Министерство образования Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

А.Ю. Поляков

МАШИННАЯ ГРАФИКА И ГЕОМЕТРИЧЕСКОЕ МОДЕЛИРОВАНИЕ

Учебное пособие

2001

Поляков Ю.А.
Машинная графика и геометрическое моделирование: Учебное пособие. – Томск: Томский межвузовский центр дистанционного образования, 2001. – 142 с.

1. ВВЕДЕНИЕ

Термин «компьютерная графика» стал неотъемлемой частью современности и объединяет довольно широкий круг операций по обработке графической информации с помощью компьютера. Можно выделить несколько основных направлений в компьютерной графике:

1. **Визуализация научных (расчетных или опытных) данных.** Большинство современных математических программных пакетов (например, Maple, Matlab) имеют средства для изображения 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 каф. КСУП ТУСУР, визуальные образы, выводимые на экран, являются основой для решения математических и проектных задач. Визуализация полученных опытным путем данных позволяет представить информацию в удобной для анализа форме и широко используется, например, в томографии.

2. **Геометрическое проектирование и моделирование.** Этим направлением компьютерной графики решаются задачи начертательной геометрии – построения 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.

3. **Распознавание образов.** Задачей данного направления является распознавание и классификация графический информации. Например, при сканировании текста возникает проблема перевода «фотографии» текста в набор отдельных символов формирующих слова. Программное обеспечение многих современных сканеров позволяет выполнить такую операцию. Кроме того, существуют специализированные программы распознавания текста, например, FineRiader. Задачи распознавания образов решаются при анализе аэро- и космических фотоснимков, в различных системах контроля и т.д.

4. **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 программами из этой области компьютерной графики можно назвать Photoshop (обработка растровых изображений), CorelDraw (создание векторной графики), 3D Studio Max (трехмерное моделирование).

5. **Виртуальная реальность.** Решаются задачи моделирования внешнего мира в различных приложениях от компьютерных игр до тренажеров.

Приведенная классификация направлений применения компьютерной графики является во многом условной, и, наверняка, найдутся задачи, которые можно отнести сразу к нескольким пунктам.

В данном курсе основное внимание уделено рассмотрению основ использования компьютерной графики для визуализации расчетных данных.

Программирование под 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 типы:

HANDLE – 32-разрядное целое используется в качестве дескриптора (числового идентификатора какого-либо ресурса);

HWND – дескриптор окна;

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 – указатель на строку символов типа CHAR;

LPCSTR – константный указатель на строку символов типа CHAR;

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.

Работа Windows-программ (их часто еще называют *приложениями*) 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 этой процедуры начинается выполнение всех Windows-программ. WinMain должна выполнить следующие основные действия:

1. Определить и зарегистрировать класс (в смысле вид, стиль, тип) 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, который только и делает, что возвращает необработанные сообщения обратно Widows.

Затем Windows вызывает функцию окна программы с данным сообщением в качестве аргумента. Анализируя сообщение, функция окна инициирует соответствующие операции. Сообщения, обработка которых не предусмотрена функцией окна, передаются обратно Windows, которая выполняет обработку «по умолчанию». Ниже приведен фрагмент минимальной программы для Windows.

```
// Каркас программы для Windows 95
#include <windows.h>

//Определение функции окна
LRESULT CALLBACK WinFunction(..., UINT message, ...);

// Имя класса окна
char szWindowName[]= «MyFirstWindow»;

// функция WinMain
int WINAPI WinMain(...)
{
     HWND hwnd;
     MSG msg;
     WNDCLASSEX wcl;
     // Определение класса окна. Задается стиль, форма курсора,
фон и др.
     ...
wcl.lpszClassName = szWindowName;  // Имя класса окна
wcl.lpfnWndProc = WinFunction;     // Функция окна
...
// Регистрация класса окна
if(!RegisterClassEx(&wcl)) return 0;

// Создание окна
hwnd=CreateWindow(  szWindowName,  /* Имя класса окна*/
«Каркас программы для Windows 95», /* Заголовок*/
... /* Положение, размер и др. */
);
     // Отображение окна
     ShowWindow(hwnd);

     // Цикл обработки сообщений
     while(GetMessage(&msg, ...))
     {
          DispatchMessage(&msg); // Возвращает управление Windows
     }
     return msg.wParam;
}

// Функция окна вызывается Windows, которая передает ей сообщения
LRESULT CALLBACK WinFunction (..., UINT message, ...)
{
     switch(message){
     case WM_DESTROY: // Завершение программы
```

```
            PostQuiMessage(0);
            break;
      default: // Передает необработанные сообщения Windows
            return DefWindowProc(..., message, ...);
};
```

Рассмотрим выполнение данной программы. Схематично оно показано на рис. 1.1.

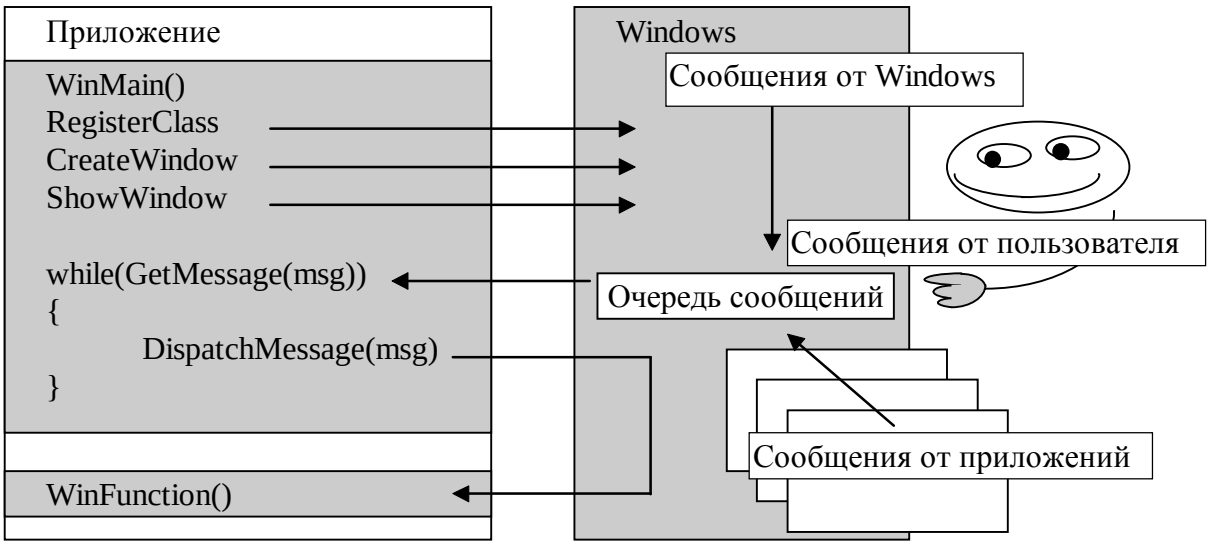


Рис. 1.1. Схема обработки сообщений приложением

Все программы, написанные для Windows 95, 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 95. Функция окна, используемая в программе, называется WinFunction(). Она объявлена как функция обратного вызова, так как вызывается Windows для взаимодействия с программой. Программа начинает выполнение с вызова функции WinMain(). Первое, что выполняет функция WinMain(), – определяет класс окна путем заполнения полей структуры WINDCLASSEX. С помощью данной структуры программа указывает Windows функцию окна и стиль окна. Регистрация класса окна выполняется с помощью API-функции RegisterClassEx(). После этого создается окно с помощь 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 получаются через параметр функции типа структуры MSG. Если очередь сообщений пуста, то вызов GetMessage() передает управление Windows. При завершении работы с программой 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 95 значение кода возврата.

Любое приложение взаимодействует с Windows через Application Programming Interface (API). API содержит несколько сот функций, которые реализуют все необходимые системные действия, такие как выделение памяти, создание окон, вывод на экран и т.д. Функции API содержаться в библиотеках динамической загрузки (Dynamic Link Libraries или 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.

Среда разработки программ Visual C++ предназначена для создания приложений –программ, со 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. Такие приложения представляют собой совокупность объектов, которыми являются само приложение и все его компоненты: документы, облики документов, диалоги и т.д.

Прежде чем продолжать изложение, кратко остановимся на определении основных принципов объектно-ориентированного стиля программирования (ООП).

Основные понятия и принципы ООП: объект, класс и наследование. 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 data member (данное-член класса), а метод – member function (функция-член класса). 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 функция-член класса B». Теперь рассмотрим такой фрагмент кода:

```
// Полиморфная функция
void ShowMessage( A& x) {... x.VirtualMethod(); ...}
A ObjectA; //Будет выдано сообщение «Я метод класса A»;
ShowMessage(ObjectA);
B ObjectB; //Будет выдано сообщение «Я функция-член 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 поражены – предками.

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 (более 200 членов), которые позволяют создавать Windows-приложения на основе объектно-ориентированного подхода. (Имена классов библиотеки MFC начинаются с префикса «C», а имена переменных класса с «m_»; рекомендуется и нам далее придерживаться этого правила) Достоинством MFC является то, что использование ее классов позволяет избежать большей части рутинной работы и примерно в три раза сокращает объем программ. Инкапсулируя (скрывая в себе) 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 писать программы под Windows. Как уже отмечалось, у всех Windows-приложений фиксированная структура, определяемая функцией WinMain. Структура приложений, постро-

енного из объектов классов библиотеки MFC является так же жестко заданной.

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 производственные классы для определенности CApp и CMainWin соответственно. Это 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 очередь сообщений.
4. Переопределить функцию InitInstance() класса CWinApp.
5. Создать экземпляр класса, определяющего приложение – объект theApp.

Ниже приведена минимальная программа, написанная на основе 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()
```

```
{
     m_pMainWnd=new CMainWin; // Создание объекта – главного окна
     m_pMainWnd->ShowWindow(m_nCmdShow);
     m_pMainWnd->UpdateWindow();
     return TRUE;
}

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

//////////////////////////////////////////////////////
// Начало выполнения приложения
CApp theApp; // Глобальный объект – приложение
//////////////////////////////////////////////////////
```

Результат выполнения программы показан на рис. 1.2. 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 очередь сообщений для класса CMainWin. Команды

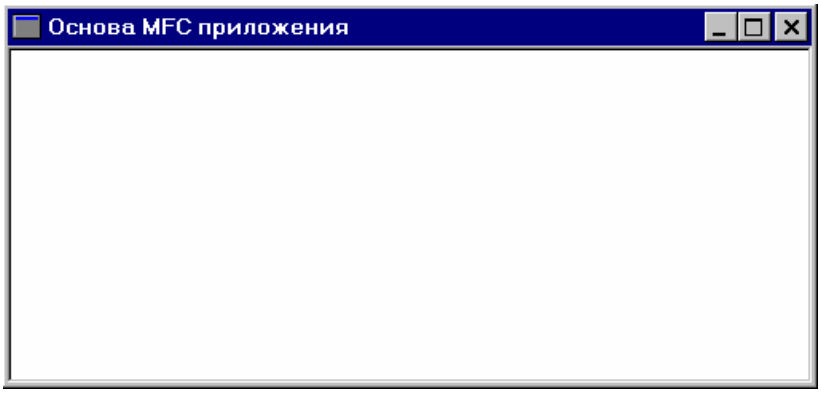


Рис. 1.2. Окно, создаваемое минимальной MFC-программой

BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd) и END_MESSAGE_MAP() представляют собой очередь сообщений главного окна; между ними должны помещаться вызовы функций обработки поступающих сообщений. В приведенном примере никакие сообщения не обрабатываются.

Рассмотрим подробнее как осуществляется обработка сообщений. Сообщение – это некоторое уникальное значение целого типа. 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 очередь сообщений программы.

2. Включить в описание класса прототип функции-обработчика сообщения.

3. Включить в программу реализацию функции-обработчика.

Названия макрокоманд обычно состоят из префикса ON_ и названия сообщения, например, ON_WM_LBUTTONDOWN(). Макрокоманда сообщения помещается в очередь сообщения окна. У одного приложения может быть несколько очередей сообщений. К какому из окон относиться очередь, определяется в параметрах 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 на практике обработку сообщений – включим в предыдущую программу обработку сообщения WM_LBUTTONDOWN. После того, как пользователь нажмет левую клавишу 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 и координаты точки. Для этого выполним следующие действия:

1. Модифицируем очередь сообщений:

```
// Очередь сообщений приложения
BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

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

```
class CMainWin: public CFrameWnd
{
public:
     CMainWin();
     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 клавиша мыши в точке x=%d, y=%d",
point.x, point.y);
// MessageBox - метод кдасса CWnd
// (базового для CFrameWnd и,соответственно, CMainWin)
// инкапсулирует одноименную функцию вывода сообщений API
     MessageBox(Str,                "Пришло                сообщение",
MB_OK|MB_ICONINFORMATION);
     CFrameWnd::OnLButtonDown(nFlags, point); // стандартный об-
работчик
}
```

Работа программы показана на рис. 1.3.

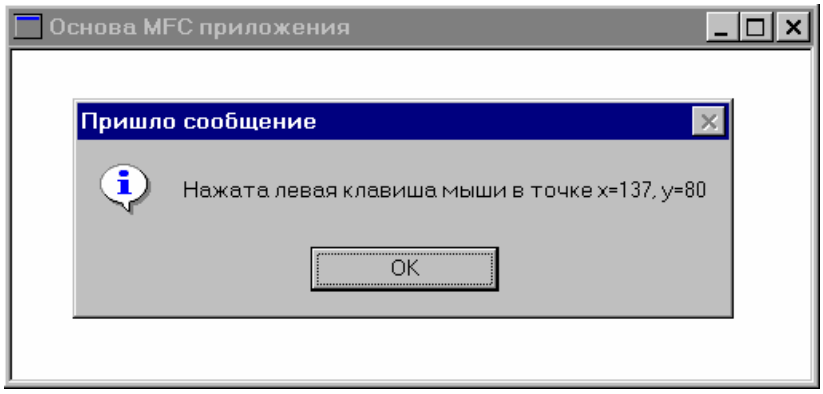


Рис. 1.3. Работа приложения с обработкой сообщения WM_LBUTTONDOWN

Visual C++ содержит инструментальное средство AppWizard, позволяющее значительно упростить процесс создания Windows-приложений на основе MFC. В процессе диалога пользователь определяет тип и характеристики проекта, который он хочет построить. 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 себя описания классов, описания ресурсов и т.д.

Рассмотрим работу AppWizard на примере создания простого приложения, которое однако будет содержать полноценный Windows-интерфейс пользователя и позволит нам, наконец, приступить к изучению предмета – компьютерной графики. Данное приложение будет создавать Windows-окно со стандартными элементами управления. Особенностью нашего приложения будет то, что при нажатии левой клавиши мыши в рабочей области окна на экране будет рисоваться линяя, соединяющая данную точку с точкой, в которой было предыдущее нажатие.

Итак, создание приложения начинается с выбора команды New меню File. После этого на экране появляется диалоговое окошко (рис. 1.4) выбора типа проекта.

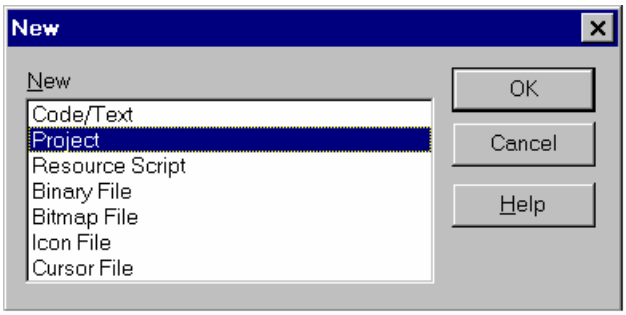


Рис. 1.4. Диалог выбора типа проекта

Выберем создание проекта и нажмем OK. В следующем диалоге (рис. 1.5) необходимо задать имя проекта, выбрать его размещение на диске, а

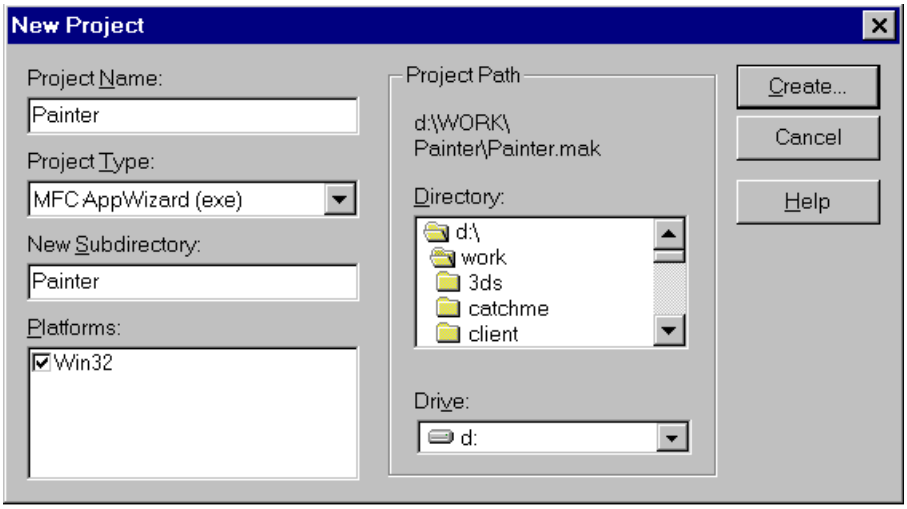


Рис. 1.5. Создание проекта

также конкретизировать тип проекта. Назовем проект незатейливо – «Painter» (что, кажется, в переводе означает художник). В каталоге Work на диске D:\

будет создан подкаталог Painter, где и будет размещаться одноименный проект. Тип проекта, как и договаривались, выберем MFC AppWizard.

Далее в работу включается AppWizard. На первом шаге он предлагает выбрать тип приложения (рис. 1.6). Мы выбираем тип Single Document (остальные типы мы обсудим позже) и нажимаем кнопку Next. На следующем

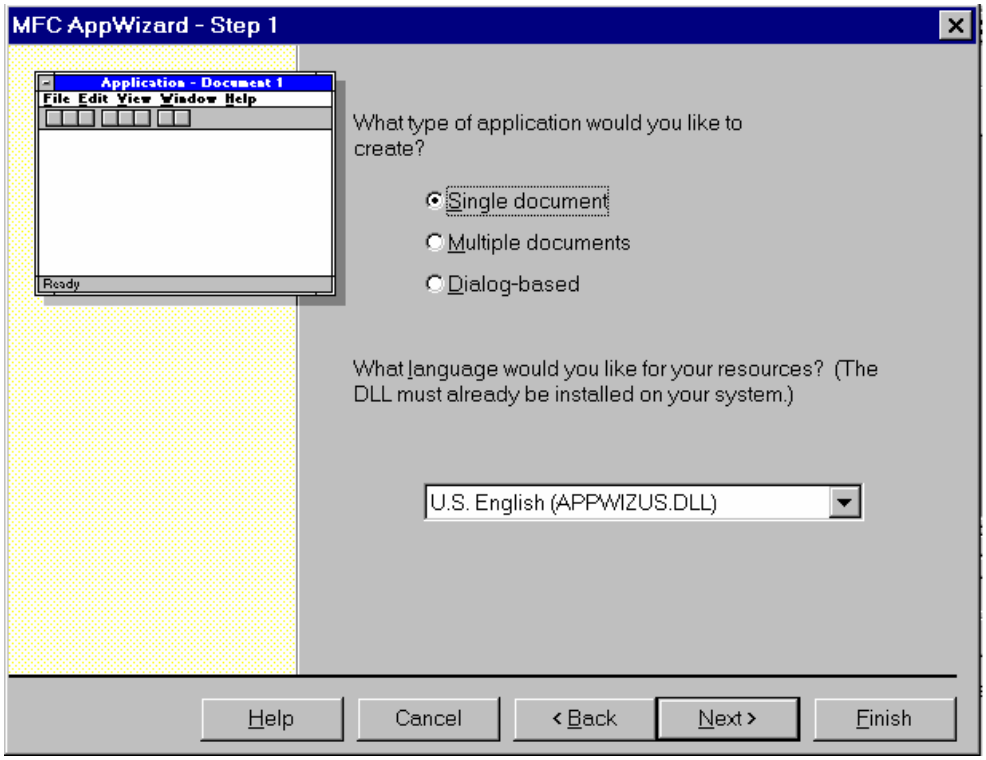


Рис. 1.6. Выбор типа приложения

шаге нам будет предложена поддержка баз данных (по умолчанию не поддерживается) – ничего не меняем и идем дальше (кнопка Next). На третьем шаге нам предложат поддержку OLE – пока обойдемся без нее – кнопка Next. На следующем шаге спрашивают «Какие возможности мы хотели бы иметь у своей программы?» – ничего не трогаем, жмем кнопку Next. На пятом шаге нам предложат комментарии в исходных текстах программы и кое-что еще – тут мы тоже ничего не меняем, жмем Next и благополучно добираемся до заключительного шага, где нам сообщат какие классы MFC будут использованы для создания нашего приложения. Нажимаем кнопку Finish. После этого в последний раз предупреждают, что сейчас начнется создание приложения, выводиться информация о его типе, классах, файлах и возможностях; – жмем OK. После этого будет создан проект нашего приложения и появится его окошко (рис. 1.7). Проект уже можно откомпилировать (команда Project-Build) и запустить на выполнение (Project-Execute).

Придадим теперь приложению новые графические свойства – для этого включим в описание класса CPainterDoc (файл Paintdoc.h) следующие строки,

выделенные жирным шрифтом. Файл Paintdoc.h можно открыть, дважды щелкнув левой клавишей мыши на его имени в окне проекта.

```
// Paintdoc.h : interface of the CPainterDoc class
//
/////////////////////////////////////////////////////////////////////////
/////////
#define MAXPOINTS 100
class CPainterDoc : public CDocument
{
protected: // create from serialization only
     CPainterDoc();
     DECLARE_DYNCREATE(CPainterDoc)
// Attributes
public:
     WORD m_nIndex; //количество точек
     CPoint m_Points[MAXPOINTS];   //координаты точек
```

Остальной код файла оставим без изменения.
В файле Paintdoc.cpp дополним конструктор класса CPainterDoc инициализацией переменной m_nIndex:

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

CPainterDoc::CPainterDoc()
{
     // TODO: add one-time construction code here
     m_nIndex=0;
}
```

Далее используем средство, предоставляемое Visual C++ для автоматизации создания обработчиков сообщений. Выполним команду Project-ClassWizard. На экране должно появиться диалоговое окно (рис. 1.8) с открытой вкладкой Message Maps. Выберем из списка Class Name имя класса CPainterView, а из списка сообщений (Messages) сообщение WM_LBUTTONDOWN и нажмем клавишу AddFunction. После этого ClassWizard вставит в очередь сообщений макрокоманду ON_WM_LBUTTONDOWN(), а в описание класса CPainterView имя функции-обработчика данного сообщения – OnLButtonDown. Нам остается только отредактировать реализацию данной функции. Для этого в списке функций-членов класса CPainterView выберем функцию OnLButtonDown и нажмем клавишу EditCode – ClassWizard откроет нам тело данной функции в файле Paintvw.cpp. Добавим в него

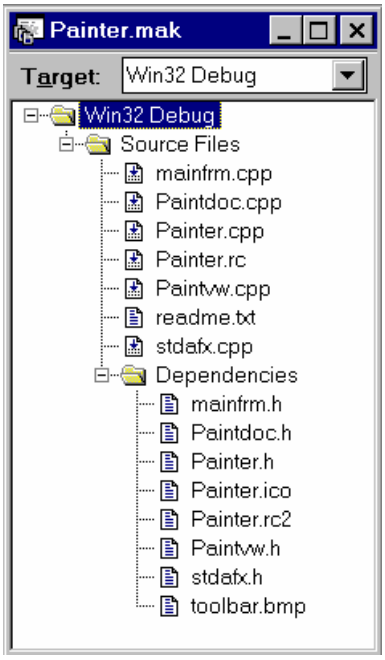


Рис. 1.7. Окно проекта

строки, выполняющие сохранение координат указателя мыши, в которых была нажата левая клавиша. Ниже приведен код функции OnLButtonDown() (жирным выделены добавленные строки).

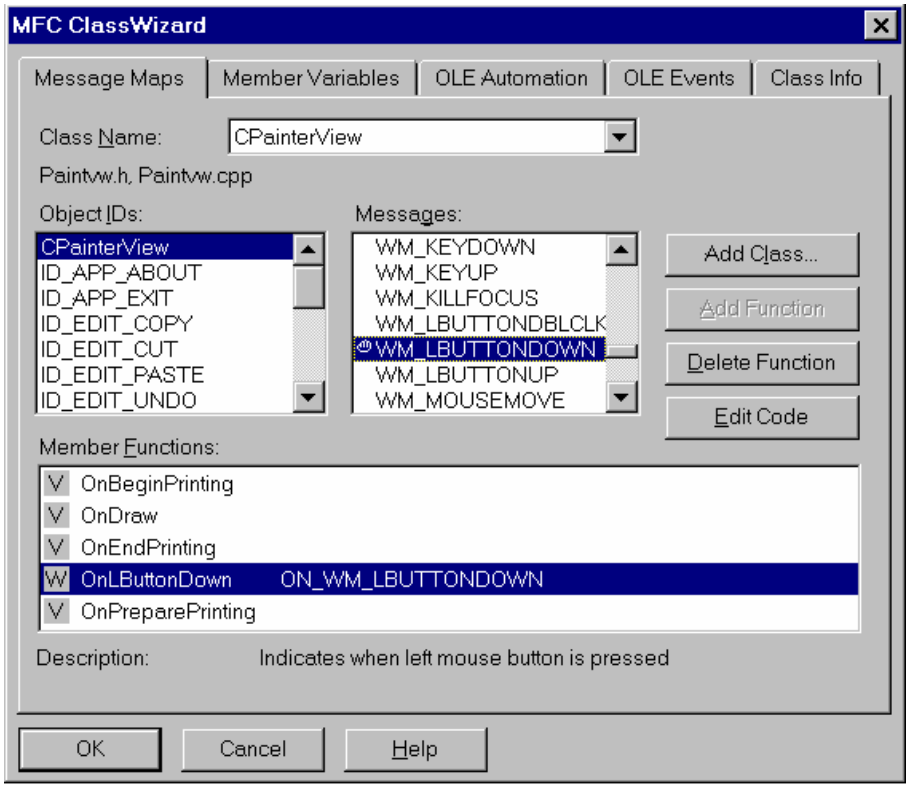


Рис. 1.8. Диалоговое окно ClassWizard

```
// файл Paintvw.cpp
/////////////////////////////////////////////////////////////////////////
////////
// CPainterView message handlers
void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
//получили указатель на объект-документ
     CPainterDoc *pDoc=GetDocument();
     if(pDoc->m_nIndex==MAXPOINTS) //исчерпали ресурс
          AfxMessageBox("Слишком много точек");
     else
     {
          //запоминаем точку
          pDoc->m_Points[pDoc->m_nIndex++]=point;
          //указываем, что содержимое окна надо перерисовать
          Invalidate();
          // указываем, что документ изменен
          pDoc->SetModifiedFlag();
     }
     CView::OnLButtonDown(nFlags, point);
}
```

Добавим в метод класса CPainterView строки выполняющие вывод на экран линий, соединяющих заданные точки. Для этого из списка Messages выберем OnDraw и нажмем кнопку EditCode. – ClassWizard откроет нам тело данной функции OnDraw() в файле Paintvw.cpp. Ниже приведен код этой функции (жирным выделены добавленные строки).

```
// файл Paintvw.cpp
/////////////////////////////////////////////////////////////////////////
/////////
// CPainterView drawing

void CPainterView::OnDraw(CDC* pDC)
{
     CPainterDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     // TODO: add draw code for native data here
     // ставим первую точку
     if(pDoc->m_nIndex>0) pDC->MoveTo(pDoc->m_Points[0]);
     //соединяем линиями остальные точки
     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.

Для того, чтобы наши бессмертные творения стали воистину таковыми, модифицируем функцию Serialize класса CPainterDoc в файле paintdoc.cpp. Данная функция является членом класса CDocument и унаследована классом CPainterDoc. Она отвечает за сохранение и загрузку данных. Об особенностях ее работы мы поговорим позднее (если потребуется), а сейчас просто добавим в нее выделенные жирным строки. (Обратите внимание, что AppWizard добавил в тело функции комментарии типа «// TODO:», которые подсказывают нам в каком месте и какой код необходимо добавить). Заметьте, что для того, чтобы функция была откомпилирована без ошибок, переменная m_nIndex должна быть объявлена типа WORD.

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

void CPainterDoc::Serialize(CArchive& ar)
{
     if (ar.IsStoring())
     {
          // TODO: add storing code here
          ar << m_nIndex;
          for(int i=0; i<m_nIndex; i++) ar << m_Points[i];
     }
     else
```

```
    {
        // TODO: add loading code here
        ar >> m_nIndex;
        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. Для того, чтобы наша программа стала полностью полноценной необходимо определить функцию создания нового рисунка. Для этого добавим в метод OnNewDocument класса CPainterDoc строки, выделенные жирным шрифтом (файл paintdoc.cpp). Так как в одно-документных 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). Результат работы программы показан на рис. 1.9.

```
BOOL CPainterDoc:: OnNewDocument()
{
    //сначала вызывается метод базового класса
    if (!CDocument::OnNewDocument()) return FALSE;
    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
    m_nIndex=0; // сбросили счетчик
    UpdateAllViews(NULL); // перерисовали
    return TRUE;
}
```

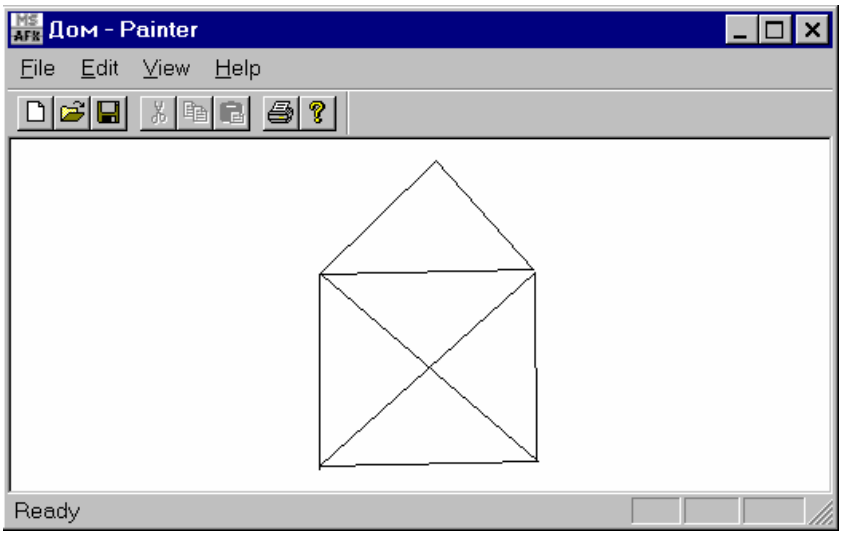


Рис. 1.9. Программа Painter в действии

Смотри также рекомендуемую литературу [7, 8, 9, 10].

2. АРХИТЕКТУРА ПРИЛОЖЕНИЙ DOCUMENT-VIEW. ГРАФИЧЕСКИЕ МЕТОДЫ КЛАССА CDC. ПРЕДСТАВЛЕНИЕ ГЕОМЕТРИЧЕСКИХ ОБЪЕКТОВ НА C++. ПРОЕКТ PAINTER 2

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

Достоинство архитектуры Document-View, в том, что она позволяет создавать приложения «сфокусированные» на данных. В таких приложениях данные могут быть представлены одновременно в различной форме. 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. Они создаются как экземпляры классов, производных от класса CDocument библиотеки MFC. Класс CDocument имеет хорошо развитые методы загрузки, сохранения и управления данными. Каждый документ сопровождается как минимум одним объектом-обликом (их может быть несколько, как в приведенном выше примере с графиком). Облик является объектом, предназначенным для отображения данных документа на экране и обеспечения взаимодействия с пользователем. Облики создаются как объекты классов, производных от класса CView из MFC. Для изменения данных документа облики используют методы документа. Логически облики привязаны к документу. Классы CDocument и CView имеют средства для общения между собой. Объект-документ хранит список всех объектов обликов, представляющих его 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, представлением данных которого он занимается. Этот метод очень важен, он позволяет 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! Вместо этого, 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.

С помощью AppWizard мы создали основу однодокументного приложения. В определение класса CPainterDoc, производного от CDocument, мы добавили переменные, в которых сохраняются данные:

```
WORD m_nIndex;   //количество точек
CPoint m_Points[MAXPOINTS];//координаты точек
```

В конструкторе класса мы произвели инициализацию переменной m_nIndex. При выборе команды «создать новый документ» автоматически вызывается метод OnNewDocument() класса CDocument (в однодокументных приложениях используется один объект-документ, поэтому нового объекта не создается и, соответственно, конструктор не вызывается); его мы переопределили так чтобы в нем переменная m_nIndex обнулялась. Для обновления содержимого облика использовали функцию UpdateAllViews(NULL) (можно ее удалить и посмотреть, что получиться). Для перерисовки же окна при нажатии левой кнопки, была использована функция Invalidate() класса облика. В функции-обработчике нажатия клавиши мыши и в функции OnDraw() для доступа к данным документа использована функция GetDocument().

В качестве параметра в функцию OnDraw() передается указатель на объект класса CDC библиотеки MFC. Данный класс используется для организации вывода информации на экран и другие устройства и имеет множество различных методов. Рассмотрим некоторые из методов этого класса.

Метод **CDC::SelectObject** имеет несколько переопределенных форм:
//Устанавливает активное перо.
CPen* SelectObject(CPen* pPen);
//Устанавливает активную кисть.
CBrush* SelectObject(CBrush* pBrush);
Возвращает указатель на предыдущий объект.

Метод **CDC::Ellipse**

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

Рисует текущей кистью и пером эллипс в пределах заданного параметрами (x1, y1, x2, y2) прямоугольника.

Метод **CDC::Rectangle**

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

Рисует текущей кистью и пером прямоугольник, размеры которого заданы параметрами (x1, y1, x2, y2).

Этих методов пока достаточно для реализации новой версии программы Painter.

Добавим в новую версию программы Painter возможность рисовать не только прямые, но и примитивные фигуры типа круг и квадрат. Заодно вспомним, как реализуются полиморфные функции, а также посмотрим, как добавлять в меню программы новые команды и в панель инструментов новые кнопки.

Начнем с того, что создадим небольшую иерархию классов графических объектов – фигур. В основе нашей иерархии поместим класс CBasePoint и породим от него два класса: CCircle и Csquare (рис. 2.1). В классе CBasePoint определим данные m_x и m_y – позиция базовой точки фигуры, m_dScale – масштаб; и виртуальные методы ответственные за сохранение объекта, отображение его в контексте устройства. В класс CCircle добавим переменную, хранящую радиус круга, а в класс CSquare – переменную-сторону квадрата. Описание данных классов поместим в файл Shapes.h, текст которого приведен ниже.

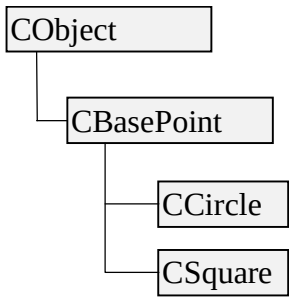


Рис. 2. 1. Иерархия

```
// файл Shapes.h
//////////////////////////////////////////////////
//класс базовая точка

class CBasePoint: public CObject
{
     DECLARE_SERIAL(CBasePoint)
 protected:     // Виртуальный метод сериализации
     virtual void Serialize(CArchive& ar);
 public:
     //конструкторы
     CBasePoint(int x, int y); //конструктор с параметрами
     CBasePoint();//конструктор без параметров
     ~CBasePoint(){}; //деструктор
// Данные
     double m_dScale; //масштаб фигуры
     LONG m_x; // координата X
```

```
      LONG m_y; // координата Y
// Методы
      // Отображение фигуры на экране - виртуальный метод
      virtual void Show(CDC *pDC);
      // Виртуальный метод, сообщающий область захвата
      virtual void GetRegion(CRgn &Rgn);
};

////////////////////////////////////////////////
//класс круг
class CCircle: public CBasePoint
{
      DECLARE_SERIAL(CCircle)
 protected:     // Виртуальный метод сериализации
      virtual void Serialize(CArchive& ar);
 public:
      //конструкторы
      CCircle(int x, int y, WORD r);
      CCircle();
      ~CCircle(){};
//Данные
      WORD m_wRadius; //радиус круга
//Методы
      // Отображение фигуры на экране
      virtual void Show(CDC *pDC);
      // Виртуальный метод сообщающий область захвата
      virtual void GetRegion(CRgn &Rgn);
};

////////////////////////////////////////////////
//класс квадрат
class CSquare: public CBasePoint
{
      DECLARE_SERIAL(CSquare)
 protected:     // Виртуальный метод сериализации
      virtual void Serialize(CArchive& ar);
 public:
      //конструкторы
      CSquare(int x, int y, WORD s);
      CSquare();
      ~CSquare(){};
//Данные
      WORD m_wSide; //длина стороны
//Методы
      // Отображение фигуры на экране
      virtual void Show(CDC *pDC);
      // Виртуальный метод, сообщающий область захвата
      virtual void GetRegion(CRgn &Rgn);
};
```

Реализацию функций данного класса поместим в файл Shapes.cpp.

```
// файл Shapes.cpp
////////////////////////////////////////
//реализация классов
#include "stdafx.h"
```

```
#include "shapes.h"

CBasePoint::CBasePoint(int x, int y)
{
      m_dScale=1.0;
      m_x=x;
      m_y=y;
};

CBasePoint::CBasePoint()
{
      m_dScale=1.0;
      m_x=100;
      m_y=100;
};

void CBasePoint::Show(CDC* pDC)
{
      int r=1;
      CBrush *pOldBrush=pDC->SelectObject(&CBrush(RGB(100,100,100)));
      pDC->Ellipse(m_x-r, m_y-r, m_x+r, m_y+r);
      pDC->SelectObject(pOldBrush);
}

void CBasePoint::GetRegion(CRgn &Rgn)
{
      int r=1;
      Rgn.CreateEllipticRgn(m_x-r, m_y-r, m_x+r, m_y+r);
}

IMPLEMENT_SERIAL(CBasePoint, CObject , 1)
void CBasePoint::Serialize(CArchive &ar)
{
      if(ar.IsStoring())
      {
            ar<<m_x;
            ar<<m_y;
            ar<<m_dScale;
      }
      else
      {
            ar>>m_x;
            ar>>m_y;
            ar>>m_dScale;
      }
};


CCircle::CCircle(int x, int y, WORD r): CBasePoint(x, y)
{
      m_wRadius=r;
}
CCircle::CCircle(): CBasePoint()
{
      m_wRadius=25;
```

```
}

void CCircle::Show(CDC* pDC)
{
      int r=m_dScale*m_wRadius;
      CBrush              *pOldBrush=pDC->SelectObject(&CBrush(HS_CROSS,
RGB(200,0,0)));
      pDC->Ellipse(m_x-r, m_y-r, m_x+r, m_y+r);
      pDC->SelectObject(pOldBrush);
}

void CCircle::GetRegion(CRgn &Rgn)
{
      int r=m_dScale*m_wRadius;
      Rgn.CreateEllipticRgn(m_x-r, m_y-r, m_x+r, m_y+r);
}

IMPLEMENT_SERIAL(CCircle, CBasePoint, 1)
void CCircle::Serialize(CArchive &ar)
{
      if(ar.IsStoring())
      {
            ar<<m_wRadius;
      }
      else
      {
            ar>>m_wRadius;
      }
      CBasePoint::Serialize(ar);
};

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

CSquare::CSquare(): CBasePoint()
{
      m_wSide=40;
}

void CSquare::Show(CDC* pDC)
{
      int s=m_dScale*m_wSide/2;
      CBrush          *pOldBrush=pDC->SelectObject(&CBrush(HS_DIAGCROSS,
RGB(0,200,0)));
      pDC->Rectangle(m_x-s, m_y-s, m_x+s, m_y+s);
      pDC->SelectObject(pOldBrush);
}

void CSquare::GetRegion(CRgn &Rgn)
{
      int s=m_dScale*m_wSide/2;
      Rgn.CreateRectRgn(m_x-s, m_y-s, m_x+s, m_y+s);
}
```

```
IMPLEMENT_SERIAL(CSquare, CBasePoint, 1)
void CSquare::Serialize(CArchive &ar)
{
      if(ar.IsStoring())
      {
            ar<<m_wSide;
      }
      else
      {
            ar>>m_wSide;
      }
      CBasePoint::Serialize(ar);
};
```

Макрокоманды DECLARE_SERIAL (*имя класса*) и IMPLEMENT_SERIAL (*имя класса*, *имя базового класса*, *версия формата файла*) в совокупности с функцией Serialize(CArchive &ar) обеспечивают возможность удобного сохранения и считывания данных документа. Метод Serialize класса CBasePoint является виртуальным и переопределяется во всех классах от него производных.

Включим файл Shapes.cpp в проект Painter. Для этого выполним команду Project-Files, укажем имя файла в поле File name, нажмем кнопку Add, а затем кнопку Close.

Хранение динамически создаваемых в программе объектов-фигур будем осуществлять в списке указателей. Объект-список будет членом класса CPainterDoc. 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.

Для поддержки работы с классами шаблонов надо подключить файл afxtempl.h. Для этого в файл stdafx.h добавим соответствующую строку.

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxtempl.h>      //Работа с шаблонами
```

Добавим в описание класса CPainterDoc строки, выделенные жирным шрифтом.

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

```
//////////////////////////////////////////////////////////////////
#define MAXPOINTS 100
class CBasePoint;

class CPainterDoc : public CDocument
{
protected: // create from serialization only
      CPainterDoc();
      DECLARE_DYNCREATE(CPainterDoc)
// Attributes
public:
      WORD m_nIndex;    //количество точек
      CPoint m_Points[MAXPOINTS]; //координаты точек
      // список указателей на объекты-фигуры
      CTypedPtrList<CObList, CBasePoint*> m_ShapesList;
      CBasePoint* m_pCurShape; //указатель на текущую фигуру
// Operations
public:
      void AddCircle(CPoint &point);   //добавить круг
      void AddSquare(CPoint &point);   //добавить квадрат
      BOOL DeleteLastShape();          //удалить последнюю фигуру

// Overrides

...// далее без изменений
```

В файл Paintdoc.cpp добавим директиву подключения файла Shapes.h и реализацию новых функций.

```
#include "Shapes.h"

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

CPainterDoc::CPainterDoc()
{
      // TODO: add one-time construction code here
      m_nIndex=0;
      m_pCurShape=NULL;
}

CPainterDoc::~CPainterDoc()
{
      while(m_ShapesList.GetCount()>0) //пока в списке есть фигуры
            delete m_ShapesList.RemoveHead();//удаляем первую из них
}

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

      // TODO: add reinitialization code here
      // (SDI documents will reuse this document)
      m_nIndex=0;
```

```
      while(m_ShapesList.GetCount()>0)
            delete m_ShapesList.RemoveHead();
      m_pCurShape=NULL;
      UpdateAllViews(NULL);

      return TRUE;
}

/////////////////////////////////////////////////////////////////////////
// CPainterDoc serialization

void CPainterDoc::Serialize(CArchive& ar)
{
      if (ar.IsStoring())
      {
            // TODO: add storing code here
            ar << m_nIndex;
            for(int i=0; i<m_nIndex; i++) ar << m_Points[i];
      }
      else
      {
            // TODO: add loading code here
            ar >> m_nIndex;
            for(int i=0; i<m_nIndex; i++) ar >> m_Points[i];
      }
      //список вызовет метод Serialize для каждого элемента
      m_ShapesList.Serialize(ar);
}

/////////////////////////////////////////////////////////////////////////
// CPainterDoc commands
void CPainterDoc::AddCircle(CPoint &point)
{
      m_ShapesList.AddTail(new CCircle(point.x, point.y, 50));
};

void CPainterDoc::AddSquare(CPoint &point)
{
      m_ShapesList.AddTail(new CSquare(point.x, point.y, 40));
};

BOOL CPainterDoc::DeleteLastShape()
{
      if(m_ShapesList.GetCount()>0)
      {
            delete m_ShapesList.RemoveTail();
            return TRUE;
      }
      else return FALSE;
};
```

В класс CPainterView введем переменную m_CurOper характеризующую текущую операцию и переменную m_LastOper – предыдущая операция.

```
// Paintvw.h : interface of the CPainterView class
```

```
//
/////////////////////////////////////////////////////////////////////////

class CPainterView : public CView
{
protected: // create from serialization only
      CPainterView();
      DECLARE_DYNCREATE(CPainterView)

// Attributes
public:
      CPainterDoc* GetDocument();
      //текущая операция
      enum      CurOper{OP_NOOPER,      OP_DRAWLINE,      OP_DRAWCIRCLE,
OP_DRAWSQUARE} m_CurOper, m_LastOper;
// Operations
public:

...// без изменений

/////////////////////////////////////////////////////////////////////////
```

Далее изменим реакцию на нажатие клавиш мыши таким образом, что если текущая операция – рисование круга (OP_DRAWCIRCLE) или рисование квадрата (OP_DRAWSQUARE) в точке нажатия клавиши рисуется круг или квадрат, соответственно. Если же текущая операция OP_NOOPER – будем рисовать прямые. При нажатии правой клавиши будем отменять предыдущие действия.

```
//файл paintvw.cpp
/////////////////////////////////////////////////////////////////////////
// CPainterView message handlers

void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
      CPainterDoc *pDoc=GetDocument();
      switch(m_CurOper)
      {
      case OP_NOOPER: // рисуем линию
            if(pDoc->m_nIndex==MAXPOINTS)
                  AfxMessageBox("Слишком много точек");
            else
            {
                  pDoc->m_Points[pDoc->m_nIndex++]=point;
                  Invalidate();
                  pDoc->SetModifiedFlag();
            }
      break;
      case OP_DRAWCIRCLE: // рисуем круг
            pDoc->AddCircle(point);
            Invalidate();
            pDoc->SetModifiedFlag();
      break;
```

```
        case OP_DRAWSQUARE:   //рисуем квадрат
              pDoc->AddSquare(point);
              Invalidate();
              pDoc->SetModifiedFlag();
        break;
        }
        m_LastOper=m_CurOper;
        m_CurOper=OP_NOOPER;
        CView::OnLButtonDown(nFlags, point);
}

void CPainterView::OnRButtonDown(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call default
        CPainterDoc *pDoc=GetDocument();
        switch(m_LastOper)
        {
        case OP_NOOPER:
              if(pDoc->m_nIndex>0) pDoc->m_nIndex--;
              Invalidate();
              pDoc->SetModifiedFlag();
        break;
        case OP_DRAWCIRCLE:
        case OP_DRAWSQUARE:
              if(pDoc->DeleteLastShape())
              {
                    Invalidate();
                    pDoc->SetModifiedFlag();
              }
        break;
        }

        Invalidate();
        pDoc->SetModifiedFlag();
        CView::OnRButtonDown(nFlags, point);
}
```

Для переключения (инициации) операций рисования фигур в меню программы включим дополнительные команды. Для этого откроем файл ресурсов (дважды щелкнем на файле с именем Painter.rc в окне проекта) и затем откроем ресурс-меню (см. рис. 2.3).

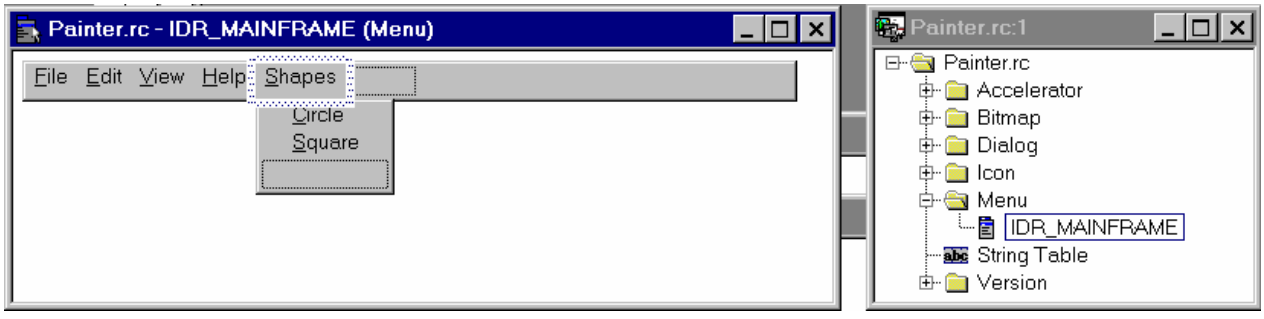


Рис. 2.3. Редактирование меню

Дважды щелкнув на панели меню, откроем диалоговое окошко, в котором зададим свойства пункта меню. Для пункта Shapes оно будет выглядеть как на рис. 2.4.

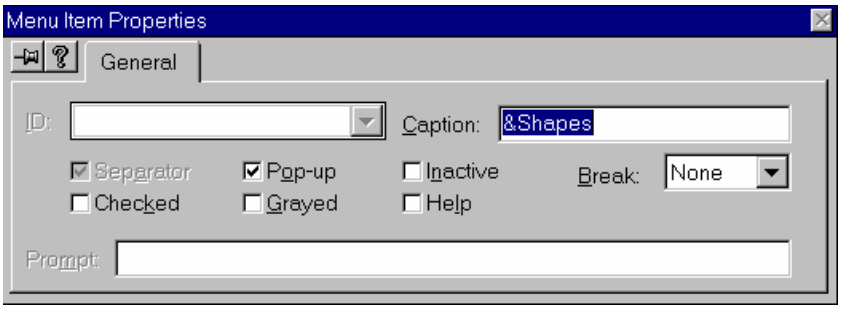


Рис. 2.4. Свойства пункта Shapes

Для пункта Circle зададим свойства – рис. 2.5. Аналогично для Square.

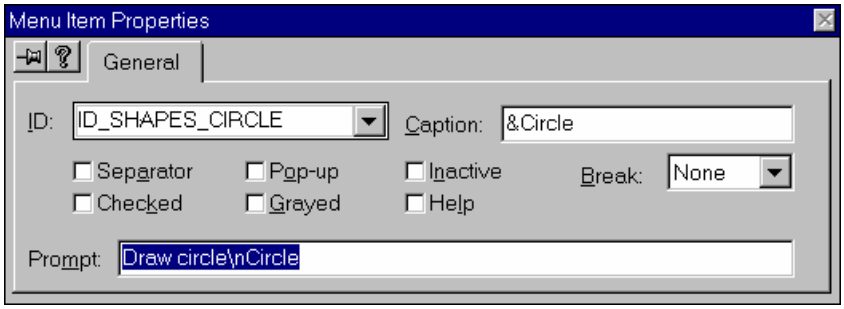


Рис. 2.5. Свойства пункта Circle

Добавим с помощью ClassWizard обработчики данных команд в класс CPainterView (файл Paintvw.cpp) см. рис. 2.6. Кроме того, добавим функции-обработчики изменения состояния меню (сообщение UPDATE_COMMAND_UI). Отредактируем добавленные функции следующим образом.

```
// файл paintvw.cpp
void CPainterView::OnShapesSquare()
{
      m_CurOper=OP_DRAWSQUARE;
}

void CPainterView::OnUpdateShapesSquare(CCmdUI* pCmdUI)
{
      //Если рисуем квадрат, кнопка нажата
      pCmdUI->SetCheck(m_CurOper==OP_DRAWSQUARE);
}

void CPainterView::OnShapesCircle()
{
      m_CurOper=OP_DRAWCIRCLE;
}
```

```
void CPainterView::OnUpdateShapesCircle(CCmdUI* pCmdUI)
{
      //Если рисуем круг, кнопка нажата
      pCmdUI->SetCheck(m_CurOper==OP_DRAWCIRCLE);
}
```

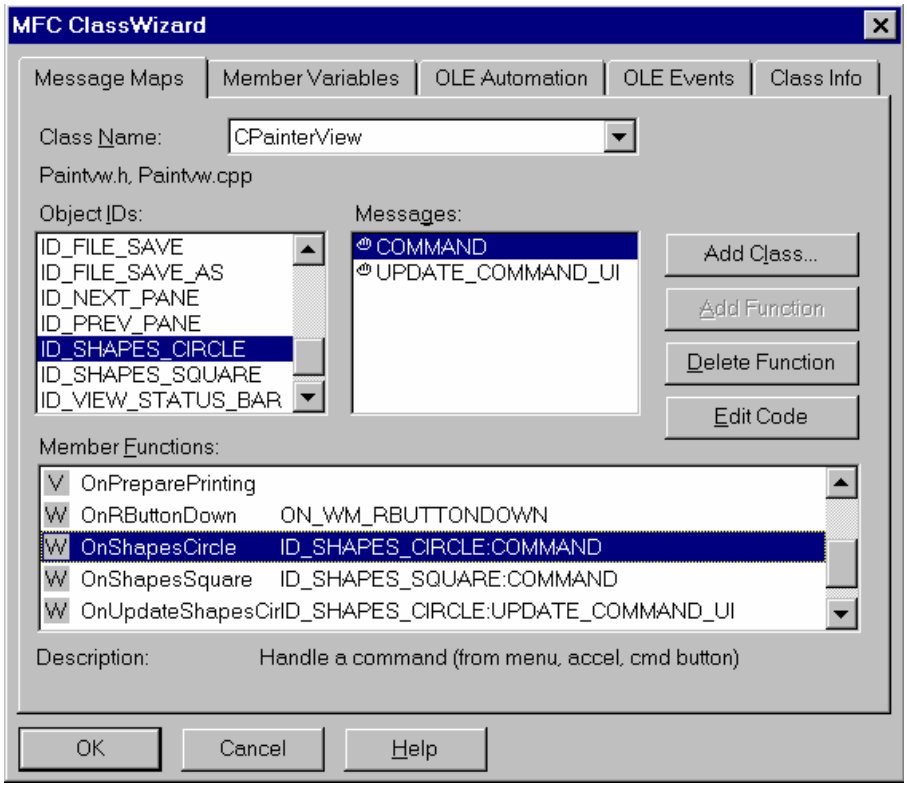


Рис. 2.6. Добавление обработчика команды ON_SHAPES_CIRCLE

Добавим в функцию OnDraw класса CPainterView, код, осуществляющий вывод на экран фигур, хранящихся в списке m_ShapesList.

```
 // файл paintvw.cpp
void CPainterView::OnDraw(CDC* pDC)
{
      CPainterDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      // TODO: add draw code for native data here
      // Все точки, хранящиеся в документе, соединяем прямыми
      if(pDoc->m_nIndex>0) pDC->MoveTo(pDoc->m_Points[0]);
      for(int i=1; i<pDoc->m_nIndex; i++)
            pDC->LineTo(pDoc->m_Points[i]);

      //Выводим все фигуры, хранящиеся в списке
      POSITION pos=NULL;
      CBasePoint* pShape=NULL;
      if(pDoc->m_ShapesList.GetCount()>0)
            pos=pDoc->m_ShapesList.GetHeadPosition();
```

```
        while(pos!=NULL)
        {
                pShape=pDoc->m_ShapesList.GetNext(pos);
                if(pShape!=NULL) pShape->Show(pDC);
        }
}
```

Остается добавить только кнопки в панель инструментов. Для этого отредактируем изображение панели инструментов. Щелкнем дважды на ресурсе (IDR_MAINFRAME) – изображении панели инструментов. Появиться растровое изображение и панель инструментов, похожая на инструменты программы Paint (из стандартных Windows'овских). Удлиним панель на две кнопки и с помощью предложенных инструментов нарисуем две кнопки (круг и квадрат) см. рис. 2.7.

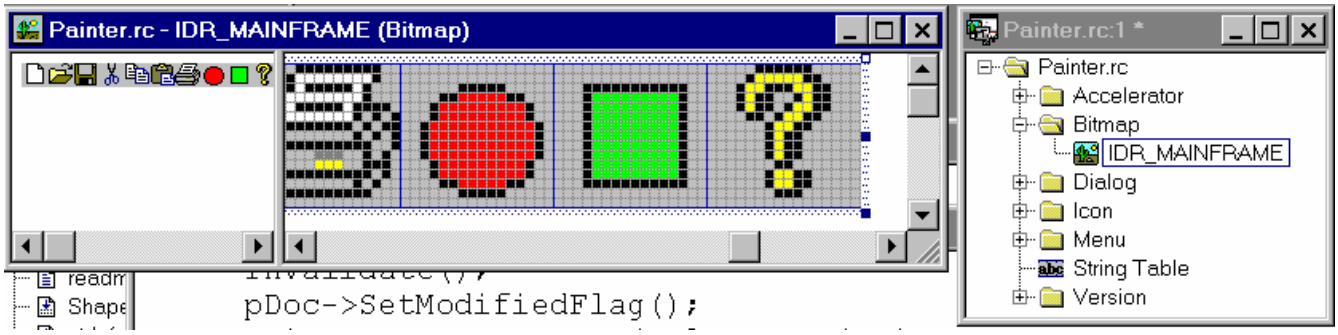


Рис. 2.7. Добавление кнопок в панель инструментов

Теперь остается только добавить описание данных кнопок в файл mainfrm.cpp.

```
// файл mainfrm.cpp
/////////////////////////////////////////////////////////////////////////////
//////
// arrays of IDs used to initialize control bars

// toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] =
{
        // same order as in the bitmap 'toolbar.bmp'
        ID_FILE_NEW,
        ID_FILE_OPEN,
        ID_FILE_SAVE,
                ID_SEPARATOR,
        ID_EDIT_CUT,
        ID_EDIT_COPY,
        ID_EDIT_PASTE,
                ID_SEPARATOR,
        ID_FILE_PRINT,
                ID_SEPARATOR,
        ID_SHAPES_CIRCLE,
        ID_SHAPES_SQUARE,
```

```
        ID_SEPARATOR,
    ID_APP_ABOUT,
};
```

Компилируем и запускаем программу. Практически у нас получился почти Corel Draw: рисует, сохраняет и загружает свои файлы. Результат работы программы показан на рис. 2.8.

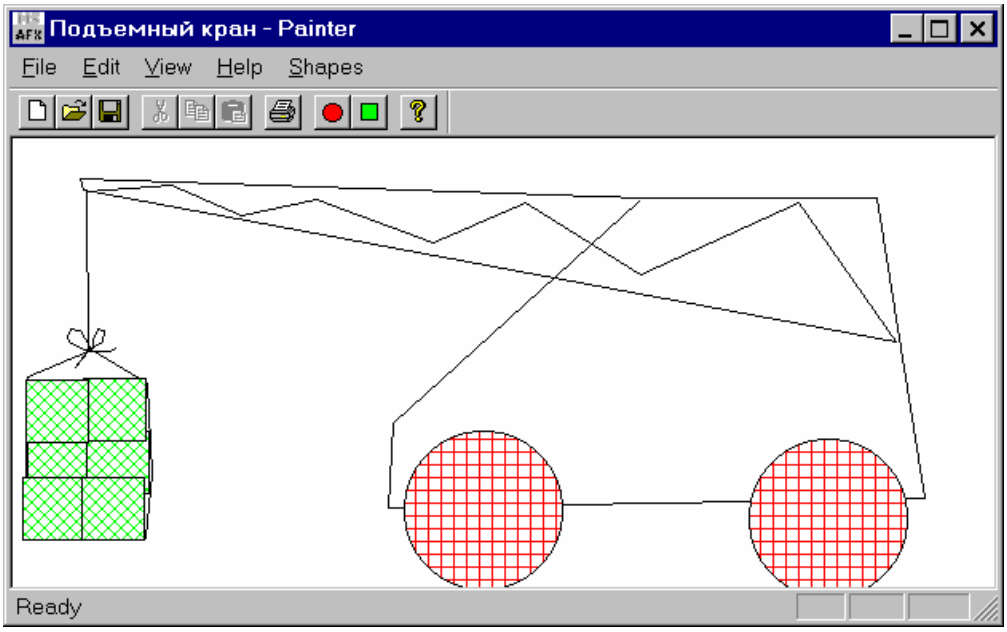


Рис. 2.8. Программа Painter 2 в действии

Смотри также рекомендуемую литературу [7, 8, 10].

3. ГЕОМЕТРИЧЕСКИЙ ИНСТРУМЕНТ ДЛЯ АЛГОРИТМОВ КОМПЬЮТЕРНОЙ ГРАФИКИ. ПРОЕКТ PAINTER 3

Вектор - направленный отрезок прямой.

Обозначения: *P, Q* – концевые точки отрезка;

a, ***b***, ***c***, – векторы;

0 – вектор с нулевой длиной;

-***a*** – вектор длиной $|\boldsymbol{a}|$, направленный обратно ***a***;

p, к – вещественные числа;

$|\boldsymbol{a}|$ – длина вектора; равна расстоянию между концевыми точками.

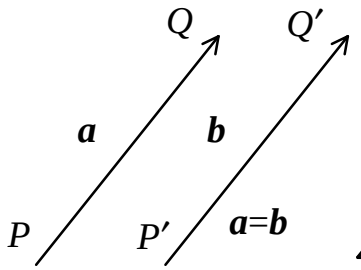


Рис. 3.1

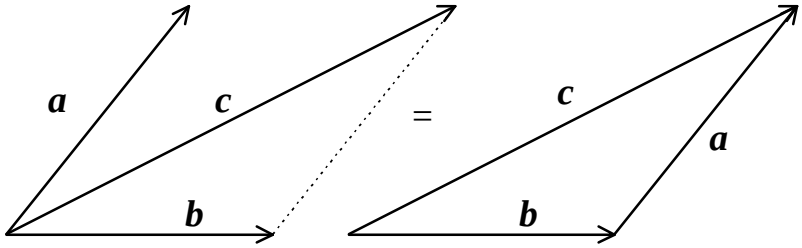


Рис. 3.2

Свойства:

1. При параллельном переносе вектор не изменяется (рис. 3.1).

2. Сумма векторов тоже вектор: $\boldsymbol{a}+\boldsymbol{b}=\boldsymbol{c}$ (рис. 3.2).

3. Произведение *p****a*** – вектор длиной равной $|p\|\boldsymbol{a}|$, если *p*=0 или ***a***=***0***, то *p****a***=***0***;

если p>0, результирующий вектор совпадает по направлению с ***a***;

если p>0, результирующий вектор имеет направление противоположное ***a***.

1. Для векторов выполняются следующие правила:

a+***b***=***b***+***a***;

(***a***+***b***)+***c***=***a***+(***b***+***c***);

a+***0***=***a***;

a+(-***a***)=***0***;

p(***a***+***b***)=*p****a***+*p****b***;

$(p+k)\boldsymbol{a}=p\boldsymbol{a}+k\boldsymbol{a}$;

$1\boldsymbol{a}=\boldsymbol{a}$;

$0\boldsymbol{a}=\boldsymbol{0}$.

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^o, направление вектора k совпадает с поступательным движением винта с правой резьбой (рис. 3.3). Начальная точка векторов обозначается буквой *O*.

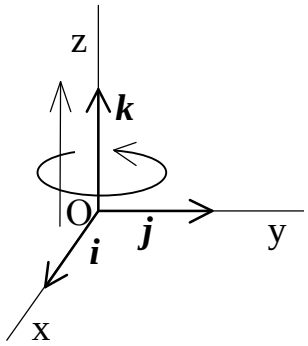


Рис. 3.3

Любой вектор $\boldsymbol{V}$ может быть записан как линейная комбинация $\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$: $\boldsymbol{V}=x\boldsymbol{i}+y\boldsymbol{j}+z\boldsymbol{k}$, где x, y, z – координаты конечной точки P вектора $\boldsymbol{V}$=OP. Вектор $\boldsymbol{V}$ может быть обозначен:

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

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

$$\boldsymbol{a}\cdot\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\cos\gamma, \qquad (3.1)$$

где γ – угол между $\boldsymbol{a}$ и $\boldsymbol{b}$. Применяя это правило к единичным векторам $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ находим

$$\boldsymbol{i}\cdot\boldsymbol{i}=\boldsymbol{j}\cdot\boldsymbol{j}=\boldsymbol{k}\cdot\boldsymbol{k}=1;\ \boldsymbol{i}\cdot\boldsymbol{j}=\boldsymbol{j}\cdot\boldsymbol{i}=\boldsymbol{j}\cdot\boldsymbol{k}=\boldsymbol{k}\cdot\boldsymbol{j}=\boldsymbol{k}\cdot\boldsymbol{i}=\boldsymbol{i}\cdot k=0 \qquad (3.2)$$

Важные свойства скалярного произведения:

$$c(k\boldsymbol{u}\cdot\boldsymbol{v})=ck(\boldsymbol{u}\cdot\boldsymbol{v});$$

$$(c\boldsymbol{u}+k\boldsymbol{v})\boldsymbol{w}=c\boldsymbol{u}\cdot\boldsymbol{w}+k\boldsymbol{v}\cdot\boldsymbol{w};$$

$$\boldsymbol{u}\cdot\boldsymbol{v}=\boldsymbol{v}\cdot\boldsymbol{u};$$

$\boldsymbol{u}\cdot\boldsymbol{u}=\boldsymbol{0}$, если $\boldsymbol{u}=\boldsymbol{0}$.

Скалярное произведение векторов $\boldsymbol{u}=[u_1,u_2,u_3]$ и $\boldsymbol{v}=[v_1,v_2,v_3]$:

$$\boldsymbol{u}\cdot\boldsymbol{v}=u_1v_1+u_2v_2+u_3v_3. \qquad (3.4)$$

Соотношение (3.4) следует из $\boldsymbol{u}\cdot\boldsymbol{v}=(u_1\boldsymbol{i}+u_2\boldsymbol{j}+u_3\boldsymbol{k})\cdot(v_1\boldsymbol{i}+v_2\boldsymbol{j}+v_3\boldsymbol{k})$ с учетом свойств скалярного произведения и соотношения (3.2).

Детерминанты. Рассмотрим систему уравнений:

$$\begin{cases} a_1 x + b_1 y = c_1 \\ a_2 x + b_2 y = c_2 \end{cases}. \qquad (3.5)$$

Чтобы решить систему (3.5), умножим первое уравнение на b_2, а второе на $-b_1$ и сложим – получим: $(a_1 b_2 - a_2 b_1)x = b_2 c_1 - b_1 c_2$. Затем первое умножим на $-a_2$, а второе на a_1 и сложим, в результате получим: $(a_1 b_2 - a_2 b_1)y = a_1 c_2 - a_2 c_1$,

если $a_1 b_2 - a_2 b_1 \neq 0$, то:

$$x = \frac{b_2 c_1 - b_1 c_2}{a_1 b_2 - a_2 b_1}; \qquad y = \frac{a_1 c_2 - a_2 c_1}{a_1 b_2 - a_2 b_1}. \qquad (3.6)$$

Выражение в делителе может быть записано:

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

С помощью детерминантов уравнение (3.5) может быть записано:

$$x = \frac{D_1}{D}, \qquad y = \frac{D_2}{D}, \qquad (D \neq 0),$$

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

D_i (i=1, 2) получается заменой i-го столбца на правую часть системы (3.5). Такой способ пригоден для решения не только систем двух уравнений и называется «правилом Крамера».

Детерминант третьего порядка:

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

Аналогично расписываются детерминанты более высоких порядков.

Свойства детерминантов.

1. Значение D если строки записать в столбцы (при транспонировании) не измениться:

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

2. Замена двух строк (столбцов) меняет знак детерминанта.

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

3. Если любую строку (столбец) умножить на число, то:

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

4. Если строка (столбец) изменяется путем добавления соответствующих элементов другой строки (столбца), умноженных на константу, то значение D не измениться:

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

5. 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 (R^2) пространстве, проходящей через точки $P_1(x_1, y_1)$, $P_2(x_2, y_2)$:

$$\begin{vmatrix} x & y & 1 \\ x_1 & y_1 & 1 \\ x_2 & x_2 & 1 \end{vmatrix} = 0. \qquad (3.7)$$

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

Плоскость в R^3, проходящая через точки $P_1(x_1, y_1, z_1)$, $P_2(x_2, y_2, z_2)$, $P_3(x_3, y_3, z_3)$:

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

Векторное произведение векторов $\boldsymbol{a}$ и $\boldsymbol{b}$ обозначается $\boldsymbol{a} \times \boldsymbol{b}$: $\boldsymbol{v} = \boldsymbol{a} \times \boldsymbol{b}$.

Свойства:

1. Если $\boldsymbol{a} = c\boldsymbol{b}$, c-скаляр, то $\boldsymbol{v} = \boldsymbol{a} \times \boldsymbol{b} = \boldsymbol{0}$, иначе длина вектора $\boldsymbol{v}$ равна:

$$|\boldsymbol{v}| = |\boldsymbol{a}||\boldsymbol{b}| \sin \gamma ,$$

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

2. Пусть k - некоторая константа, тогда справедливы соотношения:

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

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

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

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

Для правой ортогональной системы координат определяемой векторами $\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$, справедливо соотношение $\boldsymbol{i} \times \boldsymbol{i} = \boldsymbol{j} \times \boldsymbol{j} = \boldsymbol{k} \times \boldsymbol{k} = 0$; $\boldsymbol{i} \times \boldsymbol{j} = \boldsymbol{k}$; $\boldsymbol{j} \times \boldsymbol{k} = \boldsymbol{i}$; $\boldsymbol{k} \times \boldsymbol{i} = \boldsymbol{j}$; $\boldsymbol{j} \times \boldsymbol{i} = -\boldsymbol{k}$; $\boldsymbol{k} \times \boldsymbol{j} = -\boldsymbol{i}$; $\boldsymbol{i} \times \boldsymbol{k} = -\boldsymbol{j}$. 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:

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

что может быть записано в форме:

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

Однородные координаты.

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

$$ax + by + cw = 0. \qquad (3.8)$$

Уравнения типа (3.8) 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 (3.8) описывает плоскость, проходящую через начало координат и заданную прямую линию.

Если считать, что (*x, y, w*) иная форма записи (x/w, y/w), то тогда *w* не должно быть равно 0. Однако некоторые полезные свойства однородных координат проявляются именно при отсутствии такого требования.

Рассмотрим систему.

$\begin{cases} 2x + 3y - 6 = 0 \\ 4x + 6y - 24 = 0 \end{cases}$ – задает две параллельные линии и не имеет решения.

При замене координат на однородные:

$$\begin{cases} 2x + 3y - 6w = 0 \\ 4x + 6y - 24w = 0 \end{cases} \qquad (3.9)$$

система имеет, по крайней мере, одно решение (*x*=0, *y*=0, *w*=0).

Система $\begin{cases} 2x + 3y = 0 \\ w = 0 \end{cases}$ эквивалентна системе (3.9), следовательно, верно соотношение:

$\frac{x}{y} = \frac{-2}{3}$, отсюда получим, что решение системы (3.9) состоит из всех точек (3*k*, -2*k*, 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 (3*k*, -2*k*, *w*), при *w*–>0.

Использование однородных координат. В обычных двумерных координатах линейное преобразование на плоскости может быть записано:

$[x'\ y'] = [x\ y]\boldsymbol{A}$, 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] отображаются в $[a_1\ a_2]$ и $[b_1\ b_2]$. Не зависимо от $\boldsymbol{A}$, точка [0 0] отобразится в [0 0], поэтому таким способом нельзя выполнить операцию переноса. В однородных координатах точка в двумерном пространстве задается тройкой (x, y, w) и преобразование записывается в виде $[x'\ y'\ w'] = [x\ y\ w]\boldsymbol{A}$, $\boldsymbol{A} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}$.

В этом случае имеют место следующие преобразования:

$$[1\,0\,0]\boldsymbol{A} = [a_1\ a_2\ a_3],$$

$$[0\,1\,0]\boldsymbol{A} = [b_1\ b_2\ b_3],$$

$$[0\,0\,1]\boldsymbol{A} = [c_1\ c_2\ c_3].$$

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

Преобразования на плоскости.

Рассмотрим систему уравнений:

$$\begin{cases} x' = x + a \\ y' = y \end{cases}. \qquad (3.10)$$

Система (3.10) может означать:

1) Перемещение всех точек в плоскости x-y вправо на расстояние a (рис. 3.4).

2) Смещение координатных осей влево на расстояние a (рис. 3.5).

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

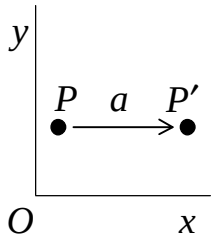


Рис. 3.4

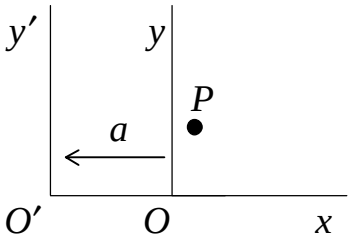


Рис. 3.5

Рассмотрим далее операцию поворота точки $P(x, y)$ вокруг начала координат O на угол φ в точку $P'(x', y')$ (рис. 3.6). Новые координаты точки рассчитываются с помощью системы уравнений:

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

Если P=(1, 0), то $x' = a$, $y' = c$, из рис. 3.7 следует, $a = \cos\varphi$, $c = \sin\varphi$.

Аналогично, если P=(0, 1): $b = -\sin\varphi$, $d = \cos\varphi$ (см. рис. 3.8).

Таким образом:

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi \\ y' = x\sin\varphi + y\cos\varphi \end{cases}. \qquad (3.11)$$

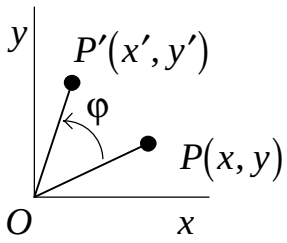


Рис. 3.6

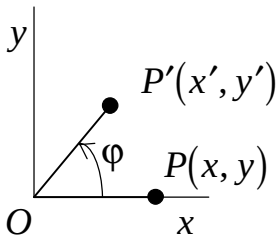


Рис. 3.7

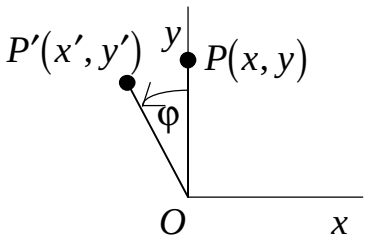


Рис. 3.8

Матричная форма записи двумерных преобразований.

Поворот вокруг точки O:

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

Поворот вокруг точки (x_0, y_0) запишется:

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

В матричной форме $[x'\ y'] = [x_0\ y_0] + [x - x_0\ y - y_0]\begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix}$.

Более удобно совместить операцию переноса и поворота в одной матрице.

Операция переноса: $\begin{cases} x' = x + a \\ y' = y + b \end{cases}$: $[x'\ y'\ 1] = [x\ y\ 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$ – запись в однородных координатах.

Поворот на угол φ вокруг точки O в однородных координатах:

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

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

$[x'\ y'\ 1] = [x\ y\ 1]\boldsymbol{R}$, где $\boldsymbol{R}$ - некоторая матрица 3х3.

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

1. Преобразование для переноса точки (x_0, y_0) в начало координат – точку O:

$[u_1\ v_1\ 1] = [x\ y\ 1]\boldsymbol{T}'$, 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:

$[u_2\ v_2\ 1] = [u_1\ v_1\ 1]\boldsymbol{R}_0$, 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):

$[x'\ y'\ 1] = [u_2\ v_2\ 1]\boldsymbol{T}$, 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:

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

следовательно:

$$\boldsymbol{R} = \boldsymbol{T}'\boldsymbol{R}_0\boldsymbol{T} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ c_1 & c_2 & 1 \end{bmatrix}, \text{ где } \begin{matrix} c_1 = x_0 - x_0\cos\varphi + y_0\sin\varphi \\ c_2 = y_0 - x_0\sin\varphi - y_0\cos\varphi \end{matrix}.$$

Реализуем на практике преобразования переноса и поворота. Для этого несколько модифицируем проект Painter.

Во-первых, дополним класс CBasePoint специальным методом Transform, выполняющим операцию поворота и переноса над заданной точкой.

```
// файл shapes.h
//////////////////////////////////////////////////
//класс базовая точка

class CBasePoint: public CObject
{
     DECLARE_SERIAL(CBasePoint)
 protected:          // Виртуальный метод сериализации
     virtual void Serialize(CArchive& ar);
 public:
     //конструкторы
     CBasePoint(int x, int y); //конструктор с параметрами
     CBasePoint();//конструктор без параметров
     ~CBasePoint(){}; //деструктор
// Данные
     double m_dScale; //масштаб фигуры
     LONG m_x;
     LONG m_y;
// Методы
     // Отображение фигуры на экране - чистый виртуальный метод
     virtual void Show(CDC *pDC);
     // Виртуальный метод сообщающий область захвата
     virtual void GetRegion(CRgn &Rgn);
     //двумерное преобразование точки point
     //поворот вокруг точки point0, на угол ang,
     //перенос по x на a, по y на b
     CPoint Transform( const CPoint &point,
            const CPoint &point0, double ang, int a, int b);

};
```

Реализация метода Transform в файле shapes.cpp. Для того, чтобы использовать математические функции подключим заголовочный файл math.h

```
// файл Shapes.cpp
//////////////////////////////////////////////////
#include "math.h"
CPoint CBasePoint::Transform(   const CPoint &point,
                const CPoint &point0, double ang, int a, int b)
{
     CPoint res;
```

```
      res.x=point0.x+(point.x-point0.x)*cos(ang)-(point.y-
point0.y)*sin(ang)+a;
      res.y=point0.y+(point.x-point0.x)*sin(ang)+(point.y-
point0.y)*cos(ang)+b;
      return res;
};
```

Фигуры классов CCircle и CSquare определяются лишь базовой точкой и радиусом, поэтому над ними можно выполнить только операцию переноса. Для иллюстрации действия преобразований поворота введем в иерархию новый класс CPoligon, так же производный от CBasePoint. Новый класс будет задавать фигуру-полигон, определяемую набором точек.

```
// файл Shapes.h
//////////////////////////////////////////////////
//класс полигон
#define MAXPOLIGONLENGTH 100
class CPoligon: public CBasePoint
{
      DECLARE_SERIAL(CPoligon)
 protected:     // Виртуальный метод сериализации
      virtual void Serialize(CArchive& ar);
 public:
      //конструкторы
      CPoligon();
      ~CPoligon();
//Данные
      POINT m_Points[MAXPOLIGONLENGTH]; // массив точек
      WORD m_nIndex;
//Методы
      // Отображение фигуры на экране
      virtual void Show(CDC *pDC);
      // Виртуальный метод сообщающий область захвата
      virtual void GetRegion(CRgn &Rgn);
      //добавить точку
      BOOL AddPoint(CPoint point);
};
```

Реализация методов класса CPoligon.

```
// файл Shapes.cpp
//////////////////////////////////////////////////
//класс полигон


CPoligon::CPoligon(): CBasePoint()
{
      m_nIndex=0;
};

CPoligon::~CPoligon()
{

};
void CPoligon::Show(CDC* pDC)
{
```

```
     CBrush         *pOldBrush=pDC->SelectObject(&CBrush(HS_DIAGCROSS,
RGB(0,0,200)));

     pDC->Polygon( m_Points, m_nIndex);

     pDC->SelectObject(pOldBrush);
}

void CPoligon::GetRegion(CRgn &Rgn)
{
     Rgn.CreatePolygonRgn( m_Points, m_nIndex, WINDING);
}
BOOL CPoligon::AddPoint(CPoint point)
{
     if(m_nIndex<MAXPOLIGONLENGTH)
     {
          m_Points[m_nIndex++]=point;
     }
     else return FALSE;
     return TRUE;
};

IMPLEMENT_SERIAL(CPoligon, CBasePoint, 1)
void CPoligon::Serialize(CArchive &ar)
{
     if(ar.IsStoring())
     {
          ar << m_nIndex;
          for(int i=0; i<m_nIndex; i++) ar << m_Points[i];
     }
     else
     {
          ar >> m_nIndex;
          for(int i=0; i<m_nIndex; i++) ar >> m_Points[i];
     }

     CBasePoint::Serialize(ar);
};
```

В меню Shapes добавим команду Poligon, а в панель инструментов, соответствующую кнопку. Данные операции рассмотрены в предыдущей лекции. Не забудьте добавить объявление кнопки в файл

Несколько модифицируем процедуру добавления новых фигур в объект-документ. Вместо методов AddCircle() и AddSquare() введем метод AddShape(CBasePoint *pShape), аргументом, которого будет являться указатель на объект-фигуру.

```
// файл Paintdoc.h
////////////////////////////////////////////////
//класс документ

class CPainterDoc : public CDocument
{
```

```
protected: // create from serialization only
      CPainterDoc();
      DECLARE_DYNCREATE(CPainterDoc)
// Attributes
public:
      WORD m_nIndex;   //количество точек
      CPoint m_Points[MAXPOINTS]; //координаты точек
      CTypedPtrList<CObList, CBasePoint*> m_ShapesList; // список ука-
зателей на фигуры
      CBasePoint* m_pCurShape; //указатель на текущую фигуру
// Operations
public:
      void AddShape(CBasePoint* pShape);
      BOOL DeleteLastShape();
...
```

Реализация метода AddShape() в файле paintdoc.cpp

```
// файл Paintdoc.cpp
/////////////////////////////////////////////////
// CPainterDoc commands

void CPainterDoc::AddShape(CBasePoint* pShape)
{
      m_ShapesList.AddTail(pShape);
};
```

Список операций объекта-облика пополним операцией OP_DRAWPOLIGON, а также введем новую переменную-указатель на активный объект-фигуру m_pCurShape:

```
// файл Paintvw.h
/////////////////////////////////////////////////
// класс - облик
class CPainterView : public CView
{
protected: // create from serialization only
      CPainterView();
      DECLARE_DYNCREATE(CPainterView)

// Attributes
public:
      CPainterDoc* GetDocument();
      //текущая операция
      enum      CurOper{OP_NOOPER,      OP_DRAWLINE,      OP_DRAWCIRCLE,
OP_DRAWSQUARE, OP_DRAWPOLIGON} m_CurOper, m_LastOper;
      CBasePoint *m_pCurShape; //указатель на последнюю фигуру
...
```

Определим обработчик команды Shapes-Poligon (для этого воспользуемся средствами ClassWizard см. предыдущую лекцию):

```
// файл Paintvw.cpp
/////////////////////////////////////////////////

void CPainterView::OnShapesPoligon()
{
```

```
      // TODO: Add your command handler code here
      m_pCurShape=new CPoligon();
      m_CurOper=OP_DRAWPOLIGON;
}

void CPainterView::OnUpdateShapesPoligon(CCmdUI* pCmdUI)
{
      pCmdUI->SetCheck(m_CurOper==OP_DRAWPOLIGON);

}
```

Модифицируем реакцию на нажатие левой клавиши мыши для того, чтобы можно было вводить полигоны, а также изменим процедуру добавления фигур круга и квадрата.

```
// файл Paintvw.cpp
////////////////////////////////////////////////////

void CPainterView::OnLButtonDown(UINT nFlags, CPoint point)
{
      CPainterDoc *pDoc=GetDocument();
      switch(m_CurOper)
      {
      case OP_NOOPER:
            if(pDoc->m_nIndex==MAXPOINTS)
                  AfxMessageBox("Слишком много точек");
            else
            {
                  pDoc->m_Points[pDoc->m_nIndex++]=point;
                  Invalidate();
                  pDoc->SetModifiedFlag();
            }
      break;
      case OP_DRAWCIRCLE:
            pDoc->AddShape(new CCircle(point.x, point.y, 20));
            Invalidate();
            pDoc->SetModifiedFlag();
            m_LastOper=m_CurOper;
            m_CurOper=OP_NOOPER;
      break;
      case OP_DRAWSQUARE:
            pDoc->AddShape(new CSquare(point.x, point.y, 40));
            Invalidate();
            pDoc->SetModifiedFlag();
            m_LastOper=m_CurOper;
            m_CurOper=OP_NOOPER;
      break;
      case OP_DRAWPOLIGON:
            if(m_pCurShape!=NULL)
                  ((CPoligon*)m_pCurShape)->AddPoint(point);
            Invalidate();
      break;
      }
      CView::OnLButtonDown(nFlags, point);
}
```

Добавим обработчик двойного щелчка левой клавиши мыши – он будет завершать ввод полигона:

```
// файл Paintvw.cpp
////////////////////////////////////////////////////
void CPainterView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
      if(m_CurOper==OP_DRAWPOLIGON)
      {
            //добавили полигон в список
            GetDocument()->AddShape(m_pCurShape);
            //сигнализируем, что документ был изменен
            GetDocument()->SetModifiedFlag();
            m_LastOper=m_CurOper;
            m_CurOper=OP_NOOPER;
      }
      CView::OnLButtonDblClk(nFlags, point);
}
```

Теперь остается модифицировать функцию OnDraw() класса-облика, так, чтобы она выводила недорисованный полигон:

```
// файл Paintvw.cpp
////////////////////////////////////////////////////
void CPainterView::OnDraw(CDC* pDC)
{
      CPainterDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      // TODO: add draw code for native data here
      if(pDoc->m_nIndex>0) pDC->MoveTo(pDoc->m_Points[0]);
      for(int i=1;i<pDoc->m_nIndex; i++)
            pDC->LineTo(pDoc->m_Points[i]);

      POSITION pos=NULL;
      CBasePoint* pShape=NULL;
      if(pDoc->m_ShapesList.GetCount()>0)
                  pos=pDoc->m_ShapesList.GetHeadPosition();
      while(pos!=NULL)
      {
            pShape=pDoc->m_ShapesList.GetNext(pos);
            if(pShape!=NULL) pShape->Show(pDC);
      }
      //недорисованный многоугольник
      if(m_CurOper==OP_DRAWPOLIGON && m_pCurShape!=NULL)
      m_pCurShape->Show(pDC);

}
```

Чтобы операция рисования полигонов, также подлежала отмене, в обработчик нажатия правой клавиши добавим единственную строчку:

```
// файл Paintvw.cpp
/////////////////////////////////////////////////
void CPainterView::OnRButtonDown(UINT nFlags, CPoint point)
{
      // TODO: Add your message handler code here and/or call default
      CPainterDoc *pDoc=GetDocument();
      switch(m_LastOper)
      {
      case OP_NOOPER:
            if(pDoc->m_nIndex>0) pDoc->m_nIndex--;
            Invalidate();
            pDoc->SetModifiedFlag();
      break;
      case OP_DRAWCIRCLE:
      case OP_DRAWSQUARE:
      case OP_DRAWPOLIGON:
            if(pDoc->DeleteLastShape())
            {
                  Invalidate();
                  pDoc->SetModifiedFlag();
            }
      break;
      }

      Invalidate();
      pDoc->SetModifiedFlag();
      CView::OnRButtonDown(nFlags, point);
```

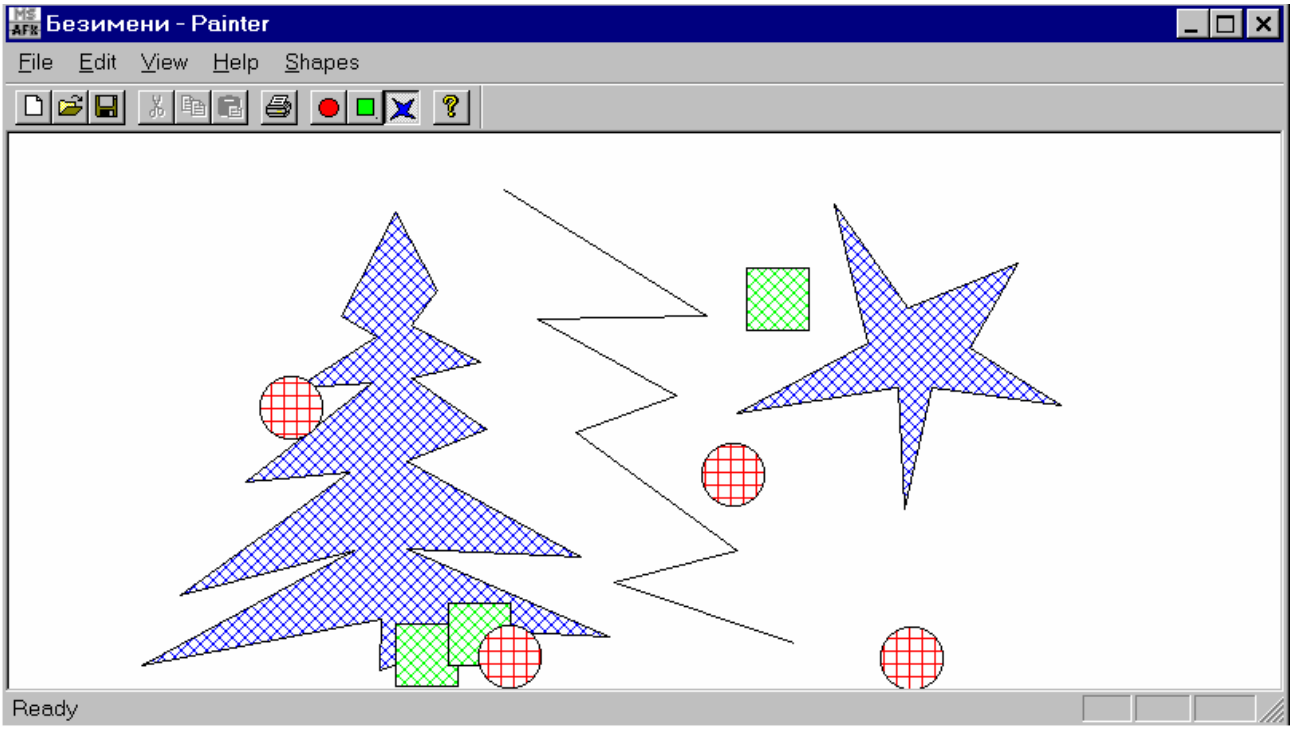


Рис. 3.9. Программа Painter с функцией рисования полигонов

Смотри также рекомендуемую литературу [1, 11].

4. ПРЕОБРАЗОВАНИЯ В ТРЕХМЕРНОМ ПРОСТРАНСТВЕ. ПАРАЛЛЕЛЬНАЯ И ПЕРСПЕКТИВНАЯ ПРОЕКЦИИ

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 уравнениями $\begin{cases} x' = x + a_1 \\ y' = y + a_2 \\ z' = z + a_3 \end{cases}$, где a_1, a_2, a_3 - константы.

В матричном виде данная операция будет выглядеть следующим образом:

$$[x', y', z', 1] = [x, y, z, 1]T,\ T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix} \qquad (4.1)$$

Поворот вокруг координатных осей может быть записан и без использования однородных координат. Для кратности записи так и поступим. В правой координатной системе (рис. 4.1) поворот вокруг оси z на угол α описывается следующей матрицей преобразования.

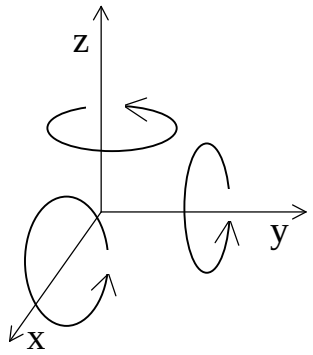


Рис. 4.1. Правая система координат

$$[x', y', z', 1] = [x, y, z, 1]R_z,\ R_z = \begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad (4.2)$$

где $c = \cos\alpha$ и $s = \sin\alpha$.

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. 4.2) $R_z \to R_x \to R_y \to R_z$.

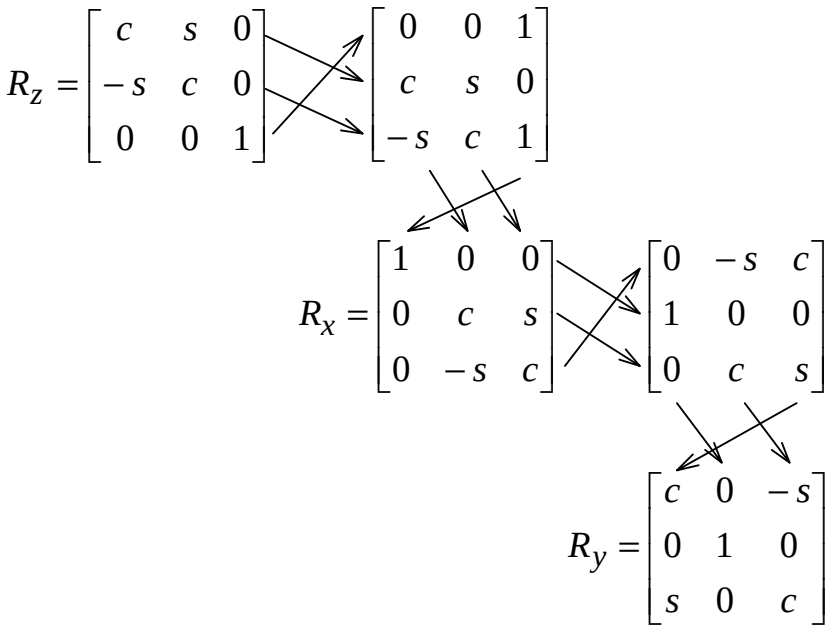


Рис. 4.2. Схема преобразования матриц

При необходимости изменения координатной системы используют инвертированные матрицы.

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a_1 & -a_2 & -a_3 & 1 \end{bmatrix}, \qquad R_z^{-1} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$R_x^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix}, \qquad R_y^{-1} = \begin{bmatrix} c & 0 & s \\ 0 & 1 & 0 \\ -s & 0 & c \end{bmatrix}. \quad (4.3)$$

Параллельная проекция.

Для выполнения преобразований необходимы: точка наблюдения, объект и экран. Экран находиться между наблюдателем и объектом (рис. 4.3). Если камера (глаз) находиться 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 множества точек $P(x, y, z)$, принадлежащих объекту в координаты точек изображения на экране $P'(X, Y)$. (x, y, z) – мировые координаты точки P объ-

екта; (X,Y) – экранные координаты ее проекции – точки P'. Преобразование включает в себя этапы, изображенные на схеме рис. 4.4.

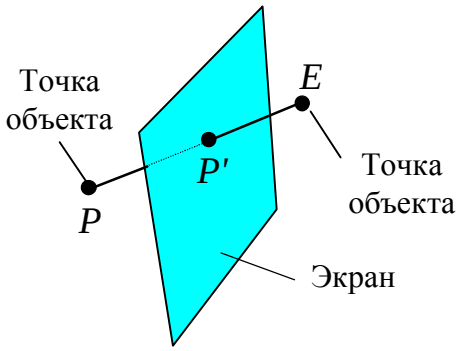


Рис. 4.3. Проекция точки объекта на экран

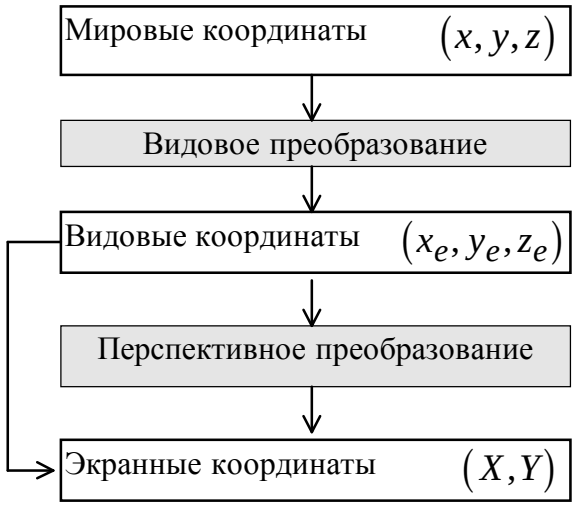


Рис. 4.4. Схема преобразования

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. Перспективное преобразование можно не выполнять. В этом случае значения x_e и y_e могут быть сразу использованы в качестве экранных координат (X,Y) точки P'.

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* (см. рис. 4.5):

$$x_e = \rho\sin\varphi\cos\theta,\ y_e = \rho\sin\varphi\sin\theta,\ z_e = \rho\cos\varphi. \qquad (4.4)$$

Вектор *EO* определяет направление наблюдения (рис. 4.5).

Система видовых координат показана на рис. 4.6. Кроме положения в пространстве, она отличается от мировых координат тем, что является левосторонней (мировая – правосторонняя). В матричном виде запись преобразования координат:

$$[x_e, y_e, z_e, 1] = [x, y, z, 1]\mathbf{V} \qquad (4.5)$$

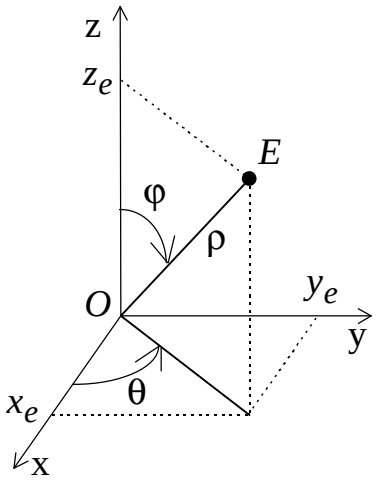


Рис. 4.5. Полярные координаты точки *E*

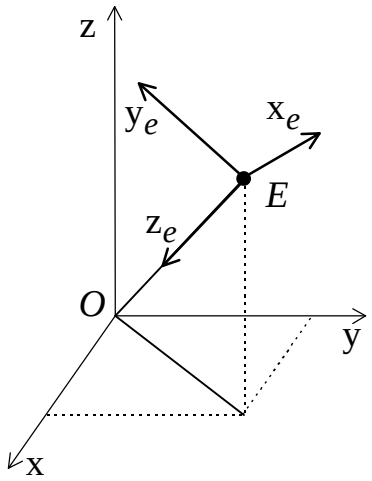


Рис. 4.6. Система видовых координат

Для получения матрицы ***V*** требуется перемножение матриц четырех элементарных преобразований:

1. Перенос из точки *O* в точку *E* (рис. 4.7).

Матрица данного преобразования: $T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_e & -y_e & -z_e & 1 \end{bmatrix}$ (4.6)

2. Поворот координатной системы вокруг оси z на угол $\frac{\pi}{2} - \theta$ (см. рис. 4.7, 4.8). В результате ось y совпадет по направлению с горизонтальной составляющей вектора OE, а ось x будет перпендикулярна этой составляющей. Так как поворот *координатной системы* выполняется в *отрицательном* направлении, матрица данного преобразования будет совпадать с матрицей поворота *точки* на такой же угол в *положительном* направлении:

$$R_z = \begin{bmatrix} \cos(\pi/2-\theta) & \sin(\pi/2-\theta) & 0 \\ -\sin(\pi/2-\theta) & \cos(\pi/2-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \sin\theta & \cos\theta & 0 \\ -\cos\theta & \sin\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.7)$$

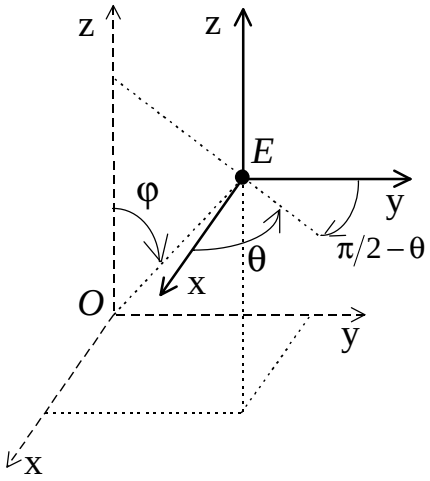


Рис. 4.7. Перенос в точку E

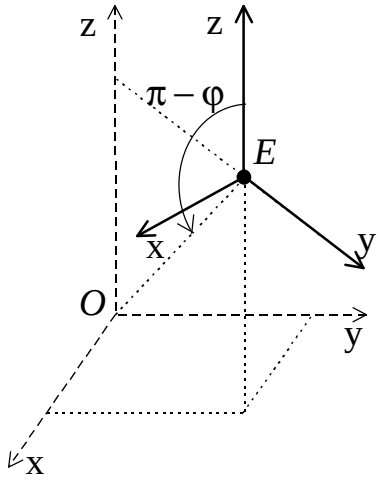


Рис. 4.8. Поворот вокруг z

3. Поворот системы вокруг оси x в положительном направлении на угол $\pi-\varphi$, что соответствует повороту точки на угол $-(\pi-\varphi)=\varphi-\pi$ (см. рис. 4.8, 4.9).

Матрица данного преобразования:

$$R_x=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos(\varphi-\pi) & \sin(\varphi-\pi)\\ 0 & -\sin(\varphi-\pi) & \cos(\varphi-\pi)\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 0 & -\cos\varphi & -\sin\varphi\\ 0 & \sin\varphi & -\cos\varphi\end{bmatrix}. \qquad (4.8)$$

4. Изменение направления оси x: $x'=-x$ (рис. 4.10).

Матрица данного преобразования:

$$\boldsymbol{M}_{yz}=\begin{bmatrix}-1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}. \qquad (4.9)$$

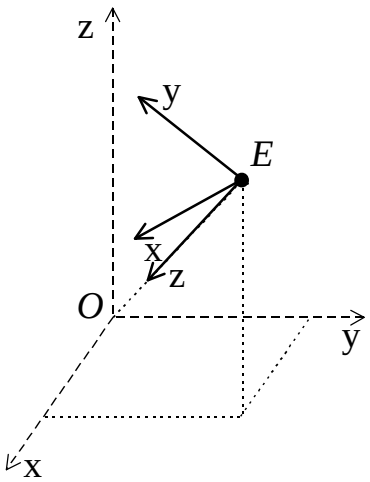


Рис. 4.9. Поворот вокруг оси x

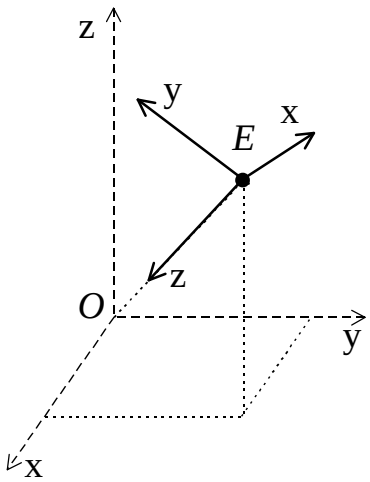


Рис. 4.10. Изменение направления оси y

Матрицу $\boldsymbol{V}$ найдем путем перемножения матриц (4.6-4.9):

$$\boldsymbol{V} = \boldsymbol{T}\boldsymbol{R}_z^*\boldsymbol{R}_x^*\boldsymbol{M}_{yz}^* \tag{4.10}$$

(* означает расширение матрицы 3x3 до 4x4 путем добавления строки и столбца 0, 0, 0, 1).

Результат перемножения – матрица преобразования в видовые координаты:

$$\boldsymbol{V} = \begin{bmatrix} -\sin\theta & -\cos\varphi\cos\theta & -\sin\varphi\cos\theta & 0 \\ \cos\theta & -\cos\varphi\sin\theta & -\sin\varphi\sin\theta & 0 \\ 0 & \sin\varphi & -\cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.11}$$

Видовые координаты точки находятся путем перемножения ее мировых координат (в однородной записи) на матрицу $\boldsymbol{V}$ (4.5).

Видовые координаты x_e, y_e – ортогональная (параллельная) проекция точки $P(x, y, z)$ – их можно непосредственно использовать для формирования изображения на экране.

Перспективные преобразования.

Рассмотрим Рис. 4.11. Пусть координата y точки P равна 0. Из подобия треугольников ΔEPO и $\Delta EP'Q$ следует:

$$\frac{P'Q}{EQ} = \frac{PO}{EO} \Rightarrow \frac{X}{d} = \frac{x}{z} \Rightarrow X = d\frac{x}{z}. \tag{4.12,а}$$

Аналогично для Y найдем:

$$Y = d\frac{y}{z} \tag{4.12,б}$$

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 (4.12) соответствующим смещением:

$$X = d\frac{x}{z} + \frac{W}{2},\ Y = d\frac{y}{z} + \frac{H}{2}, \tag{4.13}$$

где W, H, соответственно, ширина и высота экрана.

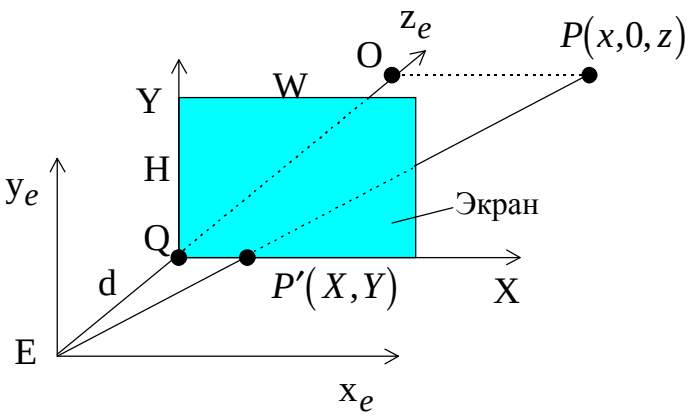


Рис. 4.11. Перспективное преобразование

По традиции реализуем рассматриваемый материал на практике – добавим в проект Painter возможности создания и манипуляции трехмерными объектами. Реализация таких возможностей потребует сравнительно большого объема работы. Поэтому посвятим этому вопросу всю следующую главу.

Смотри также рекомендуемую литературу [1, 11].

5. ПРОГРАММИРОВАНИЕ ПРЕОБРАЗОВАНИЙ В ТРЕХМЕРНОМ ПРОСТРАНСТВЕ. ПРОЕКТ PAINTER 4. СОЗДАНИЕ ТРЕХМЕРНЫХ ГРАФИЧЕСКИХ ОБЪЕКТОВ. РЕАЛИЗАЦИЯ ФУНКЦИЙ ТРЕХМЕРНЫХ ПРЕОБРАЗОВАНИЙ

Определение классов для работы с трехмерными фигурами. Для реализации поддержки трехмерных объектов включим в проект Painter необходимые классы. Как и остальные классы фигур, данные классы будут производными от класса CBasePoint (см. рис. 5.1). Прежде всего, определим две структуры:

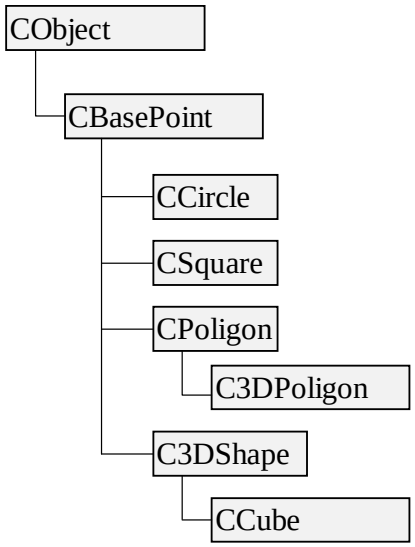


Рис. 5.1. Иерархия классов фигур программы Painter 4

POINT3D – для хранения координат точки в трехмерном пространстве;

Perspective – для хранения параметров трехмерной сцены.

Будем представлять любую трехмерную фигуру в виде набора полигонов в трехмерном пространстве. Для задания такого полигона определим класс C3DPoligon.

Класс C3DPoligon является производным от класса CPoligon и, соответственно, наследует его переменные и методы. Массив точек m_Points, определенный в классе CPoligon будем использовать для хранения экранных координат трехмерного полигона. Для хранения же мировых координат в классе C3DPoligon определим массив трехмерных точек m_3DPoints. Для расчета экранных координат полигона определим метод MakeProjection(), которому в качестве аргумента будем передавать параметры трехмерной сцены (координаты точки наблюдения, расстояние до экрана и до объекта).

Ниже приведено описание структур POINT3D, Perspective и класса C3DPoligon в файле SHAPES.H.

```
//Точка в 3D
struct POINT3D
{
     LONG x, y, z;
};

//Параметры трехмерной сцены
struct Perspective
{
     POINT3D O;               //точка, вокруг которой выполняем пово-
рот
     double rho, theta, phi,  //полярные координаты точки
                              //наблюдения (E)
     d;                       //расстояние от E до экрана
```

```
     WORD with_perspective;  //1- вкл. перспективные преобразования;
                             //0-выкл.
};

////////////////////////////////////////////////
//класс C3DPoligon
class C3DPoligon: public CPoligon
{
     DECLARE_SERIAL(C3DPoligon)
protected: // Виртуальный метод сериализации
     virtual void Serialize(CArchive& ar);
public:
     //конструкторы
     C3DPoligon();
     C3DPoligon(CPoint point);
     ~C3DPoligon();
// Данные
     POINT3D m_3DPoints[MAXPOLIGONLENGTH]; //точки в мировых
                                           координатах
// Методы
     virtual void Show(CDC *pDC);     //Отображение фигуры на экране
     BOOL AddPoint(POINT3D point);    //добавить точку
     void MakeProjection(Perspective P);   //выполнить расчет
                                           //экранных координат
};
```

Для описания трехмерной фигуры определим класс C3DShape. В классе C3DShape определим список указателей на трехмерные полигоны m_PtrPoligonList, в котором будем сохранять адреса полигонов, из которых состоит фигура. Ниже приведено описание класса C3DShape в файле SHAPES.H.

```
////////////////////////////////////////////////
//класс C3DShape
class C3DShape: public CBasePoint
{
     DECLARE_SERIAL(C3DShape)
protected: // Виртуальный метод сериализации
     virtual void Serialize(CArchive& ar);
public:
     //конструкторы
     C3DShape();
     C3DShape(CPoint point);
     ~C3DShape();
//Данные
     //параметры сцены
     Perspective m_Percpective;
     // список указателей на полигоны
     CTypedPtrList<CObList, C3DPoligon*> m_PtrPoligonList;
//Методы
     // Отображение фигуры на экране
     virtual void Show(CDC *pDC);
     // Виртуальный метод - проверка на попадание в фигуру
     virtual BOOL PointInShape(CPoint point);
     // добавить полигон
     void AddPoligon(C3DPoligon *pPoligon);
```

```
      // расчет проекции
      void MakeProjection();
      // Реакция на нажатие клавиши
      virtual void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
//нажали
};
```

Кроме списка трехмерных полигонов в классе C3DShape определим переменную m_Percpective, определенного нами типа Perspective, – это позволит нам задавать для каждой трехмерной фигуры свои параметры наблюдения. Класс C3DShape имеет метод MakeProjection(), в теле которого вызывается одноименный метод с параметром m_Percpective для всех объектов–трехмерных полигонов из списка m_PtrPoligonList.

От класса C3DShape можно порождать классы конкретных фигур, в конструкторах которых определять их форму. В качестве примера от класса C3DShape порожден класс CCube (реализующий фигуру - трехмерный куб), описание которого приведено ниже.

```
////////////////////////////////////////////////
//класс CCube
class CCube: public C3DShape
{
      DECLARE_SERIAL(CCube)
protected: // Виртуальный метод сериализации
      virtual void Serialize(CArchive& ar);
public:
      //конструкторы
      CCube();
      CCube(CPoint BasePoint, WORD side);
      ~CCube(){};
      void Create(CPoint point, WORD side);
//Данные
      WORD m_wSide; //длина стороны
};
```

У класса CCube определено два конструктора, один из них пустой – необходим для реализации механизма сериализации, другой с параметрами. Второму конструктору в качестве параметров передаются точка, вокруг которой должен быть расположен куб, и сторона куба. В теле этого конструктора вызывается метод Create(), создающий трехмерные полигоны, задающие стороны куба. После этого выполняется расчет экранных координат куба.

В проекте Painter 4 несколько модифицирован и базовый класс CBasePoint. В него введены новые переменные, задающие цвет фигуры и шаг модификации ее положения, добавлены методы определения попадания в фигуру и реакции фигуры на нажатие пользователем клавиш. Ниже приведено описание модифицированного класса CBasePoint.

```
//////////////////////////////////////////////////
//класс CBasePoint - базовая точка

class CBasePoint: public CObject
{
     DECLARE_SERIAL(CBasePoint)
 protected:          // Виртуальный метод сериализации
     virtual void Serialize(CArchive& ar);
 public:
     //конструкторы
     CBasePoint(CPoint point); //конструктор с параметрами
     CBasePoint();//конструктор без параметров
     ~CBasePoint(){}; //деструктор
// Данные
     double m_dScale; //масштаб фигуры
     LONG m_x;
     LONG m_y;
     COLORREF m_Color; //цвет
     LONG m_Step; //шаг
     BOOL m_FlagShift; //нажата клавища Shift
     BOOL m_FlagCtrl; //нажата клавища Ctrl
// Методы
     // Отображение фигуры на экране - чистый виртуальный метод
     virtual void Show(CDC *pDC);
     // Виртуальный метод сообщающий область захвата
     virtual void GetRegion(CRgn &Rgn);
     // Виртуальный метод - проверка на попадание в фигуру
     virtual BOOL PointInShape(CPoint point);
     // Виртуальный метод двумерное преобразование
     virtual CPoint Transform(const CPoint &point,
               const CPoint &point0, double ang, int a, int b);
     // Установка цвета
     virtual void SetColor(COLORREF color){m_Color=color;};
     virtual void SetStep(LONG step){m_Step=step;};
     // Реакция на нажатие клавиши
     virtual  void  OnKeyDown(UINT  nChar,  UINT  nRepCnt,  UINT  nFlags);
//нажали
     virtual  void  OnKeyUp(UINT  nChar,  UINT  nRepCnt,  UINT  nFlags);
//отпустили

};
```

Рассмотрим далее, какие изменения потребовались в классе CPaintDoc. Во-первых, в класс добавлен метод SelectShape(), который позволяет осуществить выбор фигуры. Этому методу в качестве аргумента передается точка, в которой была нажата левая клавиша мыши. Результат работы данного метода – присвоение переменной m_pCurShape типа указатель на CBasePoint адреса первой фигуры из списка, в которую попали указателем мыши. Далее этой фигуре передаются сообщения о нажатии различных клавиш. Обрабатывая эти сообщения, фигура может менять свое положение на экране. Для инициации процесса выбора фигуры в меню Shapes добавлена команда Select, а для

создания фигуры - куба – команда Cube. В панель инструментов добавлены соответствующие кнопки.

Полностью тексты файлов SHAPES.H., SHAPES.CPP, PAINTDOC.H, PAINTDOC.CPP, PAINTVW.H, PAINTVW.CPP приведены в архиве Painter4.rar.

На рис. 5.2. показана работа программы Painter 4.

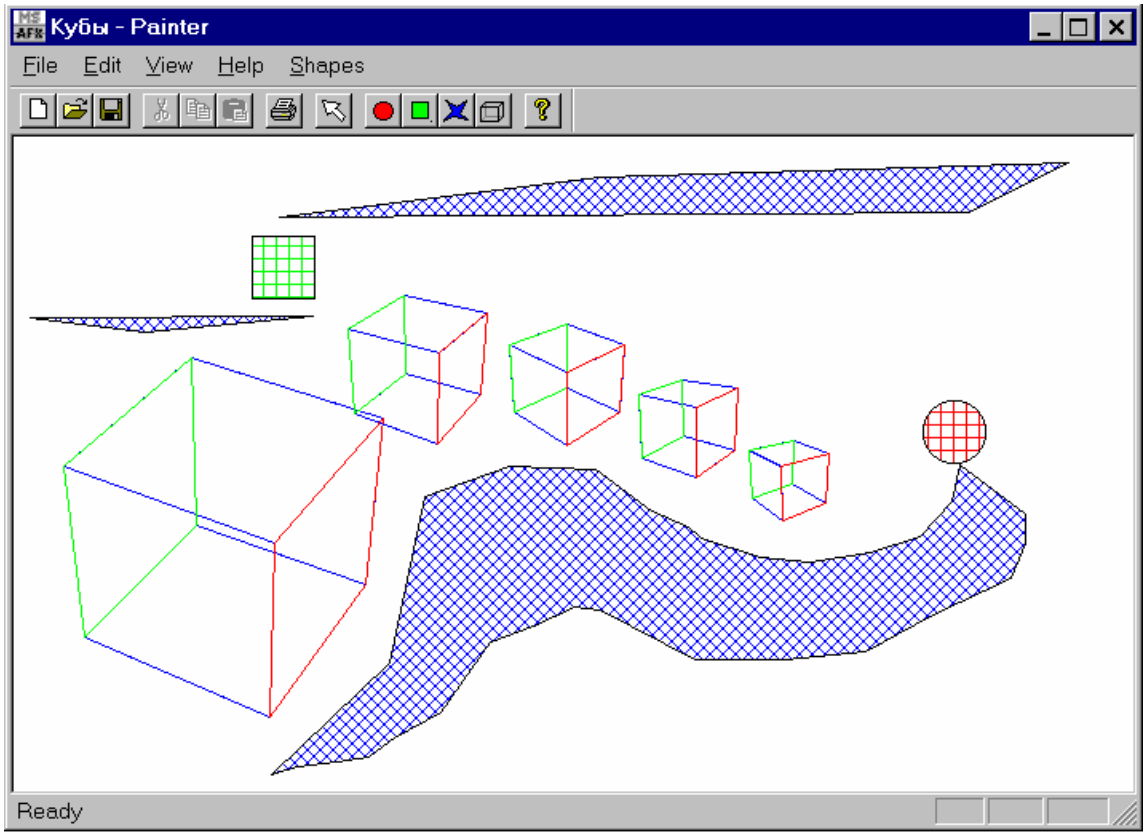


Рис. 5.2. Программа Painter 4 в работе

6. УДАЛЕНИЕ НЕВИДИМЫХ ЛИНИЙ И ПОВЕРХНОСТЕЙ. ПРОЕКТ PAINTER 5

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.

Второй подход заключается в определении для каждого пиксела экрана ближайшего к нему объекта (вдоль направления проецирования). При этом работа ведется в пространстве экранных координат.

Рассмотрим некоторые алгоритмы, реализующие первый подход.

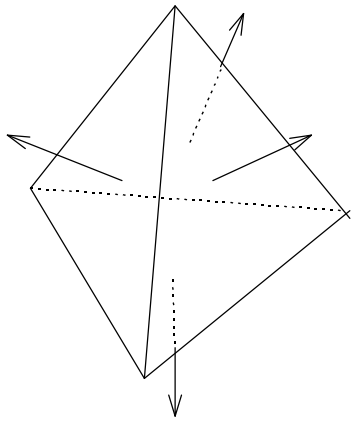


Рис. 6.1. Определение нелицевых граней

Отсечение 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 (см. рис. 6.1).

В случае, кода 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.

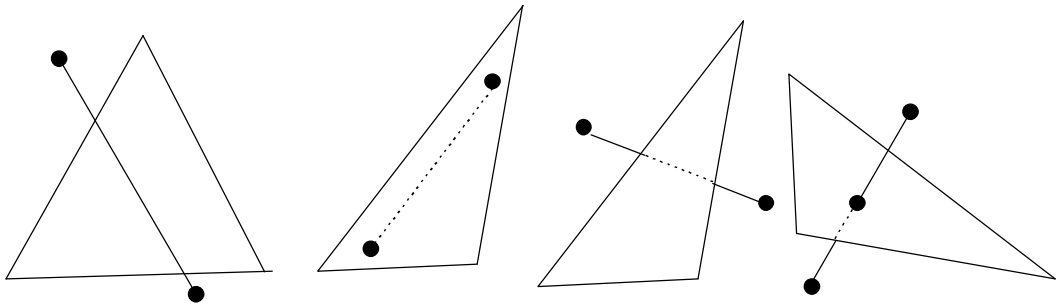
Алгоритм Робертса. Требуется, чтобы каждая грань была выпуклым многогранником. Проверяется видимость ребер многогранника путем тестирования их на перекрытие гранями.

Сначала отбрасываются все ребра, обе определяющие грани которого являются нелицевыми.

Далее ребра проверяются на перекрытие гранями. Возможны следующие случаи (рис. 6.2):

- грань не закрывает ребро (рис. 6.2,а);
- грань полностью закрывает ребро (на этом проверка видимости ребра заканчивается) (рис. 6.2,б);

• грань частично закрывает ребро (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. 6.2,в, г).



а) б) в) г)

Рис. 6.2. Варианты взаимного расположения ребра и грани

Количество проверок можно значительно сократить, если воспользоваться разбиением картинной плоскости (экрана). При этом картинная плоскость разбивается на равные клетки и для каждой клетки составляется список граней, проекции которых попадают в нее. При определении видимости ребра, сначала устанавливается, в какие клетки попадает его проекция, и далее проверяются лишь грани, содержащиеся в списках данных клеток.

Рассмотрим методы второго подхода.

Метод z-буфера.

Один из самых простых алгоритмов.

Каждому пикселу экрана сопоставляется расстояние до проецируемого на него объекта (z-буфер). Для вывода на экран произвольной грани, она сначала переводиться 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 $+\infty$), то пиксел рисуется и его значение заноситься в z-буфер.

В связи с простотой и трудоемкостью алгоритма распространены его аппаратные реализации.

Алгоритм Варнака.

Основан на «разбиении» экрана на части. Сначала разделим картинную плоскость на 4 равные части (Рис. 6.3). 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 и втором случаях часть экрана полностью закрашивается соответствующим цветом. В третьем случае данная часть разбивается еще на 4 части, для каждой из которой вновь выполняется проверка, и так далее. Разбиение можно выполнять пока размер части не будет соответствовать одному пикселу (если до этого дошло, то пиксел закрашивается цветом ближайшей к нему грани).

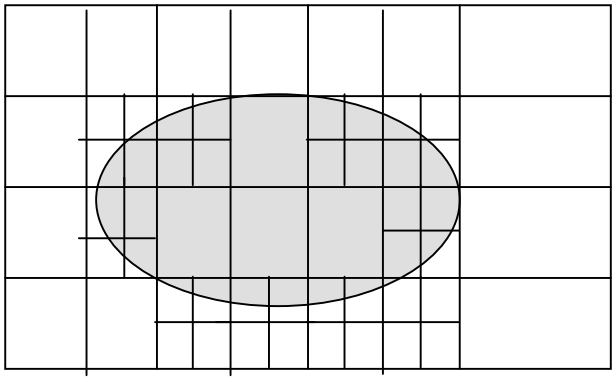


Рис. 6.3. Разбиение экрана на части

Метод построчного сканирования.

Все изображение на экране представляет собой набор вертикальных линий (полосу 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 реализации перемещения по прямоугольному лабиринту. В случае если высота пола и потолка одинакова, а стены вертикальны, задача вообще сводиться к двумерной (см. рис. 6.4 вид сверху). Разложим изображение на экране в ряд вертикальных линий. Каждая такая линия определяется пересечением вертикальной полуплоскости, проходящей через центр проекции (камеру) и заданную вертикальную линию. Видимым будет ближайшее пересечение со стенами лабиринта.

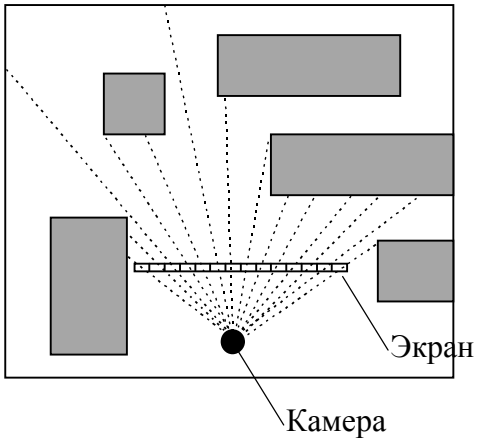


Рис. 6.4. Вид сверху на лабиринт

Каждая вертикальная линия может состоять из трех частей – пола, стены и потолка. Решая задачу в двумерном пространстве, определяем расстояние до ближайшей стены. Часть линии закрашиваем цветом пола, часть цветом стены (в зависимости от расстояния можно менять интенсивность цвета), часть цветом потолка.

Рассмотрим более подробно, как можно реализовать удаление невидимых ребер с использованием алгоритма Робертса.

Обозначения:

PQ – отрезок прямой;

ABC – треугольник;

E – точка наблюдения;

$R \in PQ$ – некоторая точка отрезка *PQ*.

Любую грань объекта с заданной точностью можно представить в виде набора треугольников (всегда выпуклый многоугольник). Такое разбиение называют триангуляцией.

Будем считать, что треугольник *ABC* закрывает точку *R*, если отрезок *ER* пересекает *ABC* в точке, внутренней как по отношению к треугольнику, так и по отношению к отрезку. Стороны треугольника и концы отрезка не являются внутренними. Треугольник может закрывать отрезок частично.

Пусть заданы видовые координаты точек *A*, *B*, *C*, *P*, *Q*.

Представим себе бесконечную пирамиду, ребрами которой являются: *EA*, *EB*, *EC*. Все позади треугольника *ABC* (внутри пирамиды) – невидимо. На рис. 6.5 *PI* и *JQ*– видимые отрезки, *IJ* – невидимый.

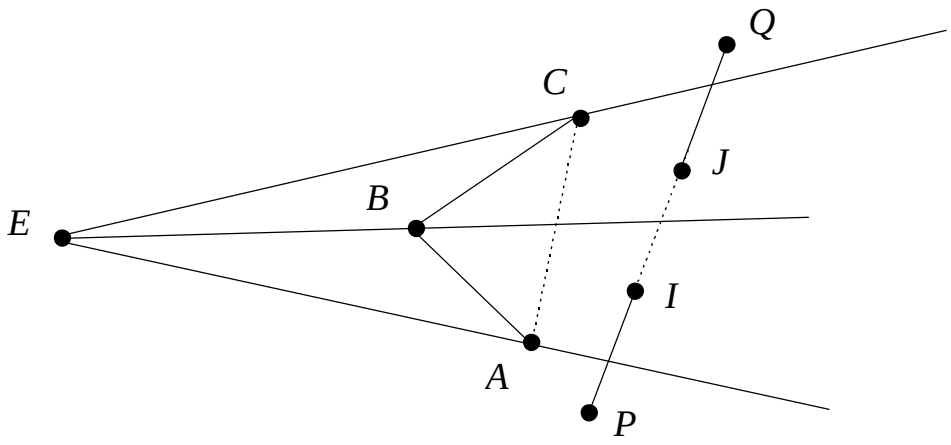


Рис. 6.5. Воображаемая бесконечная пирамида

Векторное представление отрезка *PQ*: $EP+\lambda \boldsymbol{r}$, где $\boldsymbol{r} = [r_1, r_2, r_3] = PQ$, т.е.

$$r_1 = X_Q - X_P$$

$$r_2 = Y_Q - Y_P$$

$$r_3 = Z_Q - Z_P$$

некоторая точка (*x*, *y*, *z*) принадлежит прямой отрезку *PQ*, если:

$$\begin{aligned} x &= X_Q + \lambda r_1 \\ y &= Y_Q + \lambda r_2 \ \text{ при } \lambda = [0,1]. \\ y &= Z_Q + \lambda r_3 \end{aligned} \qquad (6.1)$$

Уравнение плоскости EAB может быть записано:

$$\begin{vmatrix} x & y & z \\ X_A & Y_A & Z_A \\ X_B & Y_B & Z_B \end{vmatrix} = 0, \text{ т.к. } E = (0,0,0)$$

перепишем в виде:

$$C_1 x + C_2 y + C_3 z = 0, \qquad (6.2)$$

где

$$C_1 = Y_A Z_B - Y_B Z_A$$
$$C_2 = X_B Z_A - X_A Z_B \qquad (6.2')$$
$$C_3 = X_A Y_B - X_B Y_A$$

Подставляя правые части из (1) в (2) находим

$$\lambda = \frac{(C_1 X_P + C_2 Y_P + C_3 Z_P)}{(C_1 r_1 + C_2 r_2 + C_3 r_3)}. \qquad (6.2'')$$

Подставляя λ в (1) находим координаты точки *I*. Однако $I \in PQ$ только при $\lambda = [0,1]$, но и этого не достаточно, необходимо так же, чтобы и отрезок *AB* – пересекался плоскостью *EPQ* (рис. 6.6).

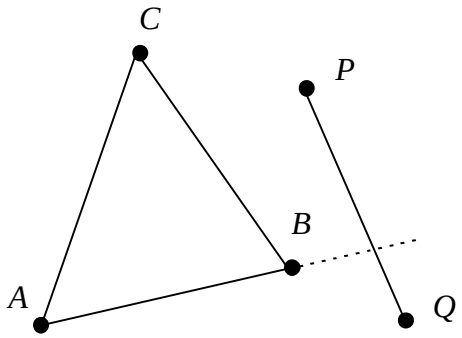


Рис. 6.6. Сечение *PQ* плоскостью *EAB*

Аналогично случаю с плоскостью *EAB* и прямой *PQ*. Для плоскости *EPQ* и прямой *AB* запишем:

$$\mu = \frac{K_1 X_A + K_2 Y_A + K_3 Z_A}{K_1 (X_B - X_A) + K_2 (Y_B - Y_A) + K_3 (Z_B - Z_A)}, \qquad (6.2''')$$

$$K_1 = Y_P Z_Q - Y_Q Z_P$$

где $K_2 = X_Q Z_P - X_P Z_Q$, $\mu = [0,1]$.

$$K_3 = X_P Y_Q - X_Q Y_P$$

Таким образом, *PQ* и плоскость *EAB* имеют общую точку тогда и только тогда, когда $0 \le \lambda \le 1$ и $0 \le \mu \le 1$.

Необходимо также рассмотреть пересечения *PQ* с плоскостями *EBC* и *ECA*. Отрезок *PQ* может иметь 0, 1, 2 пересечений с пирамидой.

Проверку надо выполнить для всех отрезков и треугольников сцены.

Тест 1.

Если *P* и *Q* находятся перед или на плоскости *ABC* (не позади нее), то отрезок *PQ* – видимый. Это случается когда:

$\boldsymbol{n}\cdot EP \le h$ и $\boldsymbol{n}\cdot EQ \le h$, где $\boldsymbol{n}=[a\ b\ c]$ – вектор коэффициентов плоскости *ABC*, *h* – правая часть уравнения плоскости:

$$ax + by + cz = h, \qquad (6.3)$$

a, b, c выбираются такими, чтобы

$$a^2 + b^2 + c^2 = 1 \text{ и } h \ge 0.$$

Тогда (3) можно записать в виде скалярного произведения:

$\boldsymbol{n}\cdot\boldsymbol{x} = h$, $\boldsymbol{n}=[a\ b\ c]$, $\boldsymbol{x}=[x\ y\ z]$,

$\boldsymbol{n}$ – вектор нормали единичной длины, перпендикулярный направлению плоскости треугольника.

Для любой точки *X* плоскости вектор *EX* имеет свойство: его скалярное произведение $\boldsymbol{n}\cdot EX = h$ – равно расстоянию между точкой *E* и плоскостью.

Плоскость, проходящая через ABC, описывается уравнением:

$$\begin{vmatrix} x & y & z & 1 \\ X_A & Y_A & Z_A & 1 \\ X_B & Y_B & Z_B & 1 \\ X_C & Y_C & Z_C & 1 \end{vmatrix} \Rightarrow \begin{vmatrix} Y_A & Z_A & 1 \\ Y_B & Z_B & 1 \\ Y_C & Z_C & 1 \end{vmatrix} x - \begin{vmatrix} X_A & Z_A & 1 \\ X_B & Z_B & 1 \\ X_C & Z_C & 1 \end{vmatrix} y + \begin{vmatrix} X_A & Y_A & 1 \\ X_B & Y_B & 1 \\ X_C & Y_C & 1 \end{vmatrix} z = \begin{vmatrix} X_A & Y_A & Z_A \\ X_B & Y_B & Z_B \\ X_C & Y_C & Z_C \end{vmatrix}$$

Отсюда следует:

$$a = Y_A(Z_B - Z_C) - Y_B(Z_A - Z_C) + Y_C(Z_A - Z_B);$$

$$b = -\big(X_A(Z_B - Z_C) - X_B(Z_A - Z_C) + X_C(Z_A - Z_B)\big);$$

$$c = X_A(Y_B - Y_C) - X_B(Y_A - Y_C) + X_C(Y_A - Y_B);$$

$$h = X_A(Y_B Z_C - Y_C Z_B) - X_B(Y_A Z_C - Y_C Z_A) + X_C(Y_A Z_B - Y_B Z_A).$$

Если $h>0$ выполним нормировку: $r = \sqrt{a^2 + b^2 + c^2}$; $a = a/r$; $b = b/r$; $c = c/r$; $h = h/r$. Иначе треугольник является нелицевым, его можно проигнорировать и перейти к рассмотрению следующего треугольника.

Тест 2.

Если прямая линия, проходящая через отрезок PQ, лежит вне пирамиды, то отрезок PQ – видимый (Рис. 6.7). Для выполнения теста подставляем значения координат точек A, B, C в левую часть уравнения $K_1x + K_2y + K_3z = 0$, определяющего плоскость EPQ. Если все три вычисленных значения имеют одинаковые знаки («+» или «-»), то все точки лежат по одну сторону от плоскости *EPQ*. Следовательно, отрезок *PQ* располагается вне пирамиды – видим. Обозначим знаки 1 и -1, соответственно, «+» и «-». Возможен также случай, когда одна из точек *A*, *B*, *C* лежит в плоскости *EPQ*

(Рис. 6.7, б) – этот случай обозначим 0. Если модуль «суммы знаков» больше или равен 2, то отрезок *PQ* видим.

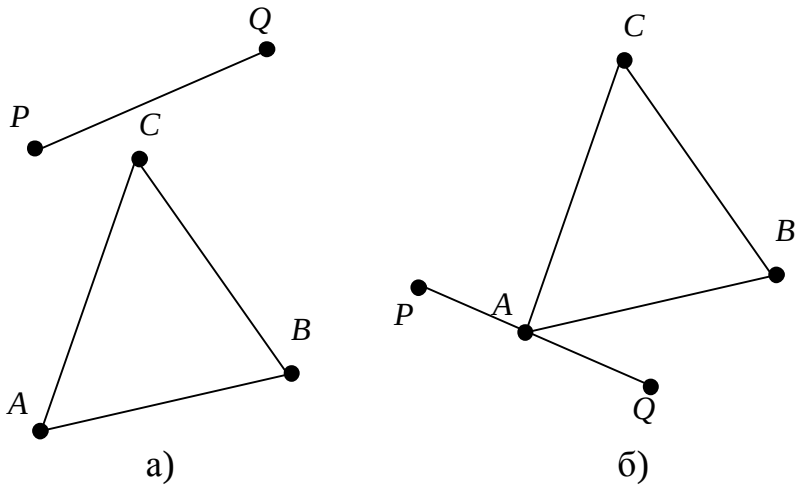


Рис. 6.7. Тест 2

Тест 3.

Найдем точку пересечения прямой PQ с плоскостями EAB, EBC, EAC.

Вычислим значения λ для точки пересечения прямой PQ с плоскостью EAB по уравнению (6.2'') и μ для точки пересечения прямой AB с плоскостью EPQ по уравнению (6.2'''). Если правая часть уравнения (6.2), описывающего плоскость EAB, при подстановке координат P и C принимает разные знаки, то это означает, что P лежит по другую сторону от AB относительно C, т.е. вне треугольника ABC однозначно. Запомним этот факт в переменной Poutside. Аналогично для Q – Qoutside. Poutside и Qoutside равны TRUE, если для P и C, а также Q и C правая часть уравнения (6.2) принимает разные знаки, а также, если для P и Q правая часть (6.2) ==0.

Таким образом, если P и Q лежат по одну сторону от треугольника ABC (Poutside и Qoutside равны TRUE) – отрезок PQ видимый (рис. 6.8).

Данный тест охватывает также случай, когда конец отрезка совпадает с вершиной треугольника.

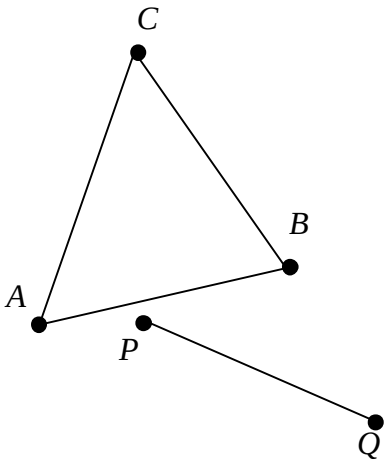


Рис. 6.8. Тест 3

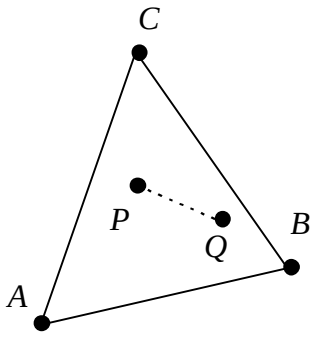


Рис. 6.9. Тест 4

Данный тест выполняется для каждой из плоскостей EAB, EBC и ECA, следовательно, имеем три пары значений (λ_i, μ_i) (i=1, 2, 3).

Найдем минимальное и максимальное из λ_i, удовлетворяющие условию $0 \le \lambda \le 1$ и $0 \le \mu \le 1$; обозначим min и max, соответственно.

Тест 4.

Если P и Q находятся внутри пирамиды (Poutside и Qoutside равны FALSE) для всех сторон треугольника ABC, то отрезок PQ позади треугольника и, следовательно, невидим (Рис. 6.9). Если PQ лежит позади отрезка AB, то он не вычерчивается.

Тест 5.

Если точка I лежит впереди треугольника, то PQ видима. Для нахождения I используем min и max λ_i из теста 3. Затем проверяем $EI \cdot \boldsymbol{n} \le h$ – точка I видима; отрезок PQ – видим. (Здесь предполагается, что PQ не может пересекать треугольник ABC во внутренних точках).

Тест 6.

Выполняется тогда, когда все предыдущие тесты не дали результатов, т.е. PQ пересекает пирамиду позади треугольника ABC.

Если значения min и max λ_i из теста 3 указывают на точки пересечения I и J, то отрезок IJ – невидим, и:

- если P за пределами треугольника ABC или перед плоскостью, PI – видимый;
- если Q вне пирамиды или перед плоскостью, то JQ – видимый;
- если P и Q не находятся одновременно вне пирамиды, то I и J совпадают.

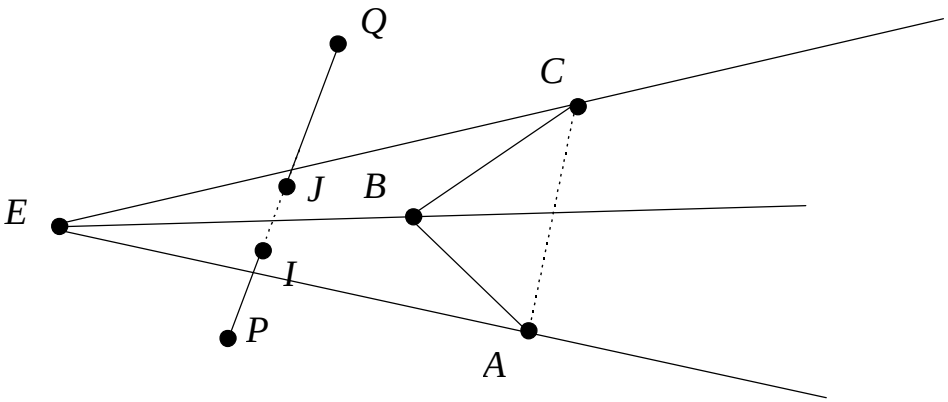


Рис. 6.10. Тест 6

В тестах 1-5 при определении видимости отрезка (по отношению к j-тому треугольнику) значение j увеличивается на 1. Затем проверяется види-

мость PQ по отношению с другими треугольниками (если не все проверены) или отрезок вычерчивается.

В тесте 6 могут появиться два новых отрезка (PI и JQ), которые должны быть проверены по отношению к оставшимся треугольникам. Здесь можно использовать рекурсивное обращение к этому же алгоритму.

На рис. 6.11 показана схема выполнения всех тестов.

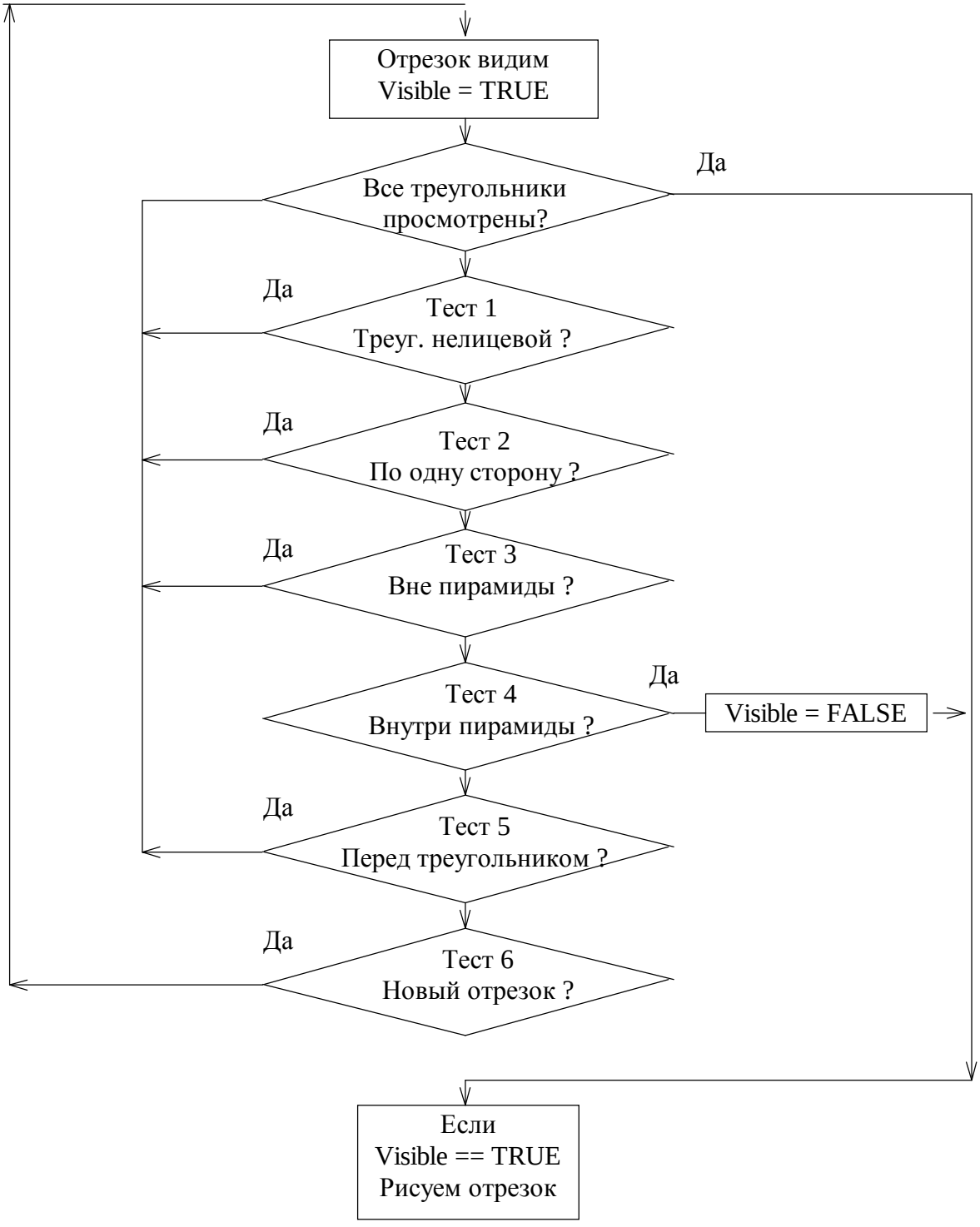


Рис. 6.11. Схема выполнения тестов

На рис. 6.12 показана работа программы с функцией удаления невидимых линий (изображаются пунктиром).

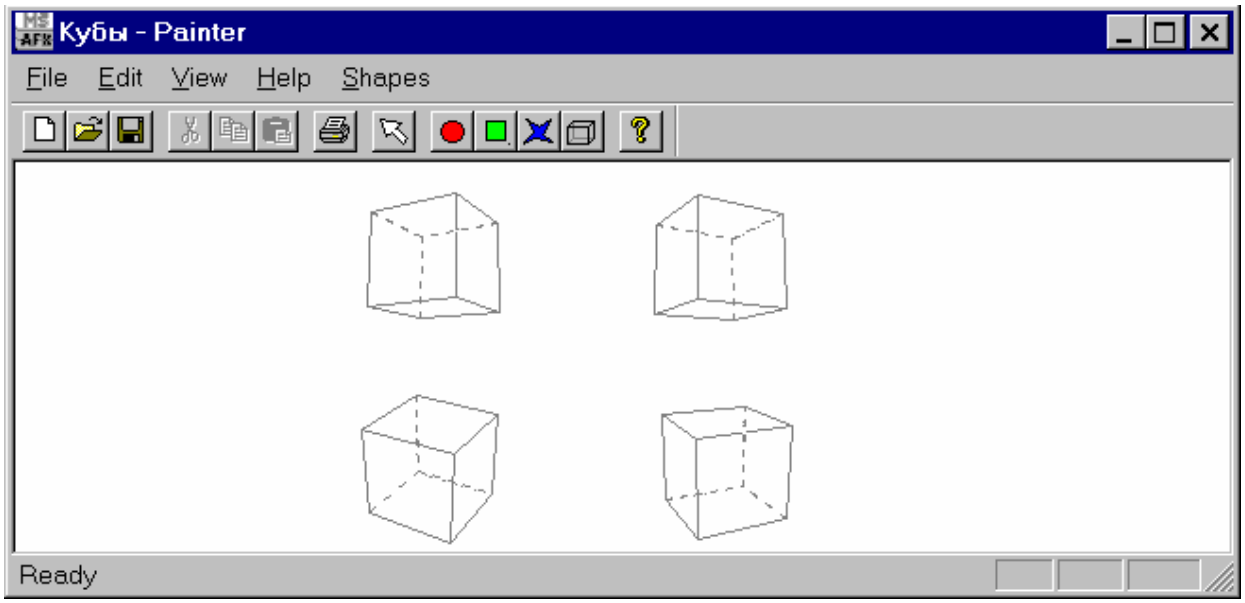


Рис. 6.12. Работа программы Painter 5

Добавим в проект Painter класс C3DObj, реализующий функции создания и отображения трехмерных фигур с удалением невидимых линий. От класса C3DObj можно порождать классы конкретных фигур, например, CSolidCub (см. рис. 6.13). Описание классов и реализация их методов приведена ниже. Полностью тексты программы приведены в архиве Painter5.rar.

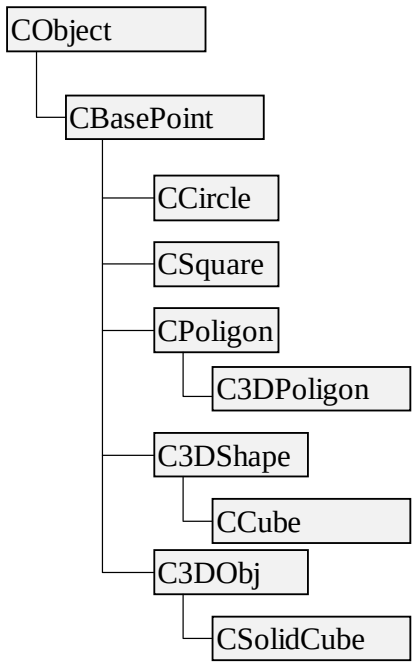


Рис. 6.13. Иерархия классов программы Painter 5

Перемещения куба по экрану осуществляется с помощью клавиш управления курсором (стрелками). Поворот куба осуществляется с помощью клавиш управления курсором при нажатой клавише Shift. Приближение и удаление камеры осуществляется стрелками "вверх", "вниз" при нажатой клавише Ctrl.

Данная реализация проекта Painter позволяет создавать трехмерные объекты с функцией удаления лишь собственных линий каждого объекта. Чтобы невидимыми становились также и объекты, перекрытые другими объектами, необходимо организовать общий для всех объектов цикл, в котором все отрезки сцены (всех объектов) будут проверяться на перекрытие всеми треугольниками всех объектов.

Надо отметить, что приведенная в программе Painter5 реализация алгоритма удаления невидимых линий годиться, разве что для иллюстрации работы алгоритма Робертса. Создание сложных трехмерных сцен с удалением невидимых линий и граней, фотореалистичной закраской, анимацией и массой других эффектов целесообразно выполнять с использованием готовых библиотек для работы с 3D графикой, таких как OpenGL и DirectX. Этим библиотекам будет посвящена отдельная лекция.

Смотри также рекомендуемую литературу [1, 2, 11].

7. ПОСТРОЕНИЕ КРИВЫХ

Задачи построения кривой, соединяющей набор базовых точек, возникают во многих областях народного хозяйства. Например, бравые капитаны дальних морей намечают маршруты своих судов, разрисовывая карты (порты в данном случае играют роль базовых точек). Отважные экспериментаторы проводят свои эксперименты 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, проходя по дистанции жизненного пути, кривые строят практически все. Более того, сам жизненный путь мало у кого бывает прямым. Поэтому неплохо было бы изучить некоторые теоретические и практические основы построения кривых.

В данной разделе мы коснемся лишь прикладного аспекта данной проблемы. Для более подробного ознакомления с математическими основами построения кривых и поверхностей можно порекомендовать книжки:

Шикин. Е.В., Боресков А.В. Компьютерная графика. Динамика, реалистические изображения. - М. Диалог-МИФИ, 1995.-288 с.

Шикин. Е.В., Плис. А.И. Кривые и поверхности на экране компьютера. Руководство по сплайнам для пользователей - М.: Диалог-МИФИ, 1996. - 240 с.

Определения

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 – 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. 7.1) – ***линейная интерполяция***. 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.

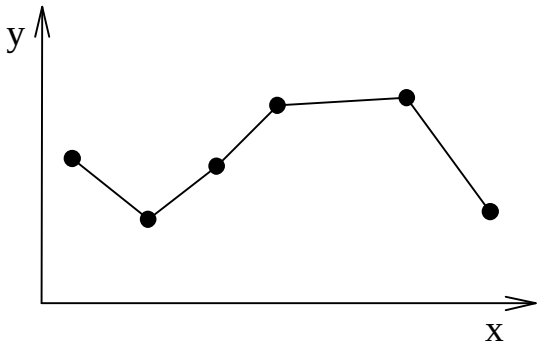


Рис. 7.1. Линейная интерполяция

Интерполяция с помощью некоторой гладкой кривой зачастую более приемлема.

Критерием гладкости является существование производных некоторой степени: 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:

$$f(\boldsymbol{x}) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = y. \qquad (7.1)$$

Неизвестные – коэффициенты полинома $a_i \quad i = (0, 1, \ldots, n)$. 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, – тем больше 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 строиться на основе четырех точек. Чтобы обеспечить гладкость в местах стыковки сегментов, построение кривой осуществляется лишь между двумя «внутренними» точками каждой четверки, а четверки выбираются с «перекрытием», т.е. первой точкой очередной четверки выбирается вторая точка предыдущей четверки. Например, сегмент 1 (рис. 7.2) строится на основе точек 1, 2, 3, 4, а сегмент 2 на основе точек 2, 3, 4, 5 и т.д. При таком подходе, чтобы построить кривую, начинающуюся в первой точке и заканчивающуюся в последней точке набора, эти точки дублируются.

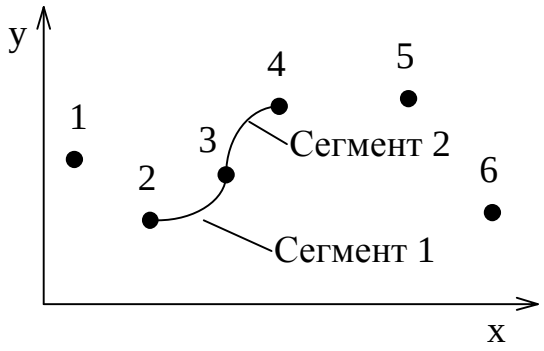


Рис. 7.2. Составная сплайновая кривая

Выше предполагалось, что координаты базовых точек заданы в виде $y_i(x_i)$ и расположены в порядке возрастания их абсциссы. Например, случай, когда у разных точек набора совпадают абсциссы (Рис. 7.3) не допускался. Поэтому сложные кривые (замкнутые или самопересекающиеся) удобно описывать при помощи параметрических уравнений.

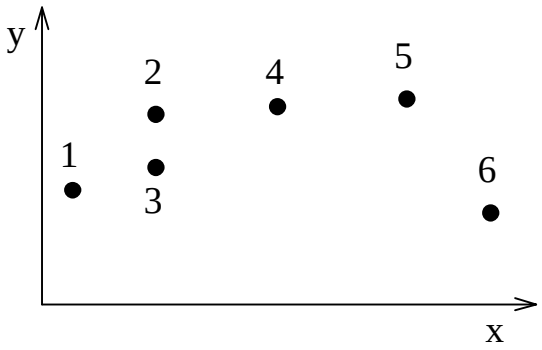


Рис. 7.3. Недопустимый набор точек

Параметрическое задание кривых.

Уравнения вида

$$x = x(t),\ y = y(t),\ z = z(t), \qquad (7.2)$$
$$\alpha \le t \le \beta$$

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 $\boldsymbol{R}$: $\boldsymbol{R}(t) = \big(x(t), y(t), z(t)\big)$, а массив опорных точек вектором $\boldsymbol{P}$: $\boldsymbol{P} = \big\{P_i(x_i, y_i, z_i), i = 0, 1, 2, \ldots, n\big\}$. Для каждой координаты рассчитывается своя независимая сплайновая кривая. Позиция на каждой из кривых $x(t)$, $y(t)$, $z(t)$ и составляет положение точки (x, y, z) сплайновой кривой. Переменная *t* называется параметром.

На практике для построения сплайновой кривой обычно используют метод составления кривой из отдельных сегментов, описываемых элементарными уравнениями, как правило, третьей степени. При этом поступают следующим образом.

1. 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.

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.

3. Разбивают диапазон изменения параметра на *m* частей (например, *m* =10).

4. На основе значений соответствующих координат четверки базовых точек и значения t_k, $k = (0, 1, \ldots, m)$ рассчитываются *m* точек сплайновой кривой.

5. Рассчитанные на предыдущем шаге точки соединяются прямыми. Таким образом, значение *m* определяет гладкость построения сплайновой кривой

Достоинства такого подхода в упрощении расчетов, использовании уравнений невысоких степеней, а также то, что при добавлении точки в базовый набор возникает необходимость пересчета лишь четырех сегментов.

При построении составной сплайновой кривой важно выполнение некоторых условий гладкости в точках их стыковки. Только в этом случае составная кривая будет удовлетворять 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 меняется также непрерывно.

Существует большое количество разных вариантов сплайновых кривых, отличающихся своими свойствами. Приведем примеры использования некоторых из них.

Интерполяционная кривая Catmull-Rom

По заданному массиву точек P_0, P_1, P_2, P_3 сплайновая кривая Catmull-Rom определяется при помощи уравнения, имеющим следующий вид:

$$\boldsymbol{R}(t)=\frac{1}{2}\left(-t(1-t)^2P_0+\left(2-5t^2+3t^3\right)P_1+t\left(1+4t-3t^2\right)P_2-t^2(1-t)P_3\right), \qquad (7.3)$$

$$0\le t\le 1.$$

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.

Сплайновая кривая Безье

По заданному массиву точек P_0, P_1, P_2, P_3 сплайновая кубическая элементарная кривая Безье описывается уравнением

$$\boldsymbol{R}(t)=\left(\left((1-t)P_0+3tP_1\right)(1-t)+3t^2P_2\right)(1-t)+t^3P_3, \qquad (7.4)$$

$$0\le t\le 1.$$

Элементарная кривая начинается в точке P_0 и заканчивается в точке P_3, касаясь при этом отрезков $P_0\,P_1$ и $P_2\,P_3$.

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 $\boldsymbol{P}_0$, ..., $\boldsymbol{P}_5$. Для построения геометрически непрерывной составной кривой дополним этот набор вспомогательной точкой $\boldsymbol{P}_*$, взятой на средине отрезка $\boldsymbol{P}_2\,\boldsymbol{P}_3$ (рис. 7.4). Составную кривую построим из двух сегментов элементарных кубических кривых Безье для четверок вершин $\boldsymbol{P}_0$, $\boldsymbol{P}_1$, $\boldsymbol{P}_2$, $\boldsymbol{P}_*$ и $\boldsymbol{P}_*$, $\boldsymbol{P}_3$, $\boldsymbol{P}_4$, $\boldsymbol{P}_5$.

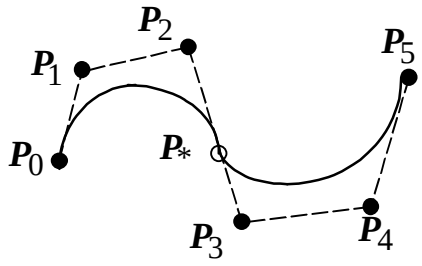


Рис. 7.4. Построение составной кривой Безье

Кубические кривые Безье довольно популярны в компьютерной графике. Например, в графическом редакторе Corel Draw кривые Безье являются основой для создания всех линий. Функция построения кривой Безье, также

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 0. Это связанно с тем, что для построения элементарной Безье кривой требуется четыре точки, а при построении составной кривой точки в местах стыковки совпадают (поэтому делим на три), конечная же точка последнего сегмента не дублируется (поэтому nCount-1).

На рис. 7.5 приведено изображение окна программы, в которой реализовано построение с помощью функции PolyBezier по набору из 10 точек генерируемых случайным образом. Для построения составной кривой по 10 базовым точкам использовано три сегмента. Хорошо видно, что в функции PolyBezier не обеспечивается геометрическая непрерывность составной сплайновой кривой.

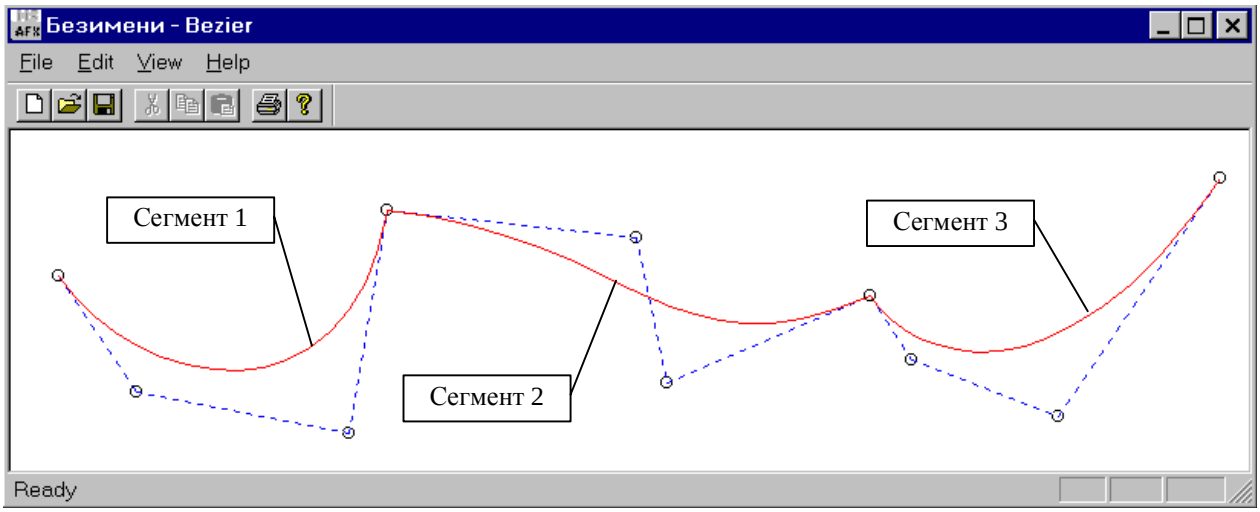


Рис. 7.5. Построение кривой функцией PolyBezier

Рис. 7.6 иллюстрирует возможность построения геометрически непрерывной составной кривой Безье при выполнения условия расположения трех точек в месте стыковки сегментов на одной прямой. Данный рисунок получен с помощью той же программы, но точки в месте стыковки второго и третьего сегментов оказались почти на одной прямой. Видно, что стыковки сегментов почти незаметно.

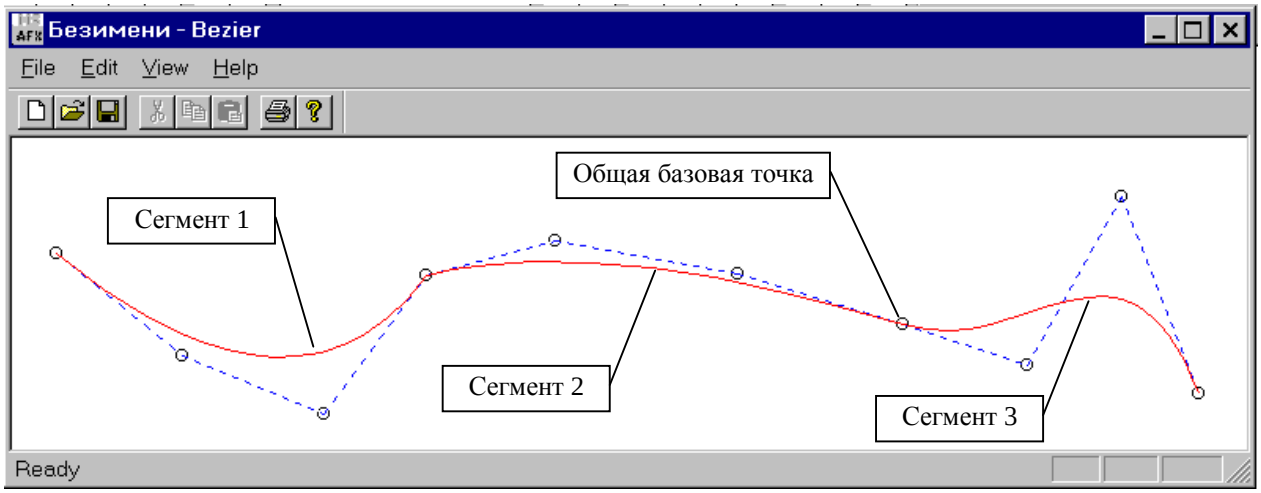


Рис. 7.6. Построение кривой функцией PolyBezier

Функция, с помощью которой были построены кривые на рис. 7.5 и 7.6 приведена ниже. Полностью текст программы приведен в архиве Bezier.rar.

```
BOOL CBezierView::DrawBezier(CDC* pDC, POINT* lpPoints, int nCount)
{
      //проверка кратности числа точек
      div_t div_result;
      div_result=div(nCount-1, 3);
      if(div_result.quot<1 || div_result.rem>0) return FALSE;

      //рисуем точки
      for(int i=0; i<nCount; i++)
            pDC->Ellipse(   lpPoints[i].x-4, lpPoints[i].y-4,
                            lpPoints[i].x+4, lpPoints[i].y+4);
      //цветные перья для рисования
      CPen PenBlue; PenBlue.CreatePen(PS_DOT, 1, RGB(0,0,255));
      CPen PenRed; PenRed.CreatePen(PS_SOLID, 1, RGB(255,0,0));
      //выберем синее перо и запомним предыдущее
      CPen *pPenOld=pDC->SelectObject(&PenBlue);
      //соеденим прямыми базовые точки
      pDC->Polyline(lpPoints, nCount);
      //красным пером нарисуем кривую Безье
      pDC->SelectObject(&PenRed);
      BOOL res=pDC->PolyBezier(lpPoints, nCount);
      //восстановим старое перо
      pDC->SelectObject(pPenOld);
      return res;
};
```

Элементарная Бета-сплайновая кривая

По заданному массиву точек P_0, P_1, P_2, P_3 Бета сплайновая кривая описывается уравнением

$$\boldsymbol{R}(t) = b_0(t)P_0 + b_1(t)P_1 + b_2(t)P_2 + b_3(t)P_3, \qquad (7.5)$$
$$0 \le t \le 1,$$

функциональные коэффициенты $b_i(t)$ задаются следующими формулами:

$$b_0(t) = \frac{2\beta_1^3}{\delta}(1-t)^3,$$

$$b_1(t) = \frac{1}{\delta}\left(2\beta_1^3 t\left(t^2-3t+3\right)+2\beta_1^2\left(t^3-3t+2\right)+2\beta_1\left(t^3-3t+2\right)+\beta_2\left(2t^3-3t^2+1\right)\right),$$

$$b_2(t) = \frac{1}{\delta}\left(2\beta_1^2 t^2(-t+3)+2\beta_1 t\left(-t^2+3\right)+2\beta_2 t^2(-2t+3)+2\left(-t^3+1\right)\right),$$

$$b_3(t) = \frac{2t^3}{\delta},$$

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, при этом 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.

Реализуем в программе Painter класс "сплайновая кривая". Такой класс можно породить от класса CPoligon, но в классе CPoligon массив точек имеет фиксированный размер, а это не очень удобно (в сплайновой кривой может быть много точек и размер массива может оказаться недостаточен). Поэтому создадим новый класс, производный прямо от CBasePoint. Назовем его CSpline. В классе CSpline определим указатели на массивы координат базовых точек. Значения каждой координаты хранятся в отдельных массивах m_pBasePointsX и m_pBasePointsY соответственно. Сплайновые точки будем хранить в массиве m_pSlinePoints, который будем отображать на экране. В классе CSpline можно реализовать сплайновые функции различных видов. В приведенном ниже описании класса реализована лишь функция построения Catmul-Rom сплайна. Аналогично можно реализовать сплайновые функции других видов. На базе класса CSpline можно реализовать классы различных фигур. Например, ниже приведен класс CSplineStar задающий четырехконечную звезду. Работа программы с реализованной фигурой сплайновая звезда показана на рис. 7.5. Для того, чтобы фигуры классов CSpline и CSplineStar были такими же «способными» как и остальные в проекте Painter, в данных классах надо еще реализовать функции сохранения и трансформации. Текст программы приведен в архиве Painter6.rar.

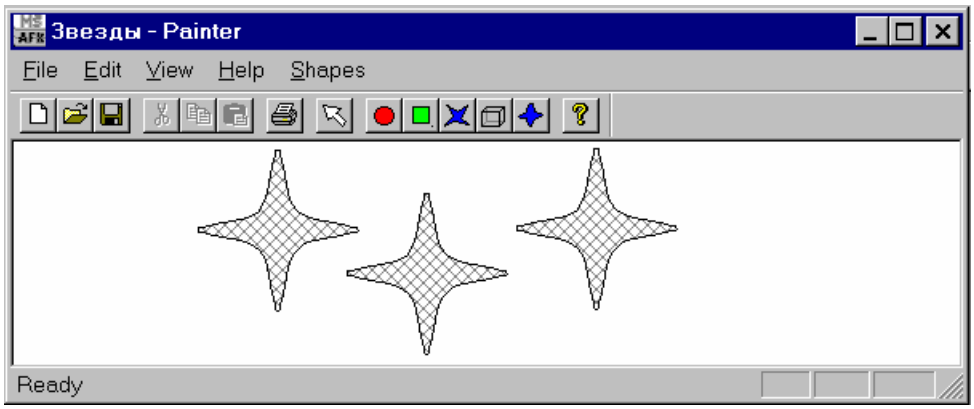


Рис. 7.7. Программа в работе

```
//файл Shapes.h
////////////////////////////////////////////////////
//класс плоский сплайн

class CSpline: public CBasePoint
{
      double Catmull_Rom(double p[4], double t);
 public:
      //конструкторы
      CSpline();
      ~CSpline();
//Данные
      double *m_pBasePointsX;   // указатель на массив базовых точек X
      double *m_pBasePointsY;   // указатель на массив базовых точек X
      POINT   *m_pSlinePoints;  // массив точек
                                // для отображения сплайна на экране
      WORD m_nBaseIndex;        // число базовых точек
      WORD m_nSplineIndex;      // число сплайновых точек
      WORD m_nM;                // параметр дискретизации сегмента
//Методы
      // Очистка памяти
      void Clear();
      // Создать Catmul-Rom сплайн
      void CreateCatmulRom(double* pX, double* pY, WORD index,
WORD m);
      // Отображение фигуры на экране
      virtual void Show(CDC *pDC);
};

////////////////////////////////////////////////////
//класс сплайновая звезда
class CSplineStar: public CSpline
{
 public:
       WORD m_Size;
      //конструкторы
      CSplineStar(CPoint point, WORD size);
      ~CSplineStar(){};
//Данные
};
```

Реализация функций классов CSpline и CSplineStar в файле Shapes.cpp.

```
// файл Shapes.cpp
////////////////////////////////////////////////
//класс плоский сплайн
CSpline::CSpline()
{
     m_pBasePointsX=NULL;
     m_pBasePointsY=NULL;
     m_pSlinePoints=NULL;
     m_nBaseIndex=0;
     m_nSplineIndex=0;
     m_nM=0;
};

CSpline::~CSpline()
{
     Clear();
};

void CSpline::Clear()
{
     if(m_pBasePointsX!=NULL)
          {delete[] m_pBasePointsX; m_pBasePointsX=NULL;}
     if(m_pBasePointsY!=NULL)
          {delete[] m_pBasePointsY; m_pBasePointsY=NULL;}

     if(m_pSlinePoints!=NULL)
          {delete[] m_pSlinePoints; m_pSlinePoints=NULL;}

     m_nBaseIndex=0;
     m_nSplineIndex=0;
     m_nM=0;
};

double CSpline::Catmull_Rom(double p[4], double t)
{
   double s=1.0-t, t2=t*t, t3=t2*t;
   return 0.5*(-t*s*s*p[0]+(2-5*t2+3*t3)*p[1]+
          t*(1+4*t-3*t2)*p[2]-t2*s*p[3]);
}

void CSpline::CreateCatmulRom(double* pX, double* pY, WORD index,
 WORD m)
{
   Clear(); //очистим на случай если уже был сплайн
   m_nBaseIndex=index;
   m_nSplineIndex=(index-3)*m+1;
   m_nM=m;
   //выделим память
   m_pBasePointsX=new double[m_nBaseIndex];
   m_pBasePointsY=new double[m_nBaseIndex];
   //запомним во внутренних переменных
   CopyMemory (m_pBasePointsX, pX, index*sizeof(double));
   CopyMemory (m_pBasePointsY, pY, index*sizeof(double));
```

```
   m_pSlinePoints=new POINT[m_nSplineIndex];
   //построим сплайн
   double  t, dt=1.0/m;
   int d, first=0;
   for(int i=1; i<m_nBaseIndex-2; i++)
   {
      t=0;
      for(d=0; d<=m; d++)
      {
           m_pSlinePoints[first+d].x=Catmull_Rom(&m_pBasePointsX[i-1],
t);
           m_pSlinePoints[first+d].y=Catmull_Rom(&m_pBasePointsY[i-1],
t);
           t+=dt;
       }
      first+=m;
   }
};

void CSpline::Show(CDC* pDC)
{
     CBrush Brush(HS_DIAGCROSS, m_Color);
     CBrush     *pOldBrush=pDC->SelectObject(&Brush);

     pDC->Polygon( m_pSlinePoints, m_nSplineIndex);

     pDC->SelectObject(pOldBrush);
}


CSplineStar::CSplineStar(CPoint point, WORD size)
{
m_Size=size;
double X[11]={  point.x,       point.x+m_Size/4,       point.x+m_Size,
point.x+m_Size/4, point.x, point.x-m_Size/4, point.x-m_Size, point.x-
m_Size/4, point.x, point.x+m_Size/4, point.x+m_Size};

double Y[11]={  point.y-m_Size,       point.y-m_Size/4,       point.y,
point.y+m_Size/4, point.y+m_Size, point.y+m_Size/4, point.y, point.y-
m_Size/4, point.y-m_Size, point.y-m_Size/4, point.y};

CreateCatmulRom(X, Y, 11, 10);
};
```

Смотри также рекомендуемую литературу [1, 2, 8, 11, 12].

8. ФОРМАТЫ ГРАФИЧЕСКИХ ФАЙЛОВ. ЦВЕТОВЫЕ МОДЕЛИ. ПАЛИТРЫ ЦВЕТОВ. МЕТОДЫ СЖАТИЯ

Графический формат – это порядок (структура), в которой 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 др. параметры линий) и набор правил (соглашений) вывода (рисования). Пример векторного рисунка приведен на рис. 8.1.

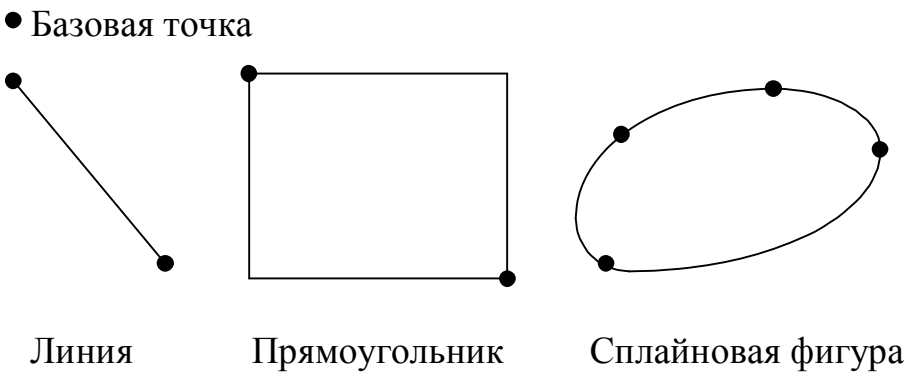


Рис. 8.1. Векторный рисунок

Растровые данные представляют собой набор числовых значений, определяющих цвета отдельных ***пикселей***. Пиксели – цветные точки, расположенные на правильной сетке и формирующие образ. Термин «растр» в компьютерной графике и полиграфии имеет несколько отличающихся значений. В нашем случае растр представляет собой массив пикселов (массив числовых значений. Для обозначения массива пикселов часто используется термин ***bitmap*** (битовая карта). В *bitmap* каждому пикселу отводиться определенное число (одинаковое для всех пикселов изображения) бит. Это число называется ***битовой глубиной*** пиксела или ***цветовой глубиной*** изображения, так как от количества бит, отводимых на один пиксел, напрямую зависит количество цветов изображения. Наиболее часто используется пиксельная глубина 1, 2, 4, 8, 15, 16, 24 и 32 бита.

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

Пример растрового рисунка приведен на рис. 8.2.

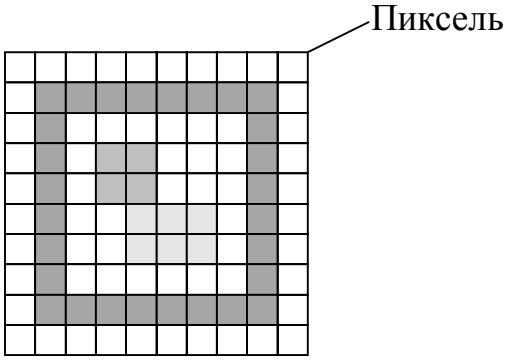


Рис. 8.2. Растровый рисунок

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

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

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, Microsoft SYLK.

Метафайловый формат позволяет хранить в одном файле и векторные и растровые данные. Примером такого формата являются файлы Corel Draw 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.

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 от 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, так как их положение в файле определяется в процессе чтения.

Как правило, в графических файлах применяются комбинации элементов данных.

Преобразование форматов. Часто возникает проблема преобразования формата данных, связанная с необходимостью обмена изображениями между различными программами. Для преобразования данных существую специализированные программы, кроме того, распространенные графические редакторы позволяют сохранять изображения в различных форматах. Поэтому преобразование растровых данных в одном формате в растровые данные в другом формате обычно не представляет сложности. Другое дело – преобразование векторных файлов в растровые. При таком преобразовании неизбежно теряется часть информации, так как происходит переход от «идеального» математического описания рисунка к его дискретному (растровому) представлению. Обратное же преобразование из растрового в векторный формат вообще представляет из себя совсем нетривиальную задачу, связанную с задачей распознавания образов.

Сжатие данных. Сжатие – процесс уменьшения физического размера блока данных. Так как для хранения изображений обычно требуется очень большое количество памяти, графические данные часто подвергаются сжатию. Обычно формат графического файла поддерживает какой-либо из методов (алгоритмов) сжатия. 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 же сжатие – напротив основано на логическом анализе информации. Примером логиче-

ского сжатия может служить преобразование строки «Томский государственный университет систем управления и радиоэлектроники» в аббревиатуру «ТУСУР». Логическое сжатие не применяется для данных изображения.

Методы сжатия бывают ***с потерями*** и ***без потерь*.** Когда данные сжимаются, а после разжимаются (т.е. распаковываются) и восстановленные данные полностью соответствуют исходной информации, то говорят, что имело место сжатие без потерь. Иначе говоря, при методе сжатия без потерь не должно происходить какого-либо изменения данных.

Методы сжатия с потерями предусматривают отбрасывание некоторых 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 бит 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, – целый набор методов сжатия, используемый в основном для обработки изображений с плавным переходом тона. Базовая реализация JPEG применяет схему кодирования по алгоритму дискретных косинус - преобразований.
- 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.) остаются всегда несжатыми.

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

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

Пиксели и цвет. Различают физические и логические пиксели.

Физические пиксели – реальные точки, отображаемые на устройстве вывода – наименьшие элементы на поверхности отображения, которыми

можно манипулировать. На практике один физический пиксель обычно формируется из нескольких более мелких цветовых точек.

Логические пиксели подобны чертям на кончике иглы – они имею местоположение и цвет, но не занимают физического пространства. 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).

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

Количество цветов пиксела напрямую связано с отводимым для него количеством бит и равно 2^n, где n- количество бит. Изображения, каждому пикселу которых отводиться один бит, называют монохромными – двуцветными. Такие изображения вполне подходят для изображения чертежей и текста. Изображения с глубиной цвета 24 бит называют ***truecolor*** (истинные цвета). Каждый пиксел такого изображения может принимать более 16 миллионов цветов. Считается, что этого вполне достаточно, чтобы правильно 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 млн. 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 так называемое квантование – преобразование цветов изображения к 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).

Палитры цветов. Палитра цветов, называемая также ***карой*** **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, данные о цветах пикселов в файле записаны в виде последовательности индексов. Использование палитр во многих случаях позволяет значительно сократить объем растровых данных.

Например, изображения с глубиной цвета в 4 бит могут иметь 16 цветов. Эти 16 цветов определены в палитре, которая обычно включаются в один файл с растровыми данными. Каждое пиксельное 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 с лишним миллионов цветов. Программа, осуществляющая визуализацию изображения, читает из файла растровые данные – индексы в таблице цветов и использует соответствующие им цвета для окрашивания пикселов экрана (рис. 8.3).

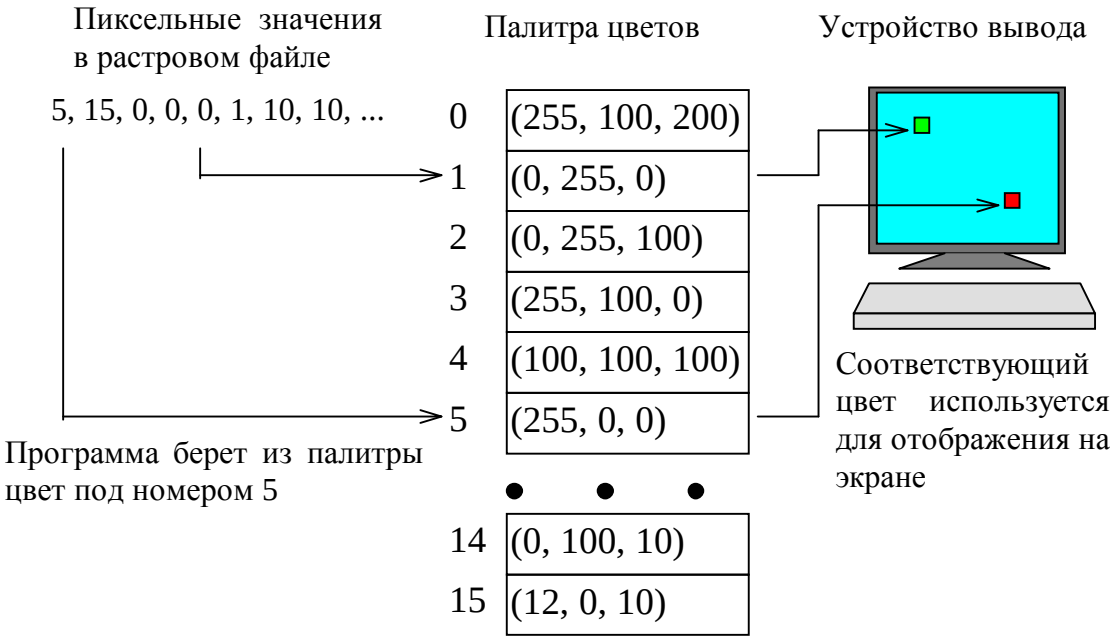


Рис. 8.4. Определение цвета с помощью палитры

Цвет. Цветовые модели. Для описания цветов применяют несколько различных математических систем (моделей). Ни одна из существующих систем представления цвета не является наилучшей. В зависимости от обстоятельств применяют различные системы.

В графических файлах обычно применяют 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 получаются путем сложения основных цветов различной интенсивности с черным. Чем больше интенсивность добавляемых цветов, тем ближе результирующий цвет к белому. 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 формирование цветных изображений на отражающих носителях, например, бумаге.

Рассмотрим наиболее распространенные цветовые модели.

Модель RGB (Red-Green-Blue – красный-зеленый-синий) – модель наиболее широко используемая в графических форматах. RGB – аддитивная модель. Каждый пиксель представляется в виде трех числовых величин – трех интенсивностей красного, зеленого и синего цветов. Каждому цвету обычно отводиться 8 бит, 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, 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, обозначаемый буквой «K». В результате получается модель CMYK, широко распространенная в полиграфии. При использовании субтрактивной модели изображение каждого пикселя (цветной точки) изображения состоит из четырех пятен основных цветов.

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

Использование классов MFC для вывода на экран растровых изображений.

Растровые изображения в MFC описываются классом CBitmap.

Класс CBitmap позволяет выполнять операции загрузки и вывода изображений, являющихся ресурсами приложения, т.е. растровые изображения должны быть заранее определены. Возможно, это связано с неспособностью класса CBitmap динамически управлять графическими данными. Для загрузки и вывода на экран растровых изображений из произвольных файлов нам потребуется написать свой класс, но этому будет посвящена следующая лекция.

Рассмотрим последовательность действий, необходимых для загрузки и отображения растровых изображений с помощью класса CBitmap.

1. Создать Bitmap-ресурс. Для этого в окне ресурсов проекта щелкнем правой клавишей мыши на папке Bitmap и выберем пункт «New Bitmap» (рис. 8.5). (Окно ресурсов проекта открывается двойным щелчком левой клавиши мыши на имени файла с расширением «.rc» в окне проекта). В результате будет создан новый ресурс с идентификатором IDB_BITMAP1 (см. рис. 8.6), запущен встроенный графический редактор среды Visual C++ и появиться панель инструментов. С помощью инструментов данного редактора можно создать растровое изображение (рис. 8.6). Кроме того, в ресурсы проекта можно импортировать уже существующее растровое изображение. Для этого нужно выполнить команду «Import» из меню Resource и указать имя BMP файла. В ресурсы проекта могут быть включены лишь изображения с палитрами из 2, 16 и 256 цветов.

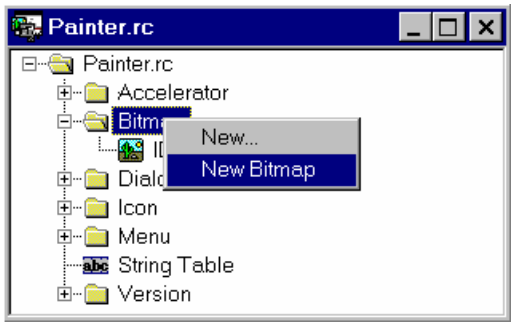


Рис. 8.5. Добавление Bitmap-ресурса

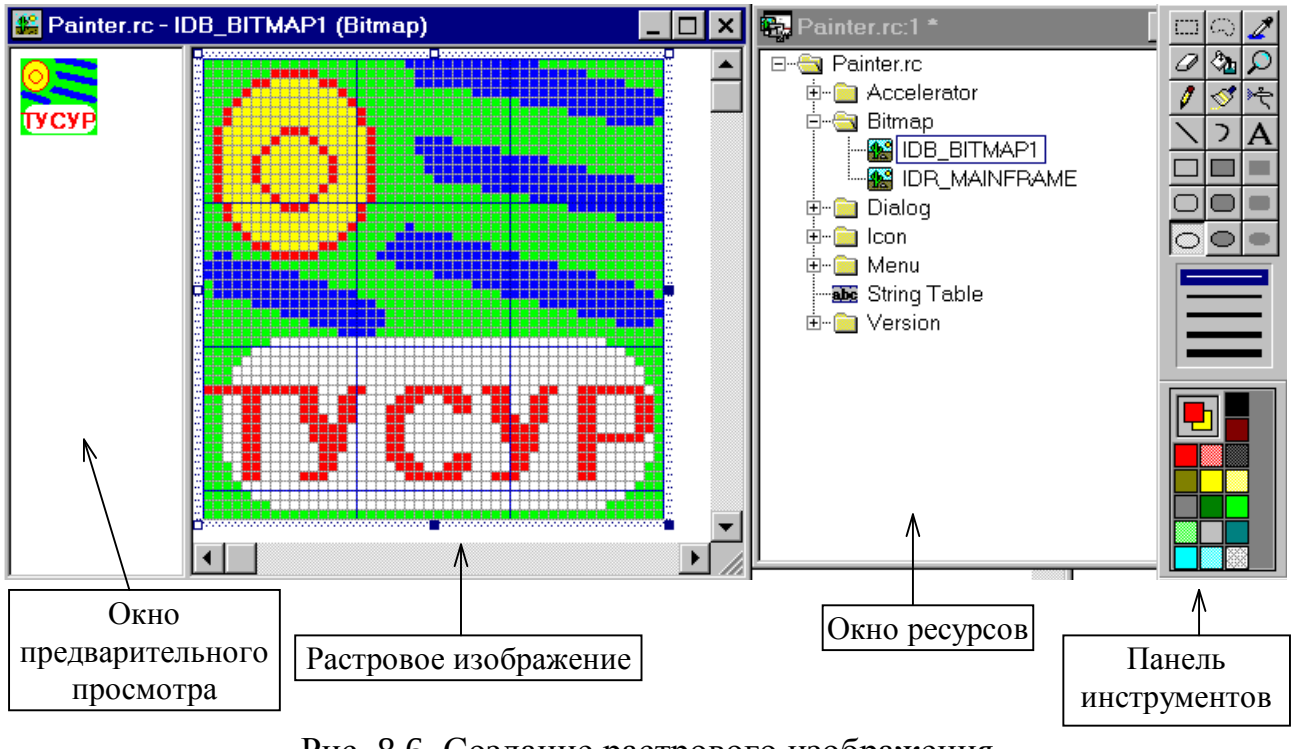


Рис. 8.6. Создание растрового изображения

Палитру цветов растрового изображения можно просмотреть в окне свойств рисунка (закладка «Palette») – см. рис. 8.7. Открыть окно свойств можно, дважды щелкнув левой клавишей мыши на свободном пространстве окна редактирования рисунка.

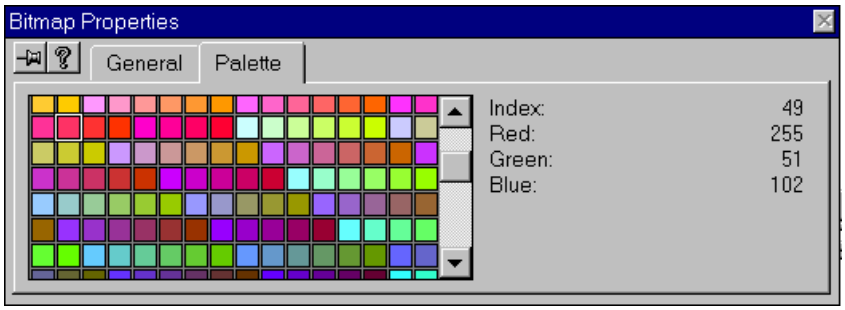


Рис. 8.7. Палитра рисунка

2. Создать в программе объект класса CBitmap и вызвать функцию CBitmap::LoadBitmap(IDB_BITMAP1), в качестве параметра указывать идентификатор растрового ресурса.

3. Получить контекст устройства, на которое выводится изображение. Для этого можно создать объект класса CClientDC, в конструктор которого передать указатель на окно, осуществляющее вывод: CClientDC DC(this). Кроме того, можно использовать указатель на контекст устройства, передаваемый в функцию OnDraw(CDC *pDC) класса-облика.

4. Создать совместимый с устройством вывода контекст области памяти, в которой храниться изображение.

5. Связать контекст памяти с изображением, загруженным в объект класса CBitmap.

6. Скопировать изображение из памяти, на которую указывает контекст памяти в контекст устройства вывода.

Для работы с растровыми изображениями дополним иерархию классов проекта Painter классом CRastPic (растровая картинка), который будет отвечать за загрузку и вывод растровых изображений.

Описание интерфейса класса и реализации его функций приведено ниже.

```
//Файл Shapes.h
//////////////////////////////////////////////////
//интерфейс класса CRastPic - растровый рисунок
class CRastPic: public CBasePoint
{
      CBitmap m_BMP;  //растровое изображение
      BITMAP     m_bmStruct; //структура данных, описывающая растровое
изображение
      DECLARE_SERIAL(CRastPic)
  protected:    // Виртуальный метод сериализации
```

```
      virtual void Serialize(CArchive& ar);

  public:
      //конструкторы
      CRastPic();
      CRastPic(CPoint point);
      ~CRastPic(){};
    // Отображение фигуры на экране
    virtual void   Show(CDC *pDC);
};


//Файл Shapes.cpp
////////////////////////////////////
// реализация класса CRastPic – растровый рисунок
IMPLEMENT_SERIAL(CRastPic, CBasePoint, -1)
void CRastPic::Serialize(CArchive &ar)
{
      CBasePoint::Serialize(ar);
};

CRastPic::CRastPic(): CBasePoint()
{
      m_BMP.LoadBitmap(IDB_BITMAP1); //загружаем растровый ресурс
      // получаем описание
      m_BMP.GetObject(sizeof(BITMAP), &m_bmStruct);
};

CRastPic::CRastPic(CPoint point): CBasePoint(point)
{
      m_BMP.LoadBitmap(IDB_BITMAP1);//загружаем растровый ресурс
      m_BMP.GetObject(sizeof(BITMAP),  &m_bmStruct);//  получаем  описа-
ние
}


void CRastPic::Show(CDC* pDC)
{
      CDC MemDC; //контекст памяти
      //создание совместимого контекста памяти
      MemDC.CreateCompatibleDC(pDC);
      //связь контекста памяти с изображением
      MemDC.SelectObject(&m_BMP);
      //копирование изображения в контекст окна
      pDC->BitBlt(m_x,  m_y,  m_bmStruct.bmWidth,  m_bmStruct.bmHeight,
&MemDC, 0,0, SRCCOPY);
}
```

Для вывода растрового рисунка на экран используется функция BitBlt() класса CDC. Аргументы функции задают позицию начала вывода, ширину и высоту области назначения (куда осуществляется вывод), указатель на контекст памяти с растровым изображением, позицию с которой считывается растровое изображение (обычно 0,0, но может иметь и другие значения), растровую операцию выполняемую над копируемым изображением и изображением области назначения.

Надо отметить, что растровых изображений в ресурсах проекта может быть более одного, размеры и количество цветов у них могут быть разными.

Растровые ресурсы могут быть использованы и для создания кистей, которые задают заливку замкнутых фигур. Пожалуй, это самый простой способ создания текстуры.

Для создания кисти на основе растрового изображения необходимо выполнить всего лишь пару действий:

1. Создать растровый ресурс размером 8x8 пикселей (назовем его IDB_BITMAP2).

2. Создать на основе этого ресурса кисть с помощью функции CreatePatternBrush() класса CBrush.

Например, добавим в класс CPoligon переменную CBitmap m_BrushBmp, а в конструкторе будем загружать в нее растровое изображение, как это показано ниже.

```
//Файл Shape.cpp
//конструктор класса CPoligon
CPoligon::CPoligon(): CBasePoint()
{
      m_nIndex=0;
      SetColor(RGB(0,0,200));
      m_BrushBmp.LoadBitmap(IDB_BITMAP2);
};

CPoligon::CPoligon(CPoint point): CBasePoint(point)
{
      m_nIndex=0;
      SetColor(RGB(0,0,200));
      m_BrushBmp.LoadBitmap(IDB_BITMAP2);
}
```

Внесем также изменения и в функцию отображения полигона.

```
void CPoligon::Show(CDC* pDC)
{
      CBrush Brush, *pOldBrush=NULL;
//создание кисти на основе шаблона - растрового изображения
      if(Brush.CreatePatternBrush(&m_BrushBmp))
      //выбор кисти в контексте окна
            pOldBrush=pDC->SelectObject(&Brush);
      pDC->Polygon( m_Points, m_nIndex);

      if(pOldBrush!=NULL) pDC->SelectObject(pOldBrush);
}
```

В качестве примера создадим шаблон кисти в виде фрагмента стены из красного кирпича (Рис. 8.8).

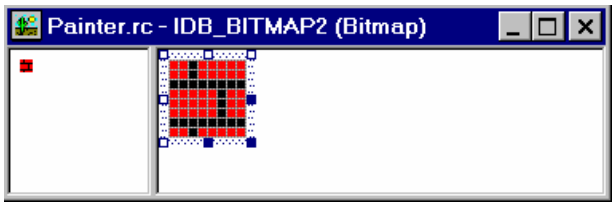


Рис. 8.8. Шаблон кисти

Работа программы реализующей функции работы с растровыми изображениями показана на рис. 8.9.

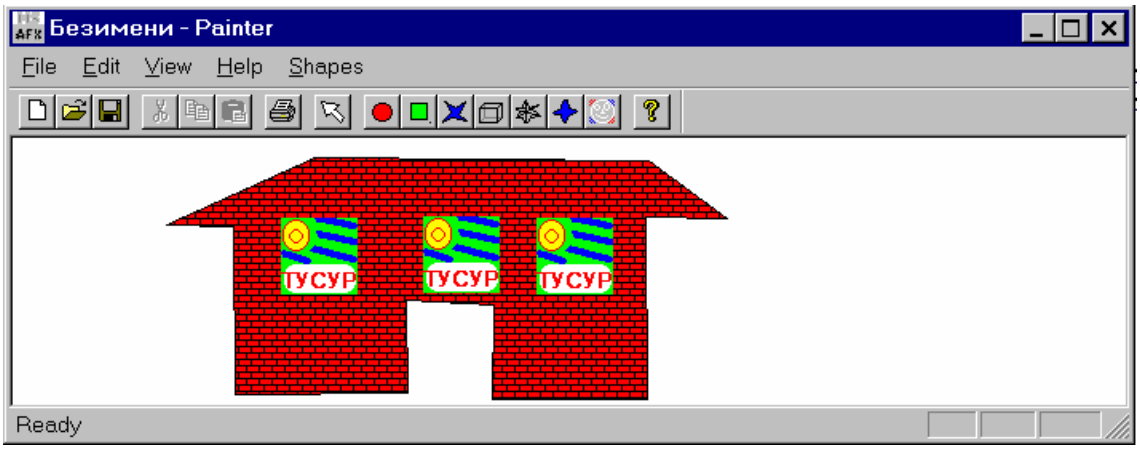


Рис. 8.9. Программа в работе

Смотри также рекомендуемую литературу [3, 4, 10].

9. ФОРМАТ MICROSOFT WINDOWS BITMAP. ЗАГРУЗКА ФАЙЛОВ В ФОРМАТЕ BMP И ВЫВОД РАСТРОВЫХ ИЗОБРАЖЕНИЙ НА ЭКРАН

Общее описание

Microsoft Windows Bitmap (BMP) собственный растровый формат ОС Windows. Практически все приложения, предназначенные для работы с изображениями в Windows, 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 ОС Windows. Первоначально формат был очень простым. Он 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 др.).

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 ОС OS/2. (Возможно, при создании систем Windows-98 и Windows-2000 в формат были внесены новые изменения.) В каждой новой версии в заголовке растра появляется новая информация. Однако грамотно написанные программы, осуществляющие чтение/отображение изображений в новом формате, способны работать и со старыми форматами.

Структура файла

Файлы DDB исходного формата BMP содержали два раздела заголовок файла и растровые данные.

Заголовок файла
Растровые данные

Файлы более поздних версий содержат четыре раздела: заголовок файла, информационный заголовок растра, палитру цветов и растровые данные.

Заголовок файла
Заголовок растра
Цветовая палитра
Растровые данные

Рассмотрим в деталях структуру данных файла формата BMP версии 3.x, так как он поддерживается большинством существующих в настоящее время приложений.

Все версии формата BMP начинаются с 14-байтового заголовка:

```
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 0. Размер файла в этом случае можно узнать у операционной системы.

В полях bfReserved1 и bfReserved2 не содержат данных и обычно устанавливаются 0. 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. Его длина составляет 40 байт.

```
typedef struct tagBITMAPINFOHEADER{

    DWORD   biSize;             /* Размер этого заголовка в байтах */
    LONG    biWidth;            /* Ширина изображения в пикселях */
    LONG    biHeight;           /* Высота изображения в пикселях */
    WORD    biPlanes;           /* Количество цветовых плоскостей */
    WORD    biBitCount          /* Количество битов на пиксель */
    DWORD   biCompression;      /* Используемые методы сжатия */
    DWORD   biSizeImage;        /* Размер растра в байтах */
```

```
    LONG    biXPelsPerMeter;    /* Горизонтальное разрешение */
                                /* в пикселях на метр */
    LONG    biYPelsPerMeter;    /* Вертикальное разрешение */
                                /* в пикселях на метр */
    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 пикселя. Если biHeight – положительное число, то изображение представляет собой растр с началом в левом нижнем углу. Если biHeight – положительное, то начало растра в левом верхнем углу.

Поле biPlanes – количество цветовых плоскостей – в BMP-файлах одна цветовая плоскость, поэтому значением этого поля всегда является 1.

В поле biBitCount указывается количество бит, отводимых под один пиксель. 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 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 0. 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 пикселях 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=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 0. 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. Оно использует только начало палитры. Обычно это поле равно 0.

За заголовком растра может следовать палитра цветов. Палитра цветов состоит из последовательности 4-байтовых структур:

```
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.

Загрузка BMP файлов

Рассмотрим, как на практике происходит загрузка и отображение файлов BMP. Для этого создадим проект ShowBM, который будет загружать и отображать файлы BMP на экране в разном масштабе. Кроме того, на примере проекта ShowBM рассмотрим создание и работу приложений с многодокументным (MDI) интерфейсом.

Для создания проекта ShowBM воспользуемся генератором приложений (AppWizard) среды Microsoft Visual C++ 2.2. Выберем команду New меню File и далее по порядку, описанному в главе 1. Новые свойства данный проект приобретет благодаря следующим трем отличиям.

1. На первом шаге создания проекта укажем, что мы создаем MDI проект (это установка по умолчанию) – см. рис. 9.1.

2. На четвертом шаге установим поддержку контекстной помощи (рис. 9.2).

3. На шестом шаге заменим базовый класс облика с CView на CScrollView (рис. 9.3). Это позволит «прокручивать» изображение в окне, если оно там целиком не помещается.

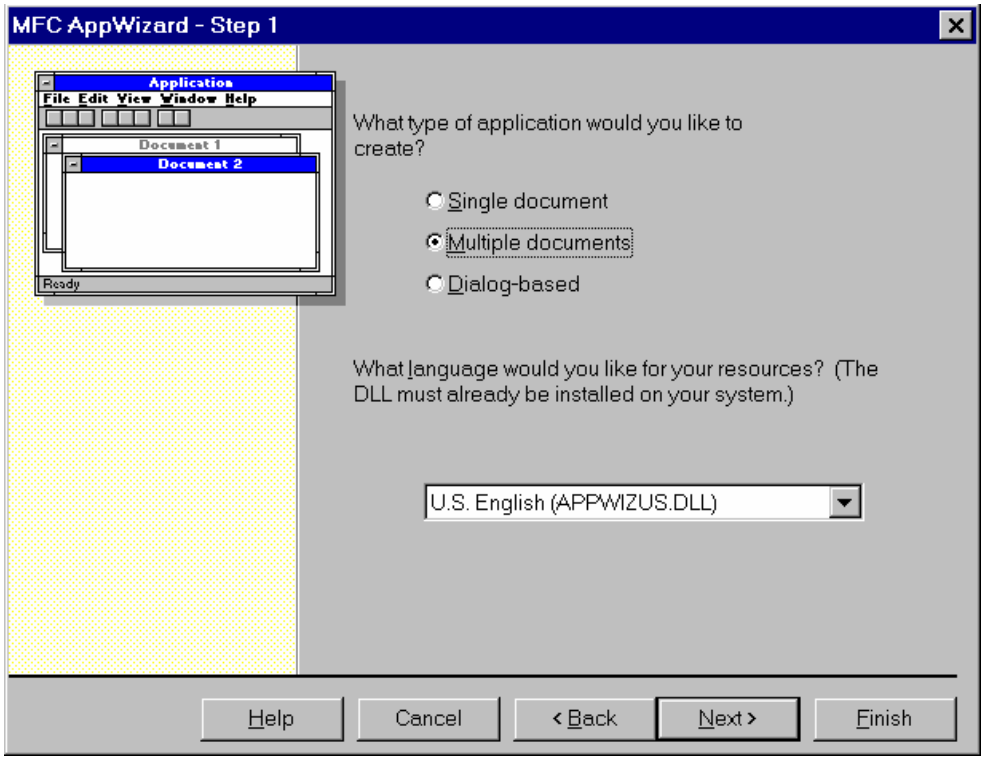


Рис. 9.1. Выбор MDI - интерфейса

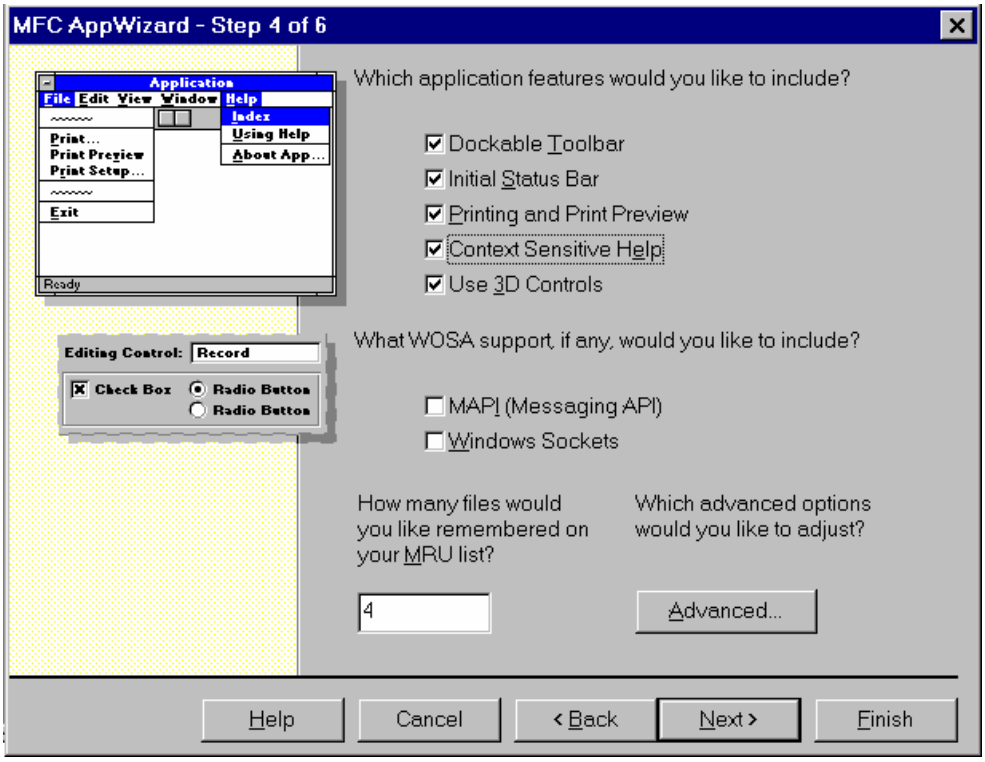


Рис. 9.2. Включение контекстной помощи

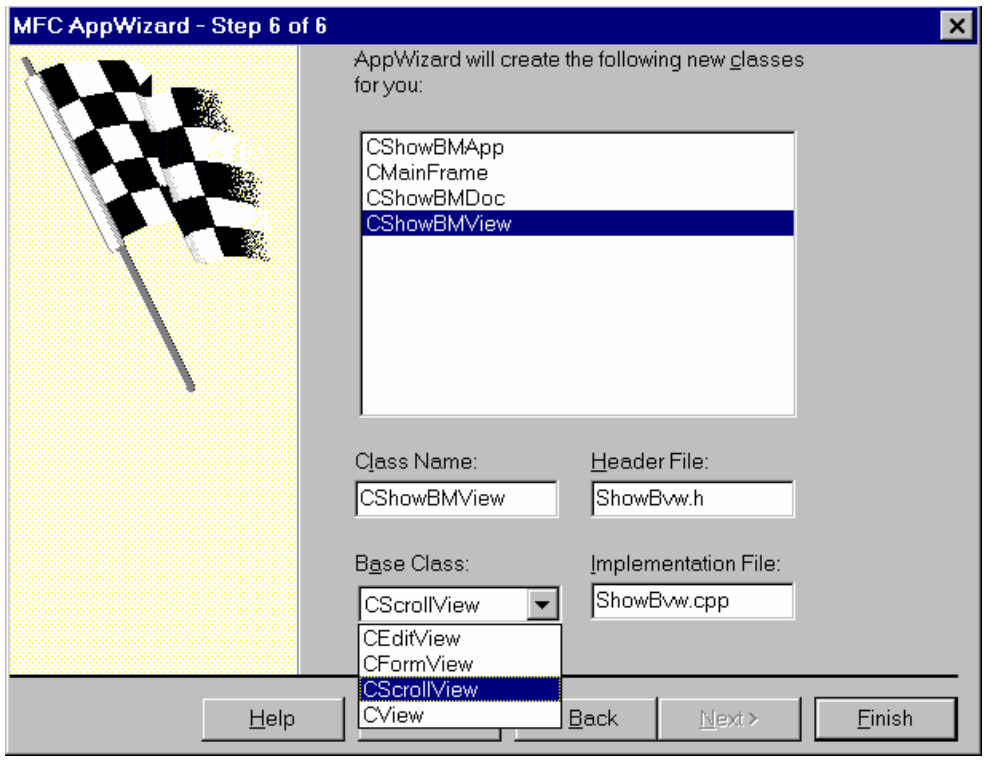


Рис. 9.3. Замена базового класса облика

Для загрузки и вывода на экран графических файлов создадим класс CRaster, который будет реализовывать все эти функции.

Ниже приведен интерфейс класса CRaster (файл raster.h).

```
// CRaster.h : interface of the CRaster class
//
/////////////////////////////////////////////////////////////////////////////
class CRaster
{
  PVOID               m_pFile;      //указатель на данные файла
  LPBITMAPFILEHEADER  m_pFI;        //описание файла
  LPBITMAPINFO        m_pBMI;       //описание изображения
  PBYTE               *m_pStartDIB; //указатель на начало байтов
                                    //изображения
public:
  CRaster();
  ~CRaster();
  void Clear(); //очистка памяти
  //возвращает указатель на описание файла
  LPBITMAPFILEHEADER GetBMFileHdrPtr(){return m_pFI;}
//указатель на заголовок растра
  LPBITMAPINFO GetBMInfoPtr(){return m_pBMI;}
//указатель на начало файла
  PVOID    GetFilePtr(){return m_pFile;};
//указатель на растровые данные
  PBYTE    GetBMBitsPtr(){return m_pStartDIB;};
//указатель на таблицу цветов
  RGBQUAD* GetBMColorTabPtr();
```

```
//возвращает
  LONG GetBMWidth(); //ширину в пикселях;
  LONG GetBMHeight();//высоту в пикселях;

//загружает из файла
  BOOL LoadBMP(CString FileName);
//выводит DIB на контекст pDC с позиции x,y, размером cx на cy
  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, DWORD rop=SRCCOPY );
};
```

Рассмотрим назначение данных и методов класса. Переменная m_pFile после открытия файла будет хранить указатель на начало данных. В переменных m_pFI и m_pBMI сохраним указатели на структуры BITMAPFILEHEADER и BITMAPINFO. В переменной m_pStartDIB будем хранить указатель на начало растровых данных.

У класса предусмотрены конструктор и деструктор. В конструкторе выполняется инициализация переменных, а в деструкторе очистка памяти. Класс содержит ряд функций, предназначенных для доступа к его данным. Метод LoadBMP() предназначен для загрузки растрового изображения в формате BMP из файла, имя которого передается ему в качестве аргумента. Вывод изображения на экран осуществляется функцией DrawBitmap(). Аргументами данной функции являются указатель на контекст экрана, координаты отображаемой области растровых данных и координаты области назначения на экране.

Рассмотрим реализацию функций класса. Ниже приведена функция LoadBMP() из файла raster.cpp.

```
BOOL CRaster::LoadBMP(CString FileName)
{
      //Очистим
      Clear();
      //Из файла собираемся только читать
      DWORD      FileAccess=GENERIC_READ,
                 FileOpenFlag=FILE_SHARE_READ,
                 PageFlag=PAGE_READONLY,
                 MappingFlag=FILE_MAP_READ;

      //Открываем файл
      HANDLE hFile=NULL;//указатель на файл
      hFile=CreateFile(FileName, FileAccess, FileOpenFlag, NULL,
                       OPEN_EXISTING, FILE_FLAG_RANDOM_ACCESS, NULL);
      if(hFile==INVALID_HANDLE_VALUE) return FALSE;

      //размер файла
      DWORD FileSize=GetFileSize(hFile, NULL);
      //создаем "проецируемый файл"
      //указатель на объект-файл проецируемый в память
      HANDLE hFileMapping=NULL;
      hFileMapping=CreateFileMapping(hFile, NULL, PageFlag, 0, FileSize,
NULL);
```

```
        //hFile независимо от исхода CreateFileMapping уже не нужен
        CloseHandle(hFile);
        if(hFileMapping==NULL)     return FALSE;
        m_pFile=MapViewOfFile(hFileMapping, MappingFlag,0,0,0);
        //hFileMapping независимо от исхода MapViewOfFile уже не нужен
        CloseHandle(hFileMapping);
        if(m_pFile==NULL) return FALSE;

        ////////////////////////////////////////////////////
        //читаем заголовок файла.
        //Это дает его размер и положение начала данных
        DWORD FilePos=0; //указатель позиции в файле
        m_pFI=(LPBITMAPFILEHEADER)m_pFile;

        //Проверяем действительно ли Windows Bitmap изображение
        if(m_pFI->bfType!=0x4D42)  { Clear(); return FALSE;}

        FilePos+=sizeof(BITMAPFILEHEADER);
        //читаем BITMAPINFO
        m_pBMI=(LPBITMAPINFO)(((PBYTE)m_pFile)+FilePos);
        //переход к началу данных
        m_pStartDIB=((PBYTE)m_pFile)+m_pFI->bfOffBits;
        if(m_pStartDIB==NULL) return FALSE;
        return TRUE;
};
```

Первым делом в теле этой функции выполняется очистка памяти с помощью метода Clear(), который будет рассмотрен ниже. Включение операции очистки позволит вызывать функцию LoadBMP() для одного и того же объекта несколько раз. В некоторых случаях это удобно.

Для вывода изображения на экран необходимо получить доступ к его растровым данным. Один из способов сделать это – загрузить изображение в оперативную память (выделить оператором new необходимое количество памяти и скопировать в нее данные из файла). Другой способ заключается в использовании Windows-механизма «файлов, проецируемых в память». В последнем случае мы перекладываем значительную часть работы по выделению и управлению оперативной памятью на «плечи» операционной системы и обходимся без буферизации данных.

Для открытия файла используем функцию CreateFile():

```
HANDLE CreateFile( LPCTSTR lpszName, DWORD fdwAccess, DWORD fdwShare-
Mode, LPSECURITY_ATTRIBUTES lpsa, DWORD fdwCreate, DWORD fdwAttrsAnd-
Flags, HANDLE  hTemplateFile );
```

Эта функция API Windows используется как для создания, так и для открытия существующих файлов. Функция имеет большое число параметров. Первый параметр – lpszName – имя файла. Второй параметр – fdwAccess – указывает способ доступа к содержимому файла. Так как мы собираемся только читать данные из файла, зададим значение этого параметра GENERIC_READ. Третий параметр – fdwShareMode – определяет режим со-

вместного использования файла. Зададим значение FILE_SHARE_READ – разрешим совместное (одновременное) чтение содержимого файла другим процессам. Следующий параметр – lpsa – указатель на структуру атрибутов защиты: обычно ему присваивается NULL. Пятый параметр – fdwCreate – определяет действия, выполняемые функцией; укажем значение OPEN_EXISTING. В этом случае функция будет открывать существующий файл или выдавать сообщение об ошибке. Следующий аргумент – fdwAttrsAndFlags – атрибуты файла и параметры доступа: укажем FILE_FLAG_RANDOM_ACCESS – сообщим системе, что собираемся считывать данные из файла путем произвольного доступа. Это позволит системе оптимизировать доступ к данным. Последнему параметру – hTemplateFile присвоим значение NULL, так как Windows 95 не поддерживает режим асинхронной работы с файлами.

Следующая функция – CreateFileMapping() – создает объект «проецируемый файл», который содержит важную информацию, необходимую операционной системе при управлении файлом, проецируемым в память. Первый аргумент функции – дескриптор файла, открытого функцией CreateFile(). Остальные аргументы задают атрибуты защиты выделяемой памяти, размер файла и др. параметры.

Функция MapViewOfFile() проецирует файловые данные на адресное пространство процесса и возвращает указатель на эти данные: запомним его в переменной m_pFile.

После того как получили указатель на файловые данные, получаем указатель на заголовок файла – структуру BITMAPFILEHEADER. Так как файл в формате BMP начинается с этой структуры, адрес начала файловых данных будет совпадать с адресом файлового заголовка.

Далее проверяем, что имеем дело с растровым файлом в формате BMP – сравниваем значение поля bfType структуры BITMAPFILEHEADER с идентификатором файлов BMP (0x4D42). Если результат сравнения положительный, то продолжаем работу с файлом.

Получаем указатель на структуру BITMAPINFO путем смещения от начала файла на количество байт в структуре BITMAPFILEHEADER. Адрес начала растровых данных получаем с помощью смещения, указанного в поле bfOffBits структуры BITMAPFILEHEADER.

На этом загрузка файла успешно завершается.

Вывод изображения на экран осуществляется функцией DrawBitmap() класса CRaster. Реализация этой функции в файле raster.cpp приведена ниже.

```
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*/, DWORD rop /*=SRCCOPY*/)
{
     if(m_pBMI==NULL || m_pStartDIB==NULL) return;
```

```
      //размеры на экране не заданы принимаем их 1:1
      if(cx==0) cx=GetBMWidth();
      if(cy==0) cy=GetBMHeight();
      //размеры выводимого фрагмента не заданы - выводим изображение с
      //начальной позиции фрагмента до конца
      if(cx0==0) cx0=GetBMWidth()-x0;
      if(cy0==0) cy0=GetBMHeight()-y0;

      HDC hdc=pDC->GetSafeHdc();
      if(hdc==NULL) return;
      //установка режима масштабирования
      int oldStretchMode=::SetStretchBltMode(hdc, COLORONCOLOR);
      ::StretchDIBits(hdc,  //контекст устройства вывода
                      x,    //левый верхний угол области назначения – x
                      y,    // – y
                      cx,   //размеры области назначения – ширина
                      cy,   // – высота
                      x0,   // x левого верхнего угла выводимого
                            //фрагмента изображения
                      y0,   // y левого верхнего угла выводимого
                            //фрагмента изображения
                      cx0, //размеры исходной области – ширина
                      cy0, // – высота
                      GetBMBitsPtr(), //растровые данные
                      GetBMInfoPtr(), //заголовок растра
                      DIB_RGB_COLORS, //опции
                      rop);           //код растровой операции
if(oldStretchMode!=0)
      //восстановление режима масштабирования
      ::SetStretchBltMode(hdc, oldStretchMode);
}
```

Первый аргумент функции – указатель на контекст устройства вывода (это может быть экран или принтер). Далее следуют аргументы, задающие координаты начала и размеры выводимой на экран части изображения, а также координаты и размеры области назначения на экране. Кроме того задается код растровой операции, выполняемой между выводимым изображением и изображением уже находящимся на экране. Значения этих аргументов заданы по умолчанию. В этом случае вывод осуществляется с позиции (0, 0), как в изображении, так и на экране; размер области на экране совпадает с размером изображения; изображение затирает существующее на экране.

Данная функция содержит в себе функцию API StretchDIBits() с похожими аргументами. В функцию StretchDIBits() передаются дополнительно указатель на растровые данные и на заголовок растра.

Если параметры функции DrawBitmap() заданы по умолчанию, то перед вызовом StretchDIBits() определяются размеры изображения с помощью функций GetBMWidth() и GetBMHeight(). Функция StretchDIBits() может выполнять масштабирование изображения до заданных размеров области назначения. Режим масштабирования выбирается функцией API SetStretchBltMode():

```
int SetStretchBltMode(HDC hdc, int iStretchMode);
```

Первый аргумент функции – hdc – дескриптор контекста устройства вывода, второй аргумент – iStretchMode – режим масштабирования.

Поддерживаются следующие режимы масштабирования:

BLACKONWHITE Выполняет булеву операцию AND между цветом удаленных пикселов и существующих. Этот режим используется, если масштабируется рисунок «черным по белому».

COLORONCOLOR Этот режим удаляет строки (столбцы) пикселей 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 приближал цвет исходной. Наиболее медленный режим. Теоретически должен давать лучшее качество, однако реально иногда искажает цвета до неузнаваемости.

При масштабировании фотографий и цветных рисунков в большинстве случаев наиболее подходящим является режим COLORONCOLOR.

Рассмотрим теперь изменения, которые надо внести в файлы проекта, созданные AppWizard -ом. Прежде всего, откроем файл ShowBM.cpp и удалим из тела функции InitInstance() вызов функции создания нового документа.

```
/////////////////////////////////////////////////////////////////////////
// CShowBMApp initialization

BOOL CShowBMApp::InitInstance()
{
...

     // create a new (empty) document
     //OnFileNew(); - закомментировали функцию
...
}
```

Далее откроем файл ShowBdoc.h и добавим в класс CShowBMDoc переменную-объект класса CRaster. Не забудем также подключить заголовочный файл raster.h.

```
// ShowBdoc.h : interface of the CShowBMDoc class

#include "raster.h"

class CShowBMDoc : public CDocument
{
...

// Attributes
public:
     CRaster    m_RastImg; //изображение
// Operations
...
};
```

Далее добавим переменные в класс CShowBMView (файл ShowBvw.h), которые будут определять размер области назначения на экране.

```
// ShowBvw.h : interface of the CShowBMView class
//шаг масштабирования
#define SCALESTEP 2
class CShowBMView : public CScrollView
{
...
// Attributes
public:
     CShowBMDoc* GetDocument();
     CSize      m_sizeTotal;     //реальные размеры изображения
     double     m_Scale;         //коэффициент масштабирования
// Operations
...
};
```

Значение переменной m_Scale будем инициализировать 1 в конструкторе облика (файл ShowBvw.cpp).

```
CShowBMView::CShowBMView()
{
     // TODO: add construction code here
     m_Scale=1;
}
```

Модифицируем функцию OnInitialUpdate() начальной инициализации облика (файл ShowBvw.cpp). Класс облика нашего приложения является производным от класса CScrollView библиотеки MFC. Класс CScrollView реализует функции прокрутки изображения в окне. Поэтому при инициализации облика необходимо задать размер области прокрутки. Изначально будем задавать размер прокрутки равным размеру изображения.

```
void CShowBMView::OnInitialUpdate()
{
     CScrollView::OnInitialUpdate();
```

```
      // TODO: calculate the total size of this view
      CShowBMDoc* pDoc = GetDocument(); //получили указатель на доку-
мент
      ASSERT_VALID(pDoc);
      // начальный размер области назначения совпадает с размером изо-
бражения
      m_sizeTotal.cx = pDoc->m_RastImg.GetBMWidth();
      m_sizeTotal.cy = pDoc->m_RastImg.GetBMHeight();
      //установить размер прокрутки
      SetScrollSizes(MM_TEXT, m_sizeTotal);
}
```

Для вывода изображения на экран (или принтер) модифицируем функцию OnDraw() – включим в ее тело вызов функции DrawBitmap() класса CRaster (файл ShowBvw.cpp).

```
void CShowBMView::OnDraw(CDC* pDC)
{
      CShowBMDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      pDoc->m_RastImg.DrawBitmap(pDC, 0, 0,
                                          m_sizeTotal.cx*m_Scale,
                                          m_sizeTotal.cy*m_Scale);
}
```

Для реализации функций масштабирования добавим в меню View нашего приложения команды «Zoom In» и «Zoom Out» – увеличение и уменьшение, а в класс CShowBMView обработчики этих команд (с помощью ClassWizard). В обработчиках будем умножать (делить) коэффициент масштабирования на константу SCALESTEP, определенную в файле ShowBvw.h, устанавливать новый размер области прокрутки окна и инициировать перерисовку содержимого окна. Обратите внимание, в функцию DrawBitmap в функции CShowBMView::OnDraw() передается размер области назначения, соответствующий размеру области прокрутки окна.

Для ускорения вызова команд масштабирования им можно назначить «горячие клавиши». Для этого надо открыть каталог ресурсов (файл ShowBM.rc) и двойным щелчком открыть таблицу «акселераторов» (Accelerator). Далее нажимаем правую кнопку мыши и выбираем New Accelerator (рис. 9.4). В появившемся окне диалога выбираем ID команды и назначаем ей клавиши. Простейший способ сделать это – нажать кнопку «Next Key Typed» и затем клавишу или сочетание клавиш, которое собираемся назначить команде. Для команд масштабирования можно назначить клавиши [Page Up] – «Zoom In», [Page Down] – «Zoom Out».

Полный текст программы приведен в архиве ShowBM.rar.

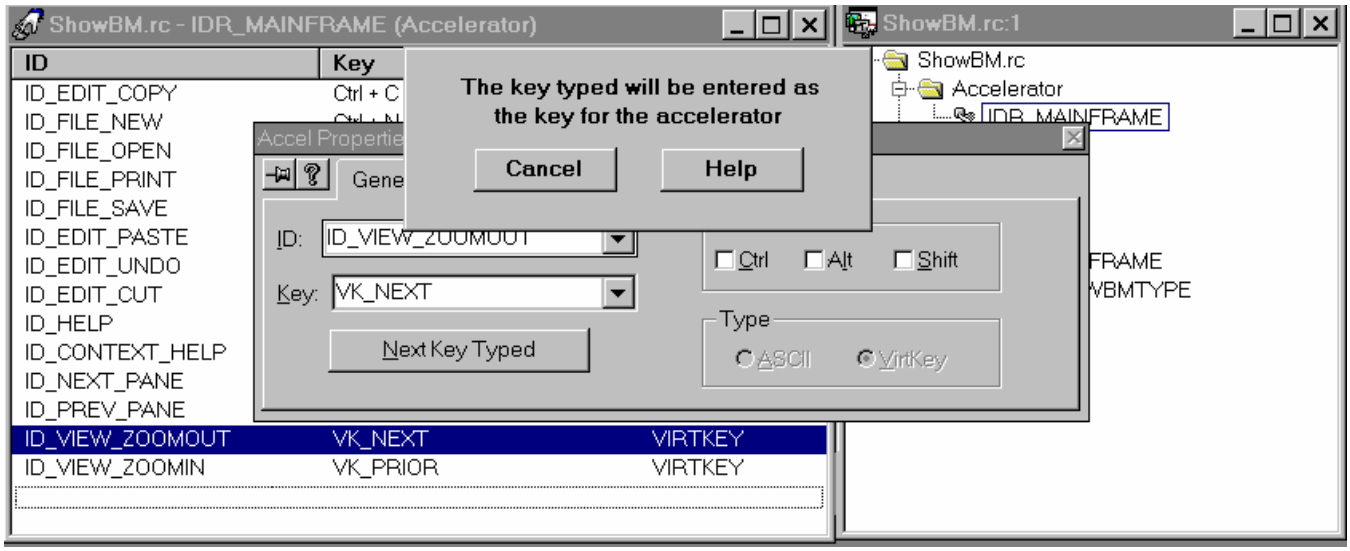


Рис. 9.4. Назначение командам горячих клавиш

Смотри также рекомендуемую литературу [3, 4, 10].

10. СОЗДАНИЕ МУЛЬТИМЕДИЙНЫХ ПРИЛОЖЕНИЙ. ГИПЕРТЕКСТ. ГИПЕРМЕДИА. ВОСПРОИЗВЕДЕНИЕ ЗВУКА И ВИДЕО

Современный компьютерный мир постоянно стремиться к расширению своих возможностей. Компьютер уже не только средство для вычислений, но и библиотека, телевизор, проигрыватель и т.д. Мультимедийными компьютерами принято называть ЭВМ оснащенные средствами для воспроизводства видео и звукозаписей. Слово "мультимедиа" в дословном переводе с английского означает "многостредственность". Мультимедиа-компьютеры обычно оснащены качественным цветным графическим дисплеем, устройством чтения компакт-дисков (CD ROM Driver-ом), звуковой картой с динамиками и микрофоном. Помимо аппаратной части для реализации мультимедийных возможностей требуется соответствующее программное обеспечение. Отличительная черта мультимедиа-программ - сочетание различных форм представления информации (текст, звук, графика, видео).

Рассмотрим далее, как с помощью функций API и классов MFC можно снабдить программу возможностью воспроизводить аудио и видео записи. Начнем с рассмотрения метода создания гипертекстовых систем.

Гипертекст. Термин "гипертекст" используется для обозначения текстовой информации, соединенной гиперсвязями. Гиперсвязи представляют собой своеобразные закладки или ссылки, по которым можно быстро переходить от одной части текстового документа к другой. Программы просмотра гипертекста называют гипертекстовыми. Примером гипертекстовой системы является справочная служба Windows. Развитием гипертекста является ***гипермедиа*** - способ организации и хранения информации - текста, рисунков, фотографий, видеозаписей - с возможностью быстрого и легкого доступа к ним. На принципах гипертекста и гипермедиа организованы практически все размещенные в Интернете страницы.

Гипертекстовые файлы это обычные текстовые файлы, снабженные гиперссылками. В гипертекстовых файлах для организации перемещения по тексту используется специальные управляющие конструкции. Такие управляющие конструкции (или команды) распознаются программой, которая визуализирует гипертекст на экране компьютера. Программа визуализации гипертекста анализирует управляющую конструкцию и выполняет необходимые действия по ее обработке. В управляющей конструкции может задаваться цвет текста, его размер, шрифт и т.д. Если управляющая конструкция обозначает гиперссылку, то обычно после выводимого на экран текста гиперссылки следует скрытый путь к тексту, на который указывает ссылка. Примером программы, осуществляющей визуализацию гипертекстовых файлов, является Internet Explorer. Кроме текста ссылка может указывать на графический, аудио или видео ресурс.

Рассмотрим вкратце, как происходит чтение и визуализация гипертекстовых файлов. Для этого разработаем принципы простой гипертекстовой системы. Пусть в нашем гипертексте различаются всего два вида специального текста: ***заголовок темы*** и ***ссылка.*** Для этого введем пару управляющих конструкций.

Пусть заголовок темы должен располагаться на отдельной строке и начинаться парой символов "#". Например:

##Это заголовок темы.

Ссылки в гипертекстовом файле могут указывать на другие файлы или на другие темы в этом же файле. Ссылки в тексте начинается с пары символов #$, затем следует название ссылки, после названия знак @, затем путь к файлу (если ссылка на текст в другом файле), затем знак | и заголовок темы, на который указывает ссылка. Если ссылка указывает просто на другую тему в том же файле, то путь можно не указывать (и вертикальную черту тогда ставить не надо). Управляющая конструкция ссылки заканчиваться значком "~" (читается "тильда"). Например:

#$Название ссылки@Путь к файлу|Тема~.

В процессе чтения нашего гипертекстового файла программа будет отслеживать появление начальных символов управляющих конструкций. Если ей встретиться пара символов "#", то это означает, что текст следующий за ними до конца строки надо выделить стилем заголовка (например, увеличить шрифт и установить синий цвет). Если встретиться управляющая конструкция ссылки, то название ссылки выделяется соответствующим стилем (например, зеленым цветом и подчеркивается). Символы управляющих конструкций не экран не выводятся.

При обработке управляющей конструкции ссылки помимо установки стиля выполняются также действия, предназначенные для ее «активизации». Для того, чтобы при выборе пользователем этой ссылки программа могла выполнить переход, при отображении ссылки на экране запоминаются координаты описывающего ее прямоугольника (активной зоны), и заголовок темы, на которую указывает ссылка.

Когда пользователь выполняет щелчок левой клавишей мыши в окне отображения текста, программа проверяет, не попал ли он в одну из активных зон. Если да, то программа открывает файл и тему, на которую указывает ссылка.

Аналогично ссылкам на текст могут быть реализованы ссылки на графические, аудио и видео файлы. В последнем случае выбор такой гиперссылки должен вызывать воспроизведение звука или показ графики.

Данная гипертекстовая система реализована в программе OpenBook, которая приведена в архиве OpenBook.rar (для правильной разархивации используйте WinRar95).

Горячие зоны на изображениях. Примерно так же создаются «горячие зоны» на изображениях. Для их создания используются специальные редакторы горячих зон. Например, программа Hotspot Editor (файл Shed.exe в каталоге bin), входящая в состав системы Microsoft Visual C++ 2.0, предназначена для создания горячих зон прямоугольной формы на растровых изображениях. Эта программа используется для подготовки картинок, включаемых в справочные (help) файлы. Координаты «горячих зон», а также контекстные ссылки, сохраняются вместе с растровым изображением в один файл в специальном формате, понятном компилятору help-файлов. Затем, при работе с файлом помощи пользователь, щелкнув на «горячей зоне» изображения, может получить справку, связанную с ее контекстной ссылкой.

Схема создания «горячих зон» может быть и другой. Например, изображение можно хранить в файле в обычном графическом формате, а координаты «горячих зон» в специальном файле данных. Причем один такой файл данных может использоваться для хранения горячих зон нескольких изображений. Каждая «горячая зона» сохраняется в такой файл в виде структуры данных, хранящей название «горячей зоны» (контекстную ссылку), имя графического файла, с которым она связана и ее координаты. В этом случае при открытии графического файла в программе визуализации из файла данных подгружаются связанные с ним «горячие зоны». «Горячие зоны» не обязательно должны быть прямоугольной формы. Для создания «горячих зон» произвольной формы может быть использован класс CRgn библиотеки MFC, который мы использовали для создания полигонов в проекте Painter. Программа визуализации при демонстрации рисунка может отслеживать щелчки мышкой и проверять их на попадание в «горячие зоны». Если попадание произошло, программа должна получить контекстную ссылку, связанную с зоной, и выполнить соответствующие действия.

Воспроизведение звука. Одним из способов воспроизведения звука в формате «WAV» является применение функции API Windows sndPlaySound(). Прототип функции:

```
BOOL sndPlaySound( LPCTSTR  lpszSoundName, UINT  fuOptions);
```

Аргументы:

lpszSoundName Имя (идентификатор) воспроизводимого звука. Это может быть указание записи секции sounds реестра Windows или имя файла на диске. Если параметр равен NULL, то воспроизведение звукозаписи останавливается.

fuOptions Параметры воспроизведения звука. Этот параметр может принимать комбинацию из следующих значений:

Значение	Описание
SND_SYNC	Синхронное воспроизведение звука. Функция не вернет управление, пока не закончится воспроизведение.
SND_ASYNC	Асинхронное воспроизведение звука. Функция возвращает управление сразу после начала воспроизведения. Для прерывания асинхронного воспроизведения можно вызвать SndPlaySound() с параметром lpszSoundName равным NULL.
SND_NODEFAULT	Если звук, указанный в параметре lpszSoundName, не найден то функция тихо завершает свою работу (без воспроизведения звука «по умолчанию»).
SND_MEMORY	Параметр lpszSoundName указывает на звук, хранимый в памяти.
SND_LOOP	Повторять воспроизведение, пока не будет вызвана sndPlaySound() с параметром lpszSoundName равным NULL. Для цикличного воспроизведения звука необходимо указать также значение SND_ASYNC.
SND_NOSTOP	Если при вызове функции sndPlaySound() уже воспроизводиться звук, то функция должна немедленно возвратить FALSE без проигрывания запрашиваемого звука.

Функция возвращает TRUE, если начато воспроизведение звука, иначе FALSE.

Для использования этой функции в программе необходимо подключить заголовочный файл MMSYSTEM.H, в котором она описана:

#include <mmsystem.h>

Воспроизведение видео. Воспроизведение видеозаписи можно осуществить с помощью класса CAnimateCtrl библиотеки MFC. Класс CAnimateCtrl позволяет просматривать видео клипы в формате AVI (Audio Video Interleaved – стандартный Windows video/audio формат), но не позволяет проигрывать сопровождающих их звук. AVI клипы – это последовательность растровых кад-

ров. Такие клипы могут быть созданы в программах 3D-Studio или Autodesk Animator. Видеофрагмент может быть сжат с помощью алгоритма RLE8 или не сжат.

Аниматор позволяет легко реализовать многопоточность в программе и может быть использован, например, для индикации работы приложения во время длительной операции, когда ее продолжительность неизвестна. В этом случае он используется как альтернатива прогрессору и показывает, сто приложение не зависло.

Класс CAnimateCtrl имеет следующие методы:

Метод	Описание
CanimateCtrl	Конструктор класса. Создает объект.
Create	Создает элемент управления – окно, в котором будет выполняться визуализация и соединяет его с объектом.
Open	Открывает AVI клип из файла или ресурса и показывает его первый кадр.
Play	Показывает AVI клип без звука.
Seek	Показывает указанный кадр клипа AVI.
Stop	Останавливает визуализацию клипа AVI.
Close	Закрывает AVI клип.

Рассмотрим некоторые методы подробнее.

BOOL Play(UINT nFrom, UINT nTo, UINT nRep);

Параметры

nFrom Номер кадра, с которого начать воспроизведение (значение должно быть меньше 65536). Значение 0 – воспроизводить с начала клипа AVI.

nTo До какого кадра воспроизводить. Значение - 1 – до последнего кадра.

nRep Количество повторов. Значение - 1 – бесконечно.

Метод возвращает ненулевое значение в случае успеха и 0 при ошибке.

BOOL Seek(UINT nTo);

Параметры

nTo Номер отображаемого кадра (значение должно быть меньше 65536). Значение 0 –показать первый кадр клипа AVI. Значение - 1 – последний кадр.

Метод возвращает ненулевое значение в случае успеха и 0 при ошибке.

Последовательность использования класса CAnimateCtrl.

1. Создать объект CAnimateCtrl.

2. Вызвать метод Create для создания окна и связи его с объектом CAnimateCtrl.

3. Открыть клип AVI из файла или ресурса методом Open.

4. Вызвать метод Play. AVI клип будет показываться, пока программа выполняется. Можно также вызвать метод Seek для визуализации одного кадра.

5. Когда работа с клипом AVI закончена, удалить его из памяти методом Close.

Рассмотрим использование класса CAnimateCtrl в среде разработки Microsoft Visual C++.

Для этого создадим приложение на основе диалога. Для его создания воспользуемся генератором приложений AppWizard. Назовем приложение MMApp (MultiMedia Application). На шаге 1 выберем тип Dialog-based. Далее можно ничего не менять и нажать кнопку Finish. В результате будет создана заготовка для нашего приложения.

Выполним следующие действия:

1. Откроем ресурсы проекта (файл MMApp.rc).

2. Откроем основной диалог нашего проекта (IDD_MMAPP_DIALOG).

3. Из элементов управления уже помещенных в него генератором приложения оставим только кнопку OK и перенесем ее в правый нижний угол окна диалога.

4. Добавим в диалоговое окно элемент управления Animate Control. Для этого выполним щелчок на кнопке Animate в панели инструментов (седьмая сверху в первом столбце) и поместим его на диалоговую панель. Двойной щелчок на элементе управления открывает окно его свойств. В нем можно изменить заданный по умолчанию идентификатор ресурса и поведение элемента. Мы ничего менять не станем, поэтому имя нашему элементу Animate Control будет IDC_GENERIC1 (рис.10.1).

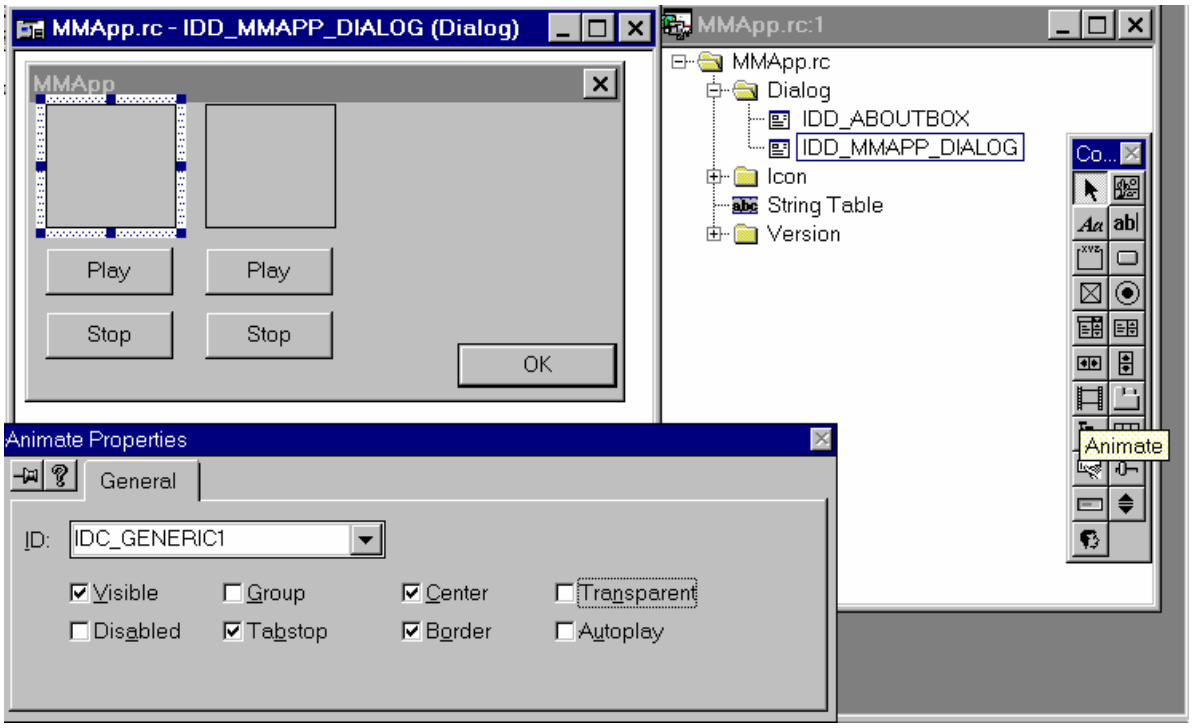


Рис. 10.1. Размещение элементов на панели диалога

1. Аналогично добавим кнопки (Button) Play и Stop. Их идентификаторы изменим на IDC_PLAY1 и IDC_STOP1 соответственно.

2. Вызовем волшебник классов ClassWizard и откроем вкладку Member Variables. Выделим идентификатор IDC_GENERIC1 и нажмем кнопку Add Variable....

3. В появившемся диалоговом окне ввести имя переменной m_AVI1 и нажать OK (рис. 10.2). В результате в класс нашего диалога будет добавлена переменная - объект класса CAnimateCtrl m_AVI1.

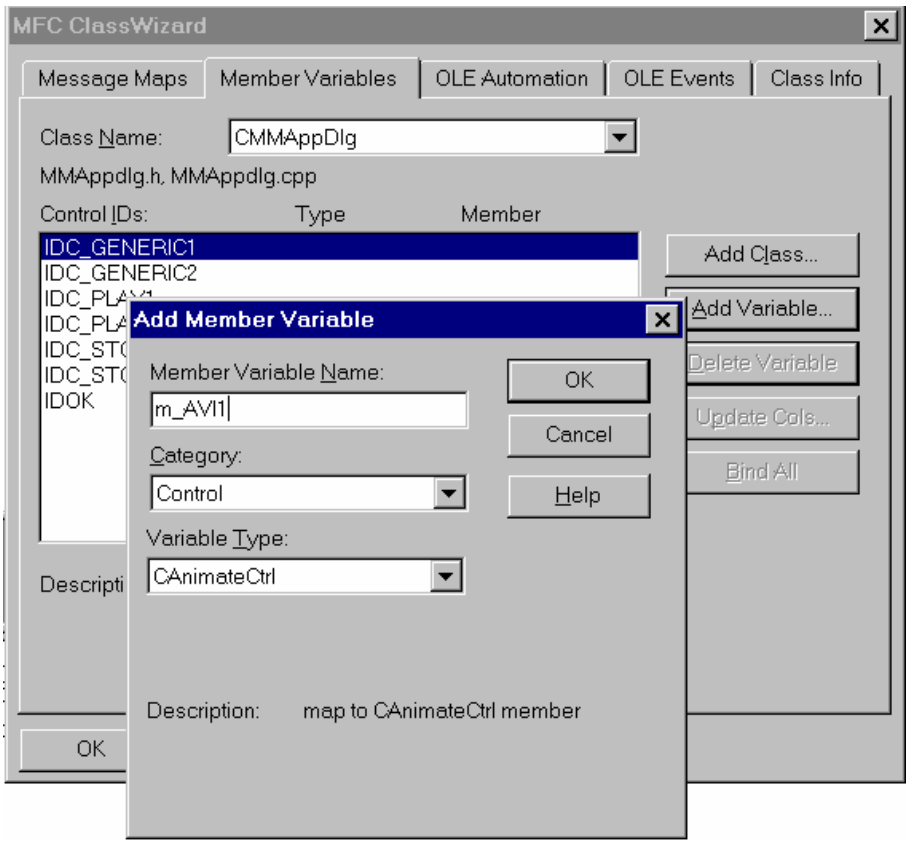


Рис. 10.2. Добавление переменной в класс

1. Добавим обработчики сообщений нажатия клавиш Start и Stop. Для этого также воспользуемся ClassWizard-ом. Откроем вкладку Message Maps, выделим идентификатор IDC_PLAY1 и сообщение BN_CLICKED и добавим функцию OnPlay1(). Аналогично для кнопки Stop (рис. 10.3).

2. Отредактируем созданные ClassWizard-ом тела функций.

```
//файл MMAppdlg.cpp
void CMMAppDlg::OnPlay1()
{
      m_AVI1.Play(0,-1,-1); //запуск воспроизведения
}

void CMMAppDlg::OnStop1()
{
      m_AVI1.Stop();  //останов воспроизведения
}
```

10.Теперь остается только открыть и закрыть файл с клипом AVI. Откроем файл в теле функции OnInitDialog().

```
//файл MMAppdlg.cpp
BOOL CMMAppDlg::OnInitDialog()
{
      ...
```

```
	// TODO: Add extra initialization here
	m_AVI1.Open(AVI1_FILENAME); //откроем файл с клипом AVI

	return TRUE;// return TRUE unless you set the focus to a control
}
```

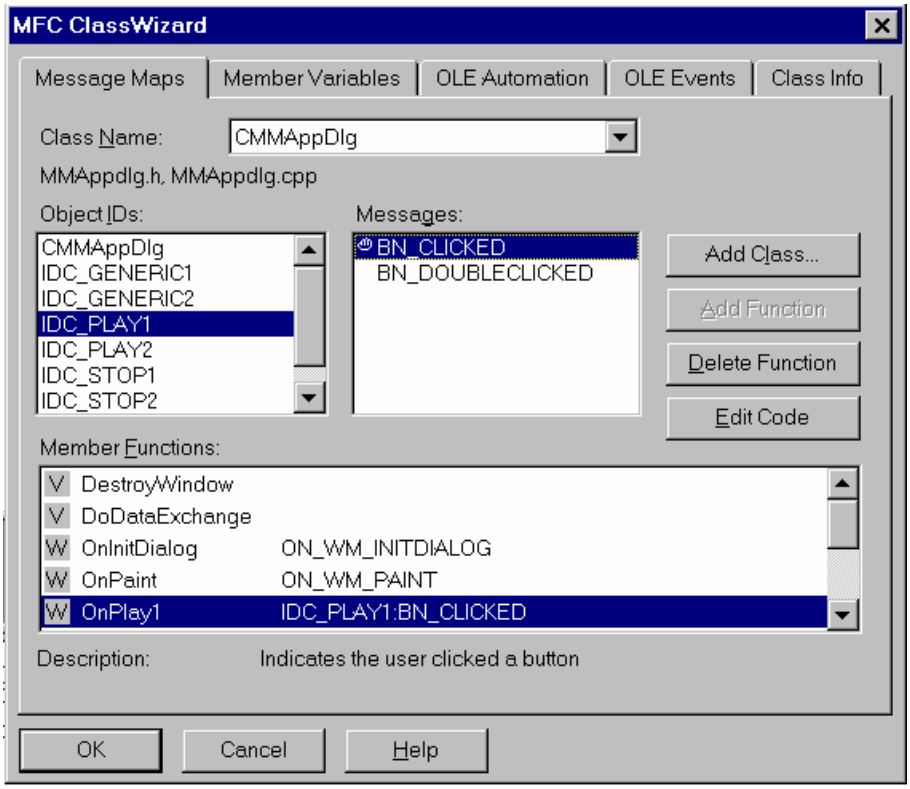


Рис. 10.3. Добавление обработчика нажатия кнопки OnPlay1

AVI1_FILENAME определен также в файле MMAppdlg.cpp:

```
#define  AVI1_FILENAME "search.avi"
```

Это сделано для удобства заменены просматриваемых файлов. Можно было бы имя файла указывать прямо при вызове функции Open().

Закрытие файла AVI можно произвести в функции DestroyWindow() перед уничтожением окна диалога.

```
//файл MMAppdlg.cpp
BOOL CMMAppDlg::DestroyWindow()
{
	m_AVI1.Close();
	return CDialog::DestroyWindow();
}
```

Аналогичным образом можно создать еще несколько элементов управления Animate Control для визуализации клипов AVI. Так как воспроизведение AVI с помощью функции Play происходит асинхронно, то одновременно можно просматривать несколько клипов. В отличие от воспроизведения звука в этом случае конфликтов не возникает.

Смотри также рекомендуемую литературу [3, 4, 10].

11. ПРОГРАММИРОВАНИЕ ГРАФИКИ С ИСПОЛЬЗОВАНИЕМ СПЕЦИАЛИЗИРОВАННЫХ БИБЛИОТЕК OPENGL И DIRECTX

В прошлых разделах мы рассмотрели некоторые математические и программные основы компьютерной графики. 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.

В настоящее время наиболее известны две графические библиотеки - это OpenGL и DirectX.

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 9x/NT.

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 "открытая" означает независимость от производителей. Библиотеку OpenGL могу 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 систем программирования: СИ, Дельфи, Фортрана и др. Общие принципы использования OpenGL в любой системе программирования одинаковы. OpenGL используется в приложениях моделирования и визуализации, применяется для вывода графических данных в САПР, дизайнерских системах 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 работу создания 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 представляет развитые механизмы управления графическими сценами и выполняет определенную автоматизацию.

С точки зрения программиста библиотека OpenGL представляет собой множество команд, часть которых позволяет создавать двумерные и трехмерные объекты, а часть 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. OpenGL хорошо структурирована и включает драйверы основного оборудования, что освобождает разработчика от забот о специфичности 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:

- полный набор базовых команд OpenGL (около 300 штук) для описания форм объектов, преобразования координат, управления освещением, цветом, текстурой, туманом, вывода растровых картинок и т.д. Имена базовых команд в СИ-реализации 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, обраба-

тывать ошибки. Имена команд библиотеки утилит в СИ-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 СИ-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 пикселей и двойной буферизацией. Win32-функции в Windows-приложениях используются вместо команд библиотеки AUX. Имена Win32-функций не имеют специальных префиксов.

Ниже будет рассмотрен пример использования OpenGL.

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 затем и стала основой библиотеки DirectX.

DirectX представляет собой набор интерфейсов прикладного программирования (API) и программных инструментов, позволяющих создавать в Windows 9x/NT (а также в будущих ОС 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 DOS-овского уровня. В добавок DirectX содержит функции, позволяющие работать со звуком, портами ввода-вывода и другими устройствами. DirectX состоит из пяти основных компонент:

- DirectDraw. Компонент специализируется на работе с графикой. Благодаря ему программы получают прямой доступ к видеоадаптеру компьютера, что позволяет им очень быстро переносить изображения из памяти на экран. Библиотека спроектирована так, что она может использовать все аппаратные возможности видеокарты по обработке изображений. Если какие-то требуемые возможности не реализованы аппаратно, то DirectDraw в состоянии их эмулировать программно.

- Direct3D – специализируется на работе с трехмерными объектами. В сочетании с DirectDraw позволяет реализовать быстрый вывод трехмерной графики.
- DirectSound – предназначен для работы с устройствами воспроизведения звука. DirectSound работает значительно быстрее стандартных MCI - функций Windows, позволяет синхронизировать происходящее на экране со звуковыми эффектами, позволяет 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 взаимодействия между программами.

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 решать приложение. И та и другая библиотеки встроены в операционные системы Windows 98/NT 5.0. Однако интерфейс 3D-графики OpenGL встроен в архитектуру операционной системы на более высоком уровне чем DirectX и для связи с аппаратурой опирается на DirectX. С другой стороны, по мнению некоторых экспертов OpenGL лучше выполняет визуализацию (рендеринг) трехмерных сцен, чем, выполняющий те же функции, Direct3D. Кроме того, некоторые авторы утверждают, что компонент DirectX 5.0, называемый DrawPrimitive, который в DirectDraw выполняет рендеринг простых линий и точек, в значительной степени основан на методах визуализации, используемых в OpenGL.

Таким образом можно сделать следующие выводы.

1. Качество визуализации трехмерных сцен с использованием OpenGL и DirectX если и отличается, то не существенно.

2. Считается, что DirectX работает быстрее, чем OpenGL.

3. DirectX в отличии от OpenGL имеет развитые компоненты управления звуком взаимодействия с периферийными устройствами.

4. OpenGL проще инициализировать.

Видимо поэтому OpenGL используется в основном не в играх, а в приложениях моделирования и визуализации. DirectX же используется в мультимедийных приложениях, играх, тренажерах и других сложных программах, предусматривающих активное взаимодействие с пользователем.

Пример OpenGL.

Создадим однодокументное MFC-приложение, которое будет выводить на экран движущуюся фигуру.

Для реализации графики с использованием OpenGL программа должна прежде всего должна 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 может быть 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 вся остальная часть окна 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 сходиться вдалеке. В параллельной же проекции они всегда остаются параллельными.

5. Требуется установить точку наблюдения (точку, в которой находиться камера или глаз наблюдателя) и точку, куда направлен «взгляд».

6. Задать ориентацию системы координат.

Создаем проект:

1. Запустите MSVisualC++.
2. Щелкните меню File->New->Project->MFC AppWizard(exe).
3. Выберете каталог и имя проекта задайте FirstGL, щелкните OK.
4. Step1: Поставьте переключатель на Single document, далее OK.
5. Щелкните Finish.
6. Далее щелкаете Project->Settings->Link->Object/library modules: и добавьте туда opengl32.lib, glu32.lib и glaux.lib (имена файлов разделяются пробелами без запятых).

В CFirstGLView объявите закрытую (private) переменную m_hGLRC типа HGLRC. Там же объявите функцию int SetWindowPixelFormat(HDC) и открытую (public) функцию Display. Вот, что должно получиться:

```
//firstvw.h
class CFirstGLView : public CView
{
protected: // create from serialization only
     CFirstGLView();
     DECLARE_DYNCREATE(CFirstGLView)
private:
CClientDC  *m_pDC;       //контекст устройства рисования (экран)
HGLRC      m_hGLRC; //контекст воспроизведения OpenGL
//установка параметров воспроизведения
int        SetWindowPixelFormat(HDC);

// Attributes
public:
     CFirstGLDoc* GetDocument();
// Operations
public:
void      Display();     //вывод на экран
...
```

Вставьте код функций CFirstGLView::SetWindowPixelFormat и CFirstGLView::Display в файл firstvw.cpp. В начале файла подключите заголовочный файл math.h, который необходим для выполнения расчетов.

```
//firstvw.cpp
#include <math.h>   //требуется для вычисления sin и cos

int CFirstGLView::SetWindowPixelFormat(HDC hDC)
{
     int GLPixelIndex; //номер режима
//PIXELFORMATDESCRIPTOR - структура, определяющая характеристики
//контекста воспроизведения инициализируем структуру значениями
//для полноцветного RGB режимва
PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR),  // размер структуры
    1,                              // номер версии
    PFD_DRAW_TO_WINDOW |            // разрешаем вывод в окно
    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 и
```

```
    0,                                  // без вспомогательного
                                        //буфера
    PFD_MAIN_PLANE,                     // основная плоскость
    0,                                  // резервный компонент
    0, 0, 0                             // без масок слоев
    };


//находит формат пикселей контекста устройства (монитора)
//наиболее близкий к заданному формату пикселей
//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(PIXELFORMATDE
     SCRIPTOR),&pfd)==0)
     return 0;
}
//устанавливаем режим
if (SetPixelFormat( hDC, GLPixelIndex, &pfd)==FALSE)
     return 0;

return 1;
}

void  CFirstGLView::Display(void)
{
     double    rf=0.5,   //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.,1.,1.,1.);
     //подготовка буфера
     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

          glTranslated(x,y,0);     //смещаем оси координат
          glRotated(alfa, 1.0, 1.0, 1.0);    //поворачиваем оси
          glColor3d(1,0,0);                  //устанавливаем цвет
          auxWireTorus(rf, rf*3);       //рисуем фигуру
          //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 отобразить объекты
     SwapBuffers(wglGetCurrentDC());    //выводим на экран
}
```

С помощью Class Wizard переопределите и отредактируйте функцию CFirstGLView::PreCretaeWindow следующим образом:

```
//firstvw.cpp
BOOL CFirstGLView::PreCreateWindow(CREATESTRUCT& cs)
{
     cs.style |= (WS_CLIPCHILDREN | WS_CLIPSIBLINGS);
     return CView::PreCreateWindow(cs);
}
```

С помощью Class Wizard добавьте в класс CFirstGLView обработчики сообщений WM_CREATE, WM_DESTROY и WM_SIZE. Отредактируйте их следующим образом:

```
//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;
     //установка параметров контекста воспроизведения
     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
     //использовать текущий цвет для задания свойств материала
     glEnable(GL_COLOR_MATERIAL);
     //для вычисления цвета использовать текущие параметры
     glEnable(GL_LIGHTING);
     glEnable(GL_LIGHT0);      //учитывать источник света №0
     //положение направленного источника
     GLfloat pos[4] = {5, 5, 5, 0};
     //направление источника освещения
     GLfloat dir[3] = {-1,-1,-1};
     glLightfv(GL_LIGHT0, GL_POSITION, pos); //задаем положение
     //и направление источника освещения с номером 0
     glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);
     return 0;
}

void CFirstGLView::OnDestroy()
{
     //делаем контекст отображения не активным
```

```
     if(wglGetCurrentContext()!=NULL)
          wglMakeCurrent(NULL, NULL);
     //уничтожаем контекст отображения
     if(m_hGLRC!=NULL)
     {
          glDeleteContext(m_hGLRC);
          m_hGLRC = NULL;
     }
     //уничтожаем контекст устройства
     if(m_pDC)
          delete m_pDC;

     CView::OnDestroy();
}

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
     double proportion=(double)cy/cx; //пропорции окна
     //делаем активной матрицу перспективных преобразований
     glMatrixMode( GL_PROJECTION );
     glLoadIdentity();
     //устанавливаем масштаб по осям координат пропорционально
     //соотношению сторон. Если окно станет прямоугольным,
     //пропорции фигуры не изменяться
     glOrtho(-5./proportion,5./proportion,   -5.,   5.,   0,10.);
     //направление взгляда
     gluLookAt( 0,0,5, 0,0,0, 0,1,0 );
     //делаем активной матрицу видовых преобразований
     glMatrixMode( GL_MODELVIEW );
}
```

В StdAfx.h включите заголовочные файлы:

```
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
```

С помощью Class Wizard добавьте функцию OnIdle в класс CFirstGLApp. Эта функция вызывается каждый раз, когда очередь сообщений окна пуста и приложение простаивает. Отредактируйте ее:

```
BOOL CFirstGLApp::OnIdle(LONG lCount)
{
((CFirstGLView*)((CMainFrame*)m_pMainWnd)->GetActiveView())-
>Display(); //перерисовать фигуру.
//продолжать посылку сообщения
     return 1;//CWinApp::OnIdle(lCount); - повторять вызов OnIdle
}
```

Откомпилируйте и запустите на выполнение полученную программу. В результате на экране должен появиться вращающийся каркасный тор, такой

же как на рис. 11.1. Вы можете заменить в функции Display фигуру на другую и поэкспериментировать с этой программой. Например, фигуру можно сделать сплошной, заменив auxWireTorus на auxSolidTorus. Надо учитывать, что в функции SetWindowPixelFormat установлен режим GRB (truecolor 16,7 мил. цветов), поэтому если ваш монитор работает в другом режиме, цвета сцены могут отличаться от ожидаемых. Для получения дополнительной информации можно обратиться к приведенным в начале лекции источникам, а так же к Help Visual C++ и MSDN.

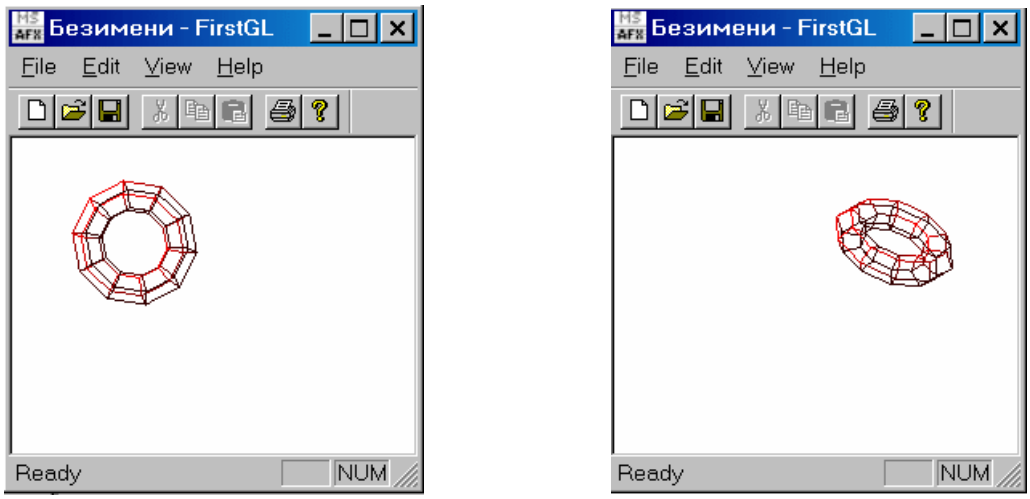


Рис. 11.1. Программа FirstGL в работе

Смотри также рекомендуемую литературу [5, 6].
Интернет-ресурсы [1].

12. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ДИСЦИПЛИНЕ «МАШИННАЯ ГРАФИКА И ГЕОМЕТРИЧЕСКОЕ МОДЕЛИРОВАНИЕ»

Цель дисциплины: научить студентов решать задачи компьютерной графики.

Задачи дисциплины: изучение методов отображения графической информации в двумерном и трехмерном пространстве, программирование алгоритмов компьютерной графики.

Семестр: 6.

В данном курсе изучение алгоритмов компьютерной графики осуществляется путем их программирования в среде разработки Microsoft Visual C++ с использованием объектно-ориентированного стиля. Примеры программ данного курса написаны с использованием Visual C++ версии 2.2. Однако изучать эти примеры можно и в более поздних версиях Visual C++. Для этого достаточно лишь открыть файл проекта (с расширением MAK) и согласиться с конвертацией его в более новый формат.

В первых главах, а также по ходу изложения (где это необходимо) рассматриваются особенности программирования «под Windows» с использованием библиотеки классов MFC и архитектуры Document-View. Однако, надо отметить, что основная задача курса – изучение методов и алгоритмов машинной графики, поэтому на них и будет сосредоточено основное внимание. Программирование же под Windows и использование Visual C++ сами по себе очень обширные темы, поэтому желательно, чтобы слушатели курса имели уже некоторое представление об этих вещах и (или) параллельно прилагали некоторые усилия по их самостоятельному изучению.

ПРОГРАММА КУРСА

№	Темы
1.	Введение: цели и задачи машинной графики; программные средства. Программирование «под Windows»: типы данных в Windows; структура Windows -приложения; API и GDI. Создание приложения в Visual C++: объектно-ориентированное программирование; библиотека MFC; минимальное MFC-приложение; обработка сообщений; инструмент AppWizard. Проект MFCApp. Проект Painter 1.
2.	Архитектура приложений Document-View. Контекст устройства; графические методы класса CDC. Представление геометрических объектов на C++: наследование, виртуальные методы и полиморфизм. Проект Painter 2.

3.	Геометрический инструмент для алгоритмов компьютерной графики. Векторы. Матрицы. Детерминанты. Скалярное произведение. Векторное произведение. Однородные координаты. Преобразования на плоскости. Реализация функции поворота и переноса. Проект Painter 3.
4.	Преобразования в трехмерном пространстве. Параллельная и перспективная проекции.
5.	Программирование преобразований в трехмерном пространстве. Создание трехмерных графических объектов. Реализация функций трехмерных преобразований. Проект Painter 4.
6.	Удаление невидимых линий и поверхностей. Отсечение не лицевых граней. Метод плавающего горизонта. Метод z-буфера. Алгоритмы упорядочивания. Метод построчного сканирования. Проект Painter 5.
7.	Построение кривых. Интерполяция и аппроксимация. Параметрическое задание кривых. Построение составных кривых. Проект Painter 6.
8.	Форматы графических файлов. Цветовые модели. Палитры цветов. Методы сжатия графических данных. Использование растровых ресурсов. Проект Painter 7.
9.	Формат Microsoft Windows Bitmap. Загрузка и сохранение файлов в формате BMP. Проект ShowBM.
10.	Создание мультимедийных приложений. Гипертекст. Гипермедиа. Воспроизведение звука и видео. Проект MMApp.
11.	Программирование графики с использованием специализированных библиотек OpenGL и DirectX.

ТЕМАТИКА КОМПЬЮТЕРНЫХ ЛАБОРАТОРНЫХ И КОНТРОЛЬНЫХ РАБОТ

№ п/п	Содержание работы
1.	**Лабораторная работа №1.** Изучение работы Windows-приложения. Обработка сообщений. Вывод графики на экран. Объектно-ориентированное программирование графики.
2.	**Контрольная работа №1.** Реализация алгоритмов двумерных преобразований.
3.	**Лабораторная работа №2.** Построение сплайновых кривых.
4.	**Контрольная работа №2.** Создание растровых ресурсов. Вывод растровых изображений.

ТЕМЫ КУРСОВЫХ ПРОЕКТОВ

1. Реализовать движение камеры (точки наблюдения) вокруг объекта по заданной траектории.
2. Реализовать конструирование трехмерных фигур из примитивов (прямоугольники, шары, пирамиды, плоскости).
3. Реализовать построение трехмерного изображения графика произвольной функции от двух переменных f(x, y) с возможностью изменения ракурса и масштаба.
4. Реализовать метод z-буфера удаления невидимых частей объектов.
5. Реализовать метод Варнака удаления невидимых частей объектов.
6. Реализовать построение графика функции одной переменной f(x) , например y=sin(x), в произвольном масштабе.
7. Закраска методом Фонга.
8. Закраска методом Гуро.
9. Создание программы рисования с помощью графических примитивов (круг квадрат и т.п.) и полигонов с возможностью построения сплайновых кривых.
10. Создать программу, визуализирующую сцену с перемещающимися сплайновыми графическими объектами.
11. Иллюстрация метода нахождения точки пересечения луча с объектами типа шар, куб.
12. Создать программу, иллюстрирующую построения центральных и перспективных проекций при визуализации трехмерных объектов.
13. Создать программу, визуализирующую сцену с перемещающимися трехмерными графическими объектами.
14. Реализация движения по прямоугольному лабиринту с использованием алгоритма построчного сканирования.
15. Создать программу, иллюстрирующую формирование цвета с использованием модели RGB.
16. Создать программу-редактор палитры цветов растровых файлов.
17. Создание редактора гипертекстовых ссылок.
18. Создание программы просмотра гипертекстовой информации.
19. Создание редактора «горячих зон» на растровых изображениях.
20. Создание гипермедиа программы просмотра растровых изображений с «горячими зонами» и гипертекста.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Шикин Е.В., Боресков А.В. Компьютерная графика. Динамика, реалистические изображения. – М.: «ДИАЛОГ-МИФИ», 1995.–288 с., ил.

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.

2. Шикин Е.В., Плис А.И. Кривые и поверхности на экране компьютера. Руководство по сплайнам для пользователей. - М.: .: «ДИАЛОГ-МИФИ», 1996. - 240 с., ил.

   В книге описаны одномерные кубические и двумерные бикубические интерполяционные и сглаживающие сплайны. Приведены примеры их программной реализации.

3. 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 видео и аудио файлов.

4. Мюррей Д., Ван Райпер У. 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 содержаться программы чтения/записи и преобразования различных форматов, исходные коды, а также Интернет-адреса, где можно добыть свежую информацию и программы.

5. Томпсон Н. Секреты программирования трехмерной графики для Windows 95. Пер. с англ. СПб.: Питер, 1997.–325 с., ил.

   Рассматривается использование библиотеки DirectX для программирования трехмерной графики.

6. 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 очень полезна и для программистов, предпочитающих другие языки. В приложении приведен пример программирования с использованием OpenGL на языке Си.

7. Биллиг В.А., Мусикаев И.Х. Visual C++ 4. Книга для программистов. - М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1996. - 352 с., ил.

   В книге рассмотрены основы программирования в Microsoft Visual C++. Очень хорошая книга для самостоятельного изучения.

8. Нортон П., Макгрегор Р. Руководство Питера Нортона. Программирование в Windows 95/NT 4 с помощью MFC. В2-х книгах. – М.: «СК Пресс», 1998. 1176 с. (в двух книгах), ил., CD-ROM.

Очень хорошее 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. Книга позволяет получить законченное представление о программировании под Win. Очень рекомендую.

9. Рихтер Д. Windows для профессионалов (программирование в Win32 API для Windows NT 3.5 и Windows 95)/Пер. с англ. – М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd», 1995. –720 с.

Подробно рассматривается использование функций API Windows. Использование виртуальной памяти. Управление процессами и потоками. Разработка DLL-библиотек.

10. Шилдт Г. MFC: основы программирования: пер. с англ. – К.: BHV, 1997. – 560 с.

Посвящена использованию MFC при программировании на Visual C++.

11. Аммерал Л. Принципы программирования в машинной графике:Пер. с англ.-М.:Сол Систем,1992.-224 с.:ил.- Третья книга серии «Машинная графика на языке Си» из 4 книг этого автора. Пожалуй, для нашего курса самая полезная.

Рассматриваются вопросы аналитической и проективной геометрии и программирования в машинной графике. Алгоритмы доведены до "готовых к работе" графических программ на языке Си. Подробно и доступно излагаются основы компьютерной графики. Изложение проиллюстрировано многими примерами. Немного староват стиль программирования, однако очень полезно почитать с точки зрения эффективного использования языка Си.

12. Павлидис Т. Алгоритмы машинной графики и обработки изображений. - М.: Радиосвязь, 1986.- 398 с.

Рассматриваются базовые алгоритмы компьютерной графики.

ИНТЕРНЕТ-РЕСУРСЫ

1. http://opengl.org.ru - Сайт посвящен программированию графики с использованием библиотеки OpenGL. На сайте можно найти много полезной

информации по этому вопросу, в частности электронную версию книги Игоря Тарасова «Введение в OpenGL». Книга позволяет быстро изучить основы использования библиотеки OpenGL. Также на сайте (в разделе «Links») содержатся ссылки на другие Интернет-ресурсы посвященные вопросам компьютерной графики.

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.enlight.ru/ - Сайт, посвящен разработке мультимедийных проектов, как его характеризуют авторы «- объемный проект, содержащий различную информации по демомейкингу, его истории и терминологии, программированию видео и аудио эффектов, математике, компьютерной графике, звуку»; «- различная информация по программированию 3D графики» и т.д. Чтобы долго не искать, можно сразу зайти на http://www.enlight.ru/faq3d/content.htm - оглавление «Часто задаваемых вопросов (по 3D графике)» - по сути, целая книга с иллюстрациями и примерами программ. Очень рекомендую.

ПРОГРАММЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНЫХ РАБОТ

Лабораторные работы предназначены для закрепления лекционного материала. Задания на лабораторные работы заключаются в модификации проектов-заготовок, с тем, чтобы придать им новые возможности. Проекты-заготовки содержаться в одноименных RAR архивах. Для их модификации архив необходимо распаковать на своем жестком диске, затем запустить Visual C++ и открыть файл проекта с расширением MAK. Возможно, в более поздних, чем 2.2, версиях Visual C++ будет предложено преобразовать проект в более новый формат. Надо согласиться с таким преобразованием. После открытия проекта-заготовки надо внести в него дополнения, которые описаны в задании на лабораторную работу.

В первой лабораторной работе рассматривается две небольшие программки. Одна из которых демонстрирует создание минимального Windows-приложения с помощью библиотеки MFC, а вторая является проектом Painter, построенным на основе архитектуры Document-View и классов библиотеки MFC, и реализует минимальные графические функции. Проект Painter будет модифицироваться в ряде лабораторных, постепенно приобретая новые свойства. В последних лабораторных работах рассматриваются многодокументный (MDI-интерфейс) проект ShowBM, который осуществляет просмотр графических файлов в формате BMP, и проект MMApp, основанный на интерфейсе диалогового окна, и выполняющий просмотр видеоклипов в формате AVI.

Задание на лабораторную работу №1

Изучение работы Windows-приложения. Обработка сообщений. Вывод графики на экран. Объектно-ориентированное программирование графики.

Лабораторная работа основана на материале лекций №1 и №2.

1. Создать и изучить работу «минимальной MFC-программы». Для этого выполнить команду File-New и в появившемся диалоге выбрать Project (рис. 1).

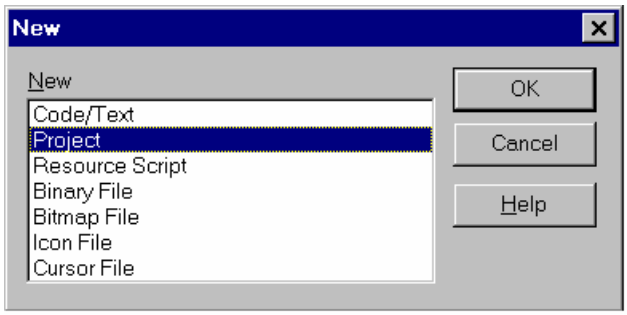


Рис. 1. Диалог выбора типа проекта

Задать имя проекта «MFCApp» и тип проекта Application (рис. 2). После нажатия кнопки Create будет предложено указать имена файлов, которые будут использованы в проекте. В поле File Name ввести имя mfcapp.cpp и нажать кнопку Add (рис. 3). Появиться предложение создать файл с таким именем – ответить OK. Затем нажать Close. Открыть файл mfcapp.cpp и поместить в него текст «минимальной MFC-программы» (см. лекцию №1). Затем выполнить команду Project-Settings и указать, что в проекте будут использоваться классы MFC (рис. 4). Откомпилировать программу – команда Project-Build; запустить программу – команда Project-Run.

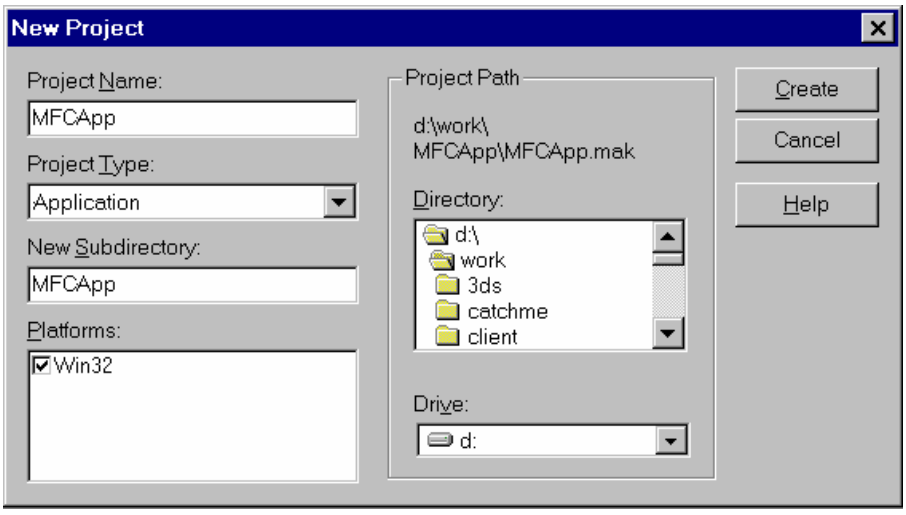


Рис. 2. Создание проекта программы MFCApp

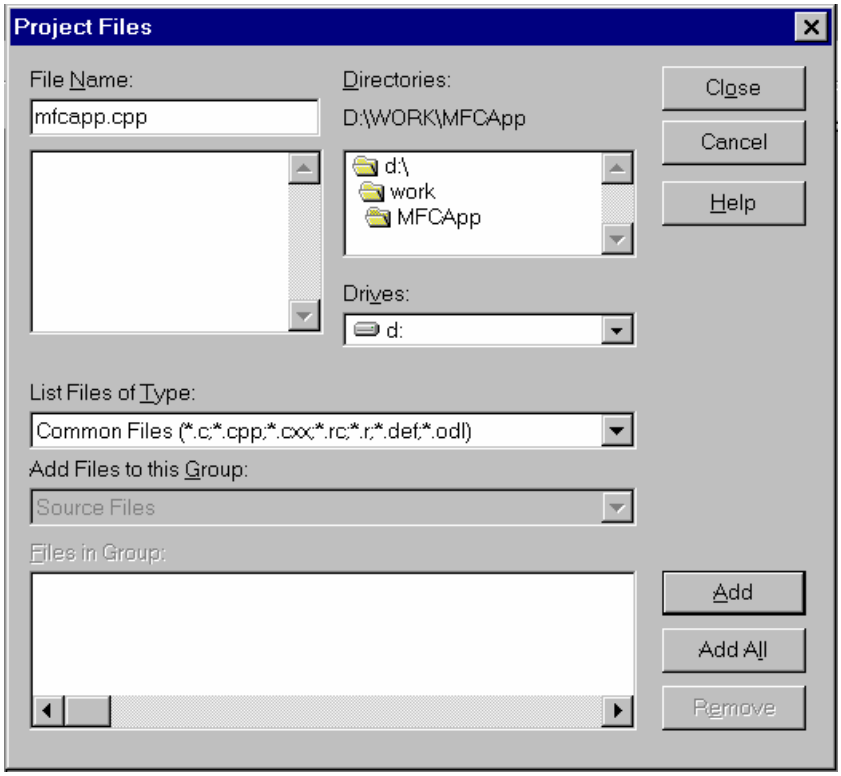


Рис. 3. Выбор файлов проекта

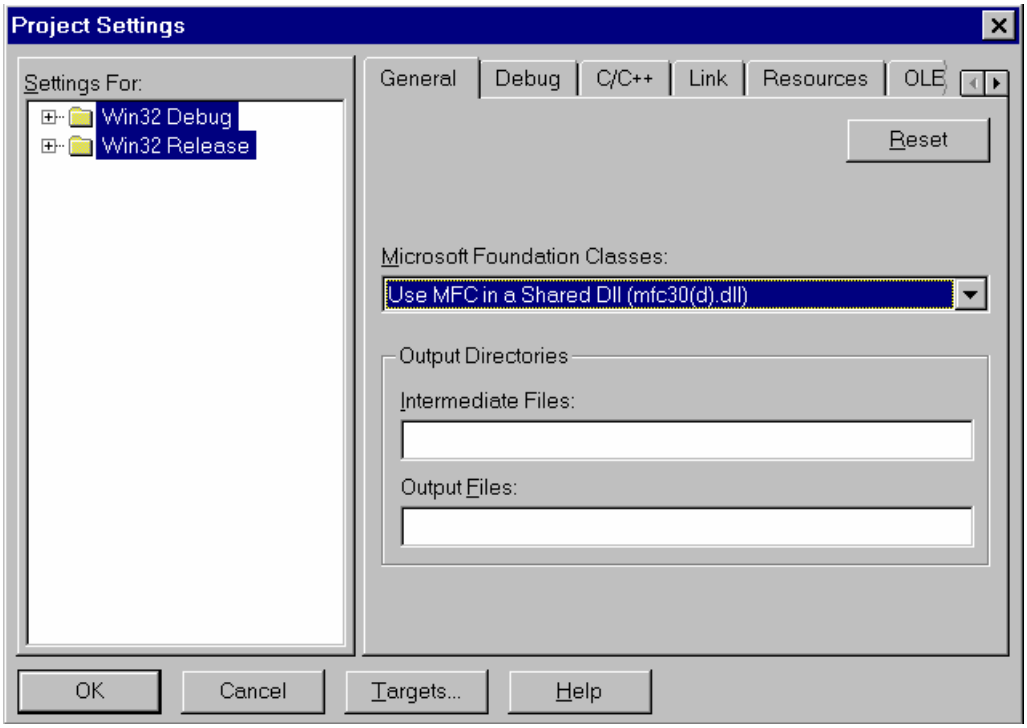


Рис. 4. Задание свойств проекта

1. Добавить в «минимальную MFC-программу» обработку сообщения WM_RBUTTONDOWN нажатия правой кнопки мыши. Выполняется аналогично добавлению обработчика нажатия левой кнопки мыши (см. главу 1).

2. Создать проект Painter (процесс создания приведен выше). Добавить в него с помощью инструмента ClassWizard (см. главу 1) обработку нажатия правой клавиши мыши: действие – отмена последней точки, введенной с помощью левой клавиши.

1. Изучить работу программы Painter 2 (см. главу 2).

2. Запрограммировать классы фигуры в соответствии с вариантом задания (фигура может быть составной из нескольких примитивов).

3. Дополнить программу Painter 2 возможностью рисования новых фигур.

Для реализации класса новой фигуры открыть файл Shapes.h и описать интерфейс нового класса, порожденного от класса CBasePint. Переопределить виртуальные функции Show и GetRegion. Реализацию данных функций поместить в файл Shapes.cpp. Добавить кнопки в панель инструментов и описать функции-обработчики сообщений их нажатия.

Варианты заданий.

1. Сегмент шара (функция Chord класса CDC) рис. 5, 1.
2. Сектор шара (функция Pie класса CDC) рис. 5, 2.
3. Эллипс (функция Ellipse класса CDC) рис. 5, 3.
4. Скругленный прямоугольник (функция RoundRect класса CDC) рис. 5, 4.
5. - 20. Составные фигуры рис. 5, 5-20 рисуются с помощью функций класса CDC.

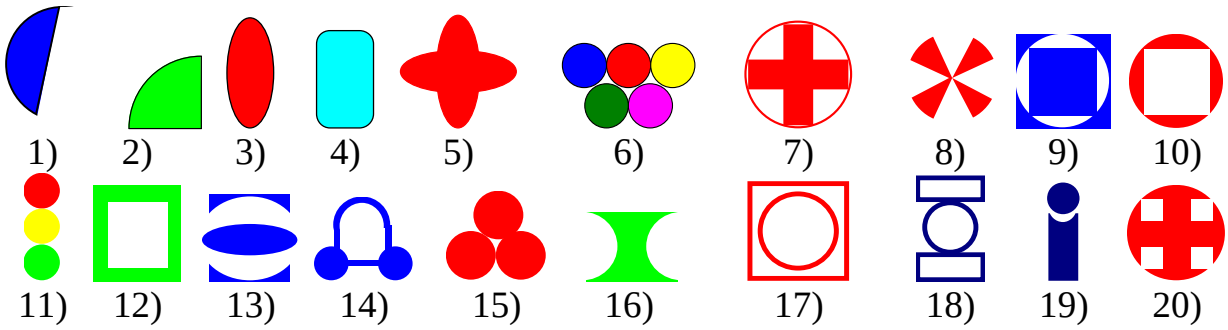


Рис. 5

Задание на контрольную работу №1

Реализация алгоритмов двумерных преобразований.

Работа основана на материале главы 3.

1. Реализовать в программе Painter функцию рисования полигонов.

2. Добавить в Painter команды поворота и переноса полигона.

Для выполнения первого пункта задания, иерархиею классов фигур надо дополнить классом полигона (см. главу 3). Операции поворота и переноса можно реализовать, применив ко всем точкам графического объекта метод Transform базового класса CBasePoint. Для реализации операций поворота и переноса необходимо:

- Добавить в класс фигуры полигон функцию преобразования положения, которая в качестве аргументов будет принимать угол, на который надо повернуть фигуру и сдвиги по x и y. В качестве точки, вокруг которой будет осуществляться поворот, можно выбрать, например, первую или последнюю точку полигона.
- Добавить в меню соответствующую команду, и обработчик этой команды, в котором должна вызываться функция преобразования положения объекта-фигуры.

3. Запрограммировать функцию рисования фигуры с помощью полигона. Варианты фигур приведены на рис. 6.

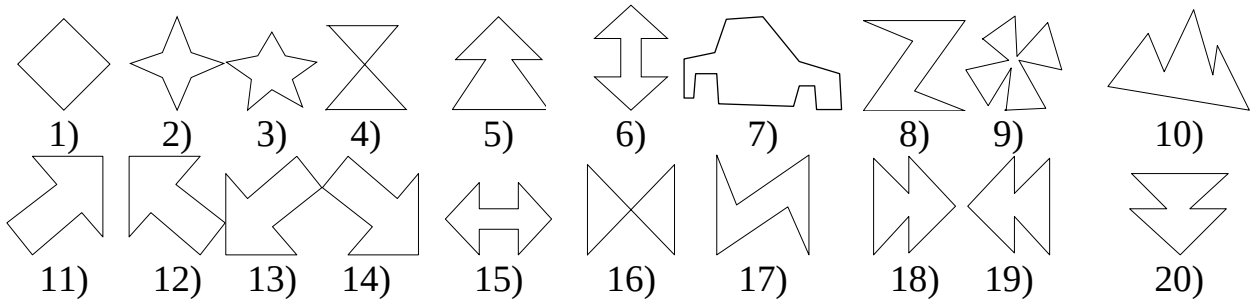


Рис. 6

Задание на лабораторную работу №2

Построение сплайновых кривых.

Лабораторная работа основана на материале главы 7.

В лекции №7 изложены основы построения сплайновых кривых. Лабораторная выполняется на базе проекта- заготовки Painter6 (архив Painter6.rar) и программы Bezier (архив Bezier.rar).

В лабораторной работе требуется:

1. Создать класс своей сплайновой фигуры на базе класса CSpline из проекта Painter6 аналогично классу CSplineStar. Добавить в программу Painter6 возможность рисования новых сплайновых фигур.

2. Написать функцию, выполняющее построение геометрически непрерывной составной сплайновой кривой Безье по произвольному числу базовых точек. Для реализации функции можно использовать описанный в лекции №7 механизм введения вспомогательных точек, и функцию PolyBezier класса CDC. За основу можно взять программу Bezier (архив Bezier.rar).

3. Запрограммировать функцию рисования фигуры с помощью непрерывной сплайновой кривой. В качестве вариантов базовых точек взять вершины фигур, приведенных на рис. 6.

Задание на контрольную работу №2

Создание растровых ресурсов. Вывод растровых изображений.

Задания основаны на материале главы 8.

1. Реализовать в программе Painter класс для работы с растровыми ресурсами.

2. Создать растровую картинку и обеспечить вывод ее на экран, аналогично другим графическим объектам в проекте Painter.

3. Создать свой растровый шаблон кисти и кисть на ее основе. Выполнить заполнение какой-либо фигуры шаблонной кистью.

Приложение 1

ОБЩИЕ ТРЕБОВАНИЯ К СОДЕРЖАНИЮ КУРСОВЫХ ПРОЕКТОВ, СВЯЗАННЫХ С РАЗРАБОТКОЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Пояснительная записка к курсовому проекту должна содержать:

1. Титульный лист.
2. Аннотацию.
3. Задание на проектирование.
4. Оглавление.
5. Введение (Общая информация, назначение программного продукта, необходимость его разработки).
6. Постановку задачи.
7. Обзор литературы (Состояние проблемной области, наличие аналогов разработки, существующие алгоритмы, средства разработки программ).
8. Анализ задания, выработка требований к программе, выбор средств реализации.
9. Описание алгоритмов программы и используемых математических моделей (формулы, схемы, общее описание алгоритмов).
10. Описание реализации программы (структурная схема программы, структура и типы данных, иерархия и описание классов, основные процедуры, их назначение и взаимосвязь, входные и выходные данные программы и отдельных процедур, смысл основных переменных программы, построение файловой системы, используемые средства среды программирования).
11. Описание программы «для пользователя» (описание меню, порядок работы с программой - может быть в виде инструкции для пользователя, и пр.).
12. Тестирование программы (описание тестовых задач и результатов тестирования).
13. Заключение (Основной итог работы, сопоставление желаемых и полученных результатов, встретившиеся проблемы, целесообразность и возможные направления дальнейшего совершенствования программы).
14. Список использованной литературы (обычно не менее 5 наименований).
15. Приложения (блок-схемы алгоритмов структурная схема и листинг основных фрагментов программы, результаты тестирования и т.д.)

При оформлении пояснительных записок необходимо обратить внимание на следующее:

1. Рисунки должны быть пронумерованы и подписаны, в тексте должны бать ссылки на рисунки.

2. Таблицы должны быть пронумерованы и иметь заголовки.

3. Листы пояснительной записки должны иметь сквозную нумерацию (с первого по последний лист, включая приложения). Номер на титульном листе не ставится.

4. Приложения должны быть пронумерованы и иметь заголовки.

5. Текст программы должен детальные комментарии.

6. Список литературы должен соответствовать требованиям ГОСТа;

7. В тексте записки должны быть ссылки на используемую литературу.

Оценка за курсовую работу выставляется с учетом правильности оформления пояснительной записки.

СОДЕРЖАНИЕ