-то мы добрались до самого интересного. Магия преобразований — вот, на мой взгляд, основная радость, которую дает цифровая обработка изображений. Используя возможности редактирования в таких программах как Adobe Photoshop или Ulead Photoimpact, человек даже с заурядными художественными способностями может превратить самую скучную фотографию в нечто более привлекательное. Конечно, за всем этим стоит прочная математическая база и кропотливый труд программистов. Далее мы рассмотрим два вида преобразований:

- *точечные* — новое значение элемента изображения (пиксела) рассчитывается только на основе его старого значения;
- *пространственные* (матричные) — при расчете нового значения пиксела учитывается не только его старое значение, но также значения некоторой области пикселов вокруг него.

Точечные преобразования удобно выполнять с помощью таблиц преобразования, которые рассмотрены *в разд. 11.6.3*.

Обычно пространственное преобразование заключается в нахождении свертки значений группы пикселов. Свертка вычисляется как сумма пиксельных значений, попавших в зону преобразования, помноженных на весовые коэффициенты. В качестве весовых коэффициентов выступают элементы матрицы преобразования. Значения элементов матрицы преобразования и определяют тип преобразования. Размер матрицы преобразования соответствует области пикселов, которые будут участвовать в преобразовании. Центральный элемент матрицы — весовой коэффициент преобразуемого пиксела (x, y). Поэтому матрицы преобразования, обычно имеют нечетный размер (например, 3×3 или 5×5 элементов). Часто свертки заключают в себе сложный и глубокий математический смысл, который, к счастью, имеет простую и понятную практическую интерпретацию. Рассмотрим, например, свертку с помощью следующей матрицы M:

$$M = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix}.$$

Новое значение пиксела $P(x, y)$ может быть рассчитано с использованием следующего псевдокода:

```
MX=3; // размер матрицы преобразования по x
MY=3; // размер матрицы преобразования по y
CountCoeffSumm=0; // счетчик суммы коэффициентов матрицы преобразования
NewP=0; // новое значение пиксела
for(j=-MY/2; j<= MY/2; j++)
```

```
for(i=-MX/2; i<=MX/2; i++)
   {
      NewP = NewP + P(x+i, y+j)*M(i, j);
      CountCoeffSumm = CountCoeffSumm + M(i, j);
   }
P(x, y)=NewP/CountCoeffSumm;
```

Здесь предполагается целочисленное деление, т. е. результат MX/2 — значение 1.

В процессе преобразования выполняется подсчет коэффициентов матрицы преобразования, а после расчета нового значения оно делится на сумму коэффициентов. Это необходимо для того, чтобы привести результат к диапазону значений суммированных пикселов.

Описанная выше свертка с помощью единичной матрицы соответствует преобразованию "размытие" (понижение четкости) изображения. Достигается такой эффект за счет усреднения значений группы пикселов, охваченной матрицей преобразования. Если значение пиксела (*x*, *y*) было выше среднего, оно уменьшится, если было ниже среднего, то увеличится. Однако это не означает, что все изображение станет монотонным, так как матрица преобразования движется по изображению вместе с координатами (*x*, *y*), средний уровень тоже изменяется. Далее будет рассмотрено применение различных матриц преобразования.

В отдельные категории преобразований выделяют *покадровые* и *геометрические* процессы. Покадровые преобразования выполняются над парой или бльшим количеством изображений. Примером покадрового преобразования является вычитание, когда находится разница между двумя картинками. Вычитание может использоваться для определения сходства (нахождения отличий) изображений. Покадровые преобразования широко применяются при обработке и сжатии потоков видеоданных.

Само название процесса "геометрический" говорит о том, что суть преобразования заключается в изменении положения или других геометрических характеристик изображения. Примерами геометрических преобразований являются поворот, сдвиг, интерполяция (масштабирование) изображения.

Все типы преобразований рассмотрены в [5]. Достоинством книги является подробное рассмотрение "классических" преобразований растровых изображений, включая рассмотрение теоретических основ, и программной реализации. Недостатком, на мой взгляд, — не очень удачный перевод и использование устаревших средств и стиля программирования. Однако последнее — это не упрек автору, а лишь свидетельство того, что книга была написана на заре "революции" в области компьютерной графики.

О "внутреннем устройстве" и практическом применении цифровых фильтров можно прочитать в [15].

Теоретическим основам обработки изображений посвящена книга [1].

Далее мы рассмотрим суть некоторых ставших классическими процедур преобразования растровых изображений — наделим программу BMViewer возможностями редактирования картинок. В программе будет реализовано несколько преобразований, но что не менее важно, мы постараемся организовать в программе структуру, которая позволила бы легко включать в нее реализацию любых новых преобразований.

11.6.1. Гистограмма яркости изображения

Что такое гистограмма? Это такой график из столбиков. А что такое "гистограмма яркости изображения"? *Гистограммой яркости изображения* принято называть график, который показывает относительную частоту появления точек (пикселов) различных степеней яркости в изображении. Например, есть у нас изображение из 16 пикселов. Пусть 8 пикселов имеют яркость 1, 2 пиксела — яркость 4, оставшиеся 6 пикселов — яркость 7. На десятибалльной шкале яркости график такого изображения может выглядеть так, как показано на рис. 11.5.

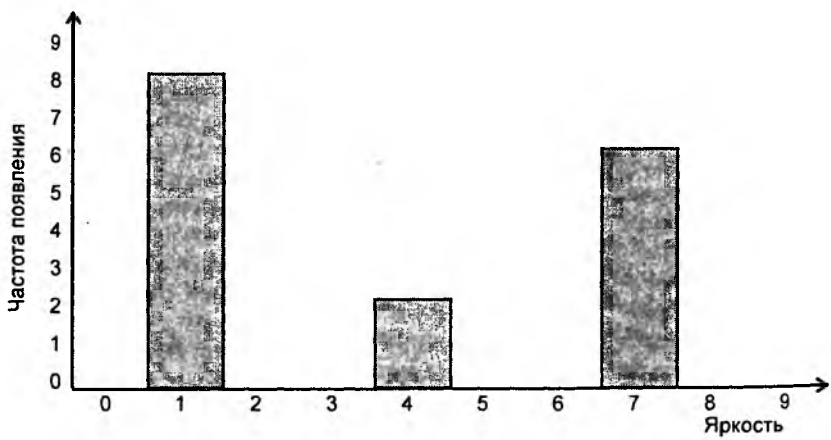


Рис. 11.5. Гистограмма яркости мнимого изображения из шестнадцати пикселов

В реальных изображениях пикселов обычно гораздо больше, а шкала яркости включает значения от 0 до 255.

Яркость RGB-пиксела рассчитывается по следующей формуле:

$$\text{Brightness} = 0{,}3 \times \text{Red} + 0{,}59 \times \text{Green} + 0{,}11 \times \text{Blue}.$$

Как видно из формулы, разные цвета имеют разные весовые коэффициенты. Это связано с различной восприимчивостью человеческого глаза к разным составляющим цвета.

Гистограмма яркости широко используется для анализа и редактирования изображений. Наш класс `CRaster` уже умеет рассчитывать гистограмму изображения. Эта возможность реализована в методе `CRaster::GetHistogram()`. Метод `GetHistogram()` получает два параметра (см. листинг 11.2):

- `DWORD *pHist` — указатель на массив, в который будут помещены значения гистограммы;
- `int Range` — размер массива, он же диапазон яркостей.

Порядковый номер значения в массиве `pHist` соответствует яркости, а значение элемента — частоте появления этой яркости в картинке. Рассчитывается гистограмма довольно просто: в цикле для всех пикселов изображения сначала рассчитывается яркость пиксела, а затем увеличивается на единицу значение соответствующего элемента в массиве `pHist`.

Поскольку расчет гистограммы уже реализован, нам остается только нарисовать ее на экране. Для показа гистограммы на экране добавим в программу специальное диалоговое окно (рис. 11.6). Напомним, что новый шаблон диалогового окна вставляется командой **Insert | Resource | Dialog | New**. Присвоим шаблону идентификатор `IDD_HIST` и добавим в него рамочку (элемент *picture* тип *frame*) — этот элемент понадобится нам для рисования гистограммы, поэтому дадим ему идентификатор `IDC_HIST_VIEW`. В шаблон также добавлено два ползунка (элементы *slider*), и два элемента *static*, в которых будут показываться значения в виде текста.

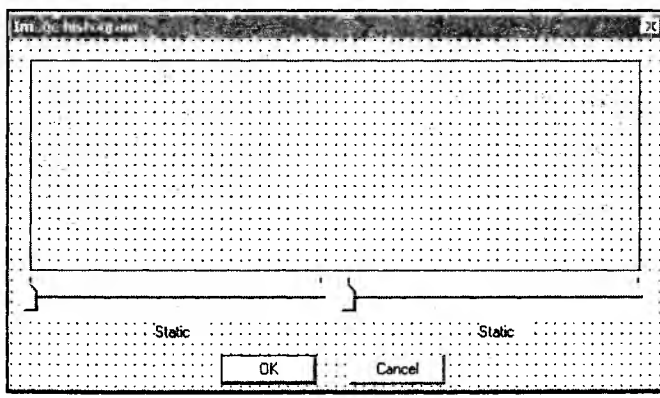


Рис. 11.6. Шаблон окна диалога **Image histogram**

Изображение гистограммы, в принципе, можно нарисовать прямо по окну диалога, определив в классе диалога обработчик сообщения `WM_PAINT` и выполнив в нем соответствующие построения. Однако бóльшую гибкость можно получить, если создать специализированный класс, который бы выводил гистограмму в заданное Windows-окно. Для создания такого класса воспользуемся все тем же ClassWizard-ом.

В окне ClassWizard-а нажмем **AddClass | New** и в появившемся окне **New Class** укажем имя нашего нового класса (рис. 11.7), назовем его `CHistView`, а в качестве базового класса укажем класс `CStatic` (из библиотеки MFC) — он обеспечивает функциональность статических (static) Windows-элементов управления: "рамка", "битмап", "пиктограмма"). Мы переопределим в классе `CHistView` обработчик сообщения `WM_PAINT` и научим его рисовать гистограмму. Затем мы сможем связать с элементом управления `IDC_HIST_VIEW` объект класса `CHistView`. Это позволит свободно размещать изображение гистограммы в нужном месте диалогового окна, не заботясь о том, чтобы у нас в коде были прописаны соответствующие координаты. Можно также разместить несколько элементов-гистограмм (например, для разных цветовых каналов) и при этом не придется каким-то образом модифицировать программный код вывода гистограммы.

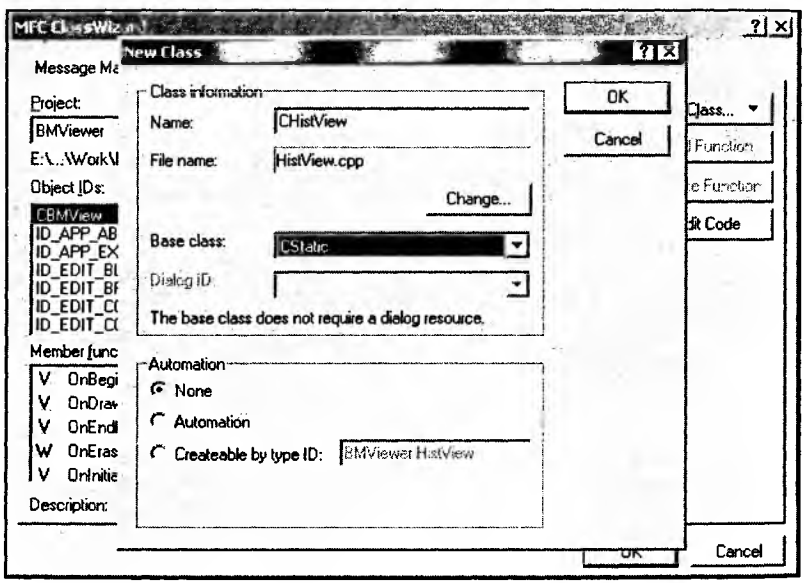


Рис. 11.7. Создание класса `CHistView`

Ползунки-слайдеры, добавленные в диалог, потребуются нам в дальнейшем для коррекции гистограммы. Об этих элементах сказано здесь, чтобы дважды не описывать этот диалог. С помощью ClassWizard создадим класс `CHistDlg`, который будет обслуживать наш диалог (просто нажмите комбинацию клавиш <Ctrl>+<W>, находясь в редакторе шаблона диалогового окна). Свяжем созданные элементы управления с объектами в классе диалога (рис. 11.8). Причем с элементом `IDC_HIST_VIEW` мы связываем объект созданного класса `CHistView`.

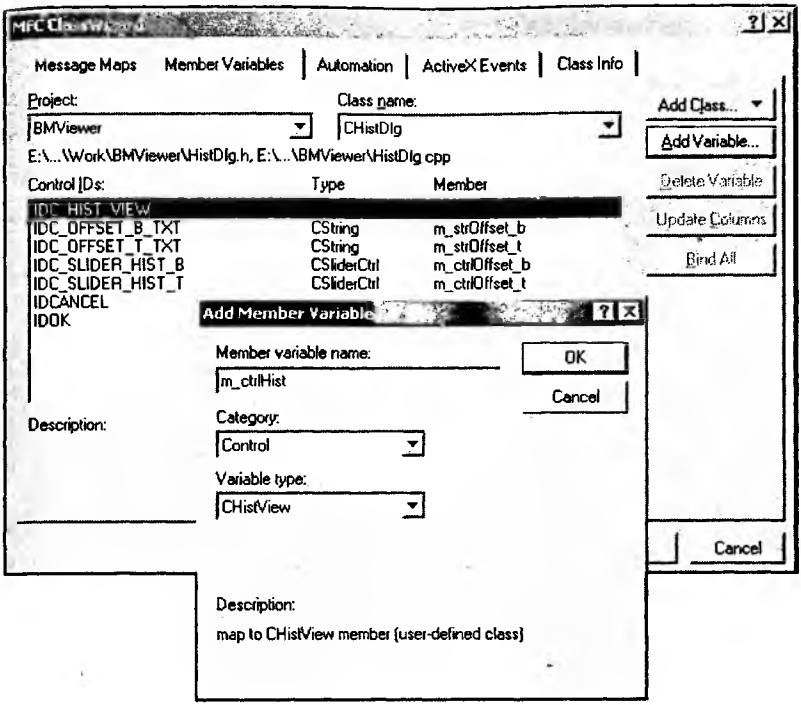


Рис. 11.8. Переменные класса `CHistDlg`

Для рисования гистограммы добавим в класс `CHistView` (опять же с помощью ClassWizard-а) обработчик сообщения `WM_PAINT` (листинг 11.8).

Листинг 11.8. Метод `CHistView::OnPaint()`. Файл HistView.cpp

```
void CHistView::OnPaint()
{
   CPaintDC dc(this); // device context for painting

   // TODO: Add your message handler code here

   if(m_pHist==NULL || m_iRange==0 ) return;

   // Найдем среднее значение
   DWORD MaxBright=0, SumBright=0;
   for(int i=0; i<m_iRange; i++)
      SumBright+=m_pHist[i];
```

```
    // Пусть максимальное (показываемое на рисунке) значение
    // будет в три раза больше среднего
    MaxBright=3*SumBright/m_iRange;
    if(MaxBright==0) return;

    // Перо для рисования гистограммы
    CPen HistPen(PS_SOLID, 2, m_Color);
    CPen *pOldPen=dc.SelectObject(&HistPen);
    CGdiObject *pOldBrush=dc.SelectStockObject(NULL_BRUSH);

    // Найдем координаты окна вывода
    CRect FrameRect;
    GetWindowRect(&FrameRect);
    ScreenToClient(&FrameRect);

...// Нарисуем гистограмму в окне
    dc.Rectangle(&FrameRect);
    FrameRect.bottom-=1;
    double kx=((double)FrameRect.Width())/m_iRange;
    double ky=((double)FrameRect.Height())/MaxBright;

    int x=0, y=0;
    for(i=0; i<m_iRange; i++)
{
       x=FrameRect.left+(kx*i);
       y=FrameRect.bottom;
       dc.MoveTo(x, y);
       y=FrameRect.bottom -(ky*m_pHist[i]);
       if(y<FrameRect.top) y=FrameRect.top;
       dc.LineTo(x, y);
    }
    if(pOldPen)
       dc.SelectObject(pOldPen);
    if(pOldBrush)
       dc.SelectObject(pOldBrush);

    // Do not call CStatic::OnPaint() for painting messages
}
```

Каждый раз, когда окно (в данном случае — элемент управления, связанный с объектом класса CHistView) должно быть показано на экране, Windows посылает ему сообщение WM_PAINT. Обрабатывая это сообщение, мы рисуем гистограмму.

Как вы наверняка заметили, гистограмма рисуется на основе значений, хранящихся в массиве, на который указывает переменная m_pHist, переменная же m_iRange задает размер массива. Эти переменные мы добавили в интерфейс класса CHistView (листинг 11.9). Для установки этих переменных добавлен метод SetData().

Листинг 11.9. Интерфейс класса CHistView. Файл HistView.h

```
class CHistView : public CStatic
{
// Construction
public:
   CHistView();

// Attributes
public:
   int            m_iRange;       //размер массива гистограммы
   const DWORD    *m_pHist;       //указатель на данные гистограммы
   COLORREF        m_Color;       //цвет, которым рисовать гистограмму

// Operations
public:
   // Устанавливает данные для отображения
   void SetData(const DWORD *pHist,  int Range)
               {m_pHist=pHist; m_iRange=Range;};
   // Устанавливает цвет рисования гистограммы
   void SetColor(const COLORREF &c) {m_Color=c;};

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CHistView)
   //}}AFX_VIRTUAL

// Implementation
public:
```

```
    virtual ~CHistView();

    // Generated message map functions
protected:
    //{{AFX_MSG(CHistView)
    afx_msg void OnPaint();
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};
```

Для того чтобы диалог заработал, надо где-то его вызвать. Поэтому добавим в меню программы **Edit** команду **Histogram**, а в класс документа — метод-обработчик этой команды (листинг 11.10).

Листинг 11.10. Метод-обработчик команды вызова диалога с гистограммой. Файл BMDoc.cpp

```
void CBMDoc::OnEditHistogram()
{
    const int Range=256;
    DWORD Hist[Range]; // гистограмма из Range градаций яркости

    // Запросим гистограмму у текущего изображения
    if(m_pCurBM==NULL || !m_pCurBM->GetHistogram(Hist, Range))
        return;
    // Создаем объект-диалог
    CHistDlg HDlg;
    // Передадим гистограмму в диалог
    HDlg.SetData(Hist, Range);
    // Покажем гистограмму
    if(HDlg.DoModal()==IDCANCEL) return;
    // Требуется выполнить коррекцию контрастности
    if(HDlg.m_iOffset_b !=0 || HDlg.m_iOffset_t!=NULL)
    {
        // Настраиваем фильтр гистограммы
        m_HistogramFilter.Init(HDlg.m_iOffset_b, HDlg.m_iOffset_t);
        // Делаем фильтр активным
        m_pCurFilter=&m_HistogramFilter;
```

```
        // Выполняем преобразование
        Transform();
    }
}
```

В листинге 11.10 все до строчки

```
// Требуется выполнить коррекцию контрастности
```

должно быть вам понятно. О том, что же это за "коррекция яркости", поговорим дальше.

Единственный момент — это передача данных гистограммы в диалог. Данные передаются в объект-диалог с помощью метода `SetData(Hist, Range)`, который передает данные уже непосредственно элементу, отображающему гистограмму. Этот метод добавлен в интерфейс класса `CHistDlg`. Выглядит он следующим образом:

```
void SetData(const DWORD *pHist, int Range)
            {m_ctrlHist.SetData(pHist, Range);};
```

где `m_ctrlHist` объект класса `CHistView`.

Полностью интерфейс класса приведен *в разд. 11.8*, листинг 11.40, реализация — листинг 11.41.

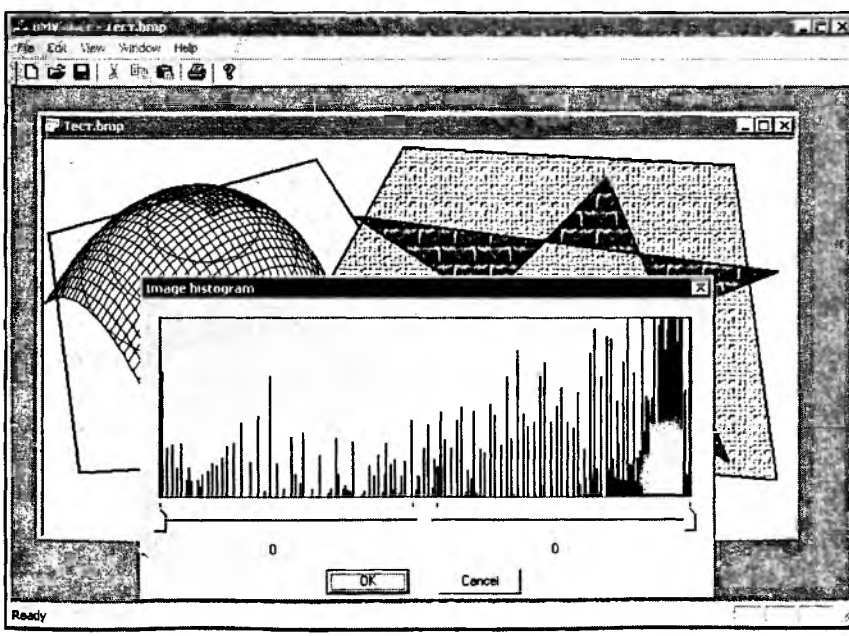


Рис. 11.9. Гистограмма яркости тестового рисунка, экспортированного из программы Painter 4.2

Теперь можно посмотреть, какие же гистограммы яркости у наших изображений. Например, на рис. 11.9 показана гистограмма тестового рисунка, который мы экспортировали в формат BMP из программы Painter 4.2 в *главе 10*.

Гистограмма (рис. 11.9) показывает, что яркость в этом рисунке распределена неравномерно (смещена в область светлых тонов), а многие значения яркости и вовсе отсутствуют. Оно и понятно, ведь это искусственный рисунок, и мы использовали при его создании далеко не все цвета и оттенки. Фотографические изображения обычно имеют более плавные гистограммы с широким спектром яркости, как, например, на рис. 11.10.

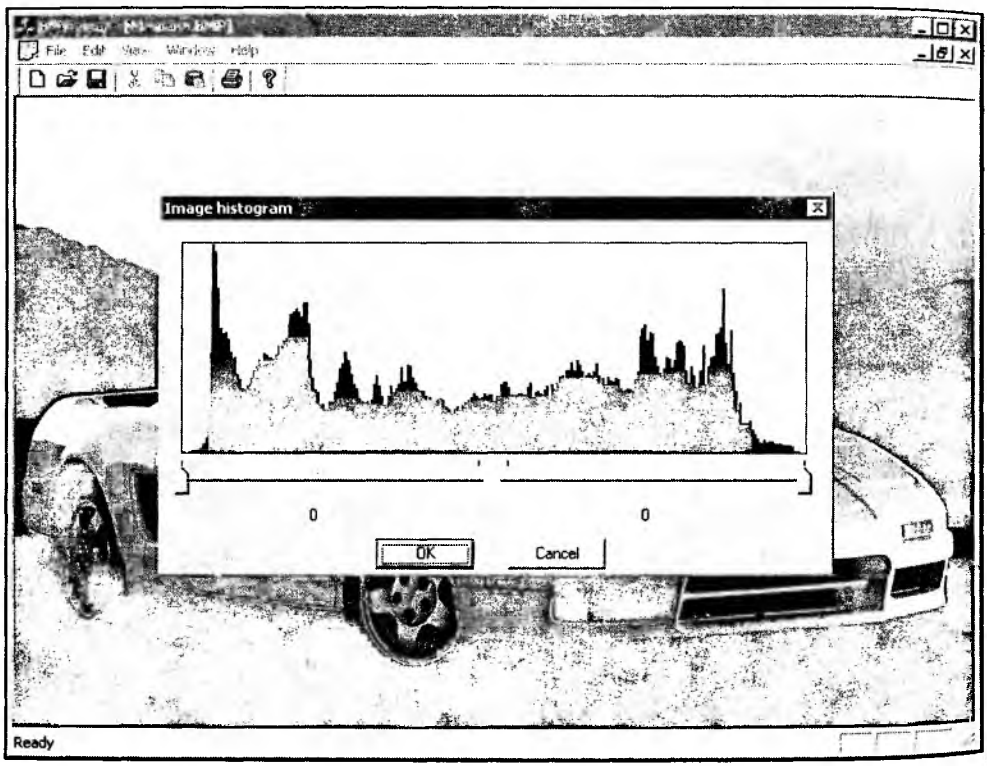


Рис. 11.10. Гистограмма яркости фотографии

Какие же выводы мы можем сделать, взглянув на рис. 11.10? Несмотря на то что диапазон яркости довольно широкий, все же он занимает не всю шкалу. Следовательно, мы можем попытаться улучшить внешний вид этой фотографии. Далее мы рассмотрим, как гистограмма может быть использована для повышения контрастности изображения, но сначала придется обсудить внутреннее устройство программной реализации.

11.6.2. Программная схема выполнения преобразований. Графические фильтры

Поскольку мы собираемся реализовать целый ряд процедур преобразования изображений, следует хорошо обдумать, как они будут уживаться между собой и взаимодействовать с остальными модулями программы. Судя по интерфейсу многих графических редакторов и организации программ обработки видеоданных, в мультимедийном программировании широко распространена концепция фильтров. Что такое фильтр? Это некоторая программа, которая, пропуская через себя данные, преобразует их некоторым образом. В нашем случае данными являются значения цветов пикселов изображения. Такой подход выглядит очень удачным, так как он позволяет создавать четко структурированные модульные программы. Используем и мы эту идею. Представим, что у нас имеется набор фильтров, пропуская через которые данные изображения мы можем добиваться различных эффектов (рис. 11.11).

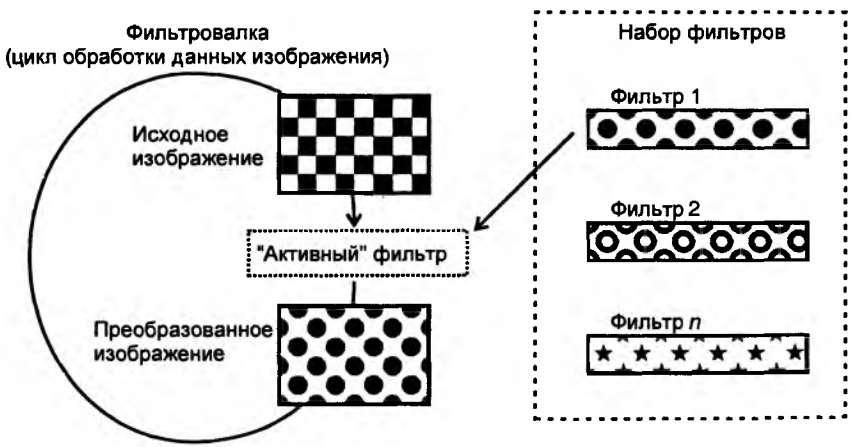


Рис. 11.11. Схема использования фильтров для преобразования изображений

Для того чтобы получить нужный эффект, достаточно просто указать программе, какой фильтр считать активным. В программе же где-то должна существовать "фильтровалка" — процедура, в которой будет выполняться само пропускание данных через фильтр.

Фильтры можно реализовать в виде классов, производных от какого-то одного базового класса. В базовом классе следует определить набор методов, общих для всех фильтров. В программе заведем переменную — указатель на активный фильтр. Используя этот указатель, "фильтровалка" будет обращаться к нужному фильтру.

Саму фильтрацию изображения можно выполнять по-разному. Например, можно было бы передать фильтру всю исходную картинку и ожидать от него уже полностью преобразованного изображения. А можно пропускать через фильтр исходное изображение по одному пикселу. В последнем случае не придется дублировать цикл обработки всего изображения в каждом фильтре, и вызывающая фильтр процедура получит полный контроль над областью изображения, к которой будет применено преобразование.

Реализуем на практике второй способ организации "фильтровалки". При этом сам процесс преобразования изображения вынесем в отдельный поток (назовем его "рабочим" потоком) выполнения программы. Это даст нам (пользователю) возможность контролировать не только область применения фильтра, но и продолжительность выполнения операции. Например, если у пользователя не хватит терпения дождаться окончания преобразования, он сможет остановить работу.

Общая схема преобразования в этом случае будет выглядеть следующим образом:

1. Пришла команда выполнить преобразование — создаем рабочий поток.
2. Уведомляем объекты-облики о том, что начали преобразование. При этом облик запускает таймер и начинает переодически интересоваться, сколько процентов работы выполнено, показывая пользователю процент выполнения.
3. В рабочем потоке выполняется преобразование и увеличивается процент выполнения.
4. По окончании преобразования (или если пользователь прервал выполнение) в объекты-облики посылаются сообщения о завершении работы и показывается преобразованная картинка.

Поскольку данными в программе BMViewer заведует класс `CBMDoc`, именно в него и поместим "фильтровалку". Для создания рабочего потока потребуется добавить в класс `CBMDoc` несколько методов:

- `Transform()` — создает рабочий поток;
- `ThreadProc()` — функция потока, запускает "фильтровалку" для конкретного объекта-документа;
- `TransformLoop()` — сама "фильтровалка";
- `InformAllViews()` — передает сообщения всем обликам документа;

Текст этих методов приведен *в разд. 11.8*, здесь же коротко рассмотрим только метод `TransformLoop()` (листинг 11.11).

Листинг 11.11. Метод CBMDoc::TransformLoop(). Файл BMDoc.cpp

```
void CBMDoc::TransformLoop()
{
```

```
if(m_pCurFilter==NULL) return;
if(!CreateCompatibleBuffer()) return;

m_EventDoTransform.SetEvent();
m_bEditable=FALSE;
InformAllViews(UM_STARTTRANSFORM);

CRaster   *pSBM=GetCurrentBMPtr(),      // источник
          *pDBM=GetBufferBMPtr();       // приемник
// Установили в фильтр источник и приемник преобразований
m_pCurFilter->SetBuffers(pSBM, pDBM);

for(LONG y=0; y<pSBM->GetBMHeight(); y++)
{
   // Процент выполнения
   InterlockedExchange(&m_lExecutedPercent,
                       100*y/pSBM->GetBMHeight());
   // Проверим, не решили ли прервать преобразование
   if(!m_EventDoTransform.Lock(0))
   {
      InformAllViews(UM_ENDOFTRANSFORM, FALSE, 0);
      m_bEditable=TRUE;
      return;
   }
   LONG x=0;
   if( m_bEditHalf ) // Преобразовать только половину изображения
   {
      // Первую половину картинки копируем в буфер без преобразования
      x=pSBM->GetBMWidth()/2;
      BYTE *pSPix=NULL,  *pDPix=NULL;
      // Указатели на начало строк
      if((pSPix=pSBM->GetPixPtr(0, y))!=NULL &&
          (pDPix=pDBM->GetPixPtr(0, y))!=NULL)
         // ВНИМАНИЕ! Предполагается, что 1 пиксел = 24 бита = 3 байта
         memcpy(pDPix, pSPix, 3*x);
   }
   // Преобразование с использованием текущего фильтра
   for(; x<pSBM->GetBMWidth(); x++)
      m_pCurFilter->TransformPix(x, y);
```

```
    }
    m_EventDoTransform.ResetEvent();
    m_bEditable=TRUE;

    SwapBM();           // сделать буфер текущим изображением
    SetModifiedFlag(); // флаг "данные изменились"

    InformAllViews(UM_ENDOFTRANSFORM, TRUE, 0);
    return;
};
```

В методе `TransformLoop()` мы сначала "зажигаем" событие "Выполняется преобразование" — объект `m_EventDoTransform` класса `CEvent`. Затем сообщаем текущему фильтру, какое изображение будет исходным, и какое — приемным (адреса объектов `CRaster`). Далее в цикле прогоняем через фильтр пикселы изображения. На текущий фильтр указывает переменная `m_pCurFilter`, которую мы завели в классе `CBMDoc` специально для этих целей. Тип этой переменной — "указатель на объект класса `CFilter`". Преобразование же данных выполняется с помощью метода `CFilter::TransformPix()`. Класс `CFilter` как раз и является базовым для всех фильтров. О нем рассказано *в разд. 11.6.4.*

В процессе преобразования перед обработкой очередной строки пикселов вычисляется процент выполнения как процент уже обработанных строк изображения. Вычисленное значение записывается в переменную `m_lExecutedPercent` с помощью API-функции `InterlockedExchange()` — эта функция позволяет предотвратить одновременное обращение к переменной из разных потоков. Далее проверяется, по-прежнему ли установлено событие `m_EventDoTransform`. И только затем обрабатываются пикселы строки. Причем в нашей программе в иллюстрационных целях мы позволяем пользователю посмотреть эффект преобразования на половине изображения. Если установлен флаг `m_bEditHalf`, первая половина строки копируется в неизменном виде.

После того как все пикселы изображения были обработаны, скидывается флаг `m_EventDoTransform`, буферное изображение становится активным и во все облики направляется сообщение `UM_ENDOFTRANSFORM` с параметром `TRUE`, который говорит о том, что преобразование завершилось и надо обновить изображение в окне облика.

Посылаемые в облики сообщения о начале и окончании кодирования определены нами в файле BMDoc.h следующим образом:

```
#define UM_STARTTRANSFORM    WM_USER+ 0x8000
#define UM_ENDOFTRANSFORM    UM_STARTTRANSFORM+1
```

WM_USER — это специальная константа, начиная с которой (вплоть до значения 0xBFFF) программист может определять сообщения для использования в своем приложении без опасений о том, что они будут конфликтовать с Windows-сообщениями. Однако в документации MSDN сказано, что в диапазоне до 0×7FFF некоторые предопределенные Windows-классы могут использовать значения в своих целях. Видимо, в связи с этим в [4] предлагается на всякий случай определять свои значения как WM_USER+7 и выше, что, в принципе, работает. Но почему +7 мне не известно, поэтому мы можем пойти дальше в своей осторожности и определить свои сообщения в диапазоне от 0×8000.

Для обработки наших сообщений в классе-облике потребуется сделать следующее:

1. В интерфейс класса `CBMView` (файл BMView.h) добавим объявление методов:

```
afx_msg LONG OnStartTransform(UINT wParam, LONG lParam);
afx_msg LONG OnEndTransform(UINT wParam, LONG lParam);
```

2. В карту сообщений класса `CBMView` (файл BMView.cpp) добавим макрокоманды:

```
ON_MESSAGE(UM_STARTTRANSFORM, OnStartTransform)
ON_MESSAGE(UM_ENDOFTRANSFORM, OnEndTransform)
```

3. Добавим в класс `CBMView` (файл BMView.cpp) реализацию этих методов:

```
LONG CBMView::OnStartTransform(UINT wParam, LONG lParam)
{
   OnStartTimer();
   return 0;
}

LONG CBMView::OnEndTransform(UINT wParam, LONG lParam)
{
   OnStopTimer();
   if(wParam)
      // обновим изображение на виртуальном экране
      UpdateVirtualScreen();
   return 0;
}
```

Как видно из приведенного текста методов, при получении сообщения `UM_STARTTRANSFORM` в объекте-облике вызывается метод `OnStartTimer()`. Этот метод создает таймер. Для обработки сообщений WM_TIMER, которые начнет посылать таймер, в класс CBMView с помощью ClassWizard добавили метод `OnTimer()`. В этом методе будет выполняться запрос процента

выполнения операции и обновляться информация о выполнении. Процент выполнения операции будем показывать в заголовке окна облика. Можно было бы, конечно, вывести индикатор выполнения (progress bar) в строку состояния, но как мы дальше увидим, наша программа позволит одновременно выполнять преобразования в нескольких рисунках, а тогда не совсем ясно, как делить единственный индикатор.

Приход сообщения `UM_ENDOFTRANSFORM` обрабатывается методом `OnEndTransform()`, который зависит от значения аргумента `wParam`:

- `TRUE` – преобразование успешно закончено — выполняет обновление экрана;
- `FALSE` – пользователь прервал операцию — не выполняет обновление экрана.

Далее им вызывается функция `OnStopTimer()`, которая разрушает таймер.

Полностью текст этих методов приведен *в разд. 11.8.*

Схема взаимодействия объектов и потоков показана на рис. 11.12.

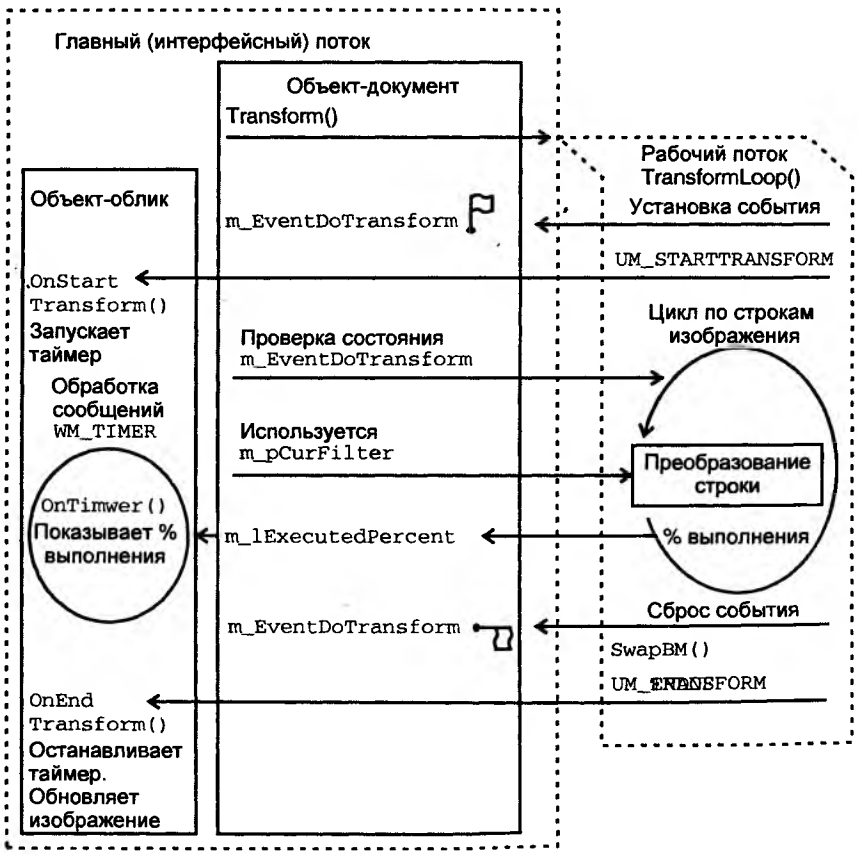


Рис. 11.12. Схема работы программы при выполнении преобразования изображения

Выделение "долгоиграющих" операций обработки данных в отдельный поток позволяет пользователю сохранить контроль над выполнением программы, т. е. программа перестает "подвисать" на таких задачах. В нашем приложении пользователь, запустив фильтрацию на одном из открытых изображений, может переключиться на просмотр и редактирование другого изображения. При необходимости пользователь может остановить выполнение преобразования, для этого в программе предусмотрим команду, которая бы сбрасывала флаг `m_EventDoTransform` При сбросе этого флага цикл выполнения преобразования CBMDoc::TransformLoop() прерывается, потоковая функция завершается и рабочий поток прекращает свое существование.

11.6.3. Таблица преобразования

Таблица преобразования — это просто массив, заполненный какими-то значениями. Размер массива равен максимальному значению, которое может принимать преобразуемая величина. С помощью такой таблицы удобно и быстро заменять одно значение другим. Таблицы преобразований применимы, когда новое значение формируется из единственного старого значения, т. е. при *точечном* преобразовании.

Например, нам надо преобразовать яркость пиксела. В этом случае старое значение яркости играет роль индекса элемента в таблице, а значение этого элемента является новым значением. Формула преобразования может выглядеть так:

```
V = TransformTable[V],
```

где `V` — значение яркости, а `TransformTable` — таблица преобразования. Конечно, таблица преобразования должна быть предварительно заполнена какими-то значениями.

Рассмотрим, например, как можно инвертировать цвет картинки.

Диапазон значений каждого 8-битного компонента цвета находится в пределах от 0 до 255.

Создадим таблицу преобразования из 256 элементов и заполним ее значениями от 255 до 0 (рис. 11.13).

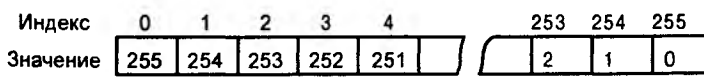


Рис. 11.13. Таблица преобразования "инверсия"

После преобразования по вышеприведенной формуле с использованием таблицы (см. рис. 11.13) интенсивность 255 будет заменена на 0, 254 — на 1 и т. д. В случае такого простого преобразования, как инверсия цвета, использование таблицы может и не дать особого выигрыша по скорости,

но если новое значение пиксела должно рассчитываться по более сложной формуле (чем $V = 255 - V$), то выигрыш будет весьма заметен. Кроме того, использование таблиц позволяет использовать единообразный подход к осуществлению различных преобразований.

11.6.4. Класс "Фильтр"

Структура (см. рис. 11.11) подразумевает существование в программе некоторого объекта-фильтра. Фильтры выполняют разные преобразования, но с точки зрения "фильтровалки" они все одинаковы и обращаться с ними она будет единообразно. Поэтому нам надо определить базовый класс `CFilter` для фильтра с минимальным, но основным набором методов, с помощью которых будет происходить общение. Интерфейс такого класса приведен в листинге 11.12.

Листинг 11.12. Базовый класс фильтров CFIlter. Файл Filter.h

```
class CRaster;
// Базовый виртуальный класс
class CFilter
{
protected:
   CRaster *m_pSourceBM;
   CRaster *m_pDestBM;
public:
   // Устанавливает исходное и приемное изображения
   void SetBuffers( CRaster *pSource, CRaster *pDest=NULL)
      {   m_pSourceBM=pSource;   m_pDestBM=pDest;};
   // Виртуальный метод преобразования пиксела
   // будет переопределен в производных классах
   virtual BOOL TransformPix(LONG x, LONG y){ return FALSE;};
};
```

Данные класса — два указателя на объекты-картинки класса `CRaster`:

- `m_pSourceBM` — адрес объекта "исходная картинка", откуда берутся данные для преобразования;
- `m_pDestBM` — адрес объекта "приемная картинка", куда помещаются преобразованные данные.

Методы класса:

- `SetBuffers()` — сообщает фильтру адреса исходного и приемного изображения;

- `TransformPix()` — преобразует данные одного пиксела с координатами (*x*, *y*). Должен быть переопределен в производных классах.

Переменная-указатель на этот класс `m_pCurFilter` заведена в классе `CBMDoc`. Этой переменной присваивается адрес текущего фильтра. В классе `CFilter` уже объявлены необходимые для фильтрования методы, они и используются в методе `CBMDoc::TransformLoop()` (см. листинг 11.11). Так как метод `CFilter::TransformPix()` объявлен виртуальным, в методе `TransformLoop()` будет происходить вызов настоящего метода преобразования активного фильтра.

Для реализации точечных методов преобразования создадим класс `CDotFilter` (листинг 11.13).

Листинг 11.13. Базовый класс для точечных фильтров `CDotFilter`. Файл Filter.h

```
// Базовый класс для точечных фильтров
class CDotFilter: public CFilter
{
protected:
   // Таблицы преобразования для компонентов цвета
   BYTE BGRTransTable[3][256];
public:
   // Метод преобразования пиксела
   BOOL TransformPix(LONG x, LONG y);
};
```

Данными этого класса являются три таблицы преобразования компонентов RGB цвета. В принципе, для методов преобразования, рассмотренных далее, достаточно было бы определить одну таблицу, но более общим подходом будет все-таки формирование трех таблиц. Потому что возможны преобразования, изменяющие цветовую гамму (оттенок) изображения. Тут-то три таблицы и пригодятся.

Для точечного фильтра переопределен метод `TransformPix()`. Он будет общим для большинства рассмотренных далее точечных фильтров. Реализация метода приведена в листинге 11.14.

Листинг 11.14. Метод `CDotFilter::TransformPix()`. Файл Filter.cpp

```
BOOL CDotFilter::TransformPix(LONG x, LONG y)
{
   BYTE *pDPix=NULL, *pSPix=NULL;
```

```
    // Источник необходим
    if(m_pSourceBM==NULL )
       return FALSE;
    // Если приемник не задан, то преобразование помещаем в источник
    if(m_pDestBM==NULL)
       m_pDestBM=m_pSourceBM;
    // Получаем указатели на пикселы в источнике и приемнике
    if((pDPix=m_pDestBM->GetPixPtr(x, y))==NULL ||
       (pSPix=m_pSourceBM->GetPixPtr(x, y))==NULL)
       return FALSE;
    // Преобразование. Порядок BGR
    *pDPix=BGRTransTable[0][*pSPix];
    *(pDPix+1)=BGRTransTable[1][*(pSPix+1)];
    *(pDPix+2)=BGRTransTable[2][*(pSPix+2)];
    return TRUE;
};
```

В принципе, в случае точечных преобразований возможна такая реализация "фильтровалки", когда исходное изображение одновременно является и приемником преобразования (конечно, в этом случае сложнее сделать отмену преобразования). Эта ситуация отрабатывается внутри метода.

Хотя формат 24-битового цвета называют RGB, в файле формата BMP компоненты цвета хранятся в обратном порядке. Это не так важно, но надо знать и учитывать, что и как делается при формировании нового значения цвета пиксела.

Все, что останется сделать в производных от `CDotFilter` классах, описывающих разные эффекты, — это реализовать инициализацию таблиц преобразования.

Для реализации пространственных (матричных) методов преобразования создадим класс `CMatrixFilter`. Интерфейс класса приведен в листинге 11.15.

Листинг 11.15. Интерфейс базового для матричных фильтров класса `CMatrixFilter`. Файл Filter.h

```
// Пространственные (матричные) фильтры
// Базовый класс
class CMatrixFilter: public CFilter
{
protected:
   int m_rangX; // размер матрицы по X и Y
```

```
    int m_rangY;
    const int *m_pMatrix; // указатель на матрицу
public:
    // Метод преобразования пиксела
    BOOL TransformPix(LONG x, LONG y);
};
```

Данные класса: размер матрицы преобразования и указатель на матрицу. Как правило, используются квадратные матрицы преобразования, но кто знает, может для чего-то будет полезна и не квадратная матрица. Поэтому указывается размер матрицы по горизонтали и вертикали. Размер матрицы определяет зону пикселов, окружающую пиксел (*x*, *y*), которая будет вовлечена в расчет нового значения пиксела (*x*, *y*). Указателю на матрицу преобразования `m_pMatrix` будет присваиваться адрес матрицы, которая будет использована в преобразовании.

Реализация метода `CMatrixFilter::TransformPix()` приведена в листинге 11.16.

Листинг 11.16. Метод `CMatrixFilter::TransformPix()`. Файл Filter.cpp

```
BOOL CMatrixFilter::TransformPix(LONG x, LONG y)
{
    BYTE *pDPix=NULL, *pSPix=NULL;
    // Источник и приемник необходимы
    if( m_pSourceBM==NULL || m_pDestBM==NULL)
        return FALSE;

    // Определяем зону перекрытия изображения и
    // матрицы преобразования. Это требуется для
    // обработки пикселов, находящихся на границах
    // изображения
    int x_start=0;
    int dx=m_rangX/2, dy=m_rangY/2;

    if(x-dx<0) x_start=dx-x;

    int y_start=0;
    if(y-dy<0) y_start=dy-y;

    int x_finish=m_rangX;
    if(x+dx>m_pSourceBM->GetBMWidth())
```

```
    x_finish-=(x+dx-m_pSourceBM->GetBMWidth());

int y_finish=m_rangY;
if(y+dy>m_pSourceBM->GetBMHeight() )
    y_finish-=(y+dy-m_pSourceBM->GetBMHeight());

// Расчет новых значений цвета пиксела
// с учетом соседей, попавших в зону действия
// матрицы преобразования
int NewBGR[3];
int count=0;
for(int c=0, mx=0, my=0; c<3; c++)
{
   NewBGR[c]=0; count=0;
   for(my=y_start; my<y_finish; my++)
   for(mx=x_start; mx<x_finish; mx++)
   {
      if((pSPix=m_pSourceBM->GetPixPtr(x+(mx-dx),
                                       y+(my-dy)))!=NULL)
      {
         NewBGR[c]+=(m_pMatrix[my*m_rangX+mx]*(*(pSPix+c)));
         count+=m_pMatrix[my*m_rangX+mx];
      }
   }
}
// Адрес пиксела в изображении-приемнике
pDPix=m_pDestBM->GetPixPtr(x, y);

// Установка нового значения в приемное изображение
for(c=0; c<3; c++)
{
   // Приведение значения к допустимому диапазону
   if(count!=0)
      NewBGR[c]=NewBGR[c]/count;
   if(NewBGR[c]<0)
      NewBGR[c]=0;
   else if(NewBGR[c]>255)
      NewBGR[c]=255;

   *(pDPix+c)=NewBGR[c];
```

```
}

   return TRUE;

};
```

В методе `CMatrixFilter::TransformPix()` сначала определяется область перекрытия изображения и матрицы преобразования. Этот шаг необходим в связи с тем, что на границах изображения пиксел может не иметь соседей с одной или двух сторон. На рис. 11.14 показаны некоторые ситуации, когда в преобразовании задействованы не все коэффициенты матрицы. Пикселу, над которым выполняется преобразование, соответствует индекс матрицы с номером 5.

Рис. 11.14. Пересечение матрицы и изображения

Новое значение пиксела формируется с учетом значений всех пикселов и коэффициентов матрицы преобразования, попавших в область перекрытия изображения и матрицы преобразования.

11.6.5. Использование гистограммы яркости для повышения контрастности изображения. Фильтр "Гистограмма"

Гистограмма показывает, насколько полно используется в изображении диапазон яркостей. Изображения со слабым контрастом имеют гистограмму яркости, сгруппированную в небольшом диапазоне значений. Гистограмма

таких изображений может быть смещена в область темных или светлых тонов, но может располагаться и в центре диапазона яркостей (рис. 11.15).

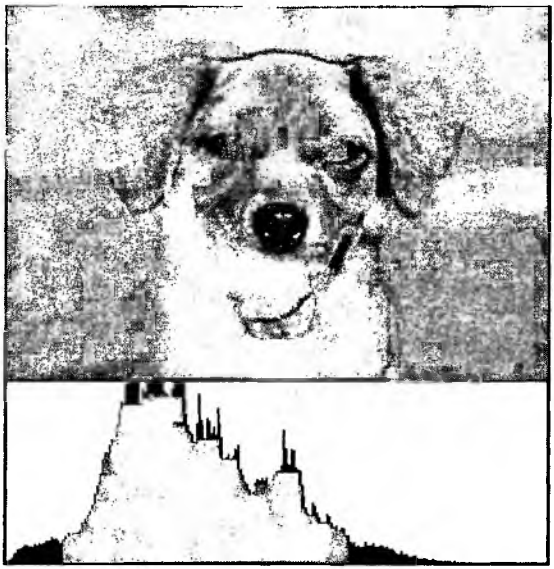


Рис. 11.15. Изображение со слабой контрастностью

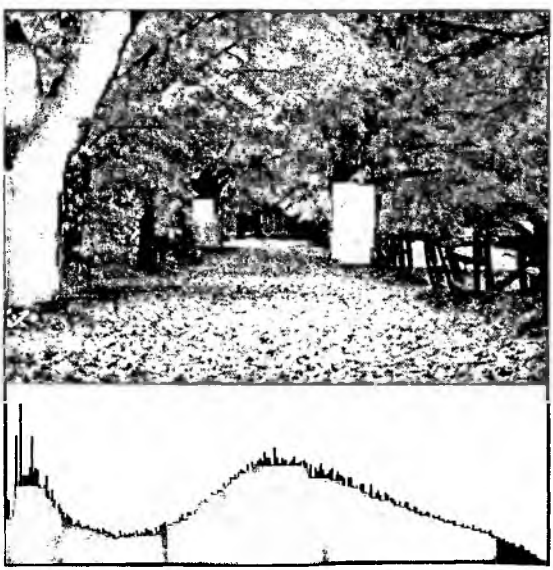


Рис. 11.16. Изображение с хорошей контрастностью

Гистограмма изображения с хорошим контрастом, как правило, равномерно занимает весь диапазон яркостей (рис. 11.16). Такие изображения обычно воспринимаются как более качественные.

В высококонтрастных изображениях также задействован весь диапазон яркостей, но гистограмма такого изображения может иметь пики, что связано с наличием больших зон темных и светлых пикселов (рис. 11.17).

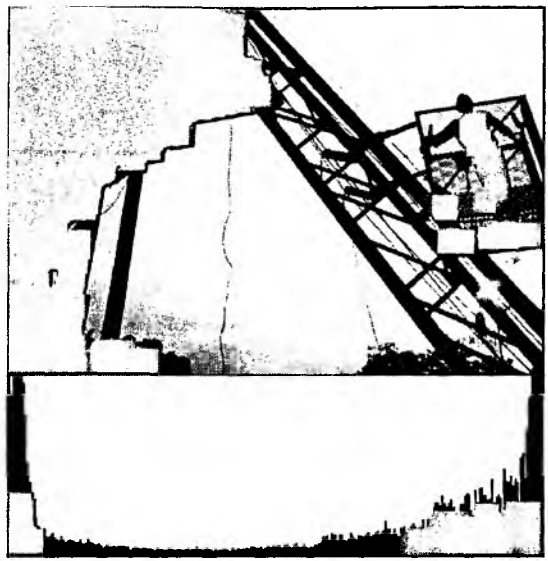


Рис. 11.17. Изображение с высокой контрастностью

Рассмотрим, как можно использовать информацию о распределении яркости для коррекции контрастности. Например, на рис. 11.10 видно, что в фотографии практически совсем нет черных пикселов и пикселов, цвет которых близок к белому.

Мы можем попытаться исправить ситуацию, "растянув" яркость пикселов картинки на весь диапазон от 0 до 255. При этом пикселы, которые были темными, станут еще темнее (вплоть до черного), а светлые пикселы станут еще светлее (вплоть до белого).

Шаблон окна диалога "Гистограмма" (`IDD_HIST`) имеет два ползунка (элементы *slider*). Используем их для того, чтобы определить нижнюю и верхнюю границы диапазона значений яркости.

Прежде всего, добавим в класс `CHistDlg` (с помощью ClassWizard) обработку сообщения `WM_INITDIALOG`. Сообщение `WM_INITDIALOG` посылается окну диалога перед тем, как оно будет показано на экране. В методе-обработчике этого сообщения установим начальные параметры ползунков и позиции "бегунков" (листинг 11.17).

Листинг 11.17. Обработка сообщения WM_INITDIALOG в классе CHistDlg. Файл HistDlg.cpp

```
BOOL CHistDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
   // Ползунок нижней границы
   m_ctrlOffset_b.SetRange(0, 127);
   // Бегунок в крайнем левом положении
   m_ctrlOffset_b.SetPos(0);
   // Ползунок верхней границы
   m_ctrlOffset_t.SetRange(128, 255);
   // Бегунок в крайнем правом положении
   m_ctrlOffset_t.SetPos(255);
   // Текст
   m_strOffset_b="0";
   m_strOffset_t="0";
   UpdateData(FALSE);
   return TRUE;
}
```

Напомним, что переменные `m_ctrlOffset_b` и `m_ctrlOffset_t` — это объекты класса `CSliderCtrl`, связанные с ползунками, а `m_strOffset_b` и `m_strOffset_t` — объекты класса `CString`, связанные с элементами "static" в окне диалога (см. рис. 11.6 и 11.8).

Далее добавим в класс `CHistDlg` обработчики еще двух сообщений: `WM_HSCROLL`, которое будет поступать при перемещении бегунка (листинг 11.18), и сообщение о нажатии пользователем кнопки **OK** в диалоге (листинг 11.19). При обработке сообщения `WM_HSCROLL` в окне диалога будет выводиться позиция бегунков. При обработке сообщения "нажата кнопка **OK**" позиции бегунков запоминаются в переменных `m_iOffset_b` и `m_iOffset_e`. Эти переменные целого типа добавлены в класс `CHistDlg` специально для того, чтобы можно было извлечь из него информацию о позициях бегунков после закрытия окна диалога.

Листинг 11.18. Обработка сообщения WM_HSCROLL в классе CHistDlg. Файл HistDlg.cpp

```
void CHistDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
   m_strOffset_b.Format("%d", m_ctrlOffset_b.GetPos());
```

```
    m_strOffset_t.Format("%d", 255-m_ctrlOffset_t.GetPos());
    UpdateData(FALSE);
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

Листинг 11.19. Обработка сообщения "нажата кнопка OK" в классе CHistDlg. Файл HistDlg.cpp

```
void CHistDlg::OnOK()
{
    m_iOffset_b=m_ctrlOffset_b.GetPos();
    m_iOffset_t=255-m_ctrlOffset_t.GetPos();
    CDialog::OnOK();
}
```

Теперь, когда все подготовительные операции выполнены, нам известен диапазон значений яркости, который надо растягивать. Остается только создать нужный фильтр и сообщить ему этот диапазон.

Коррекция яркости — это точечный процесс. Поэтому создадим класс фильтра гистограмма CHistogram как производный от класса CDotFilter (листинг 11.20).

Листинг 11.20. Интерфейс класса CHistogram. Файл Filter.h

```
// Гистограмма
class CHistogram: public  CDotFilter
{
public:
    BOOL Init(int offset_b, int offset_t);
};
```

У этого класса объявлен всего один новый метод Init(), в который передаются смещения от нижней и верхней границы диапазона яркостей (листинг 11.21). В методе Init() производится заполнение таблиц преобразования новыми значениями. При этом полный диапазон яркостей от 0 до 255 равномерно распределяется на заданный в диалоге "Гистограмма" диапазон.

Листинг 11.21. Метод CHistogram::Init(). Файл Filter.cpp

```
BOOL CHistogram::Init(int offset_b, int offset_t)
{
```

```
    int range=0;
    // Все элементы в таблицах с индексом от 0 до нижней границы
    // установим в 0
    for(int i=0, t=0; t<3; t++)
       for(i=0; i<offset_b; i++)
       {
          BGRTransTable[t][i]=0;
       }
    // Все значения в таблицах с индексом от 255 до верхней границы
    // установим в 255
    for(t=0; t<3; t++)
       for(i=255; i>=256-offset_t; i--)
       {
          BGRTransTable[t][i]=255;
       }
    // Все значения в таблицах с индексом от нижней до верхней границы
    // равномерно распределим на диапазон от 0 до 255
    double step=256./(256-(offset_b+offset_t));
    for(t=0; t<3; t++)
    {
       double value=0.;
       for(i=offset_b; i<256-offset_t; i++)
       {
          BGRTransTable[t][i]=(int)((value)+0.5);
          value+=step;
       }
    }
    return TRUE;
};
```

Вернемся теперь к листингу 11.10 и посмотрим, что же происходит после строки:

```
// Требуется выполнить коррекцию контрастности
```

После того как пользователь нажал кнопку **OK** в диалоге "Гистограмма", мы проверяем, "а не сдвинул ли он бегунки на ползунках". Если такое событие произошло, то мы инициализируем объект `m_HistogramFilter` (объявлен в интерфейсе класса `CBMDoc`), и делаем его активным, а затем вызываем метод `CBMDoc::Transform()`, запускающий рабочий поток, в котором и происходит преобразование (см. листинг 11.11).

Посмотрим, какой же эффект даст наша коррекция яркости. Установим диапазон яркостей, который должен быть растянут, передвинем бегунки так, как показано на рис. 11.18.

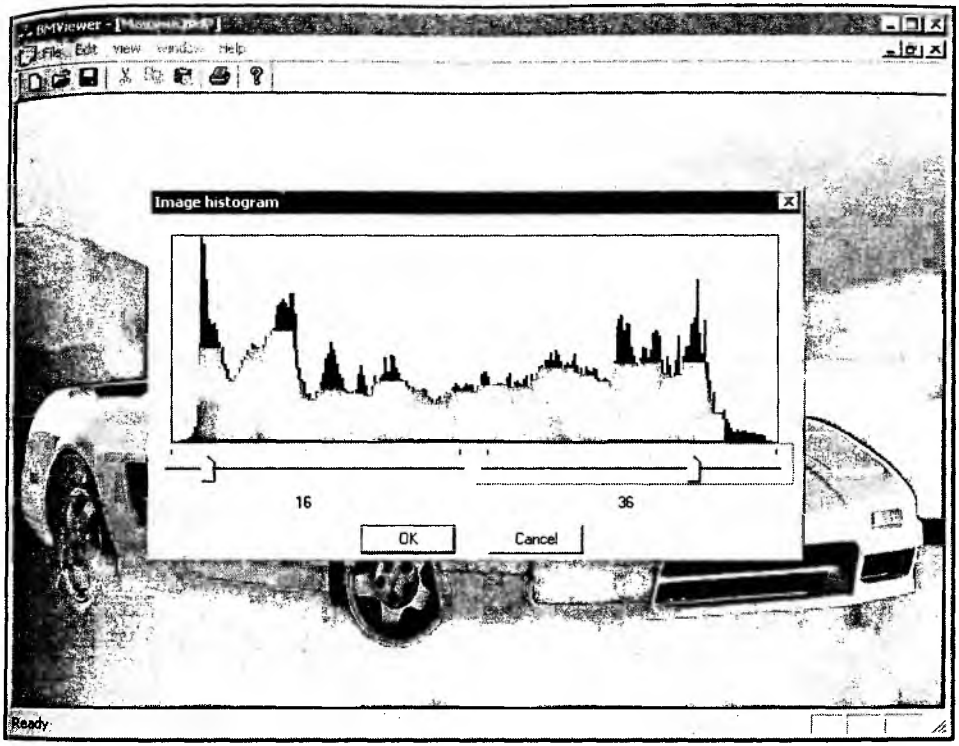


Рис. 11.18. Коррекция яркости

Результат коррекции яркости показан на рис. 11.19, а гистограмма изображения после коррекции — на рис. 11.20. Для того чтобы эффект преобразования был лучше заметен, на рис. 11.19 показано изображение, лишь половина которого была преобразована. Метод `CBMDoc::Transform()` имеет такую возможность (листинг 11.11). Гистограмма же на рис. 11.20 соответствует полностью откорректированному изображению.

Мы рассмотрели процесс коррекции яркости вручную. Эту операцию можно выполнить и автоматически. При автоматической коррекции яркости алгоритм должен сначала найти границы корректируемого диапазона. Для этого нужно задать некоторое пороговое значение яркости (например, определить его от уровня средней яркости), а затем выполнить сканирование гистограммы изображения и найти индексы элементов гистограммы, значения которых находятся на заданном пороге, — определить диапазон коррекции. Далее коррекция выполняется точно так же, как и при "ручном" варианте.

Рис. 11.19. Результат коррекции яркости

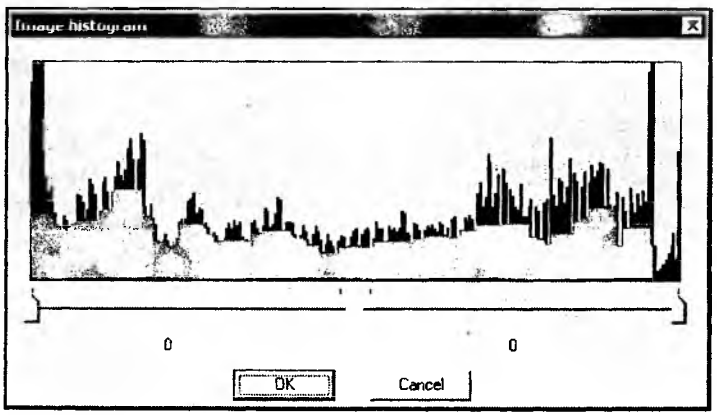


Рис. 11.20. Гистограмма яркости после коррекции

11.6.6. Фильтр "Яркость/Контраст"

Многие графические редакторы имеют возможность изменить яркость и контраст изображения. Изменение яркости заключается в простом увеличении или уменьшении интенсивности цвета всех пикселов на заданное значение (константу). Выражается это в смещении гистограммы яркости вправо или влево по шкале. Контраст же может быть как увеличен, так и уменьшен.

Данное преобразование является точечным. Для его реализации добавим в программу фильтр "Яркость/Контраст" класс `CBrightCont`, производный от класса `CDotFilter`. Интерфейс класса приведен в листинге 11.22.

Листинг 11.22. Интерфейс класса `CBrightCont`. Файл Filter.h

```
// Яркость-контраст
class CBrightCont: public  CDotFilter
{
public:
   BOOL Init(int b_offset, int c_offset);
};
```

Интерфейс класса `CBrightCont` ничем не отличается от `CHistogram` (листинг 11.20). Однако значения `b_offset` и `c_offset` параметров метода `CBrightCont::Init()` могут быть как положительными, так и отрицательными, что будет соответствовать увеличению или уменьшению яркости/контрастности изображения.

Реализация метода `CBrightCont::Init()` приведена в листинге 11.23. Этот метод инициализирует таблицы преобразования. Сначала выполняется смещение яркости на заданную величину, а затем либо "сжатие", либо "растяжение" диапазона яркости. Причем при сжатии значения яркости изменяются не равномерно, а пропорционально их удаленности от "серой середины", определенной константой `CONTRAST_MEDIAN`. Значение `CONTRAST_MEDIAN 159` задает более светлый оттенок серого (чем, например, 127 — арифметическая середина диапазона яркости) и для многих изображений дает хороший результат. Вы можете поэкспериментировать со значением этой константы и посмотреть, что из этого получится. Однако более правильный подход — не задавать середину диапазона яркости константным значением, а определять на основе гистограммы исходного изображения.

Растяжение диапазона яркости выполняется так же, как было рассмотрено в предыдущем разделе, за исключением того, что смещение по шкале яркости сверху и снизу одинаково и задается параметром `c_offset`. Еще одним отличием является то, что после преобразования яркости работа по коррекции контрастности происходит со значениями таблицы преобразования, полагая при этом, что они являются индексами в таблице, полученной после коррекции яркости.

Листинг 11.23. Метод `CBrightCont::Init()`. Файл Filter.cpp

```
// "Серая середина"
#define CONTRAST_MEDIAN 159
```

```
BOOL CBrightCont::Init(int b_offset, int c_offset)
{
   int i=0,    // Индекс цвета в таблице преобразования
      t=0,    // Индекс таблицы
      // Индекс цвета, соответствующего нижней границе яркости
      t_index=0,
      // Индекс цвета, соответствующего верхней границе яркости
      b_index=0,
      value_offset; // Смещение значения цвета
   double value=0.; // Новое значение цвета
   // Изменяем яркость
   for(i, t=0; t<3; t++)
      for(i=0; i<256; i++)
      {
         if( i+b_offset>255) BGRTransTable[t][i]=255;
         else if( i+b_offset<0) BGRTransTable[t][i]=0;
         else BGRTransTable[t][i]=i+b_offset;
      }
   // Изменяем контрастность
   if(c_offset<0)// Уменьшаем контрастность
   {
      for(i=0, t=0; t<3; t++)
      for(i=0; i<256; i++)
      if(BGRTransTable[t][i]<CONTRAST_MEDIAN)
      {
         // Расчитываем смещение в зависимости от удаленности цвета от
         // "серой середины"
         value_offset=(CONTRAST_MEDIAN-BGRTransTable[t][i])*c_offset/128;
         if(BGRTransTable[t][i]-value_offset>CONTRAST_MEDIAN)
            BGRTransTable[t][i]=CONTRAST_MEDIAN;
         else BGRTransTable[t][i]-=value_offset;
      }
      else
      {
         // Расчитываем смещение в зависимости от удаленности цвета от
         // "серой середины"
         value_offset=(BGRTransTable[t][i]-CONTRAST_MEDIAN)*c_offset/128;
         if(BGRTransTable[t][i]+value_offset<CONTRAST_MEDIAN)
            BGRTransTable[t][i]=CONTRAST_MEDIAN;
```

```
      else BGRTransTable[t][i]+=value_offset;
   }
}
else   if(c_offset>0)
//Увеличиваем контрастность
{
   // Расчет нижней границы цвета
   int offset_b=c_offset*CONTRAST_MEDIAN/128;
   // Все значения в таблице ниже нижней границы получат значения 0
   for(t=0; t<3; t++)
   for(b_index=0; b_index<256; b_index++)
   {
      if(BGRTransTable[t][b_index]<offset_b)
         BGRTransTable[t][b_index]=0;
      else break;
   }
   // Расчет верхней границы цвета
   int offset_t=c_offset*128/CONTRAST_MEDIAN;
   // Все значения выше верхней границы получат значения 255
   for(t=0; t<3; t++)
   for(t_index=255; t_index>=0; t_index--)
   {
      if(BGRTransTable[t][t_index]+offset_t>255)
         BGRTransTable[t][t_index]=255;
      else break;
   }
   // Расчет шага изменения интенсивности цвета
   double step=256./(256-(offset_b+offset_t));
   // "Растягиваем" интенсивность цветов между нижней и верхней
   // границами, чтобы они занимали весь диапазон от 0 до 255
   for(t=0; t<3; t++)
   {
      value=0.;
      for(i=b_index; i<=t_index; i++)
      {
         if(BGRTransTable[t][i]>=offset_b ||
            BGRTransTable[t][i]<256-offset_t)
         {
            value=(int)((BGRTransTable[t][i]-offset_b)*step+0.5);
```

```
                if(value>255) value=255;
                BGRTransTable[t][i]=(int)(value);
            }
        }
    }
}
return TRUE;
};
```

Для того чтобы пользователь мог указать значения коррекции яркости и контрастности, добавим в программу диалог "Яркость/Контраст" (рис. 11.21). Создание окна диалога очень мало отличается от создания окна для диалога "Гистограмма".

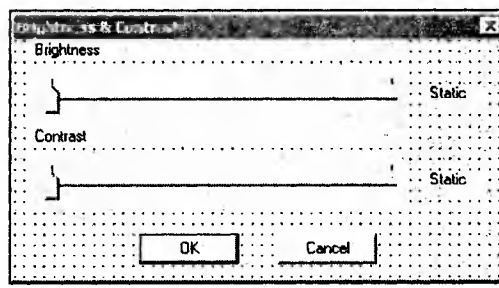


Рис. 11.21. Шаблон диалога коррекции яркости и контрастности

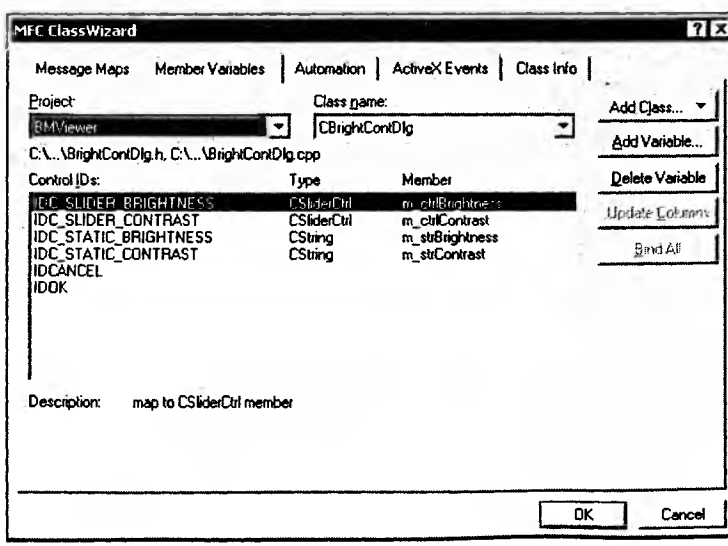


Рис. 11.22. Переменные класса `CBrightContDlg`

Присвоим диалогу идентификатор IDD_BRIGHT_CONT и создадим класс CBrightContDlg, свяжем с элементами управления переменные (рис. 11.22).

Для вызова диалога добавим в меню программы команду **Edit | Brightness and Contrast**, а в класс CBMDoc — метод обработки этой команды (листинг 11.24). В этом методе сначала создается диалог, и если пользователь нажал кнопку **OK**, то инициализируется фильтр "Яркость/Контраст" (объект m_BrightContFilter класса CBrightCont описан в интерфейсе класса CBMDoc). Затем фильтр делается активным и вызывается "фильтровалка" — метод CBMDoc::Transform().

Листинг 11.24. Метод-обработчик команды Edit-Brightness and Contrast. Файл BMDoc.cpp

```
void CBMDoc::OnEditBrightnessandcontrast()
{

   CBrightContDlg BCDlg;
   if(BCDlg.DoModal()==IDCANCEL) return;

   if(BCDlg.m_iBrightnessOffset!=0 || BCDlg.m_iContrastOffset!=0)
   {
      m_BrightContFilter.Init(BCDlg.m_iBrightnessOffset,
                              BCDlg.m_iContrastOffset);
      m_pCurFilter=&m_BrightContFilter;
      Transform();
   }
}
```

Посмотрим же на эффект, который дает фильтр яркости и контрастности. Попробуем сначала немного увеличить яркость фотографии, показанной на рис. 11.15. На рис. 11.23 показана та же фотография, яркость половины которой увеличена на 30 единиц[1] коррекции.

Гистограмма этой же картинки (если преобразование применено ко всей ее площади) показана на рис. 11.24. Видно, что по сравнению с гистограммой (рис. 11.15) она сместилась в область светлых тонов.

[1] Диапазоном коррекции считаем половину диапазона яркости (255/2 = 127), но на ползунке он представлен значением MAX_CORRECTION_OFFSET. Поэтому 1 единица коррекции равна 127/MAX_CORRECTION_OFFSET. Это просто особенность программной реализации.

Рис. 11.23. Фотография после коррекции яркости

Рис. 11.24. Гистограмма яркости фотографии после коррекции яркости

Рис. 11.25. Фотография после коррекции яркости и контраста

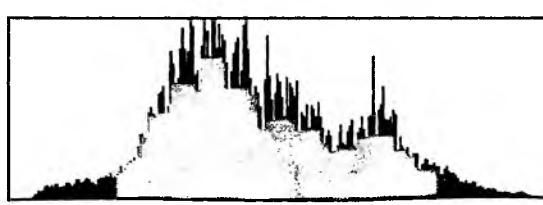


Рис. 11.26. Гистограмма яркости фотографии после коррекции яркости и контраста

Продолжим редактирование этой фотографии и попробуем изменить контрастность изображения, для этой цели можно, конечно, воспользоваться услугами коррекции контрастности диалога "Гистограмма", но можно испытать и новые средства. Увеличим контраст на 20 единиц коррекции. Результат показан на рис. 11.25.

Гистограмма яркости изображения после коррекции контраста всей фотографии показана на рис. 11.26. Изображение стало контрастней, а гистограмма теперь охватывает более широкий диапазон значений.

Конечно, яркость и контраст можно и уменьшать. Поэкспериментируйте с этим.

11.6.7. Фильтр "Инверсия цветов"

Рассмотрим далее совсем простой фильтр, который мы уже обсудили *в разд. 11.6.3*, когда говорили о таблицах преобразования. Реализуется этот фильтр классом `CInvertColors` (листинг 11.25).

Листинг 11.25. Интерфейс класса `CInvertColors`. Файл Filter.h

```
// Инверсия цветов
class CInvertColors: public  CDotFilter
{
public:
   CInvertColors();
};
```

Операция инверсии цветов не требует никаких настроечных параметров, поэтому инициализация таблиц преобразования выполняется в конструкторе класса (листинг 11.26).

Листинг 11.26. Конструктор класса `CInvertColors`. Файл Filter.cpp

```
CInvertColors::CInvertColors()
{
   for(int i=0, t=0; t<3; t++)
      for(i=0; i<256; i++)
      {
         BGRTransTable[t][i]=255-i;
      }
};
```

Вот это фильтр, все бы так просто реализовались, а какой эффект (рис. 11.27)!

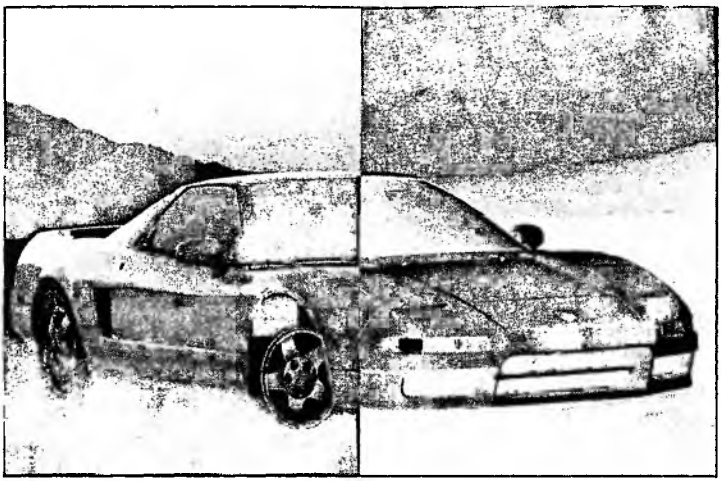


Рис. 11.27. Половина изображения обработана фильтром "Инверсия"

Конечно, как и для прочих фильтров, в классе `CBMDoc` завели объект класса `CInvertColors`, а в меню добавили соответствующую команду и в класс `CBMDoc` — метод-обработчик. В общем, "все, как у всех".

11.6.8. Фильтр "Рельеф"

Рассмотрим еще один точечный фильтр, который, однако, может быть отнесен и к пространственным (а если присмотреться, то в нем можно заметить признаки и покадрового, и геометрического процессов). Этот фильтр может быть реализован с использованием матриц или без их применения. Мы рассмотрим вариант "без". Эффект, создаваемый фильтром, может быть сравнен с получением отпечатка на незастывшем бетоне или высеканием рельефа на камне. Этот фильтр можно найти во многих графических редакторах (англоязычный вариант названия — "Emboss").

Достигается такой эффект простым вычитанием яркости пиксела из яркости пиксела, смещенного на несколько пикселов, например в сторону и вверх. Полученная разница затем смещается в область серых тонов.

Фильтр "Рельеф" реализован классом CEmboss (листинг 11.27).

Листинг 11.27. Интерфейс класса `CEmboss`. Файл Filter.h

```
// Рельеф
class CEmboss: public  CDotFilter
```

```
{
public:
   BOOL TransformPix(LONG x, LONG y);
};
```

Из-за того что преобразование в фильтре `CEmboss` отличается от точечного, пришлось переопределить метод `CDotFilter::TransformPix()`. Вот оно, достоинство объектно-ориентированного стиля, — больше ничего менять не пришлось. Вся остальная схема работы с фильтром осталась без изменений.

Реализация метода `TransformPix()` приведена в листинге 11.28. Константы `STONE_OFFSET_X` и `STONE_OFFSET_Y` задают расстояние и направление смещения вычитаемого пиксела и влияют на получаемый эффект. Можно создать диалог, в котором этими параметрами можно было бы "порулить". В приведенной реализации функции `CEmboss::TransformPix()` всем компонентам цвета задается одинаковая интенсивность. Это тоже необязательно, можно окрашивать результат в любые оттенки, как это сделано, например, в редакторе Ulead Photoimpact.

Листинг 11.28. Метод `CEmboss::TransformPix()`. Файл Filter.cpp

```
#define STONE_OFFSET_X 3
#define STONE_OFFSET_Y -3
BOOL CEmboss::TransformPix(LONG x, LONG y)
{
   BYTE *pDPix=NULL, *pSPix1=NULL, *pSPix2=NULL;
   // Источник и приемник необходимы
   if(m_pSourceBM==NULL|| m_pDestBM==NULL)
      return FALSE;

   // Получаем указатели на пикселы в источнике и приемнике
   if((pDPix=m_pDestBM->GetPixPtr(x, y))==NULL ||
      (pSPix1=m_pSourceBM->GetPixPtr(x, y))==NULL)
      return FALSE;

   if((pSPix2=m_pSourceBM->GetPixPtr(x+STONE_OFFSET_X,
                                     y+STONE_OFFSET_Y))==NULL)
      pSPix2=pSPix1;
   // Расчет яркости
   BYTE Y1, Y2;
   Y1=(BYTE)(0.11*(*pSPix1) + 0.59*(*(pSPix1+1)) + 0.3*(*(pSPix1+2)));
```

```
    Y2=(BYTE)(0.11*(*pSPix2) + 0.59*(*(pSPix2+1)) + 0.3*(*(pSPix2+2)));
    // Находим разницу и смещаем ее в серую область
    BYTE d=(Y1-Y2+255)/2;

    // Пиксел получает новые значения
    *pDPix=d;
    *(pDPix+1)=d;
    *(pDPix+2)=d;
    return TRUE;
};
```

Результат применения фильтра показан на рис. 11.28.

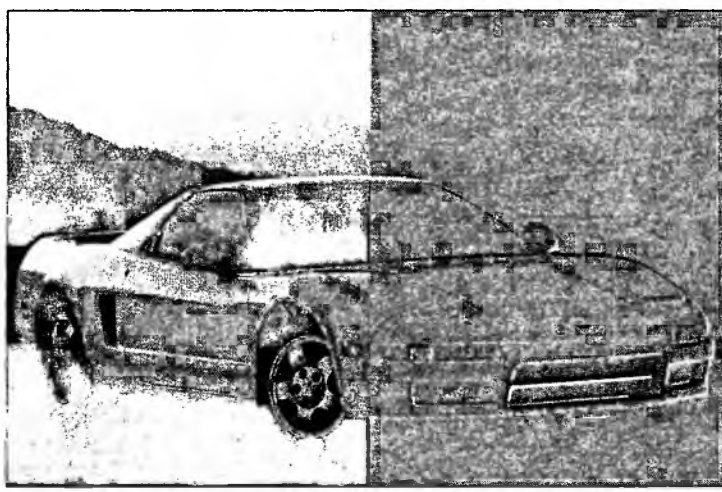


Рис. 11.28. Половина изображения обработана фильтром "Рельеф"

11.6.9. Фильтр "Размытие"

Фильтр "Размытие" — это уже пространственное преобразование. Суть преобразования мы рассмотрели *в разд. 11.6*, а все сложности закончились в методе `CMatrixFilter::TransformPix()`, см. листинг 11.16. Применение этого фильтра оказывает эффект сглаживания деталей изображения. Казалось бы, зачем портить картинку, но некоторым это нравится, а кроме того, иногда процесс "размытия" бывает полезен, в чем мы далее убедимся.

Фильтр реализуется классом `CBlur` (листинг 11.29).

Листинг 11.29. Интерфейс класса CBlur. Файл Filter.h

```
class CBlur: public CMatrixFilter
{
public:
   CBlur();
};
```

Все особенности этого фильтра заключаются в его конструкторе (листинг 11.30).

Листинг 11.30. Метод CBlur::CBlur(). Файл Filter.cpp

```
const int BlurMatrix[25]=
   {   1,   1,   1,   1,   1,
       1,   1,   1,   1,   1,
       1,   1,   1,   1,   1,
       1,   1,   1,   1,   1,
       1,   1,   1,   1,   1
   };

CBlur::CBlur()
{
   m_pMatrix=BlurMatrix;
   m_rangX=5;
   m_rangY=5;
};
```

Рис. 11.29. Половина изображения обработана фильтром "Размытие"

Матрица `BlurMatrix` задает преобразование "размытие", а в конструкторе `CBlur()` просто запоминается ее адрес и размер.

Эффект применения фильтра "Размытие" показан на рис. 11.29.

Для того чтобы эффект был явно заметен на картинке, фильтр пришлось применить несколько раз.

11.6.10. Фильтр "Контур"

Фильтр "Контур" используется для выделения высокочастотных элементов изображения. Он широко применяется при распознавании образов и в машинном зрении. Высокочастотный элемент изображения — это, например, светлый пиксел на однородном темном фоне или наоборот. Теорию этого дела вы можете прочитать в литературе, о которой я упоминал выше. Практически все очень просто. Матрица преобразования (листинг 11.31) определяет весовые коэффициенты пикселов в операции сложения (листинг 11.16).

Листинг 11.31. Матрица преобразования "Контур"

```
// Коэффициент четкости границ,
// его вполне можно задавать в диалоге
#define CONTOUR_COEFF 3
const int ConturMatrix[9]=
   {   -1*CONTOUR_COEFF,   -1*CONTOUR_COEFF,   -1*CONTOUR_COEFF,
       -1*CONTOUR_COEFF,   8*CONTOUR_COEFF,   -1*CONTOUR_COEFF,
       -1*CONTOUR_COEFF,   -1*CONTOUR_COEFF,   -1*CONTOUR_COEFF
   };
```

Представим теперь, что яркость пикселов, попавших в область действия матрицы, примерно одинакова. Это значит, что после сложения получится сумма, близкая к нулю. Если же преобразуемый пиксел (ему соответствует центральный элемент матрицы) имеет яркость, превышающую окружающие пикселы, то результат сложения будет больше нуля. Заметьте, что сумма элементов матрицы равна нулю. Поэтому изображение превратится в черное с белыми контурами. Если же центральный элемент матрицы сделать, например, равным 9, то тогда цвета изображения в основном не изменятся, будут выделены лишь границы.

Фильтр реализуется классом `CContour` (листинг 11.32).

Листинг 11.32. Интерфейс класса CContour. Файл Filter.h

```
class CContour: public CMatrixFilter
{
```

```
public:
   CContour();
};
```

В конструкторе этого класса запоминается нужная матрица и ее размер (листинг. 11.33). В общем-то, такие фильтры, как "Размытие" и "Контур", можно было бы реализовать одним классом, просто инициализировать разными матрицами.

Листинг 11.33. Метод `CBlur::CBlur()`. Файл Filter.cpp

```
CContour::CContour()
{
   m_pMatrix=ConturMatrix;
   m_rangX=3;
   m_rangY=3;
}
```

На рис. 11.30 показан результат применения фильтра "Контур" (так видит Терминатор своим страшным красным глазом)[1].

Рис. 11.30. Половина изображения обработана фильтром "Контур"

[1] Шутка.

11.6.11. Фильтр "Четкость"

Фильтр четкость иллюстрирует собой единство и борьбу противоположностей, и если вдуматься, то можно обнаружить в нем глубокий философский смысл. Для повышения четкости изображения в фильтре используется матрица "Размытие". Задача повышения четкости изображения заключается в том, чтобы выделить высокочастотные детали изображения. Светлые детали сделать ярче, темные — темнее. Для этого изображение сначала размывается, а затем определяется разность между размытым изображением и оригиналом. На величину этой разницы изменяется яркость оригинала. Таким образом однородные участки изображения не подвергнутся изменениям, а те места картинки, где присутствуют высокочастотные детали, станут контрастнее.

Фильтр реализуется классом `CSharp` (листинг 11.34).

Листинг 11.34. Интерфейс класса CSharp. Файл Filter.h

```
class CSharp: public CMatrixFilter
{
public:
   CSharp();
   BOOL TransformPix(LONG x, LONG y);
};
```

В классе `CSharp` переопределен метод `TransformPix()`, реализация метода приведена в листинге 11.35.

Листинг 11.35. Методы класса CSharp. Файл Filter.cpp

```
CSharp::CSharp()
{
   m_pMatrix=BlurMatrix;
   m_rangX=5;
   m_rangY=5;
};
// Коэффициент увеличения резкости,
// его вполне можно задавать в диалоге
#define SHARP_COEFF 3
BOOL CSharp::TransformPix(LONG x, LONG y)
{
   // Размыли пиксел
```

```
    if(!CMatrixFilter::TransformPix(x, y))
        return FALSE;
    BYTE *pDPix=NULL, *pSPix=NULL;

    pSPix=m_pSourceBM->GetPixPtr(x,y);
    pDPix=m_pDestBM->GetPixPtr(x, y);
    int d=0;
    for(int c=0; c<3; c++)
    {
        // Нашли разницу
        d=*(pSPix+c)-*(pDPix+c);
        // Усилили разницу
        d*=SHARP_COEFF;
        // Присвоили пикселу новое значение
        if(*(pDPix+c)+d <0)
            *(pDPix+c)=0;
        else
            if(*(pDPix+c)+d > 255)
                *(pDPix+c)=255;
        else
                *(pDPix+c)+=d;
    }
    return TRUE;
}
```

Результат применения фильтра показан на рис. 11.31.

Рис. 11.31. Половина изображения обработана фильтром "Четкость"

11.6.12. Фильтр "Удаление шума"

В графических редакторах часто встречаются фильтры, предназначеные для очистки изображения от различного вида шумов. Задача этих фильтров в том, чтобы каким-то образом восстановить исходную "картину мира". Методы, заложенные внутри этих фильтров, могут существенно различаться, но наверняка все включают в себя два этапа. Сначала определяется "шумовой" пиксел (как правило, в выявлении шума и состоит основная сложность). Затем значение шумового пиксела заменяется на новое, как правило, расчитанное из окружающих пикселов значение. Простейшим примером такого фильтра является фильтр "Размытие". Критерий определения шума в нем предельно категоричен — все пикселы считаются шумовыми, а способ замены тоже не отличается хитростью — используется среднее арифметическое значение. Однако в ряде случаев применение такого фильтра оказывает весьма положительный эффект.

Рассмотрим далее реализацию чуть более сложного фильтра[1], будем называть его *энтропийным*. Идея этого фильтра заключается в следующем: рассматривается группа пикселов, например 5×5 элементов; центральный пиксел этой матрицы является тестируемым. В ходе проверки рассчитывается отклонение яркости пиксела от среднего значения яркости и оценивается влияние этого отклонения на энтропию рассматриваемого фрагмента изображения. Если фильтр решает, что такого пиксела быть не должно, его "шумное" значение заменяется новым, посчитанным на основе окружающих пикселов.

Критерий определения шума в данном методе состоит в следующем:

1. Рассматривается n пикселов, попавших в рассматриваемую область. Находится сумма отклонений яркостей пикселов от среднего значения.

$$S = \sum_{i=0}^{n-1} \Delta_i , \tag{11.1}$$

где $\Delta_i = \left| \bar{b} - b_i \right|$; b_i — значение пиксела i; $\bar{b} = \sum_{i=0}^{n-1} b_i \Big/ n$ — среднее значение яркости.

2. Определяется относительный вклад отклонения Δ_k тестируемого пиксела в значение S:

$$p_k = \frac{\Delta_k}{S}, \tag{11.2}$$

[1] Описание этого и некоторых других методов удаления шума можно найти по адресу: **http://ise0.stanford.edu/class/ee368a_proj00/project10/EE368.htm**.

где k — номер тестируемого пиксела.

3. Очевидно, что $\sum_{i=0}^{n-1} p_i = 1$. Если в рассматриваемом фрагменте изображения наблюдается более-менее равномерное распределение яркостей пикселов, то и значения p_i будут не сильно отличаться от $1/n$. Шумовой же пиксел, как правило, "бросается в глаза" именно благодаря тому, что его яркость значительно отличается от средней яркости окружающих его пикселов. Величина Δ_k такого пиксела больше, чем у остальных пикселов, а значит, и его вклад в значение, рассчитанное по формуле (11.1), будет больше, и в результате значение p_k будет превышать $1/n$. Это и является критерием шумового пиксела:

$$\text{если } p_k > \frac{1}{n}\text{, значит пиксел } k \text{ является шумом.} \qquad (11.3)$$

Можно привести и более "наукообразное" обоснование такого критерия. Свойства значений p_i и способ их вычисления подталкивают назвать эту величину вероятностью. В теории информации для оценки количества информации используется энтропийная мера, которая является функцией вероятности и принимает свое значение при равномерном распределении, т. е. в нашем случае, когда $p_i = \frac{1}{n}$, $i = 0, n-1$. Если же какое-то из значений p_k превышает $1/n$, энтропия уменьшается. Следовательно, такой пиксел можно признать шумом, нарушающим плавное течение потока информации, изливаемого изображением на наблюдателя.

Практика использования данного критерия показывает, что целесообразно модифицировать условие (11.3) к следующему виду:

$$p_k > c\frac{1}{n}, \qquad (11.4)$$

где c — коэффициент (назовем его *пороговым*), влияющий на чувствительность критерия.

При $c = 1$ условие (11.4) превращается в (11.3). Опыт показывает, что наилучшие результаты достигаются при значениях $c = 1{,}5...2{,}0$. Впрочем, вы сами сможете поэкспериментировать со значением этого коэффициента.

После того как нарушитель спокойствия выявлен, остается решить, что с ним делать. Здесь возможны несколько вариантов:

1. Заменить шумовой пиксел средним значением $\bar{b}$.
2. Заменить шумовой пиксел средним значением, посчитанным с учетом значений всех пикселов за исключением самого нарушителя.
3. Заменить шумовой пиксел средним значением, посчитанным с учетом значений всех пикселов, не удовлетворяющих критерию отбора шу-

ма. Здесь предполагается, что в рассматриваемый фрагмент может попасть более одного пиксела, удовлетворяющих критерию "шум", и при расчете нового значения их не следует учитывать.

Могут, наверно, быть и другие варианты расчета "замены". В рассмотренной далее реализации мы используем подход номер 3.

Наш фильтр работает с группой пикселов, что весьма похоже на матричное преобразование, поэтому реализующий его класс (назовем его `CDeNoise`) сделаем наследником класса `CMatrixFilter` (листинг 11.36).

Листинг 11.36. Интерфейс класса `CDeNoise`. Файл Filter.h

```
class CDeNoise: public CMatrixFilter
{

public:
   double m_dK;
   int m_nWhatToDo;

   CDeNoise();
   BOOL TransformPix(LONG x, LONG y);
};
```

Однако действия, выполняемые фильтром, отличаются от реализованных в методе `CMatrixFilter::TransformPix()`, поэтому его придется переопределить (листинг 11.37).

Листинг 11.37. Методы класса `CDeNoise`. Файл Filter.cpp

```
//Brightness
BYTE Y(PBYTE pPix);

CDeNoise::CDeNoise()
{
   m_pMatrix=NULL;
   m_rangX=5;
   m_rangY=5;

   m_dK=2,0;
   m_nWhatToDo=0;
};
```

```
BOOL CDeNoise::TransformPix(LONG x, LONG y)
{

   BYTE *pDPix=NULL, *pSPix=NULL;
   // Источник и приемник необходимы
   if(m_pSourceBM==NULL || m_pDestBM==NULL)
      return FALSE;

   // Определяем зону перекрытия изображения и
   // матрицы преобразования. Это требуется для
   // обработки пикселов, находящихся на границах
   // изображения
   int x_start=0;
   int dx=m_rangX/2, dy=m_rangY/2;

   if(x-dx<0) x_start=dx-x;

   int y_start=0;
   if(y-dy<0) y_start=dy-y;

   int x_finish=m_rangX;
   if(x+dx>m_pSourceBM->GetBMWidth())
      x_finish-=(x+dx-m_pSourceBM->GetBMWidth());

   int y_finish=m_rangY;
   if(y+dy>m_pSourceBM->GetBMHeight() )
      y_finish-=(y+dy-m_pSourceBM->GetBMHeight());

   // Находим среднее значение яркости
   int mx=0, my=0;
   int avgY=0, count=0;

   for(my=y_start; my<y_finish; my++)
      for(mx=x_start; mx<x_finish; mx++)
      {
         if((pSPix=m_pSourceBM->GetPixPtr(x+(mx-dx),
               y+(my-dy)))!=NULL)
         {
```

```
            avgY+=Y(pSPix);
            count++;
         }
      }

// Находим сумму отклонений от среднего
int sumVar=0;
for(my=y_start; my<y_finish; my++)
   for(mx=x_start; mx<x_finish; mx++)
   {
      if((pSPix=m_pSourceBM->GetPixPtr(x+(mx-dx),
            y+(my-dy)))!=NULL)
      {
         sumVar+=abs(avgY-count*Y(pSPix));
      }
   }

// Адрес пиксела в изображении-источнике
pSPix=m_pSourceBM->GetPixPtr(x, y);
double Pi= sumVar>0?fabs(avgY-count*Y(pSPix))/sumVar:0;

// Адрес пиксела в изображении-приемнике
pDPix=m_pDestBM->GetPixPtr(x, y);
BYTE NewValue=255;
int NewBGR[3], count2=0;

switch(m_nWhatToDo)
{
   case 0: //Удаление шума
      if(Pi> m_dK/count) // шум
      {
         NewBGR[0]=NewBGR[1]=NewBGR[2]=0;
         count2=0;
         // Находим суммарное значение "не шумовых" пикселов
         for(my=y_start; my<y_finish; my++)
         for(mx=x_start; mx<x_finish; mx++)
         {
       if((pSPix=m_pSourceBM->GetPixPtr(x+(mx-dx),y+(my-dy)))!=NULL &&
         ((sumVar>0?fabs(avgY-count*Y(pSPix))/sumVar:0)<= m_dK/count))
```

```
                {
                   for(int c=0; c<3; c++)   NewBGR[c]+=*(pSPix+c);
                   count2++;
                }
             }
          // Заменяем шум новым значением
          for(int c=0; c<3; c++)   *(pDPix+c)=NewBGR[c]/count2;
          }
          else // не шум
                //  Просто копируем значение пиксела
                for(int c=0; c<3; c++)   *(pDPix+c)=*(pSPix+c);
   break;
   case 1:   // Выделение шума
      if(Pi> m_dK*1.0/count) // noise pixel - select
         NewValue=128;
      for(int c=0; c<3; c++) *(pDPix+c)=NewValue;
   break;
   }

   return TRUE;
}

//Расчет яркости пиксела
BYTE Y(PBYTE pPix)
{
   return( (BYTE)(0.11*(*pPix) + 0.59*(*(pPix+1)) + 0.3*(*(pPix+2))) );
}
```

Реализация метода `CDeNoise::TransformPix()` позволяет, в зависимости от значения переменной `m_nWhatToDo`, либо выполнить замену шумового пиксела, либо показать его серым цветом на белом фоне. Последнее может пригодиться при определении оптимального значения порогового коэффициента в условии (11.4).

Остается только поэкспериментировать с удалением шума. Для этого надо сначала получить зашумленную картинку. Не станем слишком хитрить, а возьмем изображение автомобиля, показанное на рис. 11.10, и в программе Microsoft Paint с помощью инструмента Airbrush нанесем на него некоторое количество белых и черных точек (рис. 11.32). Это приведет к появлению в изображении импульсного шума — значительных скачков яркости в отдельных пикселах.

Рис. 11.32. Зашумленное изображение

Теперь попробуем применить фильтр "Размытие" для удаления шума (рис. 11.33) — да, похоже, это не тот случай, когда "Размытие" хорошо справляется с удалением шума. Дело, видимо, в том, что шумовые пикселы также вносят свой вклад в среднее значение.

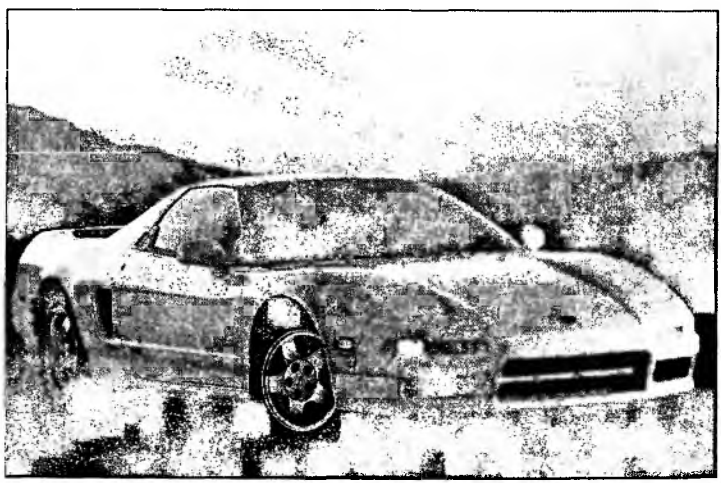


Рис. 11.33. Попытка удалить шум фильтром "Размытие"

Испытаем далее фильтр "Удаление шума". На рис. 11.34 показано изображение, к половине которого применили этот фильтр со значением $c = 2{,}0$.

Как видно, фильтр справился с задачей довольно неплохо — основная масса шума удалена. Некоторые нарекания вызывает то, что некоторые высокочастотные элементы изображения также подверглись изменениям. Напри-

мер, на тонкой полосе, проходящей через капот автомобиля, появились разрывы, то же самое можно заметить на нижнем переднем спойлере. Это недостаток метода. Регулировать появление побочных эффектов можно, изменяя значение порогового коэффициента.

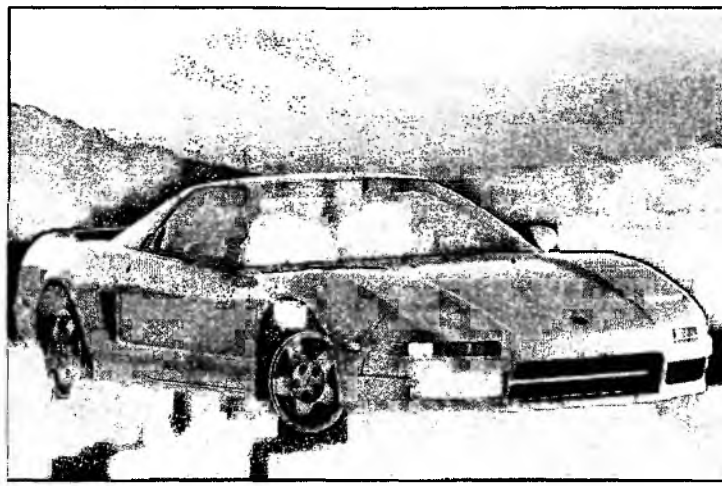


Рис. 11.34. Действие энтропийного фильтра удаления шума

Хотя приведенный пример является весьма надуманным, применение рассмотренного фильтра может оказаться полезным и в некоторых реальных ситуациях. Например, при удалении шума, характерного для изображений, полученных с цифровых камер низкого разрешения.

Похожих результатов фильтрации шума на данном изображении можно достичь и с помощью менее сложного фильтра, называемого *медианным*. Этот метод весьма популярен в обработке цифровых изображений и его реализации можно встретить во многих программах.

Медианная фильтрация предложена Дж. Тьюки в 1971 году для анализа экономических процессов. Метод медианной фильтрации является эвристическим и не является математическим решением строго сформулированной задачи. Однако во многих случаях подобный способ фильтрации весьма эффективен.

Метод медианной фильтрации весьма похож на рассмотренный ранее фильтр "Размытие". Так же, как и в фильтре "Размытие", рассматривается область пикселов, находится среднее значение, и значение центрального пиксела заменяется на найденное среднее. Отличие состоит в том, как находится среднее значение. Если в фильтре "Размытие" среднее значение определяется как среднее арифметическое яркости всех пикселов, то в медианном фильтре просто помещают значения яркости всех пикселов рассматриваемого фрагмента изображения в линейный массив или список, сортируют

получившийся набор и берут значение, оказавшееся в середине отсортированной последовательности. Возможны различные варианты зоны рассматриваемых пикселов, например в виде квадрата или креста. Процесс медианной фильтрации изображен на рис. 11.35. Тестируемый пиксел замещается медианным значением (взятым из центра) упорядоченной последовательности, поэтому если он содержал пиковое значение, то оно заменится на нечто более среднее. При этом величина пикового значения не учитывается. На рис. 11.36 показано зашумленное изображение (рис. 11.32), половина которого обработана с помощью медианного фильтра в Adobe Photoshop.

Рис. 11.36. Действие медианного фильтра удаления шума

Достоинство метода в его простоте и эффективном удалении импульсного шума. Недостаток — размытие изображения, что в некоторых случаях может оказаться неприемлемым. Как вы, наверно, заметили, в медианном фильтре отсутствует этап выявления шумового элемента, поэтому преобразованию подвергаются все пикселы и возникает размытие. Медианный метод может быть использован в сочетании с энтропийным методом определения шумового пиксела. В этом методе можно "порулить" размером и формой зоны охвата пикселов.

Кстати сказать, в компьютерной графике популярна и задача генерации шума, интересную статью на эту тему можно найти по адресу: **http://freespace.virgin.net/hugo.elias/models/m_perlin.htm**.

Для установки фильтра `CDeNoise` в меню программы добавили соответствующую команду, обработчик которой поместили в класс документа (листинг 11.40) — метод `CBMDoc::OnEditDenoise()`, а для настройки параметров фильтра специальный диалог `CDeNoiseDlg`, шаблон которого показан на рис. 11.37.

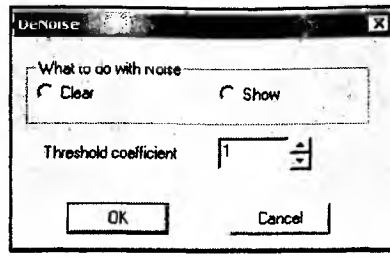


Рис. 11.37. Шаблон диалога настройки фильтра удаления шума

Интерфейс и реализация диалога `CDeNoiseDlg` приведены *в разд. 11.8*, см. листинги 11.45 и 11.46, соответственно.

11.6.13. Применение фильтров

Теперь, когда мы так много узнали о фильтрах и их внутреннем устройстве, можем использовать их со знанием дела. Например, прежде чем искать контуры рисунка, можно сначала выполнить его размытие — это уберет слабый "шум", незаметный при нормальном просмотре, но проявляющийся при выделении контуров; найденные контуры можно усилить фильтром "Четкость", затем использовать фильтр "Инверсия" и получить окончательный результат, такой, как показано на рис. 11.38.

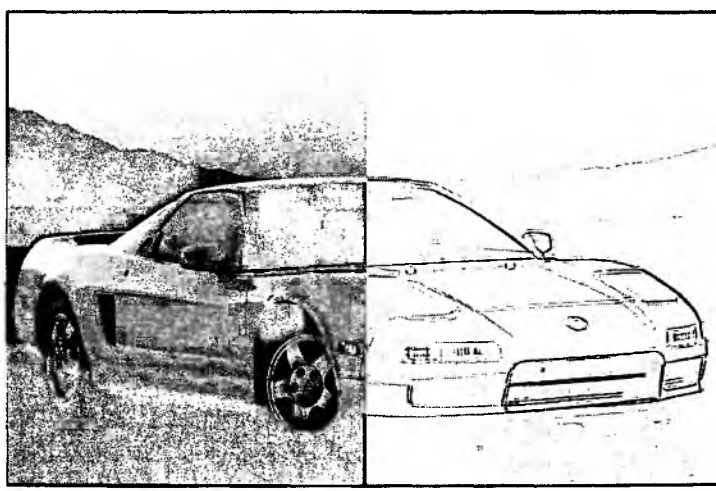


Рис. 11.38. Половина изображения последовательно обработана несколькими фильтрами

Таким образом, можно превратить отличную фотографию в неповторимый рисунок "углем". Приведенный пример, конечно, не ограничивает сферу применения рассмотренных фильтров.

11.7. Вывод изображений на печать

Вполне возможно, что у пользователя возникнет желание увековечить свое творение в твердой копии, т. е. распечатать. Однако выполнив команду **File | Print Preview**, пользователь увидит, что вся красота сжалась до размеров почтовой марки (рис. 11.39).

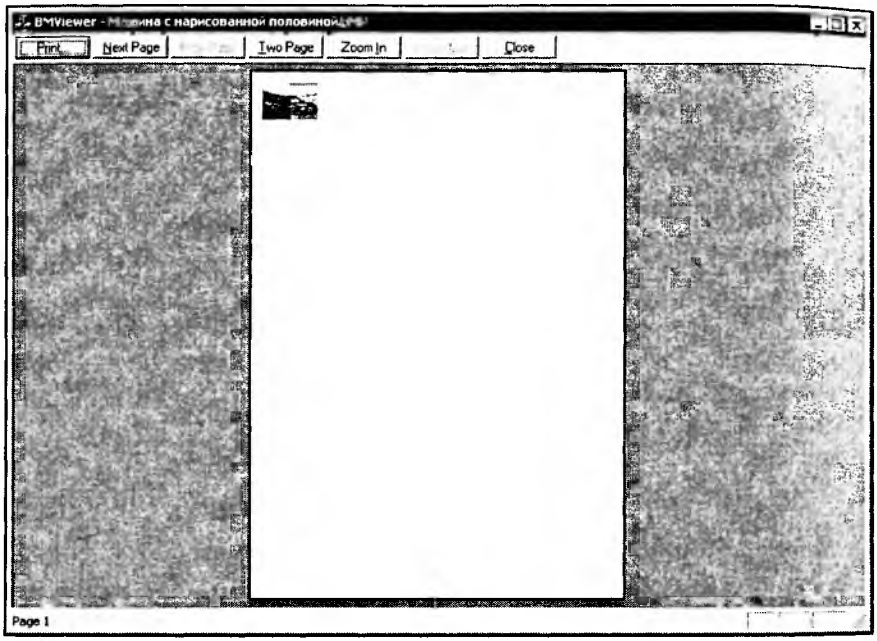


Рис. 11.39. Красота сжалась до размеров почтовой марки

Но мы-то знаем, в чем причина, мы это уже "проходили" в *разд. 4.3.1*. Там проблема была решена путем установки другого режима отображения. В программе же BMViewer облики работают в режиме отображения `MM_TEXT` (см. листинг 11.5, метод `CBMView::OnUpdate()`). Это нас вполне устраивает, так как один пиксел экрана соответствует одному пикселу изображения. Решить же проблему можно путем масштабирования размеров картинки при выводе ее на печать. Для этого переопределим в классе облика `CBMView` виртуальный метод `CView::OnPrint()`. Сделать это можно с помощью ClassWizard. Новый метод приведен в листинге 11.38.

Листинг 11.38. Метод CBMView::OnPrint(). Файл BMView.cpp

```
void CBMView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
```

```
// Разрешение устройства
   int DXRes=pDC->GetDeviceCaps(LOGPIXELSX);
   int DYRes=pDC->GetDeviceCaps(LOGPIXELSY);

   CBMDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);
   CRaster* pCurPic=pDoc->GetCurrentBMPtr();
   if(pCurPic==NULL) return;

   LONG  height, width;
   double resx, resy;
   // Перерасчет размера изображения под разрешение принтера
   width=pCurPic->GetBMWidth();
   height=pCurPic->GetBMHeight();

   resx=((double)pCurPic->GetBMInfoPtr()->bmiHeader.biXPelsPerMeter);
   resy=((double)pCurPic->GetBMInfoPtr()->bmiHeader.biYPelsPerMeter);
   // Переводим в dpi
   resx*=(25.4/1000);
   resy*=(25.4/1000);
   // Приведем к разумным пределам
   if(resx<=0 || resx> 3000) resx=72;
   if(resy<=0 || resy> 3000) resy=72;

   width=(int)(((double)width)*DXRes/resx+0.5);
   height=(int)(((double)height)*DYRes/resy+0.5);

   // Выводим изображение, соответственно новым размерам
   pCurPic->DrawBitmap( pDC, 0, 0, width, height);
};
```

В методе `OnPrint()` сначала получаем разрешение печатающего устройства с помощью метода `CDC::GetDeviceCaps()`. Этот метод возвращает разрешение в точках на дюйм (dpi). Растровая картинка тоже имеет характеристику "разрешение", которая записана в заголовке растра `BITMAPINFOHEADER` в полях `biXPelsPerMeter` и `biYPelsPerMeter`. Названия этих полей говорят о том, что разрешение в них записано в точках на метр. Это не может испугать нас, так как мы знаем, что в дюйме ровно 25,4 мм. Поэтому далее разрешение картинки приводится к единицам измерения "точек на дюйм".

К сожалению, далеко не все программы, создающие растровые изображения, заполняют поля `biXPelsPerMeter` и `biYPelsPerMeter` значениями. Бывает, что программы пишут в такое поле значение 0, как, например, Painter 4.2 (правильнее было бы записать туда разрешение экрана), а бывает, что оставляют его и вовсе неинициализированным, и там содержится какое-нибудь случайное значение. Эта ситуация проверяется на следующем этапе. Затем размеры картинки масштабируются пропорционально соотношению разрешений картинки и принтера. И наконец, используются замечательные возможности метода `CRaster::OnDraw()` по выводу изображения на контекст с масштабированием до заданного размера.

В принципе, в метод `OnPrint()` можно вставить вызов диалога, в котором пользователь мог бы задать нужные ему размеры отпечатка.

Результат работы программы с новым методом `OnPrint()` показан на рис. 11.40. Предварительно с помощью стандартного диалога, вызываемого командой **File | Print Setup**, установлена альбомная (Landscape) ориентация листа.

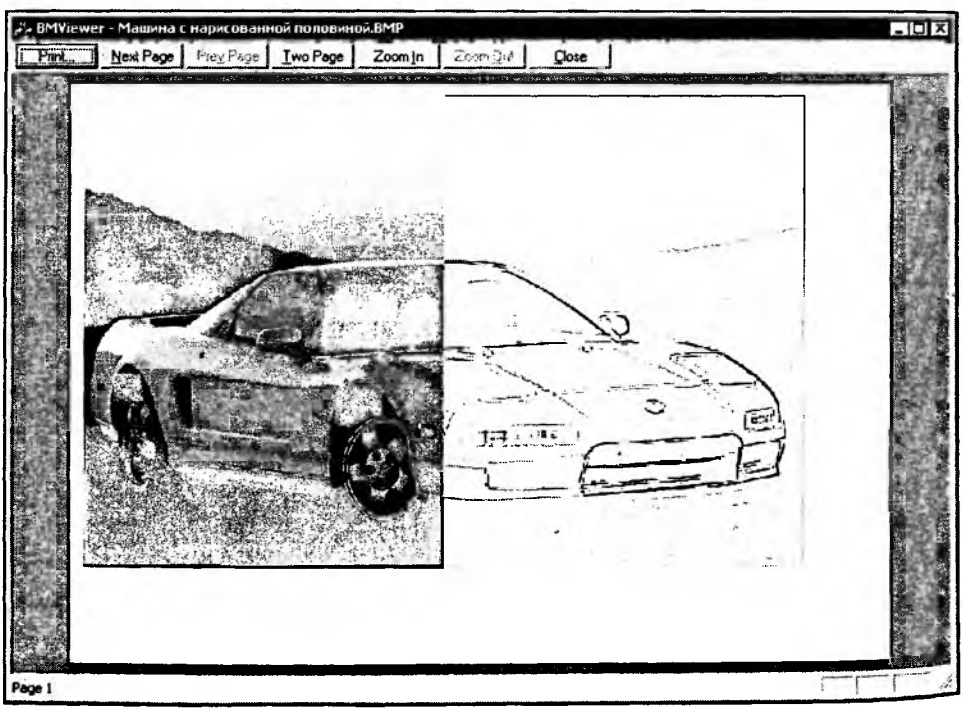


Рис. 11.40. Предварительный просмотр отпечатка

11.8. Листинг программы

Реализация основных методов была приведена выше при описании процесса создания программы. Интерфейс и реализация класса CRaster приведены в *разд. 11.2*. Основные методы класса CBMView полностью описаны в *разд. 11.5* и *11.7*. Классы фильтров описаны в посвященных им разделах. Приведем здесь лишь полные листинги (листинги 11.39 и 11.40) класса документа CBMDoc (обратите внимание, как происходит работа с буферами изображений) и классов диалогов, которые не были рассмотрены ранее (листинги 11.41—11.46). Кроме того, полный текст программы содержится на компакт-диске в каталоге Sources\BMViewer.

Листинг 11.39. Интерфейс класса CBMDoc. Файл BMDoc.h

```
// BMDoc.h : interface of the CBMDoc class

#include <afxmt.h>
#include "Raster.h"
#include "Filter.h"

#define UM_STARTTRANSFORM   WM_USER+0x8000
#define UM_ENDOFTRANSFORM   UM_STARTTRANSFORM+1

class CBMDoc : public CDocument
{
protected: // create from serialization only
    CBMDoc();
    DECLARE_DYNCREATE(CBMDoc)

// Attributes
public:
    // Флаги
    BOOL    m_bEditable;     // можем ли редактировать данные
    BOOL    m_bEditHalf;    // редактировать половину изображения
    CEvent    m_EventDoTransform; // событие - выполняется преобразование
    LONG    m_lExecutedPercent;    // процент выполнения
    // Данные
    CRaster m_BM[2];     // два буфера для изображений
   CRaster *m_pCurBM;    // указатель на активный буфер
```

```
    //Фильтры
    CFilter    *m_pCurFilter;
    CBrightCont m_BrightContFilter;
    CHistogram   m_HistogramFilter;
    CInvertColors m_InvertColorsFilter;
    CEmboss m_EmbossFilter;

    CBlur   m_BlurFilter;
    CContour m_ContourFilter;
    CSharp   m_SharpFilter;
    CDeNoise   m_DenoiseFilter;

// Operations
public:
    CSize GetCurrentBMSize();
    //Возвращает номер активного буфера
    int   GetNCurrentBM();
    //Возвращает указатель на текущую картинку
    CRaster* GetCurrentBMPtr();
    //Возвращает указатель на буфер
    CRaster* GetBufferBMPtr();
    //Поменять текущее изображение и буфер местами
    void SwapBM();
    //Нарисовать текущее изображение
    void DrawCurrent();
    // Создает буфер заданного размера
    // (при вызове без аргументов размер равен текущей картинке),
    // совместимый с текущей картинкой
    BOOL CreateCompatibleBuffer(LONG width=0, LONG height=0);
    //Запускает поток преобразования
    void Transform();

protected:
    //Функция потока преобразования
    static UINT ThreadProc(LPVOID pv);
    //Цикл преобразования с использованием активного фильтра
    void TransformLoop();
    //Посылает сообщение всем окнам
    void InformAllViews(UINT msg, WPARAM wPar=0, LPARAM lPar=0);
```

```
public:
// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CBMDoc)
   public:
   virtual BOOL OnNewDocument();
   virtual void Serialize(CArchive& ar);
   virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
   virtual BOOL OnSaveDocument(LPCTSTR lpszPathName);
   //}}AFX_VIRTUAL

// Implementation
public:
   virtual ~CBMDoc();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:
// Generated message map functions
protected:
   //{{AFX_MSG(CBMDoc)
   afx_msg void OnEditHistogram();
   afx_msg void OnEditUndo();
   afx_msg void OnEditBrightnessandcontrast();
   afx_msg void OnEditInvertcolors();
   afx_msg void OnEditBlur();
   afx_msg void OnEditSharp();
   afx_msg void OnEditContour();
   afx_msg void OnEditHalf();
   afx_msg void OnUpdateEditHalf(CCmdUI* pCmdUI);
   afx_msg void OnEditEmboss();
   afx_msg void OnUpdateEditBlur(CCmdUI* pCmdUI);
   afx_msg void OnUpdateEditHistogram(CCmdUI* pCmdUI);
   afx_msg void OnUpdateEditEmboss(CCmdUI* pCmdUI);
   afx_msg void OnUpdateEditSharp(CCmdUI* pCmdUI);
   afx_msg void OnUpdateEditUndo(CCmdUI* pCmdUI);
```

```
    afx_msg void OnUpdateEditContour(CCmdUI* pCmdUI);
    afx_msg void OnUpdateEditBrightnessandcontrast(CCmdUI* pCmdUI);
    afx_msg void OnUpdateEditInvertcolors(CCmdUI* pCmdUI);
    afx_msg void OnEditDenoise();
    afx_msg void OnEditStop();
    afx_msg void OnUpdateEditStop(CCmdUI* pCmdUI);
    afx_msg void OnUpdateEditDenoise(CCmdUI* pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Листинг 11.40. Реализация методов класса CBMDoc. Файл BMDoc.cpp

```
// BMDoc.cpp : implementation of the CBMDoc class

#include "stdafx.h"
#include "BMViewer.h"

#include "BMDoc.h"
#include "HistDlg.h"
#include "BrightContDlg.h"
#include "DeNoiseDlg.h"
#include "BMView.h"


#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CBMDoc

IMPLEMENT_DYNCREATE(CBMDoc, CDocument)

BEGIN_MESSAGE_MAP(CBMDoc, CDocument)
    //{{AFX_MSG_MAP(CBMDoc)
```

```
    ON_COMMAND(ID_EDIT_HISTOGRAM, OnEditHistogram)
    ON_COMMAND(ID_EDIT_UNDO, OnEditUndo)
    ON_COMMAND(ID_EDIT_BRIGHTNESSANDCONTRAST, OnEditBrightnessandcontrast)
    ON_COMMAND(ID_EDIT_INVERTCOLORS, OnEditInvertcolors)
    ON_COMMAND(ID_EDIT_BLUR, OnEditBlur)
    ON_COMMAND(ID_EDIT_SHARP, OnEditSharp)
    ON_COMMAND(ID_EDIT_CONTOUR, OnEditContour)
    ON_COMMAND(ID_EDIT_HALF, OnEditHalf)
    ON_UPDATE_COMMAND_UI(ID_EDIT_HALF, OnUpdateEditHalf)
    ON_COMMAND(ID_EDIT_EMBOSS, OnEditEmboss)
    ON_UPDATE_COMMAND_UI(ID_EDIT_BLUR, OnUpdateEditBlur)
    ON_UPDATE_COMMAND_UI(ID_EDIT_HISTOGRAM, OnUpdateEditHistogram)
    ON_UPDATE_COMMAND_UI(ID_EDIT_EMBOSS, OnUpdateEditEmboss)
    ON_UPDATE_COMMAND_UI(ID_EDIT_SHARP, OnUpdateEditSharp)
    ON_UPDATE_COMMAND_UI(ID_EDIT_UNDO, OnUpdateEditUndo)
    ON_UPDATE_COMMAND_UI(ID_EDIT_CONTOUR, OnUpdateEditContour)
    ON_UPDATE_COMMAND_UI(ID_EDIT_BRIGHTNESSANDCONTRAST,
    OnUpdateEditBrightnessandcontrast)
    ON_UPDATE_COMMAND_UI(ID_EDIT_INVERTCOLORS, OnUpdateEditInvertcolors)
    ON_COMMAND(ID_EDIT_DENOISE, OnEditDenoise)
    ON_COMMAND(ID_EDIT_STOP, OnEditStop)
    ON_UPDATE_COMMAND_UI(ID_EDIT_STOP, OnUpdateEditStop)
    ON_UPDATE_COMMAND_UI(ID_EDIT_DENOISE, OnUpdateEditDenoise)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CBMDoc construction/destruction

CBMDoc::CBMDoc():m_EventDoTransform(FALSE, TRUE)
{
    // TODO: add one-time construction code here
    m_pCurBM=NULL;
    m_bEditable=FALSE;
    m_bEditHalf=FALSE;
    m_pCurFilter=NULL;

    //m_pEventDoTransform=new CEvent(FALSE, TRUE);
}
```

```
CBMDoc::~CBMDoc()
{
//   if(m_pEventDoTransform) delete m_pEventDoTransform;
}

BOOL CBMDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   // TODO: add reinitialization code here
   // (SDI documents will reuse this document)
   return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CBMDoc serialization

void CBMDoc::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
   {
      // TODO: add storing code here
   }
   else
   {
      // TODO: add loading code here
   }
}

/////////////////////////////////////////////////////////////////////////////
// CBMDoc diagnostics

#ifdef _DEBUG
void CBMDoc::AssertValid() const
{
   CDocument::AssertValid();
}
```

```
void CBMDoc::Dump(CDumpContext& dc) const
{
   CDocument::Dump(dc);
}
#endif //_DEBUG
////////////////////////////////////////////////////////////////////////////
// CBMDoc operations

CSize CBMDoc::GetCurrentBMSize()
{
   CSize sz;
   if(m_pCurBM!=NULL)
   {
      sz.cx=m_pCurBM->GetBMWidth();
      sz.cy=m_pCurBM->GetBMHeight();
   }
   return sz;
};
int   CBMDoc::GetNCurrentBM()
{
   if(m_pCurBM==NULL || m_pCurBM==&m_BM[0]) return 0;
   else return 1;
};

CRaster* CBMDoc::GetCurrentBMPtr()
{
   return m_pCurBM;
};

CRaster* CBMDoc::GetBufferBMPtr()
{
   return &m_BM[1-GetNCurrentBM()];
};

void CBMDoc::SwapBM()
{
   m_pCurBM=GetBufferBMPtr();
};
```

```
BOOL CBMDoc::CreateCompatibleBuffer(LONG width/*=0*/, LONG height/*=0*/)
{
   if(!m_pCurBM) return FALSE;
   CRaster *pBuff=GetBufferBMPtr();
   if(!pBuff) return FALSE;
   return pBuff->CreateCompatible(m_pCurBM->GetBMInfoPtr(), width, height);
};

void CBMDoc::Transform()
{
   AfxBeginThread(ThreadProc, this);
}

UINT CBMDoc::ThreadProc(LPVOID pv)
{
   CBMDoc* pDoc = (CBMDoc*)pv;

    if (pDoc == NULL ||
        !pDoc->IsKindOf(RUNTIME_CLASS(CBMDoc)))
    return 1;   // Не правильный параметр

   pDoc->TransformLoop();

   return 0;
};

void CBMDoc::TransformLoop()
{
   if(m_pCurFilter==NULL) return;
   if(!CreateCompatibleBuffer()) return;

   m_EventDoTransform.SetEvent();
   m_bEditable=FALSE;
   InformAllViews(UM_STARTTRANSFORM);

   Craster   *pSBM=GetCurrentBMPtr(),  // источник
         *pDBM=GetBufferBMPtr();       // приемник
   // Установили в фильтр источник и приемник преобразований
```

```
m_pCurFilter->SetBuffers(pSBM, pDBM);

for(LONG y=0; y<pSBM->GetBMHeight(); y++)
{
   // Процент выполнения
   InterlockedExchange(&m_lExecutedPercent,
                       100*y/pSBM->GetBMHeight());
   // Проверим, не решено ли прервать преобразование
   if(!m_EventDoTransform.Lock(0))
   {
      InformAllViews(UM_ENDOFTRANSFORM, FALSE, 0);
      m_bEditable=TRUE;
      return;
   }
   LONG x=0;
   if( m_bEditHalf ) // Преобразовать только половину изображения
   {
      // Первую половину картинки копируем в буфер без преобразования
      x=pSBM->GetBMWidth()/2;
      BYTE *pSPix=NULL,  *pDPix=NULL;
      // Указатели на начало строк
      if(    (pSPix=pSBM->GetPixPtr(0, y))!=NULL &&
          (pDPix=pDBM->GetPixPtr(0, y))!=NULL)
      // ВНИМАНИЕ! Предполагается, что 1 пиксел = 24 бита = 3 байта
      memcpy(pDPix, pSPix, 3*x);
   }
   // Преобразование с использованием текущего фильтра
   for(; x<pSBM->GetBMWidth(); x++)
      m_pCurFilter->TransformPix(x, y);

}
m_EventDoTransform.ResetEvent();
m_bEditable=TRUE;

SwapBM();   // сделать буфер текущим изображением
SetModifiedFlag(); // флаг "данные изменились"

InformAllViews(UM_ENDOFTRANSFORM, TRUE, 0);
return;
```

```
};

void CBMDoc::InformAllViews(UINT msg, WPARAM wPar/*=0*/,
                                      LPARAM lPar/*=0*/)
{
   POSITION pos = GetFirstViewPosition();
   while (pos != NULL)
   {
      CBMView* pView = (CBMView*)GetNextView(pos);
      PostMessage( pView->m_hWnd, msg, wPar, lPar);
   }
}

void CBMDoc::DrawCurrent()
{
   // вывести изображение на виртуальный экран всех обликов
   POSITION pos = GetFirstViewPosition();
   while (pos != NULL)
   {
      CBMView* pView = (CBMView*)GetNextView(pos);
      pView->UpdateVirtualScreen();
   }
}

/////////////////////////////////////////////////////////////////////////////
// CBMDoc commands

BOOL CBMDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
   if (!CDocument::OnOpenDocument(lpszPathName))
      return FALSE;

   // Загружаем в первый буфер
   if(m_BM[0].LoadBMP(lpszPathName))
   {
      m_pCurBM=&m_BM[0];
      //Умеем редактировать только RGB888 данные
      if(m_pCurBM->GetBMInfoPtr()->bmiHeader.biBitCount!=24)
         m_bEditable=FALSE;
```

```
      else
         m_bEditable=TRUE;

      return TRUE;
   }

   return FALSE;
}

BOOL CBMDoc::OnSaveDocument(LPCTSTR lpszPathName)
{
   if( m_pCurBM!=NULL && m_pCurBM->SaveBMP(lpszPathName))
   {
      SetModifiedFlag(FALSE);
      return TRUE;
   }
   else
      return FALSE;
}

void CBMDoc::OnEditUndo()
{
   SwapBM();           // сделать буфер текущим изображением
   DrawCurrent();      // нарисовать
   SetModifiedFlag(); // флаг "данные изменились"
}

void CBMDoc::OnEditHistogram()
{
   const int Range=256;
   DWORD Hist[Range]; // гистограмма из Range градаций яркости

   // Запросим гистограмму у текущего изображения
   if(m_pCurBM==NULL || !m_pCurBM->GetHistogram(Hist, Range))
      return;
   // Создаем объект-диалог
   CHistDlg HDlg;
   // Передадим гистограмму в диалог
   HDlg.SetData(Hist, Range);
```

```
    // Покажем гистограмму
    if(HDlg.DoModal()==IDCANCEL) return;
    // Требуется выполнить коррекцию контрастности
    if(HDlg.m_iOffset_b !=0 || HDlg.m_iOffset_t!=NULL)
    {
       // Настраиваем фильтр гистограммы
       m_HistogramFilter.Init(HDlg.m_iOffset_b, HDlg.m_iOffset_t);
       // Делаем фильтр активным
       m_pCurFilter=&m_HistogramFilter;
       // Выполняем преобразование
       Transform();
    }
}

void CBMDoc::OnEditBrightnessandcontrast()
{

    CBrightContDlg BCDlg;
    if(BCDlg.DoModal()==IDCANCEL) return;

    if(BCDlg.m_iBrightnessOffset!=0 || BCDlg.m_iContrastOffset!=0)
    {
       m_BrightContFilter.Init(BCDlg.m_iBrightnessOffset,
                               BCDlg.m_iContrastOffset);
       m_pCurFilter=&m_BrightContFilter;
       Transform();
    }
}

void CBMDoc::OnEditInvertcolors()
{
    m_pCurFilter=&m_InvertColorsFilter;
    Transform();
}

void CBMDoc::OnEditEmboss()
{
    m_pCurFilter=&m_EmbossFilter;
    Transform();
```

```
}
void CBMDoc::OnEditBlur()
{
   m_pCurFilter=&m_BlurFilter;
   Transform();
}

void CBMDoc::OnEditContour()
{
   m_pCurFilter=&m_ContourFilter;
   Transform();
}

void CBMDoc::OnEditSharp()
{
   m_pCurFilter=&m_SharpFilter;
   Transform();
}

void CBMDoc::OnEditDenoise()
{
   static double k=m_DenoiseFilter.m_dK;
   static int what_to_do=0;
   CDeNoiseDlg DNDlg;
   DNDlg.m_dK=k;
   DNDlg.m_nWhatToDo=what_to_do;

   if(DNDlg.DoModal()!=IDOK) return;
   k=m_DenoiseFilter.m_dK=DNDlg.m_dK;
   what_to_do=m_DenoiseFilter.m_nWhatToDo=DNDlg.m_nWhatToDo;

   m_pCurFilter=&m_DenoiseFilter;

   Transform();
}


void CBMDoc::OnEditHalf()
{
```

```
    m_bEditHalf=!m_bEditHalf;
}

void CBMDoc::OnUpdateEditHalf(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_bEditHalf);
    pCmdUI->Enable(m_bEditable);
}

void CBMDoc::OnUpdateEditUndo(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(IsModified());
}

void CBMDoc::OnUpdateEditBlur(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_bEditable);
}

void CBMDoc::OnUpdateEditHistogram(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_bEditable);
}

void CBMDoc::OnUpdateEditEmboss(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_bEditable);
}

void CBMDoc::OnUpdateEditSharp(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_bEditable);
}

void CBMDoc::OnUpdateEditContour(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_bEditable);
}
```

```
void CBMDoc::OnUpdateEditBrightnessandcontrast(CCmdUI* pCmdUI)
{
   pCmdUI->Enable(m_bEditable);
}

void CBMDoc::OnUpdateEditInvertcolors(CCmdUI* pCmdUI)
{
   pCmdUI->Enable(m_bEditable);
}

void CBMDoc::OnUpdateEditDenoise(CCmdUI* pCmdUI)
{
   pCmdUI->Enable(m_bEditable);
}

void CBMDoc::OnEditStop()
{
   m_EventDoTransform.ResetEvent();
}

void CBMDoc::OnUpdateEditStop(CCmdUI* pCmdUI)
{
   pCmdUI->Enable(m_EventDoTransform.Lock(0));
}
```

Листинг 11.41. Интерфейс класса `CHistDlg`. Файл HistDlg.h

```
// HistDlg.h : header file

#include "HistView.h"
/////////////////////////////////////////////////////////////////////////////
// CHistDlg dialog

class CHistDlg : public CDialog
{
// Construction
public:
   CHistDlg(CWnd* pParent = NULL);   // standard constructor
// Data
```

```
    int      m_iOffset_b;   //коррекция яркости снизу
    int      m_iOffset_t;   //коррекция яркости сверху
// Operations
    void SetData(const DWORD *pHist, int Range)
            {m_ctrlHist.SetData(pHist, Range);};
// Dialog Data
    //{{AFX_DATA(CHistDlg)
    enum { IDD = IDD_HIST };
    CHistView     m_ctrlHist;
    CSliderCtrl   m_ctrlOffset_b;
    CSliderCtrl   m_ctrlOffset_t;
    CString       m_strOffset_b;
    CString       m_strOffset_t;
    //}}AFX_DATA

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CHistDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);     // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CHistDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnHScroll(UINT nSBCode, UINT nPos,
                           CScrollBar* pScrollBar);
    virtual void OnOK();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Листинг 11.42. Реализация методов класса `CHistDlg`. Файл HistDlg.cpp

```
// HistDlg.cpp : implementation file

#include "stdafx.h"
```

```
#include "BMViewer.h"
#include "HistDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CHistDlg dialog


CHistDlg::CHistDlg(CWnd* pParent /*=NULL*/)
   : CDialog(CHistDlg::IDD, pParent)
{
   //{{AFX_DATA_INIT(CHistDlg)
   m_strOffset_b = _T("");
   m_strOffset_t = _T("");
   //}}AFX_DATA_INIT

   m_iOffset_b=0;
   m_iOffset_t=0;
}


void CHistDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CHistDlg)
   DDX_Control(pDX, IDC_HIST_VIEW, m_ctrlHist);
   DDX_Control(pDX, IDC_SLIDER_HIST_B, m_ctrlOffset_b);
   DDX_Control(pDX, IDC_SLIDER_HIST_T, m_ctrlOffset_t);
   DDX_Text(pDX, IDC_OFFSET_B_TXT, m_strOffset_b);
   DDX_Text(pDX, IDC_OFFSET_T_TXT, m_strOffset_t);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CHistDlg, CDialog)
```

```
    //{{AFX_MSG_MAP(CHistDlg)
    ON_WM_HSCROLL()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// Operations
////////////////////////////////////////////////////////////////////////////
// CHistDlg message handlers

BOOL CHistDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // Ползунок нижней границы
    m_ctrlOffset_b.SetRange(0, 127);
    // Бегунок в крайнем левом положении
    m_ctrlOffset_b.SetPos(0);
    // Ползунок верхней границы
    m_ctrlOffset_t.SetRange(128, 255);
    // Бегунок в крайнем правом положении
    m_ctrlOffset_t.SetPos(255);
    // Текст
    m_strOffset_b="0";
    m_strOffset_t="0";
    // Цвет гистограммы
    m_ctrlHist.SetColor(RGB(0, 50, 50));

    UpdateData(FALSE);

    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}

void CHistDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    m_strOffset_b.Format("%d", m_ctrlOffset_b.GetPos());
    m_strOffset_t.Format("%d", 255-m_ctrlOffset_t.GetPos());
    UpdateData(FALSE);
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
```

```
}

void CHistDlg::OnOK()
{
   m_iOffset_b=m_ctrlOffset_b.GetPos();
   m_iOffset_t=255-m_ctrlOffset_t.GetPos();
   CDialog::OnOK();
}
```

Листинг 11.43. Интерфейс класса `CBrightContDlg`. Файл BrightContDlg.h

```
// BrightContDlg.h : header file
class CBrightContDlg : public CDialog
{
// Construction
public:
   CBrightContDlg(CWnd* pParent = NULL);   // standard constructor

   int m_iBrightnessOffset;
   int m_iContrastOffset;
// Dialog Data
   //{{AFX_DATA(CBrightContDlg)
   enum { IDD = IDD_BRIGHT_CONT };
   CSliderCtrl   m_ctrlContrast;
   CSliderCtrl   m_ctrlBrightness;
   CString       m_strBrightness;
   CString       m_strContrast;
   //}}AFX_DATA

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CBrightContDlg)
   protected:
   virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
   //}}AFX_VIRTUAL

// Implementation
protected:
```

```
    // Generated message map functions
    //{{AFX_MSG(CBrightContDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar);
    virtual void OnOK();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Листинг 11.44. Реализация методов класса CBrightContDlg. Файл BrightContDlg.cpp

```
// BrightContDlg.cpp : implementation file
#include "stdafx.h"
#include "BMViewer.h"
#include "BrightContDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CBrightContDlg dialog

CBrightContDlg::CBrightContDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CBrightContDlg::IDD, pParent)
{
    m_iBrightnessOffset=0;
    m_iContrastOffset=0;
    //{{AFX_DATA_INIT(CBrightContDlg)
    m_strBrightness = _T("");
    m_strContrast = _T("");
    //}}AFX_DATA_INIT
}
```

```
void CBrightContDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CBrightContDlg)
   DDX_Control(pDX, IDC_SLIDER_CONTRAST, m_ctrlContrast);
   DDX_Control(pDX, IDC_SLIDER_BRIGHTNESS, m_ctrlBrightness);
   DDX_Text(pDX, IDC_STATIC_BRIGHTNESS, m_strBrightness);
   DDX_Text(pDX, IDC_STATIC_CONTRAST, m_strContrast);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CBrightContDlg, CDialog)
   //{{AFX_MSG_MAP(CBrightContDlg)
   ON_WM_HSCROLL()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CBrightContDlg message handlers
#define MAX_CORRECTION_OFFSET 100
BOOL CBrightContDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Инициализация ползунков яркости и контрастности
   m_ctrlBrightness.SetRange(0, MAX_CORRECTION_OFFSET*2);
   m_ctrlBrightness.SetPos(MAX_CORRECTION_OFFSET);
   m_ctrlBrightness.SetTic(0);
   m_ctrlBrightness.SetTic(MAX_CORRECTION_OFFSET);
   m_ctrlBrightness.SetTic(MAX_CORRECTION_OFFSET*2);

   m_ctrlContrast.SetRange(0, MAX_CORRECTION_OFFSET*2);
   m_ctrlContrast.SetPos(MAX_CORRECTION_OFFSET);
   m_ctrlContrast.SetTic(0);
   m_ctrlContrast.SetTic(MAX_CORRECTION_OFFSET);
   m_ctrlContrast.SetTic(MAX_CORRECTION_OFFSET*2);

   m_strBrightness="0";
   m_strContrast="0";
```

```
    UpdateData(FALSE);

    return TRUE;
}

void CBrightContDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
    m_strBrightness.Format("%d", m_ctrlBrightness.GetPos()-
                                 MAX_CORRECTION_OFFSET);
    m_strContrast.Format("%d", m_ctrlContrast.GetPos()-
                               MAX_CORRECTION_OFFSET);
    UpdateData(FALSE);
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}

void CBrightContDlg::OnOK()
{
    // Преобразуем диапазон единиц коррекции к половине диапазона яркости
    m_iBrightnessOffset=(m_ctrlBrightness.GetPos()-MAX_CORRECTION_OFFSET)*
                        127/MAX_CORRECTION_OFFSET;
    m_iContrastOffset=(m_ctrlContrast.GetPos()-MAX_CORRECTION_OFFSET)*
                      127/MAX_CORRECTION_OFFSET;
    CDialog::OnOK();
}
```

Листинг 11.45. Интерфейс класса CDeNoiseDlg. Файл DeNoiseDlg.h

```
// DeNoiseDlg.h : header file
//

class CDeNoiseDlg : public CDialog
{
// Construction
public:
    CDeNoiseDlg(CWnd* pParent = NULL);    // standard constructor

// Dialog Data
    //{{AFX_DATA(CDeNoiseDlg)
```

```
    enum { IDD = IDD_DENOISE };
    CSpinButtonCtrl   m_ctrlSpinK;
    double   m_dK;
    int      m_nWhatToDo;
    //}}AFX_DATA

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDeNoiseDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDeNoiseDlg)
    afx_msg void OnChangePosSpinK(NMHDR* pNMHDR, LRESULT* pResult);
    virtual BOOL OnInitDialog();
    virtual void OnOK();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Листинг 11.46. Реализация методов класса `CDeNoiseDlg`. Файл DeNoiseDlg.cpp

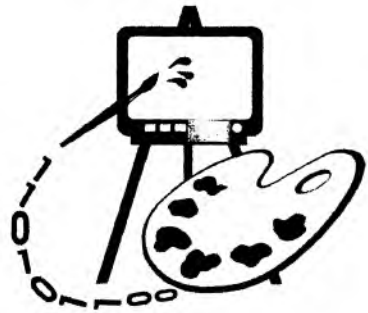
```
// DeNoiseDlg.cpp : implementation file
//

#include "stdafx.h"
#include "BMViewer.h"
#include "DeNoiseDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
////////////////////////////////////////////////////////////////////////////
// CDeNoiseDlg dialog

CDeNoiseDlg::CDeNoiseDlg(CWnd* pParent /*=NULL*/)
   : CDialog(CDeNoiseDlg::IDD, pParent)
{
   //{{AFX_DATA_INIT(CDeNoiseDlg)
   m_dK = 1.0;
   m_nWhatToDo = 0;
   //}}AFX_DATA_INIT
}

void CDeNoiseDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CDeNoiseDlg)
   DDX_Control(pDX, IDC_SPIN_K, m_ctrlSpinK);
   DDX_Text(pDX, IDC_K, m_dK);
   DDX_Radio(pDX, IDC_NOISE_SMOOTH, m_nWhatToDo);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CDeNoiseDlg, CDialog)
   //{{AFX_MSG_MAP(CDeNoiseDlg)
   ON_NOTIFY(UDN_DELTAPOS, IDC_SPIN_K, OnChangePosSpinK)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CDeNoiseDlg message handlers

#define Max_K 10
#define K_Gradation 100
void CDeNoiseDlg::OnChangePosSpinK(NMHDR* pNMHDR, LRESULT* pResult)
{
   NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;
   // TODO: Add your control notification handler code here
   m_dK=(double)pNMUpDown->iPos*Max_K/K_Gradation;
```

```
    UpdateData(FALSE);
    *pResult = 0;
}

BOOL CDeNoiseDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_ctrlSpinK.SetRange(0, K_Gradation);
    m_ctrlSpinK.SetPos(m_dK*K_Gradation/Max_K);

    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}

void CDeNoiseDlg::OnOK()
{
    UpdateData();
    CDialog::OnOK();
}
```

11.9. Заключение

Да, глава получилась не маленькая, но рассмотрено, конечно, далеко не все, чем богаты теория и практика цифровых изображений. Это и хорошо — есть простор для экспериментов. Надеемся, программа BMViewer может послужить полигоном для испытания ваших идей. Приведенную программную реализацию можно, несомненно, значительно улучшить. В текущей реализации количество фильтров ограничено, они "жестко" прописаны в классе CBMDoc. Можно было бы придумать схему динамического подключения фильтров к программе. В этом случае фильтры можно реализовывать в динамически загружаемых библиотеках (DLL). Это позволило бы расширять возможности программы уже после ее написания. Такая модель реализована в Adobe Photoshop и многих других программах. Конечно, "динамический" подход требует более серьезного осмысления задачи и планирования архитектуры приложения. Однако даже та схема, что заложена в программу BMViewer, подразумевает возможность расширения функциональности программы. Еще одним из путей развития программы, который сам собой напрашивается, является использование широких возможностей новой библиотеки GDI+, последней посвящены следующие три главы. Так что желаем успехов!



Часть IV

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ GDI+

Глава 12

Технологии .NET и GDI+: новые стандарты, новые возможности

В этой главе рассматриваются:

- основы и назначение среды .NET Framework;
- библиотека GDI+;
- создание Windows- и .NET-приложений с использованием GDI+.

За последнее время компания Microsoft подготовила программистам немало сюрпризов. Новые продукты и технологии буквально завладели вниманием "околокомпьютерного" мира. Дебют технологии .NET и основанных на ней средств разработки уже можно считать успешным. Получил широкое распространение новый язык C# (произносится "си шарп"), специально разработанный для создания .NET-приложений. В Интернете как грибы после дождя появляются все новые Web-сайты и форумы, посвященные программированию на C# и переходу на него программистов, работающих с более традиционными средами Visual C++, Visual Basic и Delphi.

Другое новшество, безусловно, вызвавшее интерес разработчиков, связанных с графикой и мультимедиа, — библиотека GDI+. Она дает возможность рядовым программистам достичь такого качества графики, какое раньше было доступно только в продуктах класса CorelDRAW! и Adobe Photoshop.

В этой главе мы познакомимся с этими технологиями и постараемся понять, какие привлекательные для программиста нововведения они в себе содержат. Впрочем, главный акцент будет сделан, конечно же, на программировании графики. Введение в .NET лишь призвано сделать понятной часть информации для тех читателей, которые пока знакомы с этой средой только понаслышке. Использование библиотеки GDI+ будет иллюстрироваться примерами как для платформы .NET, так и для "старой доброй" среды Windows.

12.1. Краткое знакомство со средой .NET Framework

Разработка сложных программных комплексов всегда была непростым делом, а уж в эпоху Интернета и мобильных технологий — и подавно. Помимо кодирования алгоритмов и формул, на практике программисты (а точнее, разработчики) сталкиваются с необходимостью повторного использования уже написанного (возможно, другими людьми) кода, доступа к самым разнообразным базам данных, взаимодействия с компьютерными устройствами, находящимися, может быть, на другой стороне земного шара. Дело осложняется зачастую тем, что создаваемые системы могут состоять из разрозненных программных компонентов, написанных на разных языках и подверженных изменениям.

Платформа .NET представляет собой попытку компании Microsoft облегчить решение этих и многих других задач. Не являясь панацеей, она тем не менее уже завоевала признание разработчиков как самая перспективная технологическая новинка последних лет в программировании.

Кроме того, провозглашаемые этой средой принципы нельзя назвать совсем уж новыми. В какой-то мере она вобрала в себя все лучшие черты существующих технологий — как конкурирующих (Java, EJB, CORBA, Delphi), так и созданных самой компанией Microsoft (COM, ADO и других).

Microsoft .NET Framework провозглашается как новая среда построения и функционирования приложений, в которой основой является Интернет. Информация становится доступной с любых устройств, освобождая программистов от необходимости привязываться к конкретной аппаратной архитектуре (системе команд процессора, разрядности машинного слова, количеству регистров и т. д.).

Изменяется также и подход к разработке приложений, позволяя трансформировать их в так называемые Web-сервисы — автономные программные компоненты, доступные для разных клиентов. Проще говоря, Microsoft .NET — это стратегия Microsoft для предоставления программных продуктов в качестве сервисов. Связующей средой для таких сервисов может выступать как Интернет, так и, например, локальные сети предприятий. А для обмена данными могут использоваться как бинарные форматы, так и "модный" в последнее время язык XML (eXtensible Markup Language, расширяемый язык разметки данных).

Все вышесказанное совсем не означает, что разрабатываемые с помощью .NET Framework приложения не будут работать без Интернета. Скорее наоборот, эта среда при необходимости облегчит перенос существующих "настольных" приложений в распределенную среду, позволив сосредоточиться на решении самих алгоритмических задач.

Примечание

Вот характерный пример: на Web-сайте RSDN.RU имеется небольшая статья с описанием компонента **RusNumber** на языке C#. Этот компонент позволяет переводить числа в соответствующие им русские числительные (с возможностью настройки суффиксов). Например, денежную сумму **1432,45 руб** он может представить в виде строки "одна тысяча четыреста тридцать два рубля сорок пять копеек". Разумеется, самая насущная сфера применения таких компонентов — "настольные" финансовые программы. Но, когда возникла такая необходимость, он был легко внесен в интернет-приложение: клиент для чтения форумов сайта **RSDN@Home**.

Итак, далее мы рассмотрим вкратце главные новинки и преимущества платформы .NET.

12.1.1. Общая среда выполнения

"Сердцем" платформы .NET является ее общая среда выполнения (Common Language Runtime, CLR). Этот набор сервисов содержится в динамической библиотеке mscoree.dll и обеспечивает функционирование всех возможностей среды. Это, в частности:

- загрузка выполняемых модулей .NET (обычных EXE- и DLL-файлов Windows, содержащих инструкции в кодах специального языка MSIL — Microsoft Intermediate Language);
- трансляция команд MSIL при первом запуске в машинные коды конкретного процессора с учетом специфики платформы;
- проверка безопасности выполняемых инструкций: загружаемые через подозрительные источники .NET-модули не смогут получить доступ к уязвимым местам компьютера (например, файловой системе) без вашего согласия;
- поддержка так называемого "управляемого" кода: программного кода, параметры выполнения которого (уничтожение неиспользуемых объектов, надлежащий вызов методов и т. д.) контролируются внешней средой. Например, программист избавляется от обязанности помнить об освобождении выделенной динамической памяти — за него об этом позаботится встроенный сборщик мусора (Garbage Collector, GC);
- единые (независимые от используемого языка программирования) механизмы обработки программных ошибок и исключений;
- взаимодействие модулей .NET с прежним, "неуправляемым" кодом — в частности, вызов методов из динамических библиотек (DLL) и работа с COM-объектами.

Об одном новшестве стоит сказать особо. При проектировании .NET была проведена кропотливая работа по объединению различных языковых типов

и особенностей "под крышей" общей системы выполнения. В результате все типы данных в .NET (даже целочисленные типы и булевы значения) являются классами, производными от общего базового класса `System.Object`.

Что это означает? Во-первых, появляется возможность разрабатывать код, который будет одинаково обрабатывать входные данные любого типа (например, механизмы сохранения состояния объектов или передачи их по сети). А во-вторых, снимаются вопросы взаимодействия компонентов, созданных на разных языках. "Строка" для класса на Visual Basic.NET означает то же самое понятие, что и "строка" для программы на C# (и представляется в памяти совершенно одинаковым образом)[1].

Более того, поддержка общей среды исполнения кода обеспечивает невиданную прежде степень межъязыковой интеграции. Например, создав базовый класс на C#, можно обеспечить наследование от него потомков в программах на C++ или VB.NET. Будем надеяться, что бушевавшие несколько десятилетий "религиозные войны" между сторонниками различных языков программирования наконец-то останутся в прошлом.

Уже сейчас поддерживается большое количество языков разработки для .NET. В комплект поставки среды Visual Studio 7.0 входят компиляторы для языков C#, Visual Basic, C++ и JScript, а также ассемблер MSIL. Кроме того, сторонними фирмами создаются компиляторы .NET для других языков программирования, включая как широко известные (Java, Smalltalk, Perl, Pascal, Cobol, Fortran), так и менее распространенные (Python, Mercury, Oberon, Eiffel). На сегодняшний день этот список состоит из более чем 20 наименований, и он быстро растет.

Отдельно заметим, что роль языка C++ нисколько не умаляется создателями .NET. Поставляемый с Visual Studio 7.0 компилятор C++ — единственный компилятор, который способен генерировать как машинные инструкции процессоров семейства x86, так и команды MSIL. Программисты могут продолжить разработку своих программ для Windows, а могут перенести их в .NET путем простой перекомпиляции исходного кода.

Однако для создания примеров использования GDI+ для .NET нами был выбран язык C#, как довольно простой для изучения и весьма популярный: вы легко сможете найти в Сети множество обучающих материалов. Для того чтобы отличить листинги кода на C++ (для платформы Windows) от листингов на C# (для .NET), мы будем указывать язык в заголовке листингов.

---

[1] Программисты, имеющие опыт работы со строками на языке C++, нас поймут — для одного этого языка создано несколько классов, поддерживающих понятие строки и несовместимых друг с другом.

12.1.2. Метаданные

Очень часто создание даже самых простых программ не сводится к написанию и компиляции кода. Практически любая среда программирования нуждается в задании дополнительных, описательных данных для правильного функционирования создаваемых программ.

Скажем, создав библиотеку динамической компоновки (DLL, Dynamically Loaded Library) на языке C++, мы нуждаемся в описании ее точек входа в специальном файле с расширением .DEF, иначе внешние программы не смогут ей воспользоваться. Технология COM также использует внешние данные, так называемые библиотеки типов (Type Libraries) — для описания интерфейсов и COM-объектов. В результате программисту на C++ приходится создавать как собственно программу (набор исходных файлов), так и дополнительные данные (файлы с расширениями def, idl, rc и т. д.) и следить за их соответствием друг другу.

Другая типичная проблема — необходимость постоянно расширять список добавочной информации. Поддержка специфических технологий (например, внесение новых нестандартных ключевых слов типа `__export` в язык C++) не решает проблемы в целом: завтра нам понадобится описать новое понятие (скажем, степень безопасности создаваемого класса), для которого ключевого слова не окажется.

Данные, описывающие другие данные (то есть "данные о данных"), принято называть *метаданными*. В разных технологиях программирования осуществляется разная степень их поддержки. К примеру, включение механизма RTTI (Run-Time Type Information) в программу на C++ позволит на этапе выполнения определять принадлежность различных объектов к общему базовому классу. Программа на Visual Basic может использовать COM-объект, содержащийся в откомпилированном файле OCX благодаря находящейся в этом же файле библиотеке типов.

Но наиболее впечатляюще реализована поддержка метаданных в среде .NET Framework. Раньше только компилятору была доступна полная информация об используемых в программе типах, методах, переменных и т. д. Основываясь на этой информации, он строил машинный код, который оперировал уже не типами, а байтами, инструкциями процессора, машинными словами, стеком и другими низкоуровневыми сущностями. Теперь же каждая программная единица в среде .NET (*сборка*), помимо собственно откомпилированного кода MSIL, обязательно содержит метаданные, описывающие как ее в целом (*манифест*), так и каждый тип, содержащийся в ней, в отдельности.

Программист может легко обращаться к любым метаданным при помощи встроенного в .NET (и в языки программирования) средства (*reflection*). Это дает гигантские возможности — в частности, исследование метаданных позволяет легко записать на диск (например, в формате XML) и затем воссоздать

(возможно, на другом компьютере) переменную любого типа — даже сложного класса, состоящую из многих составных полей. Такие средства уже существуют в .NET (сериализация и ремоутинг) и они опираются в своей работе именно на метаданные.

Кроме того, мы не ограничены в работе только существующим набором метаданных. С помощью *атрибутов* (нового языкового средства, внесенного в .NET) мы можем создавать новые виды метаданных и позволять программе анализировать их по своему усмотрению. Например, в таком сумасшедшем случае (листинг 12.1) можно указать, что "цвет" описываемого класса — красный, а "цвет" его метода `MyMethod` — синий (предположим, что атрибут `Color` введен в программе ранее):

Листинг 12.1. Применение атрибутов для расширения метаданных класса [C#]

```
[Color("Red")]
class MyClass
{
  [Color("Blue")]
  public void MyMethod(int arg);
}
```

Пользовательские атрибуты можно добавлять практически к любым элементам программы (типам, переменным, методам, параметрам и т. д.), ограничиваясь только собственной фантазией.

Заканчивая эту необъятную тему (подробнее можно прочитать статью Андрея Мартынова "Метаданные в .NET" на сайте RSDN), приведем только один пример использования метаданных, прямо связанный с программированием графики.

Библиотека GDI+ содержит довольно много перечислений (enumerations) — наборов именованных констант, позволяющих задавать параметры некоторых методов. В частности, как мы узнаем *в главе 15*, перечисление `EmfPlusRecordType` содержит 253 элемента. Что, если нам потребуется получить из численного значения элемента его строковое имя, задаваемое при компиляции? Такая задача встанет перед нами при изучении команд дискового метафайла.

Программируя на C++, нам придется вставить в свою программу таблицу (массив) из 253 вхождений, содержащую, соответственно, имена и значения каждого элемента перечисления.

Для .NET эта проблема решается тривиально благодаря мощи механизма reflection: каждый элемент любого перечисления "знает" свое строковое имя. Вызов метода `Object.ToString()` (а, как мы помним, любой тип в .NET порожден от `System.Object`) вернет строку с именем элемента.

12.1.3. Библиотека классов (.NET Framework Class Library)

Другой привлекательной составляющей среды .NET Framework является богатая коллекция программных компонентов, готовых для использования из программ на всех поддерживаемых средой .NET языках программирования, или **.NET Framework Class Library**.

Библиотека классов .NET включает тысячи компонентов, образующих целые подсистемы (сгруппированные для решения различных задач). Даже перечисление пространств имен всех входящих в библиотеку классов заняло бы слишком много места на этих страницах. Так как в нашу задачу не входит освещение работы со всеми компонентами .NET, ограничимся лишь кратким перечислением неполного (но довольно внушительного!) списка подсистем (в скобках указаны их пространства имен):

- **GDI+ (System.Drawing и System.Imaging):** "родное" для нас пространство имен. Включает в себя реализацию классов GDI+ для .NET. Будет использоваться практически в каждом примере программы на C#;
- **Windows Forms (System.Windows.Forms):** система реализации оконного графического интерфейса. Включает в себя описание окон ("forms"), визуальных компонентов ("controls") и невизуальных компонентов ("components"). Также будет активно использоваться в наших примерах;
- **ASP.NET (System.Web):** система для создания Web-приложений при помощи .NET и интернет-сервера Microsoft — Internet Information Server (IIS). Позволяет программистам начать создавать интернет-версии своих настольных программ (за счет удобства и сходства с Windows Forms);
- **Internet classes (System.Net):** компоненты для работы с интернет-протоколами. Готовые классы для использования HTTP- и почтовых протоколов, а также набор низкоуровневых классов для программирования сокетов;
- **Text processing (System.Text):** средства различной обработки текстовых данных, в том числе для преобразования кодировок текста и использования *регулярных выражений* (regular expressions);
- **Reflection (System.Reflection):** как уже отмечалось, богатая коллекция компонентов для работы с метаданными. Содержит не только средства изучения существующих метаданных, но и добавления новых метаданных в выполняющуюся программу (и даже конструирование программы "на лету"!);
- **Security Services (System.Security):** классы для обеспечения безопасности в .NET. Включают в себя как средства проверки и ограничения доступа, так и набор криптографических классов для работы с сильным шифрованием;

- **ADO.NET (System.Data):** средства доступа к различным источникам (в том числе, конечно, и к базам) данных и представления их в виде XML;
- **XML (System.Xml):** богатые средства обработки XML. Можно сказать, что этот формат — "родной" для библиотеки. И для работы с XML в этом пространстве имен существуют десятки вспомогательных классов, решающих задачи разбора и создания XML-файлов, а также сохранения ("сериализации") объектов в формате XML.

Как уже было сказано, этот список далеко не полон. Мы в своих примерах будем затрагивать в основном только компоненты GDI+. Для получения справки по другим классам, входящим в .NET, настоятельно рекомендуем обратиться к документации Microsoft по .NET Framework SDK.

12.1.4. Первые программы

Итак, настало время попробовать новую среду разработки "на вкус". Напишем две простые программы на новом языке C#. Разумеется, для создания программ для платформы .NET наиболее подходит среда Visual Studio.NET, но мы не будем вынуждать вас использовать только ее. Все примеры на C# к книге созданы так, что для их сборки необходим только бесплатный пакет .NET Framework SDK. Скачать его с сайта Microsoft можно по этому адресу **http://download.microsoft.com/download/.netframesdk/SDK/1.0/W98NT42KMeXP/EN-US/setup.exe** (131 Мбайт).

Консольный вариант "Здравствуй, мир!"

Для начала создадим пробный вариант всем известной программы "Здравствуй, мир!".

Откройте свой любимый текстовый редактор (вполне достаточно Блокнота, но для C# существует неплохой бесплатный редактор с подсветкой синтаксиса: SharpDevelop, **http://www.icsharpcode.net/**). Нам потребуется набрать в нем вот такой текст (листинг 12.2):

Листинг 12.2. Консольная программа [C#]

```
using System;

class Привет
{
  static void Main(string[] args)
  {
    string имя = "мир";
    if(args.Length>0) имя = args[0];
```

```
    Console.WriteLine("Здравствуй, {0}!", имя);
  }
}
```

Примечание

Да-да, вы не ошиблись. C# разрешает использовать кириллицу в символах идентификаторов. В данном случае, "Привет" и "имя" — вполне допустимые имена соответственно класса и переменной. Впрочем, мы демонстрируем эту возможность только в познавательных целях — практика показывает, что читаются такие "смешанные" программы весьма затруднительно. В дальнейшем для идентификаторов мы будем использовать английские имена.

Что означает первая строка? Директива `using` очень похожа на свою "сестру" в C++: она позволяет использовать символы из пространства имен без его явного указания. В данном случае так вводится видимость для пространства имен `System`. Без указания этой директивы нам пришлось бы явно писать `System.Console.WriteLine` (так как класс `Console` описан в пространстве имен `System`).

Далее идет описание класса `Привет`, который и содержит весь код нашей программы. Напоминаем, что все типы данных в .NET наследуются от класса `System.Object` (в данном случае компилятор делает это неявно).

Отметим, что в C# нет "просто функций", весь код должен находиться в методах классов, что делает этот язык очень похожим на Java. Так же как и в Java, принято считать точкой входа в программу статический метод любого класса, имеющий название `Main` (правда, в Java такой метод называется `main`). Этот метод может иметь параметр — массив строк. В этом случае в него передаются параметры командной строки запущенной программы. Обратите внимание, что массивы описываются иначе, чем в C++ — квадратные скобки указываются сразу после имени типа, а не после имени переменной.

В теле метода `Main` происходит анализ параметров и выводится на консоль либо строка "Здравствуй, мир!", либо приветствие имени, указанному в командной строке. В общем, для программистов на C++ эти строчки не составят никакой загадки.

Давайте откомпилируем этот пример. Сохраните созданный файл под именем test.cs. В среде Visual Studio (или в SharpDevelop) выберите команду компиляции в меню. Для компиляции же из командной строки необходимо выполнить следующее:

1. Убедиться, что список путей к выполняемым файлам (переменная `PATH`) содержит каталог исполняемых файлов .NET Framework (например, он может быть таким: C:\WINNT\Microsoft.NET\Framework\v1.0.3705\). Если для разработки используется операционная система семейства Windows 9*x*, то надлежащая переменная может быть установлена в файле

autoexec.bat, выполняемом при загрузке системы. При использовании ОС семейства NT необходимо настроить переменные среды (щелкнув правой кнопкой мыши на значке **Мой компьютер** и выбрав пункт меню **Свойства**).

2. Выполнить командную строку вида:

   ```
   csc test.cs
   ```

   Здесь `csc` — имя выполняемого файла компилятора C#, `test.cs` — наш исходный файл.

Если все пройдет удачно, компилятор не выдаст ошибок, и на диске получится исполняемый файл test.exe размером около 3 Кбайт. Попробуем выполнить его с параметром:

```
>test Программист
Здравствуй, Программист!
```

Ну что же, можно считать, что знакомство со средой .NET у нас состоялось.

Использование WinForms

Вторая программа будет более наглядной — для ее создания мы привлечем библиотеку Windows Forms, или, сокращенно, WinForms. Эта библиотека является основным средством создания оконного интерфейса в .NET, примерно как MFC для создания Windows-программ.

Прежде всего заметим, что главный принцип остается тем же: необходимо создать, как минимум, один класс, имеющий статический метод `Main`. Кроме того, для создания главного окна (формы) необходимо создать объект класса, порожденного от `System.Windows.Forms.Form`. Для простоты мы объединили эти два класса в один (листинг 12.3):

Листинг 12.3. Программа с использованием WinForms [C#]

```
using System.Drawing;
using System.Windows.Forms;

class FirstForm: Form
{
  Label label;

  public static void Main()
  {
    FirstForm form = new FirstForm();
    form.ShowDialog();
  }
```

```
  public FirstForm()
  {
    Text = "Первая форма";
    Height = 100;

    label = new Label();
    label.AutoSize = true;
    label.Text = "Пример для WinForms";
    label.Location = new Point(3, 3);
    label.Font = new Font("Arial", 16);
    Controls.Add(label);
  }
}
```

Так как статический метод `Main` является точкой входа в программу, в момент его выполнения еще не существует никакого объекта `FirstForm` (и никакой формы на экране, соответственно, нет). Поэтому в этом методе мы должны создать новый объект формы и вывести его на экран методом `ShowDialog()` (аналог метода `CDialog::DoModal()` в MFC).

А уже в конструкторе вновь созданного объекта происходит настройка его параметров (свойств `Text` и `Height`) и создание единственного дочернего элемента: текстовой метки (`Label`).

Как видим, пример очень компактный, но тем не менее может создавать полноценную форму. Постарайтесь самостоятельно откомпилировать и запустить его. У вас должно получиться примерно такое окно:

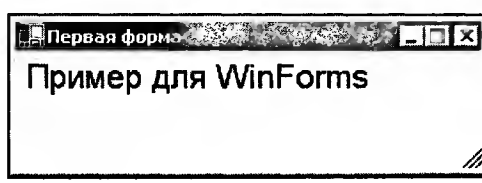


Рис. 12.1. Окно созданной WinForms-программы

Единственное обстоятельство, которое может немного смутить программистов на C++ — это непривычная работа с динамической памятью. Нет указателей в привычном смысле слова, а оператор `new` используется без последующего освобождения памяти

Добро пожаловать в мир *управляемого* (managed) кода! Все классы в C# являются ссылочными типами, и за их освобождением следит встроенный

в .NET сборщик мусора. Немного подробнее об этом будет сказано в *главе 14*, а пока просто положитесь на среду выполнения — за вашей программой "приберут".

На этом мы закончим краткое знакомство со средой .NET Framework. Полученных сведений вполне хватит для того, чтобы разобраться в приводимых в этой книге примерах программ для .NET. Если вы заинтересовались этой (безусловно, перспективной) платформой, советуем вам начать с изучения документации, поставляемой с .NET Framework SDK.

А мы перейдем к главной теме этой главы: библиотеке GDI+.

12.2. Введение в GDI+

Другой популярной новинкой, предлагаемой Microsoft, является технология GDI+. Именно она (вернее, основанный на ней новый графический интерфейс) представляет "лицо" новых операционных систем — Windows XP и Windows Server 2003.

Что же такое GDI+? Официальная документация скромно называет ее Class-based API, то есть основанным на классах интерфейсе прикладных программ. Так как она встроена в Windows XP и Windows Server 2003, ее называют частью этих операционных систем. Часто встречается также определение "библиотека" или "библиотека классов". В действительности предоставляемый GDI+ набор классов является тонкой оболочкой над множеством обычных функций, реализованных в одной динамической библиотеке GdiPlus.dll (размером около полутора мегабайт). В общем, имея все это в виду, будем для краткости далее называть ее просто библиотекой.

Итак, GDI+ — это библиотека, призванная заменить существующий уже больше 12 (или даже 19, если считать с начала разработки Windows 1.0) лет интерфейс GDI, являющийся графическим ядром предыдущих версий Windows. Она сочетает в себе (по крайней мере, по замыслу) все достоинства своего предшественника и предоставляет множество новых мощных возможностей. Кроме того, при ее проектировании заранее ставилась цель наименее болезненного переноса приложений на 64-битные платформы. Следовательно, хотя существующие GDI-приложения будут выполняться на новых версиях Windows, для новых проектов рекомендуется рассмотреть возможность использования GDI+.

12.2.1. Что новенького?

Далее мы еще будем рассматривать специфические (и такие эффектные!) возможности GDI+. Здесь же только опишем основные новшества.

Достоинства C++-реализации:

- *объектно-ориентированный интерфейс*: благодаря поддержке компилятора C++ мы "бесплатно" получаем контроль над типами и временем жизни объектов;
- *прозрачное управление памятью*: объекты ядра GDI+ создаются "в куче" с помощью собственного менеджера памяти прозрачно для программиста;
- *использование перегрузки имен функций*: функции одного назначения различаются только по своим параметрам;
- *собственное пространство имен*: все типы GDI+ описаны в пространстве имен Gdiplus, что позволяет использовать понятные имена типов (такие, как `Rect`, `Pen` и `Matrix`) без конфликтов с другими библиотеками;
- *перегрузка операторов*: предоставляет удобные операции '+' и '—' для таких типов, как `Point` и `Size`.

Архитектурные новинки библиотеки:

- *аппаратная абстракция*: как уже было замечено, упрощается перенос на 64-битные платформы;
- *новый дизайн графических функций/объектов*: теперь можно не бояться "оставить выбранной кисть в контексте перед удалением" — такая типичная для GDI ошибка!
- *разделение функций закраски и отрисовки*: предоставляет большую гибкость в рисовании, например позволяет заливать незамкнутые фигуры.
- *увеличившаяся поддержка траекторий (paths) и их взаимодействия с регионами*: теперь траектории являются полноправными объектами вне контекста рисования и могут легко трансформироваться в регионы.

Новые технологии и возможности (задержите дыхание):

- *градиентная закраска*: позволяет заливать сложные фигуры оттенками с различными законами распределения цвета, рисовать векторные примитивы (например, линии) с градиентной окраской;
- *поддержка прозрачности*: можно создавать кисти и растры с прозрачными и полупрозрачными областями, заливать области полупрозрачным цветом, назначать Color Key для растрового изображения и работать с его альфа-каналом, а также рисовать полупрозрачные (!) векторные примитивы и текст;
- *режимы улучшения изображения*: позволяют значительно улучшить пользовательское восприятие за счет сглаживания контурных неровностей (anti-aliasing) и префильтрации (interpolation) растровых изображений;
- *сплайны*: кроме уже существующих в GDI кривых Безье, поддерживается новый вид кривых: так называемые сплайны, которые имитируют поведение натянутой и изогнутой стальной полосы. Сплайны являются гладкими кривыми;

- *траектории*: как уже говорилось, траектории теперь существуют независимо от контекста рисования и представляют собой мощное средство создания сложных векторных объектов. Кроме того, появилась возможность выравнивать (flatten) траектории, то есть преобразовывать их к набору отрезков прямых;
- *координатные преобразования*: объект `Matrix` позволяет осуществлять операции поворота, переноса, масштабирования и отражения объектов GDI+;
- *регионы*: в отличие от GDI, регионы теперь не привязаны к координатам устройства и подчиняются координатным преобразованиям;
- *работа с растрами*: теперь можно практически все! Поддерживается отрисовка растров с наложением внешнего альфа-канала, масштабированием, растяжением, искажением и поворотом растров. При этом можно установить режимы отображения отдельных пикселов — от простого переноса до префильтрации (наилучшее качество изображения). Стало возможным рисовать векторные примитивы, залитые текстурами (!).
- *поддержка популярных форматов графических файлов*: необычайно приятное новшество для всех программистов, имеющих дело с разными графическими форматами. Поддерживаются форматы BMP, GIF, TIFF, JPEG, Exif (расширение TIFF и JPEG для цифровых фотокамер), PNG, ICO, WMF и EMF. Декодеры различных форматов выполнены с учетом их специфики, так что вы сможете, например, отобразить анимационный GIF или добавить комментарий к TIFF-файлу. Загруженный, созданный или модифицированный файл может быть сохранен на диск в одном из подходящих форматов. Заявлена (но пока не реализована) также возможность написания собственных декодеров;
- *формат EMF+*: разумеется, все это великолепие не могло уместиться в тесные рамки старого Enhanced Metafile. Для описания новых возможностей был создан новый формат метафайла EMF+, который позволяет сохранить на диск и затем проиграть последовательность графических команд. Существует возможность записать "дуальный" метафайл, понятный старым GDI-программам. Новые программы будут читать из него GDI+-информацию. Подробнее о метафайлах будет рассказано в *главе 14*.

12.2.2. Требования к среде выполнения

Поддержка GDI+ встроена непосредственно в операционные системы Windows XP и Windows Server 2003. Для того чтобы приложения, использующие эту библиотеку, выполнялись на предыдущих версиях Windows, необходимо установить дистрибутив gdiplus_dnld.exe размером около одного мегабайта. Найти его можно на сайте Microsoft по адресу:

http://www.microsoft.com/downloads/release.asp?releaseid=32738

В его состав входят только инструкция по установке и уже упомянутая динамическая библиотека GdiPlus.dll, которую необходимо скопировать в системный каталог Windows 98/ME, Windows NT SP6 или Windows 2000. При этом возможности, предоставляемые непосредственно ядром Windows XP (в частности, технология ClearType для качественного отображения шрифтов на LCD-мониторах), будут недоступны.

Если вы установили на своем компьютере среду .NET Framework или пакет Visual Studio.NET, то это значит, что библиотека GDI+ уже установлена.

12.2.3. Поддержка GDI+ в Windows 95

Что касается технической стороны вопроса, то эксперименты показали: установка дистрибутива gdiplus.dll в системах с Windows 95 возможна, и демонстрационные приложения с использованием GDI+ выполняются без заметных проблем.

С правовой же точки зрения этого делать нельзя. Microsoft достаточно жестко указывает в лицензионном соглашении дистрибутива, что он может устанавливаться только на следующие операционные системы: "Windows 2000, Windows Millennium Edition, Windows NT 4.0 and Windows 98". Как уже говорилось, Windows XP располагает собственной версией GDI+.

С чем связано такое ограничение? Возможно, с тем что компания из Редмонда вообще прекратила техническую поддержку Windows 95: **http://www.microsoft.com/windows/lifecycleconsumer.asp**.

12.2.4. Поддерживаемые технологии разработки

Мы будем рассматривать интерфейсы к GDI+, реализованные для WinAPI и для системы CLR, входящей в состав Microsoft .NET Framework. Кстати, на данный момент усилиями энтузиастов уже созданы неофициальные реализации и для других сред разработки (например, Delphi), но мы о них рассказывать не будем.

Заметим, что GDI+ (вернее, ее обертка для CLR) является основным средством рисования в среде .NET. Однако она имеет довольно большие отличия от реализации для C++ (не только архитектурные, но и чисто внешние, например, различающиеся имена некоторых классов). Мы будем обращать на них внимание по мере необходимости.

Набор заголовочных файлов (headers) и библиотека импорта GdiPlus.lib, необходимые для сборки демонстрационных приложений на C++, входят в состав последнего пакета Platform SDK. Те, кто до сих пор не обновил идущий с Visual Studio 6.0 Platform SDK образца 1998 года, могут загрузить его с сайта Microsoft по адресу:

http://www.microsoft.com/msdownload/platformsdk/sdkupdate/

Минимальный компонент, в состав которого входит GDI+, называется Windows Core SDK и имеет размер около 230 Мбайт. На момент написания этих строк на сайте Microsoft доступна для загрузки версия Platform SDK за февраль 2003 года.

Демонстрационные примеры для Windows будут в основном написаны с использованием Windows API, что позволит сосредоточиться на использовании GDI+. Но вы без труда сможете подключить эту библиотеку к своим MFC- или WTL-приложениям. Соответствующие примеры для платформы .NET будут написаны на языке C# для WinForms.

12.2.5. Иерархия классов GDI+

Типичное рабочее место программиста на C++, как правило, включает в себя стену, на которой гордо красуется Диаграмма классов (неважно, каких). Теперь рядом можно наклеить еще один плакат.

Ниже приведена иерархия классов GDI+. В нее не включены для краткости 8 структур данных и перечисления (enumerations) — около 50 штук.

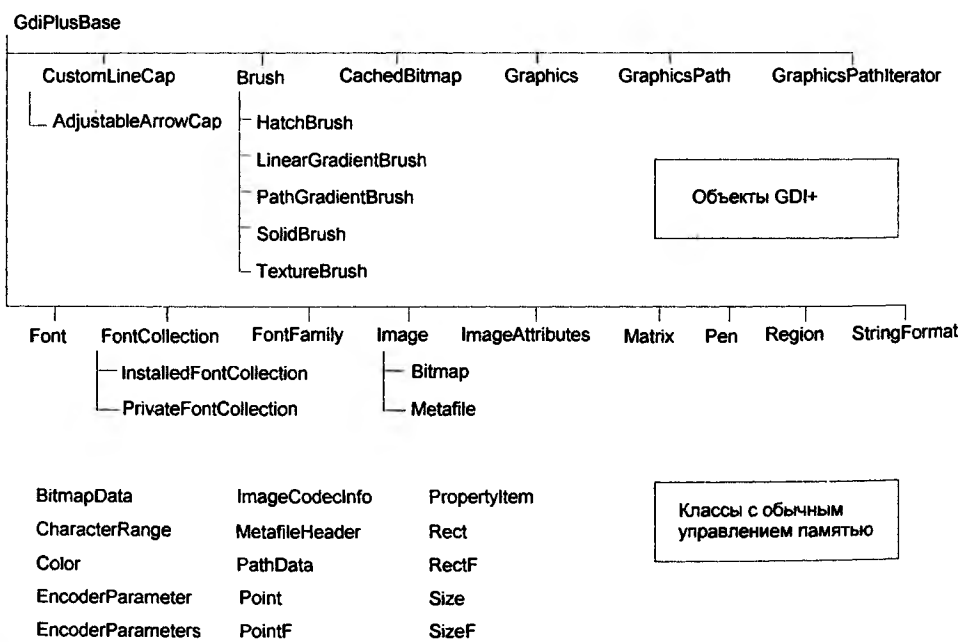


Рис. 12.2. Иерархия классов GDI+

При первом взгляде на диаграмму видно, что она очень напоминает, например, ту часть библиотеки MFC, которая отвечает за рисование, только классов

гораздо больше (40 против 15 у MFC). Это сходство и неудивительно, учитывая фирму, которая разрабатывала эти библиотеки. Основные отличия отражают новые возможности GDI+. Мы подробно рассмотрим их в следующих главах.

Как видим, большинство объектов имеют в корне иерархии класс `GdiPlusBase`. Вам не понадобится создавать экземпляры этого класса, так как он содержит только средства управления памятью (для него перегружены операторы `new/new[]` и `delete/delete[]`, которые используют функции GDI+ `GdipAlloc` и `GdipFree`). Все классы, инкапсулирующие работу с ресурсами GDI+, порождены от `GdiPlusBase`. Это не значит, что их экземпляры нельзя создавать на стеке — напротив, так даже удобнее контролировать время их жизни. Зато такая архитектура позволит, например, передавать указатель на созданный объект GDI+ в модуль, написанный с использованием других средств разработки, и безопасно его удалять в этом модуле. Кроме того, теоретически это позволит организовать "сборку мусора" для выделенных таким образом объектов — в стиле .NET.

Примечание

Не путайте управление памятью под экземпляры классов-оберток C++, которое осуществляется перегруженными операторами new/delete, и управление собственно ресурсами GDI+, которое скрыто от разработчиков в недрах соответствующих функций например, **GdipCreateSolidFill**.

Ключевым же классом в GDI+ является `Graphics` (здесь программисты на J++ могут вздрогнуть от знакомства с одноименным классом в Microsoft Java VM). Именно он содержит почти две сотни методов, отвечающих за рисование, отсечение и параметры устройства вывода. Напрашивается явная аналогия с контекстом устройства (Device Context) прежнего GDI, и эти понятия действительно тесно связаны. Из четырех конструкторов `Graphics` два создают его из HDC. Главное отличие заключается в изменении программной модели: теперь вы не работаете с хендлом, а вызываете методы класса. Хотя программистам на MFC эта концепция уже хорошо знакома.

Дальнейшее наследование (например, класс `TextureBrush` порожден от `Brush`) скорее отражает цели разработчиков (скрытие деталей реализации и повторное использование оберточного кода), чем инфраструктуру библиотеки, так как в inline-методах "родственных" классов просто содержатся вызовы различных функций GdiPlus.dll. Можно сказать, что Microsoft в очередной раз спроецировала обычный "плоский" API-языка C на объектно-ориентированную библиотеку C++.

Оставшаяся часть классов не имеет общего родителя и предназначена для упрощения работы со структурами данных GDI+. Мы познакомимся с ними подробнее при обсуждении средств растровой и векторной графики.

В среде .NET Framework не используется реализация для C++, а все классы являются "обертками" низкоуровневых типов Gdiplus.DLL. Видимо, с этим обстоятельством связаны некоторые различия в именах классов и методов в двух средах (например, Rect в C++ и Rectangle в .NET).

12.2.6. Инициализация и завершение

Перед тем как начать использовать классы и функции GDI+, необходимо инициализировать эту библиотеку. В среде .NET Framework это делается автоматически при старте WinForms-приложения, а вот программистам WinAPI придется потрудиться. Для этого где-нибудь в начале своей программы нужно поместить вызов функции `GdiplusStartup`:

```
Status GdiplusStartup( ULONG_PTR* token,
    const GdiplusStartupInput* input,
    GdiplusStartupOutput* output );
```

Поля структуры `GdiplusStartupInput` управляют различными аспектами инициализации: в частности, можно задать функцию, которая будет вызываться при возникновении ошибок или перехватывать все обращения к функциям GDI+. Эти детали мы рассматривать не будем. К счастью, конструктор по умолчанию структуры `GdiplusStartupInput` выполняет инициализацию, достаточную в большинстве случаев. При этом в качестве выходного параметра `output` можно задать `NULL`.

"Магическое значение", на которое указывает выходной параметр `token`, необходимо сохранить.

Для завершения работы с библиотекой вызовите функцию `GdiplusShutdown`:

```
VOID GdiplusShutdown( ULONG_PTR token );
```

Здесь в качестве параметра и необходимо передать то самое число, которое возвратила `GdiplusStartup` в параметре `token`.

Примечание

Вы можете вызвать `GdiplusStartup` и `GdiplusShutdown` из разных потоков, но необходимо убедиться, что вне этой пары функций никакого обращения к объектам GDI+ не происходит. В частности, будьте осторожны, объявляя глобальными экземпляры классов — ведь их деструкторы выполнятся уже после WinMain. Кроме того, как обычно, нельзя вызывать функции инициализации и очистки из `DllMain`, поскольку это может привести ко входу в бесконечную рекурсию или другим неприятностям.

12.3. Создаем первые приложения

Итак, довольно сухой теории! Настало время применить все эти сведения на практике. Для этого создадим две простые оконные программы, которые

делают одно и то же: читают рисунок из графического файла формата GIF, выводят его в окно в качестве фона, а затем выводят поверх фона текстовую строку выбранным шрифтом и градиентом цвета. Первая программа будет написана на C++, а вторая на C#.

12.3.1. Используем GDI+ в WINAPI

Итак, стартуем Visual Studio и создаем новый проект Win32 Application. Выбираем опцию **A typical "Hello, World!" application** и нажимаем **Finish**. Получившееся приложение необходимо подготовить для использования GDI+. Для этого в файле stdafx.h после строки с комментарием:

```
// TODO: reference additional headers
// your program requires here
```

добавляем следующие строки:

```
#include <GdiPlus.h>
using namespace Gdiplus;
```

Это позволит нам не указывать в дальнейшем префикс `Gdiplus::` перед указанием любых имен библиотеки. Как уже говорилось, все они содержатся в пространстве имен `Gdiplus`.

Пойдем дальше. В конце файла stdafx.cpp добавляем строку

```
#pragma comment(lib, "GdiPlus.lib")
```

Кроме того, в файле stdafx.h необходимо удалить или закомментировать строку

```
#define WIN32_LEAN_AND_MEAN // Exclude rarely-used stuff
// from Windows headers
```

Иначе компилятор выдаст множество ошибок об отсутствии символов `MIDL_INTERFACE`, `PROPID`, `IStream` и т. д.

Если полученное в результате приложение успешно собралось, значит, мы все сделали правильно. Пойдем дальше.

Найдем в сгенерированном основном файле с расширением cpp нашего проекта функцию `WinMain` и добавим в ее начале код инициализации:

```
GdiplusStartupInput gdiplusStartupInput;
ULONG_PTR gdiplusToken;
GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);
```

а в конце, перед оператором `return`, добавим код очистки:

```
GdiplusShutdown(gdiplusToken);
```

Готово. Наконец-то мы можем что-нибудь нарисовать! Найдите в теле функции `WndProc` обработчик сообщения `WM_PAINT` и замените следующим кодом (листинг 12.4):

Листинг 12.4. Текст обработчика оконного сообщения WM_PAINT [C++]

```
case WM_PAINT:
{
    RECT rc;
    GetClientRect(hWnd, &rc);
    hdc = BeginPaint(hWnd, &ps);
    // Вызываем функцию рисования
    OnPaint(hdc, rc);
    EndPaint(hWnd, &ps);
    break;
}
```

Теперь где-нибудь перед функцией `WndProc` создадим функцию `OnPaint` с кодом рисования (листинг 12.5):

Листинг 12.5. Функция рисования [C++]

```
void OnPaint(HDC hdc, const RECT& rc)
{
    // Все строки - в кодировке Unicode
    WCHAR welcome[]=L"Welcome, GDI+ !";

    // Создаем контекст рисования и устанавливаем
    // пиксельную систему координат
    Graphics g(hdc);
    g.SetPageUnit(UnitPixel);
    RectF bounds(0, 0, float(rc.right), float(rc.bottom));

    // Загружаем фоновое изображение
    // и растягиваем его на все окно
    Image bg(L"BACKGRND.gif");
    g.DrawImage(&bg, bounds);

    // Создаем кисть с градиентом на все окно
    // и полупрозрачностью
    LinearGradientBrush brush(bounds,
        Color(130, 255, 0, 0), Color(255, 0, 0, 255),
        LinearGradientModeBackwardDiagonal);
```

```
    // Готовим формат и параметры шрифта
    StringFormat format;
    format.SetAlignment(StringAlignmentCenter);
    format.SetLineAlignment(StringAlignmentCenter);
    Font font(L"Arial", 48, FontStyleBold);

    // Выводим текст приветствия, длина -1 означает,
    // что строка заканчивается нулем
    g.DrawString(welcome, -1, &font, bounds, &format, &brush);
}
```

В результате у нас получится примерно вот что:

Рис. 12.3. Результат выполнения программы

Примечание

Приведенный пример носит только ознакомительный характер. В реальном приложении, для того чтобы нарисовать растр, его, как правило, не нужно каждый раз загружать с дискового файла.

Далее мы будем пользоваться созданным макетом программы для создания других демонстрационных приложений. В качестве примеров рисования будут приводиться только интересующие нас фрагменты кода. Полные тексты приложений можно найти на компакт-диске.

12.3.2. Типичные трудности при компиляции и сборке проектов на C++

Хочется указать на несколько "подводных камней", которые могут сбить с толку при первой попытке откомпилировать и собрать проект, использующий GDI+. В основном здесь упомянуты те проблемы, с которыми сталкиваются (и постоянно спрашивают о них в различных форумах) начинающие.

Где взять GdiPlus.h?

Как уже было сказано, все заголовочные файлы, библиотека импорта и документация к библиотеке входят в состав последнего Platform SDK. Они не идут в составе Visual C++ 6.0, Visual Studio 7.0 и их сервис-паков. Счастливым обладателям Visual Studio 2003 ничего скачивать не нужно: заголовочные файлы включены в комплект поставки этой среды разработки.

Странные имена типов...

Многих новичков сбивают с толку типы параметров, используемые в GDI+: `INT`, `REAL`, `ULONG_PTR` На самом деле они являются синонимами более привычных для программиста C++ типов: `int`, `float` и `unsigned long`. Но, тем не менее, рекомендуется использовать именно предлагаемые синонимы: это улучшит совместимость ваших исходных текстов с компиляторами других фирм, а также поможет перенести их на 64-битные платформы.

Почему выдается ошибка о типе ULONG_PTR?

Похоже, что компилятор находит старый заголовочный файл basetsd.h — например, из комплекта VC++ 6.0. Измените пути поиска заголовочных файлов так, чтобы вначале были найдены файлы нового Platform SDK.

Почему компилятор не дает создать объект GDI+ при помощи *new*?

Такое поведение возможно при попытке откомпилировать MFC-приложение с использованием GDI+ в Debug-конфигурации.

В начале файла программы, видимо, имеется следующий фрагмент:

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

Этот фрагмент перекрывает стандартный распределитель памяти `new` и не дает использовать `operator new`, определенный в заголовочных файлах GDI+.

Для решения этой проблемы либо откажитесь от создания объектов GDI+ с помощью `new`, либо откажитесь от проверок динамической памяти в этом файле (удалив вышеприведенную директиву `#define`).

Примечание

Недавно на сайте Microsoft появилась соответствующая статья в Knowledge Base: Q317799 PRB: Microsoft Foundation Classes DEBUG_NEW Does Not Work with GDI+ (**http://support.microsoft.com/default.aspx?scid=kb;en-us;Q317799**).

Не забудьте про пространство имен Gdiplus и библиотеку импорта

В приводимых примерах кода используются простые имена классов, такие как `Brush` и `Rect`. Это стало возможным благодаря тому, что в начале заголовочного файла программы есть директива

```
using namespace Gdiplus;
```

Если это решение не подходит (например, в проекте уже существуют классы с такими именами), то перед именами классов необходимо ставить префикс пространства имен, например:

```
Gdiplus::Rect rect;
```

Также, если по каким-то соображениям директива

```
#pragma comment(lib, "gdiplus.lib")
```

не устраивает, в опциях компоновщика нужно явно указать библиотеку импорта gdiplus.lib.

12.3.3. Облегчаем себе жизнь: класс для автоматической инициализации библиотеки

Все приводимые в MSDN примеры ориентированы на статическую компоновку с библиотекой импорта Gdiplus.Lib. При использовании GDI+ возник довольно интересный вопрос: а что, если система, в которой запущено приложение, не поддерживает GDI+? Как в таком случае можно избежать системного сообщения об ошибке и продолжить работу с использованием GDI?

Обычно в таком случае программист реализует самостоятельную загрузку искомой DLL с помощью функции `LoadLibrary`, и в случае успеха получает адреса необходимых функций с помощью процедуры `GetProcAddress`. Но в нашем случае это неприменимо: ведь в отличие от большинства обычных

библиотек импорта, "начинка" GDI+ спрятана за структурой классов-оберток.

Здесь может помочь такая сравнительно малоизвестная возможность Visual C++, как *отложенная загрузка* (Delayed Loading) DLL. При использовании этой опции компоновщик генерирует специальные заглушки для всех импортируемых функций. Приложение стартует немного быстрее, а реальная загрузка библиотеки откладывается (отсюда и название) до первого вызова любой импортируемой функции. Подробно почитать об этой технике можно в декабрьском выпуске Microsoft System Journal (теперь MSDN Magazine) за 1998 год сразу в двух колонках: "Under The Hood" Мэтта Питрека и "QnA Win32" Джефри Рихтера.

Здесь же только упомянем, что для использования отложенной загрузки в проект необходимо включать библиотеку DELAYIMP.LIB, а динамическую библиотеку компоновать с опцией компоновщика /delayload:module_name, где module_name — имя библиотеки.

Скрыть детали использования DLL и инициализации GDI+ можно, поместив необходимый код в методы какого-нибудь класса (назовем его `InitGdiPlus`). Тогда жизнь программиста кардинально упрощается: для инициализации графической подсистемы достаточно определить в своей программе экземпляр такого класса (его описание приведено в листинге 12.6).

Листинг 12.6. Описание класса `RSDN::InitGdiPlus` [C++]

```
// InitGdiPlus.h
namespace RSDN
{
  class InitGdiPlus
  {
  public:
    InitGdiPlus();
    virtual ~InitGdiPlus();

// Была ли инициализация удачной?
    bool Good(){ return present; }
  private:
    bool present;
    ULONG_PTR token;
 };
}
```

При рассмотрении реализации данного класса учтем следующее:

Во-первых, при попытке использовать любую функцию GdiPlus.DLL в системе, где эта библиотека не установлена, возникнет структурное исключение (Structured Exception, SE). Класс должен обрабатывать такое исключение и сообщать о неудаче. В данном случае мы добавили в описание класса метод `Good()`, сигнализирующий об успехе или провале инициализации GDI+.

Во-вторых, по крайней мере в Visual Studio 6.0 SP4/SP5, существует ошибка, приводящая к аварийному завершению компоновщика при линковке Delay-loading модулей и модулей с отладочной информацией (см., например, сообщение в форуме RSDN: **http://www.rsdn.ru/Forum/Message.aspx?mid=101066**). Поэтому для отладочной версии нашего класса отложенная загрузка выполняться не будет (все равно, при отладке на вашей системе GDI+ установлена).

В-третьих, желательно максимально упростить использование класса и не выносить наружу детали реализации (в частности, опции компоновщика имеет смысл задать в виде директивы `#pragma comment`). Реализация класса с учетом этих замечаний приведена в листинге 12.7.

Примечание

К сожалению, компилятор Visual C++ 7.0 воспринимает директиву #pragma с опцией /delayload как ошибочную и выдает соответствующее предупреждение. Пользователям VC7 придется задавать эту опцию самостоятельно, в командной строке компилятора или в свойствах проекта.

Листинг 12.7. Реализация класса `RSDN::InitGdiPlus` [C++]

```
// InitGdiPlus.cpp
#define STRICT
#include <windows.h>
#include <GdiPlus.h>
#include "InitGdiPlus.h"

#pragma comment(lib, "gdiplus.lib")

#ifndef _DEBUG
#pragma comment(lib, "delayimp.lib")
#pragma comment(linker, "/delayload:gdiplus.dll")
#endif

namespace RSDN
```

```
{
  InitGdiPlus::InitGdiPlus()
  {
    present=true;
    Gdiplus::GdiplusStartupInput input;
    __try
    {
      Gdiplus::GdiplusStartup(&token, &input, 0);
    }
    __except(1)
    {
      present=false;.
    }
 }

  InitGdiPlus::~InitGdiPlus()
  {
    if(present) Gdiplus::GdiplusShutdown(token);
  }
}
```

Таким образом, этот класс самостоятельно занимается инициализацией/очисткой GDI+, а также обрабатывает структурное исключение, возникающее при отсутствии библиотеки Gdiplus.dll. Для инициализации библиотеки достаточно объявить экземпляр такого класса. Только не забывайте, что время его жизни должно быть больше времени жизни создаваемых в вашей программе объектов GDI+, иначе их деструкторы выполнятся после вызова `GdiplusShutdown`.

Примечание

Для более детального анализа возникшего исключения в конструкторе нашего класса можно применить функцию `DelayLoadDllExceptionFilter`, описанную в указанной статье Рихтера.

12.3.4. Пример *WinForms* — приложения с использованием GDI+

Для того чтобы можно было сравнить рассматриваемую реализацию GDI+ с той, что используется в .NET, рассмотрим полный текст соответствующего приложения на языке C#.

Для новичков в мире .NET будет интересно такое обстоятельство: работа с оконными Windows-сообщениями полностью скрывается библиотекой Windows Forms. Вместо этого предлагается парадигма *событий* (механизмов обратного вызова) и *делегатов* (своего рода реакций на события, напоминающих указатели на функции C++).

Как правило, для реализации рисования в окне (или элементе управления) WinForms необходимо подписаться на событие `Paint`. Мы же для простоты воспользовались тем, что это событие генерируется в теле виртуального метода `OnPaint` формы, и перекрыли его с помощью ключевого слова `override`. При этом для правильного функционирования цепочки событий необходимо вызвать оригинальный метод `OnPaint`. Текст приложения приведен в листинге 12.8.

Листинг 12.8. Текст программы для WinForms [C++]

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

// класс формы
public class GraphicsForm: Form
{
    // выполнение начнется здесь
    public static int Main()
    {
        Form fm = new GraphicsForm();
        fm.ShowDialog();
        return 0;
    }

    // перекрываем (override) виртуальный метод рисования
    protected override void OnPaint(PaintEventArgs a)
    {
        DoPaint(a.Graphics, a.ClipRectangle);
        // вызываем унаследованный метод (важно!)
        base.OnPaint(a);
    }

    // функция с кодом рисования
```

```
    protected void DoPaint(Graphics g, Rectangle clipBox)
    {
        RectangleF bounds = clipBox;
        string welcome = "Welcome, GDI+ !";
        Bitmap bg = new Bitmap("BACKGRND.gif");
        g.DrawImage(bg, bounds);
        LinearGradientBrush brush =
            new LinearGradientBrush(bounds,
                Color.FromArgb(130, 255, 0, 0),
                Color.FromArgb(255, 0, 0, 255),
                LinearGradientMode.BackwardDiagonal);
        StringFormat format = new StringFormat();
        format.Alignment = StringAlignment.Center;
        format.LineAlignment = StringAlignment.Center;
        Font font = new Font("Arial", 48, FontStyle.Bold);
        g.DrawString(welcome, font, brush, bounds, format);
    }
}
```

Как видим, помимо чисто синтаксических отличий имеются и принципиальные, например, использование в CLR-модели свойств — против использования Set-методов в C++. Кроме того, в .NET активно используются пространства имен.

Примечание

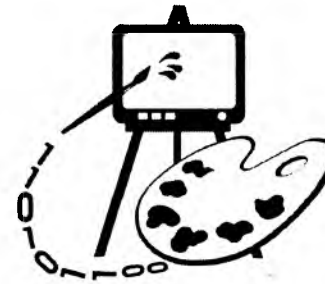
Заметим, что здесь приведен полный листинг программы, аналогичной по возможностям той, что мы создали в предыдущем разделе. Сравните объем исходных текстов этих двух примеров (на компакт-диске). Как видим, среда .NET берет на себя большинство рутинной работы по созданию класса окна и обработке оконных сообщений.

Если вы запустите приведенный пример, то увидите, что текст отрисовывается без сглаживания, характерного для предыдущего примера. Это связано с тем, что WinForms по умолчанию отключает улучшенный режим отрисовки шрифтов — и без этого причин для "торможения" достаточно. В следующей главе мы узнаем, как можно управлять режимами сглаживания.

Итак, мы познакомились с библиотекой GDI+ и научились использовать ее в своих программах для .NET и Windows API.

В следующей главе мы рассмотрим богатые возможности, которые GDI+ предоставляет для работы с растровыми изображениями.

Глава 13



Работа с растрами и графическими файлами в GDI+

В этой главе рассматриваются:

- классы библиотеки GDI+ для хранения и вывода растров;
- работа со стандартными графическими файловыми форматами;
- управление выводом растров;
- прямой доступ к растровым данным GDI+.

13.1. Основные понятия

В *главах 9–11* мы уже познакомились с использованием растров в Windows-графике. Теперь настало время убедиться, что встроенные в GDI+ средства работы с растрами делают эту библиотеку необычайно мощным инструментом. Для того чтобы двигаться дальше, необходимо прояснить некоторые используемые термины:

- источник — растровое изображение, которое используется для вывода;
- приемник — исходный растр (или графическое устройство), в который происходит вывод пикселов источника;
- прозрачность — свойство некоторых пикселов источника не отображаться, оставляя видимыми пикселы приемника;
- цвет прозрачности (*color key*) — условный цвет (или индекс в таблице цветов), который используется для указания прозрачных пикселов;
- альфа-наложение (*alpha blending*) — такой способ взаимодействия пикселов источника и приемника, когда итоговый цвет определяется их взвешенной суммой. При этом достигается эффект *полупрозрачности* источника;
- альфа-коэффициент (или величина Aplha) — используемый при альфа-наложении весовой коэффициент. В GDI+ используется целочисленный

альфа-коэффициент с интервалом допустимых значений от 0 (полная прозрачность) до 255 (полная непрозрачность);.

- альфа-канал — совокупность растровых данных, содержащих значения альфа-коэффициентов для каждого пиксела (по аналогии с красным, зеленым и синим каналами).;
- антиалиасинг (antialiasing) — методика сглаживания векторных изображений для устранения контурных неровностей.

Вооружившись этими определениями, приступим к изучению класса Bitmap.

13.2. Класс *Bitmap* — контейнер растровых изображений

Для хранения изображений в библиотеке GDI+ предназначен класс `Image`. Его потомки, классы `Bitmap` и `Metafile`, реализуют, соответственно, функциональность для работы с растровыми и векторными изображениями. Кстати, в документации .NET Framework SDK утверждается, что потомком `Image` является и класс `Icon`, но это не так: он унаследован от `System.MarshalByRefObject`. В иерархии классов GDI+ для C++ класса `Icon` не существует (все необходимые средства для работы с иконками находятся в `Bitmap`).

Итак, в этой главе мы постараемся хорошенько изучить класс `Bitmap` и особенности работы с ним. Такое внимание к единственному классу вполне обосновано: он предоставляет очень много возможностей. Например, если вы создаете растровую кисть (`TextureBrush`) или графический контекст (`Graphics`) в памяти, для их инициализации потребуется экземпляр класса `Bitmap`.

13.2.1. Поддержка основных графических форматов

Это качество является одним из наиболее привлекательных свойств библиотеки GDI+. Например, скромный и неприметный редактор Paint в Windows XP неожиданно приобрел возможность открывать и сохранять не только картинки формата BMP, но также и JPEG, TIFF, GIF и PNG, что сразу сделало его на порядок более мощным средством. Это полезное качество появилось в нем благодаря использованию GDI+ (Paint из комплекта Windows 2000 тоже поддерживал GIF и JPEG, но делал это заметно хуже, используя набор модулей расширения FLT, впервые появившихся в Office 97).

К модулям работы с графическими форматами GDI+ уже прочно прикрепилось жаргонное название "кодек" (codec, Compressor/Decompressor). Чтобы не отстать от моды, будем их так называть и мы.

У класса Bitmap существует набор перегруженных конструкторов для создания или загрузки изображений. При загрузке содержимого файла анализируется его формат (а вовсе не расширение!), и автоматически используется соответствующий кодек. Определить, какой кодек (и какой формат) был использован при загрузке можно с помощью метода Image::GetRawFormat (листинг 13.1):

Листинг 13.1. Загрузка графических файлов формата JPEG [C++]

```
Bitmap bm(L"Picture.dat"); // обратите внимание на расширение
GUID guidFileFormat;
bm.GetRawFormat(&guidFileFormat);
// проверим, действительно ли в файле находился JPEG
if(guidFileFormat == ImageFormatJPEG)
   MessageBox(0, "Это JPEG!", 0, MB_OK);
```

Константы форматов (ImageFormatXxx, где Xxx — формат) определены в заголовочных файлах GDI+. Только не спутайте их с идентификаторами кодеков, которые будут обсуждаться ниже. В .NET определение формата производится с помощью свойства:

```
public ImageFormat RawFormat {get;}
```

Кроме конструкторов, для создания растров можно использовать семейство статических методов FromXxx класса Bitmap. Они возвращают указатель на объект, который в конце работы необходимо удалять при помощи оператора delete.

Соответствующие статические методы имеются и у класса System.Drawing.Bitmap в среде Microsoft .NET Framework. Разумеется, в .NET вместо delete нужно использовать Dispose (а лучше даже пользоваться оператором using языка C#):

```
using(Bitmap bm = Bitmap.FromFile("photo.jpg"))
{
  // Используем картинку...
} // здесь bm будет освобожден
```

Немного подробнее оператор using и управление ресурсами в среде .NET рассматриваются в следующей главе.

Каждый кодек по мере возможностей учитывает специфические качества своего формата — например, загрузчик GIF правильно распознает прозрачность в GIF89a, чего так не хватало функции OleLoadPicturePath.

Имеется также возможность сохранения созданных растров в графических файлах различных форматов. Более подробно работа с кодеками будет рас-

сматриваться ниже, а пока речь пойдет о некоторых особенностях загрузки и отображения растров.

13.2.2. Загрузка из файлов и потоков (*IStream*)

Итак, для загрузки изображения из файла существует следующий конструктор:

```
Bitmap(const WCHAR* filename, BOOL useIcm);
```

Параметр `filename` должен содержать имя существующего файла. Все строковые параметры методов GDI+ требуют Unicode-строк, поэтому при передаче строковой константы в программе на C++ необходимо предварять ее префиксом 'L'.

Параметр `useIcm` определяет, будет ли при загрузке растра использоваться ICM (Image Color Management) API, и по умолчанию равен `FALSE`. Если же использовать ICM необходимо, графический файл должен содержать всю нужную информацию, например цветовые профили конкретных устройств.

Существует достаточно неприятная ошибка в коде GDI+, отвечающем за загрузку изображений из файлов: часто (но не всегда) при указании несуществующего имени файла вместо того, чтобы вернуть код ошибки в переменной `Status`, приложение завершается с выдачей сообщения об ошибке (уродливое белое прямоугольное окно Application Error).

Microsoft признает наличие этой проблемы, и в скором времени планируется выход заплатки (и очередной статьи в Knowledge Base). Сейчас единственный разумный способ избежать такого исхода — это убедиться в существовании файла перед попыткой его открытия.

Предупреждение

В коде GDI+ существует еще одна ошибка, связанная с тем, что некоторые цифровые фотокамеры не записывают в сохраняемый фотоснимок TIFF его физические размеры. Такой файл прекрасно прочитается кодеком TIFF, и при везении даже нарисуется — если явно задать пиксельные размеры получаемого рисунка. В противном случае GDI+ услужливо попытается рассчитать их на основании физических размеров — и работа приложения завершится с такой же "диагностикой". Будьте бдительны!

Более гибкие возможности загрузки таятся в таком конструкторе `Bitmap`:

```
Bitmap(IStream* stream, BOOL useIcm);
```

Он позволяет читать изображение из любого источника, поддерживающего интерфейс `IStream`. Это дает возможность загружать изображения из баз данных, структурированных хранилищ и блоков памяти и т. п. При этом также анализируется формат данных и для распаковки используется нужный кодек.

13.2.3. Создание растров из ресурсов программы

Следующий конструктор Bitmap позволяет загрузить растр из ресурсов:

```
Bitmap(HINSTANCE hInstance, const WCHAR* bitmapName);
```

Но не обольщайтесь, далеко не всякий ресурс удастся загрузить таким образом. Этот конструктор предназначен для загрузки именно BITMAP-ресурсов (которые описаны в *главе 10*) и не поддерживает, скажем, загрузку GIF. Возможно, это ограничения текущей реализации. К счастью, их достаточно легко обойти: просто предоставьте загрузчику интерфейс `IStream` с необходимыми данными. В листинге 13.2 показано, как это можно сделать, воспользовавшись функцией Windows API `CreateStreamOnHGlobal`:

Листинг 13.2. Загрузка растров из ресурсов Windows-программы [C++]

```
Bitmap* BitmapFromResource(HINSTANCE hInstance,
  LPCTSTR szResName, LPCTSTR szResType)
{
  HRSRC hrsrc = FindResource(hInstance, szResName, szResType);
  if(!hrsrc) return 0;
  // "ненастоящий" HGLOBAL - см. описание LoadResource в MSDN
  HGLOBAL hgTemp = LoadResource(hInstance, hrsrc);
  DWORD sz = SizeofResource(hInstance, hrsrc);
  void* ptrRes = LockResource(hgTemp);
  HGLOBAL hgRes = GlobalAlloc(GMEM_MOVEABLE, sz);
  if(!hgRes) return 0;
  void* ptrMem = GlobalLock(hgRes);
  // Копируем растровые данные
  CopyMemory(ptrMem, ptrRes, sz);
  GlobalUnlock(hgRes);
  IStream *pStream;
  // TRUE означает освободить память при последнем Release
  HRESULT hr = CreateStreamOnHGlobal(hgRes, TRUE, &pStream);
  if(FAILED(hr))
  {
    GlobalFree(hgRes);
    return 0;
  }
  // Используем загрузку из IStream
  Bitmap *image = Bitmap::FromStream(pStream);
```

```
  pStream->Release();
  return image;
}
```

Примечание

Вызов `Release` в вышеприведенном примере не приведет к немедленному уничтожению потока `IStream`. При загрузке из файлов и потоков объекты `Bitmap` удерживают источник данных в течение своей жизни и "отпускают" его только в деструкторе. Если об этом забыть, потом можно долго удивляться, почему к графическому файлу нет доступа после чтения его в `Bitmap`.

При загрузке графических ресурсов в .NET-приложении, можно использовать загрузку из объекта `System.IO.Stream`. Получить экземпляр объекта Stream из ресурса позволяет следующий метод класса `System.Reflection.Assembly`:

```
public virtual Stream GetManifestResourceStream(string name);
```

Здесь `name` — имя, под которым ресурс хранится в сборке (assembly).

В таком случае загрузка изображения из ресурсов будет выглядеть примерно так (листинг 13.3):

Листинг 13.3. Загрузка растров из ресурсов .NET-программы [C#]

```
Bitmap bmp = new Bitmap(
  Assembly.GetExecutingAssembly().
  GetManifestResourceStream("Demo.MyPicture.jpg"));
```

Совет

Новичков часто ставит в тупик необходимость поместить какой-нибудь файл в ресурсы исполняемого .NET-приложения. Между тем с помощью Visual Studio.NET это делается тривиально: просто добавьте файл в проект приложения, активируйте панель свойств (**Properties**) и измените у файла свойство **Build Action**, выбрав для него из списка значение **Embedded Resource**.

При таком способе добавления файла в ресурсы приложения ресурсу дается имя, состоящее из имени файла и префикса пространства имен (в данном случае **Demo**). Если же ресурс помещается в модуль с помощью утилит командной строки, вы можете сами определить имя, под которым он будет храниться в программе.

13.2.4. Более сложные варианты загрузки изображений

Помимо загрузки растровых изображений стандартных форматов из файлов или потоков, изображение можно создавать "на лету" или конвертировать из

других форматов, например объектов GDI или поверхностей DirectDraw. Вот соответствующие конструкторы:

```
Bitmap(const BITMAPINFO* gdiBitmapInfo, VOID* gdiBitmapData);
Bitmap(HBITMAP hbm, HPALETTE hpal);
Bitmap(HICON hicon);
Bitmap(IDirectDrawSurface7 * surface);
```

Важно понимать, что все эти конструкторы создают копию растровых данных. При любых модификациях объекта `Bitmap` (например, с помощью отрисовки в объект `Graphics`, созданный на его базе), источник данных остается неизменным. Более того, перенос изменений обратно в исходный объект GDI часто сопряжен с трудностями — например, может потребоваться самостоятельная работа с палитрами и битовыми массивами. Некоторые рекомендации по этому поводу можно встретить ниже, *в разд. 13.6.*

Отметим, что, на наш взгляд, в GDI+ было уделено большое внимание обратной совместимости. Вместе с тем эта совместимость в значительной степени реализована в режиме "только для чтения". Так, например, библиотека корректно прочитает из памяти растр с заголовком `BITMAPV4HEADER`, содержащим информацию об альфа-канале, но при сохранении в BMP из GDI+ будет сгенерирован только заголовок `BITMAPINFOHEADER`, и вся информация о прозрачности пропадет.

Примечание

Не стоит смешивать функциональность кодеков GDI+ и возможности работы с JPEG, которые появились в Windows 98/2000 — они полностью независимы друг от друга. Это поначалу может привести к путанице. В частности, если попытаться создать объект `Bitmap` из структуры `BITMAPINFO`, в которой поле `bmiHeader.biCompression` содержит значение `BI_JPEG`, ничего не выйдет — потому что загрузчик из растровых данных GDI+ поддерживает только простые форматы (вроде `BI_RGB` и `BI_BITFIELDS`). Для загрузки JPEG-изображений просто создайте `IStream` на блоке данных JPEG и вызовите надлежащий конструктор класса `Bitmap`.

Что же касается сохранения растров с прозрачностью, к вашим услугам форматы TIFF, PNG и GIF (об этом речь пойдет далее).

Существуют также конструкторы, позволяющие задать формат создаваемого растра:

```
Bitmap(INT width, INT height, PixelFormat format);
```

Перечисление `PixelFormat` содержит большое количество констант, определяющих растровые форматы. По умолчанию параметр `format` имеет значение `PixelFormat32bppARGB`. Наиболее быстрым для вывода является формат `PixelFormat32bppPARGB`. В этом формате каждая цветовая состав-

ляющая уже умножена на величину Alpha этого же пиксела, что избавляет от этого умножения (на каждый пиксел!) при накладывании изображения.

Можно самостоятельно выделить буфер для создаваемого растра. Вот соответствующий конструктор:

```
Bitmap(INT width, INT height, INT stride,PixelFormat format, BYTE* scan0 );
```

Здесь `stride` — величина смещения (в байтах) между концом одной строки растра и началом другой, `scan0` — указатель на массив байтов, содержащий растровые данные. Невзирая на формат, величина `stride` должна делиться без остатка на 4 (и, естественно, быть достаточно большой для хранения строки необходимой ширины). Она может также быть отрицательной (для создания растров с обратным порядком следования строк).

Смысл таких сложных манипуляций в том, что указание величины stride позволяет работать с прямоугольным фрагментом бóльшего растра, как с отдельным растром (см. схему на рис. 13.1).

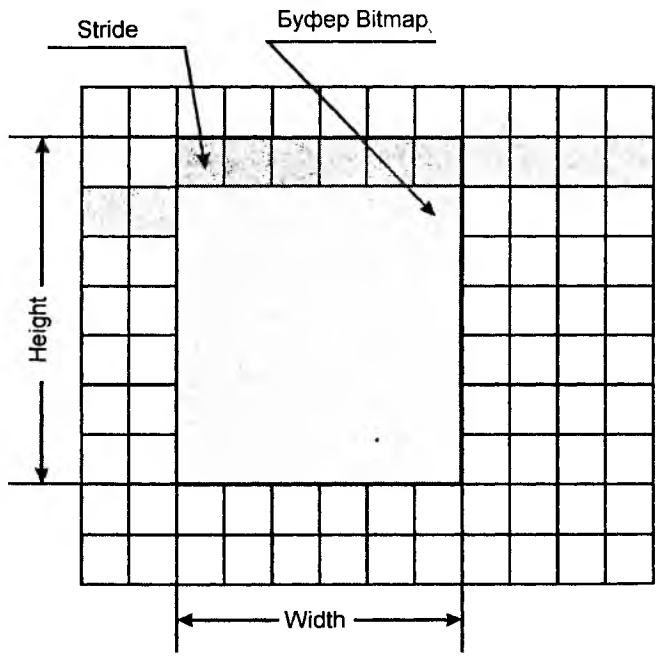


Рис. 13.1. Загрузка прямоугольного фрагмента растра с использованием величины `stride`

После уничтожения такого объекта `Bitmap` выделенный буфер не удаляется — ответственность за его освобождение лежит на программисте.

13.3. Графические форматы файлов

13.3.1. Лирическое отступление: 4 основных графических формата

Для хранения сжатых растровых данных используются в основном форматы JPEG (Joint Photographers Expert Group), GIF (Graphics Interchange Format), PNG (Portable Network Graphics) и TIFF (Tagged Image File Format). Люди, работающие в сфере полиграфии, имеют собственное мнение относительно форматов изображений и особенно не любят JPEG — за сжатие с потерями, хотя оно и позволяет достичь больших степеней компрессии. Не будем ввязываться в религиозный спор о форматах, а лучше сравним их возможности, сведенные для наглядности вместе в табл. 13.1.

Таблица 13.1. *Характеристики графических форматов*

Свойство	JPEG	GIF	PNG	TIFF
Полноцветные (True color) изображения	Поддерживает	Не поддерживает	Поддерживает	Поддерживает
Хранение нескольких кадров в одном файле	Не поддерживает	Поддерживает	Не поддерживает	Поддерживает
Прозрачность (color key)	Не поддерживает	Поддерживает	Поддерживает[1]	Не поддерживает
Прозрачность (альфа-канал)	Не поддерживает	Не поддерживает	Поддерживает	Поддерживает (в 32-битных форматах)
Сжатие без потерь качества	Не поддерживает	Поддерживает	Поддерживает	Поддерживает

В результате невольно вспоминается анекдот советских времен про умного, честного и партийного: одновременно можно выбрать только два качества из трех. Универсального формата не существует, и в каждом конкретном случае приходится идти на компромисс — выбор за вами.

---

[1] PNG является, пожалуй, уникальным форматом в сфере поддержки прозрачности. В этом формате можно сохранять полупрозрачные индексные изображения, grayscale-изображения с альфа-каналом и т. д. Однако полностью эти возможности в продуктах Microsoft (и, в частности, в GDI+) не реализованы.

Кратко остановимся на особенностях GIF. Этот формат является индексным, то есть цвет каждого пиксела в нем определяется его индексом в таблице цветов (палитре). Размер индекса ограничен величиной 8 бит, поэтому изображение может содержать не более 256 цветов одновременно. В этой модели реализована прозрачность по цветовому ключу (color key), то есть один из цветов в палитре можно назначить прозрачным. При загрузке GIF-файлов кодеком GDI+ они преобразуются в родной для этой библиотеки 32-битный растр. Пикселы прозрачного цвета при этом просто получают значение Alpha, равное 0.

Кроме того, напомним, что формат GIF имеет лицензионные ограничения: в нем используется компрессия LZW (Lempel-Ziv-Welch), патентом на которую до 20 июня 2003 года владела Unisys Corporation. В данный момент срок действия патента на территории США истек, но он все еще действует в Канаде, Великобритании, Франции, Германии и Италии. Официальную информацию по этому вопросу можно найти по адресу: **http://www.unisys.com/about__unisys/lzw/**.

Разработчики, использующие в своих продуктах библиотеки работы с LZW, должны платить отчисления Unisys, даже если используется библиотека GDI+ (подробнее см. в статье Q193543 INFO: Unisys GIF and LZW Technology License Information).

То же самое относится и к файлам формата TIFF, если для их компрессии выбран метод LZW. TIFF является контейнерным форматом и технически может содержать несколько кадров, сжатых различными методами (в том числе и JPEG).

13.3.2. Работа со списком кодеков

У каждого кодека GDI+ имеется уникальный CLSID — как и у COM-объектов. Чтобы получить список доступных кодеков, необходимо вызвать функции `GetImageDecoders` (она вернет список фильтров импорта) и `GetImageEncoders` (для получения фильтров сохранения). Эти функции заполняют переданный им буфер размером `size` байт массивом структур `ImageCodecInfo` (листинг 13.4):

Листинг 13.4. Класс ImageCodecInfo [C++]

```
class ImageCodecInfo
{
public:
  CLSID Clsid;
  GUID  FormatID;
  const WCHAR* CodecName;
```

```
  const WCHAR* DllName;
  const WCHAR* FormatDescription;
  const WCHAR* FilenameExtension;
  const WCHAR* MimeType;
  DWORD Flags;
  DWORD Version;
  DWORD SigCount;
  DWORD SigSize;
  const BYTE* SigPattern;
  const BYTE* SigMask;
};
```

Для просмотра свойств установленных в системе кодеков GDI+ мы создали утилиту на C++ с использованием MFC. Ее исходный код и откомпилированный вариант можно найти на компакт-диске, прилагающемся к книге.

Codecs.zip — проект с исходным кодом, Codecs_EXE.zip — готовое приложение.

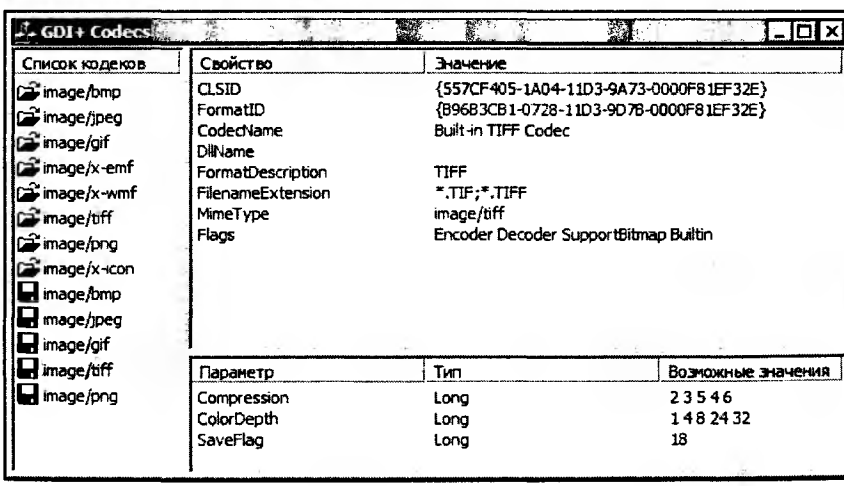


Рис. 13.2. Фрагмент вывода утилиты Codecs

Программа отображает список имеющихся в GDI+ компрессоров/декомпрессоров и позволяет просмотреть свойства каждого кодека, а для кодеков сохранения — и поддерживаемые параметры.

Первый же запуск программы дал интересные результаты. Например, несмотря на наличие двух раздельных списков — для загрузки и для сохранения — в них перечислены одни и те же кодеки (достаточно сравнить их

с CLSID). Это, конечно же, деталь реализации, которая может измениться в будущих версиях GDI+.

Немедленно возникает вопрос: а можно ли добавить в этот список свои кодеки, чтобы иметь возможность загружать файлы других форматов при помощи GDI+? Например, поле `Flags` содержит бит `Builtin`, который, судя по названию, должен показывать, является ли кодек встроенным. Увы, документированного способа пока не существует: в GDI+ версии 1.0 кодеки не являются объектами ActiveX, а встроены в код библиотеки. В следующей версии Microsoft обещает добавить поддержку внешних кодеков.

13.3.3. Сохранение изображений

При сохранении необходимо указать, кодек какого формата необходимо использовать. Функция `Image::Save` принимает Unicode-строку `filename` и указатель на GUID кодека `clsidEncoder`:

```
Status Save(
  const WCHAR* filename,
  const CLSID* clsidEncoder,
  const EncoderParameters* encoderParams
);
```

Последний параметр, `encoderParams`, представляет собой указатель на структуру `EncoderParameters` с различными настройками кодека сохранения: степенью сжатия, глубиной цвета и т. д.

В документации Platform SDK подробно описано назначение настроек при работе с различными кодеками. Для быстрого освоения допустимых значений настроек конкретного установленного кодека вы можете воспользоваться приведенной программой. Например, согласно ей для кодека JPEG параметр `Transformation` может принимать следующие значения: 13, 14, 15, 16, 17. Им соответствуют следующие значения констант из GdiPlusEnums.h:

```
EncoderValueTransformRotate90,
EncoderValueTransformRotate180,
EncoderValueTransformRotate270,
EncoderValueTransformFlipHorizontal,
EncoderValueTransformFlipVertical
```

Эти константы описывают допустимые геометрические преобразования при сохранении файлов формата JPEG. Кстати, в GDI+ реализована полезная особенность: если загрузить файл JPEG с размерами, кратными 16, то при сохранении его с одним из вышеперечисленных преобразований дополнительной потери качества не произойдет.

А откуда взять значение CLSID для выбранного кодека? Можно, конечно, жестко завести в код константу, взяв ее, например, из окна приведенной

программы. Но этот подход оправдан, только если приложение будет распространяться именно с этой версией библиотеки GDI+ — иначе GUID кодеков могут измениться. Microsoft рекомендует более гибкий способ.

Обычно для получения CLSID для кодека, скажем, JPEG, формируют строку вида "image/jpeg" и выполняют в этом списке поиск кодека, имеющего такую строку в поле `MimeType`. Реализация этого подхода имеется в документации — в разделе "Retrieving the Class Identifier for an Encoder" приведен исходный код функции `GetEncoderClsid`.

Если выбор кодека приходится делать несколько раз, то предпочтительнее будет другой метод — заполнение ассоциативного контейнера (например, `std::map`) списком кодеков и последующая выборка кодека по GUID графического формата (поле `FormatID`), с которым мы уже сталкивались, обсуждая метод `GetRawFormat`. Этот подход проиллюстрирован в листинге 13.5:

Листинг 13.5. Функция для подготовки быстрого поиска кодеков [C++]

```
// тип ассоциативного массива, хранящего пары:
// FormatID - CLSID кодека.
typedef std::map<GUID, GUID> CodecsList;

// Определяем оператор «<» для сравнения GUID.
bool operator<(REFGUID g1, REFGUID g2)
{
  return memcmp(&g1, &g2, sizeof(GUID))<0;
}
// Функция для чтения списка кодеков.
// Параметр Encoders указывает, какой список кодеков необходим:
// для сохранения или чтения.
CodecsList ReadCodecsList(bool Encoders)
{
  using namespace Gdiplus;
  UINT  num, size;
  if(Encoders)
  GetImageEncodersSize(&num, &size);
  else
  GetImageDecodersSize(&num, &size);

  // размер буфера - в байтах!
  ImageCodecInfo* pArray = (ImageCodecInfo*)(malloc(size));
  if(Encoders)
```

```
  GetImageEncoders(num, size, pArray);
  else
  GetImageDecoders(num, size, pArray);

  // заполняем map
  CodecsList codecs;
  for(UINT j = 0; j < num; ++j)
  codecs[pArray[j].FormatID]=pArray[j].Clsid;

  free(pArray);
  return codecs;
}
```

Тогда преобразование из PNG в JPEG будет выглядеть примерно так:

```
Bitmap bm(L"test.png");
CodecsList codecsList = ReadCodecsList(true);
GUID JpegId = codecsList[ImageFormatJPEG];
... // работаем с растром
bm.Save(L"test2.jpg", JpegId); // см. предупреждение
```

.NET-программистам в данном случае можно расслабиться: такая функциональность уже реализована за них. Вся черная работа по сопоставлению `FormatID` и `CLSID` будет проделана в недрах библиотеки .NET (в методе `FindEncoder`). В метод же `Save` достаточно передать именованное свойство класса `System.Drawing.Imaging.ImageFormat`, содержащее значение `FormatID` соответствующего кодека!

```
Bitmap bm = new Bitmap("test.png");
... // работаем с растром
bm.Save("test2.jpg", ImageFormat.Jpeg);
```

Предупреждение

Метод Save() не сработает, если ему будет передано имя файла, уже открытого средствами GDI+. Это происходит потому, что файл остается занятым вплоть до выполнения деструктора Bitmap.

Другая проблема возникнет, если файл доступен для записи, но имеет длину, превышающую результирующий размер изображения. `Save()` не производит усечения, и в конце файла останется "мусор".

Эти проблемы известны Microsoft. Вторая уже нашла отражение в Knowledge Base: PRB: Save Method of Bitmap Class Does Not Truncate File Size (Q312119).

Практический пример: сохранение содержимого экрана в графическом файле

В *главе 10* мы научились самостоятельно создавать на диске файлы BMP. У этого формата, как известно, существует один крупный недостаток: размер. Как говорится в шутливой задачке Юрия Нестеренко, "сколько лет будет девушке Маше, когда пользователь Вася докачает ее фотку?". В задачке шла речь о файле размером в 15 Мбайт — а ведь это размер стандартного True-Color-изображения из обычной 5-мегапиксельной цифровой камеры. В полиграфии бывают файлы и побольше.

Вообще-то дело, конечно, не в формате, а в способе сжатия. Давайте испытаем магию GDI+ на практике: напишем программу на C++, сохраняющую видимое содержимое экрана в форматах BMP и PNG (который довольно эффективно сжимает графические файлы без потерь качества). (Листинг 13.6.) Для эффективного доступа к пикселам экрана нам придется воспользоваться GDI. А для того чтобы избавиться от возни с инициализацией GDI+, мы применим описанный в *главе 12* класс `InitGdiPlus`. Для компиляции примера укажите компилятору кроме исходного файла имя InitGdiPlus.cpp:

Листинг 13.6. Сохранение содержимого экрана в форматах BMP и PNG

```
// Компиляция:
// cl grab.cpp InitGdiPlus.cpp user32.lib kernel32.lib gdi32.lib
// для пользователей VC7: добавьте ключ /link /delayload:gdiplus.dll

#define STRICT
#include <windows.h>
#include <GdiPlus.h>
#include "InitGdiPlus.h"

//Внимание: в будущих версиях GDI+ GUID- кодеков могут измениться
static const GUID guidPng =
{ 0x557cf406, 0x1a04, 0x11d3, { 0x9a, 0x73, 0x00, 0x00, 0xf8, 0x1e, 0xf3,
0x2e } };

static const GUID guidBmp =
{ 0x557cf400, 0x1a04, 0x11d3, { 0x9a, 0x73, 0x00, 0x00, 0xf8, 0x1e, 0xf3,
0x2e } };

int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine, int nShowCmd )
```

```
{
  HDC hdc = ::GetDC(NULL);
  RECT rc={0, 0, GetSystemMetrics(SM_CXFULLSCREEN),
    GetSystemMetrics(SM_CYFULLSCREEN)};
  HDC hdcMem = CreateCompatibleDC(hdc);

  HBITMAP hBitmap = CreateCompatibleBitmap(hdc, (int)rc.right,
    (int)rc.bottom);
  HBITMAP hOldBmp = (HBITMAP)SelectObject(hdcMem, hBitmap);
  BitBlt(hdcMem, 0, 0, (int)rc.right, (int)rc.bottom, hdc,
    0, 0, SRCCOPY);
  hBitmap =(HBITMAP) SelectObject(hdcMem, hOldBmp);
  DeleteDC(hdcMem);
  RSDN::InitGdiPlus init;
  if(init.Good())
  {
    Gdiplus::Bitmap bitmap(hBitmap, NULL);
    bitmap.Save(L"c:\\screen.png", &guidPng);
    bitmap.Save(L"c:\\screen.bmp", &guidBmp);
  }
  else
    MessageBox(0, "Ошибка инициализации GDI+", 0, MB_OK|MB_ICONHAND);
  DeleteObject(hBitmap);
  return 0;
}
```

Как видите, собственно сохранение файла занимает всего одну строчку кода. Мы не стали использовать описанные выше хитроумные способы поиска кодеков в таблице, а просто жестко ввели в код значения, взятые из окна программы Codecs.

Полученная утилита имеет вполне разумное применение: она позволяет создавать снимки экрана монитора — например, для написания книг по компьютерной тематике. Надеемся, она пригодится и вам.

Интересно также сравнить размеры полученных файлов. При разрешении экрана 1024 × 768 пикселов и режиме экрана TrueColor файл формата BMP получился размером почти в 3 Мбайта. Сжатый же PNG такого же разрешения имел размер 68 Кбайт, или в 45 раз меньше! Конечно, степень сжатия будет тем хуже, чем сложнее выводимое на экран изображение.

13.4. Специфические возможности файловых форматов

В этом разделе мы рассмотрим дополнительные возможности, предоставляемые различными файловыми кодеками GDI+ (и научимся обходить некоторые проблемы).

13.4.1. Сохранение GIF с прозрачностью

Это наиболее популярная тема обсуждений в новостных группах, связанных с графическими форматами GDI+. Действительно, в Web GIF-файлы с прозрачными областями приобрели чрезвычайное распространение. Загрузка прозрачного GIF-файла не составляет никакого труда — кодек сам корректно распознает прозрачные области и устанавливает у них нулевую величину альфа-канала.

Однако это и служит причиной того, что при сохранении возникают проблемы. Все дело как раз в том, что при загрузке файлы формата GIF преобразуются в 32-битный формат. Для сохранения же GIF необходима палитра. Кодек умеет сохранять такие растры только в так называемой Halftone palette — некой известной Microsoft стандартной таблице цветов. Нужные цвета при этом подгоняются смешиванием с соседними точками. Картинка начинает выглядеть довольно уродливо, и ни о какой прозрачности речи не идет.

Если вам нужно сохранять растры с прозрачностью без дополнительных сложностей, воспользуйтесь 32-битным форматом (PNG или TIFF) с альфа-каналом[1].

Тем не менее средствами GDI+ создать GIF с прозрачностью можно. Это потребует прямой работы с битами изображения, так как изображение должно будет содержать не более 256 цветов и оставаться в индексном формате. Только тогда кодек распознает первый цвет в таблице цветов, содержащий 0 в поле Alpha, как прозрачный, и правильно сохранит файл. В соответствующих статьях Knowledge Base содержится полное описание технологии такого процесса:

INFO: GDI+ GIF Files Are Saved Using the 8-Bpp Format (Q318343)

HOW TO: Save a .gif File with a New Color Table By Using GDI+ (Q315780)

HOW TO: Save a .gif File with a New Color Table By Using Visual C# .NET (Q319061)

---

[1] Здесь вас подстерегает другая неприятность: как уже говорилось, относительно новый формат PNG некорректно отображается многими браузерами, в частности Internet Explorer не понимает PNG с альфа-каналом.

13.4.2. Загрузка и сохранение многокадровых файлов

Некоторые графические форматы поддерживают хранение нескольких картинок в одном файле. Скажем, файл формата TIFF может содержать несколько изображений одной страницы, отсканированных с различным разрешением.

Другая причина существования многокадровых форматов — анимация. Те подвижные картинки, которые мелькают на миллионах Web-страниц, чаще всего являются анимированными GIF-изображениями.

Раньше было довольно сложно загрузить с диска такой файл, не прибегая к разбору его формата на низком уровне. Например, функция API `OleLoadPicture` не могла самостоятельно загружать анимированные файлы GIF. Теперь вся работа выполняется кодеком GDI+ и загрузка многокадровой картинки ничем не отличается от обычной загрузки графического файла. Узнать число кадров загруженного растра можно с помощью функции `Image::GetFrameCount`:

```
UINT GetFrameCount(
  const GUID* dimensionID
);
```

В качестве параметра она принимает указатель на GUID-константу, определяющую тип хранимых кадров. Возможные значения описаны в заголовочных файлах GDI+. Для GIF необходимо передавать константу `FrameDimensionTime`, а для файлов TIFF — `FrameDimensionPage`:

```
frameCount = bitmap.GetFrameCount(&FrameDimensionTime);
```

Перед отрисовкой нужного кадра его необходимо сделать активным с помощью функции `Image::SelectActiveFrame`. Подробности можно найти в документации и исходном коде демонстрационных приложений к статье.

С сохранением дело обстоит немного хуже. В GDI+ версии 1.0 многокадровое сохранение реализовано только для файлов формата TIFF. Анимированные GIF таким образом сохранить, к сожалению, не удастся.

Итак, рассмотрим процедуру создания файла TIFF с несколькими кадрами:

1. Создать объект `Image`, который будет дирижировать процессом сохранения. Он должен содержать первую картинку многокадрового изображения.
2. Сохранить первый кадр, используя метод `Image::Save`. Этот шаг ничем не отличается от описанного выше варианта с единственным кадром.
3. Для сохранения последующих картинок в том же файле каждый раз вызывать у главного объекта `Image` метод `SaveAdd`, передавая ему необходимый кадр.

Примечание

Здесь возможны варианты. Если все необходимые кадры находятся в исходном объекте Image, нужно просто последовательно делать их текущими (с помощью SelectActiveFrame) и вызывать вариант метода SaveAdd, принимающего только указатель на структуру EncoderParameters.

Можно также добавлять в сохраняемый файл кадры из других растров, вызывая метод SaveAdd, принимающий наряду с параметрами указатель на объект класса Image.

4. Для завершения процесса нужно вызвать метод SaveAdd с особым параметром сохранения EncoderValueFlush (см. листинг 13.7):

Листинг 13.7. Завершение сохранения многокадрового файла [C++]

```
ULONG value;
EncoderParameters params={1}; // один параметр
params.Parameter[0].NumberOfValues = 1;
params.Parameter[0].Guid  = EncoderSaveFlag;
params.Parameter[0].Type  = EncoderParameterValueTypeLong;
params.Parameter[0].Value = &value;
... // сохранение кадров
value = EncoderValueFlush; // завершить последовательность
myImage.SaveAdd(&encoderParameters);
```

13.4.3. Эскизы изображений

GDI+ предоставляет очень удобный механизм для создания *эскизов (иконок) изображений* (thumbnail images), которые содержат миниатюрную копию картинки. Поскольку эскизы имеют малые размеры и быстро отображаются, их используют для предварительного просмотра изображений на Web-страницах, в приложениях просмотра картинок (типа ACDSee) и т. д.

Вообще-то у нас уже есть все необходимое для создания эскизов. Нужно всего лишь создать Bitmap нужных размеров, инициализировать устройство вывода (Graphics) в этот растр и вывести картинку каким-либо приемлемым способом (например, установив режим вывода InterpolationModeHighQualityBicubic, о чем речь пойдет позже). Однако создателям библиотеки этого показалось мало, и они добавили в класс Image метод GetThumbnailImage, который выполняет всю эту работу:

```
Image* GetThumbnailImage(
  UINT thumbWidth,
  UINT thumbHeight,
  GetThumbnailImageAbort callback,
```

```
  VOID* callbackData
);
```

Параметры `thumbWidth` и `thumbHeight` сообщают этой функции требуемые размеры иконки. Параметры `callback` и `callbackData` используются, если нужна возможность прерывания процесса создания эскиза. По умолчанию они равны `NULL`, и эта возможность игнорируется.

Каковы же преимущества этого способа? На первый взгляд, только простота (и, следовательно, ограниченные возможности). На самом же деле `GetThumbnailImage()` является наиболее быстрым способом получения иконок. В большинстве графических форматов предусмотрено хранение миниатюры вместе с оригинальным файлом. Если передать в параметрах `thumbWidth` и `thumbHeight` значение 0, кодек формата постарается извлечь именно эту миниатюру (разумеется, при использовании неподходящего формата GDI+ придется создавать ее самостоятельно). Кроме того, при создании заведомо маленького изображения возможны другие оптимизации, которыми не обладает метод `DrawImage`.

Аналогичный метод имеется и у класса `System.Drawing.Image` в .NET Framework. Единственная разница в том, что вместо указателя на функцию `GetThumbnailImageAbort` в .NET необходимо передавать делегат `Image.GetThumbnailImageAbort`:

```
public Image GetThumbnailImage(
   int thumbWidth,
   int thumbHeight,
   Image.GetThumbnailImageAbort callback,
   IntPtr callbackData
);
```

Так как пример использования этой функции на C# довольно компактен, приведем код приложения целиком:

Листинг 13.8. Программа для создания миниатюр графических файлов [C#]

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;

class ThumbnailExtractor
{
  public static void Main(string[] args)
  {
```

```
    foreach(string arg in args) // перебор элементов массива
      SaveThumbnail(arg);
  }

  static void SaveThumbnail(string name)
  {
    try
    {
      Bitmap bm = new Bitmap(name);
      Image thumb = bm.GetThumbnailImage(0, 0, null, IntPtr.Zero);
      thumb.Save(name + "_tn.jpg", ImageFormat.Jpeg);
    }
    // возникла проблема?
    catch(Exception e)
    {
      Console.WriteLine(e); //неявно вызываем e.ToString()
    }
  }
};
```

Эта программа создает (или извлекает) эскизы всех файлов, указанных в командной строке, и сохраняет их в формате JPEG в том же каталоге, с добавлением суффикса к имени файла.

Примечание

В MSDN сказано, что вы должны создать экземпляр делегата `Image.GetThumbnailImageAbort`, даже если не хотите его использовать (указав на функцию-заглушку). Однако приведенный выше код вполне работает, передавая вместо делегата `Null`.

13.4.4. Работа с метаданными изображений

Многие приложения сохраняют в графических файлах дополнительную информацию. Весь этот набор необязательных полей принято называть модным сейчас словечком "метаданные" (действительно, в чем-то они сходны с метаданными, хранящимися в сборках .NET). В их число может входить дата съемки, информация об авторских правах на изображение, параметры разрешения и т. д. Возможность их хранения предусмотрена практически во всех графических форматах.

GDI+ упрощает работу с метаданными, абстрагируясь от деталей различных форматов и предоставляя достаточно гибкий и удобный механизм доступа.

Для этого реализован класс PropertyItem, представляющий собой обертку для любого типа данных, хранимого в файле:

```
class PropertyItem
{
public:
  PROPID  id;     // Уникальный номер
  ULONG   length; // Длина поля (в байтах)
  WORD    type;   // Тип значения (один из типов PropertyTagTypeXXX)
  VOID*   value;  // Указатель на данные
};
```

Значение поля id определяет вид хранимой информации. Каждому графическому формату соответствует свой набор возможных ID, описанных в заголовочном файле GdiPlusImaging.h.

Для демонстрации работы с метаданными приведем небольшой пример кода на C#, в результате выполнения которого выводится перечень информационных полей для заданного графического файла (листинг 13.9):

Листинг 13.9. Перечисление метаданных изображения [C#]

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;

class ImageProperties
{
  public static void Main(string[] args)
  {
    foreach(string arg in args)
    {
      Bitmap bm=new Bitmap(arg);
      foreach(PropertyItem i in bm.PropertyItems)
        Console.WriteLine("Property #{0}: type={1}, len={2}",
          i.Id, i.Type, i.Len);
    }
  }
};
```

13.5. Использование растров при работе с объектом *Graphics*

Текущая реализация библиотеки оперирует 32-битным цветом при любых цветовых вычислениях. Растры с другим представлением цвета перед обработкой приводятся к 32-битному формату. Таким образом, хороший способ экономить ресурсы процессора и память системы — это сразу работать с 32-битными растрами (что и предлагается по умолчанию). Например, при создании экземпляра `Graphics` из контекста устройства обычного GDI (HDC) создается промежуточный 32-битный буфер, в который и осуществляется вывод, а уже затем производится копирование этого буфера в контекст. Если контекст устройства имеет другую глубину цвета, понадобится дополнительное преобразование, что плохо скажется на производительности. Кроме того, такая организация работы с графическими контекстами диктует свои ограничения на обращение к контексту средствами GDI — во время существования `Graphics` этого делать нельзя. Для подробного ознакомления с этой проблемой авторы рекомендуют статью Q311221.

13.5.1. Вывод изображений и геометрические преобразования

Поддержка координатных преобразований в GDI+ — слишком большая тема, чтобы касаться ее в разговоре о растровой графике (мы обсудим координатные преобразования в *главе 14*). Однако эта библиотека предоставляет также специальные средства геометрических преобразований при выводе изображений, о которых сейчас пойдет речь.

Итак, в объекте `Graphics` для вывода растров предназначен метод `DrawImage`. Постойте, мы сказали "предназначен метод"? Правильнее будет применить множественное число: их 16 штук! По своему назначению эти перегруженные функции разделяются на две основные группы:

1. Вывод изображения или его части в прямоугольную область с возможностью масштабирования по двум осям. По своему действию эти методы похожи на функцию GDI `StretchBlt`, но с дополнительными возможностями — например, со сглаживанием результата. В качестве параметров эти методы получают в различных видах координаты прямоугольных областей источника и приемника:

```
void PaintFlower(Graphics& g, Rect& rc)
{
  ...
  g.DrawImage(flowerImage, flowerPos.X, flowerPos.Y,
  flowerImage->GetWidth(), flowerImage->GetHeight());
}
```

Примечание

Обратите внимание на выделенные параметры. Такой способ (явное указание размеров выводимого изображения) является предпочтительным, и вот почему.

Во-первых, уже упоминалось, что некоторые файлы не содержат информацию о разрешении. При выводе такого изображения без указания размеров библиотека GDI+ попытается вычислить их на основании разрешений растра и устройства вывода, и произойдет сбой.

Во-вторых, явно указав исходные пиксельные размеры растра, вы тем самым подсказываете GDI+, что масштабирования выполнять не нужно, и вывод произойдет быстрее.

2. Вывод изображения или его части в параллелограмм с соответствующим преобразованием координат всех точек исходного растра. В GDI отсутствует подобная функциональность (в семействе NT нечто подобное выполняет функция `PlgBlt`). Этим методам для работы требуется массив из трех точек, образующих вершины параллелограмма (четвертая вершина вычисляется на их основе).

Заметим, что частным случаем преобразования при последнем способе является и вращение — достаточно задать 3 координаты прямоугольника, повернутого на требуемый угол. Порядок точек в массиве должен соответствовать точкам A_1, B_1 и C_1 на приведенной схеме (рис. 13.3):

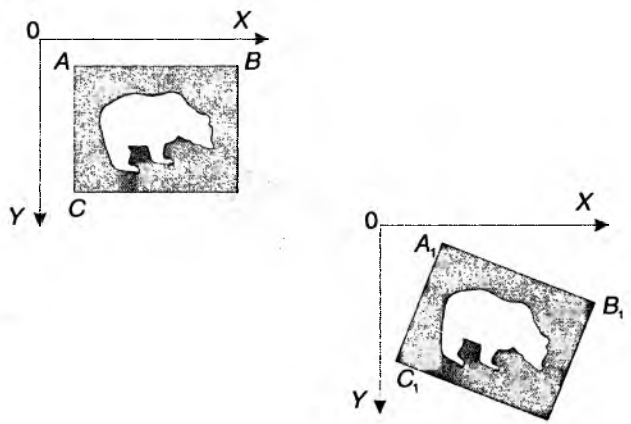


Рис. 13.3. Схема вывода растров с аффинными преобразованиями

Но в большинстве случаев для поворота выводимого растра проще будет применить координатные преобразования GDI+, рассмотренные в следующей главе.

Кроме того, для поворота изображений на угол, кратный 90 градусам, или их отражения относительно осей симметрии, в классе `Image` существует метод

```
Status RotateFlip(RotateFlipType rotateFlipType),
```

принимающий в качестве параметра элемент перечисления RotateFlipType. Все возможные варианты перечислений GDI+ находятся в файле GdiPlusEnums.h.

Для более гибкого контроля над выводом изображения существуют версии метода DrawImage, принимающие указатель на объект класса ImageAttributes. Большая часть задаваемых при этом параметров относится к цветовой коррекции, которая обсуждается ниже. Нас же сейчас может заинтересовать один метод этого класса, влияющий на геометрию выводимых изображений:

```
Status SetWrapMode( WrapMode wrap, const Color& color, BOOL clamp ).
```

Он позволяет указать, каким образом заполнять область вывода за пределами исходной картинки. Вид заполнения определяется параметром wrap из перечисления WrapMode. В частности, указав WrapModeTile, вы получите вывод мозаикой (tiling). При задании режима WrapModeClamp внешняя область заполняется цветом, указанным в параметре color. Параметр clamp в текущей реализации игнорируется.

Кстати, это справедливо не только для растров, но и для вывода объектов Metafile, которые также являются потомками класса Image.

Говоря о методе SetWrapMode класса ImageAttributes, нельзя не упомянуть о методах с таким же названием, присутствующих в классах LinearGradientBrush, PathGradientBrush и TextureBrush. Они играют аналогичную роль при задании параметров закраски областей.

13.5.2. Качество изображения

Увеличивая, растягивая и искажая исходную картинку, можно запросто получить в результате нечто безобразное — зубчатые края изображения, ступенчатые границы прямых линий Причина кроется в самой природе растровой графики: изображение несет в себе информацию, ограниченную выбранным разрешением.

Тем не менее существуют различные алгоритмы интерполяции, позволяющие добиться значительного улучшения качества результирующего изображения, или, точнее говоря, его качества с точки зрения человеческого восприятия. Некоторые из них изначально встроены в библиотеку GDI+.

Метод Graphics::SetInterpolationMode позволяет указать, какой режим (или алгоритм) интерполяции будет использован при выводе изображения с изменением его пиксельных размеров. Константы возможных режимов описаны в перечислении InterpolationMode.

В .NET-версии объекта Graphics имеется соответствующее свойство InterpolationMode.

Выигрыш в качестве в данном случае приводит к проигрышу в скорости, поэтому при использовании режима с наивысшим качеством

InterpolationModeHighQualityBicubic медленные компьютеры могут выводить изображения больших размеров в течение нескольких секунд[1]! Но только этот метод способен адекватно отображать картинку при уменьшении ее до 25 процентов (и менее) от оригинала. Этот режим очень поможет различным автоматическим генераторам иконок (*thumbnail*) изображений в ASP.NET.

На качество (и скорость) вывода растров также влияют некоторые другие установки объектов Graphics. Перечислим их с кратким описанием в табл. 13.2:

***Таблица 13.2.** Методы управления режимами качества в GDI+*

Метод	Назначение
SetSmoothingMode	Позволяет указать метод устранения ступенчатости (*antialiasing*) при выводе примитивов — линий и геометрических фигур
SetCompositingMode	Устанавливает или отключает учет прозрачности при наложении растровых изображений
SetCompositingQuality	Управляет качеством расчета цветовых компонентов при наложении растров
SetPixelOffsetMode	Задает метод учета смещения пикселов при интерполяции. Грубо говоря, определяет, являются ли координаты пикселов (или их центров) целыми числами при расчетах
SetRenderingOrigin	Устанавливает позицию начальной точки при псевдосмешении (*dithering*) цветов в 8- и 16-битных режимах

Для получения значений соответствующих настроек служат аналогичные методы с префиксом "Get". В среде Microsoft .NET Framework у класса Graphics существуют аналогичные свойства без префиксов вообще (SmoothingMode, PixelOffsetMode и т. д).

13.5.3. Устранение мерцания

Часто встречающейся проблемой при создании динамично меняющейся графики (в частности, анимации) является мерцание. Для его устранения традиционно использовалась двойная буферизация, и эта возможность также имеется в GDI+. Объект Graphics можно создать "теневым", используя

[1] Речь идет об экранном выводе, то есть умеренно больших изображений, так как бикубическая интерполяция изображений полиграфического разрешения может длиться минутами и на самых современных компьютерах.

в качестве основы готовый экземпляр Bitmap (вообще говоря, Image, но нас сейчас интересуют только растры):

```
Graphics( Image* image );
static Graphics* FromImage( Image* image );
```

При этом все операции вывода с участием такого объекта отразятся на содержимом используемого экземпляра Bitmap. Это предоставляет очень простую возможность двойной буферизации вывода — когда изображение сначала готовится "за кадром", а затем мгновенно (ну, *почти* мгновенно) переносится на экран, устраняя досадное мерцание при анимации (листинг 13.10):

Листинг 13.10. Вывод с двойной буферизацией в GDI+ [C++]

```
void OnPaint(HDC hdc, RECT& rc)
{
  Graphics g(hdc);
  Rect paintRect(0, 0, rc.right, rc.bottom);
  // Создаем временный буфер
  Bitmap backBuffer(rc.right, rc.bottom, &g);
  Graphics temp(&backBuffer);
  // Рисуем в буфер
  PaintBackground(temp, paintRect);
  PaintFlower(temp, paintRect);
  PaintBatterfly(temp, paintRect);
  // Переносим на экран
  g.DrawImage(&backBuffer, 0, 0, 0, 0,
  rc.right, rc.bottom, UnitPixel);
}
```

Для создания "теневого" Graphics подойдет далеко не всякий растр. В частности, это невозможно для индексных и Grayscale-растров (состоящих из градаций серого цвета). Это связано именно с ограничениями ядра GDI+.

Совет

Вы можете столкнуться с мерцанием даже используя двойную буферизацию. Дело в том, что при перерисовке окна ему сначала посылается сообщение WM_ERASEBKGND. Используемый по умолчанию обработчик этого сообщения закрашивает область, нуждающуюся в обновлении, кистью hbrBackground, указанной в свойствах класса окна. Для устранения такого рода мерцания необходимо, во-первых, перекрыть WM_ERASEBKGND или задать нулевую кисть hbrBackground при регистрации класса окна, а, во-вторых, позаботиться о собственном заполнении всего окна при рисовании.

Что касается WinForms, то там режим двойной буферизации уже предусмотрен. Для его включения необходимо в окне, в которое производится отрисовка (например, элементе управления или форме) установить флаги `UserPaint`, `AllPaintingInWmPaint` и `DoubleBuffer` перечисления `System.Windows.Forms.ControlStyles` (пример установки стилей приведен в листинге 13.11):

Листинг 13.11. Установка стилей для двойной буферизации WinForms [C#]

```
protected override void OnLoad(EventArgs e)
{
  SetStyle(ControlStyles.UserPaint, true);
  SetStyle(ControlStyles.AllPaintingInWmPaint, true);
  SetStyle(ControlStyles.DoubleBuffer, true);
  ... // другая инициализация
  base.OnLoad(e);
}
```

При этом, кстати, вывод довольно заметно ускоряется (несмотря на необходимость дополнительного переноса на экран) — наше демонстрационное приложение вместо 70 FPS (кадров в секунду) стало выдавать 75-80.

13.5.4. Несколько слов о производительности

Производительность в данный момент является "ахиллесовой пятой" библиотеки. Изначально GDI+ проектировалась с прицелом на аппаратную акселерацию, но в версии 1.0 она не реализована. Остается надеяться, что этот недостаток будет устранен в ближайшем будущем: ведь попиксельный расчет полупрозрачности и антиалиасинга не являются самыми быстрыми операциями.

Раз уж речь зашла о скорости, уместно будет сказать, что наиболее быстро GDI+ умеет выводить растры оптимизированного для устройства формата, представленные классом `CachedBitmap`. При их создании необходимо указать оригинальный растр и устройство, на которое будет происходить вывод изображения:

```
CachedBitmap(Bitmap* bitmap, Graphics* graphics);
```

На этот конструктор накладываются определенные ограничения. В частности, устройство вывода `Graphics` не должно быть связано с HDC принтера или метафайла. Далее, при смене характеристик устройства вывода (например, при изменении разрешения или глубины цвета экрана) `CachedBitmap` необходимо пересоздавать для работы с новым устройством, в противном случае вывод производиться не будет.

Для вывода оптимизированных растров на экран служит метод `Graphics::DrawCachedBitmap`:

```
Status DrawCachedBitmap( CachedBitmap* cb, INT x, INT y);
```

Этот метод ускоряет вывод не только на экран, но и в память за счет отказа от многих промежуточных вычислений.

Хорошая новость: `DrawCachedBitmap` поддерживает прозрачность и альфа-канал (проверено в лабораторных условиях).

Плохая новость: применение координатных преобразований к устройству вывода при этом не поддерживается (кроме координатного переноса). В частности, поворачивать такие растры при выводе, к сожалению, нельзя. Если это необходимо, примените технику промежуточного вывода в память с поворотом, а затем уже кэшируйте полученный `Bitmap`. Разумеется, этот прием не подойдет, если угол поворота меняется динамически (тогда лучше совсем отказаться от кэширования, чтобы не тратить каждый раз время на создание промежуточного изображения).

Совет

Как быть с кэшированием анимированных (многокадровых) изображений? Очень просто: вам потребуется ровно столько объектов `CachedBitmap`, сколько кадров содержится в анимации. А для того чтобы кэшировать очередной кадр, необходимо сделать его активным (как и перед выводом на экран) с помощью метода `Image::SelectActiveFrame`. Приведенный в листинге 13.12 код иллюстрирует эту технику.

Листинг 13.12. Пример кэширования многокадрового изображения GIF [C++]

```
// необходимые переменные
Bitmap *batterflyImage;        // анимированное изображение бабочки
CachedBitmap **cachedFrames;
// здесь будут размещаться кэшированные кадры
int frameCount;
int activeFrame;
...
// инициализация
frameCount = batterflyImage->GetFrameCount(&FrameDimensionTime);
cachedFrames = new CachedBitmap*[frameCount];

// создаем объект Graphics на базе окна десктопа
// и считаем, что текущий видеорежим не изменится
Graphics g(GetWindowDC(0));
```

```
// собственно кэширование
for(int i=0;i<frameCount;i++)
{
  batterflyImage->SelectActiveFrame(&FrameDimensionTime, i);
  cachedFrames[i] = new CachedBitmap(batterflyImage, &g);
}
activeFrame=0;
batterflyImage->SelectActiveFrame(&FrameDimensionTime, 0);
```

Перед тем как использовать `CachedBitmap`, подумайте о возможных неудобствах: вам придется отлавливать момент изменения видеорежима и соответственно перестраивать все оптимизированные растры. Кроме того, выгода от их применения невелика: в наших разных тестах она не превышала 9% (при избавлении от многих других тормозящих операций). Куда выгоднее оказалось вынести создание промежуточного контекста `Graphics` из функции `OnPaint` в код инициализации, хотя и это не панацея: такой буфер придется пересоздавать при изменении размеров окна.

И последнее. Ничего, напоминающего технику `DrawCachedBitmap`, в среде .NET мы не нашли. Но отчаиваться программистам .NET не нужно: замеры времени, затрачиваемого на вывод `CachedBitmap`, подозрительно совпадают с результатами вывода растров в формате `ImageFormat.Format32bppPArgb` (чемпиона по производительности среди растров GDI+). Скорее всего, львиной долей выигрыша в скорости класс `CachedBitmap` обязан именно предварительному умножению цветовых компонентов растра на величину альфа-канала, а этот формат доступен и в .NET.

13.5.5. Демонстрационные приложения

В завершение данного раздела представим два небольших демонстрационных приложения, находящихся на компакт-диске.

Первое написано с использованием C++-варианта этой библиотеки. Оно иллюстрирует применение многих описанных приемов работы: отложенную загрузку GdiPlus.DLL, загрузку растров формата GIF из ресурсов программы и использование двойной буферизации для устранения мерцания. Кроме того, в исходном коде есть (хотя и упрощенный) пример работы с анимированным изображением формата GIF.

Вы также можете использовать это приложение как полигон для собственных испытаний производительности GDI+. В обработчике `WM_PAINT` главного окна немедленно вызывается функция `InvalidateRect`, гарантируя приход следующего сообщения `WM_PAINT`, и вычисляется число кадров, выводимых в секунду.

Demo.zip — демонстрационный проект (VC++);

Demo_exe.zip — откомпилированное приложение.

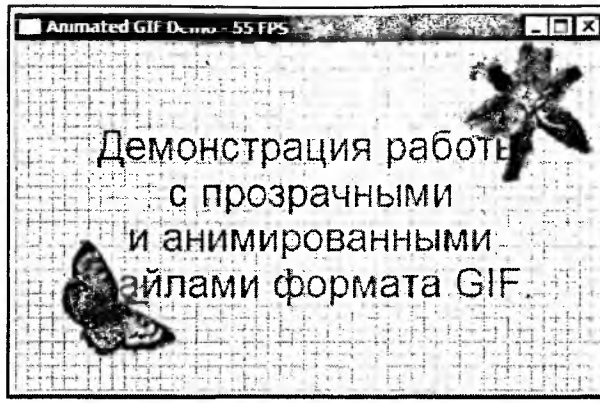


Рис. 13.4. Демонстрационная программа на C++

Второе написано на языке C# и представляет собой пример вывода анимированных файлов GIF в окне WinForms-программы. В нем также реализованы подсчет производительности (с использованием класса `System.Timers.Timer`) и двойная буферизация.

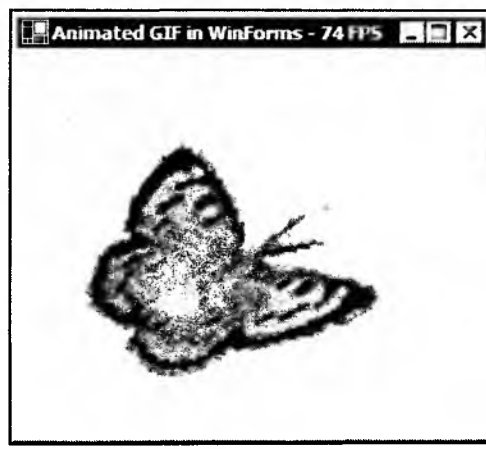


Рис. 13.5. Демонстрационная программа на C#

Animated.zip — исходный файл (C#).

Animated_exe.zip — откомпилированное приложение (требуется .NET Runtime).

Примечание

В обоих примерах вывод анимации намеренно происходит с максимально возможной скоростью для наглядного показа производительности. В реальном же приложении для вывода анимации с нужной скоростью вам потребуется получить параметры задержки кадров из графического файла (вызвав функцию `Image::GetPropertyItem` с параметром `PropertyTagFrameDelay`). Кроме того, в .NET есть специальный вспомогательный класс `System.Drawing.ImageAnimator`, который облегчает задачу анимации.

13.6. Прямая работа с растровыми данными

К сожалению, у библиотеки GDI+ существуют и ограничения. Некоторые из них связаны с дефектами текущей реализации и, возможно, вскоре исчезнут. Другие же напрямую следуют из новой архитектуры библиотеки, которая попыталась избавиться от "проклятого наследия" кое-каких архаизмов GDI. Так или иначе, нередко приходится обращаться напрямую к растровым данным, содержащимся внутри этих красивых оберток. В данном разделе мы узнаем, какие возможности для этого предоставляет GDI+.

13.6.1. Класс *Color*

В 32-битной цветовой модели наконец-то нашлось применение четвертому байту: он больше не является просто мусором для выравнивания (как в структуре `RGBQUAD`, например — *см. главу 10*), а законно хранит значение Alpha — технически говоря, величину непрозрачности пиксела. Это было учтено при проектировании класса `Color`: у него появился соответствующий конструктор:

```
Color(
  BYTE a, // alpha
  BYTE r, // red
  BYTE g, // green
  BYTE b  // blue
);
```

Если используется более традиционная форма конструктора `Color` с тремя цветовыми компонентами, то значение Alpha устанавливается равным 255 (полная непрозрачность). Кроме того, в классе `Color` описан большой набор именованных цветовых констант (например, `Color::AliceBlue`) — в них значение Alpha также равно 255.

В среде .NET принят немного другой подход: для инициализации структуры `Color` у нее существует множество статических свойств — значений цвета (например, `DeepSkyBlue`), а также методов (например, семейство методов

FromArgb). При этом величина каждого из четырех цветовых компонентов также не может превышать 8 бит:

```
public static Color FromArgb(int, int, int, int);
```

У структуры Color в .NET дополнительно имеются такие полезные качества, как возможность преобразования цвета в модель HSB (Hue-Saturation-Brightness, Оттенок-Насыщенность-Яркость):

```
public float GetBrightness();
public float GetHue();
public float GetSaturation();
```

а также реализация стандартного метода ToString, позволяющая получить строку с названием цвета (если это возможно) или с перечислением его ARGB-компонентов (в противном случае):

```
public override string ToString();
```

13.6.2. Прямой доступ к пикселам

Для получения и установки цвета определенной точки растра класс Bitmap предоставляет методы GetPixel/SetPixel:

```
Status GetPixel(INT x, INT y, Color* color);
Status SetPixel(INT x, INT y, const Color& color);
```

Что можно о них сказать? Используйте только при крайней необходимости — их производительность ужасна (как, впрочем, и производительность аналогичных функций GDI). Если есть желание нарисовать изображение этим способом, лучше один раз нарисовать его в объекте Bitmap, а потом выводить на экран кэшированное изображение из полученного растра.

Кстати, объект Graphics не предоставляет способа изменить цвет конкретной точки, как это было в GDI. С чем это может быть связано? Прежде всего, с попыткой достичь независимости графики от устройства вывода (подробнее см. статью в MSDN "INFO: Resolution-Independence in GDI+ (Q311460)".

Более быстрым будет получение доступа сразу к некоторой прямоугольной области растра. Для этого необходимо использовать метод Bitmap::LockBits:

```
Status LockBits(
  IN const Rect* rect,              // область растра для доступа
  IN UINT flags,                    // параметры доступа (чтение, запись)
  IN PixelFormat format,            // константа перечисления PixelFormat
  OUT BitmapData* lockedBitmapData // место для выходных данных
);
```

Параметр `flags` формируется из констант перечисления `ImageLockMode`. Помимо вида доступа к растру он может содержать флаг `ImageLockModeUserInputBuf`, указывающий на то, что поле `lockedBitmapData->Scan0` уже содержит указатель на выделенный пользователем буфер достаточного размера. Подробное описание этой функции и примеры ее использования можно найти в Platform SDK Documentation. В среде .NET эта функция также реализована, хотя для доступа к растровым данным вам придется применять unsafe-код.

Примечание

На самом деле вызов `LockBits` приводит к копированию указанной области во временный буфер. Изменение растровых данных в этом буфере отразится на содержимом `Bitmap` только после обратного вызова `UnlockBits` с тем же указателем `lockedBitmapData` в качестве параметра.

При этом, если указать формат временного буфера (`PixelFormat`), отличный от формата исходного растра, вызовы `LockBits/UnlockBits` потребуют дополнительных преобразований.

Для доступа к растру исходного изображения можно также воспользоваться следующим (недокументированным) обстоятельством. В GDI+ версии 1 метод `Bitmap::GetHBitmap` всегда возвращает DIB Section. Вам достаточно вызвать `GetObject()` для этого `HBITMAP`, чтобы получить растровые данные и необходимые структуры `BITMAPINFO` (пример работы с DIB Sections см. в статье Q186221). Однако на это поведение нельзя полагаться в будущих версиях библиотеки.

13.6.3. Поддержка прозрачности

Если возникла необходимость назначить прозрачным определенный цвет растра, сделать это можно несколькими способами. Во-первых, используя прямую замену цветов при помощи функций `GetPixel/SetPixel`. При этом придется пройтись по всем точкам картинки, заменяя точки выбранного цвета на прозрачные. Как уже говорилось, этот способ не является быстрым. Во-вторых, можно применить прямой доступ к памяти посредством вызова `LockBits`.

При работе в среде .NET необязательно возиться с заменой цветов и "сырыми" растрами, возвращаемыми вызовом `LockBits`. В классе `System.Drawing.Bitmap` реализован метод `MakeTransparent`, который делает прозрачным все точки растра, имеющие выбранный цвет:

```
Bitmap bm=new Bitmap("test.bmp");
// примем цвет первой точки растра за прозрачный
Color backColor = bm.GetPixel(0, 0);
bm.MakeTransparent(backColor);
```

А что, если исходную картинку модифицировать нельзя? Вы, конечно, можете воспользоваться клонированием растра или его фрагмента (используя метод `Bitmap::Clone`) и совершить замену пикселов дубликата. Однако в GDI+ есть более простой метод, поддерживающий прозрачность только при выводе картинок. Для этого необходимо создать экземпляр класса `ImageAttributes`, с которым мы уже столкнулись при обсуждении методов `Graphics::DrawImage`. Этот класс позволяет корректировать многие цветовые параметры выводимого изображения. В частности, метод `SetColorKey` позволяет указать, что определенный диапазон цветов (вернее, все пикселы, компоненты цвета которых лежат в этом диапазоне) будет заменяться при выводе прозрачным цветом. Посмотрите, например, что станет с бабочкой из демонстрационного проекта, если так модифицировать фрагмент метода `PaintBatterfly`:

```
ImageAttributes attr;
// все цвета в диапазоне (0,0,0,0) - (100,100,100,100) станут прозрачными
attr.SetColorKey(Color(0, 0, 0, 0), Color(100, 100, 100, 100));
Rect destRect(batterflyPos, Size(batterflyImage->GetWidth(),
  batterflyImage->GetHeight()));
g.DrawImage(batterflyImage, destRect, 0, 0,
  batterflyImage->GetWidth(),
  batterflyImage->GetHeight(),UnitPixel, &attr);
```

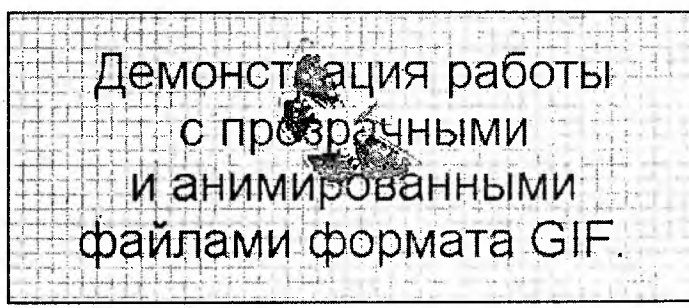


Рис. 13.6. Назначение цветового ключа целому диапазону цветов растра

Примечание

Если вы действительно вставите данный кусок в демонстрационную программу, то заметите досадную ошибку в текущей версии GDI+. При передаче в метод `Graphics::DrawImage` ненулевого указателя на класс `ImageAttributes` начинает выводиться только первый кадр анимации, несмотря на то что активным кадром может быть любой другой.

Кроме прямой замены цветов, класс `ImageAttributes` поддерживает так называемый recoloring — использование матричной алгебры для выполнения

вычислений над каждым цветовым компонентом пикселов. Так как альфа-компонент полноправно участвует в цветовых вычислениях, это позволяет использовать recoloring, например для увеличения степени прозрачности всего изображения в два раза. Вот соответствующая матрица:

```
| 1  0  0  0   0|
| 0  1  0  0   0|
| 0  0  1  0   0|
| 0  0  0 0,5  0|
| 0  0  0  0   1|
```

Каждый элемент этой матрицы является коэффициентом в диапазоне [0, 1]. Число `0,5` на главной диагонали в четвертой строке означает, что при умножении вектора из пяти элементов (четырех цветовых и одного фиктивного, необходимого для вычислений) на эту матрицу четвертый элемент вектора-результата будет равен исходному четвертому элементу, умноженному на `0,5`. А это как раз и есть компонент Alpha! Вот код, который подготавливает такое преобразование:

```
ColorMatrix colorMatrix =
{
  1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
  0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
  0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
  0.0f, 0.0f, 0.0f, 0.5f, 0.0f,
  0.0f, 0.0f, 0.0f, 0.0f, 1.0f
};
attr.SetColorMatrix(&colorMatrix,
  ColorMatrixFlagsDefault,
  ColorAdjustTypeBitmap);
```

А вот результат его применения к демонстрационному проекту:

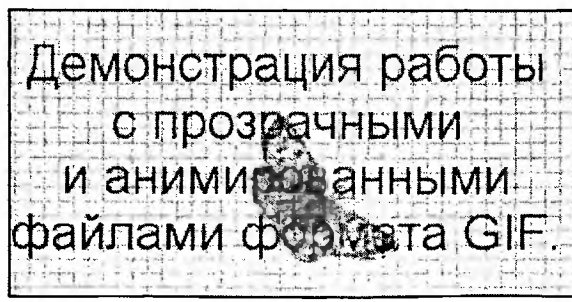


Рис. 13.7. Вывод растра с дополнительным альфа-наложением

13.6.4. Растровые операции

В форумах и группах новостей Usenet часто задается вопрос: а как можно заставить GDI+ использовать растровые операции (ROP), определенные в GDI? Например, для выделения набора объектов было бы неплохо их инвертировать (или закрасить инверсным прямоугольником). В САПР может понадобиться рисовать инверсную фигуру или контур (макет) будущего объекта. Кроме того, режим `R2_XOR` позволяет очень просто восстановить изображение под объектом, всего лишь повторно нарисовав объект на том же месте (подробнее *см. главу 6*).

Специалисты Microsoft обычно отвечают в духе дзен: "На самом деле вам не нужна такая возможность. XOR-графика попросту уродлива (согласен! — *В.Б.*). Современные 32-битные графические видеорежимы позволяют выделять и накладывать изображения с помощью альфа-канала. Применение различных кодов ROP для достижения прозрачности также устарело — прозрачность изначально реализована в GDI+". И действительно, в GDI+ вообще не поддерживаются ROP. Если попробовать "силой" выставить контексту устройства, например, режим `R2_XOR`, он будет проигнорирован при выводе.

Ну что ж, у программистов на C++ еще остается старушка GDI — только не забывайте про уже упомянутые проблемы взаимодействия, описанные в статье Q311221. А как быть работающим в среде .NET? Странно, оказывается, для .NET существует класс `System.Windows.Forms.ControlPaint`, не входящий в иерархию GDI+. Для рисования инверсных линий и прямоугольников он предоставляет методы `DrawReversibleLine`, `DrawReversibleFrame` и `FillReversibleRectangle`. Очевидно, программисты Microsoft владеют магией, позволяющей им использовать возможности, недоступные простым смертным?

Однако в Интернете уже давно существует популярная утилита Anakrino, позволяющая дизассемблировать .NET-сборки в исходный код на языках высокого уровня: C# и C++. "Натравив" ее на беззащитный метод `FillReversibleRectangle`, мы быстро разоблачим всю магию класса `ControlPaint`. (см. листинг 13.13). Оказывается, его методы обращаются к соответствующим низкоуровневым средствам GDI32.DLL (`SafeNativeMethods` — это просто внутреннее пространство имен, в котором описаны эти функции GDI):

Листинг 13.13. "Снимаем покрывало" с класса ControlPaint [C#, псевдокод]

```
public static void FillReversibleRectangle(
  Rectangle rectangle, Color backColor)
{
  int local0;
```

```
    int local1;
    IntPtr local2;
    IntPtr local3;
    int local4;
    IntPtr local5;

    local0 = ControlPaint.GetColorRop(backColor, 10813541, 5898313);
    local1 = ControlPaint.GetColorRop(backColor, 6, 6);
    local2 = UnsafeNativeMethods.GetDCEx(
      UnsafeNativeMethods.GetDesktopWindow(), IntPtr.Zero, 1027);
    local3 = SafeNativeMethods.CreateSolidBrush(
      ColorTranslator.ToWin32(backColor));
    local4 = SafeNativeMethods.SetROP2(local2, local1);
    local5 = SafeNativeMethods.SelectObject(local2, local3);
    SafeNativeMethods.PatBlt(local2, rectangle.X, rectangle.Y,
      rectangle.Width, rectangle.Height, local0);
    SafeNativeMethods.SetROP2(local2, local4);
    SafeNativeMethods.SelectObject(local2, local5);
    SafeNativeMethods.DeleteObject(local3);
    UnsafeNativeMethods.ReleaseDC(IntPtr.Zero, local2);
}
```

Как видим, для достижения необходимого эффекта используется "запрещенный" метод `SetROP2`.

Как использовать эти методы? Вот статьи MSDN Knowledge Base с соответствующими примерами:

HOW TO: Draw a Rubber Band Rectangle or Focus Rectangle in Visual C# (Q314945)

HOW TO: Draw a Rubber Band or Focus Rectangle in Visual Basic .NET (Q317479)

А вот наш собственный пример на C#, позволяющий инвертировать часть изображения. Попробуйте запустить пример, нажать где-нибудь на свободном участке формы левую кнопку мыши и потащить курсор. Выделенный участок инвертируется даже за пределами формы — взаимодействие высокоуровневой GDI+ и низкоуровневой GDI налицо!

RevFrame.zip — исходный файл (C#).

RevFrame_exe.zip — откомпилированная программа (требуется .NET Runtime).

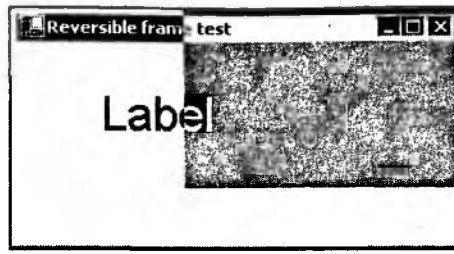
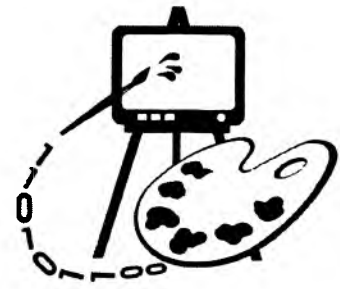


Рис. 13.8. Вывод инверсной графики средствами .NET

Вот, пожалуй, и все. Как видим, библиотека GDI+ предоставляет программистам мощнейшие средства для работы с растрами. В следующей же главе мы рассмотрим ее возможности в сфере векторной графики.

Глава 14



Построение векторных изображений средствами GDI+

Эта глава будет посвящена следующим вопросам:

- работа с графическими объектами;
- вывод векторных примитивов;
- настройка графического устройства;
- создание и изучение метафайлов.

14.1. Графические объекты

Для вывода *примитивов* (минимальных составляющих компьютерной графики, понятных устройству вывода) обычно требуются определенные настройки. Например, рисование кривых требует указания цвета и толщины линий ("пера"). При заливке сплошных фигур необходимо задать параметры "кисти" (заливка постоянным цветом, рисунком, градиентом цвета и т. д.). Для вывода текста может понадобиться создать шрифт определенных размеров и начертания.

Как правило, различные графические программные интерфейсы хранят такие настройки в виде собственных структур данных: *графических объектов*. Это справедливо как для GDI, так и для GDI+, однако использование графических объектов различается в двух этих средах. Рассмотрим эти отличия на примере маленькой задачи: рисование квадрата.

14.1.1. Stateful model в GDI

В GDI использована программная модель, называемая *моделью с сохранением состояния (stateful model)*. Это означает, что для устройства вывода запоминаются сделанные настройки и имеется понятие текущего выбранного

объекта. Перед выводом примитивов необходимый графический объект требуется выбрать в устройство вывода (с помощью функции `SelectObject`). Установив таким образом, например, толщину линии в три пиксела, затем можно вывести несколько отрезков выбранной толщины (листинг 14.1):

Листинг 14.1. Создание и использование графического объекта в GDI [C++]

```
void OnPaint(HDC hDc)
{
  HPEN hPen = CreatePen(PS_SOLID, 3, RGB(0, 0, 128));
  HGDIOBJ hOldPen = SelectObject(hDc, hPen);
  MoveToEx(hDc, 10, 10, 0);
  LineTo(hDc, 10, 100);
  LineTo(hDc, 100, 100);
  LineTo(hDc, 100, 10);
  LineTo(hDc, 10, 10);
  SelectObject(hDc, hOldPen);
  DeleteObject(hPen);
}
```

Такой подход имеет свои преимущества: установив единожды выбранное перо (`HPEN`), не нужно каждый раз передавать его в функцию вывода. Однако простота очень часто оборачивается серьезной проблемой: печально известной *утечкой ресурсов*.

Взгляните на выделенные строки. Зачем понадобилось запоминать и восстанавливать в контексте старое перо, если оно все равно не использовалось для рисования? Что произойдет, если закомментировать выделенный код?

Все дело в том, что выбранный в контексте графический объект *не может быть удален* — такая уж архитектура была выбрана при создании GDI. Следовательно, без этой строки вызов `DeleteObject` завершится неудачно, и созданное перо `hPen` останется "висеть" в *куче GDI* (GDI Heap), занимая нужную системе память.

Так как сообщение `WM_PAINT` приходит окну довольно часто, такой кусок кода будет выполняться снова и снова. В системах Windows 9*x* (где размер кучи GDI ограничен величиной 64 Кбайт) это быстро приведет к исчерпанию графических ресурсов системы. После этого все программы начнут вести себя очень странно: не перерисовывать некоторые элементы меню и иконки, рисовать текст подозрительными шрифтами и т. д. Системы на платформе NT будут "держаться на плаву" несколько дольше, так как их графическая куча может расти по мере необходимости, но и там существует

предел на число одновременно созданных графических объектов: 12 тысяч на процесс пользователя и 16 тысяч на всю систему.

Другая проблема заключается в том, что программисты на C++ привыкли к автоматической очистке используемых ресурсов благодаря использованию классов с деструкторами и "умных указателей". При этом легко вообще забыть о необходимости что-то явно удалять. Однако существующие библиотеки классов-оберток GDI (в частности, библиотека MFC) не могут за программиста решить проблему удаления выбранного в контексте объекта. Это понятно: тогда им придется постоянно проверять, является ли текущий объект выбранным в контексте, что отрицательно сказалось бы на производительности. Но такая ситуация также приводит к утечкам GDI-ресурсов.

14.1.2. Stateless model в GDI+

Отличительной для программиста особенностью GDI+ является изменение программной модели в работе с устройствами вывода. Концепция текущего выбранного в устройстве графического объекта выброшена как менее эффективная и устаревшая. Вместо последовательной установки параметров контекста (Graphics) используется перечисление атрибутов при каждом вызове графического метода. Устройство вывода как бы не обладает состоянием, а получает необходимую информацию через параметры. Такая модель носит название *модель без сохранения состояния (stateless model)*.

Примечание

Разумеется, такая классификация носит условный характер. Существуют и такие параметры (например, текущие настройки сглаживания, выбранная система координат и т. д.), которые хранятся в классе `Graphics` и связанных с ним структурах. Но *состояние примитивов* GDI+ не хранится в контексте отображения.

Теперь каждый метод, использующий для вывода графический объект, требует в качестве параметра явного указания этого объекта (листинг 14.2):

Листинг 14.2. Работа с графическими объектами в GDI+ [C++]

```
using namespace Gdiplus;
void OnPaint(Graphics &g)
{
  Pen pen(Color::Blue, 3);
  Point points[] = {
    Point(10, 10), Point(100, 10), Point(100, 100),
    Point(10, 100), Point(10, 10)
  };
```

```
  // Вывести указанным пером Pen
  // набор линий из массива точек points
  g.DrawLines(&pen, points, sizeof points / sizeof points[0]);
}
```

Это полностью исключает описанную проблему утечки ресурсов и позволяет освобождать все необходимые ресурсы в деструкторах классов-оберток GDI+ для языка C++.

Примечание

Если вы, уважаемый читатель, новичок в C++, то вас, возможно, смутит странная конструкция: `sizeof points / sizeof points[0]`. Это всего-навсего полезный прием для вычисления количества элементов произвольного массива на этапе компиляции. Размер (в байтах) массива `points` делится на размер его первого элемента.

Для C# такие ухищрения не требуются: массивы в этом языке являются объектами, и у них имеется стандартное свойство `Length`.

14.1.3. Кисти, краски, перья... и прочий "мусор"

Вы уже, наверное, догадались из названия, что сейчас речь пойдет не о самой GDI+, а о ее взаимодействии с .NET Framework. Как известно, в этой среде очистка объектов осуществляется не сразу при их выходе из области видимости, а позднее, при старте так называемого *сборщика мусора* (Garbage Collector, GC). При этом создание объекта в динамической памяти (куче) является гораздо более дешевой операцией, чем в традиционных средах. К тому же не требуется следить за удалением объекта — GC позаботится об этом сам, когда на объект не останется больше ссылок. Такие комфортные условия буквально подталкивают программистов к написанию кода в следующем стиле (листинг 14.3):

Листинг 14.3. Не рекомендуемое обращение с ресурсами в .NET [C#]

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  Graphics g = e.Graphics;
  Point[] points = {
    new Point(10, 10), new Point(10, 100),
    new Point(100, 100), new Point(100, 10), new Point(10, 10)
  };
  g.DrawLines(new Pen(Color.Blue, 3), points);
}
```

Из листинга видно динамическое создание двух объектов: массив структур `points` и экземпляр класса `Pen`. (оператор `new Point` всего-навсего вызывает конструктор структуры `Point`, а не создает ее динамически). Но нужно четко различать эти два случая. Коротко остановимся на этом, не углубляясь в детали .NET Framework.

Для создания обычного объекта (например, массива value-типов, как в приведенном примере) требуется только динамическая память. Но многие объекты управляют *ресурсами*: сущностями, количество которых по определению ограничено неким лимитом, не зависящим от нашей программы. К их числу относятся подключения к базам данных, файловые дескрипторы, графические объекты GDI+ и многое другое. При таком подходе ресурсы тоже будут освобождаться только в момент разрушения объекта. Это допустимо для демонстрационных примеров, но в серьезном приложении может привести к преждевременной исчерпанности ресурсов системы.

Как ни странно, такая опасность больше всего угрожает системам с большим количеством установленной памяти. Просто вызов GC (а следовательно, и очистка ресурсов) откладывается до момента нехватки динамической памяти, а это может произойти очень не скоро.

К счастью, разработчики GDI+ для .NET понимали эту проблему и реализовали для всех необходимых классов интерфейс `IDisposable`. Подробнее о нем можно прочитать в статье Игоря Ткачева "Автоматическое управление памятью в .NET" в первом номере журнала RSDN Magazine за 2002 год. Здесь же скажем только, что метод `IDisposable.Dispose` предписывает объекту освободить связанные с ним ресурсы. При использовании конструкции `using` в C# необходимую очистку можно выполнять неявно (листинг 14.4):

Листинг 14.4. Правильное использование графических ресурсов в .NET [C#]

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  Graphics g = e.Graphics;
  Point[] points = {
    new Point(10, 10), new Point(10, 100),
    new Point(100, 100), new Point(100, 10), new Point(10, 10)
  };
  using (Pen pen = new Pen(Color.Blue, 3))
    g.DrawLines(pen, points);
}
```

При выходе из области действия `using` компилятор сгенерирует вызов `pen.Dispose()`.

14.1.4. Разделение методов закраски и отрисовки

Переход к stateless-модели потребовал перепроектировать основные методы рисования примитивов. Раньше поведение, например функции `Rectangle`, определялось тем, какая кисть и какое перо выбраны в контексте отображения. Для рисования, скажем, незалитых прямоугольников требовалось выбрать в контексте кисть `HOLLOW_BRUSH`.

В GDI+, как уже было сказано, состояние примитивов не сохраняется в устройстве вывода (Graphics). Вместо этого практически все методы рисования замкнутых фигур имеют две версии: `DrawXXXX` для рисования контура фигур (этим методам в качестве параметра требуется экземпляр класса `Pen`) и `FillXXXX` для заливки (таким методам передается класс, порожденный от `Brush`).

14.1.5. Семейство *Brush*: набор кисточек на любой вкус

Для заливки сплошных областей соответствующие методы библиотеки требуют указывать объект класса `Brush`. Сам по себе этот класс практически бесполезен, так как его нельзя использовать непосредственно. Зато пять его потомков предоставляют богатую функциональность. Рассмотрим их чуть подробнее.

SolidBrush

Самый простой класс из семейства `Brush`. Предназначен для заливки областей однородным цветом. Этот цвет можно указать как в конструкторе класса, так и позднее, для уже сконструированного объекта:

```
// создаем кисть ярко-красного цвета
SolidBrush br(Color(255,0,0));
. . .
// устанавливаем для кисти черный цвет
br.SetColor(Color::Black);
```

Единственное (но довольно существенное) отличие класса `SolidBrush` от аналогичного графического объекта GDI — это поддержка полупрозрачности. Напомним, что класс `Color` позволяет указать помимо трех цветовых компонентов (RGB) еще и величину непрозрачности Alpha (по умолчанию, Alpha принимает значение 255 и цвет является полностью непрозрачным).

Разработчики WinForms приготовили пользователям .NET приятный сюрприз. Помимо класса `Color`, содержащего 141 константу с именем стандартного цвета, в этой среде существует и класс `System.Drawing.Brushes`, который также содержит 141 статическое свойство: кисть, уже инициализи-

рованную заданным цветом. Например, для рисования красного круга диаметром 100 единиц можно написать короткое выражение:

```
g.FillEllipse(Brushes.Red, 0, 0, 100, 100);
```

Как и одноименные константы класса `Color`, эти статические поля также содержат в поле Alpha значение 255.

HatchBrush

Класс `HatchBrush` предоставляет средства заливки двухцветным узором, основанным на *битовом шаблоне* (pattern). Конструктор этого класса на C++ выглядит так:

```
HatchBrush(
  HatchStyle hatchStyle,
  const Color& foreColor,
  const Color& backColor );
```

GDI+ содержит уже готовый набор из 53 шаблонов, описываемых перечислением `HatchStyle` (Сравните это количество с шестью предопределенными стилями в GDI). Параметры `foreColor` и `backColor` позволяют указать, соответственно, цвет линий шаблона и фоновый цвет кисти.

Вы можете контролировать расположение начальной точки шаблона относительно области рисования (метод `SetRenderingOrigin` и свойство `RenderingOrigin` в .NET).

TextureBrush

Этот класс позволяет заливать области растровым рисунком, или *текстурой*. Для инициализации объекта класса `TextureBrush` необходима картинка: экземпляр класса `Image`. Это может быть как растровое изображение (работа с которыми подробно описывалась в предыдущей главе), так и метафайл (с которыми мы познакомимся далее), что предоставляет нам гигантские возможности: нарисовав замысловатый узор, можно, в свою очередь, использовать его как "кисточку" для еще более сложного рисования. Вот короткий пример для WinForms (листинг 14.5):

Листинг 14.5. Использование класса TextureBrush [C#]

```
// читаем исходное изображение (метафайл)
private Image img = Image.FromFile("MyBrush.emf");

private void Form1_Paint(object sender, PaintEventArgs e)
{
  Graphics g = e.Graphics;
```

```
  // закрашиваем форму созданной кистью
  g.FillRectangle(new TextureBrush(img), ClientRectangle);
}
```

У класса существует несколько перегруженных конструкторов (мы не станем здесь их приводить). Помимо картинки (`Image`), они могут принимать в качестве параметра экземпляры классов `WrapMode` и `ImageAttributes`. Эти классы управляют различными аспектами вывода изображений (рассмотренными в предыдущей части при обсуждении растров).

Важно добавить, что, в отличие от класса `HatchBrush`, `TextureBrush` позволяет не только использовать для заливки рисунки с неограниченным набором цветов, но и подчиняется координатным трансформациям GDI+. Используемую текстуру можно масштабировать, перемещать и поворачивать любым желаемым образом.

LinearGradientBrush, PathGradientBrush

Мы рассмотрим эти два класса совместно, так как они очень похожи. Их назначение — заливка требуемой области цветовым переходом, или *градиентом цвета*. Мы уже использовали такой метод в примере из *главы 12* для рисования текста использовалась кисть с диагональным цветовым переходом. Вспомним этот фрагмент кода:

```
// Создаем кисть с градиентом на все окно и полупрозрачностью
LinearGradientBrush brush(
  bounds,
  Color(130, 255, 0, 0),
  Color(255, 0, 0, 255),
  LinearGradientModeBackwardDiagonal);
```

Первый параметр конструктора имеет тип `Rect` (`Rectangle` для .NET) и содержит в себе координаты прямоугольной области, в которой и происходит переход между стартовым цветом (второй параметр) и конечным цветом (третий параметр). Как видно из примера, ничто не мешает одному из цветов быть полупрозрачным (Alpha = 130). Если выводимая фигура превышает по габаритным размерам заданную прямоугольную область, то возникает вопрос: каким цветом закрашивать такие участки? Это поведение определяется методом `SetWrapMode` (так же, как и в случае с другими видами неоднородной заливки).

Последний параметр является элементом перечисления `LinearGradientMode` и определяет направление градиента, то есть расположение точек с полярными участками цвета.

Класс `PathGradientBrush` отличается от своего "близнеца" тем, что может использовать градиентный переход сразу между несколькими цветами. Положение требуемых полярных точек определяется *траекторией* (path), которая может принимать произвольную форму.

Оба эти класса также поддерживают координатные трансформации GDI+. Кроме того, закон распределения цветового перехода в них поддается регулировке (см. метод `SetSigmaBellShape` в .NET и `SetBlendBellShape` в версии для C++).

14.1.6. К штыку приравняли перо...

Класс `Pen` (для .NET — `System.Drawing.Pen`) предназначен для настройки параметров различных выводимых линий: отрезков прямых, дуг окружностей, эллипсов, сплайнов и кривых Безье. Помимо привычных для "пера" свойств — таких, как цвет линии и ее толщина, он содержит целую гамму различных дополнительных настроек. Во-первых, при создании экземпляра класса `Pen` в качестве источника можно указать класс типа `Brush` — а значит, для линий доступны все возможные варианты градиентных заливок, текстур и шаблонов (pattern fill). Соответствующие конструкторы приведены в листингах 14.6 и 14.7:

Листинг 14.6. Конструктор класса Pen из экземпляра Brush [C++]

```
Pen( const Brush* brush, REAL width=1.0 );
```

Листинг 14.7. Конструкторы класса Pen из экземпляра Brush [C#]

```
public Pen( Brush brush );
public Pen( Brush brush, float width );
```

Во-вторых, при указании цвета, как обычно в GDI+, допустимо задавать степень непрозрачности Alpha, создавая полупрозрачные линии (см. листинги 14.8 и 14.9).

Листинг 14.8. Конструктор класса Pen из сплошного цвета [C++]

```
Pen( const Color& color, REAL width=1.0);
```

Листинг 14.9. Конструкторы класса Pen из сплошного цвета [C#].

```
public Pen( Color color );
public Pen( Color color, float width );
```

Так же, как и для класса Brush, .NET Framework предоставляет более простой способ создания пера стандартного (именованного) цвета. В классе System.Drawing.Pens имеется 141 статическое свойство с именами стандартных цветов. Вот короткий пример:

```
g.DrawEllipse(Pens.Red, 0, 0, 100, 100);
```

И наконец, для задания геометрических характеристик линий в интерфейсе класса Pen существует множество настроек. Они перечислены в табл. 14.1 с кратким описанием:

Таблица 14.1. *Методы настройки стилей пера Pen*

Метод	Описание
SetAlignment	Определяет расположение точек пера относительно геометрической линии при рисовании многоугольников: внутри многоугольника или по центру его контура
SetCompoundArray	Позволяет составлять линию из набора параллельных линий различной толщины. Подробнее см. пример Clock
SetDashCap, SetDashOffset, SetDashPattern, SetDashStyle	Позволяют указать одно из 5 стандартных начертаний линий или задать свой собственный стиль прерывистого начертания
SetStartCap, SetEndCap, SetLineCap	Определяют геометрию и начертание (сплошное или прерывистое) концевых участков линии. Позволяют указать для каждого конца линии любую из восьми стандартных форм
SetCustomStartCap, SetCustomEndCap	Устанавливают произвольную форму концевых участков, основанную на траектории (Path). В качестве параметра требуют объект класса, производного от CustomLineCap. Для рисования более сложных стрелок существует готовый класс AdjustableArrowCap
SetLineJoin	Определяет внешний вид соединения отрезков многоугольника: скругленное, продолженное за пределы многоугольника или обрезанное
SetMiterLimit	Задает величину продолжения соединяемых под острым углом отрезков (MiterJoin)

Для всех перечисленных Set-методов, как обычно, существуют их Get-эквиваленты. В .NET доступ к настройкам пера осуществляется посредством свойств (без Get- и Set-префиксов).

14.2. Векторные примитивы

В этом разделе мы вкратце рассмотрим средства GDI+ для вывода традиционных векторных примитивов. Их использование не составит никакого труда для программистов, уже знакомых с GDI. А вот новинки (такие, как сплайны) заслуживают особого внимания.

14.2.1. Программа GDI+ Clock

В качестве учебного пособия здесь приводится программа **GDI+ Clock**, созданная для WinForms на языке C#.

В ней использован большой набор примитивов: прямоугольники, эллипсы и отрезки прямых. Для расчета их координат используются не геометрические преобразования GDI+ (которые будут рассматриваться в этой главе чуть позднее), а простые вычисления. В частности, созданный метод `RadialPoint` класса формы `frmClock` возвращает координаты конца секундной стрелки радиуса `radius` для значения секунд `seconds`.

Для реализации постоянной перерисовки часов воспользуемся компонентом `Timer`, позволяющим периодически генерировать событие `Tick`. Интервал генерации события определяется свойством с `Interval` (в миллисекундах). Инициализируем необходимые свойства таймера в конструкторе формы (листинг 14.10):

Листинг 14.10. Текст программы GDI+ Clock [C#]

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

namespace GDIPlusClock
{
  public class frmClock: Form
  {
    private Timer clkTimer;

    public frmClock()
    {
      clkTimer = new Timer();
      clkTimer.Enabled = true;
      clkTimer.Interval = 1000;
```

```
    clkTimer.Tick += new EventHandler(clkTimer_Tick);
    Load += new EventHandler(frmClock_Load);
    Paint += new PaintEventHandler(frmClock_Paint);
  }

  static void Main()
  {
    Application.Run(new frmClock());
  }

  private Point RadialPoint(int radius, int seconds)
  {
    Point ptCenter = new Point(this.ClientRectangle.Width/2,
      this.ClientRectangle.Height/2);
    double angle = -((seconds-15)%60)*Math.PI/30;
    Point ret = new Point(
      ptCenter.X+(int)(radius*Math.Cos(angle)),
      ptCenter.Y-(int)(radius*Math.Sin(angle)));
    return ret;
  }

  private void frmClock_Paint(object sender, PaintEventArgs e)
  {
    DateTime dt = DateTime.Now;
    Graphics g = e.Graphics;
    g.SmoothingMode = SmoothingMode.HighQuality;
    Point ptCenter = new Point(this.ClientRectangle.Width/2,
      this.ClientRectangle.Height/2);
    int radius = Math.Min(this.ClientRectangle.Width,
      this.ClientRectangle.Height)/2;
    using (LinearGradientBrush br = new LinearGradientBrush(
            this.ClientRectangle, Color.White, Color.DarkGray,
            LinearGradientMode.BackwardDiagonal))
    {
      g.FillEllipse(br, ptCenter.X-radius, ptCenter.Y-radius,
        radius*2, radius*2);
    }
    using (Pen pen = new Pen(Color.Black))
      g.DrawEllipse(pen, ptCenter.X-radius, ptCenter.Y-radius,
```

```
    radius*2, radius*2);

// рисуем отметки минут
for(int minute=0; minute<60; minute++)
{
  Point pt = RadialPoint(radius-10, minute);
  using (SolidBrush br = new SolidBrush(Color.Black))
  {
    if((minute%5)==0)
      g.FillRectangle(br, pt.X-3, pt.Y-3, 6, 6);
    else
      g.FillRectangle(br, pt.X-1, pt.Y-1, 2, 2);
  }
}

// рисуем часовые стрелки
using Pen pen = new Pen(Color.Black, 8))
{
  pen.StartCap = LineCap.Flat;
  pen.EndCap = LineCap.DiamondAnchor;
  float[] compVals = new float[]{0.0f, 0.2f, 0.5f,
    0.7f, 0.9f, 1.0f};
  pen.CompoundArray = compVals;
  g.DrawLine(pen, RadialPoint(15, 30+dt.Hour*5+dt.Minute/12),
    RadialPoint((int)(radius*0.75), dt.Hour*5+dt.Minute/12));
}

// рисуем минутные стрелки
using (Pen pen = new Pen(Color.FromArgb(100, 0, 0, 0), 6))
{
  pen.StartCap = LineCap.RoundAnchor;
  pen.EndCap = LineCap.Round;
  g.DrawLine(pen, RadialPoint(15, 30+dt.Minute),
    RadialPoint((int)(radius*0.8), dt.Minute));
}

// рисуем секундные стрелки
using (Pen pen = new Pen(Color.FromArgb(80, 20, 70, 30), 4))
{
```

```
        pen.CustomEndCap = new AdjustableArrowCap(4, 6, true);
        g.DrawLine(pen, RadialPoint(20, dt.Second+30),
          RadialPoint(radius-2, dt.Second));
      }
      using (SolidBrush br = new SolidBrush(
        Color.FromArgb(100, Color.Wheat)))
        g.FillEllipse(br, ptCenter.X-5, ptCenter.Y-5, 10, 10);
    }

    private void clkTimer_Tick(object sender, EventArgs e)
    {
      Text = DateTime.Now.ToLongTimeString()+" - GDI+ Clock";
      Invalidate();
    }

    private void frmClock_Load(object sender, EventArgs e)
    {
      // установить стили формы для двойной буферизации
      SetStyle(ControlStyles.AllPaintingInWmPaint, true);
      SetStyle(ControlStyles.DoubleBuffer, true);
      SetStyle(ControlStyles.UserPaint, true);
    }
  }
}
```

Программа отображает текущее время на стилизованном "аналоговом" циферблате:

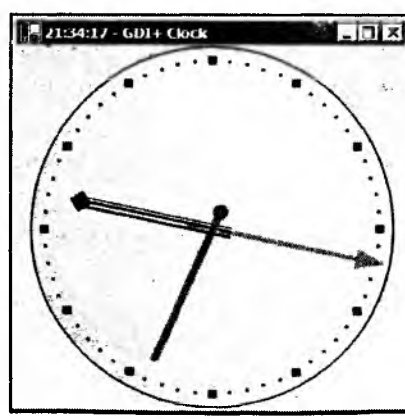


Рис. 14.1. Внешний вид программы GDI+ Clock

Заслуживает внимания отображение стрелок часов. Каждая стрелка (см. рис. 14.1) рисуется всего *одним вызовом* метода `DrawLine`! Как видите, GDI+ позволяет создавать довольно сложные изображения простыми средствами. Вы можете найти программу на компакт-диске и сами поэкспериментировать с установкой различных характеристик выводимых линий.

14.2.2. Сплайны

Одно из значений английского слова *spline* — *лекало*, то есть деревянное или металлическое приспособление для вычерчивания на бумаге кривых линий. Раньше для изготовления таких чертежных инструментов использовали металлическую пластину, которую фиксировали в заданных точках. Пластина огибала направляющие по гладкой кривой. Меняя силу натяжения и материал пластины, можно было получить целое семейство кривых для одного набора точек. Позднее эта зависимость превратилась в набор формул, описывающих *кубические сплайны*. Они широко используются в инженерных расчетах: например, для аппроксимации результатов экспериментов. При использовании GDI+ вам не понадобится знание этих формул. Для вычерчивания сплайнов достаточно вызвать функцию семейства `DrawCurve` (листинг 14.11):

Листинг 14.11. Перегруженные варианты метода DrawCurve [C#]

```
Status DrawCurve( const Pen* pen, const Point* points,
  INT count );
Status DrawCurve( const Pen* pen, const PointF* points,
  INT count );
Status DrawCurve( const Pen* pen, const Point* points,
  INT count, INT offset,
  INT numberOfSegments, REAL tension );
Status DrawCurve( const Pen* pen, const PointF* points,
  INT count, INT offset,
  INT numberOfSegments, REAL tension );
Status DrawCurve( const Pen* pen, const Point* points,
  INT count, REAL tension );
Status DrawCurve( const Pen* pen, const PointF* points,
  INT count, REAL tension );
```

Указатель `points` должен указывать на первый элемент массива структур `Point` (для целочисленных координат) или `PointF` (для координат с плавающей точкой).

Примечание

Реализация этих методов GDI+ для .NET отличается только тем, что не требует указания количества элементов массива `points`.

Как видим, имеется разновидность методов, принимающих параметр `tension` — число с плавающей точкой. Этот параметр как раз и характеризует "силу натяжения" нашей воображаемой пластины. В методах без этого параметра он принимается равным 0,5.

Характерной особенностью сплайнов является то, что построенная кривая проходит через каждую точку из заданного набора. Кроме того, для расчета координат сплайна в концевых точках требуются начальные условия. Они могут быть получены аппроксимацией, но их можно задать и явно. Для этого необходимо вызвать версию метода `DrawCurve` с параметрами `offset` и `numberOfSegments` и передать в этот метод массив с бóльшим количеством точек, чем требуется для рисования. Параметр `offset` определяет в массиве индекс точки (начиная с 0), с которой начнется рисование сегментов кривой, а параметр `numberOfSegments` указывает их количество. "Невидимые" точки будут использованы для расчетов.

Вы можете поэкспериментировать с этими дополнительными параметрами в демонстрационном приложении, модифицировав код нижеследующего метода (листинг 14.12):

Листинг 14.12. Метод `DrawSpline` демонстрационного приложения [C++]

```
void CCurveDlg::DrawSpline(Gdiplus::Graphics &g)
{
  using namespace Gdiplus;
  g.SetSmoothingMode(SmoothingModeHighQuality);
  g.DrawCurve(&Pen(Color::Blue, 3), points,
    sizeof points/sizeof points[0]);
}
```

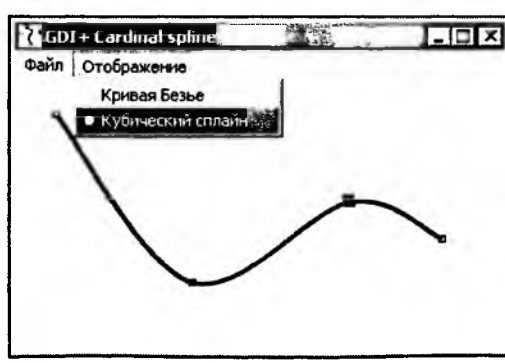


Рис. 14.2. Отображение сплайнов программой Curves

На рис. 14.2 приведен пример образуемой кривой (вы можете перемещать узловые точки).

И завершая краткое знакомство со сплайнами, заметим, что в классе `Graphics` также имеется семейство методов `DrawClosedCurve`, позволяющих выводить замкнутые кривые по заданному набору точек (листинг 14.13):

Листинг 14.13. Семейство методов `DrawClosedCurve` [C++]

```
Status DrawClosedCurve( const Pen* pen,
  const Point* points, INT count );
Status DrawClosedCurve( const Pen* pen,
  const PointF* points, INT count );
Status DrawClosedCurve( const Pen* pen,
  const Point* points, INT count, REAL tension );
Status DrawClosedCurve( const Pen* pen,
  const PointF* points, INT count, REAL tension );
```

В них для построения также используются сплайны, но с несколько иными соотношениями для конечных точек. Кроме того, как и у всех "порядочных" замкнутых фигур, поддерживаемых GDI+, у них существуют `Fill`-эквиваленты (листинг 14.14):

Листинг 14.14. Семейство методов `FillClosedCurve` [C++]

```
Status FillClosedCurve( const Brush* brush, const Point* points,
  INT count );
Status FillClosedCurve( const Brush* brush, const PointF* points,
  INT count );
Status FillClosedCurve( const Brush* brush, const Point* points,
  INT count, FillMode fillMode, REAL tension );
Status FillClosedCurve(const Brush* brush, const PointF* points,
  INT count, FillMode fillMode, REAL tension );
```

Эти методы отличаются от вышеназванных наличием параметра `Brush` (для задания способа заливки) и возможностью указания элемента перечисления `FillMode`. Он служит для определения того, будут ли заливаться внутренние области (образованные взаимным пересечением кривой). По умолчанию этот параметр принимает значение `FillModeAlternate`, что означает "не заливать внутренние области, образованные по правилу четности пересечений". При указании параметра `FillModeWinding` будет залита вся фигура, образованная внешним контуром кривой.

Примечание

В реализации GDI+ для .NET обнаружился забавный ляпсус. По неизвестной причине для метода `DrawClosedCurve` реализован также недокументированный вариант, принимающий параметр `FillMode`. Разумеется, его указание никак не влияет на поведение этой функции (которая не предназначена для заливки областей), да он и игнорируется в ее теле, как любезно сообщает утилита Anakrino.

14.2.3. Кривые Безье

С *кривыми Безье* программисты знакомы уже довольно давно. В PostScript они используются для описания шрифтов и рисования любых кривых, включая эллиптические. Windows GDI также поддерживает построение кривых Безье: например с помощью функций `PolyBezierTo` и `PolyDraw` (последняя недоступна в операционных системах Windows 9*x*).

Библиотека GDI+ никак не могла обойти вниманием столь серьезный инструмент. Кривые Безье не только поддерживаются при рисовании, но и активно используются самой библиотекой: в частности при сохранении траекторий в метафайлах.

Примечание

Своим названием кривые Безье обязаны их создателю, Пьеру Этьену Безье (1910—1999). Работая в компании Renault над CAD-системой UNISURF, он создал простое и понятное для рядового дизайнера средство описания сложных кривых. С тех пор они получили широкое распространение в конструкторских и дизайнерских системах.

Для построения кривой Безье необходимо задать четыре точки: две концевые, или опорные (end points), и две направляющие (control points). По принятому соглашению концевыми считаются первая и четвертая точки кривой, а вторая и третья точки рассматриваются как направляющие. Они в общем случае не лежат на кривой, но определяют ее форму (рис. 14.3).

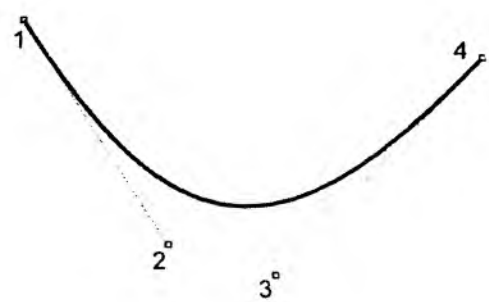


Рис. 14.3. Кривая Безье с опорными и концевыми точками

Следующие методы позволяют соединить две опорные точки кривой Безье с помощью двух направляющих точек (листинг 14.15):

Листинг 14.15. Семейство методов `DrawBezier` [C++]

```
Status DrawBezier( const Pen* pen, const Point& pt1,
  const Point& pt2, const Point& pt3, const Point& pt4 );
Status DrawBezier( const Pen* pen, const PointF& pt1,
  const PointF& pt2, const PointF& pt3, const PointF& pt4 );
Status DrawBezier( const Pen* pen, INT x1, INT y1,
  INT x2, INT y2, INT x3, INT y3, INT x4, INT y4 );
Status DrawBezier( const Pen* pen, REAL x1, REAL y1,
  REAL x2, REAL y2, REAL x3, REAL y3, REAL x4, REAL y4 );
```

В листинге 14.16 приведен фрагмент демонстрационного приложения Curves, выводящий таким образом сегмент кривой Безье:

Листинг 14.16. Метод `DrawBezier` демонстрационного приложения [C++]

```
void CCurveDlg::DrawBezier(Gdiplus::Graphics &g)
{
  using namespace Gdiplus;
  g.SetSmoothingMode(SmoothingModeHighQuality);

  // рисуем две линии к направляющим точкам
  g.DrawLine(&Pen(Color::Gray, 1), points[0], points[1]);
  g.DrawLine(&Pen(Color::Gray, 1), points[2], points[3]);

  // выводим собственно кривую
  g.DrawBezier(&Pen(Color::Blue, 3), points[0], points[1],
    points[2], points[3]);
}
```

Вы можете поэкспериментировать с программой, перемещая узловые точки и заставляя принимать кривую самые причудливые формы, — и все это всего за один вызов метода `DrawBezier`! Результат работы программы можно увидеть на рис. 14.4.

Существуют также методы `DrawBeziers`, позволяющие передать массив точек `points` для построения сразу нескольких сегментов кривой.

```
Status DrawBeziers(
```

```
  const Pen* pen, const Point* points, INT count
);

Status DrawBeziers(
  const Pen* pen, const PointF* points, INT count
);
```

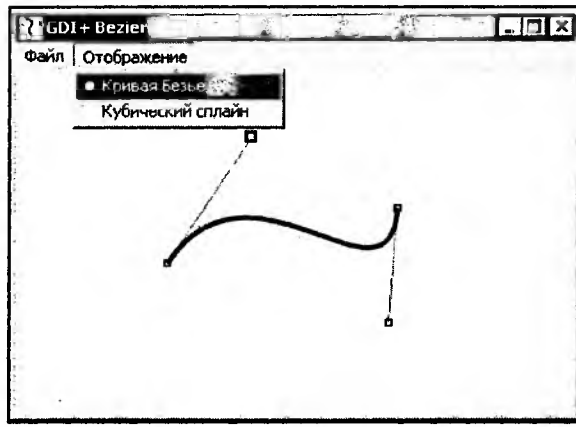


Рис. 14.4. Вывод кривых Безье программой Curves

Переменная `count` должна содержать число элементов массива `points`. Для построения N сегментов кривой необходимо передать массив, состоящий ровно из $3 \times N + 1$ точек, иначе вызов функции завершится неудачно.

14.3. Настройка устройства вывода

Не только для примитивов может понадобится настройка параметров вывода. Существует ряд настроек, которые изменяют характеристики самого графического устройства. Это, в частности:

- выбранное начало и масштаб системы координат;
- геометрические трансформации, применяемые к выводимым примитивам;
- область (регион) отсечения, применяемого к устройству;
- настройки качества выводимых примитивов.

Некоторые растровые характеристики устройства уже обсуждались в предыдущей главе. В этом разделе мы обсудим те параметры, которые влияют преимущественно на вывод векторных примитивов.

14.3.1. Устранение контурных неровностей

Как правило, человеческое зрение негативно воспринимает "зубчатые", пикселизованные изображения, особенно в дисплейной графике низкого разрешения. Например, человек способен читать текст с экрана в среднем на 30% медленнее, чем с бумаги. Для борьбы с этим явлением придумано множество технологий сглаживания — начиная от ClearType для LCD-мониторов и заканчивая Full Screen Anti-Aliasing (FSAA) для современных графических ускорителей.

Как уже упоминалось в прошлой главе, библиотека GDI+ имеет собственный набор средств для улучшения зрительного восприятия выводимой графики. При обработке растров используется *интерполяция* — вычисление промежуточных цветов выводимых пикселов, а для векторных изображений может применяться *антиалиасинг* — устранение контурных неровностей при помощи градаций цвета.

При антиалиасинге для построения, скажем, прямой линии, применяются не целочисленные алгоритмы, а их модификация с плавающей точкой. Для каждой точки выводимой линии вычисляется степень ее прозрачности (в зависимости от удаления данной точки от прямой).

Управление антиалиасингом осуществляется вызовом метода `Graphics::SetSmoothingMode` (для .NET — установкой свойства `SmoothingMode`). В качестве параметра используется элемент перечисления `SmoothingMode` (листинг 14.17):

Листинг 14.17. Перечисление SmoothingMode [C++]

```
enum SmoothingMode{
  SmoothingModeInvalid     = QualityModeInvalid,
  SmoothingModeDefault     = QualityModeDefault,
  SmoothingModeHighSpeed   = QualityModeLow,
  SmoothingModeHighQuality = QualityModeHigh,
  SmoothingModeNone,
  SmoothingModeAntiAlias
};
```

В данный момент фактически нет разницы между режимами `SmoothingModeDefault`, `SmoothingModeHighSpeed` и `SmoothingModeNone`: все они выключают антиалиасинг примитивов. Изображение при этом приобретает привычные ступенчатые края (рис. 14.5).

Для включения режима сглаживания используйте константы `SmoothingModeHighQuality` или `SmoothingModeAntiAlias` (результат приведен на рис. 14.6).

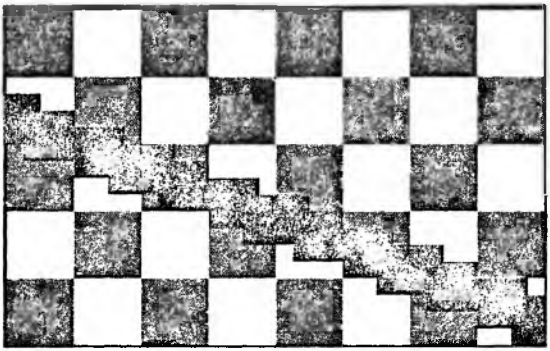


Рис. 14.5. Увеличенный фрагмент линии, выведенной без антиалиасинга

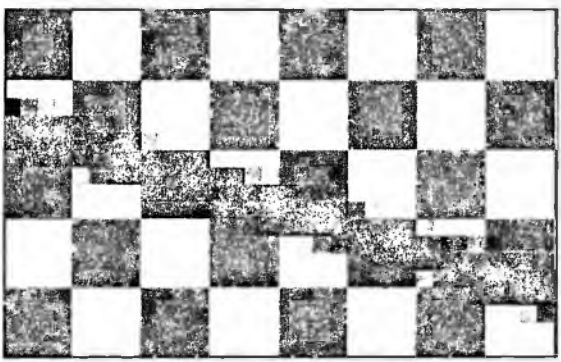


Рис. 14.6. Та же линия, выводимая при включенном антиалиасинге

Установка режима `SmoothingModeInvalid` не имеет смысла, так как вернет ошибку выполнения (а в среде .NET сгенерирует исключение).

14.3.2. Координатные преобразования GDI+

В *главе 5* мы рассмотрели математические основы координатных преобразований на плоскости. Теперь мы познакомимся с тем, какие средства GDI+ помогут нам реализовать эти преобразования.

При работе с системами координат GDI+ необходимо учитывать, что реализация библиотеки (по крайней мере, в версии 1.0) не основана на целочисленных механизмах. Внутри GDI+ координаты обрабатываются как `float`: числа с плавающей точкой обычной точности.

С одной стороны, это предоставляет бóльшую гибкость: например, программисты GDI+ уже не ограничены 16-битным лимитом координат при работе с Windows 9x/ME. С другой стороны, на центральный процессор ложится очень большая нагрузка по обсчету вещественных чисел. Даже при отключении "тяжелых" эффектов вроде антиалиасинга GDI+ заметно проиг-

рывает GDI при выводе простых векторных примитивов. Ситуация может измениться при выходе версии библиотеки, поддерживающей графические акселераторы.

Виды координатных систем

В GDI+ используется три различных системы координат: мировая (world), страничная (page) и система координат устройства (device). Вот типичный сценарий их взаимодействия:

1. Пользовательские команды вывода отдаются в мировой системе координат.
2. К координатам выводимого примитива применяются мировые преобразования, определенные в классе `Graphics`.
3. К полученным координатам применяется необходимое масштабирование, зависящее от выбранных характеристик страничной системы координат.
4. Выполняется вывод примитива в координатах устройства.

Как видите, во время исполнения простой команды (вроде рисования линии) графическая подсистема GDI+ может выполнять интенсивные вычисления. Наиболее быстро примитивы будут выводиться при отказе от мировых и страничных преобразований.

Примечание

В Windows GDI определено еще одно, четвертое координатное пространство: физическая система координат. Расчеты координат в физической системе выполняются, как правило, драйвером или контроллером устройства вывода. В GDI+ слой взаимодействия с аппаратными устройствами недокументирован: известно только, что используется программный интерфейс DCI (Device Control Interface), и система координат устройства принимается тождественно равной физической системе координат.

Класс *Matrix* и поддержка мировых преобразований

Класс `Graphics` предоставляет пользователям довольно удобные средства для выполнения наиболее распространенных преобразований в мировой системе координат: поворота, масштабирования и переноса. Для их выполнения служат, соответственно, методы `RotateTransform`, `ScaleTransform` и `TranslateTransform`:

```
Status RotateTransform( REAL angle, MatrixOrder order );
Status ScaleTransform( REAL sx, REAL sy, MatrixOrder order );
Status TranslateTransform( REAL dx, REAL dy, MatrixOrder order );
```

Обратите внимание на то, что угол поворота `angle` задается в градусах, а не в радианах, как параметры функций `sin`, `cos` и т. д.

Даже при использовании этих методов внутри класса `Graphics` все равно происходят преобразования с участием матричной алгебры. Перечисление `MatrixOrder` указывает, какой из операндов (существующая или новая матрица преобразования) будет находиться при умножении слева. По умолчанию параметр `order` принимает значение `MatrixOrderPrepend`.

Для явной подготовки преобразований библиотека предоставляет класс `Matrix`, содержащий матрицу коэффициентов 3 × 3. Его методы позволяют установить отдельные значения коэффициентов трансформации координат и задать готовый набор коэффициентов для наиболее часто применяемых преобразований.

Метод `Graphics::SetTransform` позволяет установить подготовленную матрицу преобразований в устройстве вывода. Следующие два фрагмента программы приводят к идентичному результату (листинги 14.18 и 14.19):

Листинг 14.18. Использование метода `Graphics::RotateTransform` [C++[

```
Graphics graphics(hdc);
graphics.RotateTransform(45.0f);
```

Листинг 14.19. Использование класса `Matrix` [C++]

```
Graphics graphics(hdc);
Matrix transformMatrix;
transformMatrix.Rotate(45.0f);
graphics.SetTransform(&transformMatrix);
```

Вы можете также использовать класс `Matrix` независимо от остальных классов GDI+: методы `TransformPoints` и `TransformVectors` позволяют выполнять преобразования для произвольных точек и векторов без их непосредственного рисования.

Примечание

Порядок применения различных преобразований влияет на результат. Если вы сначала примените к устройству преобразование поворота, а затем переноса, то результат будет отличаться от преобразования "перенос+поворот".

Страничные преобразования

Графические интерфейсы обычно позволяют абстрагироваться от физических характеристик устройства, позволяя указать, в какой логической системе координат будет происходить вывод.

Разумеется, для решения этих задач можно применять и мировые преобразования. В конечном итоге все сводится к привычным операциям масштабирования. Но лучше выделить независимый слой, отвечающий за логику такого рода. В GDI+ эту роль играет страничная система координат.

Для установки заранее определенной системы координат служит метод `SetPageUnit` (в .NET — свойство `PageUnit`), параметр которого определяет размер логических единиц устройства вывода (листинг 14.20):

Листинг 14.20. Перечисление `Unit` [C++, C#]

```
enum Unit
{
  UnitWorld = 0,       // мировые единицы (не документировано)
  UnitDisplay = 1,     // дисплейные единицы (для мониторов - пикселы)
  UnitPixel = 2,       // пикселы
  UnitPoint = 3,       // 1 единица равна 1/72  дюйма
  UnitInch = 4,        // 1 единица равна 1  дюйму
  UnitDocument = 5,    // 1 единица равна 1/300  дюйма
  UnitMillimeter = 6   // 1 единица равна 1  мм
};
```

Кроме этого, можно указать коэффициент масштабирования логических координат (одинаковый по обеим осям). Для этого в классе `Graphics` существует метод `SetPageScale` (в .NET — свойство `PageScale`). В приведенном ниже примере единица логической системы координат будет установлена равной 1 см независимо от того, на какое устройство будет происходить вывод:

```
Graphics graphics(hdc);
// Выбираем миллиметры в качестве логических единиц
graphics.SetPageUnit(UnitMillimeter);
// Устанавливаем масштабирование логических единиц
graphics.SetPageScale(10.0f);
```

14.3.3. Регионы и траектории

С понятием *регионов* программисты графики знакомы уже довольно давно. Говоря коротко, регион — это средство описания сложной области. В Windows API они используются для проверки вхождения точки в область, оптимизации перерисовки частично перекрытых окон, заливки сплошных фигур, а также создания окон непрямоугольной формы.

В этом разделе они упомянуты главным образом из-за их мощных функций отсечения: устройство вывода `Graphics` поддерживает установку сложных областей отсечения при помощи регионов (вызовом метода `SetClip`). Вы легко можете создать сложную фигуру, а затем рисовать "сквозь" нее, создавая интересные эффекты.

Регионы традиционно были привязаны к разрешению устройства вывода, для которого они создавались. С появлением GDI+ эта традиция нарушается: теперь регионы создаются в мировой системе координат и подчиняются всем описанным выше координатным преобразованиям. Кроме того, становится возможным применять один и тот же регион к нескольким разным устройствам вывода.

Следует помнить о том, что использование регионов для отсечения выводимых примитивов приводит к выделению большого количества памяти (для хранения скан-линий отсекаемой фигуры), и по возможности использовать прямоугольные области отсечения.

Существует набор перегруженных конструкторов, позволяющих создавать регионы из различных источников (листинг 14.21).

Листинг 14.21. Конструкторы класса `Region` [C++]

```
Region(Rect&);
Region();
Region(BYTE*,INT);
Region(HRGN);
Region(RectF&);
Region(GraphicsPath*);
```

В этом списке интереснее последний конструктор: он позволяет создавать регионы из областей, ограниченных *траекториями*.

Траектория *(path)* — это более новая концепция компьютерной графики, впервые реализованная в GDI только с выходом Windows 95. Траекторию можно определить как упорядоченную последовательность отрезков линий и кривых. Эти отрезки могут быть соединены общими точками, а могут быть и несвязанными — это зависит от графических команд, включенных в траекторию.

Объект Graphics поддерживает установку области отсечения, ограниченной траекторией так же, как и регионом — используйте для отсечения тот объект, который наиболее удобен в данный момент.

Если вы пользовались траекториями в Windows GDI, то знаете, что в этом программном интерфейсе объект траектории недоступен пользовательскому приложению. Построение траекторий при этом начинается с перевода кон-

текста устройства (HDC) в специальный режим вызовом функции BeginPath. Далее графические команды вместо непосредственного выполнения начинают запоминаться в контексте, образуя контур траектории. Это продолжается до вызова команды EndPath, которая восстанавливает исходный режим устройства и фиксирует текущую траекторию.

Объект GraphicsPath, представляющий в GDI+ средства работы с траекториями, гораздо более гибок. Он не привязан к устройству вывода и имеет набор собственных методов для построения траектории. Они сходны по названию с методами класса Graphics, предназначенными для построения таких же примитивов. В табл. 14.2 перечислены схожие методы этих двух классов. На самом деле у каждого метода существует большое количество перегруженных вариантов, различающихся типами параметров.

***Таблица 14.2.** Сопоставление методов классов GraphicsPath и Graphics*

Метод GraphicsPath	Аналог класса Graphics	Выводимый примитив или действие
AddArc	DrawArc	Эллиптическая дуга
AddBezier	DrawBezier	Кривая Безье
AddBeziers	DrawBeziers	Цепочка связанных кривых Безье
AddClosedCurve	DrawClosedCurve	Замкнутая кривая (кубический сплайн)
AddCurve	DrawCurve	Незамкнутая кривая (кубический сплайн)
AddEllipse	DrawEllipse	Эллипс
AddLine	DrawLine	Отрезок прямой
AddLines	DrawLines	Цепочка несвязанных отрезков прямых
AddPath	DrawPath	Существующая траектория
AddPie	DrawPie	Вывод сегмента дуги
AddPolygon	DrawPolygon	Цепочка связанных отрезков прямых
AddRectangle	DrawRectangle	Прямоугольник
AddRectangles	DrawRectangles	Набор несвязанных прямоугольников
AddString	DrawString	Строка текста
StartFigure	Нет	Начало новой геометрической фигуры (без замыкания предыдущей)
CloseFigure	Нет	Замыкание текущей фигуры
CloseAllFigures	Нет	Замыкание всех незамкнутых фигур

Примечание

Как и в Windows GDI, для хранения сегментов траектории применяются только прямые линии и кривые Безье. Эллиптические кривые (дуги окружностей и эллипсов) также преобразуются в кривые Безье при сохранении в объекте `GraphicsPath`. При этом возникает небольшая погрешность, заметная, если многократно увеличить координаты точек, образующих траекторию.

Построив один раз объект траектории, мы можем неоднократно его использовать. Для вывода траекторий класс `Graphics` предоставляет методы `DrawPath` (для обводки контура траектории) и `FillPath` (для заливки ее замкнутых областей), см. листинг 14.22:

Листинг 14.22. Методы класса `Graphics` для рисования траекторий

```
Status DrawPath( const Pen *pen, const GraphicsPath *path );
Status FillPath( const Brush *brush, const GraphicsPath *path );
```

Мы научимся использовать эти методы в следующем разделе.

Применение траекторий: класс *RoundRect*

Один из вопросов, задаваемых новичками в процессе изучения GDI+: "Почему не реализована функция `RoundRect`?!" Что ж, в классе `Graphics` действительно нет метода, позволяющего рисовать скругленные прямоугольники. Но ничто не помешает нам получить даже более удобную функциональность. Создадим класс, который будет возвращать траекторию (`GraphicsPath`) скругленного прямоугольника в одном из методов. Оставшиеся методы предназначены для закраски и отрисовки полученной траектории. Листинг такого класса приведен ниже (листинг 14.23).

Листинг 14.23. Реализация класса `RoundRect` [C#]

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;

namespace RSDN
{
  class RoundRect
  {
    public static GraphicsPath Create(float X, float Y,
                                      float Width, float Height,
```

```
                                    float Radius)
{
  GraphicsPath gp = new GraphicsPath();
  gp.AddLine(X + Radius, Y, X + Width - Radius, Y);
  gp.AddArc(X + Width - Radius, Y, Radius, Radius, 270, 90);
  gp.AddLine(X + Width, Y + Radius, X + Width, Y + Height - Radius);
  gp.AddArc(X + Width - Radius, Y + Height - Radius, Radius, Radius,
    0 , 90);
  gp.AddLine(X + Width - Radius, Y + Height, X + Radius, Y + Height);
  gp.AddArc(X, Y + Height - Radius, Radius, Radius, 90, 90);
  gp.AddLine(X, Y + Height - Radius, X, Y + Radius);
  gp.AddArc(X, Y, Radius, Radius, 180, 90);
  gp.CloseFigure();
  return gp;
}

public static void Draw(Graphics g, Pen p,
                        float X, float Y,
                        float Width, float Height, float Radius)
{
  using (GraphicsPath gp = Create(X, Y, Width, Height, Radius))
  {
    g.DrawPath(p, gp);
  }
}

public static void Fill(Graphics g, Brush br,
                    float X, float Y,
                    float Width, float Height, float Radius)
{
  using (GraphicsPath gp = Create(X, Y, Width, Height, Radius))
  {
    g.FillPath(br, gp);
  }
}

public static void DrawFilled(Graphics g, Pen p, Brush br,
                              float X, float Y,
```

```
                                    float Width, float Height,
                                    float Radius)
    {
      using (GraphicsPath gp = Create(X, Y, Width, Height, Radius))
      {
        g.FillPath(br, gp);
        g.DrawPath(p, gp);
      }
    }
  }
}
```

Используя класс `RSDN::RoundRect`, становится довольно просто нарисовать, например, выпуклую кнопку (листинг 14.24):

Листинг 14.24. Использование класса RoundRect [C#]

```
protected override void OnPaint(PaintEventArgs e)
{
  base.OnPaint(e);

// Создаем траекторию, содержащую скругленный прямоугольник
  GraphicsPath path = RSDN.RoundRect.Create(10, 10,
    ClientRectangle.Width-20, ClientRectangle.Height-20, 6);
  e.Graphics.SmoothingMode=SmoothingMode.HighQuality;

  // Готовим градиентную кисть для закраски области
  LinearGradientBrush br = new LinearGradientBrush(
    new Rectangle(0, 0, ClientRectangle.Width-10,
      ClientRectangle.Height-10),
    Color.DarkGray, Color.White, 90f);

  // Закрашиваем область
  e.Graphics.FillPath(br, path);

  // Обводим ее черной линией толщиной в 1 логическую единицу
  e.Graphics.DrawPath(Pens.Black, path);
}
```

14.4. Метафайлы

Начиная с самой первой версии, в Windows поддерживается концепция *метафайлов:* записываемой последовательности графических команд. Такую последовательность можно сохранить на диск в виде файла, а затем в любой момент воспроизвести как магнитофонную запись. Такой файл будет состоять из набора записей, соответствующих выполненным командам GDI.

В отличие от растровых рисунков, метафайлы не привязаны к разрешению устройства вывода, для которого они создавались. Так как координаты в графических командах GDI легко масштабируются, при увеличении изображения качество рисунка остается высоким. Разумеется, это не касается растровых команд, которые также могут присутствовать в метафайлах.

Первые версии Windows поддерживали довольно ограниченный формат метафайлов: так называемые Windows Metafiles. Они были предназначены только для 16-битной координатной системы, не содержали информации о разрешении устройства вывода, для которого были созданы, и позволяли записывать не все существующие команды GDI. Такие метафайлы сохраняются в файлах с расширением wmf.

Для Windows 95 был создан новый формат Enhanced Metafiles, свободный от недостатков своего предшественника. Он обладает настоящей независимостью от устройства вывода и поддерживает все команды GDI. Этот формат оказался настолько удачным, что стал активно использоваться в самой системе Windows: в частности, при подготовке файлов менеджера печати. Такие "улучшенные" метафайлы хранятся в файлах с расширением emf (Enhanced Metafile Format).

Библиотека GDI+ привнесла новый набор графических команд, и для их сохранения в метафайлах пришлось расширить оригинальный набор записей EMF. Такой расширенный формат получил название — кто бы мог подумать — правильно, EMF+. Но, при желании, в GDI+ можно создавать и традиционные EMF. При этом все вызовы GDI+ будут транслироваться в набор команд GDI. Разумеется, специфические для GDI+ команды (такие, как установка режима антиалиасинга) не могут быть сохранены в этом формате.

Кроме того, поддерживается и так называемый дуальный формат (Dual EMF+), содержащий двойной набор записей EMF и EMF+. Такие файлы будут открываться в обеих средах.

Для работы с метафайлами в иерархии GDI+ существует класс `Metafile`. Далее мы рассмотрим некоторые вопросы, которые могут возникнуть при его использовании.

14.4.1 Загрузка метафайлов

После прочтения предыдущей главы загрузка метафайла не должна составить для вас никакого труда. Дело в том, что класс `Metafile` является таким же полноправным потомком класса `Image`, как и `Bitmap`. Следовательно, для него работают те же самые методы загрузки — из файлов (с указанием имени) и потоков `IStream`. Вот небольшой пример на C++ (листинг 14.25):

Листинг 14.25. Загрузка метафайлов [C++]

```
// Загрузка из файла
Metafile mf1(L"Sample1.emf");
Metafile *mf2 = (Metafile*) Image::FromFile(L"Sample2.emf");

// Загрузка из потока
LPSTREAM pIS;
. . .
Metafile * mf3 = (Metafile*) Image::FromStream(pIS);
. . .
delete mf2;
delete mf3;
```

Отметим, что конструктор и метод `FromFile` требуют в качестве имени файла Unicode-строку. При указании путей с подкаталогами в .NET очень удобно воспользоваться новшеством C#, так называемыми *verbatim strings* (строковыми константами без разбора escape-последовательностей). Такие строковые константы предваряются символом "@":

```
Image img = Image.FromFile(@"C:\RSDN\GDIPlus\3\Sample.emf");
```

При этом не придется дублировать символ "\" — разделитель каталогов.

Методы `Image::FromFile` и `Image::FromStream` возвращают указатель на класс `Image`. Если вы уверены в том, что загружен будет именно метафайл (а не растр), то указатель можно "силой" привести к необходимому типу `Metafile*`. В противном случае, для выяснения типа загруженного изображения используется метод `GetRawFormat` (в .NET — свойство `RawFormat`), возвращающий GUID кодека, использованного для загрузки. Для загруженных EMF и EMF+ этот метод вернет константу формата `ImageFormatEMF` (для .NET свойство примет значение `ImageFormat.EMF`).

Кроме того, в среде .NET на нас работает вся мощь reflection. У созданного таким образом объекта класса `Image` можно вызвать стандартный метод `GetType`, который вернет информацию о типе объекта. В частности, вызов

```
Text = img.GetType().FullName;
```

присвоит заголовку формы строку "System.Drawing.Imaging.Metafile".

14.4.2. Воспроизведение

Для "проигрывания" содержимого загруженного метафайла в контекст отображения используется уже знакомый нам метод `Graphics::DrawImage`, содержащий огромное количество перегруженных вариантов (16 для версии C++ и 30 — для .NET). Вот короткая иллюстрация применения этого метода в WinForms-приложении:

```
Image img;

private void Form1_Paint(object sender, PaintEventArgs e)
{
  if(img!=null)
  {
    e.Graphics.DrawImage(img, 0, 0);
  }
}
```

Также, как и при выводе растров, поддерживаются преобразования координат и геометрические искажения исходного рисунка. Кроме того, исправно работают уже описанные в прошлой части методы исправления цветов и изменения атрибутов вывода (с использованием класса `ImageAttributes`).

Использование метафайлов привносит только одну тонкость: при использовании вариантов метода `DrawImage` с указанием ограничивающего прямоугольника будут воспроизводиться только те записи, которые соответствуют командам, попадающим в область вывода. Это определяется как по координатам команд, так и по их специфике: в частности, не будет пропущен вызов метода `DrawLine`, если концевые точки отрезка лежат вне указанной области, но сам отрезок ее пересекает.

Кроме того, в реализации класса `Graphics` для .NET Framework существует также совершенно бесполезный метод `DrawImageUnscaled`, который просто вызывает `DrawImage` с передаваемыми параметрами (иногда часть их них игнорируя). Это нетрудно выяснить, заглянув в дизассемблированный листинг каждого перегруженного варианта метода `DrawImageUnscaled`. Можно только порадоваться трудолюбию сотрудников Microsoft, не устающих плодить сущности без необходимости.

14.4.3. Создание и сохранение нового метафайла

Хотелось бы предостеречь читателей от ошибки, которую легко совершить в процессе изучения GDI+. Из предыдущих разделов видно, что загрузка и отображение метафайлов ничем не отличаются от аналогичных операций с классом Bitmap. Из этого можно сделать вывод, что и сохранение дискового метафайла будет симметричной операцией. Это не так.

Причина проста: если для преобразования метафайла в растровое изображение (например, для вывода на экран или растровый принтер) достаточно выполнить содержащиеся в нем команды, то обратное неверно. Для преобразования произвольного изображения в векторный формат требуется очень большая вычислительная работа (нетривиальный анализ), и в общем случае эта задача без участия человека неразрешима. Поэтому "векторного кодека сохранения" библиотека GDI+ не предоставляет (как можно убедиться, прочитав предыдущую главу про растры).

Вместо этого предполагается воспользоваться специальными "конструкторами сохранения" метафайлов. Их легко опознать по обязательному наличию двух параметров: строки с именем файла и параметра `referenceHdc`, который должен содержать контекст устройства отображения.

```
Metafile( const WCHAR* fileName, HDC referenceHdc,
  EmfType type, const WCHAR* description );
```

Созданный метафайл будет содержать информацию о разрешении этого контекста. Параметр `description` позволяет указать строковое описание, которое будет сохранено в метафайле. Параметр `type` определяет формат создаваемого метафайла (в частности, позволяет сохранять в уже упомянутом формате Dual EMF+):

```
enum EmfType{
  EmfTypeEmfOnly     = MetafileTypeEmf,
  EmfTypeEmfPlusOnly = MetafileTypeEmfPlusOnly,
  EmfTypeEmfPlusDual = MetafileTypeEmfPlusDual
};
```

Запись в метафайл осуществляется простым конструированием на нем объекта `Graphics` (так же, как и рисование в растровый буфер). Запись считается законченной при удалении контекста отображения.

Для иллюстрации процесса сохранения нам необходим работающий пример. Немного подшутив над своим же кодом, мы чуть модифицировали программу на C++ из *главы 12*. Теперь, вместо того, чтобы выводить текст приветствия в окно, программа сохраняет его в метафайл (листинг 14.26):

Листинг 14.26. Создание метафайла EMF+ и заполнение его записями [C++]

```
using namespace Gdiplus;
// Все строки - в кодировке Unicode
WCHAR welcome[]=L"Welcome, EMF+ !";
RectF bounds(0, 0, 400, 300);

Metafile metafile(L"Sample.emf", GetDC(0));
```

```
Graphics g(&metafile);

// Создаем кисть с градиентом на все окно
// и полупрозрачностью
LinearGradientBrush brush(bounds, Color(130, 255, 0, 0),
  Color(255, 0, 0, 255),
  LinearGradientModeBackwardDiagonal);

// Готовим формат и параметры шрифта
StringFormat format;
format.SetAlignment(StringAlignmentCenter);
format.SetLineAlignment(StringAlignmentCenter);
Font font(L"Arial", 48, FontStyleBold);

// Выводим текст приветствия, длина -1 означает,
// что строка заканчивается нулем
g.DrawString(welcome, -1, &font, bounds, &format, &brush);
```

14.4.4. Преобразование в растровое изображение

Что, если созданный или загруженный с диска метафайл необходимо преобразовать в растр и сохранить, например, в формате PNG? Первое решение напрашивается само собой: создать растровую картинку (`Bitmap`) в памяти, подготовить объект `Graphics` для вывода в этот растр и нарисовать в полученный контекст исходный метафайл. Такая техника ("теневой буфер") была подробно рассмотрена в предыдущей главе при обсуждении мерцания.

Тем не менее библиотека GDI+, как обычно, предоставляет и более простой способ "ободрать кошку"[1]. Для создания растра необходимо просто использовать конструктор `Bitmap`, принимающий в качестве параметра указатель на `Image` (в .NET — экземпляр класса `Image`). Все! В результате получается полноценный растр, который можно сохранить на диск:

```
Bitmap bm = new Bitmap(img);
bm.Save(@"C:\RSDN\GDIPlus\3\Sample.png", ImageFormat.Png);
```

Использование этого метода ограничено только тем, что полученный растр будет создан с разрешением дисплея. Заметьте, что в сохраненной картинке будет содержаться корректная информация о прозрачности (в тех областях, которые не были затронуты командами вывода).

---

[1] "Существует много способов ободрать кошку" — американская пословица. Лично мы ничего не имеем против этих милых животных.

14.4.5. Изучение команд метафайла

Формат записей метафайла документирован, и для разбора его содержимого вполне можно написать программу, которая будет читать структуры данных метафайла и интерпретировать их содержимое в зависимости от типа записи. Но зачем? Такая работа уже проделывается библиотекой GDI+ при отображении метафайлов, и разработчики могут использовать ее в своих приложениях. Для этого предназначен метод `EnumerateMetafile` класса `Image`. Так как он имеет большое количество перегруженных вариантов, мы рассмотрим в качестве иллюстрации только два объявления: одно для C++ (библиотека GDI+ для Windows) и второе для C# (WinForms).

1. [C++]

```
Status EnumerateMetafile(
  const Metafile* metafile,
  const Point& destPoint,
  EnumerateMetafileProc callback,
  VOID* callbackData,
  ImageAttributes* imageAttributes
);
```

2. [C#]

```
[ComVisible(false)]
public void EnumerateMetafile(
   Metafile metafile,
   Point destPoint,
   Graphics.EnumerateMetafileProc callback,
   IntPtr callbackData,
   ImageAttributes imageAttr
);
```

Как видим, этот метод очень похож на `DrawImage`. Точно также необходимо передать отображаемый метафайл и начальные координаты для "рисования" (`destPoint`). Подобно `DrawImage`, существует возможность изменить геометрию и атрибуты выводимого изображения. Единственные уникальные параметры — это `callback` и `callbackData`. Первый является указателем (в .NET — делегатом) на процедуру, которая будет последовательно вызываться библиотекой для каждой обнаруженной записи в метафайле. Второй параметр полезен, если для работы этой процедуры требуются какие-то

дополнительные параметры. Вот примерное определение такой процедуры (листинги 14.27 и 14.28):

Листинг 14.27. Синтаксис процедуры — обратного вызова [C++]

```
BOOL CALLBACK metaCallback( EmfPlusRecordType recordType,
  unsigned int flags,  unsigned int dataSize,
  const unsigned char* recordData, void* callbackData );
```

Листинг 14.28. Синтаксис процедуры — обратного вызова [C#]

```
public bool metaCallback( EmfPlusRecordType recordType, int flags,
  int dataSize, IntPtr recordData, PlayRecordCallback callbackData );
```

Воспользовавшись параметром `callbackData`, можно, например, передать в процедуру `metaCallback` (версии для C++) указатель на метафайл, записи которого перечисляются в данный момент. Тогда для выполнения графических команд можно будет воспользоваться методом `Metafile::PlayRecord` (листинг 14.29):

Листинг 14.29. Самостоятельное рисование записей метафайла [C++]

```
BOOL CALLBACK metaCallback( EmfPlusRecordType recordType,
  unsigned int flags, unsigned int dataSize,
  const unsigned char* recordData, void* callbackData)
{
  // Выполним явное приведение типа указателя к Metafile*
  ((Metafile*)callbackData)->PlayRecord(
  recordType, flags, dataSize, recordData);
  return TRUE;
}
```

Параметр `recordType` при каждом вызове callback-процедуры принимает значение из перечисления `EmfPlusRecordType`, содержащего аж 253 элемента (самого большого перечисления GDI+). Все эти элементы либо прямо соответствуют командам GDI/GDI+, либо обозначают служебные записи метафайла. При вызове `PlayRecord` вы можете подменять этот параметр своими значениями для выполнения совершенно других команд GDI+ в методе `PlayRecord`. Но для этого необходимо твердо знать значение соответствующих недокументированных параметров `recordData` и `dataSize` (и передавать, соответственно, измененные значения и в них).

14.4.6. Перечисление записей: специфика .NET

В документации по WinForms указывается, что среда .NET предоставляет собственную версию "делегата" для выполнения соответствующих команд метафайла в теле нашего callback-обработчика. Эта версия передается в обработчик в параметре `callbackData`. Для выполнения команды достаточно вызвать полученный "делегат" (пример скопирован из документации) (листинг 14.30):

Листинг 14.30. Пример из документации по .NET Framework [C#]

```
// Define callback method.
private bool MetafileCallback(
                EmfPlusRecordType recordType,
                int flags,
                int dataSize,
                IntPtr data,
                PlayRecordCallback callbackData)
{
  // Play only EmfPlusRecordType.FillEllipse records.
  if (recordType == EmfPlusRecordType.FillEllipse)
  {
    // Play metafile.
    callbackData(recordType, flags, dataSize, data);
  }
  return true;
}
```

Предупреждение

Внимание! Это явная ошибка .NET Framework Documentation (ох, уже в который раз!). Приведенный пример откомпилируется, но его выполнение ни к чему хорошему не приведет. О том же сказано и в [22].

Эксперименты показали, что в параметре `callbackData` всегда передается нулевое значение, независимо от фазы луны. Вызов такого "делегата" завершится исключением `NullReferenceException`.

Все-таки создать рабочую версию применения метода `EnumerateMetafile` для .NET вполне возможно. Для этого нам придется немного повозиться с методом `Metafile.PlayRecord`. Дело в том, что он в качестве параметра `data` принимает массив байтов:

```
public void PlayRecord( EmfPlusRecordType recordType,
  int flags, int dataSize, byte[] data );
```

Но в теле callback-метода нам доступен только unmanaged-указатель `IntPtr recordData` (см. объявление "делегата"). Для копирования необходимых данных в managed-массив .NET можно воспользоваться классом `System.Runtime.InteropServices.Marshal` (листинг 14.31):

Листинг 14.31. Работающий пример с перечислением записей метафайла [C#]

```
public bool DrawRecordsCallback(
  EmfPlusRecordType recordType, int flags,
  int dataSize, IntPtr recordData,
  PlayRecordCallback callbackData)
{
  // Это не сработает:
  //callbackData(recordType, flags, dataSize, recordData);

  byte[] arr = new byte[dataSize];

  // См. примечание
  if(recordData!=IntPtr.Zero)
    Marshal.Copy(recordData, arr, 0, dataSize);

  // Вызываем только выбранные пользователем команды
  if(frmItems.lbRecords.GetItemChecked(curRecord++))
    mfImage.PlayRecord(recordType, flags, dataSize, arr);

  return true;
}
```

Как видим, метод `Copy` позволяет перенести данные в управляемый "хип", что нам и требовалось.

Примечание

В книге Чарльза Петцольда [22] содержится очень похожий пример вызова `PlayRecord` с использованием `Marshal.Copy`. Но его пример страдает серьезным недостатком.

Версия Петцольда всегда вызывает метод `Copy`. При передаче 0 в параметре `recordData` этот метод попытается обратиться к несуществующему участку памяти. Автор книги сам признается, что его программа почему-то не в состоянии отображать создаваемые WinForms метафайлы, но работает с GDI EMF. Для исправления этой ошибки достаточно внести проверку такого условия, что и присутствует в вышеприведенном листинге.

Да, для достижения заветной цели (перебора записей метафайла в WinForms) нам пришлось немного потрудиться. Но эти трудности с лихвой окупаются другим замечательным качеством .NET — reflection. Любой элемент перечисления (даже такого гигантского, как `EmfPlusRecordType`), "знает" свое строковое имя, что позволяет легко создать версию приложения, перечисляющую записи выбранного метафайла поименно. Вот фрагмент демонстрационной программы (листинг 14.32):

Листинг 14.32. Код для заполнения списка записей [C#[

```
public bool EnlistRecordsCallback(
  EmfPlusRecordType recordType, int flags,
  int dataSize, IntPtr recordData,
  PlayRecordCallback callbackData)
{
  frmItems.lbRecords.Items.Add(recordType.ToString(), true);
  return true;
}
```

Просто, не правда ли? Метод `ToString` вернет строковое имя элемента перечисления, избавляя программиста от утомительного кодирования 253 строковых констант. На рис. 14.7 показан результат работы данного метода:

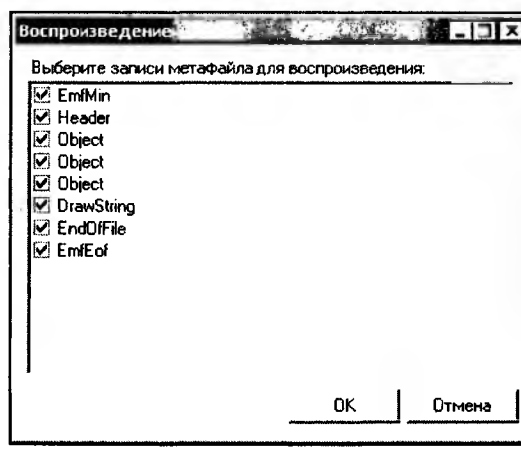
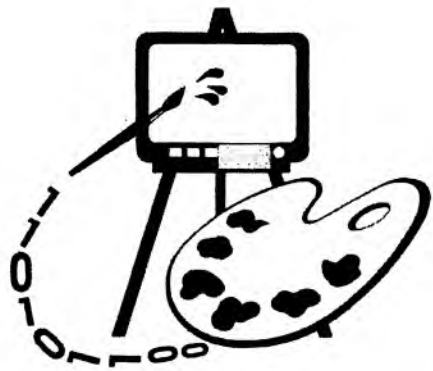


Рис. 14.7. Результат перечисления записей метафайла

Полный текст приложения (оно позволяет выполнять только выбранные пользователем команды метафайлов) находится на компакт-диске. Там же имеется откомпилированная версия, которую можно использовать для изучения содержимого различных метафайлов (например, входящих в комплект поставки Microsoft Visual Studio).

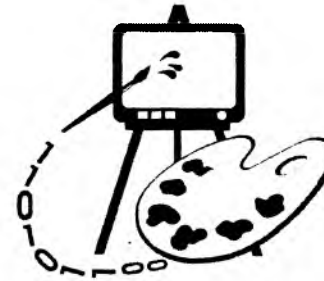


Часть V

НАЗНАЧЕНИЕ ГРАФИЧЕСКИХ БИБЛИОТЕК

Глава 15. Библиотеки OpenGL и DirectX

Глава 15



Библиотеки OpenGL и DirectX

В этой главе рассматриваются:

- назначение библиотек OpenGL и DirectX;
- пример использования библиотеки OpenGL для вывода на экран растрового и векторного изображений (программа FirstGL).

При программировании сложных, больших приложений целесообразно сконцентрировать основное внимание на задачах проекта и использовать для визуализации готовые графические библиотеки, которые позволяют избавиться от массы рутинной работы.

Необходимость создания специализированных библиотек для работы с графическими данными была обусловлена не только желанием сократить объем труда, но также и тем фактом, что стандартные средства GDI Windows работают с графикой довольно медленно. Поэтому ведущие фирмы-разработчики программного обеспечения объединили свои усилия и создали средства, позволяющие с наименьшими затратами работать с трехмерной графикой в операционной системе Windows.

В данной главе коротко рассказывается о назначении наиболее известных в настоящее время графических библиотек: OpenGL и DirectX. Приводится пример использования OpenGL для вывода на экран трехмерной сцены.

Для более подробного изучения библиотек можно порекомендовать литературу [3, 14, 20].

Кроме того, можно обратиться к документации Microsoft по DirectX (*DirectX 8.0 Programmer's Reference*), а также к электронной библиотеке Microsoft MSDN Library (*раздел Platform SDK, Graphics and Multimedia Services*).

15.1. Библиотека OpenGL

Прообразом библиотеки OpenGL стала библиотека IRIS GL, разработанная компанией Silicon Graphics для своих рабочих станций, которая оказалась

настолько удачной, что в настоящее время на ее основе разработан стандарт OpenGL. OpenGL — Open Graphics Library, открытая графическая библиотека. Термин "открытый" означает независимый от производителей. Библиотеку OpenGL могут производить разные фирмы и отдельные разработчики. Главное, чтобы библиотека удовлетворяла спецификации (стандарту) OpenGL и ряду тестов. Технология OpenGL лицензирована Microsoft и применяется в Win32 API. Доступ к функциям Win32 API может быть осуществлен из разных языков программирования: C, Delphi, Fortran и др. Общие принципы использования OpenGL в любой системе программирования одинаковы. OpenGL используется в приложениях моделирования и визуализации, применяется для вывода графических данных в системах автоматизированного проектирования (САПР), дизайнерских программах и т. д.

Процедуры OpenGL работают как с растровой, так и с векторной графикой и позволяют создавать двумерные и трехмерные объекты произвольной формы. Пространственные объекты могут быть представлены каркасными и тоновыми моделями. Для объекта может быть задан материал и наложена растровая структура. Объектами сцен являются также и источники света. Средства создания моделей объектов включают процедуры генерации стандартных трехмерных поверхностей, например сфер и правильных многогранников, кривых Безье и рациональных B-сплайнов (NURBS-сплайнов).

В библиотеке OpenGL имеются средства взаимодействия графических объектов, которые позволяют создавать эффекты прозрачности, тумана, смешивания цветов, выполнять логические операции над объектами (например, вычитание), накладывать трафарет, передвигать объекты сцены, лампы и камеры по заданным траекториям и т. д.

При работе с растровой графикой данными являются массивы пиксельных значений, создаваемые в программе или загружаемые из файла.

Единицей информации при работе с векторными объектами является вершина, из них состоят более сложные объекты. Программист создает вершины, указывает, как их соединять (линиями или многоугольниками), устанавливает координаты и параметры камер и ламп, а библиотека OpenGL берет на себя всю остальную работу по созданию изображения на экране. OpenGL хорошо подходит как для начинающих программистов, которым необходимо создать небольшую трехмерную сцену и не задумываться о деталях реализации алгоритмов трехмерной графики, так и для профессионалов, занимающихся программированием трехмерной графики, т. к. она предоставляет развитые механизмы управления графическими сценами и обеспечивает определенную автоматизацию.

С точки зрения программиста, библиотека OpenGL представляет собой множество команд, одна часть которых позволяет создавать двумерные и трехмерные объекты, а другая — управляет их отображением на экране.

Кроме широких возможностей и простоты в изучении, библиотека OpenGL обладает следующими достоинствами:

- Стабильность. Как уже отмечалось, на OpenGL существует стандарт. Все новшества и изменения предварительно объявляются и вносятся таким образом, чтобы гарантировать нормальную работу уже написанных программ.
- Надежность и переносимость. Все приложения, использующие OpenGL, гарантируют получение одинакового визуального эффекта вне зависимости от используемого оборудования и операционной системы.
- Простота использования. Библиотека OpenGL хорошо структурирована и включает драйверы основного оборудования, что освобождает разработчика от проблем со специфичностью различных графических устройств.

Основным недостатком библиотеки OpenGL считают ее сравнительную медлительность, что, видимо, является расплатой за простоту ее инициализации и использования.

Реализация Microsoft OpenGL включает в себя следующие компоненты:

- Набор базовых команд OpenGL (около 300) для описания форм объектов, преобразования координат, управления освещением, цветом, текстурой, туманом, вывода растровых картинок и т. д. Имена базовых команд в реализации на C начинаются с префикса `gl`.
- Библиотеку утилит OpenGL (GLU-библиотеку). Команды этой библиотеки дополняют базовые функции OpenGL и позволяют выполнять триангуляцию многоугольников, создавать и выводить стандартные фигуры (сферы, цилиндры и диски), строить сплайновые кривые и поверхности, обрабатывать ошибки. Имена команд библиотеки утилит в C-реализации начинаются с префикса `glu`.
- Дополнительную библиотеку OpenGL (AUX-библиотеку). Библиотека содержит функции управления окнами, обработки событий, управления цветовой палитрой, вывода стандартных 3D-объектов (тор, тетраэдр и др.), управления двойной буферизацией. Имена команд этой библиотеки в C-реализации начинаются с префикса `aux`.
- Функции, соединяющие OpenGL с Windows, — WGL-функции. Эти функции управляют контекстом воспроизведения, списками команд, шрифтами. Имена WGL-функций начинаются с префикса `wgl`.
- Win32-функции для управления форматом пикселов и двойной буферизацией. Win32-функции в Windows-приложениях используются вместо команд библиотеки AUX. Имена Win32-функций не имеют специальных префиксов.

В конце главы будет рассмотрен пример использования OpenGL.

15.2. Библиотека DirectX

Так же, как и в случае с OpenGL, причиной создания библиотеки DirectX явилась медлительность стандартных графических средств операционной системы Windows. Кроме того, желание сделать систему Windows стандартом для игровых программ подвигло Microsoft на создание технологии WinG, специально ориентированной на разработчиков компьютерных игр, которая и стала затем основой библиотеки DirectX.

DirectX представляет собой набор интерфейсов прикладного программирования (API) и программных инструментов, позволяющих создавать Windows-приложения со встроенным доступом к аппаратным компонентам, не зная подробностей аппаратной конфигурации конкретного компьютера. Другими словами, программисты получают унифицированный доступ к аппаратуре, причем без необходимости ограничиваться минимальными стандартными возможностями. Это позволяет ускорить работу с графикой до уровня DOS. Вдобавок DirectX содержит функции, позволяющие работать со звуком, портами ввода-вывода и другими устройствами. DirectX 8.0 состоит из следующих основных компонентов:

- DirectX Graphics — объединяет компоненты DirectDraw и Direct3D предыдущих версий библиотеки DirectX в единый интерфейс (API). Компонент специализируется на работе с графикой. Благодаря этому компоненту, программы получают прямой доступ к видеоадаптеру компьютера, что позволяет им очень быстро переносить изображения из памяти на экран. Библиотека спроектирована так, что может использовать все аппаратные возможности видеокарты по обработке изображений. Если какие-то требуемые возможности не реализованы аппаратно, то они эмулируются программно.
- DirectX Audio — объединяет компоненты DirectSound и DirectMusic предыдущих версий DirectX. Компонент предназначен для работы с устройствами воспроизведения звука. DirectSound работает значительно быстрее стандартных MCI-функций Windows, позволяет синхронизировать происходящее на экране со звуковыми эффектами, дает возможность замедлять и ускорять воспроизведение, выполнять смешивание звуков, создавать объемные звуковые эффекты и т. д.
- DirectInput — обеспечивает поддержку устройств ввода, например джойстика. Позволяет выполнить калибровку устройств, определить их состояние.
- DirectPlay — обеспечивает сетевую связь между компьютерами, организует взаимодействие между программами (сетевыми играми).
- DirectShow — используется для программирования мультимедиа-приложений, обеспечивает высококачественный "захват" и проигрывание мультимедийных потоков данных.

- DirectSetup — обеспечивает инсталляцию компонентов DirectX.

По мнению многих программистов, работающих с DirectX, основная сложность, возникающая при написании программ на основе этой библиотеки, заключается в определении возможностей аппаратуры и настройке драйверов. Этот недостаток, как и в случае с OpenGL, вытекает из основных достоинств библиотеки. Позволяя программисту почти напрямую работать с аппаратурой, библиотека требует от него большего внимания к "железу".

Вопрос выбора библиотеки OpenGL или DirectX надо решать, исходя из задач, которые должно выполнять приложение. И та и другая библиотеки встроены в операционные системы Windows. Однако интерфейс 3D-графики OpenGL встроен в архитектуру операционной системы на более высоком уровне, чем DirectX, и для связи с аппаратурой OpenGL опирается на DirectX. С другой стороны, по мнению некоторых экспертов, OpenGL лучше выполняет визуализацию (рендеринг) трехмерных сцен, чем выполняющий те же функции модуль Direct3D.

Таким образом, можно сделать следующие выводы.

- Качество визуализации трехмерных сцен с использованием OpenGL и DirectX если и отличается, то несущественно.
- Считается, что DirectX работает быстрее, чем OpenGL.
- DirectX в отличие от OpenGL имеет развитые компоненты управления звуком и взаимодействия с периферийными устройствами.
- OpenGL проще инициализировать.

Видимо поэтому OpenGL используется в основном не в играх, а в приложениях моделирования и визуализации. DirectX же применяется в мультимедийных приложениях, играх, тренажерах и других сложных программах, предусматривающих активное взаимодействие с пользователем или обработку больших потоков данных.

15.3. Пример использования библиотеки OpenGL

Создадим однодокументное MFC-приложение, которое будет выводить на экран движущуюся фигуру на фоне растрового изображения. Фигура будет представлять собой композицию из двух трехмерных объектов. Фигура будет вращаться вокруг своей оси и совершать круговые движения по экрану программы. В качестве заднего плана будем использовать растровую картинку, загружаемую в программу с помощью знакомого нам по *главе 11* класса `CRaster`. Визуализация трехмерной картинки будет происходить с использованием алгоритма "z-буфера" удаления невидимых линий *(см. разд. 7.3.3)*. Показ заднего плана будем включать или выключать коман-

дой **View | Background**. В режиме показа заднего плана движение объекта несколько замедляется. Это связано с тем, что алгоритму визуализации прибавляется работы.

Для реализации графики с использованием OpenGL программа должна, прежде всего, выполнить начальную инициализацию, которая включает следующие действия:

1. Подобрать и установить нужные параметры контекста воспроизведения.
2. Создать контекст воспроизведения.
3. Сделать созданный контекст воспроизведения активным. Программа может иметь несколько контекстов воспроизведения, но активным является только один.

После выполнения этих операций уже можно что-нибудь рисовать. В случае если настройки OpenGL по умолчанию не подходят, их можно изменить с помощью функции `glEnable`. Можно также настроить параметры сцены, например параметры освещения.

4. Следующее, о чем должна побеспокоиться ваша программа, — это обработка сообщения об изменениях размера вашего окна. В обработчике этого сообщения указывается часть окна, в которой будет располагаться контекст OpenGL. При этом контекст воспроизведения может занимать только часть окна, а остальная его часть может использоваться для размещения элементов управления (кнопки, поля ввода и т. п.) и других целей. Кроме того, требуется указать тип проекции, используемой в контексте отображения: перспективная или параллельная. В перспективной проекции две параллельные прямые сходятся вдалеке. В параллельной же проекции они всегда остаются параллельными.
5. Требуется установить точку наблюдения (точку, в которой находится камера или глаз наблюдателя) и точку, куда направлен "взгляд".
6. Задать ориентацию системы координат.

Итак, создадим заготовку простого однодокументного приложения (можно даже отказаться от панели инструментов и строки состояния). Назовем его FirstGL.

После создания каркаса приложения подключим к проекту библиотечные файлы OpenGL. Для этого открываем диалоговое окно свойств проекта (команда **Project | Settings**) и вкладку **Link**. В поле **Object/library modules** добавим имена файлов opengl32.lib, glu32.lib и glaux.lib (имена файлов разделяются пробелами без запятых). В файл StdAfx.h добавим строчки, подключающие заголовочные файлы:

```
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
```

15.3.1. Модификация класса облика

В классе CFirstGLView объявим переменную m_hGLRC типа HGLRC (указатель на контекст воспроизведения OpenGL) и переменную m_pDC (указатель на объект CclientDC). Интерфейс класса приведен в листинге 15.1. В класс CFirstGLView добавлены также объявления нескольких методов и переменных, назначение которых рассмотрим далее.

Листинг 15.1. Интерфейс класса CFirstGLView. Файл Firstvw.h

```
class CFirstGLView : public CView
{
protected: // create from serialization only
   CFirstGLView();
   DECLARE_DYNCREATE(CFirstGLView)
public:
   CFirstGLDoc* GetDocument();
   // Данные
   // Контекст устройства рисования
   CClientDC    *m_pDC;
   // Контекст воспроизведения OpenGL
   HGLRC    m_hGLRC;
   // Коэффициент пропорции размеров экрана
   double  m_dProportion;
   // Флаг "рисовать фон"
   BOOL     m_bViewBackground;

   // Операции
   // Установка параметров воспроизведения
   int    SetWindowPixelFormat(HDC);
   // Вывод всего изображения
   void   Display();
   // Вывод фона
   void   DisplayBackground();

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CFirstGLView)
   public:
   virtual void OnDraw(CDC* pDC);  // overridden to draw this view
   protected:
```

```
   virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
   virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
   virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
//}}AFX_VIRTUAL

// Implementation
public:
   virtual ~CFirstGLView();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
   //{{AFX_MSG(CFirstGLView)
   afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
   afx_msg void OnDestroy();
   afx_msg void OnSize(UINT nType, int cx, int cy);
   afx_msg void OnViewBackground();
   afx_msg void OnUpdateViewBackground(CCmdUI* pCmdUI);
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in Firstvw.cpp
inline CFirstGLDoc* CFirstGLView::GetDocument()
   { return (CFirstGLDoc*)m_pDocument; }
#endif
```

Метод `CFirstGLView::SetWindowPixelFormat()` предназначен для установки параметров воспроизведения OpenGL (листинг 15.2).

Листинг 15.2. Метод `CFirstGLView::SetWindowPixelFormat()`. Файл Firstvw.cpp

```
int CFirstGLView::SetWindowPixelFormat(HDC hDC)
{
   int GLPixelIndex;
```

```
// PIXELFORMATDESCRIPTOR - структура,
// определяющая характеристики контекста воспроизведения.
// Инициализируем структуру значениями для полноцветного RGB-режима
PIXELFORMATDESCRIPTOR pfd =
{
   sizeof(PIXELFORMATDESCRIPTOR),    // размер структуры
   1,                                // номер версии
   PFD_DRAW_TO_WINDOW |     // разрешаем вывод в окно или на устройство
   PFD_SUPPORT_OPENGL |     // поддержка OpenGL
   PFD_DOUBLEBUFFER,        // двойная буферизация
   PFD_TYPE_RGBA,           // режим RGB
   24,                      // 24-битовая глубина цвета
   0, 0, 0, 0, 0, 0,        // игнорируем установки для битовых
                            // плоскостей и их смещения
   0,                       // без альфа-буфера
   0,                       // без смещения битов
   0,                       // без буфера-накопителя
   0, 0, 0, 0,              // без смещения бит в буфере-накопителе
   32,                      // размер z-буфера
   0,                       // без буфера-трафарета
   0,                       // и без вспомогательного буфера
   PFD_MAIN_PLANE,          // основная плоскость
   0,                       // резервный компонент
   0, 0, 0                  // без масок слоев
};

// Находит формат пикселов контекста устройства (монитора),
// наиболее близкий к заданному формату пикселов
// GLPixelIndex - номер поддерживаемого пиксельного формата
 GLPixelIndex = ChoosePixelFormat( hDC, &pfd);
if(GLPixelIndex==0) // Выбираем индекс формата по умолчанию
{
   GLPixelIndex = 1;
   // Получаем параметры режима
   if(DescribePixelFormat(hDC,GLPixelIndex,
                          sizeof(PIXELFORMATDESCRIPTOR),&pfd)==0)
      return 0;
 }
// Устанавливаем режим
```

```
    if (SetPixelFormat( hDC, GLPixelIndex, &pfd)==FALSE)
       return 0;

    return 1;
}
```

Метод `SetWindowPixelFormat()` вызывается из метода-обработчика сообщения `WM_CREATE`, добавленного в класс `CFirstGLView` с помощью ClassWizard (листинг 15.3). В методе `OnCreate()` создаются контекст Windows-окна программы и совместимый с ним контекст OpenGL, а также устанавливаются параметры визуализации трехмерной сцены.

Листинг 15.3. Метод `CFirstGLView::OnCreate()`. Файл Firstvw.cpp

```
int CFirstGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
       return -1;
    // Создаем контекст клиентской части окна
    if( (m_pDC = new CClientDC(this))==NULL)
       return -1;
    //
    if(SetWindowPixelFormat(m_pDC->m_hDC)==FALSE)
       return -1;
    // Создаем контекст отображения OpenGL
    if( (m_hGLRC = wglCreateContext(m_pDC->m_hDC)) == NULL)
       return -1;
    // Делаем контекст отображения активным
    if(wglMakeCurrent(m_pDC->m_hDC, m_hGLRC)==FALSE)
       return -1;
    // Устанавливаем параметры отображения
    // Параметры материала
    // отражение фонового света
    GLfloat  mat_ambient[4] = {0.2, 0.2, 0.2, 1.0};
    // диффузионное отображение
    GLfloat  mat_diffuse[4] = {0.8, 0.8, 0.8, 1.0};
    // зеркальное отражение
    GLfloat  mat_specular[4] = {1.0, 1.0, 1.0, 1.0};
    // интенсивность зеркального отражения
    GLfloat  mat_shineness = 50.0;
```

```
    // Устанавливаем параметры материала
    glMaterialfv(GL_FRONT,GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT,GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT,GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT,GL_SHININESS, &mat_shineness);

    // Задаем положение источника света
    GLfloat  pos[4] = {-100, 100, 100, 0};
    glLightfv(GL_LIGHT0, GL_POSITION, pos);

    // Активизируем настройки:
    // использовать текущий цвет для задания свойств материала
    glEnable(GL_COLOR_MATERIAL);
    // для вычисления цвета использовать текущие параметры
    glEnable(GL_LIGHTING);
    // учитывать источник света №0
    glEnable(GL_LIGHT0);
    // проводить тест глубины
    glEnable(GL_DEPTH_TEST);
    // выводить на экран пикселы с наименьшими z-координатами
    glEnable(GL_LESS);

    // можно задать, как минимум, 8 источников
    return 0;
}
```

Для того чтобы размер окна OpenGL соответствовал размеру Windows-окна программы, определим в классе `CFirstGLView` с помощью ClassWizard метод обработки сообщения `WM_SIZE` (листинг 15.4).

Листинг 15.4. Метод `CFirstGLView::OnSize()`. Файл Firstvw.cpp

```
void CFirstGLView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // Вывод осуществляется в экранно-ориентированную
    // прямоугольную область окна OpenGL - видовой порт
    glViewport(0,0,cx,cy); // Устанавливаем размеры порта
    m_dProportion=(double)cy/cx; // пропорции окна
```

```
    glMatrixMode( GL_PROJECTION ); // Делаем активной матрицу GL_PROJECTION
    glLoadIdentity();
    // Устанавливаем масштаб по осям координат пропорционально
    // соотношению размеров окна.
    // Если окно станет прямоугольным, пропорции фигуры не изменятся
    glOrtho(-5./m_dProportion, 5./m_dProportion, -5., 5., 0, 10);
    // Направление взгляда
    gluLookAt( 0,0,5, 0, 0, 0,  0, 1, 0 );

    glMatrixMode( GL_MODELVIEW ); // Делаем активной матрицу GL_MODELVIEW
}
```

После завершения работы программы (закрытия окна) хорошим тоном будет вернуть все ресурсы, которые мы позаимствовали у системы. Поэтому добавим в класс `CFirstGLView` обработку еще одного сообщения `WM_DESTROY` (листинг 15.5).

Листинг 15.5. Метод `CFirstGLView::OnDestroy()`. Файл Firstvw.cpp

```
void CFirstGLView::OnDestroy()
{
    // Делаем контекст отображения неактивным
    if(wglGetCurrentContext()!=NULL)
        wglMakeCurrent(NULL, NULL);
    // Уничтожаем контекст отображения
    if(m_hGLRC!=NULL)
    {
        wglDeleteContext(m_hGLRC);
        m_hGLRC = NULL;
    }

    // Уничтожаем контекст устройства
    if(m_pDC)
        delete m_pDC;

    CView::OnDestroy();
}
```

Наконец-то пришла пора описать методы, которые предназначены для рисования (листинг 15.6). Метод `CFirstGLView::Display()` выводит на экран дви-

жущиеся трехмерные объекты. Метод `CFirstGLView::DisplayBackground()` вызывается из метода `Display()` и предназначен для вывода на экран фонового изображения. В качестве фона используется растровая картинка, загруженная при создании объекта-документа.

Листинг 15.6. Методы `CFirstGLView::Display()` и `CFirstGLView::DisplayBackground()`. Файл Firstvw.cpp

```
void  CFirstGLView::Display(void)
{
   double    rf=.5,     // радиус фигуры
             rt=3.,     // радиус траектории движения
             dalfa=2., // шаг поворота фигуры вокруг своей оси
             dbeta=1.; // шаг поворота траектории движения
   static double
             alfa=0.,  // угол поворота фигуры вокруг своей оси
             beta=0.;  // угол поворота траектории движения
   // Вычисляем положение фигуры
   double x=rt*cos(beta*3.14/180.);
   double y=rt*sin(beta*3.14/180.);
   if((beta+=dbeta)>360) beta=0;
   if((alfa+=dalfa)>360) alfa=0;
   // Рисуем сцену

   glClearColor(1.0, 1.0, 1.0, 1.); // цвет фона окна
   // Подготовка буфера
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
       DisplayBackground();    // рисуем фон
       glTranslated(x,y,0);    // смещаем оси координат
       glRotated(alfa, 1.0, 1.0, 1.0); // поворачиваем оси координат
       glColor3d(1.0, 0.0, 0.0);       // устанавливаем цвет
       auxSolidSphere(rf);             // сплошная сфера
      //auxWireSphere(rf);             // проволочная сфера
       auxSolidTorus(rf, rf*3);        // сплошной тор
      //auxWireTorus(rf, rf*3);        // проволочный тор

      // Поворачиваем и смещаем оси координат обратно,
      // чтобы преобразование не накапливалось
      glRotated(-alfa, 1.0, 1.0, 1.0);
         glTranslated(-x,-y,0);
```

```
   glFinish();  // Закончить вывод и преобразования и отобразить объекты
   SwapBuffers(wglGetCurrentDC()); // Выводим на экран
}

void  CFirstGLView::DisplayBackground()
{
   glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
   // Получаем указатель на растровую картинку фона
   CRaster *pImage=GetDocument()->GetBackground();
   // Проверяем условия, при которых картинка может быть показана
   if(!m_bViewBackground || pImage==NULL || pImage->GetBMWidth()==NULL)
      return;
   // Позиция картинки
   glRasterPos3d(-5./m_dProportion, -5, -5);
   // Прозрачность картинки задается параметром от 0 до 1
   // 0 — прозрачная, 1 — непрозрачная,
   glPixelTransferf(GL_ALPHA_SCALE, 1);
   // Выводим картинку
   glDrawPixels(pImage->GetBMWidth(), pImage->GetBMHeight(),
            GL_BGR_EXT, GL_UNSIGNED_BYTE,
            pImage->GetBMDataPtr());
}
```

Для того чтобы можно было включать или выключать показ заднего плана, добавим в меню **View** программы команду **Background**, а в класс `CFirstGLView` — методы-обработчики сообщений, связанных с этой командой (листинг 15.7).

Листинг 15.7. Обработчики сообщений команды View-Background. Файл Firstvw.cpp

```
void CFirstGLView::OnViewBackground()
{
   m_bViewBackground=!m_bViewBackground;
}

void CFirstGLView::OnUpdateViewBackground(CCmdUI* pCmdUI)
{
   pCmdUI->SetCheck(m_bViewBackground);
}
```

Метод `CFirstGLView::OnViewBackground()` управляет значением переменной `m_bViewBackground` (первоначально установленной в `FALSE` конструктором класса `CFirstGLView`).

Метод `CFirstGLView::OnUpdateViewBackground()` ставит "птичку" напротив имени команды **Background** в меню, если значение `m_bViewBackground` равно `TRUE`.

15.3.2. Модификация класса документа

В класс документа требуется добавить лишь объект класса `CRaster` и метод, возвращающий указатель на него.

```
CRaster m_BackgroundImage;
CRaster* GetBackground(){return &m_BackgroundImage;};
```

Загрузка изображения будет выполняться методом `CRaster::LoadBMP()` в конструкторе класса `CFirstGLDoc` (листинг 15.8).

Листинг 15.8. Конструктор класса `CFirstGLDoc`. Файл Firstdoc.cpp

```
CFirstGLDoc::CFirstGLDoc()
{
   m_BackgroundImage.LoadBMP("Car.bmp");
}
```

Загрузка изображения выполняется из файла с именем Car.bmp, который должен находиться в одном каталоге с приложением (рабочем каталоге).

15.3.3. Модификация класса приложения

Обновление позиции трехмерного объекта происходит каждый раз при вызове метода `CFirstGLView::Display()`. Поэтому для того чтобы происходило движение объекта, этот метод должен периодически вызываться. Вызывать этот метод можно, например, при обработке сообщений таймера. Для этого нам пришлось бы добавить в программу таймер и метод обработки его сообщений. Однако мы используем другой подход. Виртуальный метод `CWinApp::OnIdle()` вызывается каждый раз, когда очередь сообщений окна пуста и приложение простаивает. С помощью ClassWizard переопределим метод `OnIdle()` в классе `CFirstGLApp` (листинг 15.9).

Листинг 15.9. Метод `CFirstGLApp::OnIdle()`. Файл FirstGL.cpp

```
BOOL CFirstGLApp::OnIdle(LONG lCount)
{
```

```
    // Вызываем метод рисования
    ( (CFirstGLView*) ((CMainFrame*)m_pMainWnd)->GetActiveView() )->
       Display();
    return 1;// Повторять вызов OnIdle
}
```

15.4. Заключение

Результат работы программы FirstGL показан на рис. 15.1. Конечно, прочитав эту главу, вы не научитесь в совершенстве использовать OpenGL. Однако кое-какую "пищу для ума" можно извлечь из рассмотренного материала, можно поэкспериментировать с программой. Например, заменив в методе `CFirstGLView::Display()` функцию `auxSolidTorus()` на `auxWireTorus()`, можно получить изображение каркасного (проволочного) тора, можно изменить траекторию движения объектов, составить из фигур новую композицию, поменять положение источника освещения и т. д.

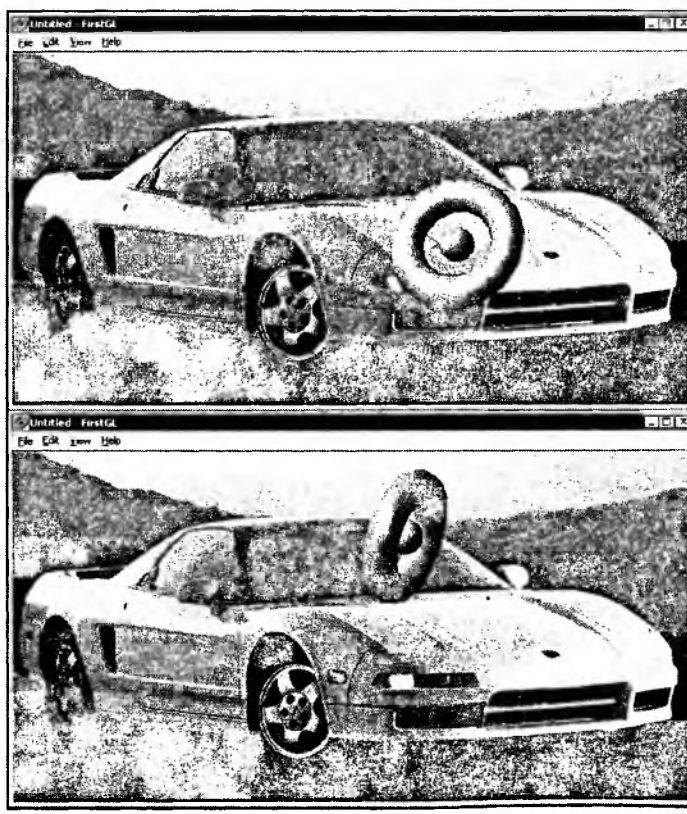


Рис. 15.1. FirstGL в работе

Заключение

Вот и пришла пора сказать что-нибудь на прощание. Мы с вами рассмотрели довольно большой круг задач, связанных с компьютерной графикой: рисовали линии и разные фигуры, выполняли преобразования на плоскости и в пространстве, поработали и с векторными, и с растровыми изображениями, познакомились с богатыми возможностями новой графической библиотекой GDI+. Надеемся, полученные знания будут вам полезны и послужат созданию новых суперпрограмм. Мы постарались сделать книгу понятной и интересной. Насколько это удалось, судить, конечно, вам.

Вот так всегда — чем больше учишься, тем лучше понимаешь, как много еще можно освоить. Это и хорошо — имеются перспективы для развития. Например, изучение работы с библиотеками DirectX и OpenGL. Интересным направлением также является написание подключаемых (plug-in) модулей для графических редакторов. В *главе 11* упоминалось о варианте реализации графических фильтров в виде DLL-модулей. Компания Adobe Systems Inc. построила архитектуру своих приложений таким образом, что их функциональные возможности могут динамически наращиваться путем подключения модулей plug-in. О том, как создавать plug-in модули можно прочитать в документации (Software Developers Toolkit, SDK), доступ к которой свободен на сайте **http://www.adobe.com**. Документация содержит подробное описание и примеры программ, изучив которые вы сможете написать свой собственный фильтр, например к Adobe Photoshop. Другой известный производитель программного обеспечения для работы с графикой и мультимедиа, фирма Ulead Systems, также поддерживает "плагинную" идеологию своих продуктов и предлагает разработчикам SDK-документацию (**http://www.ulead.com**).

Роберт Шекли в одном из рассказов дал следующее определение: "Инженеры! — Творчески одаренные и рационально мыслящие ученые, которые способны выстроить планету в любое время в любом месте". Мы желаем и вам соответствовать данному описанию. Успехов!

Описание содержимого компакт-диска

Папки	Описание	Главы
\Sources	Исходные тексты и выполняемые файлы программ-примеров.	
\Sources\Bezier	Программа Bezier для построения сплайновых кривых различных типов.	8
\Sources\BMViewer	Программа BMViewer для просмотра и редактирования растровых изображений.	11
\Sources\FirstGL	Пример использования OpenGL для визуализации трехмерной сцены и вывода на экран растрового изображения.	15
\Sources\MFCApp	"Минимальная MFC-программа".	2
\Sources\Painter1	Программа Painter1. Реализовано рисование линий.	3
\Sources\Painter2	Программа Painter2. Вывод на принтер. Рисование фигур. Установка размеров листа.	4
\Sources\Painter3	Программа Painter3. Реализованы функций редактирования, рисование полигонов, преобразования на плоскости.	6
\Sources\Painter4	Программа Painter4. Преобразования в трехмерном пространстве. Поверхности и линии уровня.	7
\Sources\Painter4.1	Программа Painter4.1. Работа с графическими ресурсами. Добавлена инструментальная панель и возможность заливки фигур растровым шаблоном.	9
\Sources\Painter4.2	Программа Painter4.2. Добавлена возможность экспорта изображений в растровый BMP-файл.	10

окончание

Папки	Описание	Главы
\Pics\Painter	Рисунки, созданные в программах Painter.	3–7, 9, 10
\Pics\Samples	Примеры растровых изображений после обработки в программе BMViewer.	11
\Sources\GDIPlus\DemoVC	Первая программа на C++ с использованием GDI+. Загрузка и вывод изображения в формате GIF, вывод полупрозрачного текста с градиентом.	12
\Sources\GDIPlus\DemoCS	Первая программа на C# для среды .NET. Те же действия с использованием WinForms.	12
\Sources\GDIPlus\Codecs	Просмотр поддерживаемых библиотекой графических форматов и свойств их кодеков.	13
\Sources\GDIPlus\Batterfly	Программа на C++ — хранение изображений GIF в ресурсах программы, вывод анимированного изображения, двойная буферизация.	13
\Sources\GDIPlus\Animated	Пример работы с анимированными растрами GIF для .NET.	13
\Sources\GDIPlus\RevFrame	Иллюстрация прямого доступа к растрам в среде .NET.	13
\Sources\GDIPlus\Clock	"Программа-часы" на C#. Демонстрация возможностей векторных примитивов с использованием WinForms.	14
\Sources\GDIPlus\Curves	Построение кубических сплайнов и кривых Безье с помощью библиотеки GDI+.	14
\Sources\GDIPlus\MetaGen	Запись изображения в метафайл. Программа на C++.	14
\Sources\GDIPlus\PlayMeta	Разбор метафайлов и выборочное воспроизведение. Пример для .NET (C#).	14

Список литературы

1. Абламейко С. В., Лагуновский Д. М. Обработка изображений: технология, методы, применение. Учеб. пособие. — Мн.: Амалфея, 2000. — 304 с.

   Описаны теоретические основы и методы обработки изображений, преобразование полутоновых изображений, распознавание образов.

2. Аммерал Л. Принципы программирования в машинной графике/Пер. с англ. — М.: Сол Систем, 1992. — 224 с. — Третья книга серии "Машинная графика на языке Си" из 4-х книг этого автора.

   Рассматриваются вопросы аналитической и проективной геометрии и программирования в машинной графике. Алгоритмы доведены до "готовых к работе" программ на языке Си. Подробно и доступно излагаются основы компьютерной графики. Изложение проиллюстрировано многими примерами. Немного старовat стиль программирования, однако очень полезно почитать с точки зрения эффективного использования языка Си.

3. Бартеньев О. В. Графика OpenGL: программирование на Фортране. — М.: ДИАЛОГ-МИФИ, 2000. — 368 с.

   Рассматриваются возможности графической библиотеки OpenGL. Примеры приводятся на языке Фортран, однако принципы использования OpenGL одинаковы для всех языков, поэтому книга полезна и для программистов, предпочитающих другие языки. В приложении приведен пример программирования с использованием OpenGL на языке Си.

4. Биллиг В. А., Мусикаев И. Х. Visual C++ 4. Книга для программистов. — М.: Русская редакция, 1996. — 352 с.

   В книге рассмотрены основы программирования в Microsoft Visual C++. Хорошая книга для самостоятельного изучения.

5. Линдли К. Практическая обработка изображений на языке Си/Пер. с англ. — М.: Мир, 1996. — 512 с.

   Рассмотрены аппаратные и программные средства вывода изображений на экран. Классические методы и алгоритмы обработки изображений. Приведена программная реализация алгоритмов на языке Си.

6. Мюррей Д., Райпер У. ван. Энциклопедия форматов графических файлов/Пер. с англ. — К.: BHV, 1997. — 672 с., CD-ROM.

Подробно рассмотрено большое количество форматов графических файлов, методы сжатия. На прилагаемом CD содержатся программы чтения/записи и преобразования различных форматов, исходные коды, а также Интернет-адреса, где можно добыть свежую информацию и программы.

7. Нортон П., Макгрегор Р. Руководство Питера Нортона. Программирование в Windows 95/NT 4 с помощью MFC: В 2-х кн. — М.: СК-Пресс, 1998. — 1176 с. (в 2-х кн.), CD-ROM.

Пособие по программированию на Си++ в 32-разрядной Windows с использованием MFC. Рассмотрен широкий круг вопросов, начиная с функции WinMain и главного цикла выбора сообщений, графический интерфейс, управление страницами памяти, написание DLL, работа с OLE, сетевые взаимодействия, ActiveX и работа с Web-страницами. Изложение проиллюстрировано интересными примерами. Код программ содержится на CD.

8. Рассохин Д. От Си к Си++. — М.: ЭДЕЛЬ, 1993. — 128 с.

В книге рассматриваются отличия языка Си++ от Си. Объясняются принципы объектно-ориентированного программирования. Приводится большое число примеров, позволяющих понять особенности ООП.

9. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной Windows/Пер. с англ. — 4-е изд. —СПб: Питер; М.: Русская редакция, 2001. — 752 с.

Подробно рассматривается использование функций API Windows. Использование виртуальной памяти. Управление процессами и потоками. Разработка DLL-библиотек.

10. Рихтер Дж. Программирование на платформе Microsoft. NET Framework. — М.: Русская редакция, 2002. — 512 с.: ил.

Новая книга признанного технического писателя знакомит профессиональных программистов с принципами устройства платформы . NET "изнутри". Специалисты Microsoft признали эту книгу лучшим введением в среду . NET.

11. Роджерс Д., Адамс А. Математические основы машинной графики. — М.: Машиностроение, 1980. — 240 с.

Введение в принципы компьютерной графики, знакомство читателя с основными методами реализации графических эффектов. Книга не привязана к конкретной платформе или технологии, а рассматривает необходимые понятия и дает полезный математический аппарат.

12. Роджерс Д. Алгоритмические основы машинной графики. — М.: Мир, 1989. —512 с.

Книга содержит анализ алгоритмов и методов современных графических систем, особое внимание уделено методам растровой графики. Алгоритмы доведены до программ на псевдокоде, легко преобразуемом в языки Паскаль, Фортран и Бейсик. Книга изобилует иллюстрациями и примерами, содержит задания для самостоятельного решения.

13. Страуструп Б. Дизайн и эволюция C++/Пер. с англ. — М.: ДМК-Пресс, 2000. — 448 с. Серия "Для программистов".

В книге описан процесс проектирования и разработки языка Си++, изложены принципы правильного применения объектно-ориентированного языка программирования.

14. Томпсон Н. Секреты программирования трехмерной графики для Windows 95/Пер. с англ. — СПб.: Питер, 1997. — 325 с.

Рассматривается использование библиотеки DirectX для программирования трехмерной графики.

15. Шилдт Г. MFC: основы программирования/Пер. с англ. — К.: BHV, 1997. — 560 с.

В книге подробно рассматривается использование MFC.

16. Шикин Е. В., Боресков А. В. Компьютерная графика. Динамика, реалистические изображения. — М. Диалог-МИФИ, 1995. — 288 с.

В книге изложены основные понятия и методы компьютерной графики: растровые алгоритмы, алгоритмы построения геометрических сплайнов, методы удаления скрытых линий и поверхностей, закрашивание, трассировка лучей, реализация алгоритмов на языке Си.

17. Шикин Е. В., Плис А. И. Кривые и поверхности на экране компьютера. Руководство по сплайнам для пользователей. — М.: Диалог-МИФИ, 1996. — 240 с.

В книге описаны одномерные кубические и двумерные бикубические интерполяционные и сглаживающие сплайны. Приведены примеры их программной реализации.

18. Шлихт Г. Ю. Цифровая обработка цветных изображений. — М.: ЭКОМ, 1997. —336 с.

Книга знакомит с основами теории цвета и ее применением в полиграфии, принципами работы сканеров, мониторов и принтеров; описывает работу с графическими и мультимедиа приложениями. Подробно описаны операции обработки растровых изображений, принципы алгоритма сжатия растровых изображений JPEG, математические основы цифровой фильтрации.

19. Эйткен П., Джерол С. Visual C++ для мультимедиа. — К.: КОМИЗДАТ, 1996. — 384 с.

Рассматривается практическое применение Visual C++ для создания мультимедийных приложений, поддерживающих гипертекст, вывод графических файлов, анимацию, воспроизведение аудио- и видеофайлов.

20. Энджел Э. Интерактивная компьютерная графика. Вводный курс на базе OpenGL — К.: Вильямс, 2001. — 592 с.

Рассматриваются математические принципы построения трехмерных объектов и методика программирования с использованием OpenGL. Обсуждаются современные тенденции создания интерактивных графических систем типа "клиент/сервер" и методика разработки графических программ. Рассматривается применение графических средств при визуализации результатов научных расчетов.

21. Фень Юань. Программирование графики для Windows. — СПб.: Питер, 2002. — 1072 с.: ил.

Очень обстоятельное, скрупулезное и детальное исследование интерфейса Windows GDI. Автор рассматривает принципы его внутреннего устройства, архитектуру графического ядра, драйверов графических устройств и системы печати. Читатель узнает разницу между реализациями GDI для систем Windows NT и 9x, а также познакомится с программным интерфейсом DirectX.

22. Charles Petzold. Programming Windows with C#. Microsoft Press, Book and CD-ROM edition (December 19, 2001). — 1200 p.

Книга содержит довольно полное руководство по программированию . NET-приложений на языке C#. Подробно рассматриваются событийная архитектура программ для WinForms и вопросы программирования с использованием библиотеки GDI+ для этой платформы.

Интернет-ресурсы

1. **http://msdn.microsoft.com** — сайт Microsoft содержит массу технической документации, программных кодов примеров, статей.
2. **http://www.codeguru.com** — сайт посвящен программированию для Windows, содержит большое количество примеров программ на различных языках. Среди прочего есть и раздел компьютерной графики.
3. **http://www.codeproject.com** — более молодой сайт по Windows-программированию, но очень быстро достиг популярности CodeGuru. Есть и статьи, посвященные библиотеке GDI+.

4. **http://www.gotdotnet.com** — сайт, созданный Microsoft для продвижения инициативы . NET Framework. Содержит большое количество учебных материалов, статей, примеров программ и интервью с создателями этой платформы.

5. **http://www.gotdotnet.ru** — версия этого сайта на русском языке.

6. **http://www.rsdn.ru** — Russian Software Developer Network, "сеть русскоязычных программистов". Проект, собравший "под крылом" статьи и форумы по программированию в Рунете (в основном, для платформ Windows и . NET Framework). Среди прочего, содержит статьи и форумы по компьютерной графике, мультимедиа и таким популярным программным интерфейсам, как GDI, GDI+, OpenGL, DirectX.

7. **http://opengl.org.ru** — сайт посвящен программированию графики с использованием библиотеки OpenGL. На нем можно найти много полезной информации по этому вопросу, в частности электронную версию книги Игоря Тарасова "Введение в OpenGL". Книга позволяет изучить основы использования библиотеки OpenGL. Также на сайте (в разделе Links) содержатся ссылки на другие интернет-ресурсы, посвященные вопросам компьютерной графики.

8. **http://www.bobpowell.net/gdiplus_faq.htm** — "Bob Powell's GDI+ FAQ", набор часто задаваемых вопросов для программистов GDI+ на платформе . NET Framework.

9. **http://www.enlight.ru** — сайт посвящен разработке мультимедийных проектов, как его характеризуют авторы — "объемный проект, содержащий различную информации по демомейкингу, его истории и терминологии, программированию аудио- и видеоэффектов, математике, компьютерной графике, звуку"; "различная информация по программированию 3D-графики".

Предметный указатель

D

E

F

G

H

I

J

L

M

O

P

R

S

А

Б

В

Г

Д

Е

З

И

К

Л

М

Т

У

Ф

Ц

Ш, Э

Поляков Алексей Юрьевич, кандидат технических наук, несколько лет преподавал курс компьютерной графики в Томском университете систем управления и радиоэлектроники, разрабатывал визуальные технологии проектирования технических систем, автор программного обеспечения и методики изготовления варио- и стереофотографий. Ведущий инженер "Moonlight Cordless Ltd.", руководитель программных проектов в области обработки цифрового видео.

Брусенцев Виталий Александрович, специалист в области автоматизации процессов документооборота, разработчик математических моделей обработки металлов и других процессов, библиотек алгоритмов для графического представления результатов экспериментов с помощью градиентов цвета и трехмерных изображений, создатель компьютерных игр и автор статей в журнале RSDN Magazine.

Современные методы программирования компьютерной графики

Методы и алгоритмы компьютерной графики

в примерах на Visual C++

2-е издание

Курс компьютерной графики охватывает широкий круг вопросов, начиная с базовых понятий и заканчивая приемами профессионального программирования. Вероятно, это первая книга на русском языке с описанием замечательных возможностей новой графической библиотеки GDI+ и способов ее использования. Весь теоретический материал сопровождается программной реализацией. Приведенные программы могут стать базой для дальнейшей самостоятельной разработки. Включенные в издание 175 рисунков помогут освоить теорию и практику. Книга ориентирована на программистов, студентов и преподавателей компьютерных специальностей.

Компакт-диск содержит текст программ и примеры изображений

ISBN 5-94157-377-4

Уровень пользователя	Средний/высокий
Категория	Программирование

БХВ-Петербург 198005 Санкт-Петербург Измайловский пр. 29
E-mail: mail@bhv.ru Internet: www.bhv.ru Тел.: (812) 251-4244 Факс: (812) 251 1295