Н. А. ТЮКАЧЕВ,
В. Г. ХЛЕБОСТРОЕВ

C#. ПРОГРАММИРОВАНИЕ 2D И 3D ВЕКТОРНОЙ ГРАФИКИ

Учебное пособие

Издание четвертое, стереотипное

· САНКТ-ПЕТЕРБУРГ ·
· МОСКВА ·
· КРАСНОДАР ·
2020

УДК 004
ББК 32.973я73

Т 98 **Тюкачев Н. А.** C#. Программирование 2D и 3D векторной графики : учебное пособие / Н. А. Тюкачев, В. Г. Хлебостроев. — 4-е изд., стер. — Санкт-Петербург : Лань, 2020. — 320 с. + CD. — (Учебники для вузов. Специальная литература). — Текст : непосредственный.

ISBN 978-5-8114-4754-1

Книга посвящена программированию векторной графики. Описываются основные методы графических классов и приводятся примеры их использования, рассматриваются аффинные преобразования на плоскости и в трехмерном пространстве и различные виды проецирования. Приводится обзор различных моделей трехмерных тел. Одна из них посвящена сложной теме — бинарные операции над множествами. Описан лучевой алгоритм определения принадлежности точки многоугольнику и многограннику. Описывается библиотека OpenGL и основные команды этой библиотеки. Приводятся простые примеры 2D графики.

Книга рассчитана на бакалавров направлений подготовки «Прикладная математика и информатика», «Математика и компьютерные науки», «Фундаментальная информатика и информационные технологии», «Математическое обеспечение и администрирование информационных систем», «Информатика и вычислительная техника», «Информационные системы и технологии», «Программная инженерия», «Информационная безопасность», студентов специальностей «Компьютерная безопасность» и «Информационно-аналитические системы безопасности», а также учащихся старших классов и лиц, самостоятельно изучающих языки программирования.

УДК 004
ББК 32.973-018я73

Рецензент
М. Г. МАТВЕЕВ — доктор технических наук, профессор, зав. кафедрой информационных технологий управления Воронежского государственного университета.

Обложка
Е. А. ВЛАСОВА

Введение

Компьютерная обработка графической информации сводится к решению трех классов задач.

1. Задачи распознавания образа: на основе имеющегося оцифрованного изображения осуществить идентификацию объекта, то есть получить его описание. Здесь выполняется преобразование *изображение → описание объекта.*
2. Задача обработки изображения: осуществить преобразование имеющегося изображения с целью изменения его свойств. Выполняется преобразование *изображение → изображение.*
3. Задача построения изображения: по описанию объекта построить его изображение на графическом устройстве. Здесь выполняется преобразование *описание объекта → изображение.*

Именно этой третьей задаче посвящена эта книга. При этом описание объекта должно иметь вид математической модели. Таким образом, в рамках компьютерной графики решаются две основные подзадачи:

- построение математической модели изображаемого объекта;
- визуализация объекта в соответствии с этой моделью.

В книге изложены основы программирования 2D и 3D графики на языке C# в среде .Net Framework, описаны свойства и методы классов `Graphics`, `Color`, `Pen`, `Brush`, `Font`, предназначенных для рисования. Описаны модели цветов: `RGB (Red, Green, Blue)`, `CMY (Cyan, Magenta, Yellow)`, `CMYK`, `HSB` и `Lab`.

В третьей главе рассмотрены задачи: интерполяции полиномами, интерполяции кубическими сплайнами, сглаживания и аппроксимации. Далее в этой главе рассматриваются аффинные преобразования на плоскости и в пространстве. Достаточно подробно описываются виды проецирования – ортографическое, аксонометрическое, косоугольное и центральное. Приведена классификация моделей трехмерных тел: каркасные модели, граничные, поверхностные модели, сплошные модели.

В четвертой главе обсуждаются простые графические проекты: сортировка элементов массива, морфинг, падение глобуса, велосипед, деформация изображений, растровый редактор, редактирование графа.

Вся пятая глава посвящена проекту «Векторный редактор». В ней обсуждаются структура данных, масштабирование, создание, прорисовка и изменение объектов, запись и чтение данных.

В шестой главе описаны проекты для построения графиков функций одной и двух переменных. В этой же главе приведен проект для построения

интерполяционных кривых многочленами Лагранжа, методом наименьших квадратов, кубическими сплайнами, кривыми Безье.

В главе «Бинарные операции» предлагается инцидентный лучевой алгоритм определения принадлежности точки многоугольнику и построенный на его основе алгоритм определения результата булевских операций над двумя многоугольниками или многогранниками.

В главе «Платоновы тела» описывается проект для построения тетраэдра, октаэдра, додекаэдра. Проект позволяет вращать тела и систему координат, рисовать тень, двигать две точки схода для моделирования перспективы, имитировать освещение и строить стереоизображение.

В последней главе «Использование графической библиотеки OpenGL» приводится порядок установки, инициализации и завершения работы с OpenGL, описываюся команды и примитивы OpenGL. К этой главе прилагаются два проекта: для двумерной и трехмерной графики.

Текст содержит большое количество примеров программного кода, способствующих усвоению материала. Книга рассчитана на студентов высших учебных заведений, учащихся старших классов, а также лиц, самостоятельно изучающих языки программирования.

Глава 1. ОСНОВНЫЕ ГРАФИЧЕСКИЕ КЛАССЫ C#

Возможности визуальной студии .NET позволяют написать сложные приложения, используя только средства управления, доступные в наборе компонентов. Набор компонентов включает в себя средства управления для показа данных (ярлыки, календари, списки и т.д.), наборы (радио-кнопки, `CheckBoxes`, списки и т.д.) и контейнеры для сбора данных (`DataGrids`, `TextBox` и т.д.). Кроме того, несколько средств управления и компонентов управляют датой и временем (`Timer` и т.д.).

Однако, часто возникает необходимость показывать данные, используя инструменты доступа через GDI+ (Graphics Device Interface — интерфейс графических устройств) и графические объекты.

Для демонстрации возможностей этих инструментов эта и следующая главы описывают основы программирования графики с простыми программами.

1.1. ПРОСТРАНСТВА ИМЕН ГРАФИЧЕСКИХ КЛАССОВ

Классы, предназначенные для рисования, заключены в библиотеку `System.Drawing.dll`, которая определяет несколько пространств имен.

Таблица 1.1

Основные пространства имен GDI+

Пространство имен	**Назначение**
`System.Drawing`	Определяет типы для визуализации: шрифты, перья, кисти и т.п. Содержит класс `Graphics`
`Drawing.Drawing2D`	Представляет классы, используемые для более развитой функциональности графики (градиентные кисти, концы перьев, геометрические трансформации и т.п.)
`Drawing.Printing`	Определяет классы, позволяющие печатать на бумаге, взаимодействовать с принтером и форматировать общий вид печати
`Drawing.Imaging`	Определяет классы, позволяющие работать с графическими файлами (изменять палитры, извлекать метаданные изображений, манипулировать метафайлами и т.п.)
`System.DrawingText`	Позволяет работать с коллекциями шрифтов

1.2. Пространство имен System.Drawing

Большинство графических классов и других типов находится в пространстве имен *System.Drawing*. В нем есть классы, представляющие кисти, перья, шрифты и изображения. Пространство имен *System.Drawing* определяет также множество служебных структур и классов, таких как *Color, Point, Size* и *Rectangle*. В таблице 1.2 перечислены некоторые основные типы.

Таблица 1.2

Основные типы пространства имен *System.Drawing*

Класс	**Назначение**
Bitmap	Представляет полотно для рисования в памяти. Инкапсулирует изображения (*.bmp и т.п.)
Brush *Brushes* *SolidBrush* *SystemBrushes* *TextureBrush*	Объекты кистей используются для заполнения внутренних областей графических фигур, таких как прямоугольники, эллипсы и многоугольники. У класса *SolidBrush* есть свойство *Color*
Color *SystemColors*	Определяют множество статических свойств, используемых для получения цветов перьев и кистей
Font *FontFamily*	Инкапсулирует свойства шрифта (название, стиль, курсив, размер и т.п.). *FontFamily* предоставляет абстракцию для группы шрифтов, имеющих сходный дизайн, но различия в стиле
Graphics	Представляет поверхность рисования и множество методов для визуализации текста, изображений и геометрических шаблонов
Icon *SystemIcons*	Представляют пиктограммы, а также набор стандартных системных пиктограмм
Image *ImageAnimator*	*Image* — абстрактный базовый класс, предоставляющий функциональность для классов *Bitmap, Icon* и *Cursor*. Класс *ImageAnimator* анимирует изображение
Pen *Pens* *SystemPens*	Перья — это объекты, используемые для рисования линий. Класс *Pen* определяет набор статических свойств, возвращающих *Pen* заданного цвета
Point *PointF*	Структуры описывают целые или вещественные координаты (*x, y*)
Rectangle *RectangleF*	Структуры описывают прямоугольные области с целыми или вещественными параметрами
Size *SizeF*	Структуры описывают ширину/высоту с целыми или вещественными параметрами

Продолжение табл. 1.2

Класс	Назначение
StringFormat	Используется для инкапсуляции различных средств текстовой компоновки (например, выравнивание, межстрочный интервал и т.п.)
Region	Тип описывает внутреннюю часть геометрического образа, состоящего из прямоугольников и путей

Все эти классы будут далее описаны, поскольку ими приходится пользоваться, но перед переходом к ним давайте подробно изучим основной класс *Graphics*.

1.3. Класс Graphics

Класс *Graphics* представляет GDI+ поверхность рисования. Графический объект поддерживает для поверхности рисования: масштаб, единицы, ориентацию поверхности рисования.

Приведем пример простейшего приложения, которое на форме рисует эллипс:

Листинг 1.1. Простейшая графическая программа

```
public partial class FormMain : Form
{
    Graphics g;
    private void Draw()
    {
        g = CreateGraphics();
        g.Clear(Color.White);
        g.DrawEllipse(Pens.Black,10,10,200,100);
        g.Dispose();
    }
    private void FormMain_Paint(object sender,
        PaintEventArgs e)
    {
        Draw();
    }
}
```

Результат работы программы показан на рисунке 1.1.

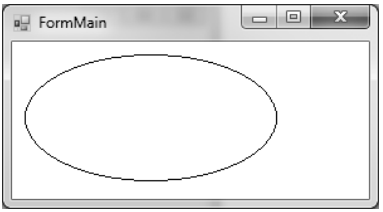

Рис. 1.1. Простейшая графическая программа

Отметим важные моменты, которые необходимо соблюдать при создании любого GDI-проекта:

- должно быть определено свойство *g* класса *Graphics*;
- необходимо создать обработчик события *onPaint*, в котором будет вызываться метод рисования *Draw()*.

В методе *Draw()*:

- методом *CreateGraphics()* необходимо создать объект *g*;
- методом *Clear()* очистить поверхность рисования;
- что-нибудь нарисовать;
- освободить память от объекта *g*.

Класс *Graphics* содержит много свойств. Часто используемые свойства перечислены в таблице 1.3, большинство которых будет демонстрироваться позже. Все они позволяют не только читать себя, но и изменять.

Таблица 1.3

Свойства класса *Graphics*

Свойство	**Класс**	**Описание**
Clip	*Region*	Возвращает или задает объект *Region*, ограничивающий область рисования данного объекта
DpiX / DpiY	*Float / single*	Задает значение горизонтального и вертикального разрешения в точках на дюйм
PageScale	*Float / single*	Задает значение для масштабирования между универсальными единицами и единицами страницы
PageUnit	*GraphicsUnit*	Возвращает или задает единицу измерения для координат страницы данного объекта

Свойство *PageScale* устанавливает масштабирование между мировыми единицами и единицами страницы.

1.4. Координаты

В определении декартовой системы координат оси *X* и *Y* проходят так, как показано на рисунке 1.2. Значения справа и выше центра координат являются положительными, а значения слева и ниже – отрицательными.

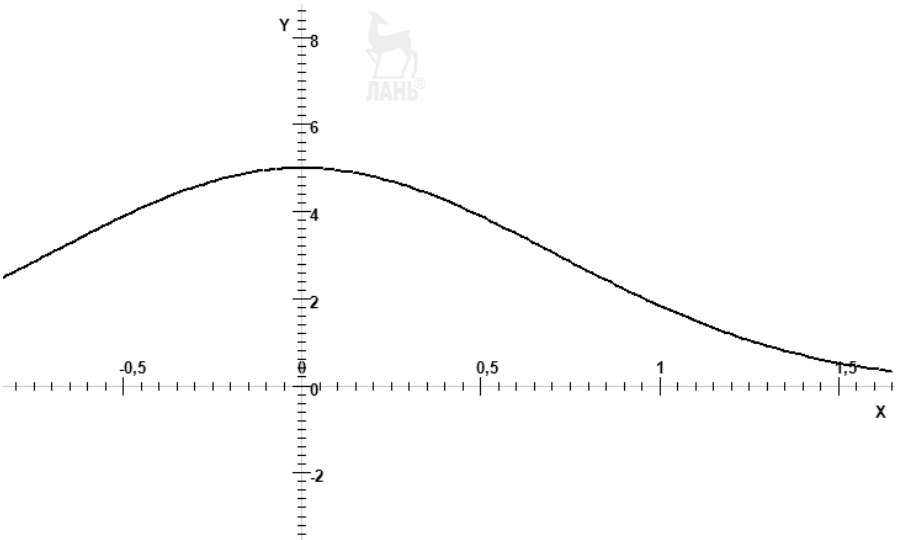

Рис. 1.2. Декартовы координаты

В графических программных средах система координат имеет начало координат в верхнем левом углу, и оси координат направлены направо и *вниз*, как показано на рисунке 1.3.

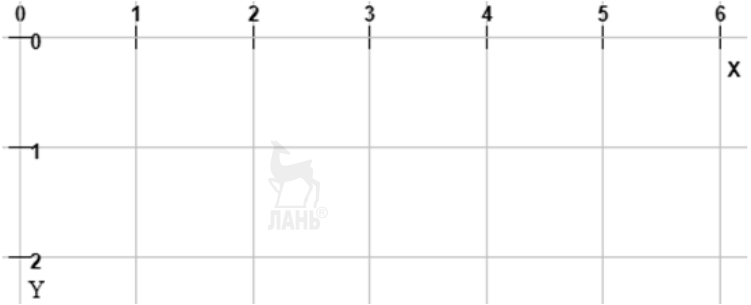

Рис. 1.3. Мировые координаты

Координаты, используемые в методах класса `Graphics`, называют *мировыми координатами*.

1.5. Преобразование координат

Мировые координаты преобразовываются в *координаты страницы мировыми преобразованиями*. Мировые преобразования, например `TranslateTransform()`, `ScaleTransform()` и `RotateTransform()`,

используются для того, чтобы установить центр и поворот системы координат.

Свойство *PageUnit* устанавливает единицу для преобразования и масштабирования рисунков. Эти единицы – одно из возможных *GraphicsUnits*-перечисленных значений, приведенных в таблице 1.4.

Таблица 1.4

Перечисление *GraphicsUnits*

Перечисленная	**Единица измерения**
Display	1/75 дюйма
Document	1/300 дюйма
Inch	1 дюйм
Millimeter	1 миллиметр
Pixel	1 пиксел
Point	1/72 дюйма
World	Мировые единицы

В следующем примере рисуется 6 эллипсов с различными единицами измерения.

Листинг 1.2. Рисование эллипсов с различными единицами измерения

```
private void Draw()
{
    g = CreateGraphics();
    g.Clear(Color.White);

    g.PageUnit = GraphicsUnit.Pixel;
    g.DrawEllipse(Pens.Black, 10, 10, 200, 100);

    g.PageUnit = GraphicsUnit.Display;
    g.DrawEllipse(Pens.Black, 10, 10, 200, 100);

    g.PageUnit = GraphicsUnit.Point;
    g.DrawEllipse(Pens.Black, 10, 10, 200, 100);

    g.PageUnit = GraphicsUnit.Millimeter;
    g.DrawEllipse(Pens.Black, 10, 10, 20, 10);

    g.PageUnit = GraphicsUnit.Document;
    g.DrawEllipse(Pens.Black, 10, 10, 200, 100);

    Pen myPen = new Pen(Color.Black,0.05F);
    g.PageUnit = GraphicsUnit.Inch;
    g.DrawEllipse(myPen, 1, 1, 2, 1);
    g.Dispose();
}
```

Для первых пяти эллипсов используется перо толщиной 1, а для шестого эллипса создается перо толщиной 0,05 дюйма. Результаты работы программы представлены на рисунке 1.4.

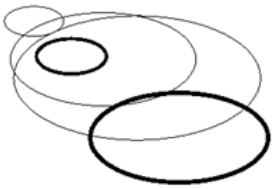

Рис. 1.4. Эллипсы с различными единицами измерения

Отметим, что:

- эллипсы, нарисованные с единицами измерения *Pixel* и *Display*, совпали между собой;
- линия эллипса, нарисованная с единицей измерения миллиметр, имеет толщину 1 мм.

1.6. Графические методы

Класс *Graphics* содержит много методов для рисования линий на графическом устройстве. Имена этих методов начинаются с *Draw*:

DrawArc	*DrawIcon*	*DrawPath*
DrawBezier	*DrawIconUnstretched*	*DrawPie*
DrawBeziers	*DrawImage*	*DrawPolygon*
DrawClosedCurve	*DrawImageUnscaled*	*DrawRectangle*
DrawCurve	*DrawLine*	*DrawRectangles*
DrawEllipse	*DrawLines*	*DrawString*

Графический класс *Graphics* также включает в себя методы заполнения областей, имена которых начинаются с *Fill*:

FillClosedCurve	*FillPie*	*FillRectangles*
FillEllipse	*FillPolygon*	*FillRegion*
FillPath	*FillRectangle*	

Методы *Draw**** рисуют контур фигуры; методы *Fill**** заполняют фигуры цветом и стилем текущей кисти.

Часто используемые методы представлены в таблице 1.5.

Таблица 1.5

Методы класса *Graphics*

Метод	**Описание**
Clear	Очищает полотно рисования и заполняет ее указанным цветом

Продолжение табл. 1.5

Метод	**Описание**
DrawString	Рисует строку, используя установленные кисть и шрифт
DrawLine	Рисует отрезок прямой линии между двумя точками
FillEllipse	Заполняет эллипс, ограниченный прямоугольником
MeasureString	Возвращает размер прямоугольника, в котором нарисована строка.
RotateTransform	Применяет вращение, определенное матрицей трансформации, к мировой системе координат
ScaleTransform	Применяет масштабирование системы координат, определенное матрицей трансформации
TransformPoints	Трансформирует массив точек, используя матрицу трансформации
TranslateTransform	Применяет смещение графического объекта, определенное матрицей трансформации

Глава 2. ПРОСТРАНСТВО ИМЕН SYSTEM.DRAWING

Пространство имен `System.Drawing` содержит в себе пять пространств имен.

1. Пространство имен `Design` содержит классы, которые расширяют логические и графические возможности интерфейса пользователя (UI). Классы этого пространства имен используются для создания настраиваемой панели элементов; редакторов значений определенных типов, которые могут редактировать и графически представлять значения поддерживаемых типов; преобразователей, которые могут преобразовывать значения определенных типов. Это пространство имен предоставляет базовую архитектуру для создания расширений пользовательского интерфейса времени разработки.
2. Пространство имен `Drawing2D` предоставляет расширенные функциональные возможности векторной и двумерной графики. В этом пространстве содержатся классы графики и графических контуров; типы, относящиеся к матрице преобразования; классы *Brush*; перечисления, связанные с линиями; перечисления, связанные с заполнением фигур и контуров.
3. Пространство имен `Imaging` расширяет функциональные возможности визуализации изображения, реализованные в пространстве имен *System.Drawing*:
   - класс *Metafile* содержит методы для записи метафайлов;
   - класс *Encoder* предоставляет возможность расширить GDI+ для поддержки любого формата изображений;
   - класс *PropertyItem* содержит методы для сохранения метафайлов в файлах изображений и их извлечения.
4. Пространство имен `Printing` обеспечивает функции, связанные с печатью.
5. Пространство имен `Text` предоставляет расширенные функциональные возможности для работы с текстом.

Так как базовые графические средства реализованы в пространстве имен `System.Drawing`, то изучение графических свойств и методов начнем с этого пространства имен.

2.1. Создание и удаление объекта класса Graphics

Класс *Graphics* содержит поверхность рисования. Существуют три вида поверхностей рисования:

- окна и компоненты;
- страницы, отправляемые на принтер;
- битовые карты и изображения в памяти.

Класс *Graphics* предоставляет методы для рисования на любой из этих поверхностей рисования. Его можно использовать для рисования дуг, кривых, кривых Безье, эллипсов, изображений, линий, прямоугольников и текста.

2.1.1. Создание поверхности рисования

Объект *Graphics* для формы можно получить минимум четырьмя различными способами.

Первый способ. Вызов метода рисования в обработчике события *Paint()*:

```
private void FormMain_Paint(object sender, PaintEventArgs e)
{
    Draw(e.Graphics);
}
```

Свойство *e.Graphics* типа *Graphics* показывает на полотно рисования *FormMain*.

Второй способ. Переопределение метода *OnPaint()*. Класс *Form* наследует метод *OnPaint()* из класса *Control*, и этот метод является обработчиком события *Paint*, которое генерируется при каждом перерисовывании элемента управления. Объект *Graphics* можно получить из класса *PaintEventArgs*, который передается событием:

```
protected override void OnPaint(PaintEventArgs e)
{
     Graphics g = e.Graphics;
     // рисование
}
```

В других случаях может потребоваться выполнение рисования непосредственно в окне, не дожидаясь генерации события *Paint*. Такая ситуация может возникать для устранения мерцания при создании кода для выбора графического объекта в окне или перетаскивании объекта мышью.

Третий способ. Объект *Graphics* можно получить, вызывая в форме метод *CreateGraphics()*:

```
Graphics g = this.CreateGraphics();
// рисование
g.Dispose();
```

В этом случае рисование будет происходить на форме. Для рисования на любом другом элементе, например на панели *panel1*, необходимо немного изменить код:

```
Graphics g = panel1.CreateGraphics();
```

Четвертый способ. При выборе графического объекта в окне или перетаскивании объекта мышью может возникать мерцание. Для устранения мерцания необходимо рисовать в памяти на *bitmap*, а затем копировать *bitmap* на полотно рисования формы:

```
Graphics gScreen;

public FormMain()
{
    InitializeComponent();
    gScreen = this.CreateGraphics();
}

public void Draw()
{
   Bitmap bitmap = new Bitmap(Width, Height);
   using (Graphics g = Graphics.FromImage(bitmap))
   {
      Draw(g);
      gScreen.DrawImage(bitmap, ClientRectangle);
   }
   bitmap.Dispose();
}
```

2.1.2. Удаление объектов рисования

При разработке приложений, использующих GDI+, приходится иметь дело с несколькими типами данных, для которых важно вызывать метод *Dispose()*, иначе некоторые ресурсы не будут освобождены. Например, классы *Graphics*, *Pen*, *Brush*, *Font* реализуют интерфейс *IDisposable*.

При получении объекта *Graphics* из метода *OnPaint()* или *Paint()* он не создается, поэтому метод *Dispose()* вызывать не надо. Но при вызове метода *CreateGraphics()* вызов метода *Dispose()* обязателен, если объект *Graphics* описан и создается в методе рисования.

Для освобождения памяти можно использовать такой код:

```
g = this.CreateGraphics();
try
    {
       // рисование
    }
```

```
finally
{
    if (g != null)
        ((IDisposable)g).Dispose();
}
```

Существует более простой способ работы с объектами, которые необходимо удалять. Для этого предназначена конструкция *using*, которая автоматически вызывает метод *Dispose()*, когда объект выходит из блока *using*. Следующий код демонстрирует применение ключевого слова *using* и полностью эквивалентен предыдущему:

```
using (Graphics g = this.CreateGraphics() )
{
     // рисование
}
```

Создание и удаление объектов рисования занимает определенное время, потеря которого заметна при частой прорисовке, например при перемещении мыши или создании мульпликаций. Поэтому графические объекты можно создать один раз, например в конструкторе *FormMain()* или в методе *FormMain_Load()*. Тогда нет необходимости после завершения рисования удалять объекты рисования.

2.1.3. Порядок выполнения действий при рисовании

1. Описать поле *g* класса *Graphics* в классе *FormMain*.
2. Создать метод рисования *Draw*():

```
private void Draw(Graphics g)
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);
    // рисование типа g.DrawLine()
}
```

3. Создать метод *onPaint()*, в котором направить *g* на *e.Graphics* и вызвать метод *Draw()*:

```
private void FormMain_Paint(object sender, PaintEventArgs e)
{
    g = e.Graphics;
    Draw(g);
}
```

2.2. Структуры представления координат

Для указания координат при рисовании используются три структуры: *Point* (точка), *Size* (размер) и *Rectangle* (прямоугольник).

Структуры *Point* или *PointF* используются для представления точки, имеющей целочисленные или вещественные координаты *X* и *Y*. Они представляют собой точку на двумерной плоскости, то есть координаты пикселя. Методы, такие как *DrawLine()*, *DrawRectangle()*, принимают *Point* в качестве аргумента. Создание структуры *Point* выполняется следующим образом:

```
Point p = new Point (1, 1);
```

Для получения координат *X* и *Y* структуры *Point* используют свойства *X* и *Y*. Эти свойства объявлены только для чтения. Поэтому конструкция *p.X = 2*. Изменения следует производить так: *p = new Point (2, 1)*.

Структуры *Size* и *SizeF* используются для представления размера в пикселях. Структура содержит свойства ширину *Width* и высоту *Height*. Создание структуры *Size* выполняется следующим образом:

```
Size s = new Size (12, 5);
```

Структуры *Rectangle* и *RectangleF* используются для указания координат прямоугольника. Свойство *Point* определяет верхний левый угол прямоугольника, а свойство *Size* — его размеры. Существует по два конструктора структур *Rectangle* и *RectangleF*. Один принимает в качестве аргументов позицию *X*, позицию *Y*, ширину и высоту. Второй принимает свойства типа *Point* и *Size*. Ниже приведены два примера объявления и конструирования структуры *Rectangle*:

```
Rectangle rl = new Rectangle (2, 2, 5, 6);
Point p = new Point (2, 2);
Size s = new Size (5, 6);
Rectangle r2 = new Rectangle (p, s);
```

Для получения и установки расположения и размеров структуры *Rectangle* предусмотрены некоторые свойства. Другие полезные свойства и методы для решения таких задач:

- определение принадлежности точки прямоугольнику;
- пересекается ли прямоугольник с другим прямоугольником;
- пересечение, вычитание, объединения двух прямоугольников.

2.3. Классы Color, Pen, Brush, Font

Перо и кисть ипользуются для отображения линий, фигур и текста. *Перо* является экземпляром класса *Pen* и назначается при рисования линий и сложных фигур. *Кисть* — экземпляр класса *Brush* и используется при заполнении фигур или для рисования текста. Объекты *Color* используются перьями и кистями для указания цвета.

2.3.1. Класс Color

Для работы с цветами пространство имен *System.Drawing* содержит следующие классы и структуры:

- *Color* – структура представляет цвета в виде каналов альфа (прозрачность), красного, зеленого и синего (ARGB). Каждый канал – это целое от 0 до 255;
- *KnownColor* – перечислимый тип. Содержит системные цвета;
- *SystemColors* – свойства этого класса являются структурами *Color*, которые представляют собой цвета Windows.

Структура *Color* используется для представления различных цветов. Цвета используются перьями и кистями для указания цвета при визуализации.

Существует около 150 системных цветов, которые могут быть доступны через структуру *Color*. Например:

```
Color myColor;
myColor = Color.Red;
myColor = Color.Aquamarine;
myColor = Color.LightGoldenrodYellow;
myColor = Color.PapayaWhip;
myColor = Color.Tomato;
```

Можно также создавать определяемые цвета с помощью метода *Color.FromArgb()*. Этот метод позволяет указать долю отдельных составляющих компонентов красного, синего и зеленого цветов:

```
Color myColor;
myColor = Color.FromArgb(23,56,78);
```

Каждый компонент должен быть целым числом от 0 до 255, где 0 означает отсутствие цвета, а 255 – полное насыщение указанного цвета. Например:

```
Color.FromArgb(0,0,0) - черный,
Color.FromArgb(255,255,255) - белый.
```

С помощью метода *FromArgb()* можно назначить компонент *альфа*. *Альфа* определяет прозрачность объектов за рисуемым графическим объектом. *Альфа*-компонент – первый из четырех аргументов в методе *Color.FromArgb()* и может быть целым числом в диапазоне от 0 до 255:

```
Color myColor;
myColor = Color.FromArgb(127, 23, 56, 78);
```

Можно также создать цвет с *альфа*-компонентом, назначив *альфа*-компонент для ранее определенного цвета:

```
Color myColor;
myColor = Color.FromArgb(128, Color.Red);
```

Модель цветов RGB не единственно возможная. Еще одна модель представления цвета — разбиение его на три компонента: оттенок,

насыщенность и яркость (HSB). Структура *Color* содержит методы для выполнения этой задачи:

- *GetHue()* — возвращает значение оттенка;
- *GetSaturation()* — возвращает значение насыщенности;
- *GetBrightness()* — возвращает значение яркости.

Более подробно о моделях цветов поговорим в пункте 2.8.

2.3.2. КЛАСС PEN

Перо назначается для рисования линий, кривых и фигур. В таблице 2.1 приведены классы перьев.

Таблица 2.1

Классы перьев

Класс	**Описание**
Pen	Определяет объект, используемый для рисования линий
Pens	Класс для стандартных цветов
SystemPens	Свойства этого класса являются объектами *Pen*, то есть цветом элемента изображения Windows толщиной 1

Класс *Pen* имеет 4 конструктора:

- *Pen(Brush brush);*
- *Pen(Color color);*
- *Pen(Brush brush, float width);*
- *Pen(Color color, float width).*

Экземпляр класса после завершения работы с ним необходимо уничтожить методом *Dispose()*, который освобождает все ресурсы, используемые данным объектом *Pen*.

Перо обладает следующими свойствами:

- *Brush* – получает или задает объект *Brush*;
- *Color* – получает или задает цвет пера;
- *float Width* – получает или устанавливает толщину пера *Pen* в единицах объекта;
- *PenType* – получает или задает стиль линий;
- *DashStyle* – задает пользовательский стиль пунктирных линий.

Свойство *PenType* перечислимого типа задает вид заполнения, используемый объектом *Pen*, и может принимать следующие значения:

- *SolidColor* – задает сплошное заполнение;
- *HatchFill* – задает заполнение штриховкой;

– *TextureFill* – задает заполнение с текстурой точечного рисунка;
– *PathGradient* – задает градиентное заполнение контура;
– *LinearGradient* – задает линейное градиентное заполнение.

Свойство перечислимого типа *DashStyle* может принимать следующие значения:

– *Solid* – сплошная линия;
– *Dash* – линия, состоящая из штрихов;
– *Dot* – линия, состоящая из точек;
– *DashDot* – штрих-пунктирная линия;
– *DashDotDot* – линия, состоящая из шаблонов «штрих-две точки»;
– *Custom* – задает произвольный тип пунктирных линий.

В следующем примере создается черное перо:

```
// черное перо толщиной 1
Pen myPen = new Pen(Color.Black);
// черное перо толщиной 5
Pen myPen = new Pen(Color.Black, 5);
```

Ниже показывается создание пера, основанное на существующей кисти с именем *myBrush*:

```
//перо с такими же свойствами, как у кисти myBrush толщиной 1
SolidBrush myBrush = new SolidBrush(Color.Red);
Pen myPen = new Pen(myBrush);
//перо с такими же свойствами, как у кисти myBrush толщиной 5
Pen myPen = new Pen(myBrush, 5);
```

Если перо создано, то его можно использовать для рисования линий. Ниже показано использование пера для рисования эллипса:

```
Pen myPen = new Pen(Color.Black);
g.DrawEllipse(myPen, 20, 30, 10, 50);
```

Для пера можно изменить свойства, влияющие на вид линии. Свойства пера:

– *Color* и *Width* — цвет и толщина;
– *StartCap* и *EndCap* — стандартные (различные стрелки) или пользовательские фигуры в начале или в конце отрезка прямой линии;
– *DashStyle* — сплошная, пунктирная или пользовательская пунктирная линия;
– *DashCap* — позволяет настраивать концы линии;
– *DashPattern* — создает перо произвольного стиля. Этому свойству необходимо передать массив четной длины, в котором задаются длины рисуемых интервалов и пропусков между ними.

Ниже приведен пример (рис. 2.1), демонстрирующий использование этого свойства:

```
Pen p = new Pen(Color.Black, 3);
float[] f = { 15, 5, 10, 5, 1, 1 };
p.DashPattern = f;
g.DrawRectangle(p, 10, 10, 80, 300);
```

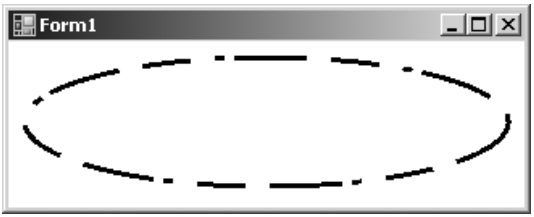

Рис. 2.1. Рисование эллипса пером произвольного стиля

Также у класса *Pen* есть несколько методов, позволяющих изменять матрицу геометрического преобразования объекта *Pen*.

2.3.3. КЛАСС BRUSH

Кисти используются для рисования сплошных фигур и отображения текста. В таблице 2.2 приведены классы кистей, каждый из которых наследуется от класса *Brush*.

Таблица 2.2

Классы кистей

Класс *Brush*	**Описание**
SolidBrush	Сплошная заливка
HatchBrush	Сплошная заливка, позволяет выбрать из представленных шаблонов
TextureBrush	Кисть, использующая изображение
LinearGradientBrush	Рисует двумя цветами, смешивая цвета вдоль указанной линии. В начале линии назначается один цвет, в конце — другой
PathGradientBrush	Задается полигон, в каждой вершине которого назначается свой цвет

Экземпляры класса *Brush* после завершения работы необходимо уничтожать методом *Dispose()*.

2.3.3.1. Класс SolidBrush

У класса *SolidBrush* добавляется только одно свойство *Color*.

Пример, приведенный ниже, показывает, как нарисовать сплошной красный эллипс на форме. Эллипс будет соответствовать размерам прямоугольника, предоставленного для него *ClientRectangle*:

```
Graphics g = this.CreateGraphics();
SolidBrush myBrush = new SolidBrush(Color.Red);
g.FillEllipse(myBrush, ClientRectangle);
myBrush.Dispose();
```

2.3.3.2. Класс HatchBrush

В классе *HatchBrush* добавляется три свойства:

- *Color BackgroundColor* – цвет интервалов между линиями штриховки;
- *Color ForegroundColor* – цвет линий штриховки;
- *HatchStyle HatchStyle* – стиль штриховки.

Перечислимый стиль *HatchStyle* предоставляет несколько десятков различных стилей штриховки.

Класс *HatchBrush* позволяет выбрать стиль кисти из большого разнообразия представленных шаблонов для рисования, а не сплошной цвет. Следующий пример показывает создание *HatchBrush*, который рисует шаблоном *Plaid* (плед), используя красный цвет как *forecolor* и синий – как свойство *backcolor*:

```
HatchBrush aHatchBrush = new HatchBrush(HatchStyle.Plaid,
     Color.Red, Color.Blue);
```

2.3.3.3. Класс TextureBrush

Можно создать кисть *TextureBrush*, содержащую изображение, и рисовать фигуры, которые затем будут залиты этим изображением. На основе текстурной кисти можно создать перо для рисования линий или шрифт для для прорисовки строк.

Класс *TextureBrush* позволяет в качестве кисти использовать изображение, хранящееся в свойстве *Image*. Свойство перечислимого типа *WrapMode* указывает, как накладывается текстура, если она меньше, чем заполняемая область. Это свойство может принимать следующие значения:

- *Tile* – наложение градиента или текстуры;
- *TileFlipX* – перед наложением разворачивает текстуру или градиент горизонтально;
- *TileFlipY* – перед наложением разворачивает текстуру или градиент вертикально;
- *TileFlipXY* – перед наложением разворачивает текстуру или градиент горизонтально и вертикально;
- *Clamp* – не производится наложение текстуры или градиента.

Класс содержит 8 конструкторов:

- *TextureBrush(Image bitmap);*
- *TextureBrush(Image image, Rectangle dstRect);*
- *TextureBrush(Image image, RectangleF dstRect);*
- *TextureBrush(Image image, WrapMode wrapMode);*
- *TextureBrush(Image image, Rectangle dstRect, ImageAttributes imageAttr);*
- *TextureBrush(Image image, RectangleF dstRect, ImageAttributes imageAttr);*
- *TextureBrush(Image image, WrapMode wrapMode, Rectangle dstRect);*
- *TextureBrush(Image image, WrapMode wrapMode, RectangleF dstRect).*

Текстурная кисть заполняет фигуры или текст с помощью изображение как шаблон. Следующий пример показывает, как создать кисть – экземпляр класса *TextureBrush*, которая рисует с помощью изображения из файла с именем *myPic.bmp*:

```
TextureBrush myBrush =
    new TextureBrush(new Bitmap("myPic.bmp"));
```

2.3.3.4. Класс LinearGradientBrush

Класс линейных градиентных заливок *LinearGradientBrush* обладает следующими важными свойствами:

- *Color[] LinearColors* – получает или задает начальный и конечный цвета градиента;
- *Rectangle* – получает прямоугольную область, которая определяет начальную и конечную точки градиента;
- *Matrix Transform* – получает или задает копию объекта *Matrix*, определяющего локальное геометрическое преобразование;
- *WrapMode* – получает или задает перечисление *WrapMode*, определяющее режим переноса объекта.

У этого класса есть несколько методов, позволяющих изменять матрицу геометрического преобразования объекта *LinearGradientBrush*.

2.3.3.5. Класс PathGradientBrush

Класс многоцветной градиентной кисти содержит 5 конструкторов и следующие важные свойства:

- *PointF CenterPoint* – получает или задает центральную точку градиента контура;
- *PointF FocusScales* – получает или задает точку фокуса для градиентного перехода;

– *Color[] SurroundColors* – получает или задает массив цветов, соответствующих точкам контура, заполняемого объектом *PathGradientBrush*;
– *WrapMode* – получает или задает объект, определяющий режим переноса для этого объекта *PathGradientBrush*.

Для работы с градиентными заливками предназначены следующие методы:

– *SetBlendTriangularShape(float focus)* – создает градиент с цветом центра и линейным переходом к одному окружающему цвету;
– *SetBlendTriangularShape(float focus, float scale)* – создает градиент с цветом центра и линейным переходом к каждому из окружающих цветов;
– *SetSigmaBellShape(float focus)* – создает градиентную кисть, которая изменяет цвет, начиная с центра контура и заканчивая границей контура. Переход от одного цвета к другому выполняется на основе колоколообразной кривой;
– *SetSigmaBellShape(float focus, float scale)* – создает градиентную кисть, которая изменяет цвет, начиная с центра контура и заканчивая границей контура. Переход от одного цвета к другому выполняется на основе колоколообразной кривой.

Градиентные кисти обеспечивают поддержку сложных затенений. С помощью *LinearGradientBrush* можно создать гладкую заливку постепенным смешиванием двух цветов вдоль линейного градиента. Следующий пример показывает, как рисовать формы с градиентом, который постепенно смешивает цвета от красного до желтого:

```
Graphics g = this.CreateGraphics();
LinearGradientBrush myBrush =
   new LinearGradientBrush(ClientRectangle,
   Color.Red, Color.Yellow, LinearGradientMode.Vertical);
g.FillRectangle(myBrush, ClientRectangle);
```

PathGradientBrush поддерживает много вариантов более сложных затенений и окраски.

2.3.4. Вывод текста с использованием класса Font

Рассмотрим основные свойства и методы класса *Font*:

– *int Height* – значение междустрочного интервала шрифта в точках;
– *string Name* – имя шрифта;
– *float Size* – размер максимального пробела шрифта в единицах, указанных свойством *Font.Unit*;

– *float SizeInPoints* – размер максимального пробела шрифта (в пунктах);
– *FontStyle Style* – сведения о стиле для шрифта;
– *GraphicsUnit Unit* – единица измерения шрифта.

Перечислимый тип *FontStyle* принимает следующие значения:
– *Regular* – стандартный текст;
– *Bold* – текст, выделенный жирным шрифтом;
– *Italic* – текст, выделенный курсивом;
– *Underline* – текст, выделенный подчеркиванием;
– *Strikeout* – текст с линией посредине.

Перечислимый тип *GraphicsUnit*, задающий единицу измерения, принимает следующие значения:
– *World* – единицу мировой системы координат;
– *Display* – единицу измерения устройства отображения. Обычно это точки для дисплеев и 1/100, 1/300, 1/600, … – дюйма для принтеров;
– *Pixel* – точку устройства;
– *Point* – пункт (1/72 дюйма);
– *Inch* – дюйм;
– *Document* – единицу документа (1/300 дюйма);
– *Millimeter* – миллиметр.

Класс *Font* имеет 13 конструкторов, самый простой из которых создает новый шрифт *Font*, используя размер и стиль:

```
public Font(FontFamily family,float emSize,FontStyle style);
```

Параметры конструктора:
– *family* – семейство шрифтов *FontFamily* или имя шрифта;
– *emSize* – ширина самой широкой буквы нового шрифта (в пунктах);
– *style* – стиль типа *FontStyle* шрифта.

В следующем примере создается экземпляр класса *Font* для прорисовки текста:

```
Font myFont = new Font("Arial", 7, FontStyle.Bold |
FontStyle.Italic);
```

Свойство *Size* представляет размер шрифта. С помощью перечисления *GraphicsUnit* в качестве единицы изменения шрифтов можно указывать одно из следующих значений:
– пункт (1/72 дюйма);
– дисплей (1/72 или 1/100 дюйма);
– документ (1/300 дюйма);
– дюйм;
– миллиметр;
– пиксель.

Выбор единицы изменения при написании программы прорисовки текста, которая должна качественно выводить текст и на дисплеях с очень высоким разрешением, и на дисплеях с низким разрешением, и на принтерах.

Для вычисления размера прямоугольника, в котором будет выведен текст, служит метод *MeasureString()* объекта *Graphics*.

Свойство *Style* определяет вид шрифта: курсивный, полужирный, зачеркнутый или подчеркнутый.

> **Замечание**. Для объектов `Font` необходимо после завершения рвботы с ними вызывать метод `Dispose()` или использовать блок `using`.

2.4. Методы рисования класса Graphics

Методы рисования класса `Graphics` можно разбить на несколько групп:

- *DrawXXX()* – рисование линий;
- *FillXXX()* – рисование областей;
- *DrawString()* – рисование строк;
- *DrawImage()* – копирование изображений.

При рисовании графических объектов используются следующие параметры:

- точка ::= *x,y | Point*;
- прямоугольник ::= *Rectangle | Point,Size | x1, y1, Width, Height*;
- массив точек ::= *Point[]*;
- карандаш ::= *Pen*;
- перо ::= *Brush*;
- шрифт ::= *Font*;

и некоторые другие параметры.

Все методы рисования многократно переопределяются с различными сочетаниями сигнатуры.

2.4.1. Методы рисования линий Draw***()

В эту группу методов входят:

- *DrawArc(Pen pen, Rectangle rect, float startAngle, float sweepAngle)* – дуга, являющаяся частью эллипса, заданного структурой *Rectangle*;
- *DrawBezier(Pen pen, Point pt1, Point pt2, Point pt3, Point pt4)* – кривая Безье, определяемая четырьмя точками типа *Point*;
- *DrawBeziers(Pen pen, Point[] points)* – несколько кривых Безье, определяемых массивом точек типа *Point*;

- *DrawClosedCurve(Pen pen, Point[] points, float tension, FillMode fillmode)* – замкнутая гладкая кривая, определяемая массивом точек типа *Point* с указанным натяжением;
- *DrawCurve(Pen pen, Point[] points, float tension)* – гладкая кривая, проходящая через точки массива *Point[]* с указанным натяжением;
- *DrawEllipse(Pen pen, Rectangle rect)* – эллипс, определяемый ограничивающим прямоугольником типа *Rectangle*;
- *DrawLine(Pen pen, Point pt1, Point pt2)* – линия, соединяющая две точки типа *Point*;
- *DrawLines(Pen pen, Point[] points)* – ломанная линии, определяемая массивом точек типа *Point*;
- *DrawPie(Pen pen, Rectangle rect, float startAngle, float sweepAngle)* – сектор, определяемый эллипсом, заданным прямоугольником типа *Rectangle* и двумя радиальными линиями;
- *DrawPolygon(Pen pen, Point[] points)* – многоугольник, определяемый массивом точек типа *Point*;
- *DrawRectangle(Pen pen, Rectangle rect)* – прямоугольник, определяемый прямоугольником типа *Rectangle*;
- *DrawRectangles(Pen pen, Rectangle[] rects)* – набор прямоугольников, определяемых прямоугольниками типа *Rectangle*.

Для методов этой группы необходимо определить параметр класса *Pen*. Для гладких кривых необходимо назначить параметр вещественного типа *tension* (натяжение), влияющий на вид гладких кривых.

Пример 1. В этом примере рисуется три линии: сплошная линия толщины 4; линия стиля «точка – тире» и линия с ромбом на конце.

Листинг 2.1. Рисование линий разными стилями

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);
    myPen = new Pen(Color.Black,4); // сплошная линия
    g.DrawLine(myPen,20,20,200,20);
    myPen.DashStyle = DashStyle.DashDot; // стиль точка-тире
    g.DrawLine(myPen, 20, 30, 200, 30);
    // линия с ромбом на конце
    myPen.Width = 7;
    myPen.EndCap = LineCap.DiamondAnchor;
    g.DrawLine(myPen, 20, 40, 200, 40);
 }
```

Результаты работы программы представлены на рисунке 2.2.

Рис. 2.2. Рисование линий разными стилями

Пример 2. В этом примере рисуется прямоугольник, эллипс, дуга и сектор.

Листинг 2.2. Рисование прямоугольника, эллипса, дуги и сектора

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);

    myPen = new Pen(Color.Black, 1);
    int dx = 0;
    g.DrawRectangle(myPen,10+dx,10,100,50);
    dx += 120;
    g.DrawEllipse(myPen, 10 + dx, 10, 100, 50);
    dx += 120;
    g.DrawArc(myPen, 10 + dx, 10, 100, 50, 0, 315);
    dx += 120;
    g.DrawPie(myPen, 10 + dx, 10, 100, 50, 0, 315);
}
```

На рисунке 2.3 представлены результаты работы программы.

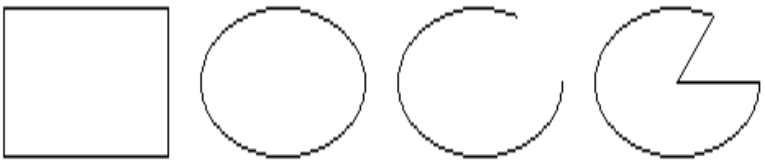

Рис. 2.3. Рисование прямоугольника,
эллипса и дуги

Пример 3. Рисование по массиву точек следующих линий (рис. 2.4):

- ломаная линия;
- линия Безье;
- фундаментальная линия с различными коэффициентами натяжения.

Листинг 2.3. Рисование фигур по массиву точек

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);

    myPen = new Pen(Color.Black, 1);
    Point[] p = {new Point() {X=10,Y=10},
      new Point() {X=15,Y=30},
      new Point() {X=30,Y=50},
      new Point() {X=40,Y=30},
      new Point() {X=60,Y=10},
      new Point() {X=80,Y=30},
      new Point() {X=100,Y=10},
    };
    g.DrawLines(myPen,p);
    g.DrawBeziers(myPen, p);
    // фундаментальная линия, натяжение = 0
    g.DrawCurve(myPen, p, 0);
    // фундаментальная линия, натяжение = 1
    g.DrawCurve(myPen, p, 1);
    // фундаментальная линия, натяжение = 3
    g.DrawCurve(myPen, p, 3);
}
```

Рис. 2.4. Рисование ломаной линии, линии Безье и фундаментальной линии

Пример 4. В следующем коде реализовано построение полигона и фундаментальной замкнутой линии с натяжением 0.5 (рис. 2.5).

Листинг 2.4. Рисование полигона и фундаментальной замкнутой линии

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);
    myPen = new Pen(Color.Black, 1);
    Point[] p = {new Point() {X=20,  Y=60},
      new Point() {X=60,  Y=40},
      new Point() {X=120, Y=20},
```

```
      new Point() {X=200, Y=40},
      new Point() {X=160, Y=60},
      new Point() {X=120, Y=100},
      new Point() {X=40,  Y=80},
     };
   g.DrawPolygon(myPen,p);
   ChangePoint(ref p);
   float tension = 0.5F;
   g.DrawClosedCurve(myPen, p, tension,
     FillMode.Alternate);
}
```

Рис. 2.5. Полигон и фундаментальная замкнутая линия

2.4.2. Методы рисования замкнутых областей Fill***()

Методы этой группы предназначены для рисования замкнутых областей и используют для заливки кисть *brush*:

- *FillClosedCurve(Brush brush, Point[] points, FillMode fillmode, float tension)* – заполняет внутреннюю часть замкнутой гладкой кривой, проходящей через массив точек типа *Point*, используя кисть и натяжение;
- *FillEllipse(Brush brush, Rectangle rect)* – рисует внутреннюю часть эллипса;
- *FillPath(Brush brush, GraphicsPath path)* – рисует внутреннюю часть объекта *GraphicsPath*. Кисть *brush* определяет параметры заливки. Объект *path* представляет контур для заливки;
- *FillPie(Brush brush, Rectangle rect, float startAngle, float sweepAngle)* – заполняет внутреннюю часть сектора, определяемого эллипсом, заданный прямоугольником типа *RectangleF*, и двумя радиальными линиями;
- *FillPolygon(Brush brush, Point[] points, FillMode fillMode)* – заполняет внутреннюю часть многоугольника, определенного массивом точек типа *Point*, используя кисть *brush*;

– *FillRectangle(Brush brush, Rectangle rect)* – заполняет внутреннюю часть прямоугольника, определяемого прямоугольником типа *Rectangle*;
– *FillRectangles(Brush brush, Rectangle[] rects)* – заполняет внутреннюю часть набора прямоугольников, определяемых массивом прямоугольников типа *Rectangle*;
– *FillRegion(Brush brush, Region region)* – заполняет внутреннюю часть объекта *Region*.

Для всех методов этой группы необходимо назначить кисть класса *Brush*. Параметр перечислимого типа *FillMode* указывает, как заполняется внутренняя часть замкнутого контура, и может принимать такие значения:

– *Alternate* – задается режим заполнения с чередованием;
– *Winding* – задается режим заполнения с поворотом.

Пример 5. Использование различных кистей для рисования прямоугольника (рис. 2.6).

Листинг 2.5. Использование различных кистей

```
SolidBrush mySolidBrush;
TextureBrush myBrushTexture;
HatchBrush myHatchBrush;
LinearGradientBrush myBrushGrad;
HatchStyle myHatchStyle;
LinearGradientMode myLinearGradientMode;

private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);
    myPen = new Pen(Color.Black, 1);

    // Рисование сплошной кистью
    int dx = 0;
    mySolidBrush = new SolidBrush(Color.Silver);
    g.FillRectangle(mySolidBrush,20+dx,20,200,100);

    // рисование кистью со стандартными стилями
    dx += 210;
    myHatchStyle = HatchStyle.Cross;
    myHatchBrush = new HatchBrush(myHatchStyle,
        myPen.Color, mySolidBrush.Color);
    g.FillRectangle(myHatchBrush,20+dx,20,200,100);

    // рисование кистью с градиентной заливкой
    dx += 210;
```

```
    myLinearGradientMode =
       LinearGradientMode.Horizontal;
    Rectangle r = new Rectangle(20+dx,20,200, 100);
    myBrushGrad = new LinearGradientBrush(r,
       Color.White,Color.Gray,myLinearGradientMode);
    g.FillRectangle(myBrushGrad, r);

    // рисование текстурной кистью
    dx += 210;
    myBrushTexture =
       new TextureBrush(new Bitmap("T.bmp"));
    g.FillRectangle(myBrushTexture,
        new Rectangle(20 + dx, 20, 200, 100));
}
```

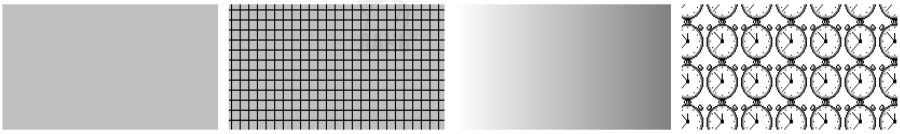

Рис. 2.6. Использование различных кистей

Пример 6. Рисование полигона и фундаментальной линии текстурной кистью (рис. 2.7).

Листинг 2.6. Рисование текстурной кистью

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);
    myPen = new Pen(Color.Black, 1);

    Point[] p = {new Point() {X=20,  Y=60},
        new Point() {X=60,  Y=40},
        new Point() {X=120, Y=20},
        new Point() {X=200, Y=40},
        new Point() {X=160, Y=60},
        new Point() {X=120, Y=100},
        new Point() {X=40,  Y=80},
       };
    myBrushTexture =
        new TextureBrush(new Bitmap("Help.bmp"));
    // полигон
    g.FillPolygon(myBrushTexture, p);
    g.DrawPolygon(myPen, p);

    // замкнутая фундаментальная линия
```

```
    float tension = 0.5F;
    g.FillClosedCurve(myBrushTexture, p,
        FillMode.Alternate, tension);
    g.DrawClosedCurve(myPen, p, tension,
        FillMode.Alternate);
    // фундаментальная линия с поворотом на 90°
    ChangePoint(ref p);
    myBrushTexture.RotateTransform(90);
    g.FillClosedCurve(myBrushTexture, p,
        FillMode.Winding, tension);
    g.DrawClosedCurve(myPen, p, tension,
        FillMode.Alternate);
}
```

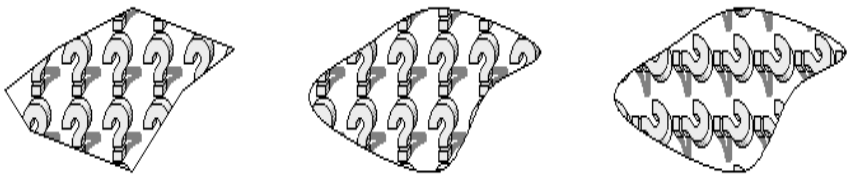

Рис. 2.7. Рисование полигона и замкнутой фундаментальной линии

Пример 7. Использование градиентных кистей при рисовании прямоугольника и эллипса (рис. 2.8).

Листинг 2.7. Использование градиентных кистей

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);

    int W = ClientRectangle.Width;
    int H = ClientRectangle.Height;
    LinearGradientBrush myBrush =
        new LinearGradientBrush(
           new Point(0, 0), new Point(W, H),
           Color.White, Color.Black);
    g.FillRectangle(myBrush, ClientRectangle);
    LinearGradientBrush myBrush2 =
        new LinearGradientBrush(
           new Point(0, 0), new Point(W, H),
           Color.Black, Color.White);
    g.FillEllipse(myBrush2,25,25,W - 50, H - 50);

    myBrush.Dispose();
    myBrush2.Dispose();
}
```

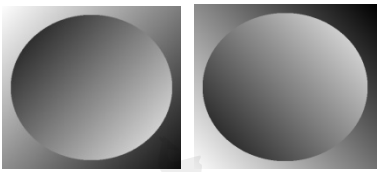

Рис. 2.8. Использование градиентных кистей

Пример 8. Использование многоцветной градиентной кисти. В этом примере мы построим треугольник, у каждой вершины которого будет назначен свой цвет. На результат существенное влияние оказывает цвет центральной точки.

Листинг 2.8. Использование многоцветной градиентной кисти

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);
    Point[] p = {new Point() {X=20,  Y=20},
                 new Point() {X=200, Y=20},
                 new Point() {X=120, Y=150}};
    PathGradientBrush myPathGradientBrush =
        new PathGradientBrush(p);
    Color c = Color.FromArgb(255/4,255/4,255/4);
    myPathGradientBrush.CenterColor = c;
    myPathGradientBrush.SurroundColors =
       new Color[3]
          { Color.Red, Color.Green, Color.Blue };
    g.FillPolygon(myPathGradientBrush, p);
}
```

На рисунке 2.9 представлен результат работы программы при белом, черном цвете и 1/4 белого цвета центральной точки.

Рис. 2.9. Использование многоцветной градиентной кисти

2.4.3. **Вывод строки**

Для рисования строки в классе *Graphics* предусмотрено шесть методов:

- *DrawString(string s, Font font, Brush brush, PointF point);*
- *DrawString(string s, Font font, Brush brush, RectangleF layoutRectangle);*
- *DrawString(string s, Font font, Brush brush, float x, float y);*
- *DrawString(string s, Font font, Brush brush, PointF point, StringFormat format);*
- *DrawString(string s, Font font, Brush brush, RectangleF layoutRectangle, StringFormat format);*
- *DrawString(string s, Font font, Brush brush, float x, float y, StringFormat format).*

Для всех шести методов необходимо назначить следующие параметры: строка, шрифт и кисть. Три метода требуют начальную точку и печатают строку без переноса, начиная от верхнего левого угла, а другие три метода требуют не точку, а прямоугольник. В последнем случае происходит разбиение строки на несколько строк и, если часть строк не помещается в выделенный прямоугольник, то они не печатаются.

При рисовании объект *Rectangle* используют для указания координат границ текста. Желательно, чтобы высота этого прямоугольника была равной высоте шрифта или кратной ей.

Некоторые методы используют параметр *format* типа *StringFormat*, выравнивающий строку относительно прямоугольника. В классе *StringFormat* содержится информация о размещении текста: выравнивании и межстрочном интервале. Некоторые свойства этого класса:

- *StringAlignment* – выравнивание строки текста относительно прямоугольника;
- *FormatFlags* – получает или задает параметры перечислимого типа *StringFormatFlags*, содержащего сведения о форматировании.

Тип *StringAlignment* позволяет принимать такие значения:

- *Near* – текст выравнивается по ближнему краю. При размещении слева направо ближний край – левый. При размещении справа налево ближний край – правый;
- *Center* – текст выравнивается по центру прямоугольника;
- *Far* – текст выравнивается по дальнему краю от положения прямоугольника. При размещении слева направо дальним

положением считается правое. При размещении справа налево дальним положением считается левое.

Пример 9. Выравнивание текста по правому краю и по центру с использованием класса *StringFormat*.

Листинг 2.9. Рисование строки с различными выравниваниями

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);

    int y = 10;
    // Прорисовка текста,
    // выровненного по левому краю.
    Rectangle rect =
       new Rectangle(10, y, 400, Font.Height);
    g.DrawRectangle(Pens.Black, rect);
    g.DrawString("текст, выровненный по левому
        краю", Font, Brushes.Black, rect);
    y += Font.Height + 20;

    // Прорисовка текста,
    // выровненного по правому краю.
    myFont = new Font("Arial", 16,
        FontStyle.Bold | FontStyle.Italic);
    rect = new Rectangle(10, y, 400, myFont.Height);
    g.DrawRectangle(Pens.Black, rect);
    StringFormat sf = new StringFormat();
    sf.Alignment = StringAlignment.Far;
    g.DrawString("текст, выровненный по правому
        краю", myFont, Brushes.Black, rect, sf);
    y += myFont.Height + 20;
    myFont.Dispose();

    // Прорисовка текста, выровненного по центру.
    myFont = new Font("Courier New", 12,
        FontStyle.Underline);
    rect = new Rectangle(10, y, 400, myFont.Height);
    g.DrawRectangle(Pens.Black, rect);
    sf = new StringFormat();
    sf.Alignment = StringAlignment.Center;
    g.DrawString("текст, выровненный по центру",
        myFont, Brushes.Black, rect, sf);
    y += myFont.Height + 20;
    myFont.Dispose();
```

```
    // Прорисовка многострочного текста.
    myFont = new Font("Times New Roman", 12);
    rect = new Rectangle(10,y,120,myFont.Height*4);
    g.DrawRectangle(Pens.Black, rect);
    String s = "Make it run, Make it right,"+
        "Make it small, Make it fast";
    g.DrawString(s, myFont, Brushes.Black, rect);
    myFont.Dispose();
}
```

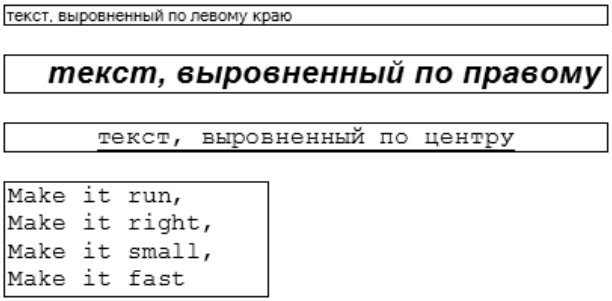

Рис. 2.10. Рисование строк с различным выравниванием

Пример 10. Рисование строки с поворотом (рис. 2.10). Для поворота необходимо использовать два метода класса *Graphics*: метод смещения – *TranslateTransform()* и метод поворота на заданный угол – *RotateTransform()*. После рисования строки необходимо проделать преобразования в обратном порядке.

Листинг 2.10. Рисование строки с поворотом

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);
    myFont = new Font("Arial", 16, FontStyle.Bold);
    g.TranslateTransform(200,200);
    g.RotateTransform(-45);
    g.DrawString("повернутый", myFont,
        Brushes.Black,0,0);
    g.RotateTransform(45);
    g.TranslateTransform(-200, -200);
    g.DrawRectangle(myPen,200,200,4,4);

    g.TranslateTransform(150, 150);
    g.RotateTransform(45);
    g.DrawString("текст",myFont,Brushes.Black,0, 0);
```

```
    g.RotateTransform(-45);
    g.TranslateTransform(-150, -150);
    g.DrawRectangle(myPen, 150, 150, 4, 4);

    myFont.Dispose();
}
```

На рисунке 2.11 прямоугольниками обозначены точки поворота.

Рис. 2.11. Рисование строк с поворотом

2.4.4. Копирование и использование изображений

Копирование реализуется методами *DrawImage()*. Это самое многочисленное семейство методов класса *Graphics*, насчитывающее более 30 перегрузок.

Абстрактный класс *Image* определен в пространстве имен *System.Drawing*. У него есть два потомка: *Bitmap* и *Metafile*. Класс *Bitmap* — общего назначения, обладающий свойствами высоты и ширины. В примерах этого пункта применяется класс *Bitmap*.

> **Замечание**. Для объектов Image необходимо после завершения работы с ними вызывать метод Dispose() или применять блоки using.

Существует несколько возможных источников растровых изображений. Растровое изображение можно:

- загрузить из файла;
- скопировать его из другого существующего изображения;
- создать в виде пустого растрового изображения, в котором можно выполнять рисование.

Изображение в файле может храниться в формате *JPEG, GIF, PNG* или *BMP*.

Выберем файл с помощью элемента *openFileDialog1*, загрузим изображение *Bitmap* из файла и выполним его копирование на поверхность рисования *g* с помощью метода *DrawImage()* в указанный прямоугольник:

```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
{
    Bitmap theImage =
       new Bitmap(openFileDialog1.FileName);
    g.DrawImage(theImage,
```

```
        new Rectangle(10,10,200,200));
    theImage.Dispose();
}
openFileDialog1.Dispose();
```

После копирования необходимо освободить объект *theImage*, хранящийся в переменной-члене класса.

Приложение создаст изображение, показанное на рисунке 2.12.

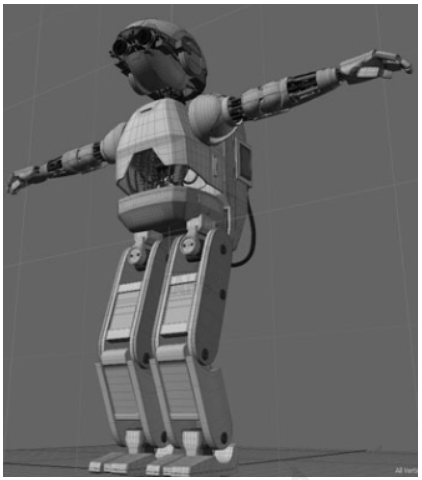

Рис. 2.12. Копирование изображения

Создадим текстурную кисть *TextureBrush* на основе файла *Star.bmp* и рассмотрим три различных примера ее применения:

- рисование эллипса;
- создание пера;
- прорисовка текста.

Чтобы выполнить прорисовку фрагмента текста с использованием кисти *TextureBrush*, выполните следующие действия.

Шаг 1. Начав с кода, созданного в предыдущем примере *DrawImage*, добавьте в класс *Form1* еще одно объявление переменной *Image*:

```
partial class Form1 : Form
{
    private Image theImage;
    private Image smallImage;
}
```

Шаг 2. В методе *OnPaint* на основе объекта *theImg* создадим объект *smallImg*, указав прямоугольник, высота и ширина которого составляют половину от размеров объекта *theImg*:

Листинг 2.11. Рисование с использованием текстурной кисти

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, ClientRectangle);
    theImg = new Bitmap("Star.bmp");
    smallImg = new Bitmap(theImg,
      new Size(theImg.Width/2, theImg.Height/2));
    Brush tBrush = new TextureBrush(smallImg,
      new Rectangle(0,0,smallImg.Width,
                    smallImg.Height));
    g.FillEllipse (tBrush, ClientRectangle);
    tBrush.Dispose();
    theImg.Dispose();
    smallImg.Dispose();
}
```

После завершения рисования освободим память от двух изображений.

Рис. 2.13. Использование текстурной кисти

Отличие от предыдущего примера состоит в том, что мы вызываем метод *FillEllipse()* класса *Graphics*, передавая ему текстурную кисть и *ClientRectangle*.

С помощью объекта *TextureBrush* создадим перо. Изменим метод рисования *OnPaint()* следующим образом.

Листинг 2.12. Рисование с использованием текстурного пера

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g. FillRectangle(Brushes.White,
        ClientRectangle);
```

```
    theImg = new Bitmap("Star.bmp");
    smallImg = new Bitmap(theImg,
      new Size(theImg.Width/2,theImg.Height/2));
    Brush tBrush = new TextureBrush(smallImg,
      new Rectangle(0,0, smallImg.Width,
        smallImg.Height));
    Pen tPen = new Pen (tBrush, 40);
    g.DrawRectangle(tPen,0,0,
      ClientRectangle.Width,ClientRectangle.Height);
    tPen.Dispose();
    tBrush.Dispose();
}
```

При запуске этой программы результат должен выглядеть так, как показано на рисунке 2.14.

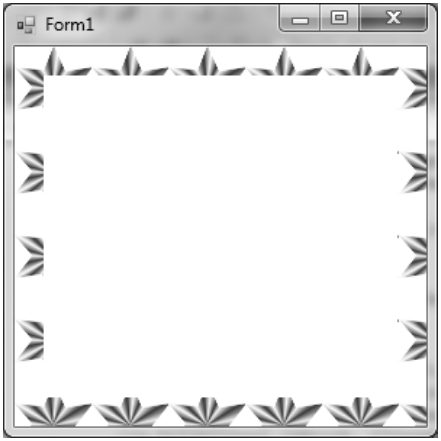

Рис. 2.14. Создание пера на основе изображения

В следующем примере выполним рисование текста, используя созданный объект *TextureBrush*. Изменим метод *OnPaint()*.

Листинг 2.13. Рисование строки с использованием текстурной кисти

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, ClientRectangle);
    theImg = new Bitmap("Star.bmp");
    smallImg = new Bitmap(theImg,
        new Size(theImg.Width/2,theImg.Height/2));
    Brush tBrush =
        new TextureBrush(smallImg,
        new Rectangle (0,0, smallImg.Width,
```

```
        smallImg.Height));
    Font tFont = new Font ("Times New Roman", 32,
        FontStyle.Bold | FontStyle.Italic);
    g.DrawString("Hello, from Beginning Visual",
        tFont, tBrush, ClientRectangle);
    tBrush.Dispose();
    tFont.Dispose();
}
```

При запуске проекта на выполнение на экране должно появиться изображение, показанное на рисунке 2.15.

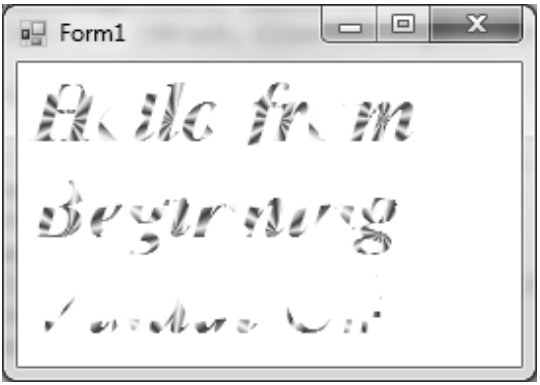

Рис. 2.15. Прорисовка текста с использованием изображения

Метод *DrawString()* в качестве аргументов принимает строку, шрифт, текстурную кисть и ограничивающий прямоугольник.

2.4.5. РАСТРОВОЕ РИСОВАНИЕ ЛИНИЙ

Существует два режима рисования линий: без сглаживания и со сглаживанием. В рисовании без сглаживания все пиксели имеют один и тот же цвет и линия рисуется ступенчато. При рисовании со сглаживанием некоторые пиксели будут закрашиваться частично, позволяя избавиться от ступенчатого представления наклонных линий. Однако для более качественного рисования требуется больше времени для рисования.

На рисунке 2.16 представлена линия, нарисованная без использования сглаживания, и та же линия, нарисованная с использованием сглаживания.

Рис. 2.16. Линия без сглаживания и со сглаживанием

Свойство перечислимого типа *SmoothingMode* класса *Graphics* возвращает или задает качество визуализации графического объекта и может принимать следующие значения:

- *Invalid* – недопустимый режим;
- *Default* – нет сглаживания;
- *HighSpeed* – высокая скорость: нет сглаживания;
- *HighQuality* – рисование со сглаживанием;
- *None* – нет сглаживания;
- *AntiAlias* – рисование со сглаживанием.

Ниже представлен пример, в котором рисуется эллипс без сглаживания, со сглаживанием *HighQuality* и со сглаживанием *AntiAlias*.

Листинг 2.14. Рисование эллипса с различными сглаживаниями

```
private void Draw()
{
    g = CreateGraphics();
    g.Clear(Color.White);

    Pen myPen = new Pen(Color.Black, 0.5F);
    g.PageUnit = GraphicsUnit.Millimeter;
    g.DrawEllipse(myPen, 10, 10, 30, 30);
    g.SmoothingMode = SmoothingMode.HighQuality;
    g.DrawEllipse(myPen, 10 + 35, 10, 30, 30);
    g.SmoothingMode = SmoothingMode.AntiAlias;
    g.DrawEllipse(myPen, 10 + 70, 10, 30, 30);
}
```

Результат работы программы представлен на рисунке 2.17.

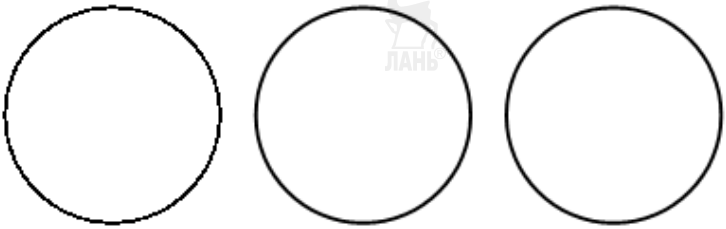

Рис. 2.17. Рисование эллипса со сглаживанием

Отличия между сглаживанием *HighQuality* и сглаживанием *AntiAlias* в глаза не бросаются. Однако два последних эллипса существенно отличаются от первого.

2.4.6. АЛГОРИТМЫ РИСОВАНИЯ ЛИНИЙ

К алгоритмам рисования линий предъявляются высокие требования:

- должны работать быстро;
- не должны потреблять большие объёмы оперативной памяти;
- не использовать медленную вещественную арифметику;
- для каждого пиксела производить как можно меньше вычислений;
- линия должна быть связным объектом, то есть его пикселы должны располагаться рядом друг с другом.

Рассмотрим наиболее известные алгоритмы растеризации отрезка: рисование DDA-линии, алгоритм Брезенхема и алгоритм Ву.

2.4.6.1. Алгоритм DDA

Самым простым и очевидным способом создания алгоритма рисования отрезка от точки (*x0;y0*) до точки (*x1;y1*) является использование параметризированного уравнения прямой линии

$$x(t) = x0 + t * \Delta x;$$
$$y(t) = y0 + t * \Delta y. \quad (2.1)$$

В этом уравнении (*x(t); y(t)*) является точкой отрезка, а значение *t* пробегает интервал от 0 до *L*. Значения *L,x,y* определяются по формулам

$$L = max(|x0 .. x1|; |y0 .. y1|);$$
$$x = x1..x0;$$
$$y = y1..y0. \quad (2.2)$$

Все числа являются вещественными, и для работы с ними используется вещественная арифметика. При отрисовке каждого пиксела значения *x(t)* и *y(t)* округляются и дают его целочисленные координаты. Именно поэтому алгоритм сокращённо называется DDA от английского выражения Digital Differential Analyzer (цифровой дифференциальный анализатор).

Листинг 2.15. Алгоритм DDA

```
private void DDA(int x1, int y1, int x2, int y2)
{
  float L=Math.Max(Math.Abs(x2-x1),Math.Abs(y2-y1));
  float dx=(x2-x1)/L;
  float dy = (y2 - y1) / L;
  float x = x1;
  float y = y1;
  for (int i=0; i<=L; i++)
  {
     x+=dx;
     y+=dy;
     g.FillRectangle(Brushes.Black,x,y,1,1);
  }
}
```

Рис. 2.18. Отрезки, построенные по алгоритмам DDA, Брезенхема и Ву

2.4.6.2. Алгоритм Брезенхема для отрезка прямой

Проблема корректной реализации растрового представления отрезка прямой линии была решена Дж. Брезенхемом (Jack E. Bresenham). В 1962 году им был разработан соответствующий алгоритм, первоначально предназначавшийся для графопостроителей, который впоследствии стал использоваться и для дисплеев.

Алгоритм Брезенхема позволяет получить непрерывный набор пикселей, представляющих отрезок прямой, проведенной между двумя точками *(X1, Y1)* и *(X2, Y2)*. Алгоритм работает следующим образом: для следующего пикселя одна координата *x* или *y* изменяется на 1, а другая – или не изменяется, или изменяется на ±1 в зависимости от расстояния точки линии до ближайшего узла координатной сетки. Алгоритм выбирает вариант для вычисления второй координаты в зависимости от знака отклонения точки линии от ближайшего узла.

В качестве независимой координаты можно выбрать *X* или *Y*, и этот выбор зависит от углового коэффициента прямой *k*:

$$k = dY / dX = (Y_2 - Y_1)/(X_2 - X_1)\,.$$

Для линий, у которых $|k|<1$, в качестве независимой следует выбирать координату *x*, а для линий, у которых $|k|>1$, – координату *y*. Будем считать, что $|k|<1$ и независимо изменяемой является координата *x*.

Запишем уравнение прямой, проходящей через 2 точки:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1), x \in [x_1, x_2]\,. \qquad (2.3)$$

В качестве очередного выбирается пиксель $P(x_0, y_0)$, как это показано на рисунке 2.19.

Следующим пикселем может быть или *Q1*, или *Q2*. Выбор *Q1* или *Q2* можно определить с помощью серединной точки. Если отрезок проходит выше серединной точки, то *Q2*, иначе – пиксель *Q1*. Введем функцию:

$$F(x, y) = (x - X_1) \times dY - (y - Y_1) \times dX. \qquad (2.4)$$

Знак функции *F(x,y)* определяет расположение точки *(x,y)* относительно прямой по правилам:

– $F(x, y) = 0$, точка (x, y) на отрезке;
– $F(x, y) > 0$, точка (x, y) ниже отрезка;
– $F(x, y) < 0$, точка (x, y) выше отрезка.

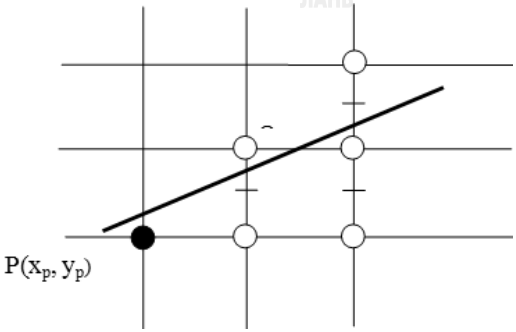

Рис. 2.19. Следующая точка линии

Координаты серединной точки равны:

$$x_p + 1, y_p + 1/2,$$

а значение функции F в серединной точке:

$$d = F(x_p + 1, y_p + 1/2). \quad (2.5)$$

Если выбран пиксель $Q1$, то значение функции в новой серединной точке будет равно:

$$d' = F(x_p + 2, y_p + 1/2) \quad (2.6)$$

и приращение функции

$$\Delta d = d' - d = dY. \quad (2.7)$$

При выборе пикселя $Q2$ значение функции в серединной точке будет равно:

$$d' = F(x_p + 2, y_p + 3/2), \quad (2.8)$$

а приращение функции

$$\Delta d = d' - d = dY - dX. \quad (2.9)$$

Приращения (2.7) и (2.9) не зависят от координат (X, Y) и определяются только dX и dY. Выберем в качестве начальной точку $(X1, Y1)$. Тогда из (2.5) для исходного значения d получим

$$d_0 = F(X_1 + 1, Y_1 + 1/2) = dY - dX/2. \quad (2.10)$$

Знак d_0 влияет на выбор следующего пикселя. Добавляя Δd к d_0, получаем следующее значение d.

Для исключения из алгоритма появления вещественных значений введем множитель 2 для величины d, переопределив ее так:

$$d = 2 * F(x_p + 1, y_p + 1/2). \quad (2.11)$$

В листинге 2.16 приведен алгоритм Брезенхема для отрезка прямой.

Листинг 2.16. Алгоритм Брезенхема для отрезка прямой

```
private void Bresenham(int x1,inty1, int x2, int y2)
{
    int x = x1; int y = y1;
    int Dx = x2 - x1; int Dy = y2 - y1;
    int e = 2 * Dy - Dx;
    for (int i = 1; i <= Dx; i++)
    {
        g.FillRectangle(Brushes.Black, x, y, 1, 1);
        if (e >= 0)
        {
            y++;
            e += -2 * Dx + 2 * Dy;
        }
        else
            e += 2 * Dy;
        x++;
    }
}
```

2.4.6.3. Алгоритм By

Недостатком алгоритма Брезенхема является ступенчатость отрезка, которая особенно хорошо заметна при низкой разрешающей способности растра. Для борьбы необходимо учесть два важных условия. Во-первых, может требоваться, чтобы производилось сглаживание только растеризуемого отрезка, а остальные части изображения не изменялись. Во-вторых, алгоритм сглаживания должен работать быстро.

Алгоритм By является модификацией алгоритма Брезенхема и производит отрисовку сглаженных отрезков. Модификация заключается в том, что на каждом шаге происходит закрашивание не одного, а сразу двух пикселов.

Интенсивность закраски каждого пиксела зависит от его расстояния до прямой и вычисляется на основе значения модуля ошибки, при этом суммарная интенсивность двух закрашиваемых пикселов должна быть равна единице. Если величина ошибки меньше нуля, то закрашиваются текущий и верхний пиксел, поскольку реальная прямая проходит между ними. Если же величина ошибки больше нуля, то закрашиваются текущий и нижний пикселы.

Листинг 2.17. Алгоритм By

```
private void Wu(int x1, int y1, int x2, int y2)
{
    int x = x1;
int y = y1;
int Dx = x2 - x1;
```

```
int Dy = y2 - y1;
int e = 2 * Dy - Dx;
float d;
SolidBrush b1, b2;
for (int i = 1; i <= Dx; i++)
{
    d = -1F * e / (Dy + Dx) / 1.15F;
    if (e >= 0)
    {
        b1 = new SolidBrush(SetColor(1F / 2 - d));
        b2 = new SolidBrush(SetColor(1F / 2 + d));
        g.FillRectangle(b1, x, y, 1, 1);
        g.FillRectangle(b2, x, y + 1, 1, 1);
        y++;
        e += -2 * Dx + 2 * Dy;
    }
    else
    {
        b1 = new SolidBrush(SetColor(1F / 2 + d));
        b2 = new SolidBrush(SetColor(1F / 2 - d));
        g.FillRectangle(b2, x, y, 1, 1);
        g.FillRectangle(b1, x, y - 1, 1, 1);
        e += 2 * Dy;
    }
    x++;
    b1.Dispose();
    b2.Dispose();
}
```

Результат работы программы представлен на рисунке 2.18. В этой программе на каждом шаге цикла приходится создавать новую кисть, цвет которой зависит от ошибки, с помощью метода *SetColor()*:

```
private Color SetColor(float t)
{
    int c = Convert.ToInt32 (255*t);
    Color res = Color.FromArgb(c, c, c);
    return res;
}
```

2.5. Класс путей GraphicsPath и регионы

В интерфейсе GDI+ существует два класса данных, которые используются в качестве аргументов различных методов рисования.

2.5.1. Пути

Класс *GraphicsPath* представляет серию линий и кривых. При конструировании пути к нему можно добавлять замкнутые линии, кривые Безье, дуги, сектора, многоугольники, прямоугольники и другие элементы.

Путь можно рисовать посредством вызова метода *DrawPath()*. Для пути можно выполнить заливку, вызывая метод *FillPath()*.

Класс содержит шесть конструкторов типа:

```
public GraphicsPath(Point[] p, byte[] types,
    FillMode fillMode)
```

где

p – массив точек типа *Point*, составляющих этот объект;

types – массив элементов перечисления *PathPointType*, определяющий тип соответствующей точки в массиве *p*;

fillMode – перечисление *FillMode*, определяющее заполнение внутренней области.

Метод *StartFigure()*, открывающий новую фигуру. Все последующие точки, добавляемые к контуру, добавляются к этой новой фигуре.

Свойства класса:

- *FillMode FillMode* – перечисление типа *FillMode*, определяющее заполнение внутренней области;
- *PathData PathData* – объект *PathData*, содержащий массивы точек (*points*) и типов (*types*);
- *PointF[] PathPoints* – точки контура;
- *byte[] PathTypes* – типы соответствующих точек в массиве *PathPoints*;
- *int PointCount* – число элементов в массиве *PathPoints* и *PathTypes*.

Свойство *FillMode* может принимать следующие значения:

- *Alternate* – режим заполнения с чередованием;
- *Winding* – режим заполнения с поворотом.

Класс *PathData* содержит следующие методы и свойства:

- *PathData()* – конструктор;
- *PointF[] Points* – массив точек контура типа *PointF*;
- *byte[] Types* – типы соответствующих точек контура.

Перечислимый тип *PathPointType* может принимать следующие значения:

- *Start* – начальная точка объекта;
- *Line* – прямая;
- *Bezier3* – кубическая линия Безье;
- *Bezier* – линия Безье;
- *PathTypeMask* – точка маски;
- *DashMode* – сегмент является пунктирным;
- *PathMarker* – маркер контура;
- *CloseSubpath* – конечные точки субконтура.

В классе есть 45 методов добавления различных линий типа добавления дуги:

```
AddArc(Rectangle rect, float startAngle, float sweepAngle),
```

которые присоединяют дугу эллипса или другие фигуры к текущей фигуре.

Необходимо отметить две важные группы методов, которые проверяют принадлежность точки контуру:

- *bool IsOutlineVisible(Point point, Pen pen)* – указывает, содержится ли точка внутри контура при его отображении с помощью указанного карандаша *Pen*;
- *bool IsVisible(Point point)* – указывает, содержится ли точка внутри этого объекта.

Другие вспомогательные методы:

- *CloseAllFigures()* – замыкает все незамкнутые фигуры, соединяя начальную и конечную точки линией, и открывает новую фигуру;
- *CloseFigure()* – замыкает текущую фигуру и открывает новую фигуру;
- *Flatten()* – преобразует каждую кривую в контуре в ломаную линию;
- *RectangleF GetBounds()* – прямоугольник, ограничивающий объект *GraphicsPath*;
- *PointF GetLastPoint()* – получает последнюю точку массива *PathPoints* для объекта *GraphicsPath*.

2.5.2. **Области**

Класс *Region* (область) — графическая форма, состоящая из прямоугольников и путей; предназначен для реализации бинарных операций с областями. Область можно нарисовать, используя метод *FillRegion()*.

Класс *Region*, кроме многочисленных конструкторов, содержит 6 важных групп методов, одна из которых проверяет принадлежность точки региону, а остальные реализуют различные бинарные операции над областями. Для области *A* класса *Region* доступны следующие методы:

- *bool IsVisible(Point point)* – проверяет, содержится ли точка *point* в области *A*;
- *Intersect(Region B)* – определяет пересечение с областью *B* *(A∩B)*;
- *Union(Region B)* – определяет объединение с областью *B* *(A∪B)*;

– *Xor(Region B)* – определяет объединение и пересечение с областью *B* (*A xor B = (A \ B) ∪ (B \ A) = (A∪B) \ (A∩B)*);
– *Complement(Region B)* – заменяет область *A* разностью *B\A*;
– *Exclude(Region B)* – заменяет область *A* разностью *A\B*.

Области конструируются из прямоугольников и путей до вызова любого бинарного метода.

Пример 11. В этом примере создается две области: *regionA* на основе пути, *regionB* – на основе прямоугольника. Затем к этим областям применяются бинарные операции.

Листинг 2.18. Бинарные операции с областями

```
private void Draw()
{
    Color cl = Color.FromArgb(255, 255, 255);
    g.Clear(cl);
    myPen = new Pen(Color.Black, 1);
    GraphicsPath path;
    Point[] p =     {
     new Point() {X=20,  Y=60},
     new Point() {X=60,  Y=40},
     new Point() {X=120, Y=20},
     new Point() {X=200, Y=40},
     new Point() {X=160, Y=60},
     new Point() {X=120, Y=100},
     new Point() {X=40,  Y=80},
    };
    byte[] types = {
     (byte)PathPointType.Start,
     (byte)PathPointType.Line,
     (byte)PathPointType.Line,
     (byte)PathPointType.Line,
     (byte)PathPointType.Line,
     (byte)PathPointType.Line,
     (byte)PathPointType.Line
    };
    path = new GraphicsPath(p,types);
    path.CloseFigure();
    Region regionA = new Region(path);
    Rectangle r1 = new Rectangle(70,40,60,100);
    Region regionB = new Region(r1);

    regionA.Union(regionB);      // A+B
    regionA.Complement(regionB); // B-A
    regionA.Exclude(regionB);    // A-B
```

```
    regionA.Intersect(regionB);  // A*B
    regionA.Xor(regionB);// A xor B

    mySolidBrush = new SolidBrush(Color.Silver);
    g.FillRegion(mySolidBrush, regionA);
    g.DrawPath(myPen, path);
    g.DrawRectangle(myPen,r1);
}
```

На рисунке 2.20 серым цветом обозначена результирующая область, а линиями – исходные области.

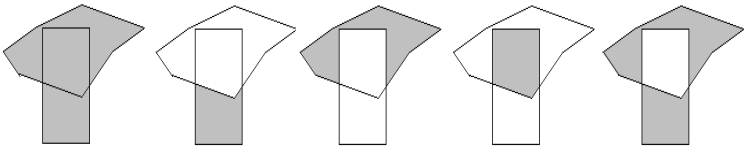

Рис. 2.20. Бинарные операции с областями

2.6. ДВОЙНАЯ БУФЕРИЗАЦИЯ С ПОМОЩЬЮ BITMAP

Если изображение состоит из большого числа графических элементов, то рисование на экране может занять заметное время, что приводит к мерцанию. Для уменьшения мерцания можно использовать свойство формы логического типа *DoubleBuffered*, которое возвращает или задает значение, указывающее, должна ли поверхность этого элемента управления перерисовываться с помощью дополнительного буфера. Однако заметного эффекта такой метод не дает.

Гораздо более эффективно применять другую технологию: вместо прорисовки непосредственно в окне делать прорисовку в памяти на элементе класса *Bitmap*, по завершении которой выполняется копирование изображения в окно. Эта технология устраняет эффект мерцания.

Рассмотрим пример использования двойной буферизации.

Листинг 2.19. Использование двойной буферизации

```
public partial class FormMain : Form
{
    Graphics g0;
    Bitmap bitmap;
    public FormMain()
    {
        InitializeComponent();
        bitmap =
           new Bitmap(ClientSize.Width,
```

```
                ClientSize.Height);
        g0 = CreateGraphics();
    }

    public void Draw()
    {
      using (Graphics g=Graphics.FromImage(bitmap))
        {
            Color cl = Color.FromArgb(255,255, 255);
            g.Clear(cl);
            // рисование
        }
        g0.DrawImage(bitmap, ClientRectangle);
    }
    private void FormMain_Paint(object sender,
        PaintEventArgs e)
    {
        Draw();
    }
}
```

В этом примере используются две поверхности рисования класса *Graphics*: поверхность *g0* связана с формой и создается в конструкторе *FormMain()*, поверхность *g* связана с элементом *bitmap*, который тоже создается только один раз в методе *FormMain()*, высота и ширина которого равны размерам прямоугольника *ClientRectangle*:

```
bitmap = new Bitmap(ClientSize.Width,ClientSize.Height);
```

В методе рисования *Draw()* строка получает объект *Graphics* из изображения:

```
using (Graphics g = Graphics.FromImage(bitmap))
```

Все операции прорисовки аналогичны выполняемым в предыдущем коде, за исключением того, что теперь они выполняются в изображении, а не непосредственно в окне. И наконец, в самом конце код выполняет рисование изображения в окне формы:

```
g0.DrawImage(bitmap, ClientRectangle);
```

Так как вначале линии рисуются в памяти, рисование занимает значительно меньше времени.

2.7. Режимы копирования

Метод *DrawImage()* копирует одно изображение (источник) на другое изображение (приемник). Свойство перечислимого типа *Drawing.CopyPixelOperation* определяет, как цвет точки источника объединяется с цветом точки приемника для получения окончательного

цвета. Перечислимый тип *CopyPixelOperation* может принимать следующие значения:

- *NoMirrorBitmap* – источник не копируется;
- *Blackness* – область приемника заполняется цветом, связанным с индексом 0 в физической палитре (черный цвет);
- *NotSourceErase* – цвета источника и приемника объединяются логическим оператором *OR*, и полученный цвет инвертируется;
- *NotSourceCopy* – инвертированная область источника копируется в область приемника;
- *SourceErase* – инвертированные цвета области приемника объединяются с цветами области источника с помощью логического оператора *AND*;
- *DestinationInvert* – область приемника инвертируется;
- *PatInvert* – цвета точек области приемника объединяются с цветами назначения с помощью логического оператора *XOR*;
- *SourceInvert* – цвета областей источника и приемника объединяются с помощью логического оператора *XOR*;
- *SourceAnd* – цвета областей источника и приемника объединяются с помощью логического оператора *AND*;
- *MergePaint* – цвета инвертированной области источника объединяются с цветами приемника с помощью оператора *OR*;
- *MergeCopy* – цвета области источника объединяются с цветами области приемника с помощью логического оператора *AND*;
- *SourceCopy* – область источника копируется в область;
- *SourcePaint* – цвета областей источника и приемника объединяются с помощью логического оператора *OR*;
- *PatCopy* – в область приемника копируется кисть, выбранная в данный момент в контексте устройства назначения;
- *PatPaint* – цвета кисти, выбранной в данный момент в контексте устройства приемника, объединяются с цветами инвертированной области источника с помощью логического оператора *OR*. Результат этой операции объединяется с цветами области приемника с помощью логического оператора *OR*;
- *Whiteness* – область приемника заполняется цветом, связанным с индексом 1 в физической палитре (белый цвет).

2.8. Модели цветов

Цвет — световая энергия, передаваемая волнами. На цвет объекта влияют:

- источники света, излучающие свет различных длин волн;
- отраженный цвет окружающих предметов.

Для описания излучаемого и отраженного цвета используются различные цветовые трехмерные модели, например RGB, CMYK, Lab.

Модели RGB, CMYK *аппаратно-зависимы*, модель Lab — *аппаратно-независима*. Во многих графических пакетах, например в Photoshop, возможно преобразовывание из одной цветовой модели в другую.

Некоторые цветовые модели:

- RGB (Red, Green, Blue);
- CMY (Cyan, Magenta, Yellow);
- CMYK (Cyan, Magenta, Yellow, Key (черный цвет));
- HSB (Hue, Saturation, Brightness (тон, насыщенность, яркость));
- Lab;
- HSV (Hue, Saturation, Value);
- HLS (Hue, Lightness, Saturation).

2.8.1. Модель RGB (Red, Green, Blue)

Модель образована на трех цветах: красном (`red`), зеленом (`green`) и синем (`blue`) и предназначена для описания излучаемых дисплеями цветов (рис. 2.21).

Доли каждого базового цвета меняются в диапазоне от 0 до 255. Смешиванием этих цветов можно получить 256^3 = 16777216 различных цветов. Модель аппаратно-зависима, так как вид базовых цветов может быть различным на разных дисплеях.

Рис. 2.21. Модель RGB

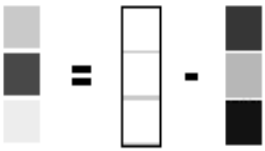

Рис. 2.22. Модель CMY = White-RGB

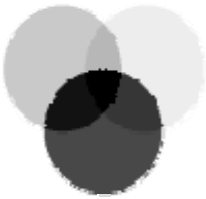

Рис. 2.23. Модель CMY

2.8.2. Модель CMY (Cyan, Magenta, Yellow)

Эта аппаратно-зависимая модель, в которой основные цвета образуются вычитанием из белого цвета 0xFFFFFF основных цветов модели RGB (рис. 2.22).

Базовые цвета модели CMY:

- голубой — 0xFFFF00;
- пурпурный — 0xFF00FF;
- желтый — 0x00FFFF.

Цвета этой модели предназначены для полиграфических машин, использующих три краски базовых цветов и печатающих в три прохода. При наложении двух цветов результат затемняется (рис. 2.23). При нулевом значении всех цветов образуется белый цвет.

2.8.3. Модель CMYK

Цветовая модель **CMYK** (Cyan, Magenta, Yellow, Key (черный цвет)) улучшает модель CMY добавлением черного цвета. Модель аппаратно-зависима.

2.8.4. Модель HSB

Аппаратно-зависимая модель **HSB** (Hue, Saturation, Brightness (тон, насыщенность, яркость)) основана на субъективном восприятии цвета человеком. Модель **HSB** основана на цветах модели RGB, в которой цвет определяется:

- тоном;
- насыщенностью — добавлением белого цвета;
- яркостью — добавлением черного цвета.

Любой цвет получается из спектрального цвета добавлением серого цвета (рис. 2.24 и 2.25). Модель HSB аппаратно-зависима.

Для человеческих глаз синий цвет кажется более темным, чем красный, а в модели HSB всем базовым цветам RGB приписывается одинаковая яркость.

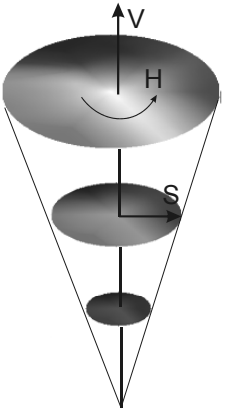

Рис. 2.24. Модели HSB и HSV:
H — частота света. Принимает значение от 0 до 360 градусов; V или B — яркость, определяющая уровень белого света. Высота конуса. S — определяет насыщенность цвета. Радиус конуса.

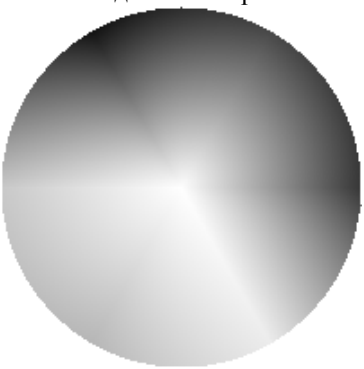

Рис. 2.25. Цветовой круг при S=1 и V=1

2.8.5. Модель Lab

Эксперименты показали — восприятие цвета зависит от человека и яркости освещения (в сумерках все цвета кажутся серыми). Международная комиссия по освещению (CIE — Commission Internationale de l'Eclairage) разработала стандарт для условий наблюдения цветов и исследовала восприятие цвета большой группы людей. Были экспериментально определены компоненты цветовой аппаратно-независимой модели XYZ.

Цветовая модель Lab — разновидность модели XYZ:

- компонент **L** — яркость изображения;
- компонент **a** изменяется от зеленого до красного цвета;
- компонент **b** изменяется от синего до желтого цвета.

Яркость в модели Lab не зависит от цвета, что позволяет регулировать контраст и резкость. Модель абстрактна и сильно математизирована, поэтому неудобна для практического применения.

Конверторы из одной цветовой модели в другую есть во многих графических программах.

2.8.6. Методы класса Color

Напомним, что в пространстве имен `System.Drawing` определена структура `Color`, у которой есть три метода, предназначенные для работы с цветовой моделью HSB:

- `float GetBrightness()` – получает значение яркости (оттенок-насыщенность-яркость (HSB)). Диапазон яркости от 0.0 до 1.0, где 0.0 представляет черное, а 1.0 – белое;
- `float GetHue()` – получает значение оттенка (оттенок-насыщенность-яркость (HSB)) в градусах для данной структуры Color. Оттенок измеряется в градусах, в диапазоне от 0.0 до 360.0, в цветовом пространстве HSB;
- `float GetSaturation()` – получает значение насыщенности (оттенок-насыщенность-яркость (HSB)). Диапазон насыщенности меняется от 0.0 до 1.0, где 0.0 представляет серость, а 1.0 – насыщенность.

2.8.7. Проект «Цветовые модели»

Рассмотрим проект, предназначенный для иллюстрирования работы с различными цветовыми моделями. Единственная форма этого проекта представлена на рисунке 2.26.

Рис. 2.26. Проект «Цветовые модели»

Помимо формы *FormMain* в проект включен файл CM.cs, в котором определен класс *CM*. В этом классе определены следующие методы перехода от одной цветовой модели к другой.

Листинг 2.20. Список методов класса CM

```
class CM
{
   public static void IntToRGB(int color,
      out float R,out float G,out float B)
   public static Color RGBtoColor(float R,
      float G,float B)

   public static void HueToRGB(float m0,
      float m2,float H,
      out float R,out float G,out float B)
   public static float RGBtoHue(float R,
      float G,float B)

   //== Hue, Lightness, Saturation = частота света,
   public static void HLStoRGB(float H, float L,
      float S,
      out float R, out float G, out float B)
   public static void RGBtoHLS(float R, float G,
      float B,
      out float H, out float L, out float S)
```

```
   //==== HSV (или HSB)=======================
   // Hue Saturation Brightness =
   // Тон Насыщенность Яркость
   public static void RGBtoHSV(float R, float G,
      float B,
      out float H, out float S, out float V)
   public static void HSVtoRGB(float H, float S,
      float V,
      out float R, out float G, out float B)
   // H определяет частоту света и принимает
   // значение от 0
   // до 360 градусов.
   // V - значение (принимает значения от 0 до 1)
   // или B –
   // яркость, определяющая уровень
   // белого света является высотой конуса.
   // S - определяет насыщенность цвета. Радиус конуса.

   //====== Hue Saturation Intensity =============
   public static void HSItoRGB(float H,float S,
      float I,
      out float R,out float G,out float B)
   public static void RGBtoHSI(float R, float G,
      float B,
      out float H, out float S, out float I)
}
   public static float Minimum(float R,float G,
      float B)
   public static float Maximum(float R,float G,
      float B)
   public static float Centre(float R, float G,
      float B)
   public static bool Verify(ref float v)
```

2.8.7.1. Методы для модели RGB

Для работы с моделью *RGB* предназначены два метода: *ColorToRGB* и *RGBtoColor*, позволяющие перейти от параметров *R,G,B* к цвету и обратно.

Листинг 2.21. Методы работы с моделью RGB

```
public static void IntToRGB(int color, out float R,out float
G,out float B)
{
    R=(color & 0x0000FF)/255F;
    G=((color & 0x00FF00) >> 8)/255F;
    B=((color & 0xFF0000) >> 16)/255F;
}
```

```
public Color RGBtoColor(float R,float G,float B)
{
    int r = (int)Math.Abs(R * 255);
    int g = (int)Math.Abs(G * 255);
    int b = (int)Math.Abs(B * 255);
    if (r < 0) r = 0; if (r > 255) r = 255;
    if (g < 0) g = 0; if (g > 255) g = 255;
    if (b < 0) b = 0; if (b > 255) b = 255;
    Color Result = Color.FromArgb(r, g, b);
    return Result;
}
```

2.8.7.2. Методы для модели HSV

Для работы с моделью HSV (или HSB) предназначены следующие методы конвертации.

Листинг 2.22. Методы работы с моделью HSV (или HSB)

```
public static void RGBtoHSV(float R, float G,
    float B, out float H, out float S, out float V)
{
    float m0, m2;
    H = 0; S = -1; V = -1;
    if (!Verify(ref R) | !Verify(ref G) |
       !Verify(ref B))
        return;
    m0 = Minimum(R, G, B); m2 = Maximum(R, G, B);
    if (m2 <= eps2)
    {
        S = 0; V = 0; return;
    }
    V = m2; S = (m2 - m0);
    if (S <= eps2) S = 0; else S = S / m2;
    if (S <= eps2)
    {
        S = 0; return;
    }
    H = RGBtoHue(R, G, B);
}

public static void HSVtoRGB(float H, float S, float V,
    out float R, out float G, out float B)
{
    float m0;
    R = -1; G = -1; B = -1;
    if (!Verify(ref V) | !Verify(ref S)) return;
    if (V <= eps2)
    {
```

```
        R = 0; G = 0; B = 0; return;
    }
    m0 = (1 - S) * V;
    if (Math.Abs(V - m0) <= eps2)
    {
        R = V; G = V; B = V; return;
    }
    HueToRGB(m0, V, H, out R, out G, out B);
}
```

Эти два метода используют пять вспомогательных методов: *Verify*, *HueToRGB*, *Minimum(R,G,B)*, *Maximum(R,G,B)* и *RGBtoHue*.

Листинг 2.23. Принадлежность переменной V интервалу [0, 1]

```
        public static bool Verify(ref float v)
        {
            bool Result=false;
            if ((v<0) || (v>1-0)) return Result;
            if (v<=Eps) v=0;
            if (v>=1-Eps) v=1;
            Result=true;
            return Result;
        }
```

Листинг 2.24. Преобразование тона (Hue) в RGB

```
public static void HueToRGB(float m0,float m2,
   float H, out float R,out float G,out float B)
{
   float mu,md,F;
   R = 0; G = 0; B = m0;
   int n;
   while (H<0) H+=360;
   n=Convert.ToInt32(Math.Truncate(H/60));
   F=(H-n*60)/60;
   n=n % 6; mu=m0+(m2-m0)*F; md=m2-(m2-m0)*F;
   switch (n)
   {
       case 0:
          R=m2; G=mu; B=m0;
          break;
       case 1:
          R=md; G=m2; B=m0;
          break;
       case 2:
          R=m0; G=m2; B=mu;
          break;
```

```
        case 3:
           R=m0; G=md; B=m2;
           break;
        case 4:
           R=mu; G=m0; B=m2;
           break;
        case 5:
           R=m2; G=m0; B=md;
           break;
    }
}
```

Листинг 2.25. Преобразование RGB в тон (Hue)

```
public static float RGBtoHue(float R,float G,float B)
{
    float Result=0;
    float F,m0,m1,m2;
    int n;
    {
        m2=Maximum(R,G,B);
        m0=Minimum(R,G,B);
        if (Math.Abs(m2-m0)<=Eps)
            return Result=0;
        m1=Centre(R,G,B);
        while (true)
        {
            if (Math.Abs(R-m2)<=Eps)
            {
                if (Math.Abs(B-m0)<=Eps) n=0;
                else n=5;
                return Result;
            }
            if (Math.Abs(G-m2)<=Eps)
            {
                if (Math.Abs(B-m0)<=Eps) n=1;
                else n=2;
                return Result;
            }
            if (Math.Abs(R-m2)<=Eps) n=3; else n=4;
            return Result;
        }
        if ((n & 1)==0) F=m1-m0; else F=m2-m1;
        F=F/(m2-m0);
        Result=60*(n+F);
        return Result;
    }
}
```

Листинг 2.26. Вычисление среднего значения из трех

```
public float Centre(float R, float G, float B)
{
    float Result = 0;
    while (true)
    {
       if ((R < G) & (R < B))
       {
          if (G < B) Result = G; else Result = B;
          return Result;
       }
       if ((G < R) & (G < B))
       {
          if (R < B) Result = R; else Result = B;
          return Result;
       }
       if (R < G) Result = R; else Result = G;
          return Result;
    }
}
```

2.8.7.3. Методы для модели HSI

Для работы с моделью HSI (Hue, Saturation, Intensity (тон, насыщенность, интенсивность)) предназначены следующие методы.

Листинг 2.27. Методы HSItoRGB и RGBtoHSI

```
public static void HSItoRGB(float H,float S,float I,
    out float R,out float G,out float B)
{
    float m0,m2;
    R=-1; G=-1; B=-1;
    if (!Verify(ref S) | !Verify(ref I)) return;
    m0=I*(1-S); m2=I;
    if (m2<=eps2)
    {
        R=0; G=0; B=0; return;
    }
    if (S<=eps2)
    {
        R=I; G=I; B=I; return;
    }
    if ((m2-m0)<=eps2)
    {
        R=I; G=I; B=I; return;
    }
    HueToRGB(m0,m2,H,out R,out G,out B);
}
```

```
public static void RGBtoHSI(float R, float G, float B,
    out float H, out float S, out float I)
{
    float m0, m2;
    H = 0; S = -1; I = -1;
    if (!Verify(ref R) | !Verify(ref G) | !Verify(ref B))
       return;
    m2 = Maximum(R, G, B); m0 = Minimum(R, G, B); I = m2;
    if (I <= eps2)
    {
        I = 0; S = 0; return;
    }
    S = m2 - m0;
    if (S <= eps2) S = 0; else S = S / m2;
    if (S < eps2)
    {
        S = 0; return;
    }
    H = RGBtoHue(R, G, B);
}
```

В проекте «Цветовые модели» оригинал изображения хранится на *bitmap1*. Цвет каждой точки раскладывается с помощью метода *IntToRGB* в доли *R,G,B*. Затем, в зависимости от выбранной модели, осуществляется преобразование *R,G,B* в параметры *S,L,V,I*. Эти параметры изменяются в соответствии со значениями компонентов *TrackBar* и снова преобразуются в *R,G,B*. Наконец, *R,G,B* преобразуются в цвет точки на *bitmap2* (листинг 2.28).

Листинг 2.28. Преобразование цветов с помощью моделей HLS, HSV и HSI

```
private void SetBitmap2()
{
    if ((bitmap1 == null) || (bitmap2 == null))
       return;
    float R; float G; float B;
    float H=0; float S=0F; float V=0F;
    float L=0; float I=0;
    for (int i=0; i<= bitmap1.Width-1;i++)
    for (int j = 0; j <= bitmap1.Height - 1; j++)
    {
        Color c = bitmap1.GetPixel(i, j);
        R = c.R/255F; G = c.G/255F; B = c.B / 255F;
    switch (flTools)
    {
        case 0:
            CM.RGBtoHLS(R,G,B, out H, out L, out S);
```

```
            break;
        case 1:
            CM.RGBtoHSV(R,G,B, out H, out S, out V);
            break;
        case 2:
            CM.RGBtoHSI(R,G,B,out H, out S, out I);
            break;
    }

    S*=trackBar1.Value/10;
    switch (flTools)
    {
        case 0:
            L *= trackBar2.Value / 10;
            break;
        case 1:
            V *= trackBar2.Value / 10;
            break;
        case 2:
            I *= trackBar2.Value / 10;
            break;
    }
    H *= trackBar3.Value / 10;

    switch (flTools)
    {
        case 0:
            CM.HLStoRGB(H,L,S, out R, out G, out B);
            break;
        case 1:
            CM.HSVtoRGB(H,S,V, out R, out G, out B);
            break;
        case 2:
            CM.HSItoRGB(H,S,I, out R, out G, out B);
            break;
    }
    bitmap2.SetPixel(i, j, CM.RGBtoColor(R, G, B));
  }
}
```

2.9. Рисование на канве принтера

GDI+ включает в себя поддержку вывода на печать, содержащуюся в классах, которые находятся в пространстве имен *Drawing.Printing*. Для написания программ, предназначенных для вывода графической информации на принтер, используются те же самые классы — *Graphics*, *Point*, *Size*,

Rectangle, Pen, Brush, Image и т.д. Но механизм их использования несколько отличается. Необходимо:

- ввести поле класса *PrintDocument*;
- создать обработчик событий с любым именем и зарегистрировать его как обработчик событий для вывода страницы:

```
PrintDocument printDoc = null;
public FormMain()
{
    InitializeComponent();
    printDoc = new PrintDocument();
    printDoc.PrintPage +=
        new PrintPageEventHandler(myPrintPage);
}
```

Обработчик событий для вывода на печать имеет следующую сигнатуру:

```
void myPrintPage(object sender, PrintPageEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawLine(Pens.Black,100,100,500,100);
}
```

Печать вызывается методом *Print()* объекта printDoc:

```
private void button1_Click(object sender, EventArgs e)
{
   printDoc.Print();
}
```

Пространство имен *Drawing.Printing* содержит три класса:

- *class PrintDocument* – определяет объект, который посылает информацию принтеру;
- *class PrintPageEventArgs* – предоставляет данные для события *PrintDocument.PrintPage*;
- *class PageSettings* – задает параметры печатаемой страницы.

Некоторые методы и события класса *PrintDocument*:

- *PrintDocument()* – конструктор;
- *event PrintPageEventHandler PrintPage* – событие происходит, когда необходимо вывести на печать текущую страницу;
- *void Print()* – начинает печать документа.

Класс *PrintPageEventArgs* содержит следующие свойства и методы:

- *Graphics Graphics* – полотно, используемое для рисования страницы;
- *Rectangle MarginBounds* – прямоугольную область, находящуюся внутри полей части страницы;

– *Rectangle PageBounds* – прямоугольную область, представляющую всю страницу;
– *PageSettings PageSettings* – параметры текущей страницы;
– *PrintPageEventArgs(Graphics graphics, Rectangle marginBounds, Rectangle pageBounds, PageSettings pageSettings)* – инициализацию нового экземпляра класса *PrintPageEventArgs*.

В конструкторе *PrintPageEventArgs()* параметры имеют следующий смысл:

– *Graphics* – полотно для рисования элемента;
– *marginBounds* – область внутри полей;
– *pageBounds* – вся область бумаги;
– *pageSettings* – параметры страницы.

Класс *PageSettings* позволяет задавать параметры страницы, выводимой на печать. Свойства класса:

– *Rectangle Bounds* – размер страницы в сотых долях дюйма с учетом ориентации страницы, задаваемой свойством *Landscape*;
– *bool Color* – определяет необходимость цветной печати страницы;
– *float HardMarginX* – координата оси *x* в сотых долях дюйма левого поля страницы;
– *float HardMarginY* – координата оси *y* в сотых долях дюйма верхнего поля страницы;
– *bool Landscape* – задает ориентацию печати: альбомное (*true*) или книжное (*false*);
– *Margins* – поля страницы в сотых долях дюйма. По умолчанию поля со всех сторон равны 1 дюйму;
– *PaperSize* – размер бумаги для страницы;
– *PaperSource* – источник бумаги для страницы;
– *RectangleF PrintableArea* – длина и ширина области печати страницы в сотых долях дюйма;
– *PrinterResolution* – разрешающая способность;
– *PrinterSettings* – параметры принтера для страницы.

Развернутый пример печати с использованием диалогов будет рассмотрен в пункте 6.1.

Глава 3. АЛГОРИТМЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

3.1. ЗАДАЧИ КОМПЬЮТЕРНОЙ ГРАФИКИ

Согласно классификации Ф. Мартинеса [29], информацию, содержащуюся в изображении, можно классифицировать следующим образом:

- определительная информация – идентификация и структура;
- топологическая информация – морфология и геометрия;
- визуальная информация – внешний вид и освещение.

Идентификация основана на именовании объектов или множеств объектов.

Структура отражает различные виды отношений объектов между собой:

- логические (принадлежность, включение и т.д.);
- топологические (близость, касание и т.д.);
- функциональные.

Морфология отражает форму объекта независимо от его положения и точки наблюдения. Все геометрические объекты являются комбинацией различных примитивов – простейших фигур, которые, в свою очередь, состоят из графических элементов – точек, линий, поверхностей.

Геометрия включает информацию о проекциях, видимости и т.д. Геометрия дополняется информацией о съемке (точке съемки, направлении, типе объектива, фокусном расстоянии, формате снимка) и отображении. Отображением называется преобразование двухмерного изображения плоскости проецирования (плана) в двухмерное изображение кадра.

Внешний вид содержит информацию, относящуюся к материалу, из которого состоит объект, в частности, о цвете, текстуре, яркости, прозрачности этого материала.

Освещение указывает на природу, число и расположение источников света, а также на условия видимости: туман, дым и т.д.

Перечислим основные математические задачи, встречающиеся при создании изображения на экране компьютера:

- преобразование системы координат и проецирование;
- удаление невидимых линий;
- освещение и тень;
- моделирование цвета;
- моделирование текстуры;
- логические операции над объектами.

Часть этих задач рассмотрена в данной главе, а также в главе 7.

3.2. Классификация алгоритмов

Алгоритмы компьютерной графики можно разделить на две группы: алгоритмы нижнего уровня и алгоритмы верхнего уровня. Алгоритмы нижнего уровня используются для рисования графических примитивов (линий, окружностей, заполнений областей и т.п.). Такие алгоритмы реализованы в графических библиотеках или реализованы аппаратно в графических процессорах.

В группе алгоритмов нижнего уровня можно выделить следующие подгруппы:

- алгоритмы, использущие простые математические методы и простые в реализации. Такие алгоритмы, как правило, не являются оптимальными;
- алгоритмы, в которых используются сложные математические методы, и эвристические алгоритмы;
- реализуемые аппаратно: рекурсивные алгоритмы, допускающие распараллеливание, алгоритмы, реализуемые в командах процессора;
- алгоритмы специального назначения, например для сглаживания линий (*Alias*).

К группе алгоритмов верхнего уровня относятся, например:

- алгоритмы удаления невидимых линий и поверхностей. Эффективность этих алгоритмов влияет на качество и скорость построения 3D изображения;
- алгоритмы построения полутоновых изображений, в которых необходимо учитывать: количество и свойства источников света, свойства поверхности объекта (преломление света, прозрачность объекта, отражение света).

3.3. Геометрические основы компьютерной графики

Для описания таких геометрических объектов, а также их взаимного расположения в пространстве, вводится *система координат* и каждой точке пространства сопоставляется набор вещественных чисел – *координат* этой точки. Числом координат в таком наборе определяется *размерность* пространства. Обычно рассматривают двумерные (2D) пространства на различных поверхностях и трехмерные (3D) пространства. В 2D-пространствах графическими элементами являются точки и линии, в 3D-пространствах к ним добавляются поверхности.

3.3.1. Графические элементы на плоскости

Простейшей формой поверхности является плоскость. Для описания геометрических объектов на плоскости используют *декартову* и *полярную*

системы координат. Координаты (x, y) и (r, φ) в этих системах связаны соотношениями

$$x = r\cos(\varphi), \quad y = r\sin(\varphi), \tag{3.1}$$

$$r = \sqrt{x^2 + y^2}, \quad tg(\varphi) = \frac{y}{x}. \tag{3.2}$$

Введем обозначение для координат точки на плоскости в декартовой и полярной системах координат:

$$p = (x, y) \equiv (r, \varphi). \tag{3.3}$$

Связь между координатами точек любой линии может быть задана *неявным уравнением* `f(p)=0` или *параметрической функцией* `p(t)`. Эти соотношения могут быть записаны в координатной или векторной форме. Векторная форма записи более компактна, однако в алгоритмах, как правило, заменяется более удобной для вычислений координатной формой.

Точка на плоскости имеет две степени свободы. Зависимость расстояния d между двумя точками p_1 и p_2 от их декартовых координат имеет вид

$$d = |p_1 - p_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}, \tag{3.4a}$$

а от полярных координат:

$$d = |p_1 - p_2| = \sqrt{r_1^2 + r_2^2 - 2r_1 \times r_2 \times \cos(\varphi_1 - \varphi_2)}. \tag{3.4b}$$

Линия на плоскости имеет одну степень свободы. Уравнения линии в неявной форме имеют вид

$$f(x, y) = 0, \tag{3.5a}$$

а параметрическая функция:

$$p(t) = [x(t), y(t)]. \tag{3.5b}$$

Неявное уравнение прямой линии имеет вид

$$A \times x + B \times y + D = 0, \tag{3.6}$$

где хотя A или B должно быть отлично от нуля. Прямую линию можно задать координатами одной из точек линии p_0 и вектором нормали $N = [N_x, N_y]$. Тогда неявное уравнение прямой линии можно записать в нормальной форме:

$$N_x \times (x - x_0) + N_y \times (y - y_0) = 0. \tag{3.7}$$

Если для задания прямой используется направляющий вектор $V = [V_x, V_y]$, то для описания прямой линии используется параметрическая функция

$$x(t) = x_0 + V_x \times t, \quad y(t) = y_0 + V_y \times t. \tag{3.8}$$

Так как векторы N и V перпендикулярны, то их скалярное призведение равно 0:

$$N_x \times V_x + N_y \times V_y = 0. \tag{3.9}$$

Сравнивая (3.6), (3.7) и (3.9), получаем для вектора нормали к прямой $N = [A,\ B]$, а для направляющего вектора $V = [-B,\ A]$.

Параметрическая функция удобна для построения частей прямой – отрезков и лучей. Для этого необходимо указать пределы изменения параметра:

- $-\infty < t < +\infty$, протяженность прямой не ограничена;
- $t \geq 0$, луч, выходящий из точки p0 в направлении вектора V;
- $t_1 \leq t \leq t_2$, отрезок прямой между точками $p0+V*t1$ и $p0+V*t2$.

Для произвольной линии на плоскости в любой регулярной (гладкой и некратной) точке $p_0 = [x_0,\ y_0] = p(t_0)$ возможна *линеаризация*, то есть построение *касательной прямой.* Уравнение касательной удобно записать в нормальной форме (3.7), где компоненты вектора нормали вычисляются как частные производные от функции в левой части неявного уравнения (3.5a) этой линии:

$$N_x = \frac{df(x,y)}{dx}\Big|_{x_0,y_0}, \quad N_y = \frac{df(x,y)}{dy}\Big|_{x_0,y_0}. \tag{3.10a}$$

Вектор нормали $N = [N_x,\ N_y]$ ортогонален касательной и направлен в ту сторону, где $f(x,y) > 0$.

Если линия задана своей параметрической функцией, то аналогичная функция для касательной имеет вид (3.9) с компонентами направляющего вектора:

$$V_x = \frac{dx(t)}{dt}\Big|_{x_0,y_0}, \quad V_y = \frac{dy(t)}{dt}\Big|_{x_0,y_0}. \tag{3.10b}$$

Выбор между описанием линии с помощью уравнения или с помощью параметрических функций зависит от решаемой задачи. При построении линий удобнее использовать их параметрическое представление или, если это возможно, явную форму уравнения вида $y = f(x)$.

В главе 6 описан ряд проектов построения линий на плоскости с использованием явных и неявных уравнений, а также параметрического представления.

3.3.2. Графические элементы в пространстве

Геометрические объекты в пространстве описываются с использованием декартовой, цилиндрической и сферической систем координат. Соответственно, положение точки в пространстве может определяться любой тройкой вещественных чисел:

- $x,\ y,\ z$ в декартовой системе координат;
- $r,\ \varphi,\ z$ в цилиндрической системе координат;
- $r,\ \theta,\ \varphi$ в сферической системе координат.

Расстояние d между двумя точками p_1 и p_2 вычисляется по формуле

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \quad (3.11a)$$

в декартовой системе координат,

$$d = \sqrt{r_1^2 + r_2^2 - 2r_1 \times r_2 \times \cos(\varphi_1 - \varphi_2) + (z_1 - z_2)^2} \quad (3.11b)$$

в цилиндрической системе координат и

$$d = \sqrt{r_1^2 + r_2^2 - 2r_1 r_2 (\sin(\theta_1)\sin(\theta_2) + \cos(\theta_1)\cos(\theta_2)\cos(\varphi_1 - \varphi_2))} \quad (3.11c)$$

в сферической системе координат.

Поверхность в пространстве имеет две степени свободы и ее можно описать с помощью неявного уравнения

$$f(x, y, z) = 0, \quad (3.12)$$

или с помощью параметрической функции

$$p(t, \tau) = \left[x(t, \tau), y(t, \tau), z(t, \tau)\right]. \quad (3.13)$$

Простейшей поверхностью является плоскость, для которой несложно записать неявное уравнение и параметрическую функцию.

В любой регулярной точке поверхности p_0 можно построить вектор нормали N с координатами:

$$N_x = \frac{df}{dx}\Big|_{x_0, y_0, z_0}, N_y = \frac{df}{dy}\Big|_{x_0, y_0, z_0}, N_y = \frac{df}{dz}\Big|_{x_0, y_0, z_0}. \quad (3.14)$$

Неявное уравнение плоскости задается четырьмя коэффициентами `A, B, C, D`:

$$A \times x + B \times y + C \times z + D = 0. \quad (3.15)$$

Хотя бы одно из чисел `A`, `B` или `C` должно быть отлично от нуля. Вектор нормали для плоскости равен `N = [A, B, C]`.

Нормальное уравнение плоскости, заданной точкой p_0 и вектором нормали N, имеет вид

$$N_x \times (x - x_0) + N_y \times (y - y_0) + N_z \times (z - z_0) = 0. \quad (3.16)$$

Параметрическая функция плоскости, заданной точкой p_0 и линейно независимыми направляющими векторами `V` и `W`, имеет вид

$$\begin{aligned} x(t, \tau) &= x_0 + V_x \times t + W_x \times \tau, \\ y(t, \tau) &= y_0 + V_y \times t + W_y \times \tau, \\ z(t, \tau) &= z_0 + V_z \times t + W_z \times \tau. \end{aligned} \quad (3.17)$$

Такая форма удобна для задания как всей плоскости, так и ее частей. Для этого достаточно указать пределы изменения параметров `t` и τ:

- $-\infty < t < +\infty$, $-\infty < \tau < +\infty$ – протяженность плоскости не ограничена;
- $-\infty < t < +\infty$, $\tau \geq 0$ – полуплоскость, находящаяся от прямой $p0+V\times t$ по одну сторону с вектором W;
- $-\infty < t < +\infty$, $0 \leq \tau \leq 1$ – полоса шириной $|W|$ в направлении вектора V;

– $0 \le t \le 1,\ 0 \le \tau \le 1$ – параллелограмм с вершинами в точках $p0$, $p0+V$, $p0+W$, $p0+V+W$.

Вектор нормали к плоскости можно представить как векторное произведение векторов V и W:

$$N = [V \times W]. \tag{3.18}$$

Линия в пространстве, как и на плоскости, имеет одну степень свободы и может быть описана либо как результат пересечения двух поверхностей, либо как траектория движения точки. В первом случае потребуется решение системы двух нелинейных уравнений $f_1(x, y, z) = 0$ и $f_2(x, y, z) = 0$, что практически крайне неудобно. Во втором случае мы имеем дело с параметрическим представлением пространственных кривых (рис. 3.1):

$$p(t) = [x(t), y(t), z(t)]. \tag{3.19}$$

Как уже отмечалось, параметрическое представление является более универсальным и удобным для проведения вычислений. При этом снимается ряд проблем, связанных с использованием многозначных функций в уравнениях для замкнутых кривых, а также с бесконечно большими значениями частных производных этих функций в некоторых точках кривых. В параметрическом представлении легко описываются замкнутые кривые и, кроме того, вместо тангенсов углов наклона кривых к координатным осям используются касательные векторы, которые никогда не бывают бесконечными.

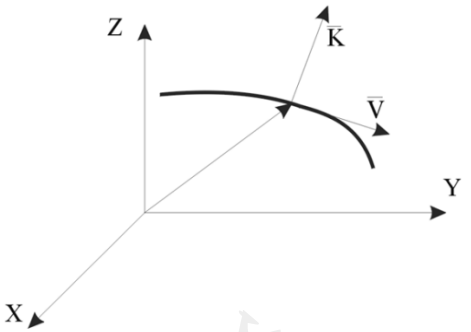

Рис. 3.1. Параметрическое описание пространственной кривой

В каждой регулярной точке $p_0 = p(t_0)$ кривой L (первая производная $p'(t_0) \neq 0$) можно вычислить единичный вектор касательной V и вектор кривизны K:

$$\begin{aligned} V &= p'(t)/|p'(t)|, \\ K &= \left[[p'(t) \times p''(t)] \times p'(t)\right]/|p'(t)|. \end{aligned} \tag{3.20}$$

Модуль вектора кривизны характеризует степень отклонения кривой L от прямой линии.

3.4. Задачи интерполяции, сглаживания и аппроксимации

При работе с линиями на плоскости или в трехмерном пространстве приходится решать одну из следующих трех задач:

- задачу *интерполяции*, состоящую в выборе линии, проходящей через заданное множество точек;
- задачу *сглаживания* – выбор линии, проходящей вблизи заданного множества точек;
- задачу *аппроксимации* – выбор линии, достаточно близкой к данной линии.

Для более строгой формулировки этих задач введем несколько понятий и обозначений.

Пусть F – функциональное пространство однозначных функций, в котором определена полунорма φ, $V \subset F$ – векторное функциональное пространство с базисом R^n. Обозначим через $\{\delta_i\}$ $i \in I = 0, 1, \ldots, m$ множество функционалов, каждый из которых ставит в соответствие вектору $v \in V$ действительное число $\delta_i(v)$. Кроме того, пусть $\{z_i\}$ $i \in I$ – множество точек, заданных на плоскости или в 3D-пространстве. Введенные понятия позволяют дать следующие формулировки вышеперечисленным задачам.

Задача интерполяции. Найти $v \in V$ такой, что $\delta_i(v) = z_i$ для любого $i \in I$.

Задача сглаживания. Найти $v \in V$ такой, что v близко к множеству $\{z_i\}$, то есть что полунорма

$$\varphi(\{\delta_i(v)\} - \{z_i\}, i \in \mathrm{I})$$

получает минимальное значение.

Задача аппроксимации. Найти $v \in V$ такой, что полунорма $\varphi(f - v)$ имеет минимальное значение. Здесь $f \in F$ – некоторая заданная функция.

Часто сглаживание называют точечной аппроксимацией, отмечая близость двух последних задач.

3.4.1. Интерполяция полиномами

Рассмотрим функции $v(t) \in V = P^m$ над одномерным множеством $[a, b] \subset \Re$. Пусть $\delta_i(v) = v(t_i) = z_i$, $i \in I$, $z_i \in \Re$, $a \le t_0 \le t_1 \le \ldots \le t_m \le b$.

Если в качестве базиса R^n пространства V выбраны степенные функции 1, t, t^2, ..., t^m, то задача интерполяции сводится к решению линейной системы уравнений

$$\sum_{j=0}^{m} v_j \times t_i^j = z_i \qquad (3.21)$$

с симметричной обратимой матрицей, называемой матрицей Вандермонда.

Решение задачи интерполяции значительно упрощается, если в качестве базиса пространства V выбраны полиномы Лагранжа:

$$L_i(t) = \prod_{j=0, j\neq i} \frac{t - t_j}{t_i - t_j}, \quad i \in I. \tag{3.22}$$

В этом базисе решением является функция

$$v(t) = \sum_{i=0}^{m} z_i \times L_i(t). \tag{3.23}$$

Чаще всего в качестве базиса R^n выбираются полиномы Ньютона

$$N_0(t) = 1, \quad N_i(t) = \prod_{j=0}^{i} (t - t_j), \quad i \in I. \tag{3.24}$$

Решение системы уравнений

$$\sum_{j=0}^{m} v_j \times N_j(t_i) = z_i \tag{3.25}$$

записывается в рекуррентном виде

$$v_0 = z_0, \quad v_i = (z_i - \sum_{j=0}^{i-1} v_j N_j(t_i)) / N_i(t_i), \quad i \in I. \tag{3.26}$$

Величины v_i называются разделенными разностями порядка i на множестве $\{z_i\}$ $i \in I$.

Необходимо отметить, что и полиномы Лагранжа, и полиномы Ньютона могут давать большие отклонения от ожидаемых значений функции.

Проект для полиномов Лагранжа приводится в пункте 6.3.2.

В случае, если в точках $\{t_i\}$, $i \in I$ заданы не только значения функции $f(t_i)$, но и производные $f'(t_i)$, то число условий на интерполяционный многочлен удваивается и он должен принадлежать пространству $V = P^{2m+1}$ полиномов степени не меньшей или равной $2m + 1$. Пусть $\{z_i\} \subset \Re$, $i = 0, 1, ..., 2m+1$ и пусть $z_i = f(t_i)$, $z_{i+m+1} = f'(t_i)$, $i = 0, 1, ..., m$. Тогда решение задачи приводит к полиному Эрмита, выраженному через полиномы Лагранжа:

$$v(t) = \sum_{i=0}^{m} z_i (1 - 2(t - t_i) L_i'(t_i)) L_i^2(t) + \sum_{i=0}^{m} z_{i+m+1} (t - t_i) L_i^2(t). \tag{3.27}$$

Интерполяция полиномами может быть использована при небольшом числе точек (не более 15). Для большего числа точек растет степень полинома и неустойчивость линии, проявляющаяся в больших межузловых осцилляциях.

3.4.2. Интерполяция кубическими сплайнами

Проблема неустойчивости глобальных интерполирующих кривых решается путем перехода к кусочной (локальной) интерполяции полиномами невысоких степеней, называемых сплайнами. Коэффициенты сплайнов подбираются из условий достаточной гладкости результирующей кривой в точках сшивания. Минимальную интегральную степень кривизны интерполирующей кривой обеспечивают сплайновые полиномы третьей степени – кубические сплайны.

По-прежнему будем рассматривать функции $v(t) \in V = P^m$ над одномерным множеством $[a, b] \subset \Re$, $\delta_i(v) = z_i$, $i = 0, 1, ..., m$, $z_i \in \Re$, $a \le t_0 \le t_1 \le ... \le t_m \le b$. Интерполяционным кубическим сплайном называется функция $v(t)$ такая, что

$$v(t_i) = z_i, i = 0,1,...,m,$$

на любом интервале $[t_i, t_{i+1}]$ являющаяся многочленом третьей степени:

$$v_i(t) = \sum_{j=0}^{3} a_j^i (t - t_i)^j \qquad (3.28)$$

и дважды непрерывно дифференцируемая.

Построение кубического сплайна требует нахождения $4 \times m$ коэффициентов a_{ij} ($i = 0, 1, \ldots, m$; $j = 0, 1, 2, 3$) полиномов 3-й степени. Эти коэффициенты определяются системой линейных уравнений, которые получаются из условий непрерывности функции, а также ее первой и второй производных в $m - 1$ промежуточных узлах, условия прохождения кривой через $m + 1$ узел и 2-х условий в граничных узлах. Такая система линейных уравнений с трехдиагональной матрицей приводится в главе 6 в рамках описания проекта приложения, реализующего построение графика сплайн-функции по заданному набору точек.

Проект для кубических сплайнов приводится в пункте 6.3.4.

При изменении координат узловых точек необходимо пересчитывать коэффициенты a_{ij}. Этого недостатка нет у некоторых методов сглаживания.

3.4.3. Сглаживание и аппроксимация

Метод наименьших квадратов. Функция $y(x) \in V$ рассматривается над одномерным множеством $[a, b] \subset \Re$. Для решения задачи требуется минимизировать сумму квадратов отклонений функции $y(x)$ от заданных значений y_i, $i = 0, 1, ..., m$.

Проект для метода наименьших квадратов приводится в пункте 6.3.3.

Кривые Безье. Другим способом сглаживания является использование кривой Безье, построенной на многочленах Бернштейна. Рассмотрим упорядоченный набор точек, определяемых множеством векторов $\{y_i\}$ $i \in I$. Ломаная $y_0 y_1 \ldots y_m$ называется контрольной ломаной, проходящей через эти точки.

Кривая Безье, порожденная множеством векторов $\{y_i\}$, – это линия, задаваемая векторной параметрической функцией вида

$$r(t) = \sum_{i=0}^{m} C_m^i \times t^i \times (1-t)^{m-i} \times y_i, \quad 0 \le t \le 1, \tag{3.29}$$

где

$$C_m^i = \frac{m!}{i! \times (m-i)!}$$

– биномиальные коэффициенты. Коэффициенты при вершинах y_i являются многочленами Бернштейна

$$C_m^i \times t^i \times (1-t)^{m-i}. \tag{3.30}$$

Многочлены Бернштейна обладают следующими свойствами:

1) определены на интервале [0, 1] и неотрицательны во всех его точках и при всех значениях параметров i и m ($i \le m$);

2) сумма многочленов равна единице:

$$\sum_{i=0}^{m} C_m^i \times t^i \times (1-t)^{m-i} = (t + (1-t))^m = 1. \tag{3.31}$$

Для $m = 5$ многочлены Бернштейна представлены на рисунке 3.2.

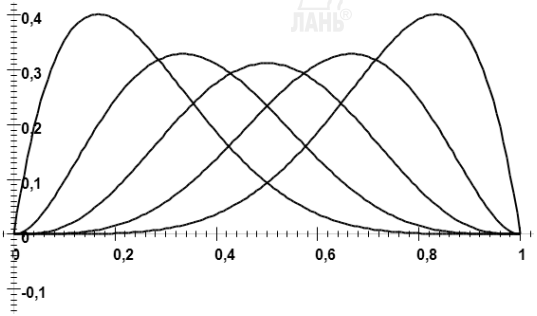

Рис. 3.2. Многочлены Бернштейна

Кривая Безье является гладкой кривой, которая проходит через концевые точки v_0 и v_m контрольной ломаной и касается ее начальных отрезков (рис. 3.3). Кривая Безье осуществляет сглаживание контрольной ломаной, заданной последовательностью векторов $\{y_i\}$.

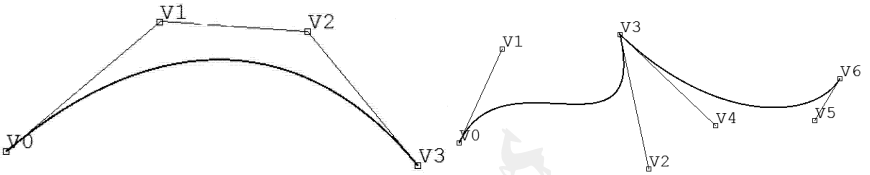

Рис. 3.3. Кривые Безье

При изменении координат любой точки y_i необходимо заново строить параметрическую функцию (3.29).

3.5. АФФИННЫЕ ПРЕОБРАЗОВАНИЯ КООРДИНАТ

В компьютерной графике используются три системы координат:

- неподвижная мировая система координат;
- объектная система координат, связанная с объектом и совершающая с ним движения в мировой системе координат;
- экранная система координат, связанная с графическим устройством.

Связь экранной системы координат с декартовой системой для двумерного пространства будет рассмотрена в пункте 5.1. Рассмотрим связь между мировой и объектной системами, используя понятие *аффинного преобразования координат*.

3.5.1. АФФИННЫЕ ПРЕОБРАЗОВАНИЯ НА ПЛОСКОСТИ

Наиболее просто такая связь описывается в простейшем случае 2D-пространства.

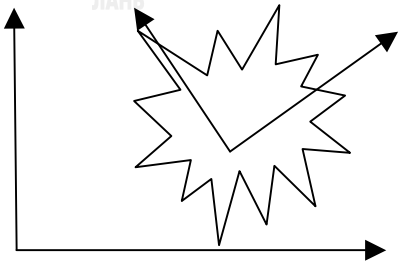

Рис. 3.4. Мировая и объектная системы координат на плоскости

Введем понятие *сцены* как системы объектов, изображение которой должно быть воспроизведено средствами компьютерной графики. Пусть некоторой точке P сцены в мировой системе координат соответствуют координаты (x, y), а в объектной системе координат – координаты (x, y). Если угол поворота объектной системы координат относительно мировой

системы координат равен φ, а начало объектной системы координат расположено в точке (x_0, y_0), то

$$\begin{aligned} \bar{x} &= (x - x_0) \times \cos\varphi + (y - y_0) \times \sin\varphi, \\ \bar{y} &= -(x - x_0) \times \sin\varphi + (y - y_0) \times \cos\varphi. \end{aligned} \tag{3.32}$$

Обратное преобразование имеет вид

$$\begin{aligned} x &= \bar{x} \times \cos\varphi - \bar{y} \times \sin\varphi + x_0, \\ y &= \bar{x} \times \sin\varphi + \bar{y} \times \cos\varphi + y_0. \end{aligned} \tag{3.33}$$

В общем случае переход от мировой системы координат к объектной системе координат включает в себя два действия – поворот на угол φ и сдвиг в направлении вектора (x_0, y_0).

Преобразования (3.32), (3.33) можно интерпретировать двояко:

- изменение координат фиксированной точки при изменении системы координат;
- изменение координат точки при использовании фиксированной системы координат.

Во втором случае отображение переводит точку $(\bar{x}, \bar{y})$ в точку (x, y). Отображение (3.33) может быть обобщено:

$$\begin{aligned} x &= \alpha \times \bar{x} + \beta \times \bar{y} + \lambda, \\ y &= \gamma \times \bar{x} + \delta \times \bar{y} + \mu. \end{aligned} \tag{3.34}$$

Для обратимости преобразования коэффициенты преобразования должны удовлетворять условию

$$\begin{vmatrix} \alpha & \beta \\ \gamma & \delta \end{vmatrix} \neq 0. \tag{3.35}$$

Преобразование координат (3.35) называется *аффинным* (*affinis* – подобный), т.к. обеспечивает сохранение отношения подобия.

Аффинные преобразования удобно представлять в матричной форме. Для этого введем векторное представление для точек пространства:

$$p = (x, y). \tag{3.36}$$

Тогда произвольное аффинное преобразование можно представить в виде

$$p = \bar{p}M + V, \tag{3.37}$$

где

$$M = \begin{pmatrix} \alpha & \gamma \\ \beta & \delta \end{pmatrix}, \; V = (\lambda, \mu). \tag{3.38}$$

Можно строго доказать, что любое аффинное преобразование представимо в виде суперпозиции элементарных преобразований – поворота, растяжения, отражения и переноса.

3.5.1.1. Преобразование переноса

$$\begin{aligned} x &= \bar{x} + \lambda, \\ y &= \bar{y} + \mu. \end{aligned} \tag{3.39a}$$

В векторной форме

$$p = \bar{p} + V. \tag{3.39b}$$

3.5.1.2. Преобразование растяжения

$$\begin{aligned} x &= \alpha \times \bar{x}, \\ y &= \delta \times \bar{y}, \end{aligned} \qquad (\alpha > 0, \delta > 0). \tag{3.40a}$$

В матричной форме

$$p = \bar{p} \begin{pmatrix} \alpha & 0 \\ 0 & \delta \end{pmatrix}. \tag{3.40b}$$

3.5.1.3. Преобразование отражения относительно оси абсцисс

$$\begin{aligned} x &= \bar{x}, \\ y &= -\bar{y}. \end{aligned} \tag{3.41a}$$

В матричной форме

$$p = \bar{p} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{3.41b}$$

3.5.1.4. Преобразование поворота

$$\begin{aligned} x &= \bar{x} \times \cos\varphi - \bar{y} \times \sin\varphi, \\ y &= \bar{x} \times \sin\varphi + \bar{y} \times \cos\varphi. \end{aligned} \tag{3.42a}$$

В матричной форме

$$p = \bar{p} \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix}. \tag{3.42b}$$

Нетрудно видеть, что преобразованию переноса соответствует аддитивный член в формуле (3.39a), тогда как остальные преобразования выполняются мультипликативно.

Чтобы сделать аффинное преобразование однородным, включить в него преобразование переноса мультипликативным образом, в проективной геометрии вводится понятие *расширенного пространства* и *однородных координат* точки. В основе этого метода лежит представление о том, что каждая точка в *n*-мерном пространстве может рассматриваться как проекция точки из (*n+1*)-мерного пространства при фиксированном значении (*n+1*)-й координаты. Для достижения однородности аффинных преобразований это значение следует выбрать равным 1.

Однородными координатами точки $p(x, y) \in$ 2D называется тройка чисел x_1, x_2, u такая, что

$$x = \frac{x_1}{u}, \quad y = \frac{x_2}{u} \tag{3.43}$$

и $u \neq 0$. В однородных координатах при $u = 1$ вектор точки p имеет вид

$$p(x, y, 1). \tag{3.44}$$

В общем случае $u \neq 1$ точке p в пространстве 2D соответствует *эквивалентный вектор* (рис. 3.5)

$$p' = (ux, uy, u). \tag{3.45}$$

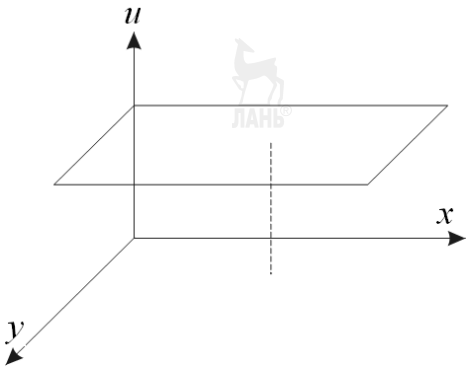

Рис. 3.5. Однородные координаты в 3D-пространстве

Любое аффинное преобразование в однородных координатах записывается в виде

$$p = \overline{p}M, \tag{3.46}$$

где M – суперпозиция матриц элементарных преобразований:

– вращения (rotation) вокруг начала координат на угол φ с матрицей

$$M \equiv R = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}; \tag{3.47}$$

– сжатия-растяжения (dilatation) вдоль осей OX и OY с матрицей

$$M \equiv D = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix}; \tag{3.48}$$

– отражения (reflection) относительно осей OX и OY с матрицей

$$M \equiv F = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

$$M \equiv F = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}; \qquad (3.49)$$

– переноса (translation) начала координат с матрицей переноса

$$M \equiv T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda & \mu & 1 \end{pmatrix}. \qquad (3.50)$$

Приведем примеры использования однородных координат для выполнения аффинных преобразований.

Пример 1. Построить матрицу поворота вокруг точки `P(a,b)` на угол φ.

1-й шаг. Перенос на вектор `P = [-a, -b]` для совмещения центра поворота с началом координат.

$$T_{-P} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{vmatrix}. \qquad (3.51a)$$

2-й шаг. Поворот на угол φ.

$$R_{\phi} = \begin{vmatrix} \cos(\varphi) & \sin(\varphi) & 0 \\ -\sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{vmatrix}. \qquad (3.51b)$$

3-й шаг. Перенос на вектор `P = [a, b]` для возвращения центра поворота в прежнее положение.

$$T_{P} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{vmatrix}. \qquad (3.51c)$$

Перемножая матрицы, получим

$$M = T_{-P} * R_{\phi} * T_{P} = \begin{vmatrix} \cos(\varphi) & \sin(\varphi) & 0 \\ -\sin(\varphi) & \cos(\varphi) & 0 \\ -a\cos(\varphi) + b\sin(\varphi) + a & -a\sin(\varphi) - b\cos(\varphi) + b & 1 \end{vmatrix}. \qquad (3.52)$$

Пример 2. Построить матрицу растяжения с коэффициентами растяжения α вдоль оси абсцисс, δ вдоль оси ординат и с центром в точке `P(a, b)`.

1-й шаг. Перенос на вектор `P = [-a,-b]` для совмещения центра поворота с началом координат.

$$T_{-P} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{vmatrix}. \qquad (3.53a)$$

2-й шаг. Растяжение вдоль координатных осей с коэффициентами α и δ соответственно.

$$D = \begin{vmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{vmatrix}. \qquad (3.53b)$$

3-й шаг. Перенос на вектор `P = [a, b]` для возвращения центра поворота в прежнее положение.

$$T_P = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{vmatrix}. \qquad (3.53c)$$

Перемножая матрицы, получим

$$M = T_{-P} * D * T_P = \begin{vmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ (1-\alpha)*a & (1-\delta)*b & 1 \end{vmatrix}. \qquad (3.54)$$

3.5.2. АФФИННЫЕ ПРЕОБРАЗОВАНИЯ В ПРОСТРАНСТВЕ

Можно ввести однородные координаты `p(x, y, z, 1)` для описания положения точки в трехмерном пространстве.

В 3D-пространстве любое аффинное преобразование можно представить суперпозицией преобразований: переноса, поворота, растяжения, отражения. Далее представлены матрицы элементарных преобразований в однородных координатах.

3.5.2.1. Матрицы вращения

Вращение вокруг оси OX на угол φ:

$$R_x = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\varphi) & \sin(\varphi) & 0 \\ 0 & -\sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \qquad (3.55a)$$

Вращение вокруг оси OY на угол ψ:

$$R_y = \begin{vmatrix} \cos(\psi) & 0 & -\sin(\psi) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\psi) & 0 & \cos(\psi) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \quad (3.55b)$$

Вращение вокруг оси OZ на угол χ:

$$R_z = \begin{vmatrix} \cos(\chi) & \sin(\chi) & 0 & 0 \\ -\sin(\chi) & \cos(\chi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \quad (3.55c)$$

3.5.2.2. Матрица сжатия-растяжения

$$D = \begin{vmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \quad (3.56)$$

α > 0 – коэффициент сжатия-растяжения вдоль оси OX,
δ > 0 – коэффициент сжатия-растяжения вдоль оси OY,
γ > 0 – коэффициент сжатия-растяжения вдоль оси OZ.

3.5.2.3. Матрицы отражения

Отражение относительно плоскости *XOY*:

$$M_z = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \quad (3.57a)$$

Отражение относительно плоскости *YOZ*:

$$M_x = \begin{vmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \quad (3.57b)$$

Отражение относительно плоскости *XOZ*:

$$M_y = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \quad (3.57c)$$

3.5.2.4. Матрица переноса

$$T = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \lambda & \mu & \nu & 1 \end{vmatrix}, \tag{3.58}$$

где $V = [\lambda \quad \mu \quad \nu]$ – вектор переноса.

Рассмотрим несколько примеров выполнения аффинных преобразований в 3D-пространстве.

Пример 1. Построить матрицу поворота на угол α вокруг прямой `L`, проходящей через точку `P(a, b, c)` и имеющую единичный направляющий вектор `(l, m, n)`.

1-й шаг. Перенос на вектор `-P = [-a, -b, -c]`.

$$T_{-P} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & -b & -c & 1 \end{vmatrix}. \tag{3.59a}$$

После этого прямая `L` будет проходить через начало координат.

2-й шаг. Совмещение оси аппликат с прямой `L`. Достигается двумя последовательными поворотами.

Первый поворот выполняется вокруг оси абсцисс до совмещения оси аппликат с проекцией прямой на плоскость `yOz`. Проекция прямой `L` на эту плоскость имеет направляющий вектор (`0, m, n`), поэтому для угла поворота φ справедливы следующие соотношения:

$$\cos(\varphi) = \frac{n}{d}, \quad \sin(\varphi) = \frac{m}{d}, \tag{3.59b}$$

где $d = \sqrt{m^2 + n^2}$.

Тогда для матрицы первого поворота получим:

$$R_x = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{n}{d} & \frac{m}{d} & 0 \\ 0 & \frac{-m}{d} & \frac{n}{d} & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \tag{3.59c}$$

Под действием этого преобразования изменятся координаты направляющего вектора прямой. Новый вектор будет равен

$$(l, \quad m, \quad n, \quad 1) \times R_x = (l, \quad 0, \quad d, \quad 1). \tag{3.59d}$$

Второй поворот выполняется вокруг оси ординат до совмещения прямой с осью аппликат. Для этого угла поворота ψ справедливы следующие соотношения:

$$\cos(\psi) = d, \quad \sin(\psi) = -l. \tag{3.59e}$$

Соответствующая матрица вращения запишется в виде

$$R_y = \begin{vmatrix} d & 0 & l & 0 \\ 0 & 1 & 0 & 0 \\ -l & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \tag{3.59f}$$

Теперь направляющий вектор прямой будет иметь координаты

$$(l, \quad 0, \quad d, \quad 1) \times R_y = (0, \quad 0, \quad 1, \quad 1). \tag{3.59g}$$

3-й шаг. Вращение вокруг прямой L на заданный угол α. Так как теперь прямая совпадает с осью аппликат, то соответствующая матрица имеет вид

$$R_z = \begin{vmatrix} \cos(\alpha) & \sin(\alpha) & 0 & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}. \tag{3.59h}$$

4-й шаг. Поворот вокруг оси ординат на угол -ψ.

5-й шаг. Поворот вокруг оси абсцисс на угол -φ. Порядок вращений существенен, так как преобразование вращения некоммутативно.

6-й шаг. Перенос на вектор `P(a,b,c)`.

Результирующая матрица получается в результате перемножения семи матриц:

$$A = T_{-P} \times R_x(\varphi) \times R_y(\psi) \times R_z(\alpha) \times R_y(-\psi) \times R_x(-\varphi) \times T_P.$$

Пример 2. Требуется подвергнуть заданному аффинному преобразованию выпуклый многогранник.

Выпуклый многогранник однозначно задается набором своих вершин $V_i = (x_i, y_i, z_i)$, $i = 1, 2, \ldots, n$. Построим матрицу

$$V = \begin{pmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ . & . & . & . \\ x_n & y_n & z_n & 1 \end{pmatrix}. \tag{3.60}$$

Умножая ее на матрицу A заданного аффинного преобразования, получим набор вершин преобразованного многогранника $V' = VA$.

3.5.3. МЕТОДЫ КЛАССА GRAPHICS

Для аффинных преобразований в пространстве имен *System.Drawing.Drawing2D* определен класс *Matrix* матрицы 3×3. В этом классе определены следующие методы:

Invert() – обращает матрицу *Matrix*, если она обратима;

Multiply(Matrix matrix) – умножает матрицу *Matrix* на матрицу, указанную в параметре *matrix*, путем добавления указанной матрицы *Matrix*;

Reset() – сбрасывает этот объект *Matrix*, чтобы получить элементы единичной матрицы;

Rotate(float angle) – добавляет в объект *Matrix* поворот по часовой стрелке вокруг начала координат на указанный угол поворота в градусах;

RotateAt(float angle, PointF point) – применяет поворот по часовой стрелке к объекту *Matrix*; поворот производится вокруг точки, указанной в параметре point, путем добавления поворота в начало;

Scale(float scaleX, float scaleY) – применяет указанный вектор масштабирования к объекту *Matrix*, добавляя вектор масштабирования в начало;

Shear(float shearX, float shearY) – применяет указанный вектор сдвига к объекту *Matrix*, добавляя преобразование сдвига в начало;

TransformPoints(Point[] pts) – применяет геометрическое преобразование, представляемое объектом *Matrix*, к указанному массиву точек;

TransformVectors(Point[] pts) – применяет только компоненты масштабирования и поворота объекта *Matrix* к указанному массиву точек;

Translate(float offsetX, float offsetY) – применяет указанный вектор смещения (*offsetX* и *offsetY*) к объекту *Matrix*, добавляя вектор смещения в начало;

VectorTransformPoints(Point[] pts) – умножает каждый вектор массива на матрицу. Элементы смещения данной матрицы (третья строка) игнорируются.

В классе *Graphics* для аффинных преобразований определены следующие методы:

Matrix Transform – возвращает или задает копию геометрического универсального преобразования объекта *Graphics*;

MultiplyTransform(Matrix matrix) – умножает универсальное преобразование данного объекта *Graphics* на указываемый

объект *Matrix*. matrix: Объект *Matrix* 4×4, на который умножается универсальное преобразование;

ResetTransform() – сбрасывает матрицу универсального преобразования данного объекта *Graphics* и делает ее единичной матрицей;

RotateTransform(float angle) – применяет заданное вращение к матрице преобразования данного объекта *Graphics*. *angle* – угол поворота в градусах;

ScaleTransform(float sx, float sy) – применяет указанную операцию масштабирования к матрице преобразования данного объекта Graphics путем ее добавления к матрице преобразования объекта;

TranslateTransform(float dx, float dy) – изменяет начало координат координатной системы путем добавления заданного сдвига к матрице преобразования данного объекта *Graphics*.

Ниже приводится пример метода рисования сдвинутого и повернутого эллипса.

Листинг 3.1. Рисование сдвинутого и повернутого эллипса

```
private void Draw()
{
    g = CreateGraphics();
    g.Clear(Color.White);
    g.DrawRectangle(Pens.Black,98,98,4,4);
    g.TranslateTransform(100, 100);
    g.RotateTransform(45);
    g.DrawEllipse(Pens.Black, 0, 0, 200, 100);
    g.Dispose();
}
```

На рисунке 3.6 изображен повернутый эллипс. Маленький прямоугольник изображает точку с координатами (100,100), относительно которой произведен поворот.

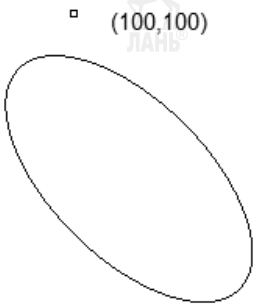

Рис. 3.6. Смещенный и повернутый эллипс

3.6. ПРОЕЦИРОВАНИЕ

Проецирование – отображение 3D-объекта на 2D-плоскость, называемую *картинной плоскостью.*

Для построения проекции из точки, называемой *центром проецирования,* проводятся линии через вершины объекта. Точка пересечения линии с картинной плоскостью – проекция соответствующей вершины объекта. В общем случае линия может проходить через несколько вершин объекта, отображая их в одну точку на картинной плоскости. Поэтому преобразование проецирования не является взаимно однозначным соответствием и, следовательно, необратимо.

Из сказанного следует, что при решении задачи проецирования кроме законов оптики в общем случае необходимо учитывать субъективное восприятие человеческим мозгом изображения, попадающего на сетчатку глаза. Отклонение математических проекций от изображений, воспринимаемых человеческим глазом, подмечено художниками давно, но лишь в последнее время предпринимаются попытки получить эмпирические формулы, описывающие эти отклонения. В качестве примера можно привести работы академика Б.В. Раушенбаха, которому за цикл работ по математической теории изображений Российской Академией наук присуждена Демидовская премия.

На рисунке 3.7 представлены наиболее часто используемые в геометрии, черчении и компьютерной графике способы проецирования (виды проекций).

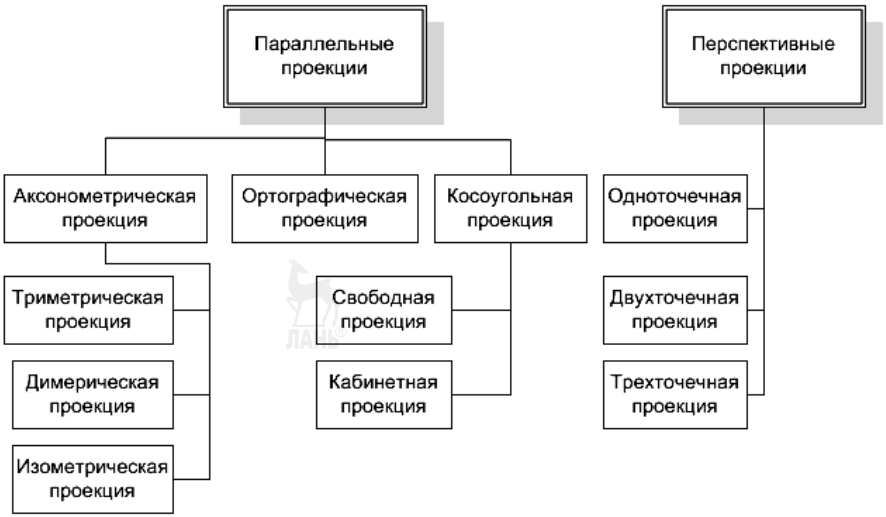

Рис. 3.7. Виды проецирования

Проекции разделяют на два группы: центральные и параллельные. Если центр проецирования размещен на конечном расстоянии от картинной плоскости, то это центральное проецирование. При удалении центра проецирования в бесконечность выполняется параллельное проецирование.

Центральная проекция дает достаточно реалистичное изображение. Параллельная проекция дает менее реалистичное изображение, сохраняя размеры и углы для граней, параллельных картинной плоскости.

3.6.1. ОРТОГРАФИЧЕСКОЕ ПРОЕЦИРОВАНИЕ

Различие между ортографическим и аксонометрическим проецированием заключается в выборе способа ориентации объектной системы координат относительно мировой системы координат. При ортографическом проецировании оси объектной системы координат направлены параллельно соответствующим осям мировой системы координат. Аксонометрическое проецирование выполняется при произвольной ориентации объектной системы координат относительно мировой системы координат.

Математически проецирование реализуется путем умножения радиус-векторов точек объекта на *матрицу проецирования*. Такая матрица является вырожденной, что соответствует необратимому характеру проективного преобразования.

Обычно картинную плоскость выбирают параллельной одной из координатных плоскостей мировой системы координат (либо совпадающей с ней). По этому признаку проекции делятся на:

- *профильные (вид сбоку),* когда картинная плоскость параллельна плоскости YOZ мировой системы координат либо совпадает с ней;
- *горизонтальные (вид сверху),* когда картинная плоскость параллельна плоскости ZOX мировой системы координат либо совпадает с ней;
- *фронтальные (вид спереди),* когда картинная плоскость параллельна плоскости XOY мировой системы координат либо совпадает с ней.

Матрицы профильного, горизонтального и фронтального ортографического проецирования имеют вид (6.61а–с), где p,q,r – смещения картинных плоскостей относительно соответствующих координатных плоскостей мировой системы координат (рис. 3.8).

$$[P_x] = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p & 0 & 0 & 1 \end{pmatrix}, \qquad (3.61a)$$

$$[P_y] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & q & 0 & 1 \end{pmatrix}, \tag{3.61b}$$

$$[P_z] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & r & 1 \end{pmatrix}. \tag{3.61c}$$

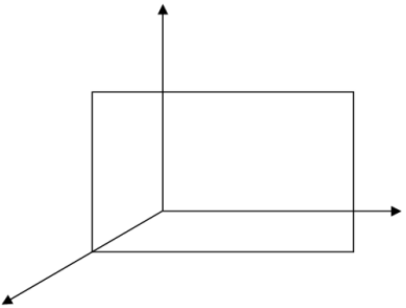

Рис. 3.8. Ортографическая проекция

Достоинство ортографического проецирования заключается в возможности точной передачи линейных и угловых размеров объектов, благодаря чему оно находит широкое применение в таких областях, как машиностроительное черчение и архитектура. В то же время при ортографическом проецировании полностью теряется информация об одном из пространственных измерений объекта. Восстановление трехмерного образа объекта возможно лишь при совместном использовании трех видов. Пример такого восстановления представлен в проекте «Example 20 Проекции».

3.6.2. Аксонометрическое проецирование

В случае аксонометрических проекций используются картинные плоскости, не перпендикулярные главным координатным осям мировой системы координат, поэтому на них изображаются сразу несколько сторон объекта. Для аксонометрических проекций проецирующие прямые также перпендикулярны картинной плоскости.

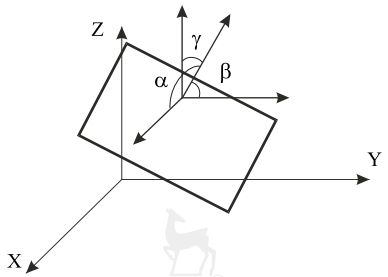

Рис. 3.9. Аксонометрическая проекция

На рисунке 3.9 показаны три угла – α, β и γ, определяющие ориентацию объектной системы координат относительно мировой системы координат. Различают три вида аксонометрических проекций:

- триметрическая проекция – нормальный вектор картинной плоскости образует с ортами координатных осей попарно различные углы;
- диметрическая проекция – два из трех указанных углов равны;
- изометрическая проекция – все углы равны.

Задача аксонометрического проецирования объекта на картинную плоскость решается в два этапа:

- выполняется поворот объектной системы координат до совмещения направления ее осей с осями мировой системы координат;
- выполняется ортографическое проецирование на картинную плоскость, параллельную (совпадающую) одной из координатных плоскостей мировой системы координат.

В дальнейшем, если не оговорено иное, мы будем рассматривать фронтальные проекции объектов, а в качестве картинной плоскости выбирать плоскость XOY мировой системы координат. В этом случае для выполнения первого этапа проецирования необходимо произвести поворот объектной системы координат на угол ψ вокруг оси OY мировой системы координат. Это обеспечит совпадения направлений осей абсцисс двух координатных систем. Затем производится поворот объектной системы координат на угол φ вокруг оси OX, после которого направления координатных осей объектной системы координат и мировой системы координат полностью совпадут. С учетом последующего ортографического проецирования полная матрица преобразования имеет вид

$$M = \begin{pmatrix} \cos\psi & 0 & -\sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

После перемножения матриц получим:

$$M = \begin{pmatrix} \cos\psi & \sin\varphi \times \sin\psi & 0 & 0 \\ 0 & \cos\varphi & 0 & 0 \\ \sin\psi & -\sin\varphi \times \cos\psi & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (3.62)$$

Элемент M_{ij} этой матрицы равен проекции i-го орта объектной системы координат на j-ю ось мировой системы координат. В общем случае триметрического проецирования углы поворотов ψ и φ выбираются произвольным образом.

Для дальнейшего удобно ввести *коэффициенты осевых искажений*:

$$m_x = \sqrt{(M_{11})^2 + (M_{12})^2},$$
$$m_y = \sqrt{(M_{21})^2 + (M_{22})^2}, \qquad (3.63)$$
$$m_z = \sqrt{(M_{31})^2 + (M_{32})^2},$$

которые являются проекциями ортов объектной системы координат на картинную плоскость (плоскость *XOY* мировой системы координат). В соответствии с вышеприведенной классификацией аксонометрических проекций при триметрическом проецировании эти коэффициенты должны быть попарно различны.

В диметрических проекциях два из трех коэффициентов осевых искажений равны друг другу. Это приводит к взаимной зависимости углов ψ и φ. В зависимости от способа выбора пары равных коэффициентов получим разные соотношения между углами, а именно:

$$\sin^2\psi = tg^2\varphi, \qquad (3.64a)$$

при $m_x = m_y \neq m_z$,

$$\cos^2\psi = tg^2\varphi, \qquad (3.64b)$$

при $m_y = m_z \neq m_x$ и, наконец,

$$\sin^2\psi = \cos^2\psi \qquad (3.64c)$$

при $m_x = m_z \neq m_y$.

В стандартной диметрической проекции соотношение коэффициентов искажения составляет

$$m_x : m_y : m_z = 2 : 2 : 1. \qquad (3.65)$$

Выражая коэффициенты осевых искажений через углы поворота и учитывая равенство $m_y^2 = 4m_z^2$, получим дополнительное условие

$$\cos^2\varphi = 4 \cdot (\sin^2\psi + \sin^2\varphi \cdot \cos^2\psi), \qquad (3.66)$$

которое совместно с (3.64) образует систему двух уравнений для углов ψ и φ. В случае, когда единичный вектор нормали к картинной плоскости лежит в

первом октанте мировой системы координат, имеем $\varphi > 0$ и $\psi < 0$. Тогда решение системы уравнений дает следующие значения для углов поворота:

$$\varphi = \arccos\left(\sqrt{\frac{8}{9}}\right) \approx 19,5^\circ, \quad \psi = -\arccos\left(\sqrt{\frac{7}{8}}\right) \approx -20,7^\circ. \qquad (3.67)$$

Матрица стандартного диметрического преобразования для этого случая имеет вид

$$M = \begin{pmatrix} 0,935 & -0,118 & 0 & 0 \\ 0 & 0,943 & 0 & 0 \\ -0,354 & -0,312 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (3.68)$$

Изометрические проекции получаются при условии равенства коэффициентов осевых искажений $m_x = m_y = m_z$, что приводит к равенствам

$$\begin{aligned} &\cos^2\psi + \sin^2\varphi \times \sin^2\psi = \cos^2\varphi, \\ &\sin^2\psi + \sin^2\varphi \times \cos^2\psi = \cos^2\psi + \sin^2\varphi \times \sin^2\psi. \end{aligned} \qquad (3.69)$$

Откуда следует, что

$$\sin^2\varphi = \frac{1}{3}, \quad \sin^2\psi = \frac{1}{2}. \qquad (3.70)$$

Стандартная изометрическая проекция получается в предположении, что, как и в предыдущем случае, единичный вектор нормали к картинной плоскости лежит в первом октанте мировой системы координат. Тогда

$$\varphi = \arccos(\sqrt{\frac{2}{3}}) \approx 35,3^\circ, \quad \psi = -\arccos(\sqrt{\frac{1}{2}}) \approx -45^\circ \qquad (3.71)$$

и матрица преобразования имеет вид

$$M = \begin{pmatrix} 0,707 & -0,408 & 0 & 0 \\ 0 & 0,816 & 0 & 0 \\ -0,707 & -0,408 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (3.72)$$

3.6.3. Косоугольное проецирование

При косоугольном проецировании пучок проецирующих лучей направлен под произвольным углом к картинной плоскости, а сама эта плоскость, как и при фронтальном ортографическом проецировании, параллельна фронтальной плоскости мировой системы координат. В этом случае удобно ввести косоугольную объектную систему координат, у которой оси абсцисс и ординат параллельны соответствующим осям мировой системы координат, а ось аппликат имеет направление, противоположное

направлению пучка проецирующих лучей. Тогда матрица косоугольного фронтального проецирования будет аналогична соответствующей ортографической матрице, отличаясь от нее ненулевыми значениями проекций третьего орта объектной системы координат на осях OX и OY мировой системы координат:

$$[K] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -c_x & -c_y & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad (3.73)$$

где $c_x = e_{3x}/e_{3z}$, $c_y = e_{3y}/e_{3z}$.

Коэффициенты осевых искажений при косоугольном проецировании оказываются равными:

$$m_x = m_y = 1, m_z = \sqrt{c^2{}_x + c^2{}_y} = tg\alpha, \qquad (3.74)$$

где α – угол между направлением проецирующего пучка и нормалью к картинной плоскости, то есть между осями аппликат объектной и мировой СК.

Таким образом, косоугольные проекции сочетают в себе свойства ортографических проекций (видов сбоку, сверху и спереди) со свойствами аксонометрии. В частности, сторона объекта, параллельная картинной плоскости, проецируется без искажения пропорций для углов и расстояний. Косоугольное проецирование применяется при построении теней, а также отражений в плоском зеркале.

Различают две стандартные косоугольные проекции: *свободную* (называемую также *военной*, или *кавальерной*) и *кабинетную*. Первая получается при условии $m_x = m_y = m_z = 1$. Это соответствует углу наклона проецирующих прямых к картинной плоскости, равному $\pi/4$. Кабинетной проекции соответствуют значения коэффициентов осевых искажений, равные $m_x = m_y = 1$ и $m_z = 0{,}5$. При этом угол наклона проецирующих прямых равен

$$\alpha = \text{arctg}(0{,}5) \approx 26{,}6^{\circ}.$$

3.6.4. **Центральное проецирование**

Обсуждение центральных проекций начнем с построения фронтальной проекции отдельной точки. Рассмотрим произвольную точку `P` с координатами (`x,y,z`). Поместим центр проецирования в точку `S(0,0,s)` и проведем из нее луч через точку `P`. Обозначим через `P*` точку пересечения луча с плоскостью XOY (рис. 3.10). Из простых геометрических соображений следует, что координаты этой точки равны:

`P*(x/(1-z/s),y/(1-z/s), 0).`

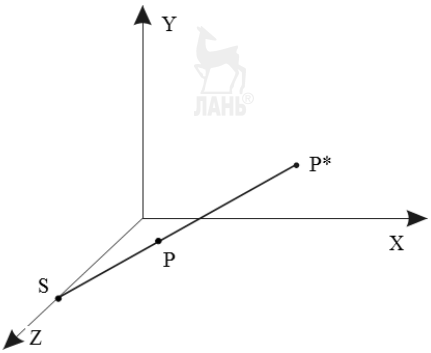

Рис. 3.10. Центральное проецирование

Этот же результат можно получить, выполнив следующее преобразование однородного вектора точки `P(x,y,z,1)` на матрицу центрального проецирования:

$$C_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/s \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (3.75)$$

Результатом этого преобразования является эквивалентный вектор

$$(x, y, 0, 1 - \frac{z}{s}),$$

которому соответствует однородный вектор

$$(x/\left(1-\frac{z}{s}\right),\ y/\left(1-\frac{z}{s}\right), 0, 1). \qquad (3.76)$$

Таким образом, задача центрального проецирования точки решается в два этапа:

- однородный вектор координат точки умножается на матрицу центрального проецирования;
- полученный эквивалентный вектор координат преобразуется в соответствующий однородный вектор.

Центральное проецирование обладает важным свойством, благодаря которому оно получило свое второе название – перспективное. Это свойство состоит в следующем: пучок параллельных прямых при центральном проецировании на картинную плоскость переходит в сходящийся пучок. Точка, в которой пересекаются проекции параллельных прямых, называется *точкой схода.*

Замечание. Пучок прямых не должен быть параллелен картинной плоскости.

Продемонстрируем это свойство на примере прямых, параллельных оси OZ. Параметрическое представление такой прямой в однородных координатах имеет вид

$$p(t) = (x_0, y_0, z_0 + vt, 1). \tag{3.77}$$

Выполнив преобразование центрального проецирования с матрицей (3.79), получим:

$$p(t) = \left(\frac{x_0}{1 - \frac{z_0 + vt}{s}}, \frac{y_0}{1 - \frac{z_0 + vt}{s}}, 0, 1 \right). \tag{3.78}$$

Устремляя значение параметра t к бесконечности, получим для точки схода

$$p_\infty(0, 0, 0, 1). \tag{3.79}$$

Таким образом, центральные проекции пучка прямых, параллельных оси OZ, при размещении проектора на этой оси имеют точку схода в начале координат (рис. 3.11).

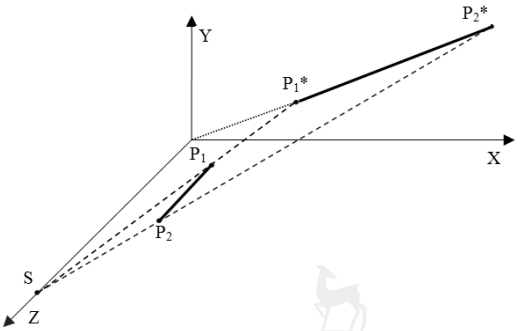

Рис. 3.11. Центральная проекция отрезка, параллельного оси OZ

Можно отделить перспективное преобразование от проецирования, введя матрицу вида

$$Q = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1/s \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.80}$$

Эта матрица преобразует вектор (3.82) к виду

$$p(t) = \left(\frac{x_0}{1 - \frac{z_0 + vt}{s}}, \frac{y_0}{1 - \frac{z_0 + vt}{s}}, \frac{z_0 + vt}{1 - \frac{z_0 + vt}{s}}, 1 \right),$$

что в пределе $t \to \infty$ дает для вектора координат точки схода значение (`0, 0, -s, 1`) (рис. 3.12).

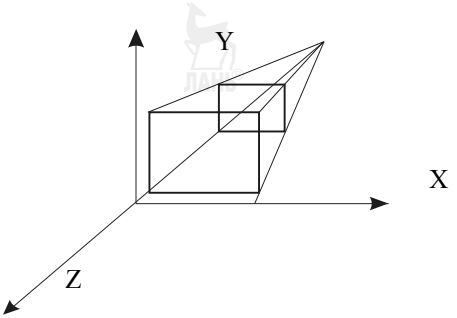

Рис 3.12. Положение точки схода на оси OZ

В общем случае произвольного положения проектора S и произвольного направления пучка параллельных прямых, задаваемого вектором V, образующим ненулевой угол с фронтальной картинной плоскостью, координаты точки схода определяются вектором

$$p_\infty = S - \frac{s_z}{V_z} V. \tag{3.81}$$

Матрица фронтального центрального проецирования в общем случае имеет вид

$$C_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -s_x / s_z & -s_y / s_z & 0 & -1/s_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.82}$$

Аналогичным образом могут быть определены матрицы для горизонтального

$$C_h = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -s_x / s_y & 0 & -s_z / s_y & -1/s_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.83}$$

и профильного

$$C_z = \begin{pmatrix} 0 & -s_y / s_x & -s_z / s_x & -1/s_x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.84}$$

центрального проецирования.

3.6.5. **Проект «Проекции»**

Для иллюстрации различных способов проецирования рассмотрим простое приложение, в котором строятся фронтальные проекции тетраэдра и куба (рис. 3.13).

В приложении используются две формы: *FormMain*, на которой строится изображение, и *FormTools*, используемая для задания опций (панель инструментов). Форма *FormTools* показана на рисунке 3.14.

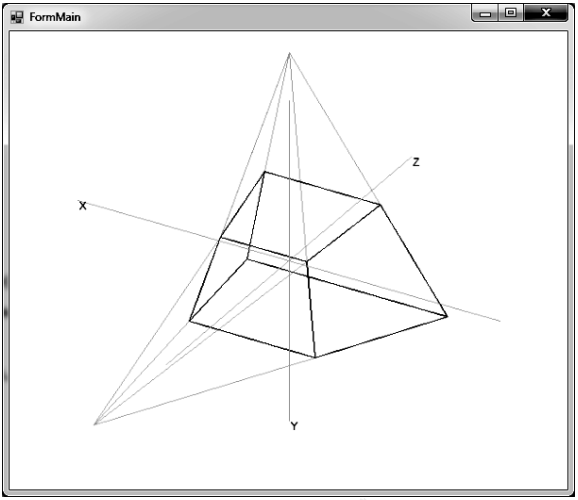

Рис. 3.13. Построение фронтальных проекций куба

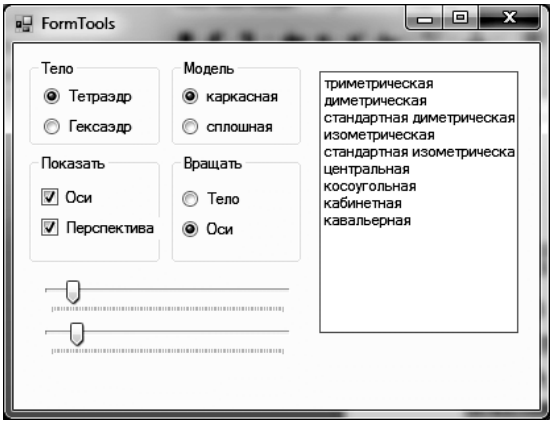

Рис. 3.14. Форма Tools с размещенной на ней панелью инструментов

В приложении реализованы следующие возможности:

- выбор способа проецирования;
- выбор модели трехмерного объекта – каркасной или сплошной;
- вращение тела или координатных осей;
- изменение масштаба изображения;
- задание углов проецирования для косоугольной проекции;
- задание положения точек схода для центральной проекции;
- построение перспективы при центральном проецировании.

Основные классы, используемые в проекте, описаны в файле *ThreeD.cs*. Здесь определены следующие классы:

- *ThreeD* – основной класс проекта, в котором описан массив тел, параметры проекта и методы преобразования системы координат;
- *TBody* – абстрактный класс тел, в котором определены массивы вершин, ребер и граней;
- *TEdge* – структура ребер;
- *TSide* – класс граней;
- *BodyTetraedr* – класс тетраэдра (потомок от *TBody*);
- *BodyHexaedr* – класс куба (потомок от *TBody*).

Общая структура классов представлена на диаграмме (рис. 3.15).

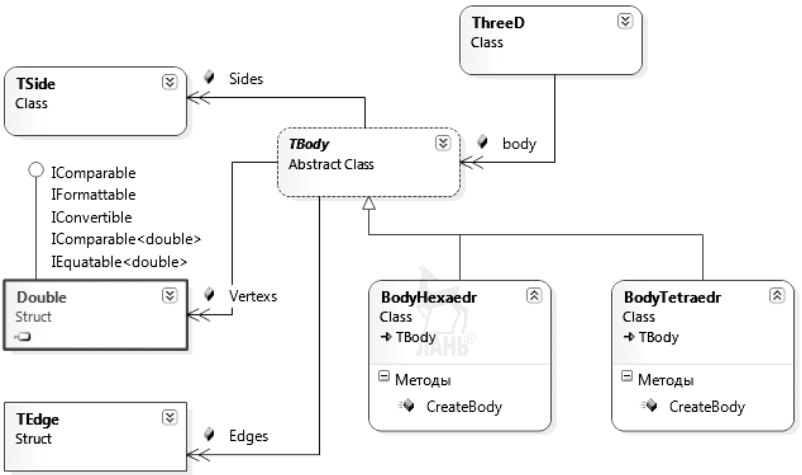

Рис. 3.15. Диаграмма классов

Основным полем класса *TD* является массив тел

```
public TBody[] body = new TBody[2];
```

Для преобразования системы координат класс *TD* обладает следующими статическими методами:

– *double[] Norm(double[] V1, V2, V3)* – метод вычисления нормали к грани;
– *double[,] Matr(int k, double fi, double p, double r)* – метод вычисления матрицы преобразования в различных системах координат;
– *double[] VM_Mult(double[] A, double[,] B)* – метод умножения матрицы на вектор;
– *double[] Rotate(double[] V, int k, double fi, double p, double r)* – метод преобразования координат вершин.

Класс тел *TBody* обладает следующими полями и методами.

Листинг 3.2. Абстрактный класс тел

```
public abstract class TBody
{
  public double[][] Vertexs;
  // массив начальных положений вершин всех тел
  public TEdge[] Edges; // массив ребер всех тел
  public TSide[] Sides; // массив граней всех тел
  public abstract void CreateBody(double Size);
}
```

Абстрактный метод *CreateBody()* перекрывается в потомках *BodyTetraedr* и *BodyHexaedr*.

Структура ребер *TEdge*.

Листинг 3.3. Структура для описания ребер

```
public struct TEdge
{
    public int p1, p2;        //номера вершин
}
```

Два поля *p1* и *p2* содержат номера вершин, на которые опирается ребро.

Класс граней.

Листинг 3.4. Класс граней

```
public class TSide
{
    public int[] p = new int[4];  // номера вершин
    double A, B, C, D;
    //коэффициенты уравнения плоскости
    double[] N; // вектор нормали к плоскости
}
```

p[] – номера вершин;

A, B, C, D – коэффициенты уравнения плоскости;

N – вектор нормали к плоскости.

Остановимся более подробно на методах класса *TD*. Конструктор этого класса создает два экземпляра тел.

Листинг 3.5. Конструктор класса TD

```
public TD()
{
    body[0] = new BodyTetraedr();
    body[0].CreateBody(1);
    body[1] = new BodyHexaedr();
    body[1].CreateBody(1);
}
```

Метод *Norm()* получает три вершины грани, вычисляет по ним два вектора и по их векторному произведению вычисляет вектор единичной нормали.

Листинг 3.6. Метод вычисления вектора нормали

```
double[] Norm(double[] V1,double[] V2,double[] V3)
{
   double[] A = new double[4];
   double[] B = new double[4];
   double[] Result = new double[4];
   A[0]=V2[0]-V1[0]; A[2]=V2[1]-V1[1]; A[3]=V2[2]-V1[2];
   B[0]=V3[0]-V1[0]; B[2]=V3[1]-V1[1]; B[3]=V3[2]-V1[2];
   double u = A[1]*B[2]-A[2]*B[1];
   double v =-A[0]*B[2]+A[2]*B[0];
   double w = A[0]*B[1]-A[1]*B[0];
   double d=Math.Sqrt(u*u+v*v+w*w);
   if (d!=0)
   {
      Result[0]=u/d;
      Result[1]=v/d;
      Result[2]=w/d;
   }
   else
   {
      Result[0]=0;
      Result[1]=0;
      Result[2]=0;
   }
   return Result;
}
```

Метод *Matr()* вычисляет матрицу поворота вокруг осей, матрицу перспективного или косоугольного преобразования системы координат.

Листинг 3.7. Метод вычисления матрицы преобразования системы координат

```
public static double[,] Matr(int k, double fi,
       double p, double r)
{
   double[,] M = new double[4,4];
   double[,] I = { { 1, 0, 0, 0 }, { 0, 1, 0, 0 },
                   { 0, 0, 1, 0 }, { 0, 0, 0, 1 } };
   M=I;
   switch (k)
   {
      case 1:  // Матрица поворота вокруг оси OX
          M[1,1] = Math.Cos(fi);
          M[1,2]=Math.Sin(fi);
          M[2,1] =-Math.Sin(fi);
          M[2,2]=Math.Cos(fi);
      break;
      case 2:  // Матрица поворота вокруг оси OY
          M[0,0]=Math.Cos(fi); M[0,2]=-Math.Sin(fi);
          M[2,0]=Math.Sin(fi); M[2,2]=Math.Cos(fi);
      break;
      case 3: // Матрица поворота вокруг оси OZ
          M[0,0]= Math.Cos(fi); M[0,1]=Math.Sin(fi);
          M[1,0]=-Math.Sin(fi); M[1,1]=Math.Cos(fi);
      break;
      case 4: // перспективное преобразование
          M[1,3]= p; M[2,3]=r;
      break;
      case 5: // косоугольное проецирование
          M[2,0]=p; M[2,1]=r;
      break;
    }
    return M;
 }
```

Метод *VM_Mult()* умножает матрицу *B* на вектор *A*.

Листинг 3.8. Метод умножения матрицы на вектор

```
static double[] VM_Mult(double[] A, double[,] B)
{
    double[] Result = new double[4];
    for (int j=0; j<=3; j++)
    {
        Result[j]=A[0]*B[0,j];
        for (int k=1; k<=3;k++)
```

```
            Result[j]=Result[j]+A[k]*B[k,j];
    }
    if (Result[3]!=0)
        for (int j=0; j<= 2;j++)
           Result[j]=Result[j]/Result[3];
    Result[3]=1;
    return Result;
}
```

Построение изображения тела в этом методе разбивается на ряд этапов. Для всех вершин тела выполняется операция поворота и проецирования. Исходные значения координат копируются в массив *VertexsT* и преобразуются с помощью метода *Rotate()*. В качестве аргументов этому методу передаются следующие величины: координаты точки, индекс преобразования *k*, параметры преобразования *fi, p, r*. Метод обращается к методу *Matr()*, формирующему матрицу преобразования в соответствии с заданными параметрами. После вызова этого метода производится перемножение вектора, задающего положение вершины, и матрицы преобразования. Ниже приведен листинг метода *Rotate()*.

Листинг 3.9. Метод Rotate()

```
public static double[] Rotate(double[] V, int k,
    double fi, double p, double r)
{
    double[,] M = Matr(k,fi,p,r);
    double[] Result = VM_Mult(V,M);
    return Result;
}
```

Модуль *FormMain.cs* включает методы построения изображения, обработки событий, связанных с манипулятором «мышь», а также ряд вспомогательных методов.

Основным является метод построения изображения *DrawBody()*, текст которого приведен в листинге 3.10.

Листинг 3.10. Метод построения изображения

```
public void DrawBody()
{
   double ax = 7;
   using (Graphics g = Graphics.FromImage(bitmap))
   {
      Point P0, P;
      Color cl = Color.FromArgb(255, 255, 255);
      g.Clear(cl);

      // изображение тела
```

```
byte k = TD.flBody;
int Lv = td.body[k].Vertexs.Length;
for (int i = 0; i < Lv; i++)
{
   for (int j = 0; j <= 3; j++)
      td.body[k].V[i][j] =
         td.body[k].Vertexs[i][j];
   td.body[k].V[i] =
         TD.Rotate(td.body[k].V[i], 2,
         TD.Bet1, 0, 0);
   td.body[k].V[i] =
      TD.Rotate(td.body[k].V[i], 1,
      TD.Alf1, 0, 0);
   td.body[k].V[i] =
      TD.Rotate(td.body[k].V[i], 4, 0,
      TD.Xs, TD.Zs);
   td.body[k].V[i] =
      TD.Rotate(td.body[k].V[i], 2,
      TD.Bet, 0, 0);
   td.body[k].V[i] =
      TD.Rotate(td.body[k].V[i], 1,
      TD.Alf, 0, 0);
}
// точки схода
double[] V1 = new double[4];
double[] V2 = new double[4];
int u,v;
if (TD.Zs!=0)
{
   V2[0]=0; V2[1]=0; V2[2]=1/TD.Zs; V2[3]=1;
   V2=TD.Rotate(V2, 2, TD.Bet, 0, 0);
   V2=TD.Rotate(V2, 1, TD.Alf, 0, 0);
   u=II(V2[0]); v=JJ(V2[1]);
   for (int i = 0; i <= 3; i++)
   {
      int a2 = II(td.body[k].V[i][0]);
      int b2 = JJ(td.body[k].V[i][1]);
      g.DrawLine(Pens.Silver, u, v, a2, b2);
   }
 }
 if (TD.Xs != 0)
 {
    V1[1] = 1 / TD.Xs; V1[0] = 0;
    V1[2] = 0; V1[3] = 1;
    V1 = TD.Rotate(V1, 2, TD.Bet, 0, 0);
    V1 = TD.Rotate(V1, 1, TD.Alf, 0, 0);
    u = II(V1[0]); v = JJ(V1[1]);
    for (int i = 0; i <= 5; i++)
```

```
            if (i != 3)
            {
               int a2 = II(td.body[k].V[i][0]);
               int b2 = JJ(td.body[k].V[i][1]);
               g.DrawLine(Pens.Silver, u, v, a2, b2);
            }
         }
         switch (TD.flModel)
         {
            case 0:     // каркасная модель
            case 1:
              int L = td.body[k].Edges.Length;
              for (int i = 0; i < L; i++)
              {
                 int a1 =
          II(td.body[k].V[td.body[k].Edges[i].p1][0]);
                 int b1 =
          JJ(td.body[k].V[td.body[k].Edges[i].p1][1]);
                 int a2 =
          II(td.body[k].V[td.body[k].Edges[i].p2][0]);
                 int b2 =
          JJ(td.body[k].V[td.body[k].Edges[i].p2][1]);
                  g.DrawLine(Pens.Black, a1,b1,a2,b2);
               }
               break;
            }
            // прорисовка осей
            if (TD.visibleОси)
            {
               P0=IJ(ToVector(-ax,0,0));
               P=IJ(ToVector(ax,0,0));
               g.DrawLine(Pens.Silver, P0, P);
               g.DrawString("X",Font,Brushes.Black,
                   P.X,P.Y);
               P0 = IJ(ToVector(0,-ax, 0));
               P = IJ(ToVector(0, ax, 0));
               g.DrawLine(Pens.Silver, P0, P);
               g.DrawString("Y",Font, Brushes.Black,
                   P.X, P.Y);
               P0 = IJ(ToVector(0,0,-ax));
               P = IJ(ToVector(0, 0, ax));
               g.DrawLine(Pens.Silver, P0, P);
               g.DrawString("Z",Font, Brushes.Black,
                   P.X, P.Y);
            }
         }
         g0.DrawImage(bitmap, ClientRectangle);
      }
```

При условии выбора соответствующей опции производится прорисовка координатных осей. В этой части метода пространственные координаты начальной и конечных точек отрезков, изображающих координатные оси, отображаются в экранные координаты. Метод *IJ* (листинг 3.11) выполняет проецирование на картинную плоскость XOY и отображение «бумажных» координат в экранные.

Листинг 3.11. Вычисление экранных координат точки

```
Point IJ(double[] Vt)
{
    switch (TD.flProection)
    {  // Аксонометрические и центральная проекции
     case 0: case 1: case 2: case 3: case 4: case 5:
          Vt = TD.Rotate(Vt, 2, TD.Bet, 0, 0);
          Vt = TD.Rotate(Vt, 1, TD.Alf, 0, 0);
          break;
       // Косоугольные проекции
       case 6: case 7: case 8:
          Vt =TD.Rotate(Vt, 5, 0, TD.q, TD.r);
          break;
    }
    Point Result = new Point(II(Vt[0]), JJ(Vt[1]));
    return Result;
}
```

Если выбрана опция изображения перспективы, производится построение точек схода. Кроме того, для наглядности штриховыми линиями строится изображение куба с учетом перспективы. Разумеется, этот куб невидим, если в качестве многогранника выбран гексаэдр.

Строится изображение тела в одной из двух моделей – каркасной или сплошной. В первом случае построение сводится к проведению отрезков, соответствующих ребрам многогранника. Во втором случае для каждой грани строится вектор нормали и производится заливка грани тем или иным цветом в зависимости от ее ориентации относительно оси OZ.

Обработчики событий *OnMouseDown* и *OnMouseUp* выполняют соответственно установку и снятие флага рисования (переменная drawing). Обработчик *FormMouseMove* содержит код, имитирующий вращение тела при перемещении мыши. Код метода показан в листинге 3.12.

Листинг 3.12. Обработчик события перемещения мыши

```
private void FormMain_MouseMove(object sender,
    MouseEventArgs e)
{
   if (drawing)
   {
```

```
        double a = e.X - Width / 2.0;
        double b = e.Y - Height / 2.0;
        switch (TD.flRotate)
        {
           case 1:
              switch (TD.flModel)
              {
                 case 0: case 1: case 5:
                    TD.Alf = Math.Atan2(b, a);
                    TD.Bet =
                       Math.Sqrt((a/10)*(a/10)+
                          (b/10)*(b/10));
                    break;
                 case 3:
                    break;
              }
              break;
           case 0:
              TD.Alf1 = Math.Atan2(b, a);
              TD.Bet1 =
                 Math.Sqrt((a/10)*(a/10)+
                    (b/10)*(a/10));
              break;
        }
        DrawBody();
     }
}
```

В зависимости от выбора опции вращения на панели инструментов имитируется либо вращение тела при неподвижных координатных осях (*flRotate* = 0), либо совместное вращение тела и координатных осей (*flRotate* = 1). В первом случае координатные оси являются осями мировой системы координат, во втором – объектной системы координат. Углы поворота осей объектной системы координат *Alf* и *Bet*, а также углы *Alf1* и *Bet1* поворота тела относительно осей мировой системы координат вычисляются через текущие координаты X и Y указателя мыши. Способ вычисления углов *Alf* и *Bet* зависит от выбранной проекции.

Стандартные проекции предполагают выбор определенных ориентаций тела относительно координатных осей, поэтому для них вращение не производится.

3.7. Моделирование трехмерных тел

В компьютерной графике используются несколько типов моделей пространственных объектов, классификация которых представлена на рисунке 3.16.

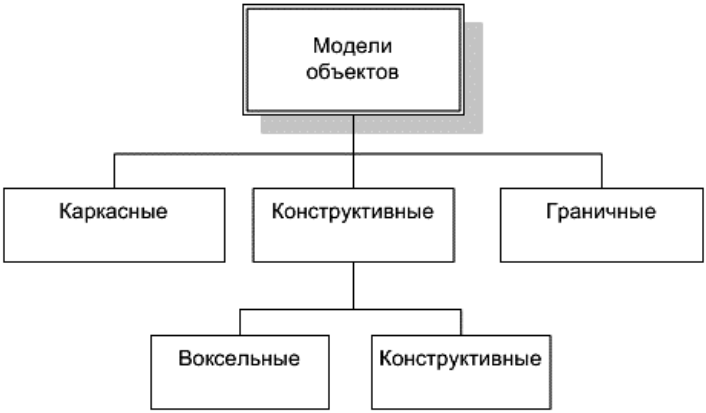

Рис. 3.16. Классификация моделей пространственных объектов

Выбор той или иной модели предопределяет способы решения таких проблем, как:

- выделение видимых частей объекта;
- назначение цвета каждому элементу объекта;
- воспроизведение различных оптических эффектов.

3.7.1. Каркасные модели

В простейшем варианте каркасная модель изображает тело в виде проволочной сетки с целью передачи формы охватывающих его поверхностей – граней. В случае граней для этого достаточно построения ребер. Для построения каркасной модели тела с неплоскими граничными поверхностями используются различные способы кусочно-линейной интерполяции. В пункте 6.2 и в главах 8 и 9 описаны проекты, реализующие примеры построения каркасных моделей многогранников и поверхностей второго порядка.

В более сложных вариантах каркасной модели передается эффект разноудаленности от наблюдателя отдельных граней объекта путем выделения особым стилем (например, пунктиром) или удалением невидимых ребер. Еще большая реалистичность изображения достигается при использовании заливки граней, когда оттенок цвета зависит от пространственной ориентации грани и места нахождения источника света.

Для построения каркасной модели тела необходимо:

- пронумеровать его вершины, ребра и грани;
- задать координаты всех вершин относительно начала координат или некоторой базовой точки;
- описать топологию каркаса путем формирования для каждой из граней списка номеров инцидентных ей вершин, перечисленных в

порядке их обхода в определенном едином для всех граней направлении.

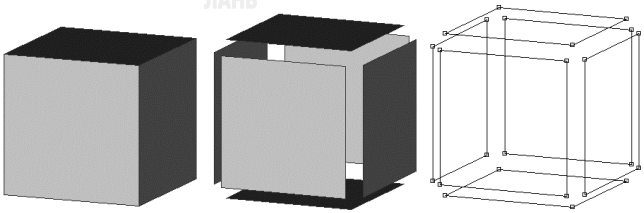

Рис. 3.17. Основные составляющие каркасной модели

3.7.2. Граничные, поверхностные модели

Граничная модель (boundary representation) представляет трехмерный объект как систему поверхностей, создающих его границы. Каждая из таких поверхностей описывается неявным уравнением или параметрической функцией с указанием границ изменения координат или параметров, что позволяет определить линии пересечения поверхностей. Граничная модель рассматривает только точки на поверхности тела, не затрагивая его объема.

В отличие от каркасной модели с плоскими гранями граничное описание позволяет изображать нелинейную поверхность каркасом с криволинейными ячейками.

3.7.3. Сплошные модели

Сплошные, или твердотельные, модели описывают точки как внутри, так и на поверхности объекта. Трехмерные тела рассматриваются как объединение (композиция) некоторых базовых блоков. Тип базовых объектов определяет различные методы моделирования этим способом.

3.7.3.1. Воксельное представление

В качестве базового блока выберем кубическую ячейку пространства – воксел. Часть пространства представим трехмерным битовым массивом c_{ijk}. Элементы массива $c_{ijk} = 1$, если куб содержится в объекте, $c_{ijk} = 0$, если куб не содержится в объекте (рис. 3.18).

Воксельная модель позволяет легко вычислить такие параметры, как объем или центр масс, и выполнять бинарные операции над телами. Однако для хорошего качества изображения требуется большой размер памяти, растущий как N^3.

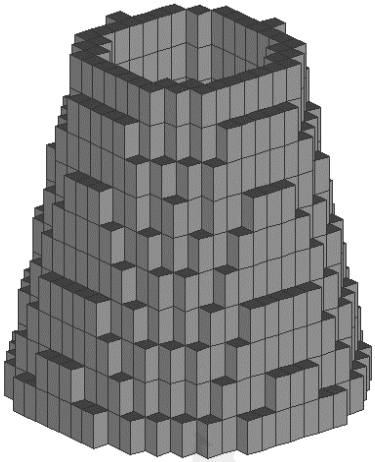

Рис. 3.18. Воксельное представление усеченного конуса
с цилиндрическим отверстием

3.7.3.2. Конструктивное моделирование

В этом случае объект задается набором плоских или объемных примитивов и операций над ними. Примитивы являются «строительными блоками» объекта и обычно описываются граничной моделью. Под операциями понимаются булевы операции над примитивами, а также геометрические преобразования, такие как перемещение, поворот, изменение размеров. Полная конструктивная модель объекта включает:

- данные о типе каждого примитива и его граничной модели, включая оптические свойства поверхностей;
- последовательность логических операций над примитивами при конструировании объекта;
- последовательность аффинных преобразований на каждом шаге конструирования.

Процесс конструирования можно представить в виде бинарного дерева с правилом обхода «снизу-вверх». В этом дереве листьями являются геометрические примитивы, а каждому узлу сопоставляется операция. Вершиной дерева является геометрический объект (рис 3.19).

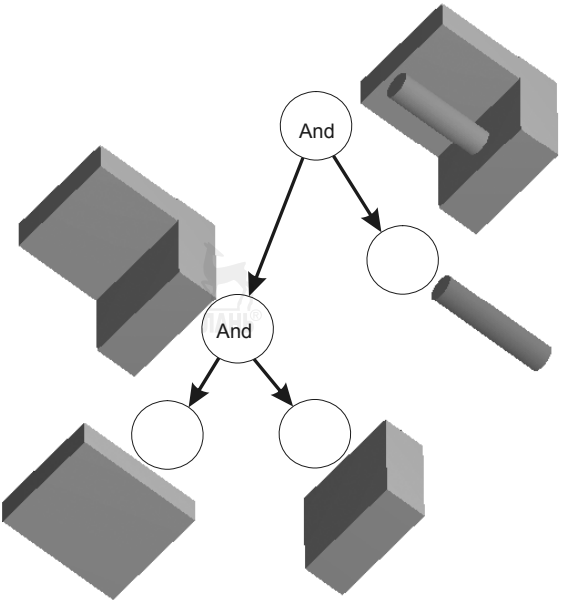

Рис. 3.19. Дерево конструктивной модели

3.8. Освещение

Для придания изображению большей реалистичности модели трехмерных объектов включают методы передачи различных оптических эффектов. Одним из основных таких эффектов является освещенность различных участков поверхности тел.

Любое тело становится видимым благодаря отражению падающего на его поверхность светового излучения. Это излучение изначально создается источниками света, однако при распространении в любой неидеально прозрачной среде рассеивается, отклоняясь от первоначального направления. Кроме того, происходит отражение прямых и рассеянных лучей от поверхностей различных тел, а также их преломление при переходе через границы раздела сред. В большинстве моделей световой поток, падающий на поверхность объекта, состоит из излучения от источников света и фонового излучения.

Интенсивность *фонового* излучения I_F постоянна для всех точек поверхности объекта. Часть фонового излучения поглощается объектом и часть отражается от его поверхности. Предполагается, что интенсивность его отраженной части равна $k_F \times I_F$, где k_F – коэффициент отражения ($0 \le k_F \le 1$).

Коэффициент k_F является функцией длины волны, что проявляется в наличии собственного цвета у отражающей поверхности.

Если есть источники света, то возникает дополнительная освещенность поверхности тела из-за отражения направленного излучения двух типов: *диффузного* и *зеркального*.

В компьютерной графике для описания зеркального отражения используется эмпирическая модель Фонга (Bui Tuong Phong, 1975). Суть ее заключается в том, что интенсивность зеркально отраженного луча зависит от угла ψ между отраженным лучом M и вектором S, направленным к наблюдателю (рис. 3.20).

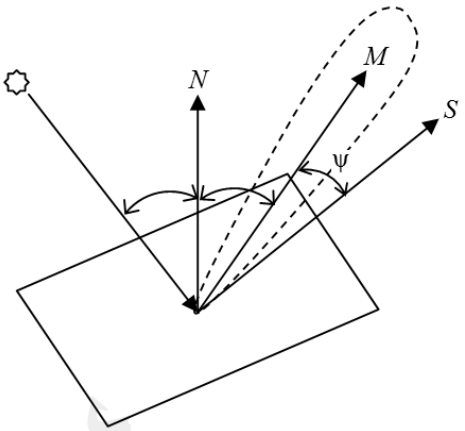

Рис. 3.20. Зеркальное отражение направленного излучения

В этой модели интенсивность зеркально отраженного излучения определяется по формуле

$$I_s = k_s(\varphi, \lambda) \times I \cos^n(\psi), \tag{3.85}$$

где $k(\varphi, \lambda)$ – коэффициент отражения; φ – угол падения света; λ – длина световой волны. Степень, в которую возводится косинус угла, влияет на размеры светового блика, наблюдаемого зрителем. Графики функции $\cos^n(\psi)$ приведены на рисунке 3.21 и являются характерными кривыми поведения функции изменения интенсивности в зависимости от свойств поверхности.

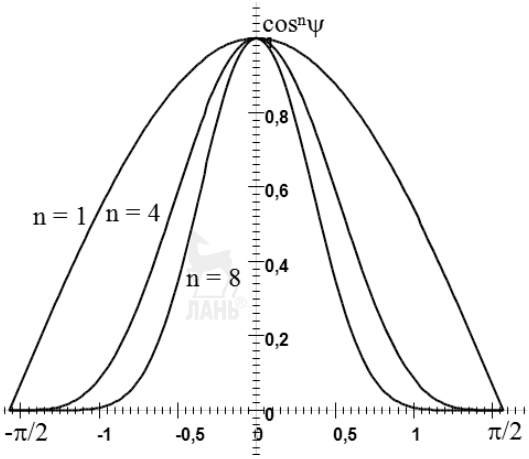

Рис. 3.21. Графики функции $\cos^n(\psi)$

Диффузное отражение излучения зависит от шероховатости поверхности и взаимодействия излучения с глубокими слоями материала. Интенсивность диффузного отраженного света пропорциональна *cos(φ)*:

$$I_d = \begin{cases} k_d(\lambda) \times I \times \cos(\varphi), & 0 \le \varphi \le \pi/2, \\ 0, & \varphi > \pi/2. \end{cases} \tag{3.86}$$

В описанной модели пока никак не учитывалась удаленность источника света от поверхности, поэтому по освещенности двух объектов нельзя судить об их взаимном расположении в пространстве. Для получения перспективного изображения необходимо учитывать зависимость интенсивности от расстояния. Обычно предполагают, что интенсивность света обратно пропорциональна квадрату расстояния до источника или линейна. Модель освещенности, учитывающая фоновое освещение, зеркальное и диффузное отражения, можно описать следующей формулой:

$$I = I_F k_F(\lambda) + \frac{I(k_D(\lambda)\cos(\varphi) + k_S(\varphi,\lambda) \cdot \cos^n(\psi))}{r + C}, \tag{3.87}$$

где r – расстояние от центра проекции; C – некоторая константа. Если центр проекции находится в бесконечности, то есть при параллельном проецировании, то в качестве r берут расстояние до ближнего объекта.

Необходимо отметить, что нет более или менее точного алгоритма вычисления коэффициентов диффузионного D и зеркального S отражений.

Рассмотрим поверхность с единичной нормалью $\bar{N}$. Пусть $\bar{L}$ – единичный вектор в направлении источника света, $\bar{M} = \bar{L} + \bar{k}$. Наблюдение ведется со стороны оси OZ с единичным вектором $\bar{k}$. Коэффициенты

диффузионного D и зеркального S отражений зависят от параметров (рис. 3.24):

- угла l между направлением к источнику света и нормалью к поверхности;
- угла m между вектором $\overline{M}$ и нормалью к поверхности;
- максимального коэффициента диффузионного отражения *Cd*;
- шероховатости материала *Kd*;
- максимального коэффициента отражения поверхностного слоя *Cs*;
- шероховатости поверхностного слоя *Ks*.

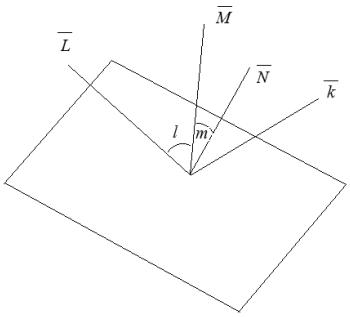

Рис. 3.22. Параметры для коэффициента диффузионного освещения

Для определения коэффициентов отражения в основном используются модели диффузного и зеркального отражений. Значения вышеперечисленных параметров для этих моделей приведены в таблице 3.1.

Таблица 3.1

Модели освещения

Диффузионное отражение	**Зеркальное отражение**
$D = Cd \times \cos l$	$S = Cs \times \cos l \times \cos m$
$D = Cd \times \cos \lvert l \rvert$	$S = 0$
$D = \cos^2 l/zs$	$S = 0$
$D = \cos^{kd} l\ 0,\ l > Pi/2$	$S = 0$
$D = \cos l/Ds$	$S = Cs \cos^{ks} l/Ds$
$D = \lvert\cos l\rvert/Ds$	$S = \cos^{ks} l/Ds$
$D = Cd \times \cos l$	$S = Cs \times \cos^{ks} m$
$D = Cd \times Exp(-(l/Kd))^2$	$S = Cs \times Exp(-(m/Kc))^2$

Модель освещения для любой точки определяется следующими выражениями:

```
ColorR = max(max(1,A+D)*CmatR+s,1)*CLightR;
ColorG = max(max(1,A+D)*CmatG+s,1)*CLightG;
ColorB = max(max(1,A+D)*CmatB+s,1)*CLightB,
```
(3.88)

где `A` — интенсивность рассеянного света; `CLightR`, `CLightG`, `CLightB` — красная, зеленая, синяя составляющие цвета источника света; `CmatR`, `CmatG`, `CmatB` — красная, зеленая, синяя составляющие цвета поверхности материала. Все параметры меняются в пределах от 0 до 1.

3.9. Моделирование цвета

Основным программным инструментом конструирования цвета графического класса Graphics является класс Color, который позволяет назначить любой из 256 × 256 × 256 = 16777216 цветов.

При этом интенсивность цвета m(Color) равна сумме интенсивностей красного (`R`), зеленого (`G`) и синего (`B`) цветов: `m(Color) = = r(R)+g(G)+b(B)`.

Получение любого цвета из трех основных (`R,G,B`) продиктовано технологией изготовления электронно-лучевых трубок и является достаточно искусственным приемом. Для оценки цветового восприятия удобнее пользоваться другими параметрами: цветовым тоном (`T`), яркостью (`L`) и насыщенностью (`S`).

Цветовой тон определяется длиной волны, преобладающей в потоке излучения (усредненная длина волн).

Яркость соответствует световой энергии, достигающей наблюдателя. Можно сказать, что яркость – это интенсивность света, искаженная сетчаткой глаза. Человеческий глаз может различать порядка тысячи уровней яркости. Насыщенность (чистота цвета) характеризует долю белого цвета в чистых спектральных цветах. Вместо параметров (*R,G,B*) можно использовать параметры (`T,L,S`) в трехмерном пространстве с цилиндрическими координатами (рис. 3.23).

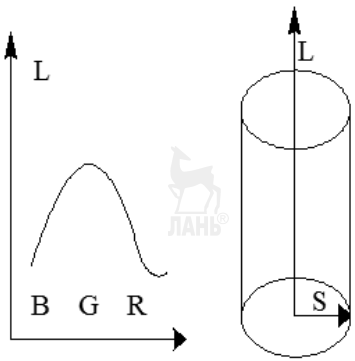

Рис. 3.23. Интенсивность света и параметры (*T*, *L*, *S*)

Переход от (*N,L,S*) к (*R,G,B*) и обратно определяется формулами:

```
T = arcsin(Sqrt(3/2)*(G-R)/S);
L = (R+G+B)/3;
S = Sqrt(R*R+G*G+B*B-R*G-R*B-G*B);.....      (3.89)
R = L-S*cos(T)/3-S*sin(T)/Sqrt(3);
G = L+2*S*cos(T)/3;
B = L-S*cos(T)/3+S*sin(T)/Sqrt(3).
```

Параметры *T,L,S*, во-первых, удобно использовать для интерактивного выбора цвета на экране монитора. Для этого необходимо представить на экране сокращенную палитру при фиксированной яркости *L*, а затем увеличить или уменьшить яркость.

Второй привлекательный момент параметров *T,L,S* заключается в удобстве использования при моделировании затененности объектов или сумерек (линейное изменение яркости от начального цвета до черного) и при моделировании дымки (линейное изменение яркости от начального цвета до серого).

3.10. Удаление невидимых ребер и граней

Алгоритм Робертса предназначен для удаления невидимых линий трехмерного выпуклого объекта. Невыпуклые объекты необходимо разбить на выпуклые части. Сначала из объекта удаляются ребра и грани, экранируемые объектом. Далее каждое из видимых ребер разбивается на видимые и невидимые части. Вычислительная мощность алгоритма Робертса равна квадрату числа объектов. В более поздних реализациях алгоритма используются предварительная сортировка по оси *z* и габаритные или минимаксные тесты, наблюдается линейная зависимость от числа объектов.

Алгоритм Робертса состоит из следующих шагов:

- нахождения нелицевых граней всех объектов;
- нахождения и удаления невидимых ребер.

Плоскость, на которой находится некоторая грань многогранника *F*, разделяет пространство на два подпространства. Подпространство называется положительным D^+, если в него смотрит нормаль к грани. Если точка наблюдения находится в D^+, то грань – *лицевая*, иначе – *нелицевая*. Для выпуклых многогранников удаление нелицевых граней полностью решает задачу удаления невидимых граней.

Для определения, находится ли точка в D^+, необходимо определить знак скалярного произведения (*l,n*), где *l* – вектор, определяющий точку наблюдения; *n* – вектор внешней нормали грани. Если (*l,n*) > 0, то грань – лицевая, иначе – нелицевая.

Выпуклый многогранник представляется набором плоскостей. Уравнение плоскости в 3D пространстве имеет вид

$$|x, y, z, 1| \times \begin{vmatrix} a \\ b \\ c \\ d \end{vmatrix} = 0, \tag{3.90}$$

где $P = |a, b, c, d|$ – коэффициенты плоскости; $S = |x, y, z, 1|$ – однородные координаты. Для точки S, лежащей на плоскости, скалярное произведение $S \cdot P = 0$. Если точка S не лежит на плоскости, то знак скалярного произведения показывает расположение точки относительно плоскости. Для точек, лежащих внутри тела, скалярное произведение имеет отрицательный знак.

Глава 4. ПРОСТЫЕ ГРАФИЧЕСКИЕ ПРОЕКТЫ

Материал предыдущих глав содержит достаточно много технических деталей. Далее представлены иллюстрации в виде реализованных проектов, демонстрирующие некоторые свойства и методы описанных классов. При подборе этих демонстрационных задач мы, с одной стороны, постарались охватить вопросы, представленные в справочном материале первых глав, с другой стороны, представить наиболее интересные с практической точки зрения задачи.

4.1. Мультипликация

Изменение изображений во времени возможно двумя способами: во-первых, можно воспользоваться методом временной задержки *Sleep(int millisecondsTimeout)* класса *Thread*; во-вторых, воспользоваться классом Timer, имеющим один обработчик события *onTimer*.

Демонстрацию метода *Sleep()* покажем на примере сортировки элементов массива, а использование компонента *Timer* – на морфинге и в проекте, показывающем движение велосипеда.

4.1.1. Сортировка элементов массива

Сортировка методом пузырька – метод, в полной мере воплощающий принцип *обменной* сортировки: сравниваются и обмениваются местами два соседних элемента. Алгоритм очень прост и приведен в листинге 4.1.

Листинг 4.1. Сортировка методом пузырька

```
void Sort()
{
    int L = Element.Length;
    for (int i = 1; i<=L-1; i++)
       for (int j = i; j<=L-1; j++)
       {
           TElement tmp;
           if (Element[j-1].inf>Element[j].inf)
          {
              tmp = Element[j];
              Element[j] = Element[j-1];
              Element[j - 1] = tmp;
          }
      }
 }
```

Этот алгоритм представлен в наглядном графическом виде в предлагаемом проекте (рис. 4.1), в котором миганием элементов обозначается их сравнение, а обмен происходит в динамике: верхний элемент уходит в сторону, а нижний становится на его место.

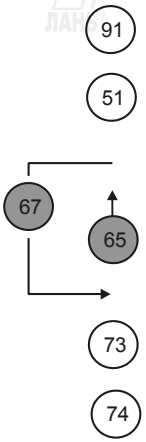

Рис. 4.1. Сортировка методом пузырька

Для описания элементов массива используется структура, представленная в листинге 4.2.

Листинг 4.2. Структура элементов массива

```
public struct TElement
{
    public int x;
    public int y;
    public int inf;
    public Color color;
}
```

В этой структуре поля *x,y* целого типа отвечают за положение элемента на экране, поле *inf* содержит значение элемента массива, поле *color* используется для цвета, которым изображается элемент.

При нажатии кнопки случайным образом заполняются элементы массива, назначаются их начальные координаты на экране (листинг 4.3) и вызывается метод сортировки.

Листинг 4.3. Начальное заполнение массива

```
void SetRandom()
{
      int L = Element.Length;
```

```
        Random rnd = new Random();
        for (int i = 0; i <= L - 1; i++)
        {
           Element[i].x = 100;
           Element[i].y = 20 + i * 40;
           Element[i].color = Color.Black;
           Element[i].inf = rnd.Next(100);
        }
}
private void buttonRandom_Click(object sender, EventArgs e)
{
    SetRandom();
    Drawing(-1,-1);
}
```

Метод сортировки, представленный в листинге 4.3, слегка изменен: добавлен метод динамического обмена местами двух элементов *Change(j,j-1)*.

Листинг 4.4. Метод сортировки

```
void Sort()
{
    int L = Element.Length;
    for (int i = 1; i<=L-1; i++)
       for (int j = i; j<=L-1; j++)
       {
           TElement tmp;
           if (Element[j-1].inf>Element[j].inf)
           {
               Change(j,j - 1, i,j);
               tmp = Element[j];
               Element[j] = Element[j-1];
               Element[j - 1] = tmp;
           }
       }
}
```

Метод динамического обмена местами двух элементов *Change()* работает 15 тактов. За эти 15 тактов координата y первого элемента меняется от *y1* до *y2*, а координата *x* не меняется. Сложнее поведение второго элемента: первые 4 такта элемент движется налево, затем за 7 тактов он опускается от *y2* до *y1*, за последние 4 такта движется направо.

Листинг 4.5. Метод динамического обмена местами двух элементов

```
void Change(int n1, int n2, int n, int m)
{
      Element[n1].color = Color.Red;
      Element[n2].color = Color.Red;
      int x1 = Element[n1].x;
      int y1 = Element[n1].y;
      int x2 = Element[n2].x;
      int y2 = Element[n2].y;
      double x;
      for (int t = 1; t <= 15; t++)
      {
            x = (y2 - y1) * t / 15;
            Element[n1].y = y1+(int)(x);
            switch (t)
            {
                  case 1: case 2: case 3: case 4:
                        x = 40 * t / 4;
                        Element[n2].x = x1 - (int)(x);
                        break;
                  case 5: case 6: case 7: case 8:
                  case 9: case 10: case 11:
                        x = (y1 - y2) * (t - 4) / 7;
                        Element[n2].y = y2 + (int)(x);
                        break;
                  case 12: case 13: case 14: case 15:
                        x = 40 * (t - 11) / 4;
                        Element[n2].x = x1 - 40 + x;
                        break;
            }
            Drawing(n,m);
            Thread.Sleep(100);
      }
      Element[n1].color = Color.Black;
      Element[n2].color = Color.Black;
      Drawing(n,m);
      }
```

Для ускорения рисование всех элементов массива эллипсами происходит на канве *gBitmap* типа *Bitmap*.

Листинг 4.6. Рисование элементов

```
void Drawing(int n, int m)
{
   const int d = 15;
```

```
    int L = Element.Length;
    string s;
    SizeF size;
    gBitmap.Clear(Color.White);

    for (int i = 0; i <= L - 1; i++)
    {
       MyPen.Color = Element[i].color;
       gBitmap.DrawEllipse(MyPen, Element[i].x - d,
           Element[i].y - d, 2 * d, 2 * d);
       s = Convert.ToString(Element[i].inf);
       size = gBitmap.MeasureString(s, Font);
       gBitmap.DrawString(s, Font, Brushes.Black,
           Element[i].x - size.Width / 2,
           Element[i].y - size.Height / 2);
    }

    if (n != -1)
    {
       MyPen0.Color = Color.Black;
       gBitmap.DrawLine(MyPen0,120,Element[n].y, 140,
           Element[n].y);
       s = "I = " + Convert.ToString(n);
       size = gBitmap.MeasureString(s, Font);
       gBitmap.DrawString(s,Font, Brushes.Black, 150,
           Element[n].y - size.Height / 2);
    }

    if (m != -1)
    {
       MyPen0.Color = Color.Red;
       gBitmap.DrawLine(MyPen0,120,Element[m].y, 140,
           Element[m].y);
       s = "J = " + Convert.ToString(m);
       size = gBitmap.MeasureString(s, Font);
       gBitmap.DrawString(s,Font, Brushes.Black, 150,
           Element[m].y - size.Height / 2);
    }
    gScreen.DrawImage(bitmap, ClientRectangle);
}
```

Во время работы черная линия показывает на *i*-й элемент, а красная – на *j*-й элемент массива (рис. 4.2).

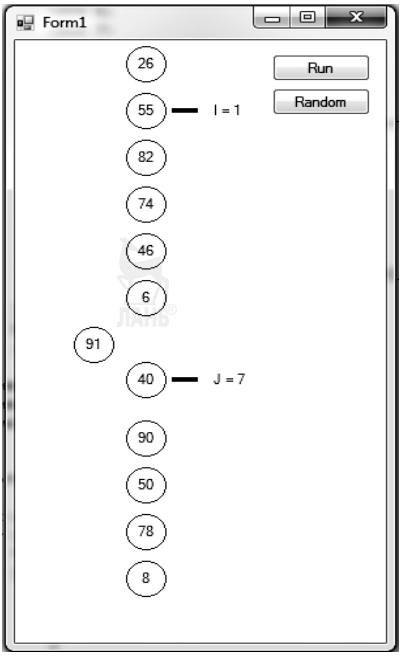

Рис. 4.2. Проект «Сортировка массива»

4.1.2. **Морфинг**

Морфинг – это превращение одного изображения в другое. В качестве примера рассмотрим превращение одной фундаментальной кривой (рис. 4.3) в другую.

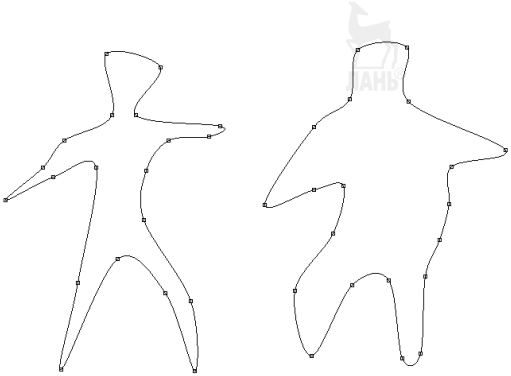

Рис. 4.3. Морфинг двух кривых

В классе *TLines* определим двумерный массив *lines[][]* для начального и конечного состояний кривой и временный массив *tmpLines[]* для промежуточного состояния кривой.

Листинг 4.7. Структура класса для морфинга

```
class TLines
{
    public static PointF[][] lines =
        new PointF[2][];
    public static PointF[] tmpLines;
}
```

Кроме этого, в классе определим два метода: метод начальной инициализации массивов *Init(int n)* и *FindPoint()*.

Листинг 4.8. Начальная инициализация массивов

```
public static void Init(int n)
{
   float r = 6;
   lines[0] = new PointF[n];
   lines[1] = new PointF[n];
   tmpLines = new PointF[n];
   double x, y;
   for (int i = 0; i < n; i++)
   {
        x = r * Math.Cos(Math.PI * 2 / n * i);
        y = r * Math.Sin(Math.PI * 2 / n * i);
        lines[0][i]=new PointF((float)(-3+x),y;
        lines[1][i] = new PointF((float)(3 + x), y);
    }
 }
```

Все три массива *lines[0]*, *lines[0]* и *tmpLines* имеют одинаковую длину.

Для перемещения точек на начальной и конечной кривой будет необходим метод поиска точки.

Листинг 4.9. Метод поиска точки

```
public static int FindPoint(ref int nLine, float x, float y)
{
   int result = -1; nLine = -1;
   double min = 1000;
   int L = lines[0].Length;
   for (int i = 0; i < 2; i++)
   for (int j = 0; j < L; j++)
   {
```

```
        float X = lines[i][j].X;
        float Y = lines[i][j].Y;
        double d = Math.Sqrt((X-x)*(X-x)+(Y-y)*(Y-y));
        if (d < min)
        {
            min = d; nLine = i; result = j;
        }
    }
    return result;
}
```

Метод поиска точки определяет не только номер точки, но и номер линии *nLine*, на которой эта точка находится. Рисование происходит в двух режимах – при незапущенном таймере и при запущенном.

Листинг 4.10. Рисование кривых

```
public void Draw()
{
    using (Graphics g = Graphics.FromImage(bitmap))
    {
        Color cl = Color.FromArgb(255, 255, 255);
        g.Clear(cl);

        int L = 0;
        if (TLines.lines[0] != null)
        L = TLines.lines[0].Length;
        Point[] p = new Point[L];
        float x, y;
        if (L > 0)
        {  // рисование в режиме редактирования
           if (!timer1.Enabled)
              for (int k=2; k<2; k++)
              {
                 for (int i = 0; i < L; i++)
                 {
                    x = TLines.lines[k][i].X;
                    y = TLines.lines[k][i].Y;
                    g.DrawRectangle(Pens.Black,
                        II(x) - 2, JJ(y) - 2, 4, 4);
                    p[i].X = II(x);
                    p[i].Y = JJ(y);
                 }
                 g.DrawClosedCurve(Pens.Black, p);
              }
           else
           {  // рисование при включенном таймере
               for (int i = 0; i < L; i++)
               {
```

```
                    x = TLines.tmpLines[i].X;
                    y = TLines.tmpLines[i].Y;
                    p[i].X = II(x);
                    p[i].Y = JJ(y);
                }
                g.DrawClosedCurve(Pens.Black, p);
            }
        }
    }
    g0.DrawImage(bitmap, ClientRectangle);
}
```

Редактирование точки, то есть ее перемещение, начинается при нажатии левой клавиши мыши поиском ближайшей точки.

Листинг 4.11. Начало перемещения точки

```
private void FormMain_MouseDown(object sender, MouseEventArgs
e)
{
   drawing = true;
   nPoint = TLines.FindPoint(ref nLine,
     (float)XX(e.X), (float)YY(e.Y));
}
```

После того как точка выбрана, происходит ее перемещение.

Листинг 4.12. Перемещение выбранной точки

```
private void FormMain_MouseMove(object sender, MouseEventArgs
e)
{
    if (drawing)
    {
        TLines.lines[nLine][nPoint].X =
           (float)XX(e.X);
        TLines.lines[nLine][nPoint].Y =
           (float)YY(e.Y);
        Draw();
    }
}
```

Морфинг реализован в единственном обработчике события таймера.

Листинг 4.13. Изменение координат временной кривой

```
private void timer1_Tick(object sender, EventArgs e)
{
    ++t;
```

```
    timer1.Enabled = t < n20;
    int L = TLines.lines[0].Length;
    float x1,x2,y1,y2;
    for (int i = 0; i < L; i++)
    {
        x1 = TLines.lines[0][i].X;
        y1 = TLines.lines[0][i].Y;
        x2 = TLines.lines[1][i].X;
        y2 = TLines.lines[1][i].Y;
        TLines.tmpLines[i].X = x1+t*(x2 - x1)/ n20;
        TLines.tmpLines[i].Y = y1+t*(y2 - y1)/ n20;
    }
    Draw();
}
```

Весь цикл заканчивается за $n20 = 20$ тактов таймера. Переменная t выступает в качестве счетчика тактов. Каждая точка начальной кривой перемещается в соответствующую ей точку конечной кривой по прямой линии. Ее положение в момент времени t определяется методом деления отрезка в заданном отношении.

4.1.3. ПАДЕНИЕ ГЛОБУСА

Пусть с высоты $H0$ падает резиновый глобус, который можно представлять как круг радиуса R. В момент времени $t1$ соприкосновения с поверхностью круг начинает превращаться в эллипс с полуосями A и B. Эти параметры достигают своих максимальных значений в момент времени $T0/2$ и возвращаются к значению R в момент времени $t2$. Затем начинается обратный процесс отскока мяча от поверхности. Временная диаграмма изменения параметров (высоты центра мяча H и полуосей A и B) представлена на рисунке 4.4.

Высота центра мяча $h(t)$ на интервале времени $[0,t1]$ меняется по параболе

$$h = -k * t^2 + H_0 \ . \tag{4.1}$$

Из условия $h|_{t=t1} = R$ можно определить t_1:

$$t_1 = \sqrt{\frac{H_0 - R}{k}}, \tag{4.2}$$

а это позволяет определить t_2:

$$t_2 = \frac{T_0}{2} + dt, \tag{4.3}$$

где $dt = \frac{T_0}{2} - t_1$.

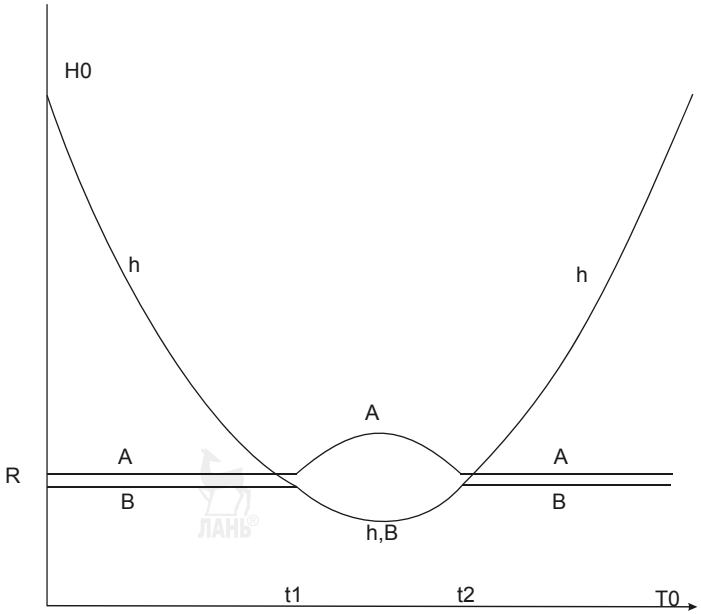

Рис. 4.4. Временная диаграмма изменения параметров мяча

Будем считать, что на интервале $[t_1, t_2]$ высота центра мяча $h(t)$ также меняется по параболе

$$h = R - dR + k_1 \left(t - T_0 / 2\right)^2, \quad t \in [0, t_1]. \tag{4.4}$$

Из условия $h\big|_{t=t1} = R$ определим k_1:

$$k_1 = \frac{dR}{\left(t_1 - T_0\right)^2}. \tag{4.5}$$

Для полуоси эллипса $A(t)$ уравнение на интервале $[t_1, t_2]$ симметрично уравнению (4.4) относительно $h = R$:

$$A = R + dR - k_1 \left(t - T_0\right)^2, \quad t \in [t_1, t_2]. \tag{4.6}$$

Для полуоси эллипса $B(t)$ уравнение совпадает с уравнением (4.4).

Для интервала $[t_2, T_0]$ уравнение изменения высоты симметрично относительно $t = T_0/2$ и имеет вид

$$h = -k \left(t - T_0\right)^2, \quad t \in [2, T_0]. \tag{4.7}$$

На интервалах $[0, t_1]$ и $[t_2, T_0/2]$ параметры A и B не меняются и равны R:

$$A = R,\ B = R, \quad t \in [0, t_1] \cup [t_2, T_0]. \tag{4.8}$$

Формулы (4.1)–(4.8) позволяют реализовать проект следующим образом. Пусть заданы константы

$$H_0=550;\ R=50;\ T_0=40;\ dR=R/5;\ k=5.0;$$

В момент запуска проекта вычислим параметры модели, описываемой уравнениями (4.1)–(4.8).

Листинг 4.14. Вычисление параметров модели

```
private void Form1_Load(object sender, EventArgs e)
{
    bitmap = new Bitmap(Width,Height);
    t1=(int)Math.Round(Math.Sqrt((H0-R)/k));
    dt=T0 / 2-t1;
    t2=T0 / 2+dt;
    k1 = dR / (t1 - T0 / 2) / (t1 - T0 / 2);
    g0 = CreateGraphics();
}
```

Листинг 4.15. Обработчик события для компонента Timer

```
private void timer1_Tick(object sender, EventArgs e)
{
    t=(t+1) % T0;
    MyDraw();
}
```

Метод рисования *MyDraw()*, приведенный в листинге 4.16, с помощью метода *GetHAB()* вычисляет параметры *h,A,B* и копирует *Bitmap1* в прямоугольник (100-*A*,*y*-*B*,100+*A*,y+*B*).

Листинг 4.16. Метод рисования

```
void MyDraw()
{
    using (Graphics g = Graphics.FromImage(bitmap))
    {
        Color cl = Color.FromArgb(255, 255, 255);
        g.Clear(cl);
        GetHAB(out h, out A, out B);
        int y = H0 - (int)Math.Round(h*(H0 - R)/H0);
        g.DrawImage(Bitmap1,130-A,y - B, 2*A, 2*B);
    }
    g0.DrawImage(bitmap, ClientRectangle);
    Thread.Sleep(20);
}
```

Метод *GetHAB* вычисляет параметры *h,A,B* по разным формулам на интервалах [*0,t1*], [*t1,t2*] и [*t2*, T_0].

Листинг 4.17. Вычисление параметров h,A,B

```
void GetHAB(out int h, out int A, out int B)
{
    A=R; B=R;
    if (t<t1)
        h=-(int)Math.Round(k*t*t)+H0;
    else
        if (t<t2+1)
    {
        h=R+(int)Math.Round(-dR+k1*(t-T0/2)*(t-
                T0/2));
        A=R+(int)Math.Round(dR-k1*(t-T0/2)*(t-
                T0/2));
        B=h;
    }
    else
        h=-(int)Math.Round(k*(t-T0)*(t-T0))+H0;
}
```

Работа проекта представлена на рисунке 4.5.

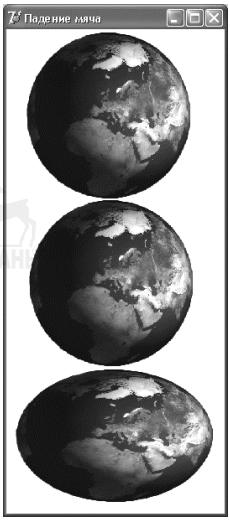

Рис. 4.5. Работа проекта «Падение мяча»

4.1.4. Велосипед

Рассмотрим еще один проект, использующий компонент *Timer1*, в котором двигаются ноги велосипедиста и спицы колес (рис. 4.6).

Рис. 4.6. Велосипед

В проекте используются следующие константы и переменные:

```
int x0=282;//центр педали
int y0=411;
int x1=218;//сидение
int y1=229;
int R0=65; int R1=23; // радиусы колеса
int R=30;  радиус колеса
int xR1=177; int yR1=403; // центр 1 и 2 колеса
int xR2 = 448; int yR2 = 400;
float Angle1=0;   // угол поворота первой педали
float Angle2 = (float)Math.PI; //угол поворота второй педали
float Angle3=0;  // угол поворота колес
float StepAngl=0.1F; // шаг угла
double L; // длина голени и бедра
```

В проекте используются две картинки – *'Velo1.bmp'* и *'Velo2.bmp'*. На первой изображен велосипед полностью, на второй (прозрачной по белому цвету) – только нижняя часть велосипеда. В момент запуска проекта картинки загружаются на *Bitmap1* и на *Bitmap2*.

Листинг 4.18. Загрузка картинок

```
public Form1()
{
    InitializeComponent();
    bitmap = new Bitmap(630,500);
    Bitmap1 = new Bitmap("Velo1.bmp");
    Bitmap2 = new Bitmap("Velo.bmp");
    Bitmap2.MakeTransparent(Color.White);
    L=(Math.Sqrt((x0-x1)*(x0-x1)+(y0-y1)*(y0-y1))+R)
```

```
                / 2.0;
    g0 = CreateGraphics();
}
```

Рисование на канве происходит в следующей последовательности:

- выводится полная картинка;
- рисуется дальняя нога;
- выводится нижняя часть велосипеда;
- рисуется ближняя нога;
- рисуются спицы.

Процесс рисования представлен в листинге 4.19.

Листинг 4.19. Рисование велосипеда

```
void DrawBicycle()
{
    using (Graphics g = Graphics.FromImage(bitmap))
    {
          Color cl = Color.FromArgb(255, 255, 255);
          g.Clear(cl);
          Rectangle r = new Rectangle(0,0,630,500);
          g.DrawImage(Bitmap1, r);
          DrawFoot(g,Angle1);
          g.DrawImage(Bitmap2, r);
          DrawFoot(g,Angle2);
          DrawWheel(g);
    }
    g0.DrawImage(bitmap, ClientRectangle);
}
```

Графическое описание движения ноги представлено на рисунке 4.7.

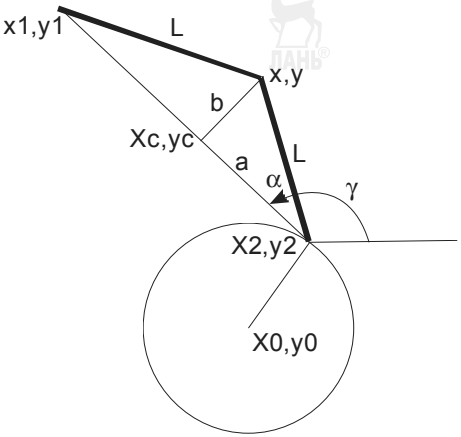

Рис. 4.7. Графическое описание алгоритма движения ноги

Педаль двигается по окружности вокруг точки x_0,y_0 и в какой-то момент времени находится в точке x_2,y_2. Нога описывается ломаной из двух звеньев длиной L, которая начинается в точке x_1,y_1, а заканчивается в точке x_2,y_2. Координаты точки x_2,y_2 вычисляются по формулам:

```
x2=x0+R*cos(Angle); y2=y0+R*sin(Angle).
```

Затем вычисляются координаты точки *xc*,*yc*:

```
xc=(x1+x2)/2; yc=(y1+y2)/2.
```

и длины катетов прямоугольного треугольника:

```
b=Sqrt(Sqr(x2-xc)+Sqr(y2-yc));
a=Sqrt(L*L-b*b).
```

Это позволяет найти угол:

```
alf=Arctan2(a,b).
```

а затем координаты точки x,y:

```
x=x2+L*cos(gam+alf); y=y2+L*sin(gam+alf).
```

Для рисования одной ноги предназначен метод *DrawFoot()*.

Листинг 4.20. Рисование одной ноги

```
void DrawFoot(Graphics g, double Angle)
{
    double x2=x0+R*Math.Cos(Angle);
    double y2 = y0 + R * Math.Sin(Angle);
    double xc = (x1 + x2) / 2;
    double yc = (y1 + y2) / 2;
    double b = Math.Sqrt((x2-xc)*(x2-xc)+
           (y2-yc)*(y2-yc));
    double a = Math.Sqrt(L * L - b * b);
    double alf = Math.Atan2(a, b);
    double gam = Math.Atan2(y1 - y2, x1 - x2);
    double x = x2 + L * Math.Cos(gam + alf);
    double y = y2 + L * Math.Sin(gam + alf);
    int u2 = Convert.ToInt32(x2);
    int v2 = Convert.ToInt32(y2);
    int u = Convert.ToInt32(x);
    int v = Convert.ToInt32(y);
    g.DrawLine(myPen, u2, v2, u, v);
    int u1 = Convert.ToInt32(x1);
    int v1 = Convert.ToInt32(y1);
    g.DrawLine(myPen, u, v, u1, v1);
}
```

Рисование спиц проще и реализуется методом *DrawWheel()*.

Листинг 4.21. Рисование спиц

```
void DrawWheel(Graphics g)
{
    const int n=25;
```

```
    for (int i=0; i<=n-1;i++)
    {
        double si = Math.Sin(a);
        double co = Math.Cos(a);
        double a=Angle3+2*Math.PI/n*i;
        int x1 = (int)Math.Round(xR1 + R0 * co);
        int y1 = (int)Math.Round(yR1 - R0 * si);
        int x2 = (int)Math.Round(xR1 + R1 * co);
        int y2 = (int)Math.Round(yR1 - R1 * si);
        g.DrawLine(Pens.Black,x2,y2,x1,y1);
        x1 = (int)Math.Round(xR2 + R0 * co);
        y1 = (int)Math.Round(yR2 - R0 * si);
        x2 = (int)Math.Round(xR2 + R1 * co);
        y2 = (int)Math.Round(yR2 - R1 * si);
        g.DrawLine(Pens.Black, x2, y2, x1, y1);
    }
}
```

4.2. Деформация изображений

В многочисленных графических редакторах используются различные фильтры изображения, изменяющие первоначальные картинки. Некоторые из этих фильтров меняют геометрию изображения. В библиотеке OpenGL, например, эта технология называется наложением текстуры на грани.

В этом параграфе описывается проект, решающий задачу деформации изображения из прямоугольной области (*0,0*), (*I2,J2*) в произвольный четырехугольник (*u0,v0*), (*u1,v1*), (*u2,v2*), (*u3,v3*). Вершины этого четырехугольника (рис. 4.8) можно перетаскивать мышью.

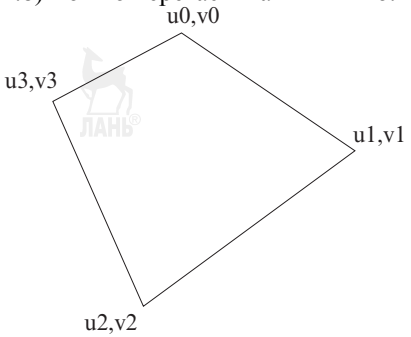

Рис. 4.8. Деформированный четырехугольник

На форму (рис. 4.9) вынесены два компонента для рисования: *Image1* и *Image2*, компонент диалога *OpenPictureDialog1* для выбора графического файла *BMP* и три пункта меню *Open1Click, N1Click* и *N2Click*, предназначенные для выбора файла BMP, прямой деформации и обратной деформации четырехугольника.

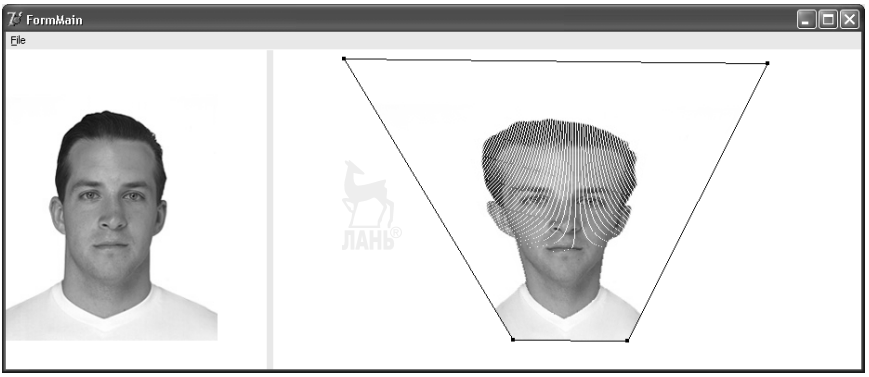

Рис. 4.9. Прямая деформация четырехугольников

После выбора графического файла с помощью компонента *openFileDialog1* изображение попадает в *bitmap1*.

Листинг 4.22. Выбор графического файла

```
private void openToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    if (openFileDialog1.ShowDialog() ==
        DialogResult.OK)
    {
        bitmap1 =
            new Bitmap(openFileDialog1.FileName);
        MyDraw();
    }
}
```

Закон преобразования прямоугольной области (*0,0*), (*I2,J2*) в четырехугольник (*u0,v0*), (*u1,v1*), (*u2,v2*), (*u3,v3*) будем искать в виде квазилинейного соотношения:

$$u=A1*x+B1*x*y+C1*y+D1;$$
$$v=A2*x+B2*x*y+C2*y+D2. \quad (4.9)$$

После подстановки всех четырех вершин и решения системы уравнений для коэффициентов этого линейного преобразования получаем

```
D1=u0
D2=v0
A1=(u1-u0)/I2
A2=(v1-v0)/I2
C1=(u3-u0)/J2
C2=(v3-v0)/J2
B1=(u2-u3-u1+u0)/I2/J2
B1=(v2-v3-v1+v0)/I2/J2
```

Затем в методе *Texture1()* все точки *bitmap1.GetPixel(x,y)* преобразуются в точки *bitmap2.SetPixel(i,j)*. Обратите внимание на то, что преобразуются не пикселы канвы компонента *Image2*, а пикселы битовой карты *bitmap2* в памяти, и только после полной подготовки *bitmap2* выводится на канву *bitmap*. Это значительно ускоряет работу программы.

Листинг 4.23. Преобразование пикселов канвы

```
public static void Texture1(out int I1_,out int J1_)
{
   int I2 = bitmap1.Width; int J2 = bitmap1.Height;
   int I1 = 0; int J1 = 0;
   I1_ = u.Min();
   J1_ = v.Min();
   int I2_ = u.Max();
   int J2_ = v.Max();
   bitmap2 = new Bitmap(I2_ - I1_, J2_ - J1_);
   using (Graphics g = Graphics.FromImage(bitmap2))
   {
      if ((I2 - I1 != 0) & (J2 - J1 != 0))
      {
         double D1 = u[0]; double D2 = v[0];
         double A1 = (u[1] - u[0]) / (I2 - I1);
         double A2 = (v[1] - v[0]) / (I2 - I1);
         double C1 = (u[3] - u[0]) / (J2 - J1);
         double C2 = (v[3] - v[0]) / (J2 - J1);
         double B1 = (u[2] - u[3] - u[1] + u[0]) /
            (I2 - I1) / (J2 - J1);
         double B2 = (v[2] - v[3] - v[1] + v[0]) /
            (I2 - I1) / (J2 - J1);
         for (int y=1; y <= bitmap1.Height - 1; y++)
         {
             for (int x=1; x<=bitmap1.Width-1; x++)
             {
                int i = (int)Math.Round(A1 * x +
                   B1 * x * y + C1 * y + D1);
                int j = (int)Math.Round(A2 * x +
                   B2 * x * y + C2 * y + D2);
                bitmap2.SetPixel(i - I1_, j - J1_,
                   bitmap1.GetPixel(x, y));
             }
          }
      }
      g.Dispose();
   }
}
```

У этого простого метода есть два недостатка:

– для больших *Bitmap* он будет работать долго;
– для маленьких *Bitmap*, как это видно на рисунке 4.9, он не будет плотно заполнять все точки *Image2*.

Немного лучше будет работать проект, в котором используется обратное преобразование от четырехугольника (*u0,v0*), (*u1,v1*), (*u2,v2*), (*u3,v3*) к прямоугольнику (*0,0*), (*I2,J2*).

Из уравнений (4.9) исключим *x*. Для переменной *y* получаем квадратное уравнение

$$A*Y^2+B*Y+C=0, \qquad (4.10)$$

где

*A=C2*B1-C1*B2;*

B=B2(u-D1)-B1*(v-D2)-A2*C1+C2*A1;*

C=A2(u-D1)-A1*(v-D2).*

Из уравнения (4.9) для переменных x и y получаем следующие соотношения:

*Y=(-B+Sqrt(D))/(2*A)*;

*X=(u-D1-C1*y)/(A1+B1*y).*

В этом случае метод *Texture2()* заполнения четырехугольника выглядит следующим образом.

Листинг 4.24. Обратное преобразование пикселов канвы

```
void Texture2(out int MinU, out int MinV)
{
   MinU = 0; MinV = 0;
   int I2=bitmap1.Width; int J2=bitmap1.Height;
   if ((I2!=0) & (J2!=0))
   {
      MinU=u.Min(); MinU=Math.Max(0,MinU);
      int MaxU=u.Max(); //MaxU=Math.Min(H1-1,MaxU);
      MinV=v.Min(); MinV=Math.Max(0,MinV);
      int MaxV=v.Max(); //MaxV=Math.Min(H2-1,MaxV);

      double D1=u[0]; double D2=v[0];
      double A1 = (1.0 * u[1] - u[0]) / I2;
      double A2 = (1.0 * v[1] - v[0]) / I2;
      double C1 = (1.0 * u[3] - u[0]) / J2;
      double C2 = (1.0 * v[3] - v[0]) / J2;
      double B1 = (1.0*u[2] - u[3] - u[1] +
          u[0])/I2/J2;
      double B2 = (1.0 * v[2] - v[3] - v[1] +
          v[0])/I2/J2;
      double A = 1.0 * C2 * B1 - C1 * B2;
```

```
      bitmap2 = new Bitmap(MaxU-MinU, MaxV - MinV);
      for (int vt=MinV; vt<=MaxV; vt++)
      {
        for (int ut=MinU; ut<=MaxU; ut++)
        if (PointInPolygon(ut,vt))
        {
           double B = (B2 * (ut - D1) - A2 * C1 +
              C2 * A1 + (D2 - vt) * B1);
           double C = A2 * (ut - D1) + A1*(D2 - vt);
           double D = 1.0 * B * B - 4 * A * C;
           if (D>=0)
           {
              double y=(-B+Math.Sqrt(D))/(2*A);
              double x=(ut-D1-C1*y)/(A1+B1*y);
              int i=(int)Math.Round(x);
              int j=(int)Math.Round(y);
              if ((0<=j) & (j<bitmap1.Height)
                 & (0<=i) & (i<bitmap1.Width))
              {
                  bitmap2.SetPixel(ut-MinU,vt- MinV,
                     bitmap1.GetPixel(i, j));
              }
           }
        }
      }
    }
}
```

В этом методе появилось два ограничения: во-первых, метод *PointInPolygon()* проверяет, что точка *ut,vt* принадлежит четырехугольнику, во-вторых, из двух решений квадратного уравнения берется только одно.

Обратное преобразование, как и следовало ожидать, заполняет четырехугольник плотно (рис. 4.10).

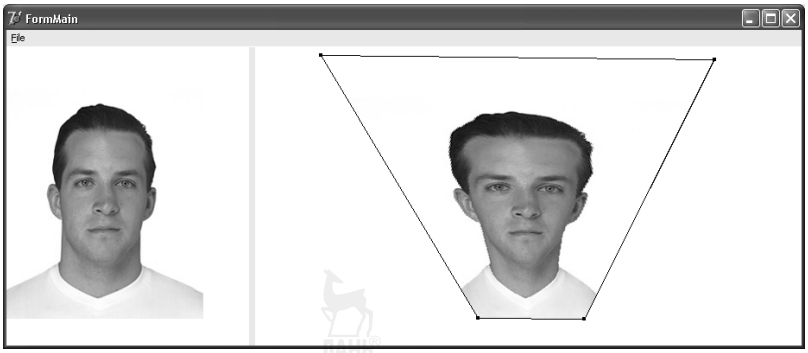

Рис. 4.10. Обратная деформация четырехугольников

У обратного механизма деформации есть свои недостатки: если плотность точек во втором четырыхугольнике меньше, чем в прямоугольнике, то у нескольких точек может быть одна и та же точка-прообраз в прямоугольнике. Особенно хорошо это видно для маленьких изображений. В модуле OpenGL для решения этой проблемы предлагается шесть вариантов. Первый, самый простой, использован у нас: для цвета точки в прямоугольнике используется цвет ближайшей точки в прямоугольнике. Второй метод использует четыре ближайших точки и усредняет их цвета. Остальные методы используют, если это необходимо, более мелкие копии прямоугольников.

Обсудим теперь реализацию механизма перетаскивания вершин четырехугольника. На компоненте *Image2* рисуются четыре точки с координатами *int[] u,v = new int[4]*, в которых находятся вершины деформированного четырехугольника. Эти точки можно перетаскивать мышью. Начало перетаскивания обрабатывается методом *Form1_MouseDown*.

Листинг 4.25. Начало перетаскивания вершины

```
private void Form1_MouseDown(object sender,MouseEventArgs e)
{
    IndexPoint=-1;
    drawing = false;
    while (!drawing & (IndexPoint < 3))
    {
       ++IndexPoint;
       drawing = (Math.Abs(u[IndexPoint]-e.X) <= 19)
          &(Math.Abs(v[IndexPoint] - e.Y+10) <= 19);
    }
}
```

В этом методе сначала определяем номер точки, в окрестности которой произошел щелчок клавиши мыши. Затем, если захвачена точка с номером *IndexPoint*, начинаем перетаскивание, запоминая начальную точку.

Перемещение мыши обрабатывается методом *Form1_MouseMove*, в которой от точки *IndexPoint* рисуются два отрезка.

Листинг 4.26. Метод обработки перемещения мыши

```
private void Form1_MouseMove(object sender,MouseEventArgs e)
{
   if (drawing)
   {
      u[IndexPoint] = e.X;
      v[IndexPoint] = e.Y+0;
      MyDraw();
   }
}
```

Завершается перемещение вершины вызовом метода *Form1_MouseUp*.

Листинг 4.27. Метод обработки отпускания клавиши мыши

```
private void Form1_MouseUp(object sender, MouseEventArgs e)
{
    drawing = false;
}
```

Метод рисования использует три *bitmap*: основной *bitmap*, на который сначала выводится четырехугольник, затем на *g*0 копируется *bitmap1* с прямоугольной картинкой и *bitmap2* с деформированной картинкой.

Листинг 4.28. Метод рисования

```
void MyDraw()
{
   using (Graphics g = Graphics.FromImage(bitmap))
   {
       Color c1 = Color.FromArgb(255, 255, 255);
       g.Clear(c1);

       for (int i=0; i<=3; i++)
       g.DrawLine(Pens.Black,u[i],v[i],
                  u[(i+1)%4],v[(i+1)%4]);
   }
   g0.DrawImage(bitmap, ClientRectangle);
   if (bitmap2!=null)
       g0.DrawImage(bitmap2, I1, J1);
   if (bitmap1 != null)
       g0.DrawImage(bitmap1, 0, 0, 250, 250);
 }
```

Заметим, что для реальных проектов механизм, реализованный в таком виде, применять сложно – все-таки он работает достаточно медленно.

Можно существенно упростить алгоритм, если разбить прямоугольную область (*0,0*), (*I2,J2*) на два треугольника. Тогда для каждого треугольника закон преобразования будем искать в виде линейного соотношения:

*u=A1*x+B1*y+C1*;

*v=A2*x+B2*y+C2*.

Ниже приводится листинг соответствующего метода:

```
public void Texture3()
{
    int Width = bitmap1.Width;
    int Height = bitmap1.Height;
```

```
int I1 = 0; int J1 = 0;
leftImg2 = u.Min();
topImg2 = v.Min();
int I2_ = u.Max();
int J2_ = v.Max();
bitmap2 = new Bitmap(I2_ - leftImg2, J2_ - topImg2);

using (Graphics g = Graphics.FromImage(bitmap2))
{
    if ((Width != 0) & (Height != 0))
    {
        double C1 = u[0];
        double C2 = v[0];
        double A1 = (1.0 * u[1] - u[0]) / Width;
        double A2 = (1.0 * v[1] - v[0]) / Width;
        double B1 = (1.0 * u[3] - u[0]) / Height;
        double B2 = (1.0 * v[3] - v[0]) / Height;

        for (int y = 1; y <= bitmap1.Height - 1; y++)
        {
            for (int x = 1; x <= Width - Width * y /
                 Height - 1; x++)
            {
                int i = (int)Math.Round(1.0 * A1 * x +
                      B1 * y + C1);
                int j = (int)Math.Round(1.0 * A2 * x +
                      B2 * y + C2);
                Color col = bitmap1.GetPixel(x, y);
                bitmap2.SetPixel(i - leftImg2,
                 j - topImg2, bitmap1.GetPixel(x, y));
            }
        }

        C1 = u[3] + u[1] - u[2];
        C2 = v[3] + v[1] - v[2];
        A1 = (1.0 * u[2] - u[3]) / Width;
        A2 = (1.0 * v[2] - v[3]) / Width;
        B1 = (1.0 * u[2] - u[1]) / Height;
        B2 = (1.0 * v[2] - v[1]) / Height;

        for (int y = 1; y <= bitmap1.Height - 1; y++)
        {
            for (int x = Width - Width * y / Height - 1;
                 x < Width; x++)
            {
                int i = (int)Math.Round(1.0 * A1 * x +
                      B1 * y + C1);
                int j = (int)Math.Round(1.0 * A2 * x +
```

```
                        B2 * y + C2);
                    Color col = bitmap1.GetPixel(x, y);
                    bitmap2.SetPixel(i - leftImg2,
                     j - topImg2, bitmap1.GetPixel(x, y));
                }
            }
        }
        g.Dispose();
    }
}
```

4.3. Растровый редактор

Опишем создание графического растрового редактора типа редактора *PaintBrush*. Внешний вид редактора представлен на рисунке 4.11.

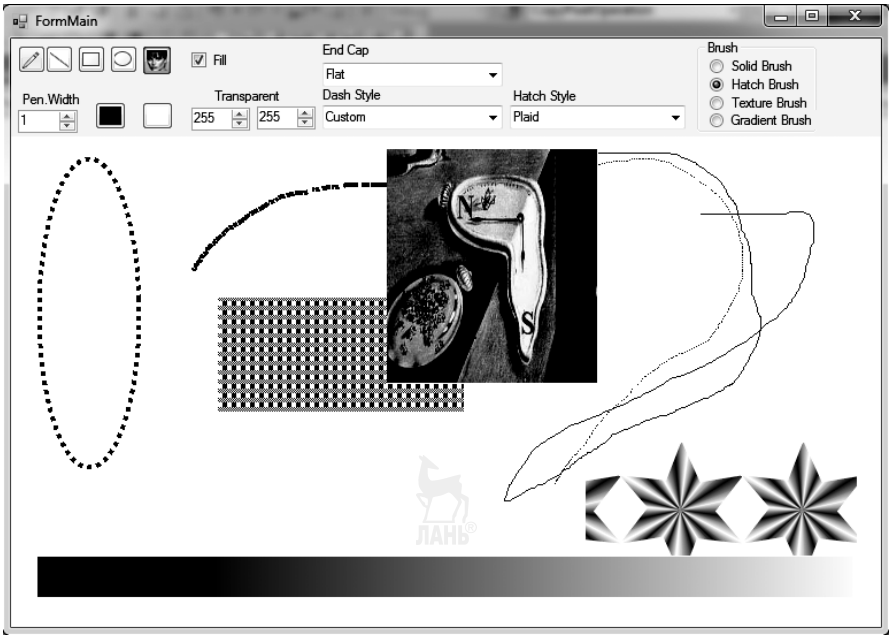

Рис. 4.11. Растровый редактор

Проект состоит из одной формы *FormMain*. На инструментальной панели выставлена группа кнопок класса *RadioButton*. Все они обладают различными значениями свойства *Tag* и одним обработчиком событий, меняющим при щелчке левой кнопкой мыши значение переменной *fl_tools*.

Листинг 4.29. Общий обработчик для группы кнопок

```
private void radioButton1_CheckedChanged(object sender,
    EventArgs e)
{
    fl_tools = Convert.ToByte((sender as RadioButton).Tag);
}
```

Переменная *fl_tools* используется при рисовании различных примитивов, и её возможные значения и назначение кнопок приведены в таблице 4.1.

Таблица 4.1

Инструментальные кнопки

Name	**Tag = tl_*****	**Назначение**
ButtonPen	*tl_Pen=0*	Рисовать линию
ButtonLine	*tl_Line=1*	Рисовать отрезок прямой
ButtonRect	*tl_Rectangle=2*	Рисовать прямоугольник
ButtonEllipse	*tl_Ellipse=3*	Рисовать эллипс
ButtonPicture	*tl_Picture=4*	Вставлять изображение

В проекте используется несколько переменных, назначение которых описано в таблице 4.2.

Таблица 4.2

Назначение переменных

Переменная	**Класс**	**Назначение**
drawing	*bool*	Флаг перемещения мыши
myPen	*Pen*	Основное перо
myPen0	*Pen*	Пунктирное перо
myBrush	*Brush*	Кисть. Может быть одной из 4 кистей
mySolidBrush	*SolidBrush*	Сплошная кисть
myBrushTexture	*TextureBrush*	Текстурная кисть
myHatchBrush	*HatchBrush*	Кисть стилей
myBrushGrad	*LinearGradient Brush*	Градиентная кисть
myHatchStyle	*HatchStyle*	Стиль кисти

Продолжение табл. 4.2

Переменная	Класс	Назначение
myLinearGradientMode	*LinearGradientMode*	Режим градиентной кисти
myBackColor	*Color*	Цвет фона
myTransparent	*byte*	Прозрачность пера
myTransparentBrush	*byte*	Прозрачность кисти
flFill	*bool*	Флаг рисования Draw или Fill
bitmap	*Bitmap*	Основной
bitmapTmp	*Bitmap*	Временный
gBitmap	*Graphics*	Основное полотно рисования
gScreen	*Graphics*	Полотно рисования формы

Введение глобальных для формы объектов кистей и перьев позволяет только один раз создавать эти объекты, что ускоряет работу программы. В программе также используется технология двойной буфферезации, то есть рисование происходит в памяти на bitmap и затем готовый bitmap копируется на форму. Это также ускоряет работу программы.

В момент запуска проекта выполняется создание объектов пера и кистей и заполняются элементы управления *comboBox*.

Листинг 4.30. Поля класса

```
class G
{
    public const byte tl_Pen = 0;
    const byte tl_Line = 1;
    const byte tl_Rectangle = 2;
    const byte tl_Ellipse = 3;
    public const byte tl_Picture = 4;

    public static Pen myPen;
    public static Pen myPen0;

    public static Brush myBrush;
    public static SolidBrush mySolidBrush;
    public static TextureBrush myBrushTexture;
    public static HatchBrush myHatchBrush;
    public static LinearGradientBrush myBrushGrad;
```

```
    public static HatchStyle myHatchStyle;
    public static LinearGradientMode myLinearGradientMode;

    public static Color myBackColor;
    public static byte myTransparent = 0xFF;
    public static byte myTransparentBrush = 0xFF;
    public static bool flFill = false;

    public static Graphics gBitmap;
    public static Bitmap bitmap;
    public static Bitmap bitmapTmp;

    // методы класса
}
```

При заполнении элементов управления *comboBox* значениями перечислимого класса используется метод *Enum.GetValues()*, который создает массивы класса *object[]*. Затем эти массивы передаются в свойство *Items*.

На инструментальную панель выставлены следующие элементы:

- три компонента класса *NumericUpDown*, позволяющие менять толщину пера, прозрачность линий и кисти;
- две кнопки класса *Button* для изменения цвета пера и кисти;
- четыре элемента класса *RadioButton*, предназначенные для выбора одной из четырех кистей (сплошной, текстурной, шаблонной и градиентной);
- четыре элемента класса *ComboBox*, предназначенные для выбора стиля линии *comboBoxDashStyle*;
- для выбора стиля конца линии *comboBoxEndCap*;
- для выбора шаблона кисти *comboBoxHatchStyle*;
- для выбора направления градиентной заливки *comboBoxLinearGradientMode*.

Далее представлены листинги обработчиков событий этих элементов.

Листинг 4.31. Изменение толщины пера

```
private void numericUpDown1_ValueChanged(object
    sender, EventArgs e)
{
    G.myPen.Width = (float)numericUpDown1.Value;
}
```

Листинг 4.32. Изменение цвета пера

```
private void buttonColorDialog_Click(object sender,
    EventArgs e)
{
    if (colorDialog1.ShowDialog() ==
        DialogResult.OK)
    {
        Color col = colorDialog1.Color;
        buttonColorDialog.BackColor = col;
        G.myPen.Color = col;
        byte R = col.R, G = col.G, B = col.B;
        G.myPen.Color =
            Color.FromArgb(myTransparent,R,G,B);
        G.myHatchBrush = new HatchBrush(myHatchStyle,
            myPen.Color, mySolidBrush.Color);
        G.myBrushGrad =
            new LinearGradientBrush(ClientRectangle,
            myPen.Color, mySolidBrush.Color,
            myLinearGradientMode);
        colorDialog1.Dispose();
    }
}
```

Листинг 4.33. Изменение цвета заливки

```
private void buttonBackColor_Click(object sender,
    EventArgs e)
{
    if (colorDialog1.ShowDialog() ==
        DialogResult.OK)
    {
        Color col = colorDialog1.Color;
        buttonBackColor.BackColor = col;
        byte R = col.R, G = col.G, B = col.B;
        G.mySolidBrush.Color =
           Color.FromArgb(myTransparentBrush,R,G,B);
        G.myHatchBrush = new HatchBrush(myHatchStyle,
            myPen.Color, mySolidBrush.Color);
        G.myBrushGrad =
            new LinearGradientBrush(ClientRectangle,
                myPen.Color, mySolidBrush.Color,
                myLinearGradientMode);
        colorDialog1.Dispose();
    }
}
```

Листинг 4.34. Изменение прозрачности пера

```
private void numericUpDown2_ValueChanged(object sender,
    EventArgs e)
{
    myTransparent = (byte)numericUpDown2.Value;
    Color col = FormMain.myPen.Color;
    byte R = col.R, G = col.G, B = col.B;
    G.myPen.Color=Color.FromArgb(myTransparent,R,G,B);
}
```

Листинг 4.35. Изменение прозрачности кисти

```
private void numericUpDown3_ValueChanged(object sender,
    EventArgs e)
{
    myTransparentBrush = (byte)numericUpDown3.Value;
    Color col = mySolidBrush.Color;
    byte R = col.R, G = col.G, B = col.B;
    G.mySolidBrush.Color =
        Color.FromArgb(myTransparentBrush, R, G, B);
}
```

Листинг 4.36. Изменение стиля пера

```
private void comboBox1_SelectedIndexChanged(object sender,
    EventArgs e)
{
    G.myPen.DashStyle =
       (DashStyle)comboBoxDashStyle.SelectedIndex;
}
```

Листинг 4.37. Изменение пера

```
private void radioButton6_CheckedChanged(object sender,
    EventArgs e)
{
    byte flBrush =
        Convert.ToByte((sender as RadioButton).Tag);

    switch (flBrush)
    {
        case 0:
            G.myBrush = G.mySolidBrush;
            break;
        case 1:
            G.myBrush = G.myHatchBrush;
            break;
        case 2:
```

```
            G.myBrush = G.myBrushTexture;
            break;
        case 3:
            G.myBrush = G.myBrushGrad;
            break;
    }
}
```

Листинг 4.38. Изменение шаблона кисти

```
private void comboBox2_SelectedIndexChanged(object sender,
    EventArgs e)
{
    G.myHatchStyle =
        (HatchStyle)comboBoxHatchStyle.SelectedItem;
    G.myHatchBrush = new HatchBrush(G.myHatchStyle,
        G.myPen.Color, mySolidBrush.Color);
}
```

Листинг 4.39. Изменение режима градиентной кисти

```
private void comboBox3_SelectedIndexChanged(object sender,
    EventArgs e)
{
    G.myLinearGradientMode =
       (LinearGradientMode)comboBoxGradMode.
       SelectedItem;
    G.myBrushGrad = new LinearGradientBrush(
        ClientRectangle,
        G.myPen.Color, G.mySolidBrush.Color,
        G.myLinearGradientMode);
}
```

Листинг 4.40. Изменение стиля конца линии

```
private void comboBoxEndCap_SelectedIndexChanged(
    object sender, EventArgs e)
{
    G.myPen.EndCap = (LineCap)comboBoxEndCap.SelectedItem;
}
```

Перейдем, наконец, к описанию рисования, которое происходит на полотне элемента *bitmap*, и в этом процессе участвуют, как обычно, три обработчика событий: *onMouseDown*, *onMouseMove* и *onMouseUp*. Напомним общую схему рисования. В обработчике события *onMouseDown* (листинг 4.32), который вызывается при нажатии левой кнопки мыши,

переменной drawing присваивается значение *true*. Затем выполняются действия в зависимости от значения переменной *fl_tools*.

Листинг 4.41. Начало перемещения мыши

```
private void FormMain_MouseDown(object sender,
     MouseEventArgs e)
{
    switch (fl_tools)
    {
        case G.tl_Pen: // Pen
            drawing = true;
            e0 = e;
            break;
        case 1: // Line
        case 2: // Rectangle
        case 3: // Ellipse
        case 4: // Image
            drawing = true;
            G.bitmapTmp = (Bitmap)G.bitmap.Clone();
            // запомнить bitmap
            e0 = e;
            break;
    }
}
```

Если *drawing = true*, то работает обработчик события перемещения мыши *onMouseMove*.

Листинг 4.42. Перемещение мыши

```
private void FormMain_MouseMove(object sender,
    MouseEventArgs e)
{
    if (drawing)
    {
        G.Draw(fl_tools, ref e0, e);
        e1 = e;
    }
    gScreen.DrawImage(G.bitmap, ClientRectangle);
}
```

При отпускании кнопки мыши вызывается обработчик события *onMouseUp*, в нем переменной *drawing* присваивается значение *false*, а также выполняются действия в зависимости от значения переменной *fl_tools*.

Листинг 4.43. Завершение перемещения мыши

```
private void FormMain_MouseUp(object sender, MouseEventArgs
e)
{
    drawing = false;
    switch (fl_tools) //
    {
        case G.tl_Picture:
            G.gBitmap.DrawRectangle(Pens.White, e0.X, e0.Y,
                e.X - e0.X, e.Y - e0.Y);
            if (openFileDialog1.ShowDialog() ==
                DialogResult.OK)
            {
                Bitmap theImage =
                     new Bitmap(openFileDialog1.FileName);
                G.gBitmap.DrawImage(theImage,
                     new Rectangle(e0.X,e0.Y,e.X-e0.X,
                          e.Y- e0.Y));
                theImage.Dispose();
            }
            openFileDialog1.Dispose();
            break;
    }
}
}
```

Теперь можно приступить к описанию рисования всех примитивов. Начнем с самого простого – рисования произвольной линии. Этот примитив рисуется при *fl_tools = tl_Pen*. При нажатии кнопки мыши запоминаем положение начальной точки в переменной *e*0 типа *MouseEventArgs*, у которой есть два поля: *e0.X* и *e0.Y*.

Листинг 4.44. Начало рисования линии

```
                case tl_Pen: // Pen
                    drawing = true;
                    e0 = e;
                    break;
```

Во время перемещения мыши рисуем линию от *e*0 до текущей точки *e* и запоминаем новое значение *e*0.

Листинг 4.45. Рисование линии

```
    case tl_Pen: // Pen
       gBitmap.DrawLine(myPen, P(e0), P(e));
       e0 = e;
       break;
```

При отпускании кнопки мыши никаких дополнительных действий кроме *drawing = false* делать не надо.

Рисование *отрезка* прямой линии немного сложнее: необходимо запомнить основной bitmap в *bitmapTmp* и начальную точку *e*0.

Листинг 4.46. Начало рисования отрезка прямой линии

```
case 1: // Line
case 2: // Rectangle
case 3: // Ellipse
case 4: // Image
    drawing = true;
    // запомнить все
    bitmapTmp = (Bitmap)bitmap.Clone();
    e0 = e;
    break;
```

Во время перемещения мыши мы должны вспомнить начальное изображение из *bitmapTmp* и нарисовать линию от *e*0 до *e*.

Листинг 4.47. Рисование отрезка прямой линии

```
if (fl_tools != tl_Pen)
{
    // вспомнить
    bitmap = (Bitmap)bitmapTmp.Clone();
    gBitmap = Graphics.FromImage(bitmap);
}
switch (fl_tools) // дорисовать
{
    case tl_Line: // Line
        gBitmap.DrawLine(myPen,P(e0),P(e));
        break;
}
gScreen.DrawImage(bitmap, ClientRectangle);
```

Никаких дополнительных действий при завершении рисования не требуется. Начало рисования прямоугольника и эллипса такое же, как у линии. Немного отличается само рисование – в зависимости от значения флага *flFill* необходимо использовать или *Draw*-, или *Fill*-методы.

Листинг 4.48. Рисование прямоугольника и эллипса

```
case tl_Rectangle: // Rectangle
    if (!flFill)
        gBitmap.DrawRectangle(myPen,
            e0.X, e0.Y, e.X - e0.X, e.Y - e0.Y);
    else
```

```
        gBitmap.FillRectangle(myBrush,
            e0.X, e0.Y, e.X - e0.X, e.Y - e0.Y);
    break;
case tl_Ellipse: // Ellipse
    if (!flFill)
        gBitmap.DrawEllipse(myPen,
            e0.X, e0.Y, e.X - e0.X, e.Y - e0.Y);
    else
        gBitmap.FillEllipse(myBrush,
            e0.X, e0.Y, e.X - e0.X, e.Y - e0.Y);
    break;
```

Для вставки изображения начало и рисование прямоугольника совпадает с рисованием прямоугольника. Заканчивается вставка изображения вызовом диалога *openFileDialog1*, загрузкой изображения в *theImage* и копированием *theImage* в выбранный прямоугольник.

Листинг 4.49. Вставка изображения

```
case tl_Picture:
    gBitmap.DrawRectangle(Pens.White, e0.X, e0.Y,
        e1.X - e0.X, e1.Y - e0.Y);
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        Bitmap theImage =
           new Bitmap(openFileDialog1.FileName);
        gBitmap.DrawImage(theImage,
           new Rectangle(e0.X,e0.Y,
               e1.X-e0.X+0,e1.Y- e0.Y+0));
        theImage.Dispose();
    }
    openFileDialog1.Dispose();
    break;
```

4.4. РЕДАКТИРОВАНИЕ ГРАФА

Проект состоит из главной формы *FormMain*, двух вспомогательных форм *FormProperty*, *FormColor* и модуля *Graph*, в котором собраны структуры данных и все реализованные алгоритмы для графов.

Рисование графа реализовано на канве формы *FormMain* и позволяет изображать граф в двух видах: в виде прямоугольников, соединенных ломаными линиями (рис. 4.12); в виде окружностей, соединенных прямыми линиями (рис. 4.13). Переключение вида графа происходит на форме *FormTools*. На форме *FormTools* находятся три инструментальные кнопки, позволяющие добавлять и перемещать узлы, создавать дуги. Форма *FormMatr* показывает матрицу смежности, а с помощью формы *FormPropertyNode* можно задавать свойства любого узла.

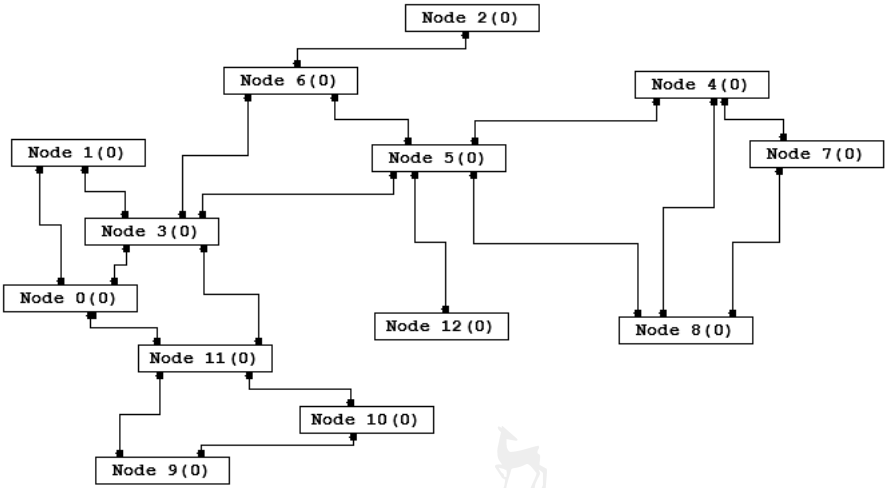

Рис. 4.12. Проект для алгоритмов на графах, узлы — прямоугольники

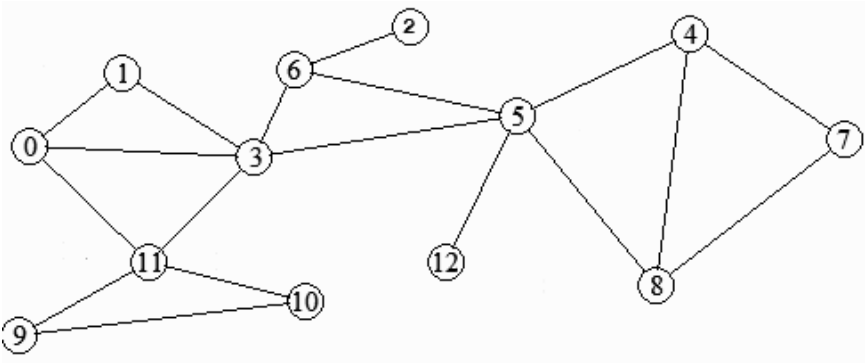

Рис. 4.13. Проект для алгоритмов на графах, узлы — окружности

4.4.1. Структура данных

Выбор структуры данных оказывает решающее значение на эффективность алгоритмов. Общая структура классов проекта представлена на рисунке 4.14.

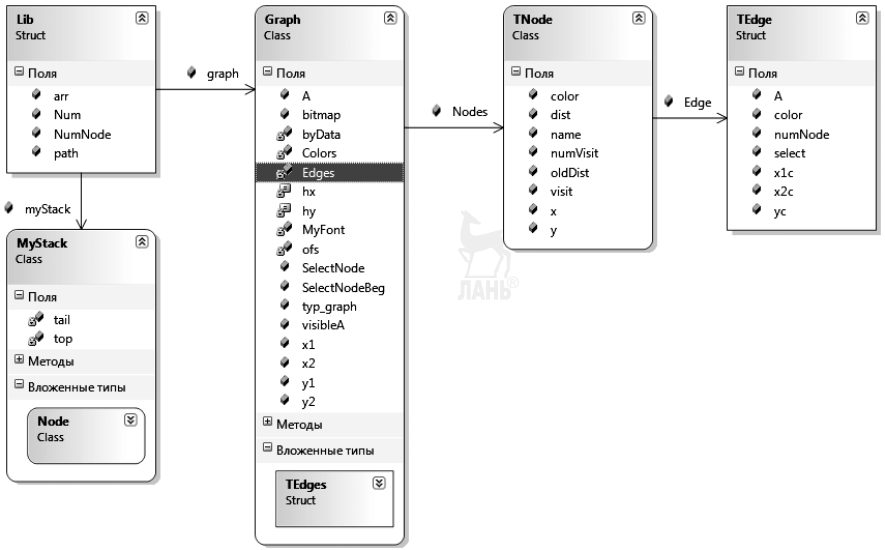

Рис. 4.14. Общая структура классов проекта

Класс *Graph* является основным.

Листинг 4.50. Класс Graph

```
public class Graph
{
   const int hx = 50, hy = 10;
   public Bitmap bitmap;
   public TNode[] Nodes = new TNode[0];// узлы
   public static TNode SelectNode;     // выделенный узел
   public static TNode SelectNodeBeg;
   public byte typ_graph = 1;
   public int[,] A;                 // матрица инцидентности
   // установить граф неориентированным
   public void SetSim()
   public Graph(int VW, int VH)
   public int FindNumEdge(int i, int j)
   public void SetA()
```

```
    public void AddNode(int x,int y)     // добавить узел
    public void AddEdge()                // добавить ребро
    public TNode FindNode(int x, int y) // найти узел
    public void DeSelectEdge()           // снять выделение
    public void Draw(bool fl)            // нарисовать
    public void Save(string FileName)    // записать
    public void Open(string FileName)    // прочитать
    // найти ребро
    public int FindLine(int x,int y, out int NumLine)
    // удалить ребро
    public void DelEdge(int NumNode, int NumEdge)
}
```

Основным полем класса графа при реализации алгоритмов является динамический массив узлов *Node[]* класса *TNode*. По массиву дуг, имеющемуся у каждого узла, можно построить матрицу инциденций *int[,] A*.

Каждый элемент массива *Node* вершин графа определяется структурой, представленной в листинге 4.51.

Листинг 4.51. Структура узлов графа

```
public class TNode
{
    public string name;  // имя узла
    public TEdge[] Edge; // массив дуг
    public bool visit;   // признак "узел посещен"
    public int x0, y0;   // координаты центра узла
    public int numVisit; // № посещения
    public Color color;  // цвет узла
    public int dist;     // минимальное расстояние
}
```

В этой структуре поле *name* предназначено для хранения имени узла, поле *Edge* описывает список ребер, выходящих из вершины, поля *x*0 и *y*0 задают координаты центра вершины. Поле *visit* будет играть важную роль при реализации многих алгоритмов, в нем мы будем отмечать посещение вершины. Поле *color* также играет вспомогательную роль при решении задач раскраски графов, а поле *dist* будет использоваться при решении задач определения кратчайших путей на графе.

В листинге 4.52 представлена структура, предназначенная для описания ребер. Самым важным в этой структуре является поле *numNode*, содержащее номер вершины, на которую показывает ребро. Не менее важную информацию содержит поле *A*, содержащее вес ребра. Для веса ребра мы ограничились целым типом, хотя в реальных задачах это поле может быть вещественным.

Листинг 4.52. Структура ребер

```
public struct TEdge
{
    public int A;             //
    public int numNode;       //
    public int x1c, x2c, yc; //
    public Color color;
    public bool select;
}
```

Поле *color* (цвет дуги) играет вспомогательную роль: при реализации некоторых алгоритмов мы будем менять цвет ребра, если пройдем по нему. Поля *x1c*, *y1c* и *yc* содержат геометрические параметры ребра (рис. 4.15). Поле указывает на расстояние горизонтальной части ребра от верхнего края формы.

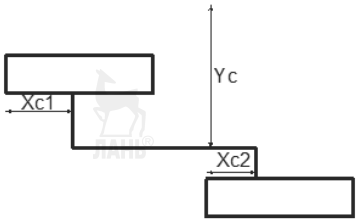

Рис. 4.15. Геометрические параметры ребра

4.4.2. ИЗОБРАЖЕНИЕ ГРАФОВ

Метод рисования графа состоит из следующих шагов. На канву *bitmap* мы сначала выводим дуги, а затем все узлы.

Листинг 4.53. Рисование графа

```
public void Draw(bool fl) // нарисовать
{
   using (Graphics g = Graphics.FromImage(bitmap))
   {
      Color cl = Color.FromArgb(255, 255, 255);
      g.Clear(cl);
      Pen MyPen = Pens.Black;
      SolidBrush MyBrush =
          (SolidBrush)Brushes.White;
      string s;
      int N = Nodes.Length;

      //Line
      for (int i = 0; i < N; i++)
```

```
{
   if (Nodes[i].Edge != null)
   {
      int L = Nodes[i].Edge.Length;
      MyBrush.Color = Color.White;
      for (int j = 0; j < L; j++)
      {
        switch (typ_graph)
        {
           case 0:
             if (Nodes[i].Edge[j].select)
               MyPen = Pens.Red;
             else
               MyPen= new
                    Pen(Nodes[i].Edge[j].color);
             int a1 = Nodes[i].x;
             int b1 = Nodes[i].y;
             int a2 =
                  Nodes[Nodes[i].Edge[j].
                  numNode].x;
             int b2 =
               Nodes[Nodes[i].Edge[j].
               numNode].y;
             g.DrawLine(MyPen,
               new Point(a1,b1),
               new Point(a2, b2));
             s = Convert.ToString(Nodes[i].
               Edge[j].A);
             SizeF size = g.MeasureString(s,
               MyFont);
             if (Lib.graph.visibleA)
             {
                g.FillRectangle(Brushes.White,
                  (a1+a2)/2 - size.Width / 2,
                  (b1+b2)/2 - size.Height/ 2,
                   size.Width, size.Height);
                g.DrawString(s, MyFont,
                   Brushes.Black,
                   (a1+a2)/2- size.Width / 2,
                   (b1+b2)/2-size.Height / 2);
              }
              break;
         }
      }
   }

   // Nodes
   for (int i=0; i<N; i++)
```

```
        {
          if (Nodes[i] == SelectNode)
             MyPen = Pens.Red;
          else
             MyPen = Pens.Silver;
          if (Nodes[i].visit)
             MyBrush.Color = Color.Silver;
          else
            if (Nodes[i] == SelectNode)
              MyBrush.Color = Color.Yellow;
            else
              MyBrush.Color = Color.LightYellow;
          switch (typ_graph)
          {
            case 0:
              MyBrush.Color = Nodes[i].color;
              g.FillEllipse(MyBrush,Nodes[i].x-hy,
                 Nodes[i].y - hy, 2 * hy, 2 * hy);
              g.DrawEllipse(Pens.Black,
                 Nodes[i].x-hy,
                 Nodes[i].y-hy,2*hy,2*hy);
              s = Convert.ToString(i);
              SizeF size =
                  g.MeasureString(s,MyFont);
              g.DrawString(s, MyFont, Brushes.Black,
                  Nodes[i].x - size.Width/2,
                  Nodes[i].y - size.Height/2);
              break;
         }
         if (fl)
           g.DrawLine(MyPen, new Point(x1,y1),
               new Point(x2,y2));
    }
}
```

Как и раньше, для перетаскивания узлов и создания новых дуг реализованы обработчики событий *onMouseDown*, *onMouseMove* и *onMouseUp*.

4.4.3. Чтение и запись графов

Запись графа будем осуществлять с помощью файлового потока, который создадим с помощью класса `FileStream`. Запись состоит из следующих шагов:

1) создать экземпляр класса *FileStream*;

2) создать байтовый массив *byData[]*, в который поместятся все данные о графе;

3) заполнить массив данными из графа;

4) записать массив *byData[]*;
5) закрыть файловый поток.
Все эти шаги реализованы в методе *Save()*.

Листинг 4.54. Запись данных о графе

```
public void Save(string FileName)  // записать
{
    ofs = 0;
    FileStream aFile =
       new FileStream(FileName, FileMode.Create);
    int N = LengthFile();
    byData = new byte[N];
    int L1 = Nodes.Length;
    IntInData(L1);
    for (int i = 0; i <= L1 - 1; i++)
    {
          IntInData(Nodes[i].x);
          IntInData(Nodes[i].y);
          StrInData(Nodes[i].name);
          int L2 = 0;
          if (Nodes[i].Edge != null)
               L2 = Nodes[i].Edge.Length;
          IntInData(L2);
          for (int j = 0; j <= L2 - 1; j++)
          {
              IntInData(Nodes[i].Edge[j].A);
              IntInData(Nodes[i].Edge[j].x1c);
              IntInData(Nodes[i].Edge[j].x2c);
              IntInData(Nodes[i].Edge[j].yc);
              IntInData(Nodes[i].Edge[j].numNode);
          }
    }
    aFile.Write(byData, 0, N);
    aFile.Close();
}
```

Для записи потребовалось три вспомогательных метода. Первый – *LengthFile()* – предназначен для вычисления длины байтового массива, в который поместятся все данные о массиве.

Листинг 4.55. Вычисление длины байтового массива

```
protected int LengthFile()  // вычислить размер файла
{
    int n = 4;
    int L1 = Nodes.Length;
    for (int i=0; i<=L1-1; i++)
```

```
    {
          n += 16+4*Nodes[i].name.Length;
          int L2=0;
          if (Nodes[i].Edge != null)
              L2 = Nodes[i].Edge.Length;
          n += L2 * 20;
    }
    return n;
}
```

Второй метод перемещает переменную целого значения в байтовый массив и сдвигает смещение *ofs* на 4.

Листинг 4.56. Перемещение целого значения в байтовый массив

```
protected void IntInData(int k)
{
    byte[] byByte;
    byByte = BitConverter.GetBytes(k);
    byByte.CopyTo(byData, ofs); ofs += 4;
}
```

Для перемещения строки в байтовый массив предназначен метод *StrInData()*. Так как предполагается использование кириллицы в строках, то приходится использовать кодировку *UTF-32*, которая требует 4 байта на символ. Перемещение происходит в четыре этапа:

1) перемещаем длину строки в основной байтовый массив *byData*;

2) перемещаем строку в символьный массив *charData[]*;

3) с помощью класса *Encoder* перемещаем символьный массив во вспомогательный байтовый массив *byByte[]*;

4) вставляем вспомогательный байтовый массив *byByte[]* в основной байтовый массив *byData*.

Листинг 4.57. Перемещение строки в байтовый массив

```
protected void StrInData(string s)
{
    byte[] byByte;
    int L = s.Length; IntInData(L);
    char[] charData = s.ToCharArray();
    byByte = new byte[4 * charData.Length];
    Encoder e = Encoding.UTF32.GetEncoder();
    e.GetBytes(charData, 0, charData.Length, byByte,
       0, true);
    byByte.CopyTo(byData, ofs); ofs += 4 * L;
}
```

Чтение файла происходит в том же порядке (листинг 4.58):

1) создать экземпляр класса *FileStream*;

2) создать байтовый массив *byData[]*, в который поместятся все данные о графе;

3) прочитав файл, заполнить массив данными из файла;

4) пройдя по массиву *byData[]*, создать все узлы и ребра графа;

5) закрыть файловый поток.

Листинг 4.58. Чтение данных о графе

```
public void Read(string FileName) // прочитать
{
    ofs = 0;
    FileStream aFile =
       new FileStream(FileName, FileMode.Open);
    int N = (int)aFile.Length;
    byData = new byte[N];
    aFile.Read(byData, 0, N);
    int L1 = DataInInt();
    Nodes = new TNode[L1];
    for (int i = 0; i <= L1 - 1; i++)
    {
          Nodes[i] = new TNode();
          Nodes[i].x = DataInInt();
          Nodes[i].y = DataInInt();
          Nodes[i].name = DataInStr();
          int L2 = DataInInt();
          Nodes[i].Edge = new TEdge[L2];
          if (L2 != 0)
              for (int j = 0; j <= L2 - 1; j++ )
    {
       Nodes[i].Edge[j].A = DataInInt();
       Nodes[i].Edge[j].x1c = DataInInt();
       Nodes[i].Edge[j].x2c = DataInInt();
       Nodes[i].Edge[j].yc = DataInInt();
       Nodes[i].Edge[j].numNode = DataInInt();
       Nodes[i].Edge[j].color = Color.Silver;
    }
    }
    aFile.Close();
}
```

Для чтения потребовался вспомогательный метод *DataInInt()*, который извлекает целое значение из массива *byData[]*

Листинг 4.59. Извлечение целого значения из массива byData[]

```
        protected int DataInInt()
        {
            int result = BitConverter.ToInt32(byData, ofs);
```

```
            ofs += 4;
            return result;
        }
```

и метод извлечения строки из массива *byData[]*.

Листинг 4.60. Извлечение строки из массива byData[]

```
protected string DataInStr()
{
    byte[] byByte;
    int L = DataInInt();
    byByte = new byte[4 * L];
    for (int j = 0; j <= 4 * L - 1; j++)
          byByte[j] = byData[j + ofs];
    char[] charData = new char[L];
    Decoder d = Encoding.UTF32.GetDecoder();
    d.GetChars(byByte,0,byByte.Length,charData, 0);
    string s = "";
    for (int j = 0; j < charData.Length; j++)
          s += charData[j];
    ofs += 4 * L;
    return s;
}
```

Извлечение строки снова происходит в четыре этапа:

1) извлекаем длину строки *L*;

2) заполняем вспомогательный массив *byByte[]* длиной *4*L*;

3) с помощью класса *Decoder* перемещаем данные из массива *byByte[]* в символьный массив *charData[]*;

4) перемещаем данные из массива *charData[]* в строку.

Глава 5. **ВЕКТОРНЫЙ РЕДАКТОР**

Векторный редактор типа редактора *Corel Draw* значительно сложнее растрового. Даже существенно «облегченная» версия проекта, представленная ниже, занимает места в 5 раз больше, чем растровый редактор. Внешний вид редактора представлен на рисунке 5.1.

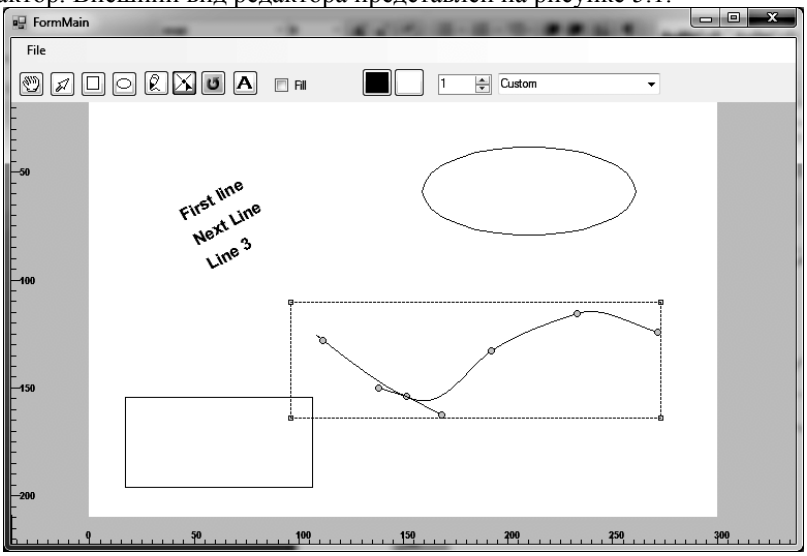

Рис. 5.1. Многооконный векторный редактор

Большую часть классов, необходимых для работы проекта, соберем в файле *ClassLib.cs*. Из функциональных возможностей в проекте оставлены следующие: рисование прямоугольника, эллипса, линии Безье и фрагмента текста; перетаскивание, масштабирование и поворот этих объектов. Режим работы этих методов назначают кнопки класса *RadioButton*, изменяющие значение переменной *flTools*.

Таблица 5.1

Инструментальные кнопки

Name	**Tag=tl_...**	**Назначение**
ButtonMove	*tl_Move=0*	Перемещать объект
ButtonPen	*tl_AddLineBz=1*	Рисовать линию Безье

Продолжение табл. 5.1

Name	Tag=tl_...	Назначение
ButtonRotate	*tl_Rotate=4*	Вращать объект
ButtonMovePoint	*tl_MovePoint=8*	Перемещать точку
ButtonText	*tl_Text=10*	Вставить текст
ButtonRect	*tl_Rect=17*	Рисовать прямоугольник
ButtonEllipse	*tl_Ellipse=18*	Рисовать эллипс

5.1. Структура данных

Структура классов проекта представлена на рисунке 5.2.

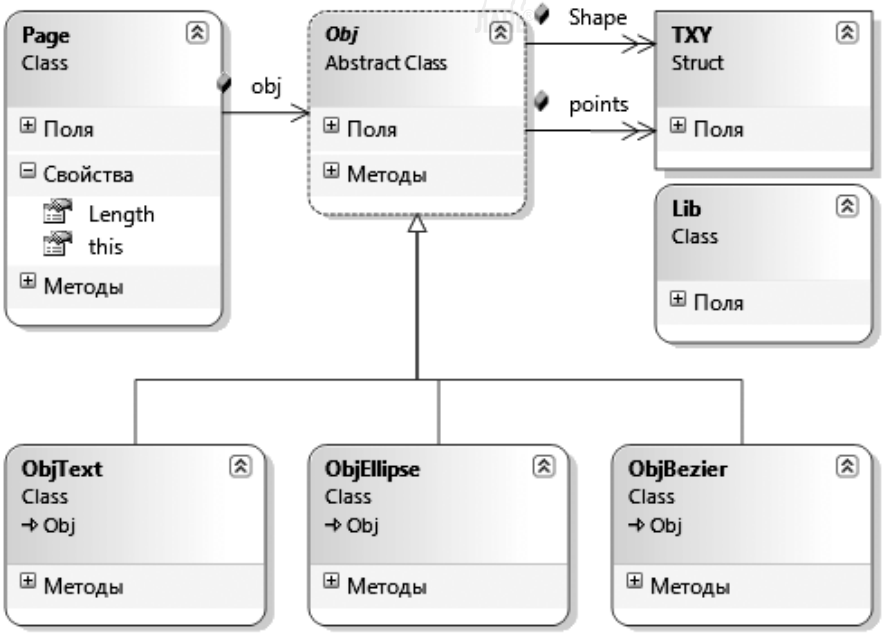

Рис. 5.2. Структура классов проекта

Одна из основных проблем, возникающих при создании векторного редактора, это проблема масштабирования. Поэтому в основном классе проекта *Page* (листинг 5.1)

Листинг 5.1. Основной класс векторного редактора

```
class Page
{
    public static double xMin, yMin, xMax, yMax;
    public static double pageWidth, pageHeight;
    private int length;
    public Obj[] obj;
    public int Length
    { get { return length; } }
    public Obj this[int index]
    {
        get { return obj[index]; }
    }
}
```

представлены поля *xMin, yMin, xMax, yMax*, задающие окно на бумаге, ширину листа бумаги *pageWidth* и высоту листа бумаги *pageHeight*. Информация об объектах содержится в массиве *obj*, каждый элемент которого является объектом класса *Obj*.

Индексатор

```
public Obj this[int index]
```

позволяет обращаться к элементам массива *obj[]* через экземпляр класса *Page*. Класс геометрических объектов *Obj* объявлен абстрактным.

Листинг 5.2. Класс геометрических объектов

```
public abstract class Obj
{
    public bool select;  // признак выделения
    public Color pColor; // цвет линии
    public byte pWidth;  // толщина линии
    public TXY Pc;       // центр поворота
    public double a;     // угол поворота
    public TXY[] Shape = new TXY[4];//
    public byte typeObj; //
    public Pen pen;      //
    public TXY[] points; //
}
```

Каждый объект может быть выделен, за выделение отвечает поле *select* (рис. 5.3).

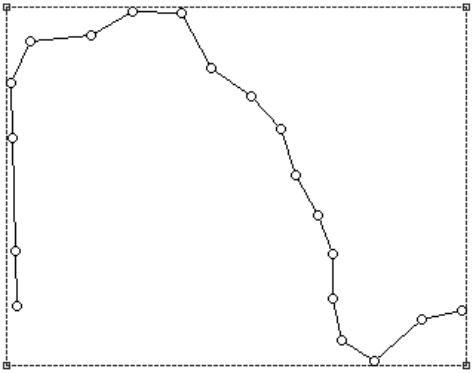

Рис. 5.3. Выделенный векторный объект

У каждого объекта есть свойства: цвет линии *color*; стиль линии *pStyle*; толщина линии *pWidth*; цвет заливки *bColor*; стиль заливки *bStyle*. Объект можно поворачивать, точка *Pc* типа *TXY* задает точку поворота. Вокруг выделенного объекта рисуется пунктирный четырехугольник, вершины которого определяются массивом из четырех элементов *Shape* типа *TXY*. В дальнейшем мы будем называть эти точки «шейпами» с номерами 0...3. За вид объекта отвечает поле *typeObj*, принимающее значения: *toBezierPen* = 6 для незамкнутой линии Безье; *toText* = 10 для фрагмента текста; *toRect* = 4 для прямоугольника; *to_Ellipse* = 5 для эллипса. Самым важным полем в классе *Obj* является поле *points*, представляющее собой массив координат точек типа *TXY*.

Листинг 5.3. Структура координат точек

```
public struct TXY
{
    public double x;
    public double y;
    public double alf;
}
```

В этой структуре помимо обычных координат *x,y* добавлено поле *alf*, отвечающее за угол касательной к линии в точке. Это поле необходимо для сглаживания кривых в точках сопряжения фрагментов линий Безье. Более подробно мы будем обсуждать кривые Безье в следующем пункте.

Класс *Lib* играет вспомогательную роль. В нем введен ряд статических полей, доступных из всех классов.

Листинг 5.4. Структура координат точек

```
public class Lib
{
    public static int numPointNode;
    public static int numPoint;
    public static int numObj;
    public static int numShape;
    public static Color defaultColor;
    public static byte defaultWidth;
}
```

В этом классе:

- поле *numObj* отвечает за номер выделенного объекта;
- поле *numShape* указывает на номер шейпа, то есть за номер угловой точки прямоугольника выделения;
- поле *numPoint* содержит номер выделенной точки линии;
- поле *numPointNode* содержит номер точки сопряжения линий Безье;
- поле *defaultColor* указывает на цвет линии по умолчанию;
- поле *defaultWidth* указывает на толщину линии по умолчанию.

5.2. Масштабирование

Проект оперирует со следующими геометрическими объектами:

- *Страница*. Весь проект – это массив страниц *pages* типа *Page* или, в нашем случае, одна страница *page*. У каждой страницы есть ширина *pageWidth* и высота *pageHeight*, измеряемые в миллиметрах;
- *Физический прямоугольник*. Это реальный двухмерный участок, задаваемый параметрами *xMin,yMin,xMax,yMax*. Эти параметры измеряются в километрах, метрах или сантиметрах;
- *Экран монитора или принтер*, на который проецируются страницы с окнами. В этом случае мы имеем дело с пикселями формы или канвы принтера, положение которых по горизонтали и вертикали задается целыми числами. Ширина формы задается параметром *I2*, а высота – параметром *J2*.

Необходимо спроецировать физический прямоугольник на страницу, а страницу спроецировать на экран или на канву принтера с возможностью масштабирования и скроллинга.

В этом пункте выведем функциональную зависимость между координатами экрана и координатами страницы. При работе с экраном монитора необходимо обеспечить две функциональные возможности: *масштабирование* и *скроллинг*.

Для построения изображения на бумаге выбирается прямоугольник с размерами (*xMin, xMax*)*(*yMin, yMax*). Основная проблема,

возникающая при построении изображения на экране монитора, обусловлена необходимостью масштабировать прямоугольник (*xMin,xMax*) × (*yMin,yMax*) в прямоугольник на экране (*I1,I2*) × (*J1,J2*) (рис. 5.4).

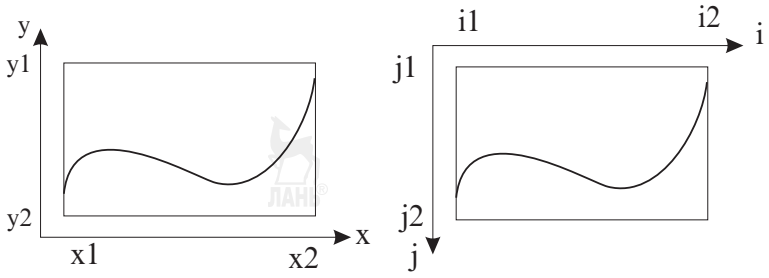

Рис. 5.4. Масштабирование графиков

Масштабирование по осям *OX* и *OY* реализуется при помощи линейных зависимостей (5.1), (5.2), графики которых представлены на рисунке 5.5.

$$\frac{x - X_1}{X_2 - X_1} = \frac{i - I_1}{I_2 - I_1}; \tag{5.1}$$

$$\frac{y - Y_1}{Y_2 - Y_1} = \frac{j - J_2}{J_1 - J_2}. \tag{5.2}$$

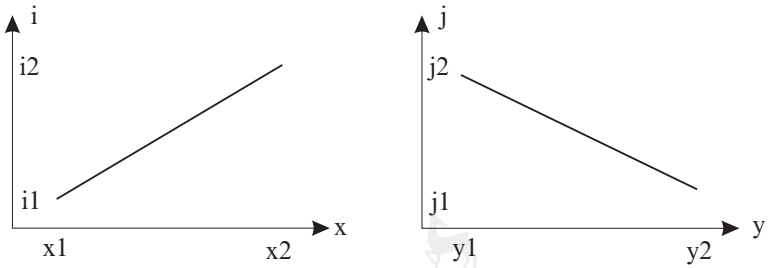

Рис. 5.5. Функции масштабирования по осям OX и OY

Уравнение (5.1) реализовано в виде метода масштабирования *II(x)*, текст которого приведен в листинге 5.5.

Листинг 5.5. Прямой метод масштабирования горизонтальных координат

```
public static int II(double x)
{
    return I1+(int)((x-Page.xMin)*
        (I2-I1)/(Page.xMax-Page.xMin));
}
```

Решив уравнение (5.1) относительно *x*, найдем зависимость горизонтальной страничной координаты от экранной координаты *I*. В листинге 5.6 приведен метод обратного масштабирования горизонтальных координат.

Листинг 5.6. Обратный метод масштабирования горизонтальных координат

```
static double XX(int I)
{
    return Page.xMin+(I-I1)*(Page.xMax - Page.xMin)
        / (I2 - I1);
}
```

Точно так же можно найти закон преобразования для вертикальных координат из уравнения (5.2) в виде метода масштабирования *JJ(y)*, текст которого приведен в листинге 5.7.

Листинг 5.7. Прямой метод масштабирования вертикальных координат

```
public static int JJ(double y)
{
     return J1 + (int)((y - Page.yMin) * (J2 - J1) /
         (Page.yMax - Page.yMin));
}
```

Решив уравнение (5.2) относительно *y*, найдем зависимость вертикальной страничной координаты от экранной координаты *J*. В листинге 5.8 приведен метод обратного масштабирования вертикальных координат.

Листинг 5.8. Метод обратного масштабирования вертикальных координат

```
static double YY(int J)
{
    return Page.yMin + (J-J1) * (Page.yMax-
         Page.yMin) / (J2-J1);
}
```

Изменять размеры окна можно двумя способами: масштабированием при помощи колеса мыши и перетаскиванием окна мышью.

Для подключения обработчика события вращения колеса мыши необходимо, во-первых, в момент начальной загрузки приложения через делегата *MouseEventHandler* передать имя обработчика события *FormMain_MouseWheel*.

Листинг 5.9. Добавление обработчика события

```
private void FormMain_Load(object sender, EventArgs e)
{
```

```
    MouseWheel += new
        MouseEventHandler(FormMain_MouseWheel);
}
```

Во-вторых, написать код метода `FormMain_MouseWheel()`, реализущего масштабирование.

Листинг 5.10. Изменение размеров окна

```
void FormMain_MouseWheel(object sender, MouseEventArgs e)
{
    double x = XX(e.X);
    double y = YY(e.Y);
    if (e.Delta < 0)
          coeff = 1.03F;
    else
          coeff = 0.97F;
    Page.xMin = x - (x - Page.xMin) * coeff;
    Page.xMax = x + (Page.xMax - x) * coeff;
    Page.yMin = y - (y - Page.yMin) * coeff;
    Page.yMax = y + (Page.yMax - y) * coeff;
    Page.SetY();
    Draw();
}
```

В этом методе или увеличиваются, или уменьшаются габариты окна (*xMin, xMax*) * (*yMin, yMax*) относительно точки, в которой находится указатель мыши (*e.X,e.Y*).

Статический метод *SetY()* класса *Page*

Листинг 5.11. Выравнивание пропорций окна

```
public static void SetY()
{
    int kx =
        Screen.PrimaryScreen.Bounds.Width;
    int ky =
        Screen.PrimaryScreen.Bounds.Height;
    yMax = ky*(xMax-xMin)/kx/2F + pageHeight / 2F;
    yMin = -ky*(xMax-xMin)/kx/2F + pageHeight / 2F;
}
```

необходим для выравнивания пропорций окна на бумаге по вертикали и горизонтали в зависимости от отношения размеров экрана.

Перетаскивание окна мышью при *flTools = 0* происходит в обработчике события *FormMain_MouseMove()*.

Листинг 5.12. Изменение размеров окна при перетаскивании

```
void FormMain_MouseMove(object sender, MouseEventArgs e)
{
   if (drawing)
       switch (flTools)
       {
           case 0:
               double dx = XX(e.X) - XX(e0.X);
               double dy = YY(e.Y) - YY(e0.Y);
               Page.xMin -= dx; Page.yMin -= dy;
               Page.xMax -= dx; Page.yMax -= dy;
               Draw();
               break;
        }
}
```

5.3. Создание объектов

Начнем с самых простых объектов – прямоугольника и эллипса. При нажатии кнопки мыши мы, как обычно, поднимаем флаг перемещения *drawing = true*, запоминаем координаты левого верхнего и правого нижнего углов.

Листинг 5.13. Начало рисования прямоугольника и текста

```
void FormMain_MouseDown(object sender, MouseEventArgs e)
{
    drawing = true;
    e0 = e;
    switch (flTools)
    {
        case 4: case 5:// rect
            drawing=true;
            x0=e.X; y0=e.Y; xt=e.X; yt=e.Y;
            break;
    }
}
```

При перемещении мыши стираем фокусный четырехугольник, меняем координаты нижнего правого угла и рисуем фокусный четырехугольник в новом месте.

Листинг 5.14. Перемещение мыши при рисовании прямоугольника и эллипса

```
void FormMain_MouseMove(object sender, MouseEventArgs e)
{
    if (drawing)
    switch (flTools)
```

```
    {
        case 4: // Add Rect
           xt=e.X; yt=e.Y;
           Draw();
           g0.DrawRectangle(Pens.Black,x0,y0,
               xt-x0,yt-y0);
           break;
        case 5: // Add Ellipse
           xt = e.X; yt = e.Y;
           Draw();
           g0.DrawEllipse(Pens.Black, x0, y0,
               xt - x0, yt - y0);
           break;
     }
}
```

Создание нового элемента заканчивается в обработчике события *FormMain_MouseUp()*.

Листинг 5.15. Завершение создания прямоугольника и эллипса

```
void FormMain_MouseUp(object sender, MouseEventArgs e)
{
   drawing = false;
   switch (flTools)
   {
       case 4:
          Lib.numObj = page.AddRect(flTools, XX(x0),
              YY(y0), XX(xt), YY(yt));
          page[Lib.numObj].MakeShape();
          Draw();
          break;
        case 5:
          Lib.numObj = page.AddEllipse(flTools,
              XX(x0), YY(y0), XX(xt), YY(yt));
          page[Lib.numObj].MakeShape();
          Draw();
          break;
    }
}
```

Если мы создаем прямоугольник, то при отпускании кнопки мыши увеличиваем на 1 длину массива *obj[]*, конструктором *ObjRect()* задаем параметры объекта (*pStyle, pColor, pWidth, pColor, bStyle, typeObj*), создаем массив *points* длиной 4 элемента под угловые точки прямоугольника и заполняем эти элементы массива координатами точек.

Листинг 5.16. Создание экземпляра прямоугольника

```
public int AddRect(byte flTools, double u1,
    double v1, double u2, double v2)
{
    int L;
    if (obj == null) L = 0; else L = obj.Length;
    Array.Resize<Obj>(ref obj, ++L);
    length++;
    obj[L - 1] = new ObjRect();
    obj[L - 1].points = new TXY[4];
    obj[L - 1].points[0].x = u1;
    obj[L-1].points[0].y = v1;
    obj[L - 1].points[1].x = u1;
    obj[L-1].points[1].y = v2;
    obj[L - 1].points[2].x = u2;
    obj[L-1].points[2].y = v2;
    obj[L - 1].points[3].x = u2;
    obj[L-1].points[3].y = v1;
    return L - 1;
}
```

Создание эллипса заканчивается несколько иначе – зададим 30 точек на эллипсе, а сам эллипс в дальнейшем будем рисовать как замкнутую ломаную линию.

Листинг 5.17. Создание экземпляра эллипса

```
public int AddEllipse(byte flTools, double u1, double v1,
    double u2, double v2)
{
    int NUM_POINT_ELLIPSE = 30;
    int L;
    if (obj == null) L = 0; else L = obj.Length;
    Array.Resize<Obj>(ref obj, ++L);
    length++;
    obj[L - 1] = new ObjEllipse();
    obj[L - 1].points = new TXY[4];
    double x0=(u1+u2)/2; double y0=(v1+v2)/2;
    double a=Math.Abs((u1-u2)/2);
    double b=Math.Abs((v1-v2)/2);
    int Lp=NUM_POINT_ELLIPSE;
    obj[L - 1].points = new TXY[Lp];
    for (int i = 0; i <= Lp - 1; i++)
    {
       obj[L-1].points[i].x =
          x0 + a*Math.Cos(2*i*Math.PI/Lp);
       obj[L-1].points[i].y =
          y0 + b*Math.Sin(2*i*Math.PI/Lp);
```

```
    }
    return L - 1;
}
```

Вместо четырех угловых точек прямоугольника мы создаем *NUM_POINT_ELLIPSE = 60* точек на эллипсе, координаты которых вычисляем по параметрическому уравнению эллипса.

Создание любого объекта метод *MakeShape()* заканчивает расчетом координат четырех шейповых точек. Код метода приведен в листинге 5.18.

Листинг 5.18. Расчет координат четырех шейповых точек

```
public void MakeShape()
{
    for (int i = 0; i <= 3; i++)
    {
        Shape[i].x = points[0].x;
        Shape[i].y = points[0].y;
    }
    Pc.x = 0; Pc.y = 0;
    int L = points.Length;
    for (int i = 0; i <= L - 1; i++)
    {
        Pc.x = Pc.x + points[i].x;
        Pc.y = Pc.y + points[i].y;
        if (Shape[0].x > points[i].x)
            Shape[0].x = points[i].x;
        if (Shape[2].x < points[i].x)
            Shape[2].x = points[i].x;
        if (Shape[0].y > points[i].y)
            Shape[0].y = points[i].y;
        if (Shape[2].y < points[i].y)
            Shape[2].y = points[i].y;
    }
    Pc.x = Pc.x / L;
    Pc.y = Pc.y / L;
    a = Math.Atan2(points[0].y-
        Pc.y,points[0].x-Pc.x);
    Shape[1].x = Shape[2].x;
    Shape[1].y = Shape[0].y;
    Shape[3].x = Shape[0].x;
    Shape[3].y = Shape[2].y;
}
```

Наиболее сложным процессом является рисование кривой Безье. На начальном этапе добавляется только одна точка в массив *Points*.

Листинг 5.19. Начало создания кривой Безье

```
void FormMain_MouseUp(object sender,
    MouseEventArgs e)
{
   drawing = false;
   switch (flTools)
   {
       case 6: // Add Line Bezier
          page.UnSelect();
          drawing = true;
          Lib.numObj =
              page.AddBezier(flTools,XX(e.X),
              YY(e.Y));
          x0=e.X; y0=e.Y; xt=e.X; yt=e.Y;
          break;
   }
}
```

Метод *AddBezier()* добавления объекта класса *ObjBezier*.

Листинг 5.20. Добавление объекта кривой Безье

```
int AddBezier(byte flTools, double u, double v)
{
    int L;
    if (obj == null) L = 0; else L = obj.Length;
    Array.Resize<Obj>(ref obj, ++L);
    length++;
    obj[L - 1] = new ObjBezier();
    obj[L - 1].select = true;
    obj[L - 1].points = new TXY[1];
    obj[L - 1].points[0].x = u;
    obj[L - 1].points[0].y = v;
    return L-1;
}
```

Увеличиваем на 1 длину массива *obj[]*, конструктором *ObjBezier()* задаем параметры объекта (*pStyle, pColor, pWidth, pColor, bStyle, typeObj*), создаем массив *points* длиной 1 элемент под первую точку линии и заполняем этот элемент массива координатами точек.

Напомним, что кривая Безье проходит через $3N + 1$ точку, поэтому при перемещении мыши мы должны добавлять по 3 точки, если отклонение положения мыши от последней точки кривой больше *DELTA_BZ = 40* пикселов.

Листинг 5.21. Добавление трех точек при движении мыши

```
case 6: // Bezier
    byte DELTA_BZ=40;
    if ((Math.Abs(e.X-x0)>=DELTA_BZ) |
       (Math.Abs(e.Y-y0)>=DELTA_BZ))
    {
        if (page[Lib.numObj] is ObjBezier)
           (page[Lib.numObj] as
               ObjBezier).AddBezierPoint(XX(e.X), YY(e.Y));
           Draw();
           x0=e.X; y0=e.Y;
    }
```

Метод *AddBezierPoint()* есть не у всех элементов массива *obj[]*, а только у элементов класса *ObjBezier*. Поэтому приходится проверять оператором *is*, что *obj[]*, элемент класса *ObjBezier*. Метод *AddBezierPoint()* добавляет в массив *points* три точки, корректируя их положение в соответствии с предыдущим фрагментом линии Безье.

Листинг 5.22. Добавление трех точек при движении мыши

```
public void AddBezierPoint(double u, double v)
{
    int L = points.Length;
    L = L + 3;
    Array.Resize<TXY>(ref points, L);
    points[L - 1].x = u;
    points[L - 1].y = v;

    double dx = points[L - 1].x - points[L - 4].x;
    double dy = points[L - 1].y - points[L - 4].y;
    if (L == 4) points[0].alf = Math.Atan2(dy, dx);
    points[L-1].alf = Math.Atan2(dy, dx) + Math.PI;

    if (L >= 6) //
    {
       dx = points[L - 1].x - points[L - 7].x;
       dy = points[L - 1].y - points[L - 7].y;
       points[L-4].alf=Math.Atan2(dy,dx) + Math.PI;
    }
    SetBz(L - 1);
}
```

При добавлении новых трех точек вычисляется угол в новой точке сопряжения *Points[L-1].alf*, корректируется угол в предыдущей точке сопряжения *Points[L-4].alf* и методом *SetBezier* вычисляются

координаты в добавленных промежуточных точках с номерами *L-2* и *L-3* (рис. 5.6).

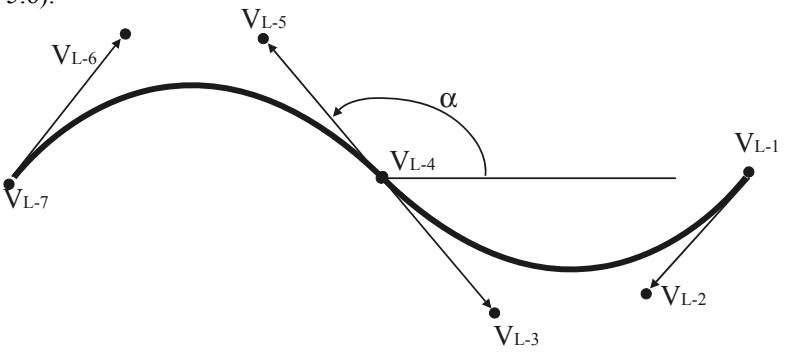

Рис. 5.6. Добавление фрагмента кривой Безье

Если во время перемещения курсор попадает в начальную точку кривой Безье, то меняется тип линии *typeObj* = 1, добавляется три новых точки и кривая замыкается.

5.4. Перемещение объектов

Любой объект окружен прямоугольником, четыре угловых точки которого определяются координатами массива *Shape* типа *TXY*. В начале перемещения мы обращаемся к методу *FindObj()*, который должен по координатам точки *u,v* попытаться определить номер объекта *numObj* и, возможно, если точка *u,v* попадает в окрестность угловой точки, номер *numShape*.

Листинг 5.23. Метод поиска объекта

```
public bool FindObj(double u, double v)
{
   double Eps = 1.0;
   Lib.numObj = -1; Lib.numShape = -1;
      Lib.numPointNode = -1;
   int L0 = obj.Length; int i = L0;
   bool Result = false;
   while ((i > 0) & !Result)
      Result = (obj[--i].Shape[0].x - Eps <= u) &
               (u <= obj[i].Shape[2].x + Eps) &
               (obj[i].Shape[0].y - Eps <= v) &
               (v <= obj[i].Shape[2].y + Eps);
   if (Result)
   {
       Lib.numObj = i; obj[i].select = true;
```

```
        bool Ok = false; i = -1;
        while ((i < 3) & !Ok)
          Ok=(Math.Abs(obj[Lib.numObj].Shape[++i].x-
             u) < Eps)
             & (Math.Abs(obj[Lib.numObj].Shape[i].y-
             v)<Eps);
        if (Ok) Lib.numShape = i; // найден Shape
    }
    return Result;
}
```

Итак, при перемещении объектов обработчик события *onMouseDown* выполняет следующие действия.

Листинг 5.24. Начало перемещения объектов

```
case 1:
    page.UnSelect();
    if ((page.obj != null) && (page.Length != 0) &
       page.FindObj(XX(e.X), YY(e.Y)))
    {
        int n = Lib.numObj;
        if (n==-1)      // Move Rect
        {
            drawing=true;
            x0=e0.X; y0=e0.Y;
        }
        else                Move Shape
            if (n != -1)
            {
                drawing=true;
                switch (n)
                {
                   case 0:
                      x0 =
                    II(page[n].Shape[2].x);
                      y0 = JJ(page[n].Shape[2].y);
                      break;
                   case 1:
                      x0 = II(page[n].Shape[0].x);
                      y0 = JJ(page[n].Shape[2].y);
                      break;
                   case 2:
                      x0=II(page[n].Shape[0].x);
                      y0 = JJ(page[n].Shape[0].y);
                      break;
                   case 3:
                      x0=II(page[n].Shape[2].x);
                      y0 = JJ(page[n].Shape[0].y);
```

```
                    break;
                }
            }
        }
        Draw();
break;
```

Метод *FindObj()* устанавливает номер объекта *numObj* и номер угловой точки прямоугольника *numShape*. Если *numShape* = -1, то мы будем перемещать объект и запоминаем координаты текущей точки *x0,y0*. Иначе перемещаем *Shape* с номером *NumShape* и в *x0,y0* запоминаем координаты противоположного угла прямоугольника.

При перемещении мыши обрабатывается событие *MouseMove*.

Листинг 5.25. Действия при перемещении мыши

```
case 1: // Move
    if (Lib.numObj != -1)
    {
        if (Lib.numShape == -1)
        {
            dx = XX(e.X) - XX(x0);
            dy = YY(e.Y) - YY(y0);
            page[Lib.numObj].MoveObj(dx, dy);
            x0 = e.X; y0 = e.Y;
        }
        else
          page[Lib.numObj].ChangeMoveShape(XX(e.X),
              YY(e.Y));
        Draw();
    }
break;
```

При перемещении всего объекта вызывается метод *MoveObj()*,

Листинг 5.26. Метод перемещения объекта

```
        public void MoveObj(double dx, double dy)
        {
            for (int i = 0; i <= 3; i++)
            {
                Shape[i].x += dx;
                Shape[i].y += dy;
            }
            int L = points.Length;
            if (L > 0)
                for (int i = 0; i <= L - 1; i++)
                {
                    points[i].x += dx;
```

```
                points[i].y += dy;
            }
        Pc.x += dx;
        Pc.y += dy;
    }
```

в которой на *dx,dy* увеличиваются координаты точек *Shape*, центра вращения и всех точек объекта *Points*.

Если же перемещается одна из угловых точек, то вызывается метод *SetMoveShape*, который реализует масштабирование точек объекта с различными неподвижными угловыми точками.

Листинг 5.27. Перемещение шейпа

```
public void ChangeMoveShape(double xm, double ym)// для всех
{
   int L = points.Length;
   double dx = Shape[2].x - Shape[0].x;
   double dy = Shape[2].y - Shape[0].y;
   double kx, ky;
   switch (Lib.numShape)
   {
      case 0:
         kx = (Shape[2].x - xm) / dx;
         ky = (Shape[2].y - ym) / dy;
         for (int i = 0; i <= L - 1; i++)
         {
             TXY p = points[i];
            p.x = xm + kx * (p.x - Shape[0].x);
            p.y = ym + ky * (p.y - Shape[0].y);
         }
         Shape[0].x = xm;
         Shape[0].y = ym;
         break;
      case 1:
         kx = (xm - Shape[0].x) / dx;
         ky = (Shape[2].y - ym) / dy;
         for (int i = 0; i <= L - 1; i++)
         {
             p.x=Shape[0].x+kx*(p.x- Shape[0].x);
             p.y = ym + ky * (p.y - Shape[0].y);
         }
         Shape[2].x = xm;
         Shape[0].y = ym;
         break;
      case 2:
         kx = (xm - Shape[0].x) / dx;
         ky = (ym - Shape[0].y) / dy;
         for (int i = 0; i <= L - 1; i++)
```

```
            {
               p.x=Shape[0].x+kx*(p.x-Shape[0].x);
               p.y=Shape[0].y+ky*(p.y-Shape[0].y);
            }
            Shape[2].x = xm;
            Shape[2].y = ym;
            break;
         case 3:
            kx = (Shape[2].x - xm) / dx;
            ky = (ym - Shape[0].y) / dy;
            for (int i = 0; i <= L - 1; i++)
            {
               p.x = xm + kx * (p.x - Shape[0].x);
               p.y=Shape[0].y+ky*(p.y-Shape[0].y);
            }
            Shape[0].x = xm;
            Shape[2].y = ym;
            break;
      }
      MakeShape();
      Shape[1].x=Shape[2].x; Shape[1].y = Shape[0].y;
      Shape[3].x=Shape[0].x; Shape[3].y = Shape[2].y;
}
```

5.5. Поворот объектов

Поворот объекта начинается в обработчике события *MouseDown* с проверки выделенности объекта *NumObj* <> -1 и определения номера выделенной угловой точки *NumShape*.

Листинг 5.28. Начало поворота объекта

```
case 7:
   drawing=(Lib.numObj!=-1) & (page.Length!=0) &
   page.FindObj(XX(e.X),YY(e.Y))&(Lib.numShape!=-1);
       Draw();
       break;
```

Если *drawing = true*, то в обработчике события *MouseMove* определяется угол между точкой *x,y* и центром вращения,

Листинг 5.29. Вращение объекта

```
case 7:
    if (Lib.numObj != -1)
    {
       double xc = page[Lib.numObj].Pc.x;
       double yc=page[Lib.numObj].Pc.y;
       double al=Math.Atan2(YY(e.Y)-yc,XX(e.X)-xc);
```

```
        if (Math.Abs(al - page[Lib.numObj].a) > 0.1)
        {
          page[Lib.numObj].RotateObj(xc,yc,al-
              page[Lib.numObj].a);
          Draw();
        }
    }
break;
```

а затем вызывается метод *RotateObj()* для изменения координат точек объекта.

Листинг 5.30. Метод изменения координат объекта при вращении

```
public void RotateObj(double x0, double y0, double al0)
{
   int L=points.Length;
   for (int i = 0; i <= L - 1; i++)
   {
        double R=Math.Sqrt((points[i].x-
            x0)*(points[i].x-x0) +
            (points[i].y-y0) * (points[i].y - y0));
        double al=Math.Atan2(points[i].y-
            y0,points[i].x-x0);
        points[i].x = x0 + R * Math.Cos(al + al0);
        points[i].y = y0 + R * Math.Sin(al + al0);
        points[i].alf = points[i].alf + al0;
   }
   MakeShape();
}
```

Поворот заканчивается пересчетом координат шейпов с помощью метода *MakeShape()*.

5.6. Перемещение точек

Перемещение точек начинается с проверки выделенности объекта и определения номера перемещаемой точки методом *FindPoint()*.

Листинг 5.31. Начало перемещения точки

```
case 8: // Move Point
    drawing = (page.Length!=0) & (Lib.numObj!=-1) &
    page[Lib.numObj].FindPoint(XX(e.X), YY(e.Y));
    Draw();
    break;
```

На самом деле метод *FindPoint()* определяет два номера точки: на первом этапе определяется номер точки сопряжения *numPointNode*, для

которой выполняется условие `i % 3 = 0`; на втором этапе проверяются промежуточные точки (`i % 3 ≠ 0`) и определяется номер точки *numPoint*.

Листинг 5.32. Метод поиска точки

```
public bool FindPoint(double u, double v)
{
   Lib.numPoint = -1; double Eps = 1.0;
   int L = points.Length; int i = -3;
   bool Result = false;
   while ((i < L - 1) & !Result)
   {
      i += 3;
      Result = (Math.Abs(points[i].x - u) < Eps) &
               (Math.Abs(points[i].y - v) < Eps);
   }
   if (Result)
   {
      Lib.numPointNode = i; Lib.numPoint = i;
   }
   else
   {
      i = -1;
      while ((i < L - 1) & !Result)
         if (++i % 3 != 0)
             Result=(Math.Abs(points[i].x-u)<Eps) &
                 (Math.Abs(points[i].y - v) < Eps);
         if (Result) Lib.numPoint = i;
         else Lib.numPointNode = -1;
   }
   return Result;
}
```

5.7. Прорисовка объектов

Для ускорения рисования объектов реализуется метод *DrawObj* на канве *bitmap* с последующим копированием на канву формы.

Листинг 5.33. Вызов методов рисования объектов

```
public void Draw()
{
   I2 = ClientSize.Width; J2 = ClientSize.Height;
   using (Graphics g = Graphics.FromImage(bitmap))
   {
      g.Clear(Color.LightBlue);
      g.FillRectangle(Brushes.White,II(0), JJ(0),
         II(Page.pageWidth)-
```

```
            II(0),JJ(Page.pageHeight)-JJ(0));
         int L;
         if (page.obj == null) L = 0;
         else
            L = page.obj.Length;
         for (int i = 0; i < L; i++)
         {
            page[i].DrawObj(g, II, JJ);
            if (Lib.numObj == i)
            {
               DrawShape(g, page[i].Shape);
               page[i].DrawSelect(g, II, JJ);
               if (Lib.numPointNode != -1)
                  page[i].DrawLine_Ellipse(g);
               if (flTools == 7)
                  page[i].DrawRotate(g, II, JJ);
            }
         }
      }
      g0.DrawImage(bitmap, ClientRectangle);
}
```

Рисование объектов *ObjRect* и *ObjEllipse* как замкнутых ломаных линий осуществляется виртуальным методом базового класса *Obj*.

Листинг 5.34. Рисование объектов ObjRect и ObjEllipse

```
public virtual void DrawObj(Graphics g,TIJ II,TIJ JJ)
{
    pen = GetPen();
    int L = points.Length;
    for (int i = 0; i < L; i++)
       g.DrawLine(pen, II(points[i].x),
          JJ(points[i].y),
          II(points[(i+1)%L].x),
          JJ(points[(i + 1) % L].y));
}
```

Класс *ObjBezier* перекрывает этот метод, вызывая стандартный метод *DrawBeziers()*.

Листинг 5.35. Рисование объектов ObjBezier

```
public override void DrawObj(Graphics g,
    TIJ II,TIJ JJ)
{
    int L = points.Length;
    if (typeObj == 6)
    {
        Pen pen = GetPen();
```

```
        IBz = new Point[L];
        for (int i = 0; i <= L - 1; i++)
        IBz[i] =
            new Point(II(points[i].x),
                JJ(points[i].y));
        g.DrawBeziers(pen, IBz);
    }
}
```

Класс *ObjText* также перекрывает метод *DrawObj()*.

Листинг 5.36. Рисование объектов DrawObj

```
public override void DrawObj(Graphics g, TIJ II, TIJ JJ)
{
    double alpha =-Math.Atan2(points[3].y-points[0].y,
    points[3].x-points[0].x);
    int L = text.Length;
    double dx = (points[0].x) - (points[1].x);
    double dy = (points[0].y) - (points[1].y);
    double h = Math.Sqrt(dx*dx + dy*dy) / L;
    for (int i = 0; i < L; i++)
    {
      double p = Math.PI/2;
      g.TranslateTransform(
           II(points[0].x + i*h*Math.Cos(-alpha+p)),
           JJ(points[0].y+i*h*Math.Sin(-alpha+p)));
      g.RotateTransform((int)(-alpha*360/p));
      g.DrawString(text[i],aFont,Brushes.Black,0,0);
      g.RotateTransform((int)(alpha * 360 / p));
      g.TranslateTransform(
          -II(points[0].x+i*h*Math.Cos(-alpha+p)),
          -JJ(points[0].y+i*h*Math.Sin(-alpha+p)));
    }
}
```

Каждая строка перед выводом методом *TranslateTransform()* смещается в свою точку поворота, затем методом *RotateTransform()* поворачивается на угол *alpha*. После вывода строки матрица преобразования координат возвращается в исходное состояние обратными преобразованиями.

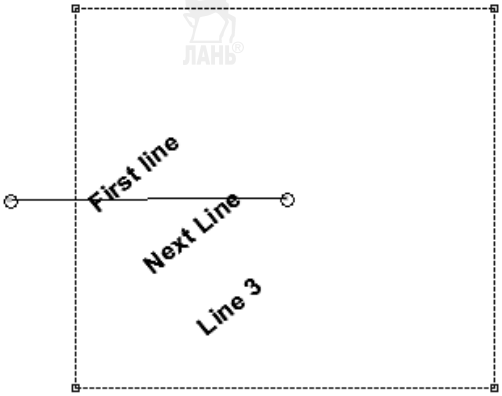

Рис. 5.7. Вывод строки под углом

Если объект выделен, то есть выполняется условие *Lib.numObj = i*, то вызывается метод рисования шейпов *DrawShape()*.

Листинг 5.37. Рисование шейпов

```
void DrawShape(Graphics g, TXY[] XY)
{
    int[] dx = { -3,  3, 3, -3 };
    int[] dy = { -3, -3, 3,  3 };
    for (int i=0; i<=3; i++)
    {
        g.DrawLine(MyPen0,II(XY[i].x)+dx[i],
           JJ(XY[i].y) + dy[i],
           II(XY[(i + 1) % 4].x) + dx[(i + 1) % 4],
           JJ(XY[(i+1) % 4].y) + dy[(i + 1) % 4]);
        g.DrawRectangle(Pens.Green,
           II(XY[i].x)- 2+dx[i],
           JJ(XY[i].y)+dy[i]-2,4,4);
    }
}
```

Если для выделенной линии Безье выполняется условие *Lib.numPointNode* != -1, то есть необходимо перемещать точку на линии, то вызывается метод рисования «усов» для выделенной точки.

Листинг 5.38. Рисование «усов» для точек объекта ObjBezier

```
public override void DrawLine_Ellipse(Graphics g)
// усы для Безье
{
   int a0 = 3;
   int n = Lib.numPointNode;
```

```
    int L = points.Length;
    if (Lib.numPointNode != (L - 1))
    {
       g.DrawLine(Pens.Black, IBz[n], IBz[n + 1]);
       g.FillEllipse(Brushes.LightGreen,
           IBz[n+1].X-a0,
           IBz[n + 1].Y - a0, 2 * a0, 2 * a0);
       g.DrawEllipse(Pens.Black, IBz[n + 1].X - a0,
           IBz[n + 1].Y - a0, 2 * a0, 2 * a0);
    }
    if (Lib.numPointNode != 0)
    {
       g.DrawLine(Pens.Black, IBz[n], IBz[n - 1]);
       g.FillEllipse(Brushes.LightGreen,
           IBz[n - 1].X - a0,
           IBz[n - 1].Y - a0, 2 * a0, 2 * a0);
       g.DrawEllipse(Pens.Black, IBz[n - 1].X - a0,
           IBz[n - 1].Y - a0, 2 * a0, 2 * a0);
    }
}
```

5.8. Печать

Печать на принтере или на графопостроителе осуществляется тем же методом *DrawObj*. Но вызов его происходит несколько иначе.

Листинг 5.39. Вызов метода рисования для печати

```
procedure TFormMain.ButtonPrintClick(Sender: TObject);
begin
  if PrintDialog1.Execute then
  if ActiveMDIChild<>nil then
  with ActiveMDIChild as TFormEd do
    DrawObj(2,IIP,JJP);
end;
```

Во-первых, при вызове метода *DrawObj* ему передаются указатели на другие методы масштабирования *IIP* и *JJP*:

```
function IIP(x: real): integer;
begin
  with Printer,Page do
  Result:=Trunc((x-xMin)*PageWidth/(xMax-xMin));
end;

function JJP(y: real): integer;
begin
  with Printer,Page do
  Result:=Trunc((y-yMin)*PageHeight/(yMax-yMin));
end;
```

которые используют ширину и высоту канвы принтера. Во-вторых, значением параметра `fl = 2` мы сообщаем методу `DrawObj` о том, что надо перед рисованием дать команду `BeginDoc`, а после рисования – команду `EndDoc`, в-третьих, рисование происходит не на канве `Bitmap`, а на канве принтера.

5.9. Запись и чтение данных

При записи проекта возникает три проблемы: 1) данные представляют собой трехуровневое дерево, так как число объектов на странице может быть произвольным, число точек объекта также может быть произвольным; 2) некоторые объекты содержат строки произвольной длины; 3) для объектов со строками необходимо сохранять параметры шрифта, а стиль шрифта – это свойство множественного типа. Все эти проблемы позволяет решить запись в файловый поток `aFile` класса `FileStream`.

Запись графа будем осуществлять с помощью файлового потока, который создадим с помощью класса `FileStream`. Запись состоит из следующих шагов:

1) создать экземпляр класса `FileStream`;

2) создать байтовый массив `byData[]`, в который поместятся все данные о графе;

3) заполнить массив данными из графа;

4) записать массив `byData[]`;

5) закрыть файловый поток.

Все эти шаги реализованы в методе `Save()`. Метод `Save()`, приведенный в листинге 5.40, осуществляет запись в файл. Обратите внимание на то, что перед записью объектов сначала записывается число объектов `L1`, а перед записью точек записывается число точек `L2`.

Листинг 5.40. Запись проекта в файловый поток

```
public void Save(string FileName)
{
    ofs = 0;

    FileStream aFile =
       new FileStream(FileName, FileMode.Create);
    int N = LengthFile();
    byData = new byte[N];

    DoubleInData(xMin);
    DoubleInData(yMin);
    DoubleInData(xMax);
    DoubleInData(yMax);
```

```
    DoubleInData(pageWidth);
    DoubleInData(pageHeight);

    int L1 = obj.Length;
    IntInData(L1);
    for (int i = 0; i <= L1 - 1; i++)
    {
          ByteInData(obj[i].typeObj);
          BoolInData(obj[i].select);

          Color pColor;
          byte R = obj[i].pColor.R,
          G = obj[i].pColor.G,
          B = obj[i].pColor.B;
          int c = 0;
          c = c | R | (G << 8) | (B << 16);
          IntInData(c);

          ByteInData(obj[i].pWidth);
          TXYInData(obj[i].Pc);
          DoubleInData(obj[i].a);
          for (int j=0; j<=3; j++)
              TXYInData(obj[i].Shape[j]);

          int L2 = obj[i].points.Length;
          IntInData(L2);
          for (int j=0; j<=L2-1; j++)
              TXYInData(obj[i].points[j]);
    }
    aFile.Write(byData, 0, N);
    aFile.Close();
}
```

Для записи потребовалось несколько вспомогательных методов. Первый – *LengthFile()* – предназначен для вычисления длины байтового массива, в который поместятся все данные о массиве.

Листинг 5.41. Вычисление длины байтового массива

```
        protected int LengthFile()
        // вычислить размер файла
        {
            int n = 6*8+4;
            int L1 = Length;
            for (int i = 0; i <= L1 - 1; i++)
            {
                n += 135+4;
                int L2 = obj[i].points.Length;
                n += L2 * 24;
```

```
        }
        return n;
    }
```

Остальные методы перемещают значения величин в байтовый массив и сдвигают смещение *ofs* на 4.

Листинг 5.42. Перемещение значений в байтовый массив

```
    protected void IntInData(int k)
    {
        byte[] byByte;
        byByte = BitConverter.GetBytes(k);
        byByte.CopyTo(byData, ofs); ofs += 4;
    }
    protected void ByteInData(byte k)
    {
        byte[] byByte;
        byByte = BitConverter.GetBytes(k);
        byByte.CopyTo(byData, ofs); ofs += 1;
    }
    protected void BoolInData(bool k)
    {
        byte[] byByte;
        byByte = BitConverter.GetBytes(k);
        byByte.CopyTo(byData, ofs); ofs += 1;
    }
    protected void DoubleInData(double k)
    {
        byte[] byByte;
        byByte = BitConverter.GetBytes(k);
        byByte.CopyTo(byData, ofs); ofs += 8;
    }
    protected void TXYInData(TXY k)
    {
        byte[] byByte;
        byByte = BitConverter.GetBytes(k.x);
        byByte.CopyTo(byData, ofs); ofs += 8;
        byByte = BitConverter.GetBytes(k.y);
        byByte.CopyTo(byData, ofs); ofs += 8;
        byByte = BitConverter.GetBytes(k.alf);
        byByte.CopyTo(byData, ofs); ofs += 8;
    }
```

Для перемещения строки в байтовый массив предназначен метод *StrInData()*. Так как предполагается использование кириллицы в строках, то приходится использовать кодировку UTF-32, которая требует 4 байта на символ. Перемещение происходит в четыре этапа:

1) перемещаем длину строки в основной байтовый массив *byData*;

2) перемещаем строку в символьный массив *charData[]*;

3) с помощью класса *Encoder* перемещаем символьный массив во вспомогательный байтовый массив *byByte[]*;

4) вставляем вспомогательный байтовый массив *byByte[]* в основной байтовый массив *byData*.

Запись полей типа строка реализует метод *WriteStr()* (листинг 5.43), в котором сначала записывается длина строки *L*, а затем *L* символов строки.

Листинг 5.43. Метод записи строки

```
protected void StrInData(string s)
{
    byte[] byByte;
    int L = s.Length; IntInData(L);
    char[] charData = s.ToCharArray();
    byByte = new byte[4 * charData.Length];
    Encoder e = Encoding.UTF32.GetEncoder();
    e.GetBytes(charData, 0, charData.Length,
        byByte, 0, true);
    byByte.CopyTo(byData, ofs); ofs += 4 * L;
}
```

Чтение файла происходит в том же порядке (листинг 5.44):

1) создать экземпляр класса *FileStream*;

2) создать байтовый массив *byData[]*, в который поместятся все данные о графе;

3) прочитав файл, заполнить массив данными из файла;

4) пройдя по массиву *byData[]*, создать все узлы и ребра графа;

5) закрыть файловый поток.

Листинг 5.44. Чтение проекта из нетипизированного файла

```
public void Read(string FileName) // прочитать
{
    ofs = 0;
    FileStream aFile =
          new FileStream(FileName, FileMode.Open);
    int N = (int)aFile.Length;
    byData = new byte[N];
    aFile.Read(byData, 0, N);

    xMin=DataInDouble();
    yMin=DataInDouble();
    xMax=DataInDouble();
    yMax=DataInDouble();
    pageWidth=DataInDouble();
    pageHeight=DataInDouble();
```

```
    int L1 = DataInInt();
    obj = new Obj[L1];
    for (int i = 0; i <= L1 - 1; i++)
    {
     byte b = DataInByte();
     switch (b)
     {
        case 4: obj[i] = new ObjRect(); break;
        case 5: obj[i] = new ObjEllipse();
          break;
       case 6: obj[i] = new ObjBezier();
         break;
     }
     obj[i].typeObj=b;
     obj[i].select=DataInBool();

     Color pColor;
     int c = DataInInt();
     byte R=(byte)(c&0xFF);
     byte G = (byte)(c & (0xFF00>>8));
     byte B = (byte)(c & (0xFF0000>>16));

     obj[i].pColor = Color.FromArgb(R, G, B);

     obj[i].pWidth=DataInByte();
     obj[i].Pc=DataInTXY();
     obj[i].a=DataInDouble();
     for (int j = 0; j <= 3; j++)
         obj[i].Shape[j]=DataInTXY();

     int L2 = DataInInt();
     obj[i].points = new TXY[L2];
     for (int j = 0; j <= L2-1; j++)
         obj[i].points[j]=DataInTXY();
    }

    aFile.Close();
}
```

Для чтения строк предназначен метод *ReadStr* (листинг 5.45), который сначала считывает длину строки, назначает длину строки, а затем считывает символы строки. Извлечение строки снова происходит в четыре этапа:

1) извлекаем длину строки *L*;

2) заполняем вспомогательный массив *byByte[]* длиной 4×L;

3) с помощью класса *Decoder* перемещаем данные из массива *byByte[]* в символьный массив *charData[]*;

4) перемещаем данные из массива *charData[]* в строку.

Листинг 5.45. Чтение строки

```
protected string DataInStr()
{
    byte[] byByte;
    int L = DataInInt();
    byByte = new byte[4 * L];
    for (int j = 0; j <= 4 * L - 1; j++)
          byByte[j] = byData[j + ofs];
    char[] charData = new char[L];
    Decoder d = Encoding.UTF32.GetDecoder();
    d.GetChars(byByte,0,byByte.Length, charData, 0);
    string s = "";
    for (int j = 0; j < charData.Length; j++)
          s += charData[j];
    ofs += 4 * L;
    return s;
}
```

Для чтения потребовалось несколько вспомогательных методов, которые извлекают значения из массива *byData[]*.

Листинг 5.46. Вспомогательные методы извлечения значения из byData[]

```
protected int DataInInt()
{
    int result = BitConverter.ToInt32(byData, ofs);
    ofs += 4;    return result;
}
protected byte DataInByte()
{
    byte result = byData[ofs];
    ofs += 1;    return result;
}
protected bool DataInBool()
{
    bool result=BitConverter.ToBoolean(byData, ofs);
    ofs += 1;    return result;
}
protected double DataInDouble()
{
   double result=BitConverter.ToDouble(byData, ofs);
   ofs += 8;
   return result;
}
protected TXY DataInTXY()
{
    TXY result;
    result.x= BitConverter.ToDouble(byData, ofs);
```

```
    ofs += 8;
    result.y = BitConverter.ToDouble(byData, ofs);
    ofs += 8;
    result.alf = BitConverter.ToDouble(byData, ofs);
    ofs += 8;
    return result;
}
```

Такой способ хранения информации имеет минимум один недостаток: при изменении структуры данных проекта нет преемственности по предыдущим версиям. Но небольшое изменение позволит этот недостаток устранить: необходимо в файл записывать и читать номер версии структуры, а этот номер использывать при чтении данных.

Глава 6. ГРАФИКИ ФУНКЦИЙ

6.1. ГРАФИК ФУНКЦИИ ОДНОЙ ПЕРЕМЕННОЙ

6.1.1. СТРУКТУРА ПРОЕКТА

Проект состоит из файлов *FormMain.cs* и *ClassNode.cs*. На инструментальной форме выставлены компоненты, позволяющие менять размеры окна на бумаге, рисовать или не рисовать координатную сетку (рис. 6.1). Проект позволяет строить график функции, заданной в нескольких строках оператором языка C#, строить графики производных и интегралов, вычисленных приближенно, а также строить приближенное решение дифференциального уравнения первого порядка. В проекте используются методы масштабирования *II(x)* и *JJ(y)*, обсуждавшиеся в пункте 5.2.

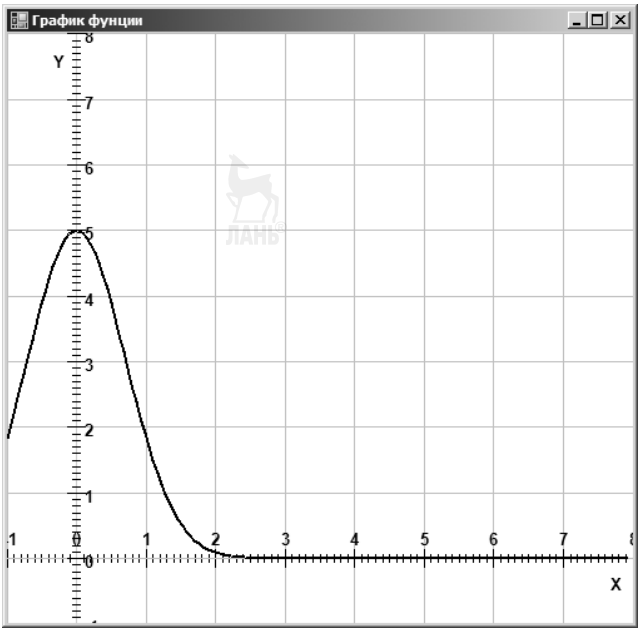

Рис. 6.1. Форма проекта построения графика функции одной переменной

При запуске проекта вызывается конструктор *FormMain*, в котором определяются габариты окна на экране.

Листинг 6.1. Активизация проекта

```
        public FormMain()
        {
            InitializeComponent();
            I2 = ClientSize.Width;
            J2 = ClientSize.Height;
        }
```

График функции строится отрезками прямых линий, соединяющих узловые точки. Для того чтобы график выглядел достаточно плавным, возьмем 150 точек по горизонтали. Для рисования вызывается метод *MyDraw()*, выводящий график.

Листинг 6.2. Метод построения графика

```
public void MyDraw(IJ II, IJ JJ, Graphics g)
{
    g = this.CreateGraphics();
    const int n = 150;
    try
    {
        Color cl = Color.FromArgb(255, 255, 255);
        g.Clear(cl);
        // Оси OX и OY
        aFont = new Font("Arial", 7,
            FontStyle.Bold);
        OX(); OY();
        aFont.Dispose();
        // график функции
        double h = (x2 - x1) / n;
        for (int i = 1; i<n; i++)
           g.DrawLine(Pens.Black,II(x1+(i-1)*h),
              JJ(F(x1+(i-1)*h)), II(x1+i*h),
              JJ(F(x1+i*h)));
     }
     finally
     {
        if (g != null)
           ((IDisposable)g).Dispose();
     }
}
```

Ось *OX* строится достаточно просто (листинг 6.3).

Листинг 6.3. Построение оси OX

```
void OX(IJ II, IJ JJ, Graphics g)
{
```

```
    g.DrawLine(Pens.Black, II(x1), JJ(0),
        II(x2), JJ(0));
    double h1 = HH(x1, x2);
    int k1 = (int)Math.Round(x1 / h1) - 1;
    int k2 = (int)Math.Round(x2 / h1);
    byte Digits = GetDigits(Math.Abs(x2 - x1));
    for (int i = k1; i <= k2; i++)
    {
        g.DrawLine(Pens.Black,
           II(i*h1),JJ(0)-7,II(i*h1),JJ(0)+7);
        for (int j = 1; j <= 9; j++)
           g.DrawLine(Pens.Black,II(i*h1+j*h1/10),
              JJ(0)-3, II(i*h1+j*h1/10),JJ(0)+3);
        string s = Convert.ToString(
            Math.Round(h1 * i, Digits));
        g.DrawString(s, aFont, Brushes.Black,
            II(i*h1)-5,JJ(0)-13);
    }
}
```

На осях системы координат выводятся деления. Проблема заключается в том, что при разных значениях *X1,X2* число делений, а это определяет шаг, с которым выводятся деления, должно быть примерно одинаковым. Метод *HH(a1,a2)* (листинг 6.4) позволяет вычислить такой шаг делений любого интервала (*a1,a2*), при котором число делений всегда будет в диапазоне 5...10.

Листинг 6.4. Вычисление шага

```
        double HH(double a1,double a2)
        {
            double Result=1;
            while (Math.Abs(a2-a1)/Result<1)
                Result/=10.0;
            while (Math.Abs(a2-a1)/Result>=10)
                Result*=10.0;
            if (Math.Abs(a2-a1)/Result<2.0)
                Result/=5.0;
            if (Math.Abs(a2-a1)/Result<5.0)
                Result/=2.0;
            return Result;
        }
```

Другой полезный метод *GetDigits*, используемый при построении осей, позволяет определить число знаков мантисы в зависимости от длины интервала *dx*.

Листинг 6.5. Установка формата

```
byte GetDigits(double dx)
{
    byte Result;
    if (dx>=5) Result =0;
    else
       if (dx>=0.5) Result=1;
       else
          if (dx>=0.05) Result=2;
          else
             if (dx>=0.005) Result=3;
             else
                if (dx>=0.0005) Result=4;
                else Result=5;
     return Result;
 }
```

Вычисленное значение *Digits* используется при печати числа, определяющего деление:

```
string s = Convert.ToString(Math.Round(h1 * i, Digits));
g.DrawString(s, aFont, Brushes.Black, II(i*h1)-5,JJ(0)-13);
```

На инструментальной панели выставлено 4 элемента класса *Button*. Все они имеют разные значения свойства *Tag* (0,1,2,3) и вызывают один обработчик события *button1_Click()*.

Листинг 6.6. Изменение размеров окна

```
private void button1_Click(object sender,
    EventArgs e)
{
    int fl = Convert.ToInt32(
       (sender as Button).Tag);
    switch (fl)
    {
       case 0:
          x1 += 0.1; x2 += 0.1;
          break;
       case 1:
          x1 -= 0.1; x2 -= 0.1;
          break;
       case 2:
          y1 += 0.1; y2 += 0.1;
          break;
       case 3:
          y1 -= 0.1; y2 -= 0.1;
          break;
    }
```

```
    MyDraw(II,JJ,g);
}
```

В зависимости от значения свойства *Tag* происходят те или иные изменения размеров окна *X1,Y1,X2,Y2*, то есть меняется масштаб рисования. Затем вызывается метод перерисовки графика *MyDraw()*.

6.1.2. Печать

Принтер, как и экран, обладает графическим полотном класса *Graphics*, на котором можно рисовать тем же методом *MyDraw()*. Но это полотно обладает другими шириной и высотой. Поэтому в этот метод необходимо передавать другие методы масштабирования.

Процесс печати состоит из нескольких этапов. Во-первых, необходимо ввести поле класса *PrintDocument*:

```
PrintDocument prtdoc;
```

Во-вторых, необходимо ввести параметры окна на бумаге *int I1p, I2p, J1p, J2p*, значения которых будут вычислены по установкам принтера перед печатью, и определить через них методы масштабирования.

Листинг 6.7. Методы масштабирования для принтера

```
int IIp(double x)
{
    return I1p+(int)((x-x1)*(I2p-I1p) / (x2 - x1));
}
int JJp(double y)
{
    return J2p+(int)((y-y1)*(J1p-J2p) / (y2 - y1));
}
```

В-третьих, вместо того чтобы реагировать на событие *onPaint()*, необходимо создать обработчик событий, которому можно дать любое имя, и зарегистрировать его как обработчик событий для вывода страницы. Именно в этом обработчике задаются параметры окна на бумаге и вызывается метод *MyDraw()*.

Листинг 6.8. Обработчик события для печати

```
private void prtDocPrintPage(object sender,
    PrintPageEventArgs e)
{
    e.Graphics.Clip = new Region(e.MarginBounds);
    I1p = e.MarginBounds.Left;
    I2p = e.MarginBounds.Right;
    J1p = e.MarginBounds.Top;
    J2p = e.MarginBounds.Bottom;
    MyDraw(IIp,JJp,e.Graphics);
}
```

Печать начинается выбором пункта меню или нажатием кнопки.

Листинг 6.9. Вызов диалогов и печать

```
private void printToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    prtdoc = new PrintDocument();
    prtdoc.PrintPage +=
       new PrintPageEventHandler(prtDocPrintPage);
    // создать PrintDialog,
    // основанный на PrintDocument
    PrintDialog pdlg = new PrintDialog();
    pdlg.Document = prtdoc;
    // показать PrintDialog
    if (pdlg.ShowDialog() == DialogResult.OK)
    {
       // создать PageSetupDialog,
       // основанный на PrintDocument и PrintDialog
       PageSetupDialog psd = new PageSetupDialog();
       // Получает или задает значение, показывающее
       // выполняется ли автоматическое
       // преобразование параметров полей из
       // миллиметров в сотые доли дюйма и обратно.
       psd.EnableMetric = true;
       psd.Document = pdlg.Document;
       // показать PageSetupDialog
       if (psd.ShowDialog() == DialogResult.OK)
       {
           // применить назначения обоих диалогов
           prtdoc.DefaultPageSettings =
               psd.PageSettings;
           // решить какое действие взять
           if (flPrint) // печать на бумаге
               prtdoc.Print();
           else         // preview на экране
           {
               PrintPreviewDialog prvw =
                   new PrintPreviewDialog();
               prvw.Document = prtdoc;
               prvw.ShowDialog();
           }
        }
    }
}
```

В этом методе:

1) создается объект *prtdoc* класса *PrintDocument*;

2) регистрируется обработчик событий *prtDocPrintPage* для вывода страницы;

3) создается и показывается *PrintDialog*, основанный на *PrintDocument*;

4) создается и показывается *PageSetupDialog*, основанный на *PrintDocument* и *PrintDialog*;

5) к объекту *prtdoc* применяется назначения обоих диалогов;

6) в зависимости от флага *flPrint* вызывается метод *prtdoc.Print()* для печати на бумаге или создается и показывается *PrintPreviewDialog*, основанный на prtdoc.

6.2. График функции двух переменных

Рассмотрим задачу построения поверхности, описываемой функцией *z = f(x,y)*, в параллелепипеде (*a1,a2*)*(*b1,b2*)*(*c1,c2*). Прежде всего приведем формулы суперпозиции двух поворотов системы координат относительно точки (*x0,y0,z0*). Сдвиг в точку (*x0,y0,z0*), поворот относительно оси *OZ* на угол α с последующим поворотом вокруг оси *OX* на угол β описывается соотношениями (6.1) и (6.2), соответственно:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x - x0 \\ y - y0 \\ z - z0 \end{pmatrix}; \qquad (6.1)$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta \\ 0 & \sin\beta & \cos\beta \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix}. \qquad (6.2)$$

После перемножения матриц в соотношениях (6.1) и (6.2) получаем

$$\begin{aligned} x &= (x-x0)*\cos\alpha - (y-y0)*\sin\alpha; \\ y &= ((x-x0)*\sin\alpha+(y-y0)*\cos\alpha)*\cos\alpha-(z-z0)*\sin\alpha; \\ z &= ((x-x0)*\sin\alpha+(y-y0)*\cos\alpha)*\sin\alpha+(z-z0)*\cos\alpha. \end{aligned} \qquad (6.3)$$

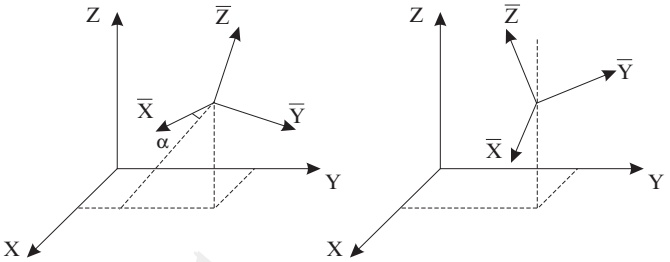

Рис. 6.2. Сдвиг и поворот системы координат

Будем считать, что оси `OX` и `OY` лежат в плоскости монитора, ось `OZ` перпендикулярна экрану. Будем сжимать координаты точки (`x,y`) к единственной точке схода (0,0) по формулам (6.4) и (6.5):

$$x_n = \frac{x}{z/a+1}; \tag{6.4}$$

$$y_n = \frac{y}{z/a+1}. \tag{6.5}$$

Параметр `A` подбирается экспериментально.

Для того чтобы не утяжелять листинги, мы и ранее в тексте и далее сознательно не используем возможности объектно ориентированного программирования, то есть не создаем классы. Но на примере этой простой задачи отступим от этого правила и в файле `Lib3dGraph.cs` создадим класс `TGraph3D`.

Тестирующая программа на экране монитора строит проекцию одной из шести поверхностей, например `z = 1+2*x*y-x-y` при (`x,y`) $\in$ [0,1] $\times$ $\times$ [0,1] (рис. 6.3).

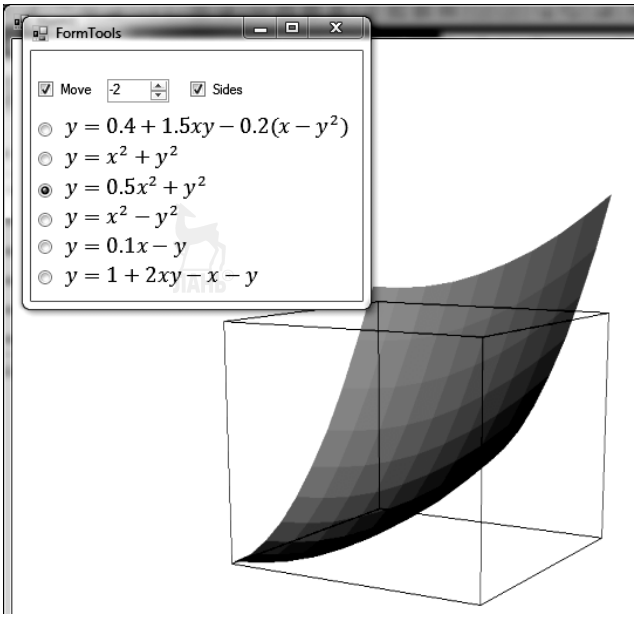

Рис. 6.3. Поверхность z=1+2*x*y-x-y в единичном кубе

На рисунке 6.4 представлена диаграмма классов проекта.

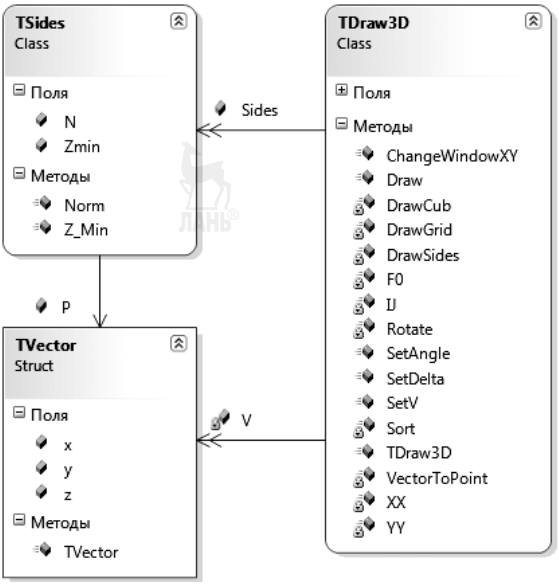

Рис. 6.4. Диаграмма классов

Основной класс *TGraph3D* наследуется от *object* и содержит следующие поля, методы и свойства.

Листинг 6.10. Класс TGraph3D

```
public class TDraw3D
{
    double Xmin;     // размеры окна на бумаге
    double Xmax;
    double Ymin;
    double Ymax;
    double fX1;      // куб в мировой СК
    double fX2;
    double fY1;
    double fY2;
    double fxc;      // центр вращения
    double fyc;
    double fzc;
    public double FA;
    int n;   число разбиений по X
    int m;   число разбиений по Y
    int I2;
    int J2;
    public bool fTypeFace=false;
    //true-Line,false-Face
```

```
    double Alf;
    double Bet;
    public TSides[] Sides;
    static TVector[,] V;
    public Bitmap bitmap;
    public SolidBrush myBrush;

    public TDraw3D(int VW,int VH)
    public F Func;
    public void SetV()
    private Point IJ(TVector P)
    private double XX(int I)
    private double YY(int J)
    private void DrawCub(Graphics g)
    void DrawGrid(Graphics g)
    TVector Rotate(TVector P)
    void Sort()
    Point VectorToPoint(TVector P)
    void DrawSides(Graphics g)
    public void Draw()
    public void SetAngle(double x, double y)
    public void ChangeWindowXY(int u, int v,
       int Delta)
    public void SetDelta(MouseEventArgs e0,
       MouseEventArgs e)
}
```

Используемые в классе типы описаны в следующем листинге.

Листинг 6.11. Вспомогательные типы

```
public delegate double F(double x, double y);

public struct TVector
{
    public double x, y, z;
    public TVector(double x, double y, double z)
}
public class TSides
{
    public TVector[] P = new TVector[4];
    public double N;
    public double Zmin;
    public double Z_Min()
    public double Norm()
}
```

Преобразование координат по формулам (6.3)–(6.5), одноточечное проецирование и масштабирование при переходе к экранным координатам описаны в главе 3 и реализованы в методе *IJ()*.

Листинг 6.12. Метод преобразования СК и масштабирования

```
private Point IJ(TVector P)
{
   double Xn=(P.x-fxc)*Math.Cos(Alf)-
       (P.y-fyc)*Math.Sin(Alf);
   double Yn = ((P.x-fxc)*Math.Sin(Alf)+(P.y-fyc)*
       Math.Cos(Alf))*Math.Cos(Bet)-
       (P.z-fzc)*Math.Sin(Bet);
   double Zn = ((P.x-fxc)*Math.Sin(Alf)+(P.y-fyc)*
       Math.Cos(Alf))*Math.Sin(Bet)+
       (P.z-fzc)*Math.Cos(Bet);
   Xn = Xn/(Zn*FA+1);
   Yn = Yn/(Zn*FA+1);
   int X = (int)Math.Round(I2*(Xn - Xmin) /
       (Xmax - Xmin));
   int Y = (int)Convert.ToInt32(J2*(Yn-Ymax)/
       (Ymin-Ymax));
   return new Point(X, Y);
}
```

Метод *SetV* использует указатель на пользовательский метод *Func* для задания массива узловых точек *V*.

Листинг 6.13. Метод вычисления элементов массива V

```
public void SetV()
{
    V = new TVector[n,m];
    double hx = (fX2-fX1)/(m-1);
    double hy = (fY2-fY1)/(n-1);
    for (int i=0; i <= n-1; i++)
    for (int j = 0; j <= m - 1; j++)
    {
        V[i, j].x = fX1 + j * hx;
        V[i, j].y = fY1 + i * hy;
        V[i, j].z = Func(fX1 + j*hx, fY1+i* hy);
    }
}
```

Область определения функции *z = f(x,y)* точками $x_i = x1+hx*i$, $y_j = y1+hy*j$, $i = 0,...,n$, $j = 0,1,...,m$ разбивается на *n*m* прямоугольников, над каждым из которых поверхность функции интерполируется ребрами четырехугольника. Построение спроектированных на экран четырех-

угольников реализуется с помощью методов *DrawLine* или *Polygon()* в методе *Draw3D()*.

Листинг 6.14. Рисование графика

```
public void Draw3D()
{
    I2 = bitmap.Width;
    J2 = bitmap.Height;
    using (Graphics g = Graphics.FromImage(bitmap))
    {
          Color cl = Color.FromArgb(255, 255, 255);
          g.Clear(cl);
          if (!fTypeFace)
              DrawGrid(g);
          else
              DrawSides(g);
          DrawCub(g);
    }
}
```

Листинг 6.15. Рисование графика ребрами

```
void DrawGrid(Graphics g)
{
    Point P0, P1;
    for (int i = 0; i <= n - 2; i++)
     for (int j = 0; j <= m - 2; j++)
     {
          P0 = IJ(V[i, j]);
          P1 = IJ(V[i, j + 1]);
          g.DrawLine(Pens.Black, P0, P1);
          P1 = IJ(V[i + 1, j]);
          g.DrawLine(Pens.Black, P0, P1);
     }
     for (int j = 0; j <= m - 2; j++)
     {
          P0 = IJ(V[n - 1, j]);
          P1 = IJ(V[n - 1, j + 1]);
          g.DrawLine(Pens.Black, P0, P1);
     }
     for (int i = 0; i <= n - 2; i++)
     {
          P0 = IJ(V[i, m - 1]);
          P1 = IJ(V[i + 1, m - 1]);
          g.DrawLine(Pens.Black, P0, P1);
     }
}
```

Листинг 6.16. Рисование графика гранями

```
void DrawSides(Graphics g)
{
    int L=(n-1)*(m-1);
    Sides = new TSides[L];
    for (int j = 0; j <= L - 1; j++)
          Sides[j] = new TSides();
    int q =-1;
    for (int j=0; j<=m-2; j++)
    for (int i=0; i<=n-2;i++)
    {
          q++;
          Sides[q].P[0]=Rotate(V[i+0,j+0]);
          Sides[q].P[1]=Rotate(V[i+1,j+0]);
          Sides[q].P[2]=Rotate(V[i+1,j+1]);
          Sides[q].P[3]=Rotate(V[i+0,j+1]);
          Sides[q].N = Sides[q].Norm();
          Sides[q].Zmin=Sides[q].Z_Min();
    }
    Sort();

    for (int i = 0; i <= L - 1; i++)
    {
          Point[] w = new Point[4];
          for (int j = 0; j <= 3; j++)
               w[j] = VectorToPoint(Sides[i].P[j]);
          byte R = 0;
          byte G = (255 * Math.Abs(Sides[i].N));
          byte B = (255 * Math.Abs(Sides[i].N));
          myBrush.Color = Color.FromArgb(R, G, B);
          g.FillPolygon(myBrush, w);
    }
}
```

В методе *Draw3D()* после очистки канвы рисуются оси повернутой системы координат и методом *Polygon()* выводятся фрагменты поверхности. Поверхность рисуется над единичным квадратом, стороны которого разбиты на *n* и *m* частей соответственно. Каждый фрагмент представляет собой четырехугольник, вершинами которого являются значения функции над узлами сетки в повернутой системе координат.

При создании класса вызывается конструктор *TGraph3D.Create()*:

Листинг 6.17. Инициализация данных

```
public TDraw3D(int VW,int VH)
{
    Xmin = -2; Xmax =  2; Ymin = -2; Ymax =  2;
```

```
    fX1  =  0; fX2  =  1; fY1  =  0; fY2  =  1;
    fxc  = 0.5; fyc  = 0.5; fzc  = 0.5;
    n    = 10; m    = 10;
    Alf  = 4.31; Bet  = 4.92;
    FA   = -0.2;
    fTypeFace = false;
    bitmap = new Bitmap(VW, VH);
    Func = F0;
    SetV();
    myBrush = new SolidBrush(Color.White);
}
```

в которой задается точка смещения начала координат (*xc = 0.5; yc = 0.5; zc = 0.5*), коэффициент перспективы (*FA = 0*), два угла поворота системы координат (*Alf = 4.31; Bet = 4.92*), габариты окна на бумаге (*Xmin = -2; Ymin = -2; Xmax = 2; Ymax = 2*), указатель на пользовательский метод *Func* и создается свой *Bitmap*.

Изменение многих свойств должно приводить к повторному вычислению элементов массива *V*. Поэтому в методах, отвечающих за изменение этих свойств, предусмотрен вызов метода *SetV()*. Так, например, для свойства *fX1* метод изменения выглядит так:

Листинг 6.18. Метод изменения свойства Graph3D.X1

```
public double X1
{
    get { return fX1; }
    set
    {
        if (fX1 != value)
        {
            fX1 = value; SetV();
        }
    }
}
```

При создании главной формы тестирующего проекта создадим экземпляр класса *Ob* типа *TGraph3D*, передавая ширину и высоту формы.

Листинг 6.19. Создание экземпляра класса TGraph3D

```
private void FormMain_Load(object sender, EventArgs e)
{
   MouseWheel += new MouseEventHandler(Form1_MouseWheel);
   g = Program.formMain.CreateGraphics();
   Ob = new TDraw3D(ClientRectangle.Width,
                    ClientRectangle.Height);
}
```

Рисование на форме проходит в два этапа: сначала методом рисуем на внутреннем *Bitmap* класса, а затем копирум этот *Bitmap* на канву формы или любого другого компонента.

Листинг 6.20. Рисование на форме

```
public static void MyDraw()
{
   Ob.Draw();
   g.DrawImage(Ob.bitmap,
     Program.formMain.ClientRectangle);
}
```

Перемещение мыши с нажатой клавишей по форме приводит к изменению углов *Alf* и *Bet*. Нажатие клавиши мыши вызывает метод *FormMain_MouseDown*, в котором поднимается флаг перемещения *drawing = true*.

Листинг 6.21. Начало поворота графика

```
private void FormMain_MouseDown(object sender,
    MouseEventArgs e)
{
    drawing = true;
    e0 = e;
}
```

При перемещении мыши по форме вызывается обработчик события.

Листинг 6.22. Поворот графика при перемещении мыши

```
private void FormMain_MouseMove(object sender,
    MouseEventArgs e)
{
    if (drawing)
    {
        if (!flMove)
        {
            double x = e.X-ClientRectangle.Width/2;
            double y = e.Y-ClientRectangle.Height/2;
            Ob.SetAngle(x, y);
        }
        else
        {
            Ob.SetDelta(e0, e); e0 = e;
        }
        MyDraw();
    }}
```

При перемещении курсора мыши углы *Alf* и *Bet* меняются по закону

```
a = x-Width / 2; b = y-Height / 2;
Alf = arctan(b/a);
Bet = Sqrt(Sqr(a/10)+Sqr(b/10));
```

то есть угол *Alf* является углом между линией, соединяющей точку положения мыши и центр формы, а угол *Bet* – расстоянием между точкой положения мыши и центром экрана. Этот алгоритм реализован в методе *SetAngle()*.

Листинг 6.23. Изменение углов системы координат

```
public void SetAngle(double x, double y)
{
   Alf = Math.Atan2(y, x);
   Bet = Math.Sqrt((x/10)*(x/10)+(y/10)*(y / 10));
}
```

Завершается поворот графика опусканием флага перемещения мыши.

Листинг 6.24. Завершение поворота графика

```
private void FormMain_MouseUp(object sender,
    MouseEventArgs e)
{
   drawing = false;
}
```

При изменении свойств класса с помощью компонентов инструментальной формы, например свойства *Graph3D.nX*, во многих случаях предусмотрено повторное изменение элементов массива *V*. Поэтому достаточно просто вызвать метод рисования формы *FormMain*.

Листинг 6.25. Изменение свойств с инструментальной формы

```
private void TFormTools3DGr_SpinEdit1Change(object sender)
{
  if (SpinEdit1.Text!="")
  {
    Graph3D.nX = SpinEdit1.Value;
    FormMain.MyDraw();
  }
}
```

6.3. Интерполяция функций

На практике часто встречается задача: по заданным на плоскости значениям (x_i,y_i), i=0,1,...,n построить функцию либо точно проходящую через эти точки, либо проходящую как можно ближе к этим точкам (рис. 6.5). Ниже рассмотрены четыре способа решения этой задачи: интерполяционный многочлен Лагранжа, метод наименьших квадратов, интерполяция кубическими сплайнами и интерполяция кривыми Безье.

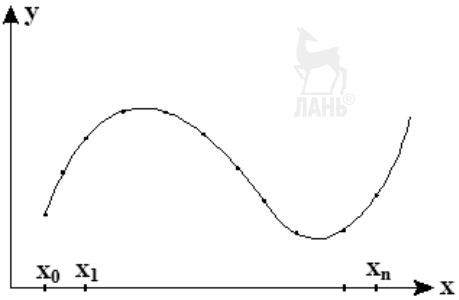

Рис. 6.5. Задача интерполяции

6.3.1. Проект для построения интерполяционных кривых

Проект предназначен для рисования линий по множеству заданных на плоскости точек с использованием четырех интерполяционных методов: интерполяционного многочлена Лагранжа, метода наименьших квадратов, интерполяции кубическими сплайнами и интерполяции кривыми Безье.

Проект состоит из двух форм: главной формы `FormMain` и формы для настройки параметров `FormTools` и модуля `Lbr.pas`. В проекте (рис. 6.6) реализованы следующие методы:

- выбор одного из методов;
- изменение масштаба (`Zoom`);
- перемещение точек;
- добавление узловых точек;
- удаление узловых точек.

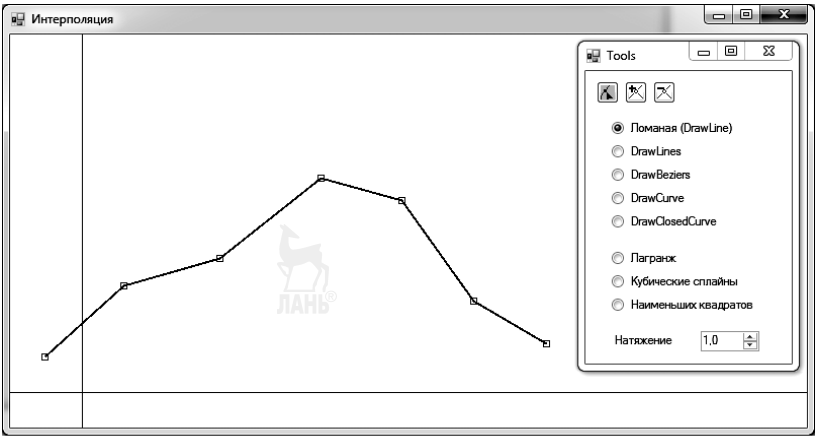

Рис. 6.6. Внешний вид формы проекта

Для описания узловых точек в проекте используется следующая структура:

```
struct TPoint2
{
    public double x, y;
    public TPoint2(double x, double y)
    {
        this.x = x; this.y = y;
    }
}
```

Сами узловые точки хранятся в коллекции *ArrayList points*. При запуске проекта создается массив длиной 7 элементов, и элементы этого массива заполняются случайными данными (листинг 6.26).

Листинг 6.26. Первоначальное заполнение массива

```
public FormMain()
{
    InitializeComponent();
    I2 = ClientSize.Width;
    J2 = ClientSize.Height;
    points = new ArrayList();
    Random rnd = new Random();
    for (int i = 1; i <= 7; i++)
        points.Add(new
            TPoint2(i,rnd.Next(100)/10));
    gScreen = CreateGraphics(); // Invalidated
}
```

Метод рисования определяется свойством *ItemIndex* компонента *RgMethod* на форме *FormTools*, которое может принимать следующие значения:

0: ломаная;
1: Лагранж;
2: наименьших квадратов;
3: кубические сплайны;
4: Безье.

Рисование линий осуществляется методом *Draw* на канве формы *FormMain*. В листинге 6.27 приведен основной фрагмент кода метода *Draw()* с реализацией рисования ломаной линии.

Листинг 6.27. Метод рисования линий

```
public void MyDraw()
{
    I2 = ClientRectangle.Width;
    J2 = ClientRectangle.Height;
```

```
    Bitmap bitmap =
       new Bitmap(ClientRectangle.Width,
       ClientRectangle.Height);
    using (Graphics g = Graphics.FromImage(bitmap))
    {
       g.FillRectangle(Brushes.White,
            ClientRectangle);
       g.DrawLine(Pens.Black,
            new Point(II(x1), JJ(0)),
            new Point(II(x2), JJ(0)));
       g.DrawLine(Pens.Black,
            new Point(II(0),JJ(y1)),
            new Point(II(0), JJ(y2)));
       double u, v, u0=0, v0=0;
       int L = points.Count;
       Point[] p = new Point[L];
       for (int i = 0; i <= L - 1; i++)
       {
           p[i].X = II(((TPoint2)points[i]).x);
           p[i].Y = JJ(((TPoint2)points[i]).y);
       }

       switch (flFunc)
       {
           case 0: // DrawLine
              for (int i = 1; i <= L - 1; i++)
                 g.DrawLine(Pens.Black,
                 p[i-1].X,p[i-1].Y,p[i].X,p[i].Y);
           break;
            остальные варианты
       }
       for (int i = 0; i < points.Count; i++)
           g.DrawRectangle(Pens.Black,
       p[i].X - 3, p[i].Y - 3, 6, 6);
       gScreen.DrawImage(bitmap, ClientRectangle);
    }
    bitmap.Dispose();
}
```

В проекте предусмотрена возможность перетаскивания любой точки мышью. Поэтому наряду с традиционными методами масштабирования, которые обсуждались в предыдущем пункте,

Листинг 6.28. Методы прямого масштабирования

```
static int II(double x)
{
    return I1+(int)((x-x1)* (I2 - I1) / (x2 - x1));
}
```

```
static int JJ(double y)
{
    return J2+(int)((y-y1)* (J1 - J2) / (y2 - y1));
}
```

используются методы обратного масштабирования

Листинг 6.29. Методы обратного масштабирования

```
static double XX(int I)
{
    return x1 + (I - I1) * (x2 - x1) / (I2 - I1);
}
static double YY(int J)
{
    return y1 + (J - J2) * (y2 - y1) / (J1 - J2);
}
```

и созданы обработчики трех событий *onMouseDown*, *onMouseMove*, *onMouseUp*. На инструментальной форме *FormTools* есть четыре кнопки, изменяющие переменную *fl_tools*. Эта переменная принимает значения 0...3: 0 – перемещение точек; 1 – добавление точек; 2 – удаление точек; 3 – изменение окна просмотра на плоскости. При *fl_tools* = 0 с помощью метода *FindPoint* определяется номер *NumPoint* точки, ближайшей к (*X, Y*), и поднимается флаг, разрешающий перемещения – *mydrawing* = true.

Листинг 6.30. Метод обработки нажатия клавиши мыши

```
private void Form1_MouseDown(object sender,MouseEventArgs e)
{
    e0 = e;
    switch (flTools)
    {
        case 0: // Move
           NumPoint = FindPoint(XX(e.X),YY(e.Y));
           mydrawing = true;
           break;
        case 1: // Add
           points.Add(new TPoint2(XX(e.X),
               YY(e.Y)));
           MyDraw();
           break;
        case 2: // Delete
           NumPoint = FindPoint(XX(e.X), YY(e.Y));
           if (NumPoint >= 0)
           {
               points.Remove(points[NumPoint]);
```

```
                MyDraw();
            }
            break;
        }
 }
```

Добавление точек: при *fl_tools* = 1 увеличивается на 1 длина динамического массива точек *Point* и добавленный элемент заполняется преобразованными координатами точки, в которой мы щелкнули мышью.

Удаление точек: при *fl_tools* = 2 с помощью метода *FindPoint* определяется номер *NumPoint* точки, ближайшей к (*X,Y*), и эта точка удаляется из массива *Point*. Отметим, что удаление точки требует пересчета коэффициентов для метода наименьших квадратов и кубических сплайнов.

Изменение окна просмотра: при *fl_tools* = 3 поднимается флаг, разрешающий перемещения, – *mydrawing* = *true*, и первый раз рисуется фокусный прямоугольник.

Во время перемещения мыши работает метод *FormMouseMove*, в котором при перемещении точки (*fl_tools* = 0) изменяются координаты выделенной точки, обновляются коэффициенты для метода наименьших квадратов и кубических сплайнов и методом *Draw()* перерисовывается весь график. При изменении окна просмотра стирается фокусный прямоугольник, меняются координаты его подвижного угла и фокусный прямоугольник рисуется в новом месте.

Листинг 6.31. Метод обработки перемещения мыши

```
private void Form1_MouseMove(object sender,MouseEventArgs e)
{
    if (mydrawing)
       switch (flTools)
       {
          case 0: // Move
             if ((Math.Abs(e.X-e0.X)>5) ||
                 (Math.Abs(e.Y-e0.Y)>5))
             {
                e0 = e;
                points[NumPoint] =
                   new TPoint2(XX(e.X), YY(e.Y));
                if (flFunc == 7) Gauss();
                MyDraw();
             };
             break;
        }
 }
```

Заканчивается движение мыши вызовом метода *FormMouseUp*, в котором опускается флаг перемещения *mydrawing* = *false*. Действие

предусмотрено только при изменении окна просмотра: меняются параметры окна просмотра (*X1*, *Y1*, *X2*, *Y2*).

Листинг 6.32. Метод обработки нажатия клавиши мыши

```
private void Form1_MouseUp(object sender, MouseEventArgs e)
{
     mydrawing = false;
}
```

6.3.2. Интерполяционный многочлен Лагранжа

График функции, определенной интерполяционным многочленом Лагранжа, проходит через все точки (x_i,y_i):

$$L(x) = \sum_{i=0}^{n} y_i \prod_{j=0, j<>i}^{n} \frac{x - x_j}{x_i - x_j}. \qquad (6.6)$$

Этот метод чрезвычайно прост в использовании, но обладает существенным недостатком: отклонение значений функции от ожидаемых может быть достаточно большим (рис. 6.9).

Для вычисления значений многочлена Лагранжа по уравнению (6.6) можно воспользоваться методом *Lagr*.

Листинг 6.33. Метод Лагранжа

```
double Lagr(double x)
{
    double p, s = 0;
    int L = points.Count;
    TPoint2[] P = new TPoint2[L];
    for (int i = 0; i <= L - 1; i++)
    {
        P[i].x = ((TPoint2)points[i]).x;
        P[i].y = ((TPoint2)points[i]).y;
    }
    for (int i = 0; i <= L - 1; i++)
    {
        p = 1;
        for (int j = 0; j <= L - 1; j++)
            if (i != j)
                p *= (x-P[j].x)/(P[i].x - P[j].x);
        s += p * P[i].y;
    }
    return s;
}
```

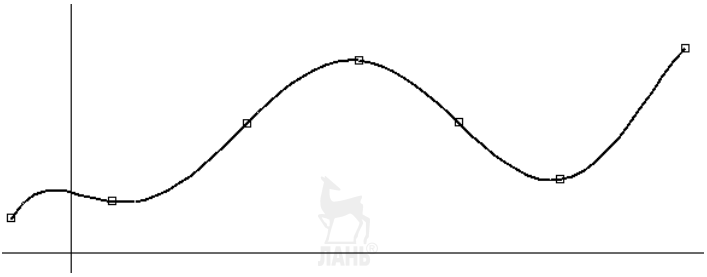

Рис. 6.7. Метод Лагранжа

Приведем фрагмент текста программы, использующей метод *Lagr* для построения графика по множеству точек (x_i, y_i), $i = 0,1,...,n$.

Листинг 6.34. Построение линии многочлена Лагранжа

```
case 1: // Lagrang
        double h = SetH();
        u = ((TPoint2)points[0]).x-h;
        for (int i = 0; i <= m; i++)
        {
           u += h; v = Lagr(u);
           if (i != 0)
           g.DrawLine(Pens.Black,
               II(u0),JJ(v0),II(u),JJ(v));
           u0 = u; v0 = v;
        }
        break;
```

Метод *SetH* определяет шаг изменения переменной *x* при разбиении интервала на *m* частей.

Листинг 6.35. Определение шага изменения по x

```
double SetH()
{
    int L = points.Count;
    double Xmin = ((TPoint2)points[0]).x;
    double Xmax = ((TPoint2)points[0]).x;
    for (int i = 0; i <= L - 1; i++)
    {
        if (Xmin > ((TPoint2)points[i]).x)
            Xmin = ((TPoint2)points[i]).x;
        if (Xmax < ((TPoint2)points[i]).x)
            Xmax = ((TPoint2)points[i]).x;
    }
    return (Xmax-Xmin)/m;
}
```

Дело в том, что первая и последняя точки не обязательно удовлетворяют условиям $x_{первая}=min(x_i)$, $x_{последняя}=max(x_i)$. Поэтому формула $H = (x_{последняя} - x_{первая})/m$ неприемлема.

6.3.3. Метод наименьших квадратов

Методом наименьших квадратов можно искать интерполяционную функцию $F(x)$ среди многочленов степени m:

$$F(x) \in \left\{ S(x) = \sum_{j=0}^{m} a_i x^j \right\} \tag{6.7}$$

так, чтобы сумма квадратов отклонений была минимальна:

$$D = \sum_{i=0}^{n} \left(\sum_{j=0}^{m} a_j x^j{}_i - y_i \right)^2 \to \min. \tag{6.8}$$

При этом функция $F(x)$ проходит через точки (x_i,y_i), если степень многочлена на единицу меньше числа точек.

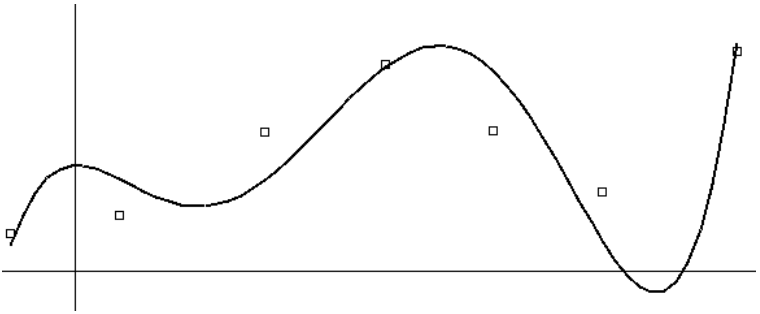

Рис. 6.8. Метод наименьших квадратов

Для определения коэффициентов a_j, $j = 0,...,m$ запишем необходимые условия экстремума:

$$\frac{\partial D}{\partial a_k} = 0, \quad k = 0,1,...,n \tag{6.9}$$

или

$$\sum_{i=0}^{n} x_i^k \left(\sum_{j=0}^{m} a_j x^j{}_i - y_i \right)^2 = 0, \quad k = 1,...,m. \tag{6.10}$$

После преобразований получаем следующую систему линейных уравнений:

$$\sum_{j=0}^{m} C_{kj} a_j = B_k, \quad k = 0,...,m, \tag{6.11}$$

где

$$C_{kj} = \sum_{i=0}^{n} x_i^{k+j}, \quad B_k = \sum_{i=0}^{n} x_i^{k} y_i. \tag{6.12}$$

Для вычисления элементов матрицы линейных уравнений по формулам (6.12) и решения этой системы методом Гаусса можно воспользоваться методом *Gauss*. Необходимо отметить, что система линейных уравнений для метода наименьших квадратов, как правило, хорошо обусловлена и не имеет нулевых элементов, а это позволяет максимально упростить алгоритм решения системы.

В методе *Gauss()* параметры имеют следующий смысл: *n+1* – число точек; *m* – степень многочлена; *x, y* – массивы узловых точек; *c* – массив коэффициентов многочлена.

Листинг 6.36. Метод Гаусса

```
public static void Gauss()
{
    int n = points.Count;
    TPoint2[] P = new TPoint2[n];
    for (int i = 0; i <= n - 1; i++)
    {
          P[i].x = ((TPoint2)points[i]).x;
          P[i].y = ((TPoint2)points[i]).y;
    }
    double[,] A=new double[CountStep+1,CountStep+1];
    double[] B = new double[CountStep + 1];
     // коэффициенты матрицы
    for (int j = 0; j <= CountStep; j++)
     for (int k = j; k <= CountStep; k++)
     {
          A[j, k] = 0;
          for (int i = 0; i <= n - 1; i++)
          {
               double p = 1;
               for (int l = 1; l < j + k; l++)
                    p *= P[i].x;
               A[j, k] += p;
          }
          A[k, j] = A[j, k];
     }
     //свободные члены
     for (int k = 0; k <= CountStep; k++)
     {
          B[k] = 0;
          for (int i = 0; i <= n - 1; i++)
```

```
            {
                  double p = P[i].y;
                  for (int l = 1; l < k; l++)
                        p *= P[i].x;
                  B[k] += p;
            }
      }
      //решение системы: прямой ход
      for (int i = 0; i <= CountStep - 1; i++)
      for (int j = i + 1; j <= CountStep; j++)
      {
            for (int k = i + 1; k <= CountStep; k++)
            A[k,j] += -A[i,j]*A[k,i]/A[i,i];
            B[j] += -B[i]*A[i,j]/A[i,i];
      }
      //решение системы: обратный ход
      for (int j = CountStep; j >= 0; j--)
      {
            C[j] = B[j];
            for (int k = j + 1; k <= CountStep; k++)
                  C[j] +=  - A[k, j] * C[k];
            C[j] /= A[j, j];
      }
}
```

Ниже приводится текст программы, в которой для массива (x_i, y_i), $i = 0,...,n$ методом наименьших квадратов определяются коэффициенты многочлена степени M и строится график этого многочлена.

Листинг 6.37. Фрагмент кода рисования линии для метода наменьших квадратов

```
case 7: // Gauss
    h = SetH();
    u = ((TPoint2)points[0]).x - h;
    for (int i = 0; i <= m; i++)
    {
        u += h; v = F2(u);
        if (i != 0)
            g.DrawLine(Pens.Black, II(u0),
                JJ(v0), II(u), JJ(v));
        u0 = u; v0 = v;
    }
    break;
```

Метод *F2()* вычисляет значение полинома по известным коэффициентам *C*.

Листин 6.38. Вычисление значения полинома

```
double F2(double x)
{
    int L = C.Length;
    double result = 0;
    for (int i = L-1; i>=0; i--)
        result = result*x+C[i];
    return result;
}
```

6.3.4. КУБИЧЕСКИЕ СПЛАЙНЫ

Сплайн-функция интерполирует значения (x_i, y_i) набором функций, каждая из которых определена на интервале $[x_i, y_i]$:

- функция $S(x)$ проходит через все точки (x_i, y_i): $S(x_i) = y_i$, $i = 0,...,n$;
- на всем интервале $[x_0, x_n]$ функция $S(x)$ дважды непрерывно дифференцируема.

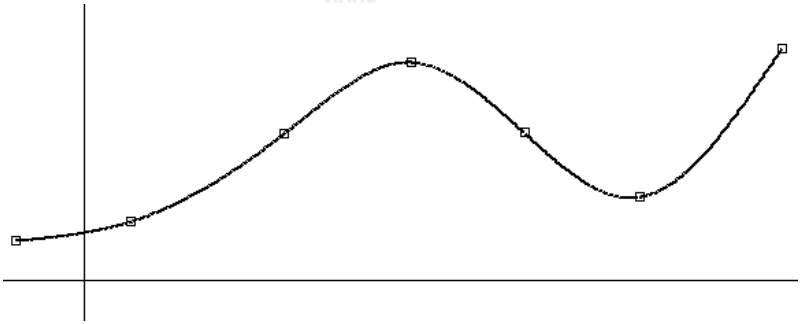

Рис. 6.9. Кубический сплайн

Будем считать, что на i-м интервале сплайн-функция определяется многочленом третьей степени

$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i, i = 1,\ldots,n. \quad (6.13)$$

Необходимо определить $4n$ коэффициентов a_i, b_i, c_i, d_i. Условия прохождения многочленов через узловые точки дают $2n$ уравнений

$$S_i(x_i) = d_i = y_i, \quad (6.14)$$

$$S_i(x_i) = a_i D_i^3 + b_i D_i^2 + c_i D_i + y_i = y_{i-1}, i = 1,\ldots,n, \quad (6.15)$$

где $D_i = x_{i-1} - x_{i-1}$.

Условие гладкости первых производных во внутренних точках дает n-1 соотношений $S_i^{'}(x_i) = S_{i+1}^{'}(x_i)$.

$$c_i = 3a_{i+1}D_{i+1}^2 + 2b_{i+1}D_{i+1} + c_{i+1},\ i = 1,\ldots,n-1. \tag{6.16}$$

Условие гладкости вторых производных дает еще n-1 соотношений

$$S_i^{''}(x_i) = S_{i+1}^{''}(x_i) \tag{6.17}$$

или

$$2b_i = 6a_{i+1}D_{i+1} + 2b_{i+1}, i = 1,\ldots,n-1. \tag{6.18}$$

Два недостающих условия следуют из условия обращения в ноль вторых производных на границах интервала $S_1^{''}(x_0) = 0$ и $S_n^{''}(x_n) = 0$:

$$6a_1D_1 + 2b_1 = 0,\ \ D_1 = x_0 - x_1, \tag{6.19}$$

$$b_n = 0. \tag{6.20}$$

Из соотношений (6.17) и (6.18) найдем a_i:

$$a_i = \begin{cases} \dfrac{b_{i-1} - b_i}{3D_i},\ i = 2,\ldots,n, \\ \dfrac{-b_1}{3D_1},\ \ i = 1. \end{cases} \tag{6.21}$$

Подставим a_i в соотношение (6.15):

$$\frac{(b_{i-1} - b_i)D_i^2}{3} + b_iD_i^2 + c_iD_i + y_i = y_{i+1},\quad i = 2,\ldots,n, \tag{6.22}$$

$$2b_1D_1^2 + c_1D_1 = y_0 - y_1,\ \ i = 1$$

и найдем c_i:

$$c_i = \begin{cases} \dfrac{y_{i-1} - y_i}{D_i} - D_ib_{i-1},\ i = 2,\ldots,n, \\ \dfrac{(y_0 - y_1)}{D_1} - b_1D_1/3,\ \ i = 1. \end{cases} \tag{6.23}$$

После исключения коэффициентов a_i и b_i из соотношения (6.15) и преобразования для коэффициентов b_i получаем трехдиагональную систему линейных уравнений

$$\frac{2*(D_1 + D_2)*b_1}{3} + \frac{b_2 * D_2}{3} = \frac{-(y_1 - y_2)}{D_2} + \frac{y_0 - y_1}{D_1}, \tag{6.24}$$

$$\frac{b_{i-1}D_i}{3}+\frac{2(D_i+D_{i+1})b_i}{3}+\frac{b_{i+1}D_{i+1}}{3}=-\frac{y_i-y_{i+1}}{D_{i+1}}+\frac{y_{i-1}-y_i}{D_i},\quad i=2,...,n-1,$$

$$b_n=0.$$

Рассмотрим теперь решение трехдиагональной системы линейных уравнений:

$$\begin{pmatrix} b_1 & c_1 & 0 & ... & 0 \\ a_2 & b_2 & c_2 & ... & 0 \\ 0 & a_3 & b_3 & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & b_n \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ ... \\ x_n \end{pmatrix}=\begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ ... \\ d_n \end{pmatrix}.$$

Трехдиагональная система решается в два прохода: за первый проход элементы матрицы под диагональю преобразуются в 0, за второй, обратный, ход определяются x_i, `i=1,...,n`. Для решения трехдиагональных систем можно воспользоваться методом `System_3D()`, который получает массивы `double[] a,b,c,d` и возвращает решение через массив `d`.

Листинг 6.39. Метод решения трехдиагональных систем

```
static void System_3D(double[] a, double[] b, double[] c,
      ref double[] d)
// решение трехдиагональных систем
{
    int L=a.Length-1;
    Array.Resize<double>(ref d,L+1);
    d = new double[L+1];
    // прямой ход
    for (int i=2; i<=L-1; i++)
    {
        b[i]=b[i]-c[i-1]*a[i]/b[i-1];
        d[i]=d[i]-d[i-1]*a[i]/b[i-1];
    }
    // обратный ход
    d[L-1]=d[L-1]/b[L-1];
    for (int i=L-2; i>=1; i--)
        d[i]=(d[i]-d[i+1]*c[i])/b[i];
}
```

Для определения коэффициентов многочленов (6.13) по формулам (6.24), (6.23), (6.21) и (6.15) можно воспользоваться методом `SetSystem3D`, который использует массивы координат узловых точек `x,y` и возвращает коэффициенты многочленов третьей степени `a,b,c,d`.

Листинг 6.40. Метод подготовки коэффициентов для системы

```
public static void SetSystem3D(out double[] a,
     out double[] b,out double[] c,out double[] d)
{
    object[] arr = points.ToArray();
    int L=arr.Length;
    TPoint2[] point = new TPoint2[L];
    for (int i=0; i<L; i++)
    {
           point[i].x=((TPoint2)arr[i]).x;
           point[i].y=((TPoint2)arr[i]).y;
    }
    double[] delt = new double[L];
    double[] w = new double[L];
    double[] q = new double[L];
    double[] p = new double[L];
    a = new double[L];
    b = new double[L];
    c = new double[L];
    d = new double[L];
    double[] r = new double[L];
    for (int i=1; i<=L-1; i++)
          delt[i]=point[i-1].x-point[i].x;
    //вычисление коэффициентов для системы уравнений
    for (int i=1; i<=L-2; i++)
    {
       w[i]=-(point[i].y-point[i+1].y)/delt[i+1]+
           (point[i-1].y-point[i].y)/delt[i];
       p[i]=delt[i]/3;
       q[i]=2*(delt[i]+delt[i+1])/3;
       r[i]=delt[i+1]/3;
    }
    // Решение трехдиагональной системы уравнений
    System_3D(p,q,r,ref w);
    for (int i=1; i<=L-1;i++) b[i]=w[i];
    b[L-1]=0;
    // Вычисление остальных коэффициентов многочлена
    c[1]=(point[0].y-point[1].y)/delt[1]-
          2*b[1]*delt[1]/3;
    a[1]=-b[1]/(3*delt[1]);  d[1]=point[1].y;
    for (int i=2; i<=L-1; i++)
    {
          d[i]=point[i].y;
          a[i]=(b[i-1]-b[i])/(3*delt[i]);
          c[i]=(point[i-1].y-point[i].y)/delt[i]-
              delt[i]*b[i-1]/3-2*delt[i]*b[i]/3;
    }
}
```

Ниже приводится фрагмент текста программы, в котором строится график функции по этим многочленам.

Листинг 6.41. Метод рисования

```
for (int i=1; i<=L-1; i++)
{
    v = ((TPoint2)points[i]).x;
    u = ((TPoint2)points[i-1]).x;
    h=(v-u)/m;
    u -= h;
    for (int j=0; j<=m; j++)
    {
      u+=h; v=F3(i,u);
      if (j != 0)
         g.DrawLine(Pens.Black, II(u0), JJ(v0),
             II(u), JJ(v));
      u0 = u; v0 = v;
    }
 }
```

Для вычисления значения многочлена на *i*-м интервале используется метод *F3(i,u)*.

Листинг 6.42. Вычисление значения многочлена на i-м интервале

```
double F3(int i, double u)
{
    double x = ((TPoint2)points[i]).x;
    return ((a[i] * (u-x) + b[i]) * (u-x) + c[i]) *
           (u-x) + d[i];
}
```

6.3.5. КРИВЫЕ БЕЗЬЕ

В следующем примере выполняется рисование кривой Безье.

Напомним, что в векторной параметрической форме уравнения, описывающие кривую Безье (рис. 6.10) по четырем точкам V_0, V_1, V_2, V_3, выглядят так:

$$X = V_0 t^3 + 3V_1 t^2 (1-t) + 3V_2 t (1-t)^2 + V_3 (1-t)^3. \qquad (6.25)$$

В этом уравнении параметр $t \in [0,1]$. Векторы $V_1 - V_0$ и $V_2 - V_3$ касаются кривой в точках V_0 и V_3. Для кривой, состоящей из нескольких фрагментов, должно выполняться условие: количество точек = $3N + 1$.

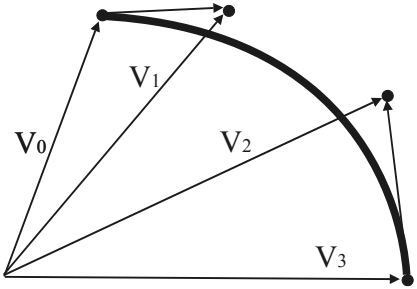

Рис. 6.10. Фрагмент кривой Безье

Для рисования можно пользоваться уравнениями (6.25), а можно использовать метод *DrawBeziers()*, который требует массива координат точек. Мы будем пользоваться методом *DrawBeziers*, и в этом случае фрагмент кода, рисующий линию, выглядит так:

Листинг 6.43. Рисование линии Безье

```
case 3: // DrawBeziers
   if (L % 3 == 1)
   {
     g.DrawBeziers(Pens.Black, p);
     for (int i = 0; i < L - 1; i = i + 3)
     {
       g.DrawLine(Pens.Black,
            p[i].X,p[i].Y,p[i+1].X,p[i+1].Y);
       g.DrawLine(Pens.Black,
            p[i+3].X,p[i+3].Y,p[i+2].X,p[i+2].Y);
     }
   }
   break;
```

На рисунке 6.11 приведен пример линии Безье, опирающейся на 7 точек.

Кроме линий Безье в C# есть методы:

- `DrawCurve(Pen pen, PointF[] points, float tension);`
- `DrawClosedCurve(Pen pen, Point[] points;`
- `float tension, FillMode fillmode),`

которые рисуют замкнутую (или незамкнутую) фундаментальную кривую, определяемую массивом структур `points` с указанным натяжением.

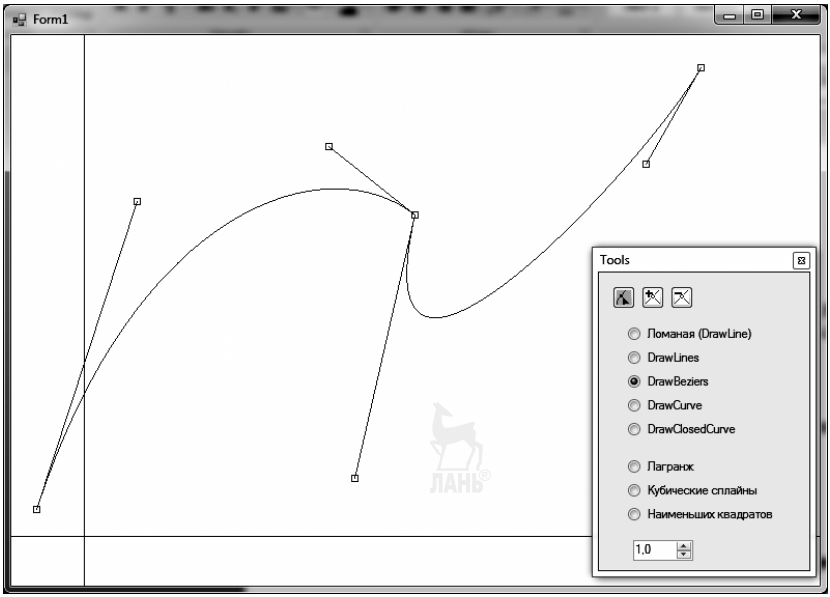

Рис. 6.11. Кривые Безье

Глава 7. **БИНАРНЫЕ ОПЕРАЦИИ**

Операции объединения, вычитания, пересечения 2D и 3D тел используются при проектировании в архитектуре и машиностроении. Эта задача является достаточно сложной. В пункте 2.5 описаны классы путей `GraphicsPath` и регионов `Region`, позволяющие выполнять эти операции в пространстве 2D. Однако эти классы не позволяют получить доступ к опорным точкам результирующего множества. Обзор известных алгоритмов для бинарных операций [44] для многоугольников показал, что они работают с ограничениями на вид многоугольников и не могут быть обобщены на пространство 3D.

Нумерация бинарных операций над множествами и введение индексов границ, описанная в этой главе, позволяет сформулировать общий алгоритм и для 2D- и для 3D-пространств [44].

Из элементов двух множеств A и B пространства M, используя операции объединения $\cup$, пересечения $\cap$ и разности $\setminus$, можно составить новое множество C (рис. 7.1).

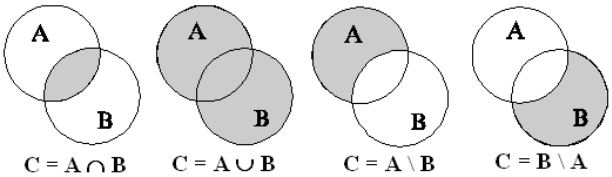

Рис. 7.1. Операции над множествами

В общем случае над двумя множествами A и B можно рассматривать шестнадцать различных операций.

7.1. ЛУЧЕВОЙ АЛГОРИТМ ОПРЕДЕЛЕНИЯ ПРИНАДЛЕЖНОСТИ ТОЧКИ

Для многоугольника вычисляется число пересечений луча, например горизонтального, проведенного из точки Q достаточно далеко за пределы многоугольника. Если это число чётное, то точка не принадлежит многоугольнику.

Основные проблемы возникают при прохождении луча Q через вершины: для многоугольника общего вида (рис. 7.2) в вершине может сходиться несколько ребер и ребра могут совпадать с лучом.

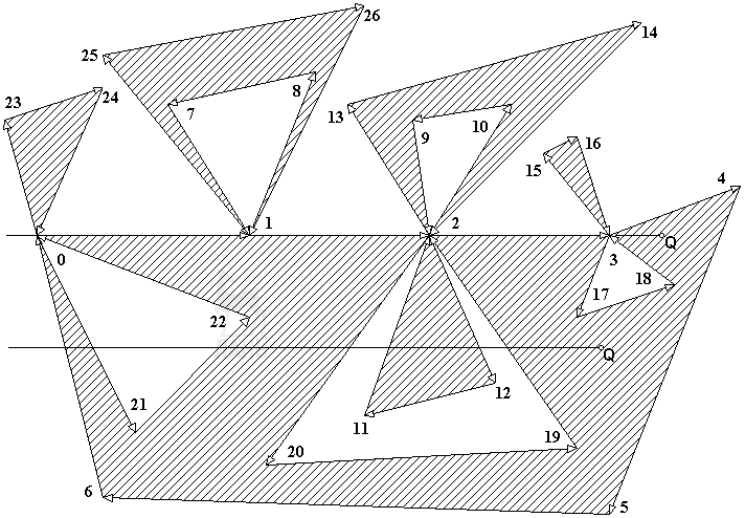

Рис. 7.2. Пример графа, удовлетворяющего условию Эйлера

Алгоритм предлагает в этом случае засчитывать пересечение луча только с концом ориентированного ребра. Важным аспектом является глобальная по всему контуру упорядоченность ребер, обеспечивающая то, что в вершине не могут сходиться концами два последовательных ребра.

Как уже утверждалось ранее, для многогранников глобальная упорядоченность ребер и граней, то есть существование замкнутых циклов по графу ребер и граней, невозможна. Поэтому необходимо использовать иной алгоритм, который будем называть инцидентным.

7.1.1. Инцидентный лучевой алгоритм для многоугольников

У полигонов, ограничивающих многоугольники, для любой вершины степень положительна и четна. Для алгоритмов принадлежности точки особую сложность представляют вершины, степень которых больше 2. Для дальнейшего анализа введем понятие множество смежных вершин. В плоском случае через вершину v может проходить несколько фрагментов контура. Каждый фрагмент может содержать только два ребра E_0 и E_1 с нормалями N_0 и N_1, у которых внешние нормали не меняют ориентацию, то есть скалярное произведение векторных произведений должно быть больше нуля:

$$(E_0 \times N_0,\ E_1 \times N_1) > 0. \qquad (7.1)$$

Определение. Два инцидентных ребра $e_1=(v,v')$ и $e_2=(v,v'')$ вершины v образуют *семейство инцидентных ребер*, если их внешние нормали не меняют знак.

Таким образом, для вершины $v = \langle X;\ V_s,\ E_v,\ F_v,\ \overline{N} \rangle$ множество инцидентных ребер E_v разбивается на n семейств инцидентных ребер $E_{v,n}$, не меняющих направление внешней нормали:

$$E_v = \cup E_{v,n}.$$

Две смежные вершины, одна из которых является началом инцидентного ребра e_1, а вторая – концом ребра e_2, образуют *семейство смежных вершин*.

У вершины v_0, изображенной на рисунках 7.3а, б, существует 6 смежных вершин, которые могут входить в три семейства.

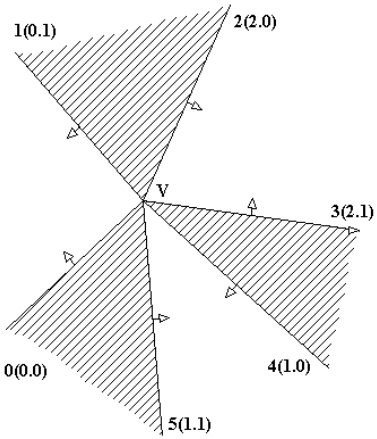

Рис. 7.3а

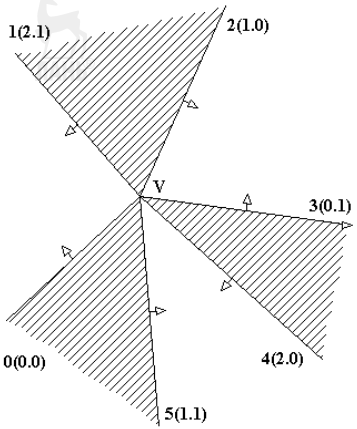

Рис. 7.3б

В случае, показанном на рисунке 7.3а, это семейства [0(0,0), 1(0,1)], [4(1,0), 5(1,1)], [2(2.0), 3(2,1)]. В случае, показанном на рисунке 7.3б, это семейства [0(0.0), 3(0,1)], [2(1.0), 5(1,1)], [4(2.0), 1(2,1)]. Первое число после номера вершины обозначает номер семейства, второе – номер вершины в семействе. Всего может быть шесть различных вариантов семейств. На рисунках 7.3а, б представлены только два из них. Недопустимы варианты, в которых происходит смена направления нормали, например вершины 1 и 5 не могут быть смежными в одном семействе.

Лучевой алгоритм определения принадлежности в плоском случае сводится к следующим шагам.

1. Находим те вершины, которые попадают на луч.
   a) Для каждой такой вершины строим семейство инцидентных вершин.
   b) Для каждого семейства определяем, принадлежит ли вершина интервалу проекций инцидентных вершин на ось Y. Если принадлежит, то число пересечений увеличиваем на 1.
2. Для всех ребер без вершин считаем число пересечений.

Таким образом, задача определения принадлежности свелась от двумерной к одномерной – к определению принадлежности точки интервалу.

7.1.2. ER-модель

Для описания многоугольника или триангулированной поверхности будем использовать модель «сущность – связь» (ER-модель).

Рассмотрим многоугольники общего вида, которые могут быть определены тремя сущностями: многоугольники (`polygons`), ребра (`edges`) и вершины (`vertex`). В общем случае эти сущности связаны между собой тремя отношениями «многие ко многим» (многоугольники – ребра, многоугольники – вершины, ребра – вершины) и тремя отношениями «один ко многим» (многоугольник – инцидентные многоугольники, ребро – инцидентные ребра, вершина – инцидентные вершины). Но для алгоритма принадлежности достаточно оставить пять связей «один ко многим» (рис. 7.4).

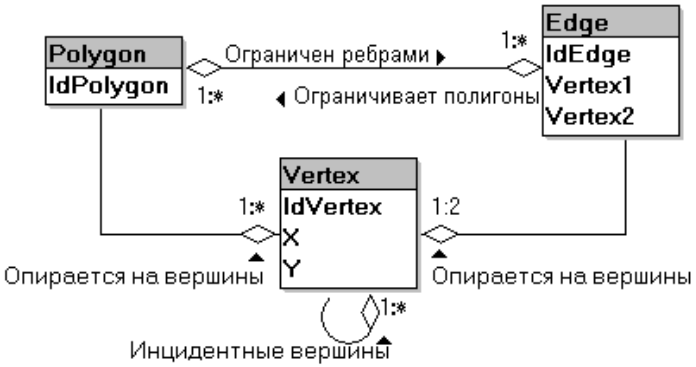

Рис. 7.4. ER-модель для многоугольников общего вида

Обращаем внимание на то, что для реализации алгоритма для каждой вершины потребуется знать семейства ее соседей, то есть необходимо оставить связь «вершина – инцидентные вершины».

Структура сущности `Vertex` помимо координат `x,y` содержит поле `Family` – двумерный массив семейств номеров инцидентных вершин и поле `Visit` – признак посещения вершины, используемый при реализации алгоритма принадлежности. Этот признак запрещает обрабатывать вершину дважды.

Структура ребер `Edge` содержит массив `IdVertex` номеров вершин, на который она опирается, и массив `IdPolygon` номеров многоугольников, которому ребро принадлежит. Кроме этого, у ребра есть поле `Norm` – вектор внешней нормали.

Наконец, структура многоугольников *Polygon* содержит массив *IdVertex* номеров вершин, на который она опирается, и массив *IdEdge* номеров ребер, составляющих контур многоугольника.

7.1.3. Детализация алгоритма для многоугольников

Введем бит принадлежности точки внутренней области *Parity*, принимающий значения 0 и 1 и меняющий значение при входе в область и при выходе из нее.

Алгоритм состоит из двух шагов.

1. Находим те вершины, которые попадают на луч.
   a) Для каждой такой вершины строим семейство инцидентных вершин.
   b) Для каждого семейства определяем, принадлежит ли вершина интервалу проекций инцидентных вершин на ось *Y*. Если принадлежит, то меняем бит принадлежности.
2. Для всех ребер без вершин считаем число пересечений.

Второй шаг проблем не вызывает. Рассмотрим более подробно шаг 1.1 плоского алгоритма. На рисунке 7.2 представлен ориентированный граф из 27 узлов, для которого существует эйлеров цикл: 0, 1, 8, 7, 1, 25, 26, 1, 2, 12, 11, 2, 10, 9, 2, 20, 19, 2, 13, 14, 2, 3, 17, 18, 3, 15, 16, 3, 4, 5, 6, 0, 23, 24, 0, 21, 22, 0. Таких циклов существует достаточно много. Так, например, для вершины с номером 1 существует три фрагмента путей по вершинам (0, 1 ,8; 7, 1, 25; 26, 1, 2), но возможен путь по вершинам (7, 1, 25; 26, 1, 8; 0, 1, 2). Для вершины с номером 2 существует пять фрагментов путей (1, 2, 12; 11, 2, 10; 9, 2, 20; 19, 2, 13; 14, 2, 3).

Впрочем, построение эйлерова цикла не обязательно для построения семейств инцидентных вершин.

1. Можно, например, упорядочить инцидентные ребра вершины по углу и каждую последовательную пару соответствующих инцидентных вершин считать семейством. Такой подход требует выполнения тригонометрических вычислений с неизбежной потерей точности и замедляющих работу алгоритма. Поэтому мы его рассматривать не будем.
2. Второй локальный способ сводится к последовательной сборке инцидентных вершин в пары, удовлетворяющие условию 1.
3. Еще один локальный способ формирования семейств будет предложен ниже при обсуждении алгоритмов для многогранников.

Тем не менее вернемся к проблеме глобальной упорядоченности ребер, то есть к алгоритму нахождения эйлерова цикла. Для реализации алгоритма нахождения эйлерова цикла потребуется рабочий стек *Stack* и еще один стек для эйлерова пути *Path*. Напомним алгоритм определения эйлерова пути.

1. Очистить признаки посещения узлов.
2. Начать движение из любого узла, например с номером $v = 0$, поместив этот номер в стек *Stack*.
3. Если стек *Stack* не пуст, то для текущего узла найти ребро, соединяющее с инцидентным узлом, по которому еще не ходили. Иначе происходит выход из алгоритма.
4. Если такое ребро найдено, то поместить в стек *Stack* инцидентный узел, пометить то, что ребро пройдено, и перейти на шаг 3. Иначе выполнить шаг 5.
5. Если ребро не найдено, то взять номер узла из стека *Stack*, то есть вернуться назад, поместить номер узла в стек пути *Path* и перейти на шаг 3.

Результат работы алгоритма накапливается в стеке *Path*:

Path = (0,8,7,1,25,26,1,1,12,11,2,10,9,2,20,19,2,13,14,2,2,17,18,3,15,16,3,3,4,5,6,0,23,24,0,21,22,0).

Однако для ветвящихся контуров типа, представленного на рис. 7.2, стек *Path* не совсем точно отражает путь: во-первых, сильно ветвящиеся вершины, например 1 или 2 или 3, несколько раз подряд попадают в стек; во-вторых, рядом оказываются вершины, не соединенные ребрами. Для корректировки пути введем еще два стека: *PathTemp* – вспомогательный и *PathDouble* – для повторяющихся вершин. Алгоритм состоит из двух проходов. На первом проходе собираем повторяющиеся вершины в стек *PathDouble*, на втором – вставляем из стека *PathDouble* вершины, между которыми не существует ребра. Во время первого прохода стек *Path* разрушается, поэтому приходится создавать вспомогательный стек *PathTemp*. Первый проход состоит из следующих шагов.

1. Взять номер вершины из стека *Path* в переменную *Old*.
2. Если стек *Path* не пуст, взять номер вершины в переменную *v*. Иначе выход.
3. Если номера вершин *v* и *Old* совпадают, то положить *v* в стек *PathDouble*.
4. Иначе положить *v* в стек *PathTemp*.
5. Назначить *Old* ← v и перейти к шагу 2.

Второй проход работает со стеком *PathTemp* и состоит из следующих шагов.

1. Взять номер вершины из стека *PathTemp* в переменную *Old*.
2. Положить в стек *Path*.
3. Если стек *PathTemp* не пуст, взять номер вершины в переменную *v*. Иначе выход.
4. Найти ребро, у которого первая вершина *Old*, а вторая – *v*.

5. Если такое ребро найдено, то положить v в стек *Path*. *Old* $\leftarrow v$. Перейти к шагу 3. Иначе взять из стека *PathDouble* номер вершины *Old*, положить его в стек *Path* и перейти к шагу 4.

В результате работы алгоритма стек *Path* содержит один из возможных эйлеровых путей, который, например, для графа, изображенного на рисунке 7.4, содержит следующие вершины:

```
Path=(0,1,8,7,1,25,26,1,2,12,11,2,10,9,2,20,19,2,13,14,2,3,
17,18,3,15,16,3,4,5,6,0,23,24,0,21,22,0).
```

Получение семейств инцидентных вершин из пути не составляет труда: необходимо для каждой вершины правого и левого соседа поместить в новое семейство.

На шаге 1:2 алгоритма для каждого семейства определяем, принадлежит ли проекция вершины P_y на ось Y открытому интервалу проекций инцидентных вершин (v_y,w_y) одного семейства на ось Y. Если принадлежит, то число пересечений увеличиваем на 1. Возможны пять вариантов принятия решения (рис. 7.5–7.7):

1. $P_y \notin (v_y,w_y)$ – принимается решение: нет смены бита принадлежности Parity.
2. $P_y \in (v_y,w_y)$ – принимается решение: смена бита принадлежности *Parity*.
3. Одна инцидентная вершина находится перед вершиной на луче – отложенное решение. В переменную *sign* запоминаем знак проекции на ось *Y* другой вершины.
4. Одна смежная вершина находится за вершиной на луче – принимается решение с учетом знака *sign* отложенного решения (рис. 7.6, 7.7).
5. Обе инцидентные вершины находятся на луче – нет смены бита принадлежности.

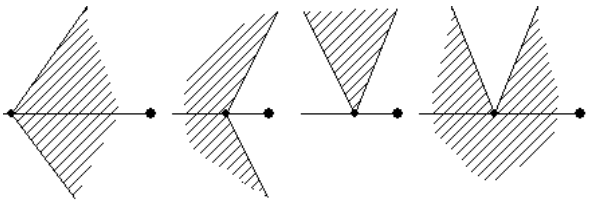

Рис. 7.5. Принимается решение: есть смена бита или нет

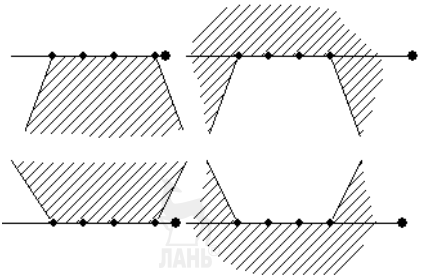

Рис. 7.6. Непринятые отложенные решения

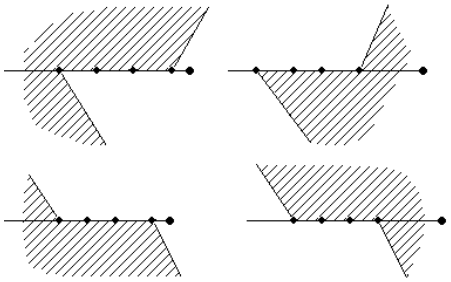

Рис. 7.7. Принятые отложенные решения

Бит принадлежности меняется или в случае 2, или, может быть, в случае 4. Отметим, что для корректного анализа отложенных решений необходима упорядоченность вершин на луче справа налево.

7.2. Булевы операции над множествами

Эта глава самая короткая, но очень важная для оверлейных алгоритмов. В ней вводится понятие индексов элементов множеств, индексов границ множеств и нумерация всех 16 булевых операций связывается с правилами вычисления их результата, формулируется и доказывается теорема об индексах границ результирующего множества.

7.2.1. Индексы множеств и границ

Введем понятия индексов точек множеств и индексов границ [42–44]. Из элементов двух множеств A и B пространства M, используя булевы операции объединения $\cup$, пересечения $\cap$, разности $\setminus$ и т.д., можно составить новое множество C (рис. 7.8).

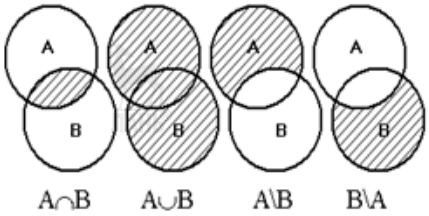

Рис. 7.8. Операции над множествами

Без ограничения общности можно считать, что $A \cap B \neq \varnothing$, то есть существуют x такие, что $(x \in A) \wedge (x \in B)$. Каждому элементу $x \in M$ поставим в соответствие число (рис. 7.9). Пронумеруем непересекающиеся части пространства M, в котором определены множества A и B: $A \cap B$ – 0; $A \setminus B$ – 1; $B \setminus A$ – 2; $M \setminus (A \cup B)$ – 3.

Определение 1. Числа 0, 1, 2, 3 будем называть *индексами* (*Index*) внутренних элементов множеств $A \cap B$, $A \setminus B$, $B \setminus A$ и $M \setminus (A \cup B)$:

$$\begin{aligned} &Index{=}0\text{: } A \cap B = \{x\text{: } x \in A \text{ и } x \in B\};\\ &Index{=}1\text{: } A \setminus B = \{x\text{: } x \in A \text{ и } x \notin B\};\\ &Index{=}2\text{: } B \setminus A = \{x\text{: } x \in B \text{ и } x \notin A\};\\ &Index{=}3\text{: } \neg A \cap \neg B = \{x\text{: } x \notin A \text{ и } x \notin B\}. \end{aligned} \qquad (7.2)$$

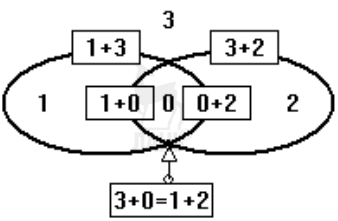

Рис. 7.9. Индексы подмножеств и границ при разбиении

Далее вместо термина индекс внутреннего элемента множества будем употреблять термин «индекс множества».

Понятие индекса тесно связано со значениями битов в двоичном представлении числа. Используя операции битовых сдвигов налево *shl* и направо *shr*, введем операцию $Bit_k(n)$ вычисления k-го бита числа n:

$$Bit_k(n) = (n \text{ and } (1 \text{ shl } k)) \text{ shr } k. \qquad (7.3)$$

Определение 2. Индексы множества A равны номерам ненулевых битов в двоичном числе 0011_2, а индексы множества B равны номерам ненулевых битов в числе 0101_2:

$$Index(A) = \{k \mid Bit_k(0011_2) \neq 0\}; \quad (7.4)$$

$$Index(B) = \{k \mid Bit_k(0101_2) \neq 0\}. \quad (7.5)$$

Определение 3. Каждому граничному элементу $\partial A \cup \partial B$ множеств A и B поставим в соответствие число, которое будем называть *индексом границы*. Индекс границы $\partial(A_1,A_2)$ равен сумме индексов непересекающихся множеств A_1 и A_2, которые она разделяет:

$$Index(\partial(A_1,A_2)) = Index(A_1) + Index(A_2). \quad (7.6)$$

На рисунке 7.9: 1+0 = 1 – индекс граничной точки между множествами с индексами 0 и 1; 2+0 = 2 – индекс граничных точек между множествами с индексами 0 и 2; 3+0=1+2=3 – индекс узловых точек; 3+1 = 4 – индекс точки между множествами с индексами 1 и 3; 3+2 = 5 – индекс граничной точки между множествами с индексами 3 и 2.

В любой узловой точке – точке пересечения границ – сходятся две или четыре (например, для операции исключающее или с номером $6_{10} = 0110_2$) границы нового множества C.

Операции объединения, вычитания, пересечения 2D и 3D тел являются достаточно сложными. Однако бинарная нумерация операций над множествами позволяет сформулировать общий алгоритм сбора ребер и граней в результирующее тело, опирающийся на теорему об индексах [42–44].

7.2.2. НУМЕРАЦИЯ БУЛЕВЫХ ОПЕРАЦИЙ

В общем случае над двумя множествами A и B можно рассматривать шестнадцать различных операций, каждая из которых имеет номер от 0 до 15. Для дальнейших рассуждений будем использовать двоичную нумерацию этих операций (табл. 7.1). Двоичный номер операции полностью определяет ее результат. Так, например, операция с номером $7_{10}=0111_2$ – это операция объединения «или».

Пусть индекс элемента $x \in M$ соответствует номеру бита в двоичном представлении номера операции (табл. 7.1). В этой же таблице приведены логические операции, соответствующие множественным операциям.

Таблица 7.1

Булевы операции над множествами

N_2	Логика	Множества	Логическая операция
0000	$\forall$A,B: FALSE	$\varnothing$	
0001	$A \wedge B$	$A \cap B$	Конъюнкция, пересечение
0010	$A \wedge \neg B$	$A \setminus B$	Разность
0011	A	A	A
0100	$\neg (A \rightarrow B)$	$B \setminus A$	Разность
0101	B	B	B
0110	$\neg (A \Leftrightarrow B)$	$(A \setminus B) \cup (B \setminus A)$	исключающее «или», симметричная разность

Продолжение табл. 7.1

N_2	Логика	Множества	Логическая операция
0111	A ∨ B	A ∪ B	Дизъюнкция или объединение
1000	¬A ∧ ¬B	(M\A) ∩ (M\B)	стрелка Пирса
1001	A ⇔ B	¬(A\B)∪¬(A\B)	эквивалентность
1010	¬B	M\B	не B
1011	A → B	A ∪ (M\B)	импликация, A влечет B
1100	¬A	M\A	не A
1101	B → A	(M\A) ∪ B	импликация, B влечет A
1110	¬A ∨ ¬B	(M\A) ∪ (M\B)	штрих Шеффера
1111	∀A,B: TRUE	M	

Так, например, 1 в нулевом бите номера операции пересечения $C = A \cap B$ указывает на то, что в множество C входят только элементы множества с индексом 0. Операция с номером $3_{10} = 0011_2$ соответствует множеству A, операция с номером $5_{10} = 0101_2$ соответствует множеству B, операция с номером $10_{10} = 1010_2$ соответствует множеству $\neg B$, операция с номером $12_{10} = 1100_2$ соответствует множеству $\neg A$.

Ясно, что практический интерес представляют прежде всего операции с номерами 1, 2, 4, 6, 7: $A \cap B$, $A \setminus B$, $B \setminus A$, $(A \setminus B) \cup (B \setminus A)$, $A \cup B$.

> **Замечание 1.** Восемь операций с номерами от $8_{10}=1000_2$ до $15_{10}=1111_2$ операцией отрицания сводятся к операциям с номерами от $0_{10}=0000_2$ до $7_{10}=0111_2$, а эти восемь операций при помощи формул Моргана могут быть сведены только к двум: например, к операции отрицания и операции «и».

Для всех возможных шестнадцати логических операций введем обозначения: $A \circ_i B$, $i = 0,1,...,15$. В частности, для наиболее употребительных операций имеем такие обозначения:

$A \circ_1 B = A \wedge B$; $A \circ_7 B = A \vee B$; $A \circ_9 B = A \Leftrightarrow B$; $A \circ_{11} B = A \rightarrow B$; $A \circ_{12} B = \neg B$; $A \circ_{10} B = \neg A$.

При перестановке аргументов в логических операциях значение операции не меняется, если изменить номер операции. В частности, для бинарных операций $A \circ_i B$ после перестановки A и B можно использовать операцию $B \circ_j A$. Номер j получается из номера i перестановкой первого и второго битов.

> **Замечание 2**. В четкой логике по номеру операции n и значениям A и B, которые могут принимать значения 0 или 1, можно вычислить результат. Для этого необходимо с помощью A и B вычислить номер бита k=(1-A)+2×(1-B) и с помощью сдвиговых операций (2) вычислить значение бита в номере операции: $A \circ_n B = Bit_k(n)$.

7.2.3. Признак принадлежности результату булевой операции

Между номерами операций и индексами граничных элементов результирующего множества есть связь.

Теорема об индексах [42–44]. Сумма номеров пар различных битов в двоичном представлении номера операции совпадает с индексами граничных элементов результирующего множества:

$$Index(\partial(A \circ_n B)) = \{k_1+k_2 \mid Bitk_1(n) \neq Bitk_2(n)\}. \quad (7.7)$$

Доказательство сводится к проверке утверждения для всех 16-ти бинарных операций. Так, например, для операции исключающее «или» $C = (A \setminus B) \cup (B \setminus A)$ с номером $6_{10} = 0110_2$ граница состоит из элементов с номерами: 0+1, 2+3, 1+3, 2+3, 3+0 = 1+2. В двоичном числе 0110_2 различны следующие пары битов: (0,1), (0,2), (2,3), (1,3). Элементы пересечения границ (0+3 = 1+2) входят в границы любого нового множества. Для операции пересечения $A \cap B$ с номером $1_{10}=0001_2$ граница множества C состоит из элементов с индексами 0+1, 0+2, 0+3 = 1+2. Эти же пары различных битов есть в числе 0001_2: (0,1), (0,2), (0,3).

> **Замечание 3**. Порядок нумерации множеств A∩B, A\B, B\A и M\(A∪B) влияет на порядок нумерации булевых операций.

Под выполнением булевой операции над многоугольниками A и B понимается выполнение операции над ограниченными ими областями и получение множества многоугольников, описывающего полученную область.

В качестве исходных данных здесь могут быть произвольные полигональные области любого класса, в частности, многосвязные области с отверстиями и неограниченной кратностью вершин.

Глава 8. **ПЛАТОНОВЫ ТЕЛА**

Простейшими трехмерными объектами являются платоновы тела – правильные выпуклые многогранники, все грани которых – правильные многоугольники и все углы при вершинах равны между собой.

Существует всего пять правильных многогранников. Этот факт впервые был доказан Евклидом. Приведем его рассуждения. Пусть к каждой вершине правильного многогранника примыкает m граней и каждая из них является правильным n-угольником. Внутренний угол у каждой грани равен

$$\varphi_n = \frac{\pi(n-2)}{n}, \tag{8.1}$$

а сумма внутренних углов, примыкающих к вершине, равна

$$m\varphi_n = m\frac{\pi(n-2)}{n}. \tag{8.2}$$

Поскольку телесный угол при вершине не является плоским, то отсюда следует неравенство

$$m\varphi_n < 2\pi. \tag{8.3}$$

В результате получаем систему неравенств:

$$\begin{aligned} &m(n-2) < 2n, \\ &m \geq 3, \\ &n \geq 3. \end{aligned} \tag{8.4}$$

Эта система имеет пять целочисленных решений, соответствующих многогранникам, представленным на рисунках 8.1–8.5.

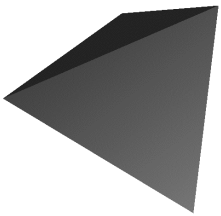

Рис. 8.1. Тетраэдр (3,3)

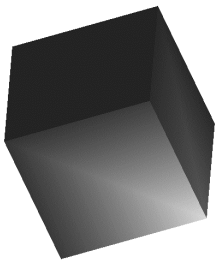

Рис. 8.2. Куб (4,3)

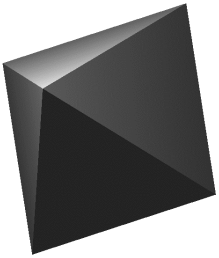

Рис. 8.3. Октаэдр

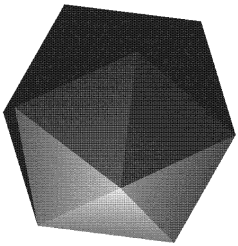

Рис. 8.4. Икосаэдр

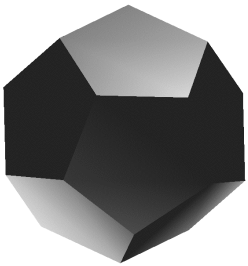

Рис. 8.5. Додекаэдр

8.1. Построение платоновых тел

Посмотрим, как можно построить некоторые платоновы тела в компьютерной графике.

8.1.1. Тетраэдр

Хотя тетраэдр имеет всего четыре грани, каждая из которых представлена в виде правильных треугольников, вычерчивание его трехмерной проекции – непростая задача. Один из способов построения тетраэдра предполагает использования куба в качестве вспомогательного объекта, как это показано на рисунке 8.6. Сначала рисуется куб, на всех гранях проводятся диагонали, а затем ребра куба стираются.

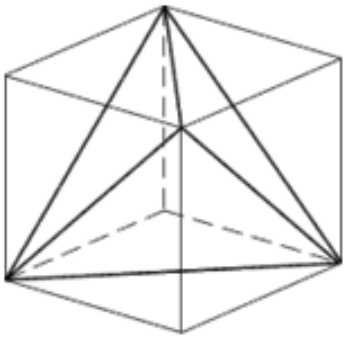

Рис. 8.6. Построение тетраэдра

8.1.2. Октаэдр

На рисунке 8.7 изображен октаэдр. Все грани октаэдра являются равнобедренными треугольниками.

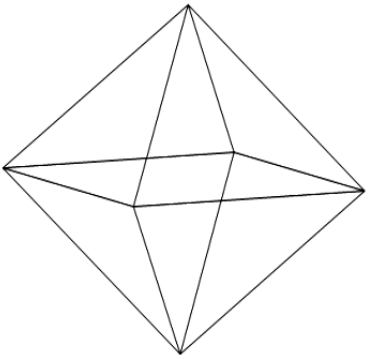

Рис. 8.7. Октаэдр

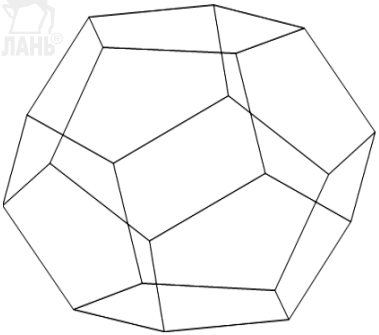

Рис. 8.8. Додекаэдр

8.1.3. Додекаэдр

Додекаэдр имеет двенадцать граней, тридцать ребер, двадцать вершин. Каждая из двенадцати граней является правильным пятиугольником. Додекаэдр вписывается в куб (рис. 8.8), и это его свойство можно использовать для конструирования.

8.2. Проект для построения платоновых тел

Проект «Платоновы тела» предназначен для рисования всех пяти платоновых тел: тетраэдра, гексаэдра, октаэдра, икосаэдра и додекаэдра и состоит из двух форм: главной формы *FormMain* и формы для настройки параметров *FormTools*. В проекте (рис. 8.9) реализованы следующие функции:

1) выбор одного из тел;
2) вращение системы координат или тела;
3) рисование тела гранями или ребрами;
4) изменение размера тела;
5) изменение масштаба (*Zoom*);
6) окрашивание граней полутонами;
7) рисование тени на плоскости *z = const*;
8) изменение положения точек схода на осях *OZ* и *OY*.

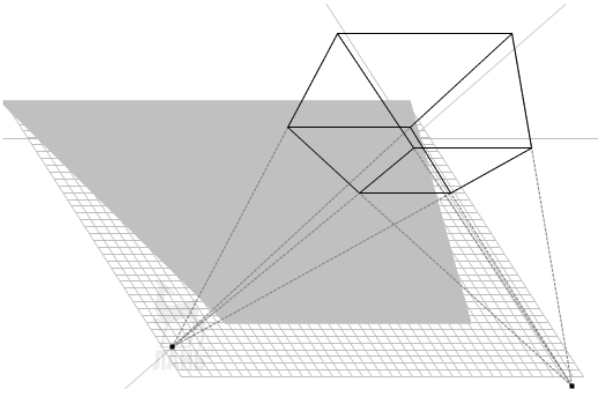

Рис. 8.9. Проект для построения платоновых тел

8.2.1. Структура данных

В проекте используется структура данных, показанная на рисунке 8.10.

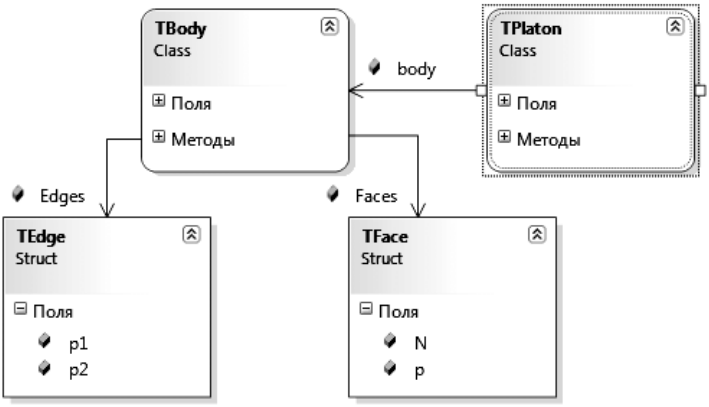

Рис. 8.10. Структура классов проекта

Основой проекта является класс `TPlaton`, в котором содержатся следующие поля:

- `double Xmin, Xmax, Ymin, Ymax` – прямоугольник в мировых координатах;
- `int I2, J2` – ширина и высота окна на экране;
- `double Alf, Bet` – углы поворота системы координат;
- `double Alf1, Bet1` – углы поворота тела относительно системы координат;
- `double Xs, Zs` – точки схода для перспективы;

– *double[] Sv* – координаты источника света;
– *bool visibleOXYZ* – признак видимости осей;
– *bool visiblePoint* – признак видимости точек схода;
– *bool visibleShadow* – признак видимости тени;
– *bool flRotate* – признак вращения тела или системы координат;
– *byte flEdge* – признак изображения (ребрами, гранями, стерео);
– *double AnglStereo* = 0.015 – угол для стереоизображения;
– *TBody body* – платоново тело;
– *TBody body0* – эталонный куб, используемый для перспективы.

Класс *TBody*, описывающий тело:

– *double[][] Vertexs* – массив вершин тела;
– *double[][] VertexsT* – массив временных вершин тела;
– *TEdge[] Edges* – массив ребер тела;
– *TFace[] Faces* – массив граней тела.

Структура *TSide* описывает грани:

```
public struct TFace
{
    public int[] p;             // номера вершин
    public double[] N;          // вектор нормали
}
```

Поле массива *p[]* показывают на номера вершин грани, поле *N* представляет собой вектор нормали к грани. Следующая структура предназначена для описания ребер:

```
public struct TEdge
{
    public int p1, p2;          // номера вершин
}
```

Два поля *p1* и *p2* переменных этой структуры показывают на номера вершин ребра.

> **Замечание.** Такая структура данных, во-первых, избыточна (ребра, например, можно вычислить по грани), во-вторых, это не единственно возможный вариант.

8.2.2. Инициализация тел

Класс *TBody* имеет конструктор, который получает параметр и может вызвать один из 5 методов для задания параметров одного из пяти платоновых тел (листинг 8.1).

Листинг 8.1. Конструктор класса *TBody*

```
public TBody(byte fl)
{
    switch (fl)
    {
```

```
                case 0:
                    Tetraedr(SizeBody);
                    break;
                case 1:
                    Hexaedr(SizeBody);
                    break;
                case 2:
                    Octaedr(SizeBody);
                    break;
                case 3:
                    Icosahedron(SizeBody);
                    break;
                case 4:
                    Dodecahedron(SizeBody);
                    break;
            }
        }
```

В листинге 8.2 приведен метод, инициализирующий данные для тетраэдра, который опирается на 4 вершины, имеет 6 ребер и 4 грани. Существенным моментом при задании номеров вершин, на которые опирается грань, является порядок их обхода. Это сказывается при вычислении нормали к грани и, в зависимости от порядка обхода, нормаль может быть либо внешней к телу, либо внутренней.

Листинг 8.2. Задание данных для тетраэдра

```
private void Tetraedr(double Size)
{
    Vertexs = new double[4][];
    VertexsT = new double[4][];
    for (int i=0; i<4; i++)
    {
        Vertexs[i] = new double[4];
        VertexsT[i] = new double[4];
    }

    Vertexs[0][0] = Size;
    Vertexs[0][1] = -Size; Vertexs[0][2] = -Size;
    Vertexs[1][0] = Size;
    Vertexs[1][1] = Size; Vertexs[1][2] = Size;
    Vertexs[2][0] = -Size;
    Vertexs[2][1]=-Size; Vertexs[2][2]= Size;
    Vertexs[3][0] = -Size;
    Vertexs[3][1] = Size; Vertexs[3][2] = -Size;
    for (int i = 0; i < 4; i++)
        Vertexs[i][3] = 1;
    Edges  =  new TEdge[6];
    Edges[0].p1 = 0; Edges[0].p2 = 1;
```

```
    Edges[1].p1 = 0; Edges[1].p2 = 2;
    Edges[2].p1 = 1; Edges[2].p2 = 2;
    Edges[3].p1 = 3; Edges[3].p2 = 0;
    Edges[4].p1 = 3; Edges[4].p2 = 1;
    Edges[5].p1 = 3; Edges[5].p2 = 2;
    Faces  =  new TFace[4];
    for (int i = 0; i < 4; i++)
        Faces[i].p  =  new int[3];

    Faces[0].p[0] = 0;
    Faces[0].p[1] = 1; Faces[0].p[2] = 2;
    Faces[1].p[0] = 1;
    Faces[1].p[1] = 3; Faces[1].p[2] = 2;
    Faces[2].p[0] = 0;
    Faces[2].p[1] = 2; Faces[2].p[2] = 3;
    Faces[3].p[0] = 0;
    Faces[3].p[1] = 3; Faces[3].p[2] = 1;
}
```

Методы инициализации данных для остальных платоновых тел приведены в проекте.

8.2.3. Преобразование координат

Перед рисованием графических элементов (линий и граней), которые опираются на точки с координатами в мировой системе координат, необходимо осуществить преобразования этих координат: поворот системы координат, перспективное проецирование и переход от мировых координат к экранным. Для этого предназначен следующий набор методов класса *TPlaton*.

Массив вершин *Vertexs* задает тело в неподвижной системе координат. В процессе вращения мы будем использовать вспомогательный массив *VertexsT*. При рисовании учитывается, что тело повернуто в неподвижной системе координат на углы $Alf1_X$, $Bet1_Y$, а сама система координат повернута на углы Alf_X, Bet_Y. Между двумя этими поворотами осуществляется перспективное проецирование. Все это реализуется методом *Rotate* следующим образом.

Листинг 8.3. Последовательность преобразования точек

```
L = body.Vertexs.Length;
for (int i = 0; i < L; i++)
{
   body.VertexsT[i] =
       Rotate(body.Vertexs[i],0,Alf1,0,0);
   body.VertexsT[i] =
       Rotate(body.VertexsT[i],1,Bet1,0,0);
   body.VertexsT[i] =
```

```
        Rotate(body.VertexsT[i],3,0,Xs, Zs);
    body.VertexsT[i] =
        Rotate(body.VertexsT[i],0,Alf,0, 0);
    body.VertexsT[i] =
        Rotate(body.VertexsT[i],1,Bet,0, 0);
}
```

Код метода *Rotate* представлен в листинге 8.4.

Листинг 8.4. Поворот вокруг осей координат и проецирование

```
public double[] Rotate(double[] V, int k, double fi, double
p, double r)
{
     double[][] M = new double[4][];
     for (int i = 0; i < 4; i++)
         M[i] = new double[4];
     for (int i = 0; i < 4; i++)
     {
         M[3][i] = 0; M[i][3] = 0;
     }
     M[3][3] = 1;
     switch (k)
     {
        case 0:
           M[0][0]=1; M[0][1]=0;     M[0][2]=0;
           M[1][0]=0; M[1][1]= Math.Cos(fi);
           M[1][2]=Math.Sin(fi);
           M[2][0]=0; M[2][1]=-Math.Sin(fi);
           M[2][2]=Math.Cos(fi);
           break;
        case 1:
           M[0][0]=Math.Cos(fi); M[0][1]=0;
           M[0][2]=-Math.Sin(fi);
           M[1][0]=0;    M[1][1]=1; M[1][2]=0;
           M[2][0]=Math.Sin(fi); M[2][1]=0;
           M[2][2]=Math.Cos(fi);
           break;
        case 2:
           M[0][0]= Math.Cos(fi);
           M[0][1]=Math.Sin(fi);
           M[0][2]=0;
           M[1][0]=-Math.Sin(fi);
           M[1][1]=Math.Cos(fi);
           M[1][2]=0;
           M[2][0]= 0;    M[2][1]=0;    M[2][2]=1;
           break;
        case 3:
           M[0][0]= 1; M[0][1]=0; M[0][2]=0;
```

```
            M[1][0]= 0; M[1][1]=1; M[1][2]=0;
            M[2][0]= 0; M[2][1]=0; M[2][2]=1;
            M[1][3]= p; M[2][3]=r;
            break;
    }
    return VM_Mult(V, M);
}
```

Умножение 4-вектора на матрицу реализуется методом *VM_Mult*.

Листинг 8.5. Умножение 4-вектора на матрицу

```
        double[] VM_Mult(double[] A, double[][] B)
        {
            double[] result = new double[4];
            for (int j = 0; j < 4; j++)
            {
                result[j] = A[0] * B[0][j];
                for (int k=1; k<4; k++)
                    result[j] +=A[k]*B[k][j];
            }
            if (result[3] != 0)
                for (int j = 0; j < 3; j++)
                    result[j] /= result[3];
            result[3] = 1;
            return result;
        }
```

Для проецирования на плоскость экрана предназначен, во-первых, метод *IJ()*.

Листинг 8.6. Преобразование вектора в экранные координаты

```
Point IJ(double[] Vt)
{
    Point result;
    Vt=Rotate(Vt, 0, Alf,0,0);
    Vt=Rotate(Vt, 1, Bet,0,0);
    result = new Point(II(Vt[0]), JJ(Vt[1]));
    return result;
}
```

Во-вторых, стандартный набор методов преобразования координат.

Листинг 8.7. Преобразование в экранные координаты и обратно

```
int II(double x)
{
    return (int)Math.Round((x - Xmin)*I2/
        (Xmax - Xmin));
}
```

```
int JJ(double y)
{
    return (int)Math.Round((y - Ymax)*J2/
        (Ymin - Ymax));
}
double XX(int I)
{
    return Xmin+(Xmax-Xmin)*I/I2;
}
double YY(int J)
{
    return Ymax + (Ymin - Ymax) * J / J2;
}
```

8.2.4. РИСОВАНИЕ

Рисование тела происходит на канве *bitmap* в трех режимах: ребра, грани и стерео. Подготовка изображения реализуется методом класса *TPlaton*.

Листинг 8.8. Рисование платоновых тел

```
public void Draw()
{
    using (Graphics g = Graphics.FromImage(bitmap))
    {
       Color cl;
       if (flEdge==2)
          cl = Color.FromArgb(0, 0, 0);
       else
          cl = Color.FromArgb(255, 255, 255);
       g.Clear(cl);
       g.SmoothingMode = SmoothingMode.HighQuality;

       if (visibleOXYZ)   // рисование осей
          OXYZ(g);
       if (visibleShadow) // рисование тени
          Shadow(g);
       if (visiblePoint)  // рисование перспективы
          DrawPoint(g);
       DrawBody(g);       // рисование тела
     }
  }
```

Начало рисования платоновых тел реализуется методом *DrawBody()*.

Листинг 8.9. Рисование платоновых тел

```
public void DrawBody(Graphics g)
{
    int L;
    if ((flEdge == 0) | (flEdge == 2))
    {
       L = body.Vertexs.Length;
       for (int i = 0; i < L; i++)
       {
          body.VertexsT[i] =
            Rotate(body.Vertexs[i],0, Alf1, 0, 0);
          body.VertexsT[i] =
            Rotate(body.VertexsT[i], 1, Bet1, 0, 0);
          body.VertexsT[i] =
            Rotate(body.VertexsT[i], 3, 0, Xs, Zs);
          body.VertexsT[i] =
            Rotate(body.VertexsT[i], 0, Alf, 0, 0);
          body.VertexsT[i] =
            Rotate(body.VertexsT[i], 1, Bet, 0, 0);
       }
    }
    switch (flEdge)
    {
        case 0:  // Edges
           DrawEdge(g);
           break;
        case 1:  // faces
           DrawFaces(g);
           break;
        case 2: // stereo
           DrawStereo(g);
           break;
    }
}
```

Если тело рисуется ребрами, то необходимо циклом перебрать все ребра и нарисовать отрезки от первой точки ребра до второй, получая доступ к вершинам следующим образом *VertexsT[Edges[i].p1]*.

Листинг 8.10. Рисование платоновых тел ребрами

```
void DrawEdge(Graphics g)
{
   int L = body.Edges.Length;
   for (int i = 0; i < L; i++)
   {
      int x1 =
         II(body.VertexsT[body.Edges[i].p1][0]);
```

```
      int y1 =
         JJ(body.VertexsT[body.Edges[i].p1][1]);
      int x2 =
         II(body.VertexsT[body.Edges[i].p2][0]);
      int y2 =
         JJ(body.VertexsT[body.Edges[i].p2][1]);
      g.DrawLine(Pens.Black, x1, y1, x2, y2);
   }
}
```

При рисовании тела гранями необходимо циклом (*i = 0...L-1*) перебрать все грани. Для каждой грани циклом (*j = 0...L₀-1*) перебрать все вершины. Доступ к вершинам реализуется через номера вершин грани *VertexsT[Sides[i].p[j]]*.

Листинг 8.11. Рисование платоновых тел гранями

```
void DrawFaces(Graphics g)
{
    int L1 = body.Faces.Length;
    int L0 = body.Faces[0].p.Length;
    Point[] w = new Point[L0];

    double[][] Vn = new double[3][];
    double[][] Wn = new double[3][];
    for (int i = 0; i < L1; i++)
    {
       for (int j = 0; j < L0; j++)
       {
          double[] Vt =
             body.Vertexs[body.Faces[i].p[j]];
          Vt = Rotate(Vt, 0, Alf1, 0, 0);
          Vt = Rotate(Vt, 1, Bet1, 0, 0);
          if (j <= 2) Vn[j] = Vt;
          Vt = Rotate(Vt, 3, 0, Xs, Zs);
          Vt = Rotate(Vt, 0, Alf, 0, 0);
          Vt = Rotate(Vt, 1, Bet, 0, 0);
          w[j].X = II(Vt[0]);
          w[j].Y = JJ(Vt[1]);
          if (j <= 2) Wn[j] = Vt;
       }
       body.Faces[i].N = Norm(Vn[0], Vn[1], Vn[2]);
       double[] NN = Norm(Wn[0], Wn[1], Wn[2]);
       double d = Math.Abs(NN[2]);
       Color col = Color.FromArgb(0, 0,
          (byte)(Math.Round(255 * d)));
       SolidBrush br = new SolidBrush(col);
       if (NN[2] < 0)
```

```
  g.FillPolygon(br, w);
    }
}
```

Для каждой грани нам потребуется два единичных вектора нормали: вектор нормали *Faces[i].N* к грани повернутого тела в неподвижной системе координат для вычисления освещенности и вектор нормали *NN* к грани повернутого тела в повернутой системе координат для вычисления невидимых граней.

Для вычисления этих нормалей потребуется три вершины грани в неподвижной системе координат *Vn* и три вершины грани в повернутой системе координат *Wn*.

Нормали вычисляются методом *Norm()*:

```
Faces[i].N=Norm(Vn[0],Vn[1],Vn[2]);
NN=Norm(Wn[0],Wn[1],Wn[2]);
```

Текст метода *Norm* приведен в листинге 8.12.

Листинг 8.12. Вычисление вектора нормали

```
double[] Norm(double[] V1,double[] V2,double[] V3)
{
   double[] Result = new double[4];
   double[] A = new double[4];
   double[] B = new double[4];
   A[0]=V2[0]-V1[0]; A[1]=V2[1]-V1[1];
   A[2]=V2[2]-V1[2];
   B[0]=V3[0]-V1[0]; B[1]=V3[1]-V1[1];
   B[2]=V3[2]-V1[2];
   double u=A[1]*B[2]-A[2]*B[1];
   double v=-A[0]*B[2]+A[2]*B[0];
   double w=A[0]*B[1]-A[1]*B[0];
   double d=Math.Sqrt(u*u+v*v+w*w);
   if (d!=0)
   {
        Result[0]=u/d; Result[1]=v/d; Result[2]=w/d;
   }
   else
   {
        Result[0]=0; Result[1]=0; Result[2]=0;
   }
   return Result;
 }
```

Использование вектора нормали *NN* для отсечения невидимых граней, у которых проекция этого вектора на ось *Z* отрицательна, очень просто и сводится к проверке:

```
   if (NN[2] < 0) g.FillPolygon(br, w);
```

Использование вектора нормали *Faces[i].N* требует дополнительных пояснений, которые будут даны в следующем пункте.

8.2.5. РИСОВАНИЕ ПОЛУТОНАМИ

Напомним, что класс цвета точки *Color* содержит информацию о долях синего (*B*), зеленого (*G*) и красного (*R*) цветов. Поэтому для рисования полутонами синего цвета, например, необходимо определить некую целочисленную функцию *f(x)* с диапазоном изменения [0...255] и цвет определять соотношением *Color = f(x)*$00010000.

Будем считать, что выбор цвета грани определяется длиной проекции на ось *Z* единичной нормали *Faces[i].N* в неподвижной системе координат и реализуется операторами:

Листинг 8.13. Вычисление цветов

```
double d = Math.Abs(NN[2]);
Color col =
    Color.FromArgb(0, 0, (byte)(Math.Round(255 * d)));
```

Итак, мы предположили, что источник света находится в бесконечности на оси *Z*.

Достаточно просто решить более общую задачу, когда источник находится в бесконечности на прямой, проходящей через любую точку и центр тела. В этом случае необходимо найти через скалярное произведение проекцию вектора нормали к грани на эту прямую.

8.2.6. ПОСТРОЕНИЕ ТЕНИ

Будем считать, что:

- источник света находится в точке с координатами *Sv = (-0.5, -0.5,5)*. Впрочем, эти координаты меняются на форме *FormToos*;
- тело отбрасывает тень на плоскость *z=const* (рис. 8.11).

Для нахождения координат точки A' на плоскости *Z = Zh* необходимо записать условие колинеарности двух векторов

$$\begin{vmatrix} i & j & k \\ x_i - x_0 & y_i - y_0 & z_i - z_0 \\ x - x_0 & y - y_0 & z - z_0 \end{vmatrix} = 0, \qquad (8.5)$$

что приводит к системе уравнений для определения *x* и *y*:

$$\begin{aligned} &(y_i - y_0)(z_h - z_0) - (z_i - z_0)(y_i - y_0) = 0; \\ &(x_i - x_0)(z_h - z_0) - (x - z_0)(z_i - z_0) = 0; \\ &(x_i - x_0)(y - y_0) - (x - x_0)(y_i - y_0) = 0. \end{aligned} \qquad (8.6)$$

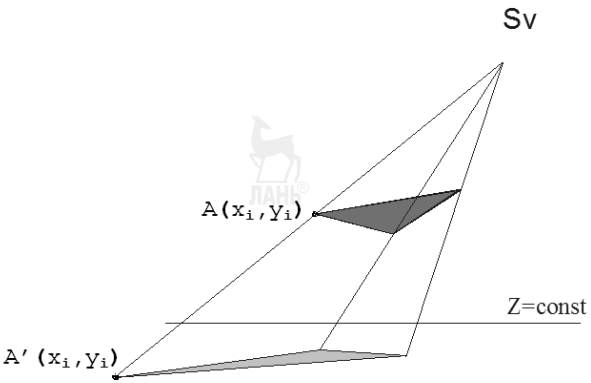

Рис. 8.11. Проекция грани на плоскость

Решая уравнения (8.6) относительно переменных x и y, получаем:

$$y = y_0 + \frac{(y_i - y_0)(z_h - z_0)}{z_i - z_0};$$

$$x = x_0 + \frac{(x_i - x_0)(z_h - z_0)}{z_i - z_0}. \qquad (8.7)$$

Соотношения (8.7) используются при построении тени от многогранника следующим образом: переберем все грани тела циклом *i=0...L1-1*. Для каждой грани переберем все вершины циклом *j=0...L0*-1. Вершину повернем вместе с телом на углы *Alf1* и *Bet1*. Затем найдем координаты точки `Vt` на плоскости проекции. Повернем систему координат на углы *Alf* и *Bet*. Наконец, с помощью методов *II()* и *JJ()* найдем координаты образа точки на экране. После вычисления всех таких точек методом канвы *Polygon* строится тень от одной грани.

Листинг 8.14. Построение тени от многогранника

```
private void Shadow(Graphics g)
{
   // источник света
   Point P=IJ(Sv);
   g.FillRectangle(Brushes.Red, P.X-2,P.Y-2,5,5);
   g.DrawLine(Pens.Red, P.X - 4, P.Y, P.X + 4, P.Y);
   g.DrawLine(Pens.Red, P.X, P.Y - 4, P.X, P.Y + 5);
   // сетка
   Point P1,P2;
   double Zh = -1;
   for (int i = 0; i < 41; i++)
   {
```

```
        P1 = IJ(ToVector(-1 + i * 0.2/4, -1, Zh));
        P2 = IJ(ToVector(-1 + i * 0.2/4, 1, Zh));
        g.DrawLine(Pens.Silver,P1.X,P1.Y,P2.X, P2.Y);
    }
    for (int i = 0; i < 41; i++)
    {
        P1 = IJ(ToVector(-1 , -1+ i * 0.2 / 4, Zh));
        P2 = IJ(ToVector( 1 , -1+ i * 0.2 / 4, Zh));
        g.DrawLine(Pens.Silver,P1.X,P1.Y,P2.X, P2.Y);
    }
    // Тень
    int L1 = body.Faces.Length;
    int L0 = body.Faces[0].p.Length;
    Point[] w = new Point[L0];
    for (int i = 0; i < L1; i++)
    {
       for (int j = 0; j < L0; j++)
       {
          double[] Vt =
             body.Vertexs[body.Faces[i].p[j]];
          Vt = Rotate(Vt, 0, Alf1, 0, 0);
          Vt = Rotate(Vt, 1, Bet1, 0, 0);
          Vt = Rotate(Vt, 3, 0, Xs, Zs);
          Vt[0] = Sv[0] + (Vt[0] - Sv[0]) *
             (Zh - Sv[2]) / (Vt[2] - Sv[2]);
          Vt[1] = Sv[1] + (Vt[1] - Sv[1]) *
             (Zh - Sv[2]) / (Vt[2] - Sv[2]);
          Vt[2] = Zh;
          Vt = Rotate(Vt, 0, Alf, 0, 0);
          Vt = Rotate(Vt, 1, Bet, 0, 0);
          w[j].X = II(Vt[0]);
          w[j].Y = JJ(Vt[1]);
       }
       g.FillPolygon(Brushes.Silver, w);
    }
}
```

Завершает построение тени рисование сетки на плоскости *z = Zh*

Листинг 8.15. Построение сетки на плоскости z = Zh

```
    // сетка
    Point P1,P2;
    double Zh = -1;
    for (int i = 0; i < 41; i++)
    {
        P1 = IJ(ToVector(-1 + i * 0.2/4, -1, Zh));
        P2 = IJ(ToVector(-1 + i * 0.2/4, 1, Zh));
        g.DrawLine(Pens.Silver,P1.X,P1.Y,P2.X, P2.Y);
```

```
}
for (int i = 0; i < 41; i++)
{
    P1 = IJ(ToVector(-1 , -1+ i * 0.2 / 4, Zh));
    P2 = IJ(ToVector( 1 , -1+ i * 0.2 / 4, Zh));
    g.DrawLine(Pens.Silver,P1.X,P1.Y,P2.X, P2.Y);
}
```

и построение изображения источника света.

Листинг 8.16. Рисование источника света

```
// источник света
Point P=IJ(Sv);
g.FillRectangle(Brushes.Red, P.X-2,P.Y-2,5,5);
g.DrawLine(Pens.Red, P.X - 4, P.Y, P.X + 4, P.Y);
g.DrawLine(Pens.Red, P.X, P.Y - 4, P.X, P.Y + 5);
```

8.2.6.1. Перспективное проецирование

Для реалистичного изображения трехмерного тела необходимо учитывать искажение его проекции в соответствии с законами перспективы.

В главе 3 приведена общая матрица преобразования однородных координат, которая имеет вид

$$\begin{vmatrix} a & x & x & p \\ x & d & x & q \\ x & x & e & r \\ m & n & l & s \end{vmatrix}, \qquad (8.8)$$

где `a,d,e` — масштабирование; `m,n,l` — смещение; `p,q,r` — проецирование; `s` — комплексное масштабирование; `x` — вращение.

Всего различают три типа проекций: одноточечная ($r \neq 0$); двухточечная ($p, q \neq 0$); трёхточечная ($p, q, r \neq 0$).

На фотографиях, картинах, экране изображения кажутся нам естественными и правильными. Такие изображения называют перспективными. Одно из основных свойств таких изображений — то, что более удаленные предметы изображаются в меньших масштабах, параллельные прямые в общем случае непараллельны. В итоге геометрия изображения оказывается достаточно сложной, и по готовому изображению сложно определить те или иные части объекта.

Обычная перспективная проекция — это центральная проекция на плоскость прямыми лучами, проходящими через точку — *центр проецирования*. Один из проецирующих лучей перпендикулярен к плоскости проецирования и называется *главным*. Точка пересечения этого луча и плоскости проекции — *главная точка* картины.

$$\begin{vmatrix} x & y & z & 1 \end{vmatrix} * \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} \frac{x}{rz+1} & \frac{y}{rz+1} & \frac{z}{rz+1} & 1 \end{vmatrix}. \qquad (8.9)$$

Меняя p, q и r, мы регулируем точку схода.

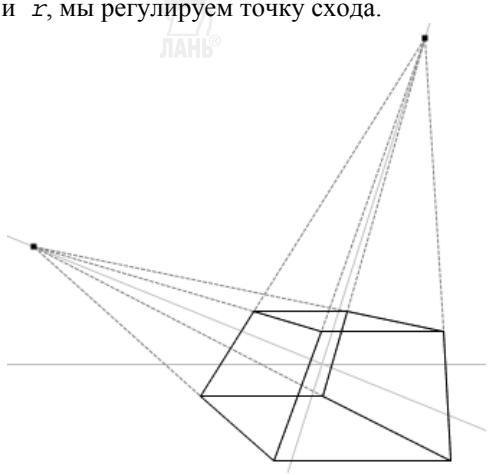

Рис. 8.12. Двухточечное проецирование

Перспективное двухточечное проецирование реализуется матрицей:

Листинг 8.17. Перспективное двухточечное проецирование

```
    case 3:
       M[0][0]= 1; M[0][1]=0; M[0][2]=0;
       M[1][0]= 0; M[1][1]=1; M[1][2]=0;
       M[2][0]= 0; M[2][1]=0; M[2][2]=1;
       M[1][3]= p; M[2][3]=r;
       break;
}
```

которая используется методом `Rotate` следующим образом:

```
VertexsT[i]=Rotate(Vertexs[i],  1, Alf1,0,0);
VertexsT[i]=Rotate(VertexsT[i], 2, Bet1,0,0);
VertexsT[i]=Rotate(VertexsT[i], 4, 0,Xs,Zs);
VertexsT[i]=Rotate(VertexsT[i], 1, Alf,0,0);
VertexsT[i]=Rotate(VertexsT[i], 2, Bet,0,0);
```

8.2.6.2. Стереоизображение

Напомним, что плоское изображение ребрами многогранника в повернутой системе координат реализуется циклом по всем ребрам.

Листинг 8.18. Рисование ребрами

```
void DrawEdge(Graphics g)
{
   int L = body.Edges.Length;
   for (int i = 0; i < L; i++)
   {
      int x1 =
         II(body.VertexsT[body.Edges[i].p1][0]);
      int y1 =
         JJ(body.VertexsT[body.Edges[i].p1][1]);
      int x2 =
         II(body.VertexsT[body.Edges[i].p2][0]);
      int y2 =
         JJ(body.VertexsT[body.Edges[i].p2][1]);
      g.DrawLine(Pens.Black, x1, y1, x2, y2);
   }
}
```

Для создания стереоизображения, на которое можно смотреть сквозь очки с красной и синей линзами, нарисуем два совмещенных изобращения, слегка повернутых вокруг оси *OY* в разные стороны на угол *AnglStereo*. Изображение красными линиями повернем на угол *AnglStereo*, а изображение синими линиями – на угол *-AnglStereo*. Простые расчеты показывают, что угол *AnglStereo* должен меняться в диапазоне 0,01…0,03 радиан.

Листинг 8.19. Стереорисование ребрами

```
void DrawStereo(Graphics g)
{
   Color c = Color.FromArgb(0xCF, 0, 0);
   Pen myPenR = new Pen(c, 2);
   Pen myPenB = new Pen(Color.Blue, 2);
   int L = body.Edges.Length;
   for (int i = 0; i < L; i++)
   {
      TEdge e = body.Edges[i];
      double[] V1 = Rotate(body.VertexsT[e.p1], 1,
         AnglStereo, 0, 0);
      double[] V2 = Rotate(body.VertexsT[e.p2], 1,
         AnglStereo, 0, 0);
      g.DrawLine(myPenR, II(V1[0]), JJ(V1[1]),
         II(V2[0]), JJ(V2[1]));

      V1 = Rotate(body.VertexsT[e.p1], 1, -
         AnglStereo, 0, 0);
      V2 = Rotate(body.VertexsT[e.p2], 1, -
```

```
            AnglStereo, 0, 0);
        g.DrawLine(myPenB, II(V1[0]), JJ(V1[1]),
            II(V2[0]), JJ(V2[1]));
    }
}
```

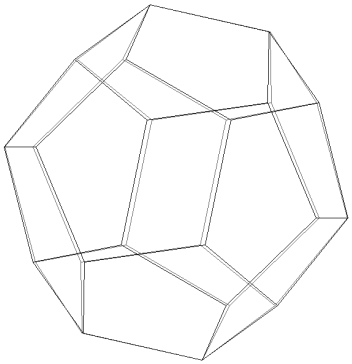

Рис. 8.13. Стереоизображение многогранника

Красное изображение рисуется на канве *Bitmap*, синее – на канве *BitmapBlue*. Затем *BitmapBlue* копируется на *Bitmap* в режиме *cmSrcPaint*, в котором цвета точек *Bitmap* и *BitmapBlue* складываются операцией OR. В результате общие точки изображений имеют цвет $FF00FF.

Глава 9. Использование графической библиотеки OpenGL

Библиотека OpenGL состоит из нескольких сотен методов, позволяющих описывать трехмерные графические объекты и операции над ними. Эти методы позволяют:

- рисовать геометрические и растровые примитивы;
- работать с цветом в режиме *RGBA* и в индексном режиме;
- реализовывать видовые преобразования и задавать модель освещения;
- удалять невидимые ребра и грани; использовать различные эффекты (сопряжения цветов, прозрачности, наложения текстуры, освещения распределенными и точечными источниками света, «тумана»);
- использование B-сплайнов.

Работа с цветом в режиме *RGBA* позволяет устанавливать доли красного, зелёного, синего цветов и использовать ещё один параметр – альфа-параметр, отвечающий за прозрачность (1 – полная непрозрачность, 0 – полная прозрачность).

Под видовыми преобразованиями имеются в виду геометрические преобразования системы координат: смещения и повороты системы координат.

Имеются графические примитивы: точки, отрезки прямых линий, многоугольники, пиксельные прямоугольники или битовые карты. Примитив задается множеством координат своих вершин. С вершинами связываются следующие атрибуты: цвет, нормаль и координаты текстуры.

В OpenGL предусмотрены операции с плоскими и трехмерными объектами: матрицы преобразований, задание коэффициентов для уравнения освещенности и т.п. Необходимо отметить, что в OpenGL нет средств описания структур сложных объектов.

Задание координат, цветов, векторов нормали и точек привязки текстур происходит между командами *glBegin(DrawMode)* и *glEnd()*.

Для построения треугольника с вершинами в точках (10,10,0), (10,20,0) и (20,0,10) необходимо, например, написать такую последовательность:

```
glBegin(GL_TRIANGLES);
  glVertex3i(10,10,0);
  glVertex3i(10,20,0);
  glVertex3i(20,0,10);
glEnd();
```

В OpenGL предусмотрено десять геометрических объектов:

– *GL_POINTS* – последовательность точек;
– *GL_LINE_STRIP* – ломаная линия;
– *GL_LINE_LOOP* – замкнутая ломаная линия;
– *GL_LINES* – отрезок линии;
– *GL_TRIANGLE_STRIP* – множество связных треугольников;
– *GL_TRIANGLE_FAN* – множество треугольников, связанных веером;
– *GL_TRIANGLES* – треугольник;
– *GL_QUAD_STRIP* – связные четырехугольники;
– *GL_QUADS* – четырехугольник;
– *GL_POLYGON* – выпуклый многоугольник.

Из этих объектов можно строить более сложные объекты.

Нормаль – трехмерный вектор, используемый для формирования освещенности поверхности.

Координаты вершины подвергаются видовому преобразованию однородной системы координат умножением вектора на матрицу 4×4. Эта матрица содержит операции сдвига, поворота и масштабирования. В OpenGL допустимо проводить геометрические преобразования не только путем последовательных поворотов вокруг осей *X*, *Y* или *Z*, но и относительно произвольного вектора.

Для задания доли каждого цвета можно использовать целые числа в диапазоне 0...255 и вещественные число от 0.0 до 1.0.

Освещенность изменяется в зависимости от следующих параметров: свойств материала; отражающих свойств поверхности объекта; параметров источника света.

Вид изображения зависит от параметров: модели освещенности, расположения источников света, мощности источников света, свойств среды и т.д.

Все команды, кроме задания координат, цветов, векторов нормали и точек привязки текстур, могут быть размещены и вне блока *glBegin…glEnd*, что ускоряет работу программы.

Команды в OpenGL выполняются сервером сразу после их поступления. Этот режим используется в приложениях, в которых могут изменять координаты вершин примитивов и режимов их вывода.

Если координаты вершин примитивов не меняются, то для ускорения работы можно использовать дисплейные списки, содержащие последовательности команд OpenGL.

Дисплейный список находится на сервере и выводится на экран при поступлении команды *glCallList()*.

Мы не ставим перед собой задачу описать все возможности библиотеки OpenGL. Более подробную информацию можно получить в документации

или, например, в книге Ю. Тихомирова [40], в которой приведено множество примеров, разработанных в среде Visual C++.

9.1. Установка и завершение работы с OpenGL

Прямой поддержки библиотеки OpenGL в .NET Framework нет. Мы подключим библиотеку Tao Framework.

Tao Framework – библиотека с открытым исходным кодом, предназначенная для разработки кросс-платформенного программного обеспечения в среде .NET Framewrok.

Некоторые библиотеки, включенные в Tao Framework:

- OpenGL – предназначена для визуализации 2D и 3D графики;
- FreeGLUT – аналог библиотеки GLUT;
- DevIL (OpenIL) – предназначена для работы с изображениями. Библиотека позволяет читать 43 формата и записывать изображения 17 форматов;
- OpenAL – предназначена для работы с аудиоданными;
- PhysFS – предназначена для работы с архивами;
- SDL – используется при написании мультимедийных приложений;
- GNU/Linux ODE – программный интерфейс, реализующий динамику твёрдого тела и обнаружение столкновений;
- FreeType – предназначена для рисования шрифтов;
- FFmpeg – предназначена для работы с аудио- и видеоданными.

Процесс установки библиотеки Tao Framework описан на сайте http://www.esate.ru/page/uroki-OpenGL-c-sharp/. Но для первоначальной работы с OpenGL не обязательно устанавливать весь пакет: достаточно иметь пять DLL:

- freeglut.dll;
- Tao.FreeGlut.dll;
- Tao.FreeType.dll;
- Tao.OpenGl.dll;
- Tao.Platform.Windows.dll.

Далее описывается последовательность действий при создании нового оконного проекта.

Шаг 1. Создайте новый проект.

Шаг 2. В обозревателе решений нажмите правую кнопку мыши на строке «Ссылки». Появится окно (рис.9.1), в котором на закладке «Обзор» необходимо найти и выделить четыре DLL: Tao.FreeGlut.dll; Tao.FreeType.dll; Tao.OpenGl.dll; Tao.Platform.Windows.dll.

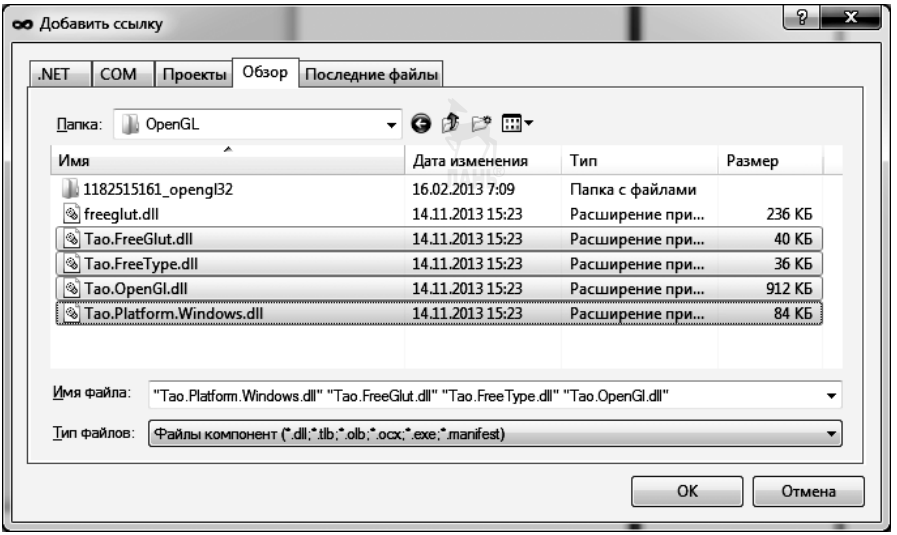

Рис. 9.1. Выбор библиотек DLL

После нажатия кнопки *OK* в ссылках добавятся четыре выбранных файла (рис. 9.2).

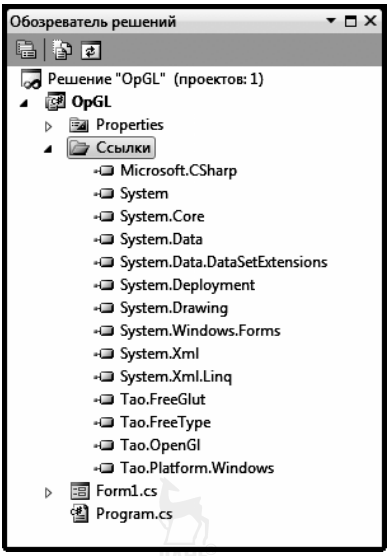

Рис. 9.2. Обозреватель решения

Шаг 3. На панели элементов на закладке «Общие» нажмите правую кнопку и в меню выберите строку «Выбрать элементы» (рис. 9.3).

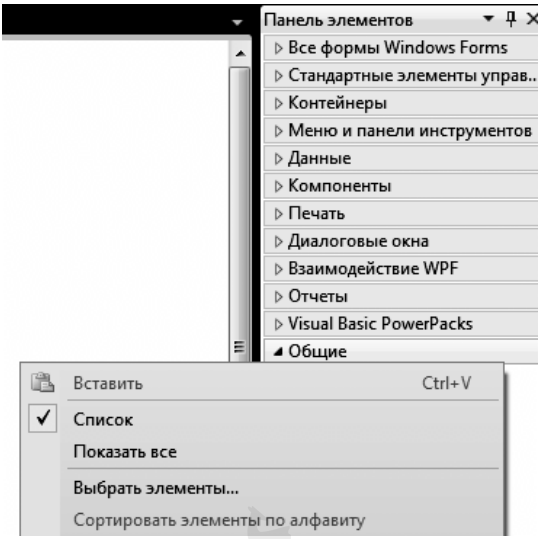

Рис. 9.3. Выбор элементов

В окне «Выбор элементов» нажмите кнопку «Обзор» (рис. 9.4).

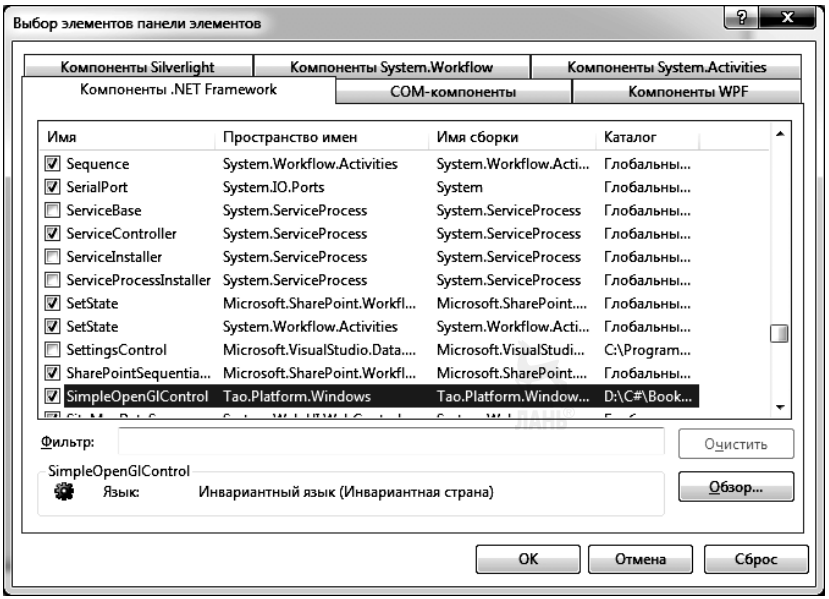

Рис. 9.4. Добавление компонента

Найдите файл Tao.Platform.Windows.dll на диске (рис. 9.5).

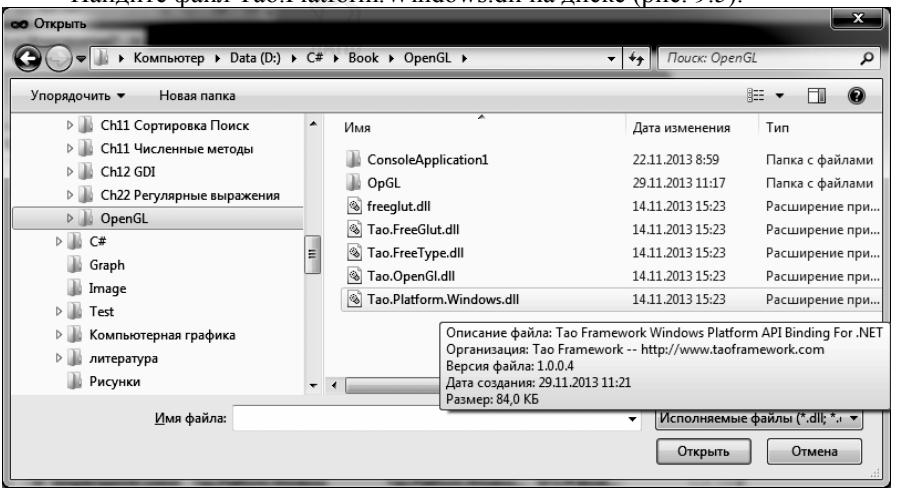

Рис. 9.5. Установка компонента Tao.Platform.Windows.dll

После нажатия кнопки «Открыть» на панели элементов добавится элемент управления `SimpleOpeGLControl` – основной компонент для работы с OpenGL (рис. 9.6). Этот компонент необходимо добавить на форму и переименовать, например `CGL`.

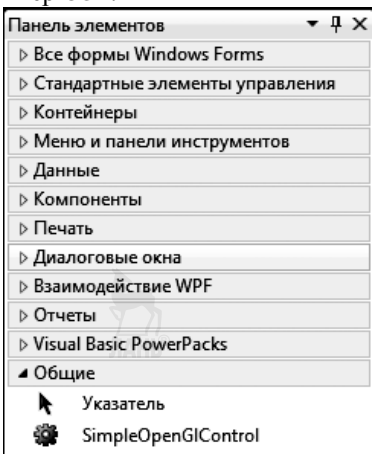

Рис. 9.6. Элемент управления *SimpleOpeGLControl*

После запуска проекта на выполнение в папке `Debug` появятся 4 файла: `Tao.FreeGlut.dll`, `Tao.FreeType.dll`, `Tao.OpenGl.dll`, `Tao.Platform.Windows.dll`.

Шаг 4. В папку *Debug* скопируйте файл *freeglut.dll*, который не является сборкой и поэтому не может быть добавлен через обозреватель решения.

Шаг 5. В конструкторе формы добавьте строчку для инициализации этого объекта.

Листинг 9.1. Инициализация объекта CGL

```
public Form1()
{
    InitializeComponent();
    CGL.InitializeContexts();
}
```

Шаг 6. Создайте класс *G*, а в нем метод *InitGL()*.

Листинг 9.2. Инициализация параметров

```
public static void InitGL(SimpleOpenGlControl CGL)
{
    Glut.glutInit(); // инициализация библиотеки Glut
    Glut.glutInitDisplayMode(Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
       Glut.GLUT_DEPTH);
    // очистка окна
    Gl.glClearColor(255, 255, 255, 1);
    // установка порта вывода по элементу CGL
    Gl.glViewport(0, 0, CGL.Width, CGL.Height);
    // настройка проекции
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    Glu.gluPerspective(45, (float)CGL.Width
          / (float)CGL.Height, 0.1, 200);
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
    // настройка параметров OpenGL для визуализации
    Gl.glEnable(Gl.GL_DEPTH_TEST);
}
```

Шаг 7. Для работы с библиотеками необходимо подключить следующие пространства имен.

Листинг 9.3. Подключение пространств имен

```
// для работы с библиотекой OpenGL
using Tao.OpenGl;
// для работы с библиотекой FreeGLUT
using Tao.FreeGlut;
// для работы с элементом управления SimpleOpenGLControl
using Tao.Platform.Windows;
```

Шаг 8. Метод *InitGL()* вызывается на форме в обработчике *Form1_Load()*.

Листинг 9.4. Вызов метода InitGL( )

```
private void Form1_Load(object sender, EventArgs e)
{
    G.InitGL(CGL);
}
```

Шаг 9. В классе *G* создайте метод рисования *Draw( )*, который будет вызван в обработчике *Form1_Paint()*.

Листинг 9.5. Метод рисования

```
public static void Draw()
{
    initLight();
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
       Gl.GL_DEPTH_BUFFER_BIT);

    Gl.glMaterialfv(Gl.GL_FRONT_AND_BACK,
        Gl.GL_AMBIENT, mat1_amb); // фоновый цвет материала
    Gl.glMaterialfv(Gl.GL_FRONT_AND_BACK,
        Gl.GL_DIFFUSE, mat1_dif); // диффузионные свойства
    Gl.glMaterialfv(Gl.GL_FRONT_AND_BACK,
        Gl.GL_SPECULAR, mat1_spec); // зеркальные свойства
    Gl.glMaterialf(Gl.GL_FRONT_AND_BACK,
        Gl.GL_SHININESS, mat1_shininess);

    Gl.glLoadIdentity();
    Gl.glPushMatrix();
    Gl.glTranslated(0, 0, -6);
    Gl.glRotated(45, 1, 0, 0);
    Gl.glRotated(45, 0, 1, 0);
    Glut.glutSolidTorus(0.5, 1.5, 32, 32);  // рисование тора
    Gl.glPopMatrix();
    Gl.glFlush();
}
```

Шаг 10. Метод *Draw()* вызывает метод настройки источника света.

Листинг 9.6. Настройки источника света

```
static void initLight()
{
    float[] light_ambient = { 0.0f, 0.0f, 0.0f, 1.0f };
    float[] light_diffuse = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] light_specular = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] light_position = { 1, 1, 1, 0.0f };
    // параметры источника света
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_AMBIENT, light_ambient);
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_DIFFUSE, light_diffuse);
```

```
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_SPECULAR, light_specular);
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_POSITION, light_position);
    // включаем освещение и источник света
    Gl.glEnable(Gl.GL_LIGHTING);
    Gl.glEnable(Gl.GL_LIGHT0);
    // включаем z-буфер
    Gl.glEnable(Gl.GL_DEPTH_TEST);
}
```

В результате запуска проекта будет нарисован тор (рис. 9.7).

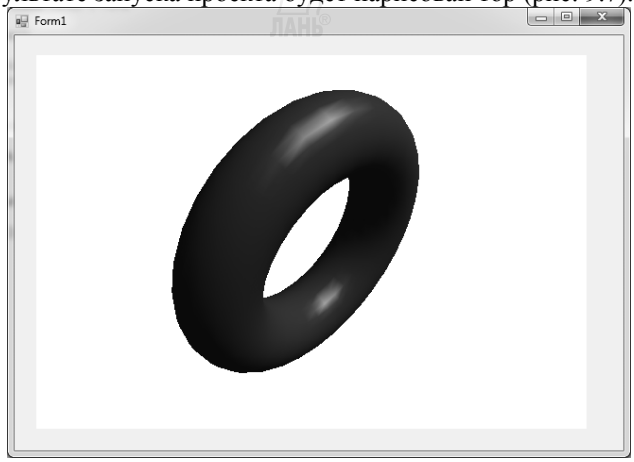

Рис. 9.7. Тор

В программе используются следующие команды:

- *glViewport(x: Glint; y: Glint; width: GLsizei; height: GLsizei)* – определяет область вывода;
- *x*, *y* – определяют левый верхний угол экрана в пикселах (по умолчанию 0,0);
- *width* и *height* – определяют ширину и высоту окна экрана;
- *glOrtho(left, right, bottom, top, near, far: GLdouble)* – умножает текущую матрицу на ортографическую матрицу. Параметры:
- *left*, *right* – координаты левой и правой вертикальных плоскостей отсечения;
- *bottom*, *top* — координаты нижней и верхней горизонтальных плоскостей отсечения;
- *near, far* – расстояния до ближней и дальней плоскостей отсечения. Эти расстояния отрицательны, если плоскости должны быть за наблюдателем.

– *glClear(mask: GLbitfield)* – очищает буферы в пределах окна экрана. Параметр *mask* образуется с помощью битовой операции *or* из масок, указывающих буферы, которые должны очищаться;
– *glFlush* – гарантирует завершение команд за конечное время;
– *glFinish* – блокирует дальнейшее выполнение программы, пока все предыдущие *GL*-команды не будут завершены.

9.2. КОМАНДЫ И ПРИМИТИВЫ OPENGL

9.2.1. СИНТАКСИС КОМАНД

В синтаксисе названия команд определяется число аргументов, их тип, символом *v* указывается, что в качестве аргумента используется указатель на массив:

```
NameCommand [1|2|3|4] [b|s|i|f|d|ub|us|ui] [v]
```

– Одна из цифр 1|2|3|4 показывает на число аргументов.
– Символы b|s|i|f|d|ub|us|ui показывают на тип аргументов. Типы аргументов, допустимые в OpenGL, приведены в таблице 9.1.
– Символ *v* указывает на то, что в качестве аргумента используется указатель на массив.

Таблица 9.1

Соответствие символов и типов

Символы	**Тип OpenGL**	**Тип C#**
b	*GLbyte*	*sbyte*
s	*GLshort*	*short*
u	*GLint*	*int*
f	*GLfloat*	*float*
d	*GLdouble*	*double*
ub	*GLubyte*	*byte*
us	*GLushort*	*uint*
ui	*GLuint*	*cardinal*

В таблице 9.2 показано соответствие типов OpenGL и C#, не участвующих в именах команд OpenGL.

Таблица 9.2

Соответствие типов

Тип OpenGL	**Тип C#**
GLenum	*Cardinal*
GLboolean	*bool*

Продолжение табл. 9.2

Тип OpenGL	Тип C#
GLbitfield	*Cardinal*
GLsizei	*int*
GLclampf	*float*
GLclampd	*double*

В качестве примера приведем команду, определяющую координаты одной точки на плоскости: *glVertex2f*(300,300).

9.2.2. Вершины

Вершина — это точка в 2D- или 3D-пространстве, координаты которой задаются командами:

```
glVertex[2 3 4][s i f d](coord: GLtype);
glVertex[2 3 4][s i f d]v(coord: PGLtype);
```

Параметр coord задает 4 однородные координаты вершины (*x*,*y*,*z*,*w*). Команда *glVertex2* определяет координаты (*x*,*y*,0,1), команда *glVertex3* определяет координаты (*x*,*y*,*z*,1), команда *glVertex4* определяет четыре однородные координаты (*x*,*y*,*z*,*w*).

9.2.3. Примитивы

В библиотеке OpenGL определены следующие примитивы: точки, линии, многоугольники, прямоугольники пикселов. Примитив или группа однородных примитивов задается вершинами и связанными с этой группой свойствами. Группа определяется скобками:

```
glBegin(mode: GLenum);
   …
glEnd().
```

Параметр *mode* определяет тип фигур. Возможные значения этого параметра приведены в таблице 9.3.

Таблица 9.3

Значения параметра *mode*

mode		**Значение**
GL_POINTS	1	Вершина рассматривается как независимая точка
GL_LINES	2	Пара вершин представляет собой независимый отрезок

Продолжение табл. 9.3

mode		**Значение**
GL_LINE_LOOP	2	Замкнутая ломаная линия: многоугольник с *N* вершинами
GL_LINE_STRIP	2	Ломаная линия
GL_TRIANGLES	3	Каждые 3 вершины задают треугольник
GL_TRIANGLE_STRIP	3	Группа связанных треугольников. Треугольник с номером *N* опирается на вершины с номерами *N*+2, *N*+1 и *N*
GL_TRIANGLE_FAN	3	Группа связанных треугольников с общей вершиной. Треугольник с номером *N* опирается на вершины с номерами *N*+2, *N*+1 и 1
GL_QUADS	4	4 вершины задают четырехугольник
GL_QUAD_STRIP	4	Группа связанных четырехугольников. Четырехугольник с номером *N* опирается на вершины с номерами 2×*N*-1, 2×*N*, 2×*N*+2 и 2×*N*+1
GL_POLYGON	3	Рисуется выпуклый многоугольник

У группы примитивов могут быть следующие свойства:

- при неактивизированных источниках света команда *glColor* определяет цвет примитивов. При активизированных источниках света цвет примитивов определяется взаимодействием цвета источников света и цвета примитивов;
- команда *glNormal* определяет нормаль к плоскости, к которой принадлежит эта вершина;
- команда *glMaterial* задает свойства материала (цвет диффузионного отражения, рассеянный цвет, степень зеркального отражения, цвет зеркального отражения);
- координаты текстуры устанавливаются командой *glTexCoord*.

9.3. Плоская графика

В папке «Example 36 OpenGL 2D» представлен проект, использующий плоские приметивы OpenGL.

Метод инициализации графики представлен в листинге 9.7.

Листинг 9.7. Инициализация графики

```
public static void InitGL(SimpleOpenGlControl CGL)
{
    Glut.glutInit(); // инициализацию библиотеки Glut
    Glut.glutInitDisplayMode(Glut.GLUT_RGB);
    // цвет очитки окна
    Gl.glClearColor(255, 255, 255, 1);

    // установка порта вывода в соотвествии с CGL
    Gl.glViewport(0, 0, CGL.Width, CGL.Height);

    Gl.glOrtho(0, CGL.Width, 0, CGL.Height, 0, 1);
}
```

В листинге 9.8 приведен пример кода, рисующего 10 точек случайного цвета со случайными координатами (рис. 9.8). Для назначения цвета точек командой *glColor** необходимо отключить источники света командой *glDisable*.

Листинг 9.8. Рисование 10 точек

```
public static void DrawPoint()
{
    if (POINT_SMOOTH)
        Gl.glEnable(Gl.GL_POINT_SMOOTH);
    else
        Gl.glDisable(Gl.GL_POINT_SMOOTH);
    Gl.glEnable(Gl.GL_POINT_SIZE);
    Gl.glEnable(Gl.GL_POINT_SIZE_RANGE);
    Gl.glEnable(Gl.GL_POINT_SIZE_GRANULARITY);

    for (int i = 1; i < 10; i++)
    {
        Gl.glPointSize(3 * i + 1);
        Gl.glBegin(Gl.GL_POINTS);
        Gl.glColor3bv(RGB());
        Gl.glVertex2f(rnd.Next(400), rnd.Next(400));
        Gl.glEnd();
    }
}
```

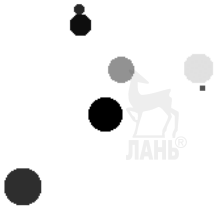

Рис. 9.8. Точки

Точки могут изображаться прямоугольниками, если отключен режим сглаживания командой *glDisable* с параметром *GL_POINT_SMOOTH* или кружками, если этот режим включен. Команда *glPointSize* задает размер точек. В режиме сглаживания размер точек не может быть любым и ограничен величиной 10.

Для задания случайного цвета используется метод

Листинг 9.9. Метод задания случайного цвета

```
static byte[] RGB()
{
    byte[] res = new byte[3];
    res[0] = Convert.ToByte(rnd.Next(256));
    res[1] = Convert.ToByte(rnd.Next(256));
    res[2] = Convert.ToByte(rnd.Next(256));
    return res;
}
```

В листинге 9.10 приведен фрагмент кода, рисующего 10 отрезков со случайными координатами, случайными цветами для каждой вершины (рис. 9.9).

Листинг 9.10. Рисование отрезков случайным цветом

```
public static void DrawLine()
{
    if (LINE_SMOOTH)
        Gl.glEnable(Gl.GL_LINE_SMOOTH);
    else Gl.glDisable(Gl.GL_LINE_SMOOTH);
    Gl.glLineWidth(10);
    Gl.glBegin(Gl.GL_LINES);
    for (int i = 1; i < 20; i++)
    {
        Gl.glColor3bv(RGB());
        Gl.glVertex2f(rnd.Next(400), rnd.Next(400));
    }
}
```

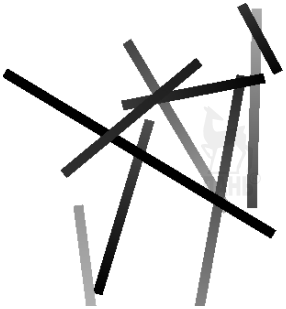

Рис. 9.9. Линии

Отрезки изображаются в виде параллелограмма, малые стороны которого горизонтальны. При включенном режиме сглаживания (устранения ступенчатости) *glEnable(GL_LINE_SMOOTH)* линии рисуются прямоугольниками. Этот режим требует дополнительных временных затрат. Каждой вершине приписывается свой цвет. Поэтому цвета точек отрезка плавно меняются от цвета одного конца отрезка до цвета, заданного на другом конце отрезка.

В следующем примере рисования треугольников (листинг 9.11) плавное изменение цветов, заданных для каждой вершины треугольника, дает интересный визуальный эффект (рис. 9.10).

Листинг 9.11. Рисование треугольников

```
static void DrawTriangles()
{
    Gl.glBegin(Gl.GL_TRIANGLES);
    for (int i = 1; i < 21; i++)
    {
        Gl.glColor3bv(RGB());
        Gl.glVertex2f(rnd.Next(400), rnd.Next(400));
    }
}
```

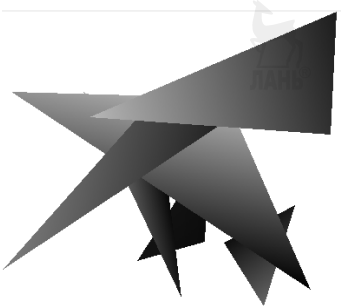

Рис. 9.10. Треугольники

Рисование остальных типов плоских фигур также просто и мы не будем рассматривать эти типы.

9.4. Трехмерная графика

Для демонстрации работы с трехмерными объектами предназначен проект, находящийся в папке «Example 35 OpenGL 3D». Вид проекта представлен на рисунке 9.11.

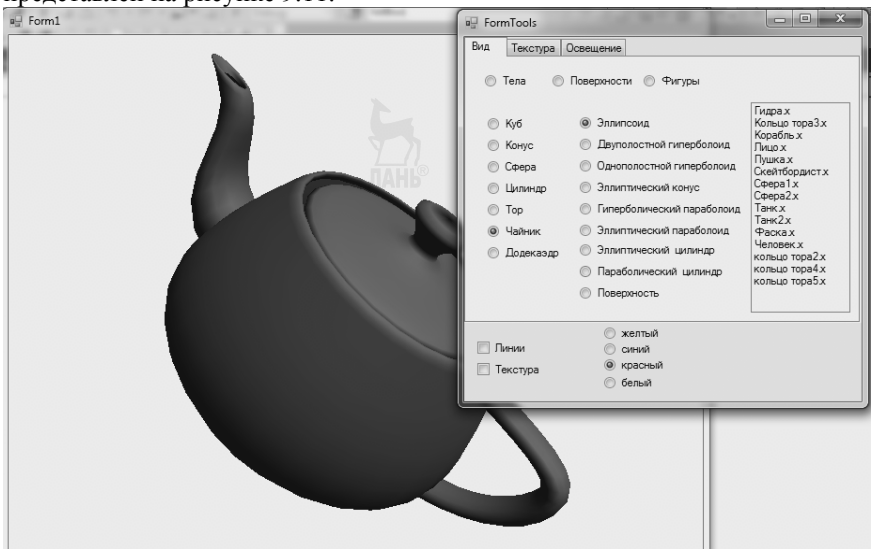

Рис. 9.11. Проект для работы с трехмерными объектами

Проект позволяет рисовать многогранники модуля GLUT, квадратичные поверхности, заданные параметрическими уравнениями, поверхность, заданную таблично, различные фигуры, созданные в других графических пакетах, и накладывать на эти поверхности текстуру из файлов BMP.

9.4.1. Инициализация OpenGL

Прежде чем вызывать один из методов для рисования, необходимо в момент запуска проекта инициализировать библиотеку OpenGL.

Листинг 9.12. Инициализация библиотеки OpenGL

```
public static void InitGL(SimpleOpenGlControl CGL
)
{
    Glut.glutInit();
```

```
    Glut.glutInitDisplayMode(Glut.GLUT_RGB |
       Glut.GLUT_DOUBLE | Glut.GLUT_DEPTH);
    // очитка окна
    Gl.glClearColor(255, 255, 255, 1);
    // установка порта вывода в соотвествии с CGL
    Gl.glViewport(0, 0, CGL.Width, AnT.Height);
    // настройка проекции
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    Glu.gluPerspective(45,(float)CGL.Width/(float)CGL.Height,
        0.1,200);
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
    // настройка параметров OpenGL для визуализации
    Gl.glEnable(Gl.GL_DEPTH_TEST);
}
```

Перед рисованием необходимо настроить параметры OpenGL, то есть установить свойства материала и командой *glLightfv* расставить источники света. Подробный разговор об этих командах пойдет в следующих параграфах. А пока приведем самый простой вариант метода рисования.

Листинг 9.13. Настройка параметров перед рисованием

```
static void initLight()
{
    float[] light_ambient = { 0.0f, 0.0f, 0.0f, 1.0f };
    float[] light_diffuse = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] light_specular = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] light_position = { pos[0], pos[1], pos[2],0.0f};
    // параметры источника света
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_AMBIENT, light_ambient);
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_DIFFUSE, light_diffuse);
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_SPECULAR,
       light_specular);
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_POSITION,
       light_position);
    // включаем освещение и источник света
    Gl.glEnable(Gl.GL_LIGHTING);
    Gl.glEnable(Gl.GL_LIGHT0);
    // включаем z-буфер
    Gl.glEnable(Gl.GL_DEPTH_TEST);
}
```

9.4.2. Многогранники модуля DGLUT

Небольшой набор многогранников представляет сама библиотека OpenGL. Она, в частности, позволяет рисовать сферы, цилиндры и диски. Для рисования этих фигур предназначены следующие команды:

```
gluCylinder;
gluDisk;
gluPartialDisk;
gluSphere;
```

В библиотеке DGLUT.DLL представлен широкий набор многогранников: куб, сфера, конус, тор, октаэдр, икосаэдр, додекаэдр, тетраэдр и чайник. Для изображения этих тел предназначены следующие методы:

- для куба — *DrawBox*, *glutWireCube*, *glutSolidCube*;
- для сферы — *glutWireSphere*, *glutSolidSphere*;
- для конуса — *glutWireCone*, *glutSolidCone*;
- для тора — *glutWireTorus*, *glutSolidTorus*;
- для додекаэдра — *glutWireDodecahedron*, *glutSolidDodecahedron*;
- для октаэдра — *Octaheadron*, *glutWireOctaheadron*, *glutSolidOctaheadron*;
- для икосаэдра — *Icosahedron*, *glutWireIcosahedron*, *glutSolidIcosahedron*
- для тетраэдра — *Tetrahedron*, *glutWireTetrahedron*, *glutSolidTetrahedron*;
- для чайника — *Teapot*, *glutWireTeapot*, *glutSolidTeapot*.

Как и в примере, рассмотренном в предыдущем параграфе, существуют методы двух типов: *Wire* и *Solid*. Если параметр режима рисования принимает значение *GL_LINE_LOOP*, то реализуется рисование ребрами. Если же параметр режима рисования принимает значение *GL_TRIANGLES* или *GL_QUADS*, то грани рисуются треугольниками или четырехугольниками.

Наиболее сложной фигурой является чайник, изображение которого представлено на рисунке 9.12.

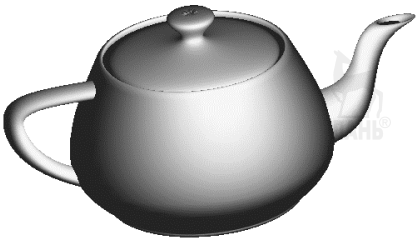

Рис. 9.12. Чайник

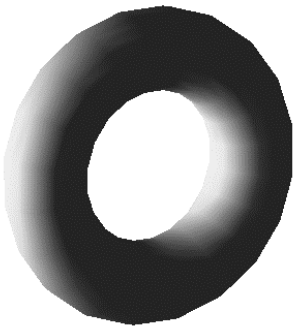

Рис. 9.13. Тор

Метод рисования будет выглядеть так:

Листинг 9.14. Метод рисования

```
public static void Draw()
{
    if (flTexture)
       Gl.glTexParameteri(Gl.GL_TEXTURE_2D,
           Gl.GL_TEXTURE_MIN_FILTER, Gl.GL_NEAREST);
    initLight();
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
        Gl.GL_DEPTH_BUFFER_BIT);
    Gl.glLoadIdentity();
    Gl.glPushMatrix();
    Gl.glTranslated(0, 0, zoom);
    Gl.glRotated(alf, 1, 0, 0);
    Gl.glRotated(bet, 0, 1, 0);
    Gl.glRotated(alf + bet, 0, 0, 1);

    switch (flView)
    {
       case 0: // тела
           if (G.flLine)
              Wire();
           else
              Solid();
           break;
       case 1:  // поверхности
           if (Id_FaceType != 8)
              DrawFaces(Id_FaceType);
           else
              WireFigure();
           break;
       case 2:  // фигуры
           WireFigure();
           break;
    }
    Gl.glPopMatrix();
    Gl.glFlush();
    Program.formMain.CGL.Invalidate();
}
```

Если использовать готовые фигуры библиотеки GLUT, то метод их рисования будет выглядеть так:

Листинг 9.15. Метод рисования готовых фигур

```
        static void Solid()
        {
```

```
            switch (G.flTools)
            {
                case 0:
                    Glut.glutSolidCube(2);
                    break;
                case 1:
                    Glut.glutSolidCone(2, 2, 32, 32);
                    break;
                case 2:
                    Glut.glutSolidSphere(2, 32, 32);
                    break;
                case 3:
                    Glut.glutSolidCylinder(2, 2, 32, 32);
                    break;
                case 4:
                    Glut.glutSolidTorus(0.5, 1.5, 32, 32);
                    break;
                case 5:
                    Glut.glutSolidTeapot(2);
                    break;
                case 6:
                    Glut.glutSolidDodecahedron();
                    break;
            }
        }
```

Задачу построения многогранников рассмотрим на примере тетраэдра (рис. 9.14).

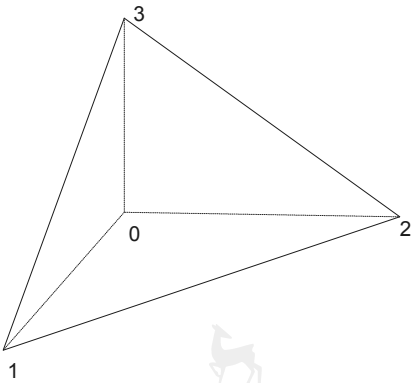

Рис. 9.14. Тетраэдр

Для определения параметров тетраэдра потребуются вершины, грани и нормали к граням.

Координаты четырех вершин заданы массивом *V0*:

```
float[,] V0 = ((1,0,0),(0,0,0),(0,1,0),(0,0,1));
```

Для задания номеров точек, на которые опираются четыре треугольные грани тетраэдра, в модуле DGLUT введен двухмерный массив граней `TetFaces`:

```
int[,] TetFaces = ((0,1,2),(0,3,1),(3,2,1),(2,3,0));
```

Наконец, для задания четырех нормалей к граням тетраэдра введен массив `TetPoints`:

```
float[,] TetPoints = ((0,0,-1),(0,-1,0),(-1,0,0),(1,1,1));
```

Задание последовательности команд построения тетраэдра для OpenGL представлена в методе *TetraBox* (листинг 9.16).

Листинг 9.16. Построение тетраэдра

```
void TetraBox(float Size, int DrawType)
{
  float[,] V = new float[4,3];
  for (int i=0; i<4; i++)
    for (int j=0; j<3; j++)
       V[i,j]=Size*V0[i,j];
  for (int i=0; i<4; i++)
  {
    Gl.glBegin(DrawType);
      Gl.glNormal3fv(TetPoints[i,0]);
      Gl.glVertex3fv(V[TetFaces[i,0],0]);
      Gl.glVertex3fv(V[TetFaces[i,1],0]);
      Gl.glVertex3fv(V[TetFaces[i,2],0]);
    Gl.glEnd();
  }
}
```

В этом методе после масштабирования координат вершин циклом определяются четыре грани. Описание каждой грани начинается командой *glBegin(DrawType)*. Параметр *DrawType* может принимать два значения: *GL_LINE_LOOP* (рисовать ребра) или *GL_TRIANGLES* (рисовать треугольники). Для каждой грани командой *glNormal3fv* задается нормаль, командами *glVertex3fv* – три вершины.

Для рисования тетраэдра ребрами и гранями предназначены два метода (листинг 9.17).

Листинг 9.17. Рисование тетраэдра ребрами и гранями

```
void glutWireTetra(float Size)
{
  TetraBox(Size, GL_LINE_LOOP);
}

void glutSolidTetra(float Size)
{
  TetraBox(Size, GL_TRIANGLES);
}
```

9.4.3. Списки команд

Другой способ рисования заключается в следующем: в начале работы программы командой *glNewList...glEndList* подготавливается список или несколько списков команд, в которых перечислены команды-примитивы (листинг 9.18). Рисование может быть вызвано в любой момент, пока существует список, командой *glCallList(N)* с указанием номера списка.

В листинге создается список с номером 1, задающий две пересекающиеся фигуры: куб и тетраэдр. Во время рисования будет показана новая фигура, получающаяся объединением куба и тетраэдра (рис. 9.15).

Листинг 9.18. Подготовка списка примитивов

```
public static void MyDrawBoxList(float Size, int DrawType)
{
    float[][] V3 = new float[4][];
    for (int i = 0; i < 4; i++)
       V3[i] = new float[3];

    float[][] V = new float[8][];
    for (int i = 0; i < 8; i++)
    {
       V[i] = new float[3];
       for (int j = 0; j < 3; j++)
           V[i][j] = Size * BoxVertex[i][j] / 2;
    }
    V3[0][0] = Size; V3[1][0] = 0; V3[2][0] = 0; V3[3][0]= 0;
    V3[0][1] = 0; V3[1][1] = 0; V3[2][1] = Size; V3[3][1]= 0;
    V3[0][2] = 0; V3[3][2] = Size; V3[1][2] = 0; V3[2][2]= 0;

    Gl.glColor3f(1.0f, 1.0f, 0.0f);
    Gl.glNewList(1, Gl.GL_COMPILE);
    for (int i = 0; i < 6; i++)
    {
       Gl.glNormal3fv(BoxPoints[i]);
```

```
            Gl.glBegin(Gl.GL_LINE_LOOP);
               for (int j = 0; j < 4; j++)
                  Gl.glVertex3fv(V[BoxFaces[i][j]]);
            Gl.glEnd();
         }

         Gl.glColor3f(0.0f, 1.0f, 0.0f);
         for (int i = 0; i < 4; i++)
         {
            Gl.glBegin(Gl.GL_LINE_LOOP);
               Gl.glNormal3fv(TetPoints[i]);
               Gl.glVertex3fv(V3[TetFaces[i, 0]]);
               Gl.glVertex3fv(V3[TetFaces[i, 1]]);
               Gl.glVertex3fv(V3[TetFaces[i, 2]]);
            Gl.glEnd();
         }
         Gl.glEndList();

         Gl.glColor3f(1.0f, 1.0f, 0.0f);
         Gl.glNewList(2, Gl.GL_COMPILE);
         for (int i = 0; i < 6; i++)
         {
            Gl.glNormal3fv(BoxPoints[i]);
            Gl.glBegin(Gl.GL_QUADS);
               for (int j = 0; j < 4; j++)
                  Gl.glVertex3fv(V[BoxFaces[i][j]]);
            Gl.glEnd();
         }

         Gl.glColor3f(0.0f, 1.0f, 0.0f);
         for (int i = 0; i < 4; i++)
         {
            Gl.glBegin(Gl.GL_TRIANGLES);
               Gl.glNormal3fv(TetPoints[i]);
               Gl.glVertex3fv(V3[TetFaces[i, 0]]);
               Gl.glVertex3fv(V3[TetFaces[i, 1]]);
               Gl.glVertex3fv(V3[TetFaces[i, 2]]);
            Gl.glEnd();
         }
         Gl.glEndList();
}
```

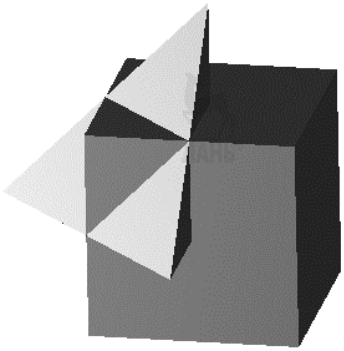

Рис. 9.15. Объединение куба и тетраэдра

9.4.4. Изображение квадратичных поверхностей

В проекте возможно построение восьми квадратичных поверхностей, заданных параметрически. Уравнения этих поверхностей задает метод:

Листинг 9.19. Уравнения квадратичных поверхностей

```
public static void XYZ(int n, float t1, float t2,out float x,
     out float y, out float z)
{
    float a, b, c;
    x = 0; y = 0; z = 0;
    a = size[0]; b = size[1]; c = size[2];
    switch (n)
    {
       case 0:
         x = a * (float)Math.Sin(t1) * (float)Math.Sin(t2);
         y = b * (float)Math.Cos(t1);
         z = c * (float)Math.Sin(t1) * (float)Math.Cos(t2);
         break;
       …
       case 7:     //
         x = a * t2;
         y = b * t1;
         z = c * t2 * t2;
         break;
     }
}
```

Одна из поверхностей – однополостной гиперболоид – изображена на рисунке 9.16.

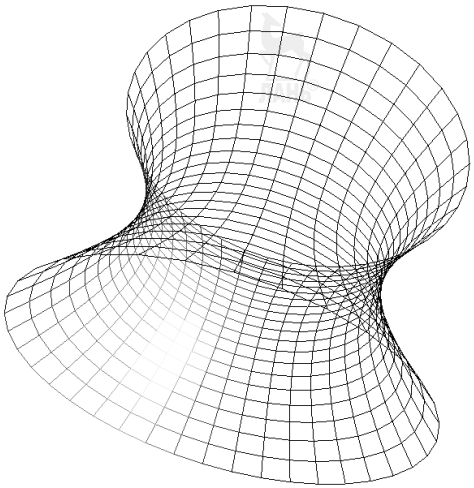

Рис. 9.16. Однополостной гиперболоид

Последовательность действий при рисовании поверхностей представлена в листинге 9.20.

Листинг 9.20. Рисование квадратичных поверхностей

```
static void DrawFaces(int k)
{
    float h1, h2, t1, t2;
    float[] x0 = new float[3];
    float[] x1 = new float[3];
    float[] x2 = new float[3];
    float[] x3 = new float[3];
    float[] N = new float[3];
    if (k != 8)
    {
        int DrawType;
        if (flLine)
            DrawType = Gl.GL_LINE_LOOP;
        else
            DrawType = Gl.GL_QUADS;
        h1 = (a2 - a1) / (n2 - n1);
        h2 = (b2 - b1) / (m2 - m1);
        for (int j = n1; j < n2; j++)
           for (int i = m1; i < m2; i++)
           {
               t1 = h1 * j; t2 = h2 * i;
               XYZ(k,t1, t2, out x0[0],out x0[1], out x0[2]);
```

```
                t1 = h1 * j; t2 = h2 * (i + 1);
                XYZ(k,t1,t2, out x1[0], out x1[1], out x1[2]);
                t1 = h1 * (j + 1); t2 = h2 * (i + 1);
                XYZ(k,t1,t2, out x2[0], out x2[1], out x2[2]);
                t1 = h1 * (j + 1); t2 = h2 * i;
                XYZ(k,t1,t2, out x3[0], out x3[1], out x3[2]);
                N = getNormal(k, x0, x1, x2, t1, t2);
                Gl.glBegin(DrawType);
                Gl.glNormal3fv(N);
                Gl.glTexCoord2d(h1 * (j + 0 - n1) / (a2 - a1),
                    h2 * (i + 0 - m1) / (b2 - b1));
                Gl.glVertex3fv(x0);
                Gl.glTexCoord2d(h1 * (j + 0 - n1) / (a2 - a1),
                    h2 * (i + 1 - m1) / (b2 - b1));
                Gl.glVertex3fv(x1);
                Gl.glTexCoord2d(h1 * (j + 1 - n1) / (a2 - a1),
                    h2 * (i + 1 - m1) / (b2 - b1));
                Gl.glVertex3fv(x2);
                Gl.glTexCoord2d(h1 * (j + 1 - n1) / (a2 - a1),
                    h2 * (i + 0 - m1) / (b2 - b1));
                Gl.glVertex3fv(x3);
                Gl.glEnd();
                if (sg != 0)
                {
                    x0[1]=-x0[1]; x1[1]=-x1[1]; x2[1]= -x2[1];
                    x3[1] = -x3[1];
                    N = getNormal(k, x0, x1, x2, t1, t2);
                    Gl.glBegin(DrawType);
                    Gl.glNormal3fv(N);
                    Gl.glTexCoord2d(h1*(j - n1) / (a2 - a1),
                        h2 * (i + 0 - m1) / (b2 - b1));
                    Gl.glVertex3fv(x0);
                    Gl.glTexCoord2d(h1 * (j - n1) / (a2 - a1),
                        h2 * (i + 1 - m1) / (b2 - b1));
                    Gl.glVertex3fv(x1);
                    Gl.glTexCoord2d(h1*(j + 1 - n1)/(a2 - a1),
                        h2 * (i + 1 - m1) / (b2 - b1));
                    Gl.glVertex3fv(x2);
                    Gl.glTexCoord2d(h1*(j+1 - n1) / (a2 - a1),
                        h2 * (i + 0 - m1) / (b2 - b1));
                    Gl.glVertex3fv(x3);
                    Gl.glEnd();
                }
            }
        }
}
```

По прямоугольной сетке *n1*×*m1* вычисляются координаты четырех точек и строится четурехугольник. Многие поверхности двухсвязны (это

показывает переменная *sg*, принимающая отрицательное значение): вторая половина получается изменением знака у координаты *y*.

9.4.5. Изображение поверхности, заданной таблично

В геоинформационных системах часто возникает проблема изображения участка земной поверхности или слоя породы. Мы рассмотрим случай, когда на прямоугольном участке *(Gx1,Gx2)×(Gy1,Gy2)* в узлах регулярной сетки заданы значения *float V[i,j]*. По оси *X* прямоугольник разбит на *nCol* частей, а по оси *Y* – на *nRow* частей. Такого типа данные представлены, например, в файле XYZ.txt. Для чтения файла предназначен метод *ReadXYZ()*.

Листинг 9.21. Чтение табличных данных

```
public static void ReadXYZ(string FileName)
{
    FileStream aFile =
        new FileStream(FileName, FileMode.Open);
    StreamReader f = new StreamReader(aFile);

    string s = f.ReadLine();
    char[] sep = { ' ', ',', ';', '\t' };
    string[] st;
    float[,] V;


    s = f.ReadLine();
    st = s.Split(sep,
        StringSplitOptions.RemoveEmptyEntries);
    int nCol = Convert.ToInt32(st[0]);
    int nRow = Convert.ToInt32(st[1]);
    V = new float[nCol, nRow];

    float Gx1, Gx2, Gy1, Gy2, Gz1, Gz2;

    s = f.ReadLine();
    st = s.Split(sep,
        StringSplitOptions.RemoveEmptyEntries);
    Gx1 = Convert.ToSingle(Str(st[0]));
    Gx2 = Convert.ToSingle(Str(st[1]));

    s = f.ReadLine();
    st = s.Split(sep,
       StringSplitOptions.RemoveEmptyEntries);
    Gy1 = Convert.ToSingle(Str(st[0]));
    Gy2 = Convert.ToSingle(Str(st[1]));
```

```
s = f.ReadLine();
st = s.Split(sep,
    StringSplitOptions.RemoveEmptyEntries);
Gz1 = Convert.ToSingle(Str(st[0]));
Gz2 = Convert.ToSingle(Str(st[1]));

for (int j = 0; j < nRow; j++)
{
    int i = 0;
    while (i < nCol)
    {
        s = f.ReadLine();
        st = s.Split(sep,
        StringSplitOptions.RemoveEmptyEntries);
        int Ls = st.Length;
        for (int k = 0; k < Ls; k++)
        {
            V[i + k, j] =
               Convert.ToSingle(Str(st[k]));
            if (V[i + k, j] < Gz1)
               Gz1 = V[i + k, j];
            if (V[i + k, j] > Gz2)
               Gz2 = V[i + k, j];
        }
        i += Ls;
    }
}
f.Close();
aFile.Close();

float[] size = { 1f, 1.6f, 1.6f };

Zmin=(Gz1-(Gz2+Gz1)/2)/(Gz2 - Gz1) * size[0];
Zmax=(Gz2-(Gz2+Gz1)/2)/(Gz2 - Gz1) * size[0];
const float d = 4f;
float h1 = d / nRow;
float h2 = d / nCol;
int Lp = nRow * nCol;
TMyBody.Points = new List<TPoints>();
TMyBody.Texture = new List<TPoints2>();
TPoints p;
TPoints2 q;

for (int j = 0; j < nRow; j++)
    for (int i = 0; i < nCol; i++)
    {
        p = new TPoints();
```

```
            p.x = -d / 2 + j * h1;
            p.y = -d / 2 + i * h1;
            p.z = (V[i, j] - (Gz2 + Gz1) / 2) /
                (Gz2 - Gz1) * size[0];
            TMyBody.Points.Add(p);
            q = new TPoints2();
            q.x = 1f * j / nRow;
            q.y = 1f * i / nRow;
            TMyBody.Texture.Add(q);
        }

    int L = 2 * (nRow - 1) * (nCol - 1);
    TMyBody.Faces = new List<TFace>();
    TFace face;
    for (int j = 0; j < nRow - 1; j++)
        for (int i = 0; i < nCol - 1; i++)
        {
            face = new TFace();
            face.Vertex[0] = j * nCol + i;
            face.Vertex[1] = j * nCol + i + 1;
            face.Vertex[2] = (j + 1) * nCol + i + 1;

            face.Norm = getNormalP(
                TMyBody.Points[face.Vertex[0]],
                TMyBody.Points[face.Vertex[1]],
                TMyBody.Points[face.Vertex[2]]);
            TMyBody.Faces.Add(face);
        }

    for (int j = 0; j < nRow - 1; j++)
        for (int i = 0; i < nCol - 1; i++)
        {
            face = new TFace();
            face.Vertex[0] = j * nCol + i;
            face.Vertex[1] = (j + 1) * nCol + i + 1;
            face.Vertex[2] = (j + 1) * nCol + i;
            face.Norm =
                getNormalP(
                TMyBody.Points[face.Vertex[0]],
                TMyBody.Points[face.Vertex[1]],
                TMyBody.Points[face.Vertex[2]]);
            TMyBody.Faces.Add(face);
        }

    TMyBody.Norms = new List<TPoints>();
    for (int i = 0; i < Lp; i++)
    {
        p = new TPoints();
```

```
            p.x = 0;
            p.y = 0;
            p.z = 0;
            for (int j = 0; j < L; j++)
                if ((i == TMyBody.Faces[j].Vertex[0]) ||
                    (i == TMyBody.Faces[j].Vertex[1]) ||
                    (i == TMyBody.Faces[j].Vertex[2]))
                    p = SumPoint(p,
                        TMyBody.Faces[j].Norm);
            p = NormaTo1(p);
            TMyBody.Norms.Add(p);
        }
    }
```

Поверхность будем задавать треугольниками, 2×(*nCol*-1)× (*nRow*-1) треугольников задаются в методе *WireFigure()*.

Листинг 9.22. Задание треугольников поверхности

```
    static void WireFigure()
    {
        if (TMyBody.Faces == null) return;

        int NumFace = TMyBody.Faces.Count;
        if (NumFace > 0)
        {
            int DrawType;
            if (flLine)
                DrawType = Gl.GL_LINE_LOOP;
            else
                DrawType = Gl.GL_TRIANGLES;
            int n = 0;
            for (int i = 0; i < NumFace; i++)
            {
                Gl.glBegin(DrawType);
                n = TMyBody.Faces[i].Vertex[0];
                Gl.glNormal3f(TMyBody.Norms[n].x,
                   TMyBody.Norms[n].y,
                   TMyBody.Norms[n].z);
                Gl.glTexCoord2d(TMyBody.Texture[n].x,
                   TMyBody.Texture[n].y);
                Gl.glVertex3f(TMyBody.Points[n].x,
                   TMyBody.Points[n].y,
                   TMyBody.Points[n].z);

                n = TMyBody.Faces[i].Vertex[1];
                Gl.glNormal3f(TMyBody.Norms[n].x,
                   TMyBody.Norms[n].y,
```

```
                    TMyBody.Norms[n].z);
                Gl.glTexCoord2d(TMyBody.Texture[n].x,
                    TMyBody.Texture[n].y);
                Gl.glVertex3f(TMyBody.Points[n].x,
                    TMyBody.Points[n].y,
                    TMyBody.Points[n].z);

                n = TMyBody.Faces[i].Vertex[2];
                Gl.glNormal3f(TMyBody.Norms[n].x,
                    TMyBody.Norms[n].y,
                    TMyBody.Norms[n].z);
                Gl.glTexCoord2d(TMyBody.Texture[n].x,
                    TMyBody.Texture[n].y);
                Gl.glVertex3f(TMyBody.Points[n].x,
                    TMyBody.Points[n].y,
                    TMyBody.Points[n].z);
                Gl.glEnd();
            }
        }
    }
```

Поверхность, заданная в файле *XYZ.txt*, изображена на рисунке 9.17.

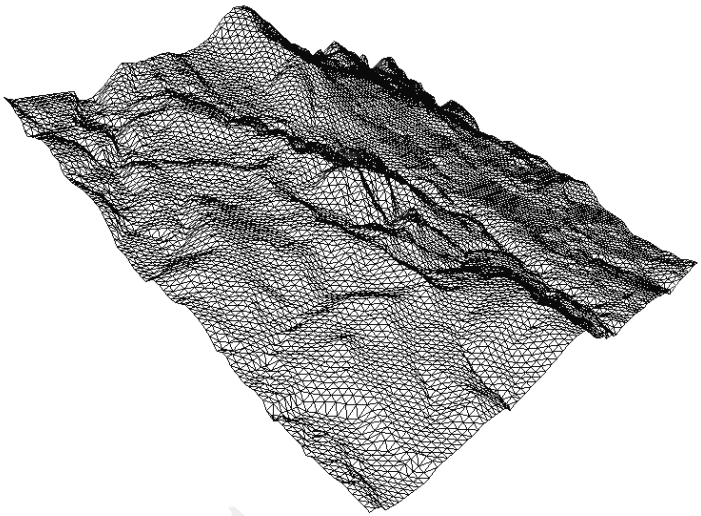

Рис. 9.17. Поверхность, заданная таблично

9.5. Геометрические преобразования

Основная задача, возникающая при работе с трехмерной графикой, сводится к следующим действиям по преобразованию координат:

– Шаг 1. Необходимо определить трехмерный параллелепипед в мировом пространстве и с помощью видовой матрицы $M_{видовая}$ поворотами и переносами преобразовать координаты объекта в видовые координаты в локальной системе координат, связанной с объектом;
– Шаг 2. С помощью матрицы проекций $M_{проекций}$ преобразовать видовые координаты в усеченные координаты;
– Шаг 3. Усеченные координаты учетом перспективы преобразовать в нормализованные координаты;
– Шаг 4. И, наконец, нормализованные координаты преобразованием области вывода перевести в экранные координаты.

В OpenGL используется три типа матриц 4×4: видовые матрицы $M_{видовая}$; матрица проекций $M_{проекций}$ и матрицы текстуры $M_{текстуры}$. Для работы с матрицей, используя команду `glMatrixMode(mode: GLenum)`, необходимо сделать ее текущей.

Параметр `mode` может принимать следующие значения:
– *GL_MODELVIEW* — матрица видовых преобразований;
– *GL_PROJECTION* — матрица проекций;
– *GL_TEXTURE* — матрица текстуры.

Загрузить матрицу можно 3 способами:
– командой *glLoadIdentity* задать единичную матрицу;
– командой *glLoadMatrix(M: PGLType)* загрузить матрицу *M*;
– задать матрицу командами переноса *glTranslate* и вращения *glRotate*.

Команда *glMultMatrix(M: PGLType)* предназначена для перемножения текущей матрицы *T* на матрицу *M*.

Ось *X* расположена в плоскости экрана и направлена вправо, ось *Y* расположена в плоскости экрана и смотрит вверх. Ось *Z* направлена на наблюдателя (СК – правосторонняя) или от него (СК – левосторонняя).

Координаты точек любого объекта описываются в мировой правосторонней системе координат. Перед выводом изображения необходимо преобразовать эти координаты к видовой левосторонней СК.

9.5.1. Преобразование области вывода

Область вывода на экране задается командой
`glViewPort(Left, Top, Width, Height: GLInt)`
которая определяет преобразование нормализованных координат в координаты окна на экране.

9.5.2. Проецирование

В OpenGL есть два вида проекций: ортографическая и перспективная с 1 точкой схода.

Ортографическая проекция определяется командой:

```
glOrtho(left, right, bottom, top, near, far: GLdouble)
```

Параметры *left*, *right*, *bottom*, *top*, *near*, *far* задают усеченную пирамиду видимости.

Эта команда создает матрицу проектирования *M*:

$$M = \begin{Vmatrix} 2/(right-left) & 0 & 0 & t_x \\ 0 & 2/(top-bottom) & 0 & t_y \\ 0 & 0 & -2/(far-near) & t_z \\ 0 & 0 & 0 & 1 \end{Vmatrix},$$

где $t_x = -\dfrac{right+left}{right-lef}$; $t_y = -\dfrac{top+bottom}{top-bottom}$; $t_z = -\dfrac{far+near}{far-near}$.

При малых значениях параметров *near* и *far*: если во время выполнения тело не попадает в заданную усеченную пирамиду, то возникает ошибка.

Для перспективного проецирования необходимо задать усеченную пирамиду видимости (рис. 9.18).

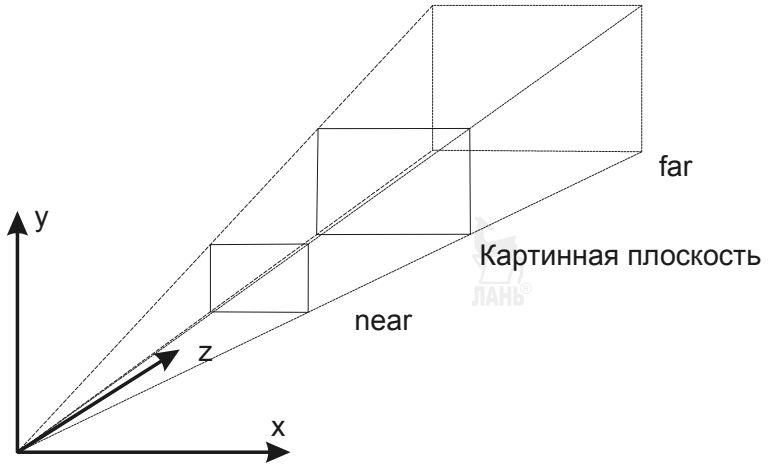

Рис. 9.18. Усеченная пирамида видимости

Задать усеченную пирамиду можно командами *glFrustum* или *gluPerspective*. Команда *glFrustum* явно задает параметры пирамиды:

```
glFrustum (left, right, bottom, top, near, far: GLdouble)
```

Параметры *left, right, bottom, top* определяют размеры ближней секущей плоскости, а оба расстояния до секущих плоскостей должны быть положительными. Команда создает следующую матрицу преобразования:

$$M = \begin{Vmatrix} \frac{2*near}{right-left} & 0 & A & 0 \\ 0 & \frac{2*near}{top-bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{Vmatrix},$$

где $A = \frac{right+left}{right-lef}$; $B = \frac{top+bottom}{top-bottom}$; $C = -\frac{far+near}{far-near}$; $D = \frac{2*far*near}{far-near}$.

Команда *gluPerspective(angley, aspect, near, far: GLdouble)* задает угол видимости в градусах *angley*, отношение сторон aspect и расстояния до плоскостей отсечения. Эта команда эквивалентна команде *glFrustum* при *Left = -right, bottom = -top, tg(angley/2) = top/nead, aspect = right/top*.

В примере, приведенном ниже, устанавливается ортографическая проекция в окно с верхним левым углом (0,0), шириной *Width* и высотой *Height*:

```
glMatrixMode(gl_Projection);
glLoadIdentity;
glOrtho(0,0,Width,Height,-2,2);
```

Замечание. Перед установкой любой матрицы необходимо сбросить ее в единичную командой glLoadIdentity, так как следующая команда умножает ее на текущую матрицу.

9.5.3. Преобразование от мировой системы координат к видовой (локальной)

Определив матрицу проекций, необходимо сделать текущей видовую матрицу:

```
glMatrixMode(gl_ModelView);
```

Есть несколько команд для переноса, вращения и масштабирования системы координат:

– *glTranslate[f d](x,y,z: GLtype)* — переносит объект на вектор (*x,y,z*);
– *glRotate[f d](angle,x,y,z: GLtype)* — поворачивает объект на угол *angle* вокруг вектора (*x,y,z*).
– *glScale[f d](x,y,z: GLtype)* — масштабирует объект с коэффициентами *x,y,z*.

В листинге 9.23 приведен пример рисования с использованием видовой матрицы.

Листинг 9.23. Рисование с использованием видовой матрицы

```
public static void Move()
{
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    // текущая матрица - видовая
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
        Gl.GL_DEPTH_BUFFER_BIT);
    Gl.glPushMatrix(); // запомнили текущую матрицу
    // поворот вокруг точки x0[0..2]
    Gl.glTranslatef(0.2f, 0.2f, 0.2f);
    Gl.glRotatef(45, 1.0f, 0.0f, 0.0f);
    Gl.glRotatef(145, 0.0f, 1.0f, 0.0f);
    Gl.glRotatef(145, 0.0f, 0.0f, 1.0f);
    Gl.glTranslatef(-0.2f, -0.2f, -0.2f);
    // рисование
    Draw();
    Gl.glPopMatrix();  // восстановили матрицу
    // завершение рисования
    Gl.glFlush();
}
```

9.6. Цвет, освещение, свойства материала

Освещенность поверхностей зависит от свойств:
– диффузионного отражения;
– зеркального отражения;
– диффузионного пропускания света;
– направленного пропускания света;
– вектора нормали;
– свойства материала;
– параметров источников света;
– параметров освещения.

9.6.1. Цвет

В библиотеке OpenGL возможно использование двух режимов установки цветов: индексного (цвета определяются из палитры по индексу) и ARGB — задание значений долей красного, зеленого и синего цветов.

Для работы в режиме RGBA предоставлены две команды:

```
glColor[3 4][b s i f d](components: GLtype)
glColor[3 4][b s i f d]v(components: GLtype).
```

Назначенный цвет действует на все создаваемые после этого примитивы. Так, при рисовании треугольника можно задать одинаковый цвет для всех вершин (листинг 9.24).

Листинг 9.24. Задание однородного цвета фигуры

```
  glColor3f(0.0,0.0,1.0);
  glBegin(GL_TRIANGLES);
    glVertex2f(100,100);
    glVertex2f(100,200);
    glVertex2f(200,200);
  glEND;
  glFinish;
```

или в режиме плавного изменения цветов *glEnable(GL_SMOOTH)* назначить для вершин разные цвета (листинг 9.25).

Листинг 9.25. Задание неоднородного цвета фигуры

```
  glEnable(GL_SMOOTH);
  glBegin(GL_TRIANGLES);
    glColor3f(0.0,0.0,1.0);
    glVertex2f(100,100);
    glColor3f(0.0,1.0,0.0);
    glVertex2f(100,200);
    glColor3f(1.0,0.0,0.0);
    glVertex2f(200,200);
  glEND;
  glFinish;
```

Этот прием можно использовать для сглаживания углов между гранями многогранника, определяя для вершин цвета в соответствии с нормалями к поверхности гладкого тела, а не к грани.

9.6.2. Нормали

Нормаль к поверхности определяет направление пропускания света и направление зеркального отражения. Нормаль можно приписывать каждой вершине. Для определения ее координат предусмотрены две команды:

```
glNormal3[b s i f d](coords: type)
glNormal3[b s i f d]v(coords: type)
```

Целочисленные аргументы приводятся в диапазон [0...1].

9.6.3. Свойства материала

Материал может обладать рассеянным и диффузионным цветом, цветом зеркального отражения и излучаемым светом.

Для определения этих свойств предназначены две функции:

```
glMaterial[i f](face, pname: Glenum; param: GLtype)
glMaterial[i f]v(face, pname: Glenum; params: PGLtype)
```

В листинге 9.26 приведен пример установки параметров материала.

Листинг 9.26. Установка параметров материала

```
// красный
// - цвет диффузионного отражения материала
        static float[] mat3_dif = { 0.9f, 0.2f, 0.0f };
// - рассеянный цвет материала
        static float[] mat3_amb = { 0.2f, 0.2f, 0.2f };
// - цвет зеркального отражения материала
        static float[] mat3_spec = { 0.6f, 0.6f, 0.6f };
// - степень зеркального отражения
        static float mat3_shininess = 1.9f * 128;
{
// фоновый цвет материала
Gl.glMaterialfv(Gl.GL_FRONT_AND_BACK, Gl.GL_AMBIENT,
mat3_amb);
// диффузионные свойства
Gl.glMaterialfv(Gl.GL_FRONT_AND_BACK, Gl.GL_DIFFUSE,
mat3_dif);
// зеркальные свойства
Gl.glMaterialfv(Gl.GL_FRONT_AND_BACK, Gl.GL_SPECULAR,
mat3_spec);
Gl.glMaterialf(Gl.GL_FRONT_AND_BACK, Gl.GL_SHININESS,
    mat3_shininess);
}
```

9.6.4. ИСТОЧНИКИ СВЕТА

Каждый точечный источник света задается командами:

```
GlLight[i f](light, pname: GLenum; param: GLfloat)
GlLight[i f]v(light, pname: GLenum; params: PGLfloat)
```

Аргумент *light* задает номер источника света, который может меняться в диапазоне от 0 до *GL_MAX_LIGTS* = 8. Аргумент *pname* определяет устанавливаемый параметр. Возможные значения *pname* приведены в таблице 9.4.

Таблица 9.4

Значение *params*

pname	**Значения params**
GL_SPOT_EXPONENT	Целое [0...128] или вещественное значение, задающее распределение интенсивности света. Чем больше параметр, тем сильнее сфокусирован источник света
GL_SPOT_CUTOFF	Целое [0...90,180] или вещественное значение, задающее максимальный угол разброса света
GL_CONSTANT_ATTENUATION *GL_LINEAR_ATTENUATION* *GL_QUADRATIC_ATTENUATION*	По одному целому или вещественному положительному значению, задающих факторы постоянного, линейного и квадратичного ослабления света

Для команды *GlLight[i f]v* добавляются еще 5 значений параметра *pname* (табл. 9.5).

Таблица 9.5

Значение *params*

pname	**Значения params**
GL_AMBIENT	4 целых или вещественных значения *RGBA*, определяющих интенсивность фонового освещения
GL_DIFFUSE	4 целых или вещественных значения *RGBA*, определяющих интенсивность диффузионного освещения
GL_SPECULAR	4 целых или вещественных значения *RGBA*, определяющих интенсивность зеркального освещения
GL_POSITION	4 целых или вещественных значения, определяющих положение источника света
GL_SPOT_DIRECTION	4 целых или вещественных значения, определяющих вектор направления света

Если параметры *i*-го источника света заданы, то их можно включать и выключать командами *glEnable(GL_LIGHTi)* и *glDisable(GL_LIGHTi)*.

Листинг 9.27. Установка одного источника света

```
static void initLight()
{
    float[] light_ambient = { 0.0f, 0.0f, 0.0f, 1.0f };
    float[] light_diffuse = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] light_specular = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] light_position = { pos[0],pos[1], pos[2], 0.0f};
    // параметры источника света
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_AMBIENT, light_ambient);
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_DIFFUSE, light_diffuse);
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_SPECULAR,
       light_specular);
    Gl.glLightfv(Gl.GL_LIGHT0, Gl.GL_POSITION,
       light_position);
    // включаем освещение и источник света
    Gl.glEnable(Gl.GL_LIGHTING);
    Gl.glEnable(Gl.GL_LIGHT0);
    // включаем z-буфер
    Gl.glEnable(Gl.GL_DEPTH_TEST);
}
```

В OpenGL есть еще одна команда, определяющая модель освещения.

```
GlLightModel[i f](pname: Glenum; param: GLenum)
GlLightModel[i f]v(pname: Glenum; params: PGLenum)
```

Аргумент определяет значения параметров модели освещения (табл. 9.6).

Таблица 9.6

Значение *params*

pname	**Значения param**
GL_LIGHT_MODEL_LOCAL_VIEWER	Одно логическое значение, определяющее положение наблюдателя. При *false* направление обзора всегда направлено вдоль оси *Z*. При *true* наблюдатель находится в начале системы координат
GL_LIGHT_MODEL_TWO_SIDE	Одно логическое значение, определяющее одно- (*false*) или двухстороннее (*true*) освещение граней
GL_LIGHT_MODEL_AMBIENT	4 целых или вещественных значения, определяющих интенсивность фонового освещения

9.7. Текстура

Текстура может быть одномерной или двумерной. Двумерная текстура накладывает на грани прямоугольные изображения. Ширина и высота таких изображений должны быть степенью двойки.

Наложение двумерной текстуры состоит из следующих пунктов:

- вершинам грани необходимо назначить точки изображения;
- методами *glTexImage1d* или *glTexImage2d* создать карту текстуры, передавая ей массив долей цветов *RGB* из изображения;
- задать параметры методом *glTexParameter[f|i|fv|iv]*;
- командой *glEnable(GL_TEXTURE_1|2D)* включить режим текстуры.

9.7.1. Назначение точки карты текстуры вершине

Назначение точки карты текстуры вершине реализуется командой *glTexCoord2d*. Например, вершине x назначается точка карты *s,t*:

```
glTexCoord2d (s,t);
glVertex3fv(x)
```

Параметры *t* и *s* меняются в диапазоне [0,1] и накрывают всю карту текстуры.

Замечание. Некоторые многогранники модуля OpenGL (gluSphere, gluCylinder и glutSolidTeapot) сопоставляют свои вершины точкам карты текстуры.

Замечание. Для двумерной текстуры размеры массива образа должны быть степенью двойки

9.7.2. Создание двумерной карты текстуры

Пусть изображение для текстуры имеет ширину *Width* и высоту *Height*, а информация о долях цветов *RGB* каждого пиксела находится в элементах динамического массива *pixels*.

После заполнения массива *pixels* методом *glTexImage2d* устанавливается двумерная текстура. У метода *glTexImage2d* имеются следующие параметры.

Листинг 9.28. Параметры метода установления текстуры из изображения

```
glTexImage2d(
  target: GLenum;     // размерность текстуры (GL_TEXTURE_2D)
  level,              // 0, число уровней детализации текстуры
  components: GLint; // 3 - красный, зеленый и синий
  width,              // ширина изображения
  height: GLsizei;    // высота изображения
  border: GLint;      // 0 или 1 - ширина границы
  format,             // GL_RGB
```

```
  _type: GLenum;      // GL_UNSIGNED_BYTE - тип данных
  pixels: Pointer     // указатель на данные о цветах
);
```

Ниже приводится текст метода *GetTextureFromFile*, который загружает изображение из файла с именем *NameFile* в *Bitmap*, формирует массив *pixels* и передает эти данные в карту текстуры.

Листинг 9.29. Загрузка изображения из файла

```
public static void GetTextureFromFile(string FileName)
{
    Bitmap Bitmap = new System.Drawing.Bitmap(FileName,
       true);
    byte[] pixels = new byte[Bitmap.Width*Bitmap.Height * 3];

    for (int i = 0; i < Bitmap.Width; i++)
       for (int j = 0; j < Bitmap.Height; j++)
       {
          Color color = Bitmap.GetPixel(i, j);
          int color1 = color.R+(color.G << 8)+(color.B<< 16);
          pixels[j * Bitmap.Width * 3 + 3 * i + 0] =
             (byte)(color1 & 0xFF);
          pixels[j*Bitmap.Width*3+3*i+1]=
             (byte)((color1&0xFF00)>>8);
          pixels[j * Bitmap.Width * 3 + 3 * i + 2] =
             (byte)((color1 & 0xFF0000) >> 16);  //
       }
    Gl.glTexImage2D(Gl.GL_TEXTURE_2D, 0, 3, Bitmap.Width,
       Bitmap.Height, 0, Gl.GL_RGB,
       Gl.GL_UNSIGNED_BYTE, pixels);
    Bitmap.Dispose();
}
```

9.7.3. Включить режим наложения текстуры

Включение режима наложения двумерной текстуры реализуется командой *glEnable(GL_TEXTURE_2D)*.

9.7.4. Привязка текстуры к многоугольникам

Можно достаточно легко изменить, например, метод рисования куба `DrawBox`, привязав к вершинам узловые точки (0,0), (0,1), (1,1), (1,0) карты текстуры, соблюдая порядок обхода. Для этого достаточно воспользоваться командой *glTexCoord2d*.

Листинг 9.30. Задание куба с привязкой к шаблону текстуры

```
for (int i = 0; i < 6; i++)
{
    Gl.glNormal3fv(BoxPoints[i]);
    Gl.glBegin(Gl.GL_QUADS);
       Gl.glTexCoord2d(0, 0);
       Gl.glVertex3fv(V[BoxFaces[i][0]]);

       Gl.glTexCoord2d(0, 1);
       Gl.glVertex3fv(V[BoxFaces[i][1]]);

       Gl.glTexCoord2d(1, 1);
       Gl.glVertex3fv(V[BoxFaces[i][2]]);

       Gl.glTexCoord2d(1, 0);
       Gl.glVertex3fv(V[BoxFaces[i][3]]);
    Gl.glEnd();
}
```

В качестве изображения выбран файл *24.bmp* размером 512×512. Результат рисования показан на рисунке 9.19.

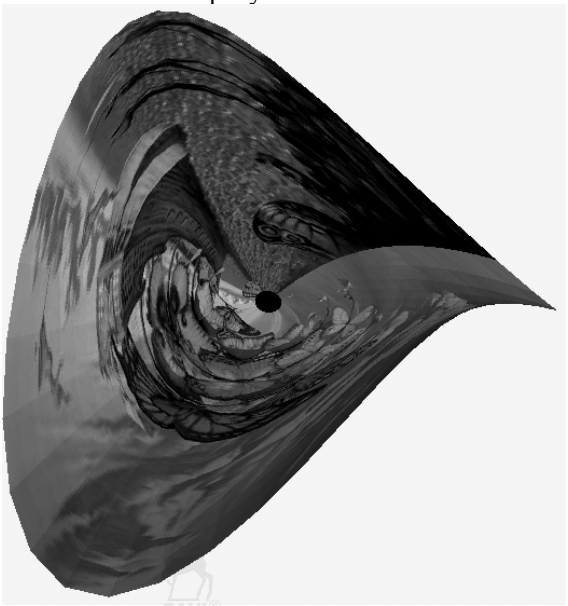

Рис. 9.19. Текстура, наложенная на гиперболический параболоид

9.7.5. Текстура на поверхности, заданной таблично

Также просто накладывается текстура на поверхность, заданную таблично, типа той, что изображена на рисунке 9.17.

Возьмем в качестве примера файл *8.bmp* размера 512×512 с картой (рис. 9.20).

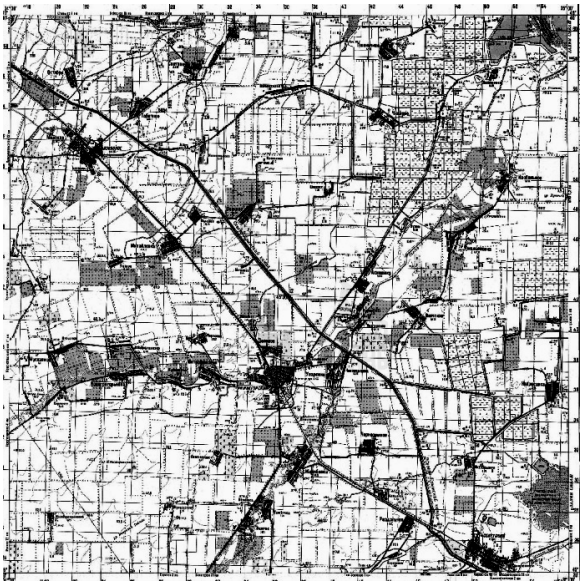

Рис. 9.20. Изображение карты местности

Нам придется слегка изменить метод *GetSurface()*, представленный в листинге 9.30, снабдив каждую вершину привязкой к карте текстуры. При вычислении параметров команды *glTexCoord2d(s,t)* надо помнить, что параметры *s* и *t* меняются от 0 до 1 и эти единичные интервалы необходимо разбивать на *nRow* и *nCol* частей соответственно.

Листинг 9.31. Рисование поверхности с текстурой

```
static void DrawSurface()
{
    if (TMyBody.Faces == null) return;

    int NumFace = TMyBody.Faces.Count;
    if (NumFace > 0)
    {
       int DrawType;
       if (flLine)
```

```
            DrawType = Gl.GL_LINE_LOOP;
        else
            DrawType = Gl.GL_TRIANGLES;
        int n = 0;
        for (int i = 0; i < NumFace; i++)
        {
           Gl.glBegin(DrawType);
           n = TMyBody.Faces[i].Vertex[0];
           Gl.glNormal3f(TMyBody.Norms[n].x,
              TMyBody.Norms[n].y, TMyBody.Norms[n].z);
           Gl.glTexCoord2d(TMyBody.Texture[n].x,
              TMyBody.Texture[n].y);
           Gl.glVertex3f(TMyBody.Points[n].x,
              TMyBody.Points[n].y, TMyBody.Points[n].z);
           n = TMyBody.Faces[i].Vertex[1];
           Gl.glNormal3f(TMyBody.Norms[n].x,
              TMyBody.Norms[n].y, TMyBody.Norms[n].z);
           Gl.glTexCoord2d(TMyBody.Texture[n].x,
              TMyBody.Texture[n].y);
           Gl.glVertex3f(TMyBody.Points[n].x,
              TMyBody.Points[n].y, TMyBody.Points[n].z);
           n = TMyBody.Faces[i].Vertex[2];
           Gl.glNormal3f(TMyBody.Norms[n].x,
              TMyBody.Norms[n].y, TMyBody.Norms[n].z);
           Gl.glTexCoord2d(TMyBody.Texture[n].x,
              TMyBody.Texture[n].y);
           Gl.glVertex3f(TMyBody.Points[n].x,
              TMyBody.Points[n].y, TMyBody.Points[n].z);
           Gl.glEnd();
        }
    }
}
```

Результат рисования представлен на рисунке 9.21.

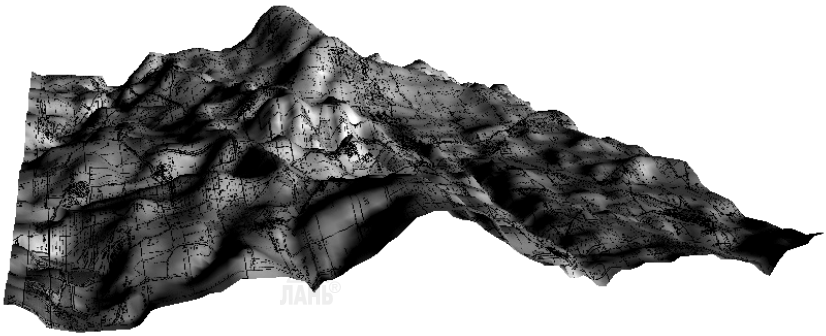

Рис. 9.21. Поверхность с текстурой карты местности

9.8. Чтение данных из текстового файла

Программа Caligari™, использующая другую графическую библиотеку *DirectX*, может записывать данные о многогранниках в текстовых файлах с расширением *.*x*. В папке «Фигуры» находится 20 таких файлов: гидра.x, кольцо тора2.x, корабль.x и т.д.

Файлы содержат, в частности, информацию о координатах вершин, треугольниках поверхности, нормалях к каждой точке, координатах привязки текстуры к каждой вершине, информацию о материале и имеют следующую структуру:

```
…
      Mesh Hedra01Mesh {
            42;
            -0.000000;-0.000000;4.000000;,
            -2.000000;3.236068;-1.236068;,
             …
            -1.788854;1.105573;0.000000;,
            1.788854;-1.105573;-0.000000;;
            80;
            3;1,8,0;,
            3;2,9,1;,
             …
            3;27,21,41;,
            3;2,27,41;;
            MeshNormals {
                  42;
                  -0.000000;-0.000000;1.000000;,
                  -0.500000;0.809017;-0.309017;,
                  …
                  -0.850651;0.525731;0.000000;,
                  0.850651;-0.525731;-0.000000;;
             }
            MeshTextureCoords {
                  42;
                  0.000000;0.500000;
                  0.654508;0.904509;
                  …
                  0.500000;0.638197;
                  0.500000;0.361803;;
             }
            MeshMaterialList {
                  1;
                  80;
                  0,
                  …
```

Данные, находящиеся в начале файла, для нас интереса не представляют. Информация о вершинах начинается после строки, у которой во второй позиции есть слово *Mesh*. В следующей строке указано количество вершин (в нашем случае – 42). Затем следует указанное количество строк, в каждой из которых присутствует три координаты вершины. В следующей строке находится число граней (в нашем случае – 80). Затем следует указанное количество строк, в каждой из которых присутствует число вершин грани и номера этих вершин. После строки со словом *MeshNormals* идет строка с числом нормалей, а затем следует указанное количество строк, в каждой из которых присутствует три координаты единичной нормали. После строки со словом *MeshTextureCoords* идет строка с числом точек привязки карты текстуры. Затем следует указанное количество строк, в каждой из которых присутствует две координаты точки привязки.

Для чтения текстового файла предназначен метод *ReadObject(var MyBody: TMyBody; FileName: string)*, который заполняет поля переменной *MyBody: TMyBody*.

Листинг 9.32. Класс данных для текстового файла

```
class TMyBody
{
    public static List<TPoints> Points;
    public static List<TFace> Faces;
    public static List<TPoints> Norms;
    public static List<TPoints2> Texture;
}
```

Алгоритм метода *ReadObject* достаточно прост, находится в модуле *G.cs* и поэтому здесь не приводится.

Подготовка данных для рисования фигуры треугольниками осуществляет метод *GetSurface()*, который для каждой из трех вершин задает вектор нормали и координату карты текстуры. Текст приведен в листинге 9.31.

Задание вектора нормали для каждой вершины позволяет получить сглаженное изображение фигуры с наложением текстуры (рис. 9.22).

В заключение этого параграфа напомним, что этот проект находится в папке «Example 35 OpenGL 3D».

Рис. 9.22. Одна из фигур, данные для которой взяты из текстового файла «Лицо.x»

Литература

1. Bresenham J.E. Algorithm for computer control of a digital plotter // IBM Systems Journal. – 1965. – Vol. 4, No. 1. – Pp. 25–30.
2. Bresenham J.E. A Linear Algorithm for Incremental Digital Display of Circular Arcs // Communication of the ACM. – 1977. – Vol. 20. – Pp. 100–106.
3. Cyrus M. Generalized two- and threedimensional clipping / M. Cyrus, J. Beck // Computer and Graphics. – 1978. – Vol. 3. – Pp. 23–28.
4. Digital Fantasy art 3d wallpapers (англ.). – Коллекция трехмерной графики международных дизайнеров. – Электронный ресурс: http://www.fantasyartdesign.com.
5. Encarnacao J. Einfurung in die Graphische Datenverarbeiterung // Eurographics'89. Tutorial Notes 1. – Hamburg, FRG. – 1989. – September 4–8. – 122 s.
6. Free 3D design, 2D art software (англ.). – Электронный ресурс: http://www.freeartsoftware.com.
7. Free 3D models, 3ds max (англ.). – Подборка бесплатных 3D моделей. – Электронный ресурс: http://www.artist-3d.com.
8. Mike Lischke. Программа просмотра 3DS-файлов, а также пакет GLScene. – Электронный ресурс: http://www.lischke-online.de.
9. Phong Bui-Tuong. Illumination for Computer-Generated Pictures // Communication of the ASM. – 1975. – 18(6). – Pp. 311–317.
10. Serban Iulian. GDI+ Custom Controls with Visual C# 2005 // Iulian Serban, Brezoi Dragos, Radu Tiberiu, Adam Ward. – BIRMINGHAM – MUMBAI. 2006. – P. 271.
11. Sobkow Mark S. A Fast Two-Dimensional Line Clipping Algoritm via Line Encoding / Mark S. Sobkow, Paul Pospisil, Yang Yee-Hong // Computer & Graphics. – 1987. – Vol. 11, No. 4. – Pp. 459–467.
12. Strauss Paul S. An object-oriented 3D graphics toolkit / Paul S. Strauss, R. Carey // In Proceeding of SIGGRAPH'92. – 1992. – Pp. 341–349.
13. Sutherland I.E. Reentrant Polygon Clipping / I.E. Sutherland, G.W. Hodgman // Communications of the ACM. – 1974. – 17(1). – Pp. 32–42.
14. Абраш Майкл. Программирование графики. Таинства. – Киев : EvroSYB, 1995. – 383 с.
15. Адамс Дж. DirectX: продвинутая анимация: пер. с англ.. – М. : КУДИЦ-ОБРАЗ, 2004. – 478 с.

16. Библиотека программирования графики SciTech MGL. – Электронный ресурс: http://www.scitechsoft.com.
17. Вельтмандер П.В. Основные алгоритмы компьютерной графики: учеб. пособие: в 3 книгах. – Новосибирск: Новосибирский гос. ун-т, 1997.
18. Вирт Н. Алгоритмы + структуры данных = программы / пер. с англ. Л.Ю. Иоффе; под ред. Д. Б. Подшивалова. – М. : Мир, 1985. – 406 с.
19. Гилой В. Интерактивная машинная графика / пер. с англ. – М. : Мир, 1981.
20. Гончаров Д. DirectX 7.0 для программистов: учеб. курс / Д. Гончаров, Т. Салихов. – СПб. : Питер, 2001. – 520 с.
21. Графика и мультимедиа. Научно-образовательный сетевой журнал о компьютерной графике, машинном зрении и обработке изображений – Электронный ресурс: http://cgm.graphicon.ru:8080/.
22. Графика/Мультимедиа – Электронный ресурс: http://www.delphisources.ru/pages/sources_list.html.
23. Джамбруно М. Трехмерная графика и анимация / пер. с англ. Е.В. Кикиневой [и др.]; под ред. А.Н. Кушнира. – 2-е изд. – М. : Издат. дом «Вильямс», 2002. – 638 с.
24. Иванов В.П. Трехмерная компьютерная графика / В.П. Иванов, А.С. Батраков; под ред. Г.М. Полищука. – М. : Радио и связь, 1995. – 224 с.
25. Коллекция трехмерной графики. – Электронный ресурс: http://www.neosurrealismart.com.
26. Компьютерная графика, обработка изображений и мультимедиа. Сайт поддерживается сотрудниками и аспирантами лаборатории компьютерной графики и мультимедиа при факультете ВМиК МГУ. – Электронный ресурс: http://graphics.cs.msu.su/courses/cg/library/.
27. Корриган Дж. Компьютерная графика: Секреты и решения / пер. с англ. Д.А. Куликова. – М. : Энтроп, 1995. – 350 с.
28. Маров М.Н. Энциклопедия 3D Studio Max 6. – СПб. : Питер, 2006.
29. Мартинес Ф. Синтез изображений. Принципы, аппаратное и программное обеспечение / пер. с франц. – М. : Радио и связь, 1990. – 192 с..
30. Набор компонентов VisIt – Электронный ресурс: http://www.signsoft.com/downloadcenter/index.html.
31. Онлайн-журнал, посвященный трехмерной графике и анимации. – Электронный ресурс: http://www.render.ru.
32. Павлидис Т. Алгоритмы машинной графики и обработки изображений / пер. с англ. – М. : Радио и связь, 1986.

33. Поляков А. Программирование графики. GDI+ и DirectX / А. Поляков, В. Брусенцев. – СПб. : БХВ-Петербург, 2005. – 357 с.
34. Прэтт У. Цифровая обработка изображений: в 2 книгах / пер. с англ. – М. : Мир, 1982.
35. Роджерс Д.Ф. Математические основы машинной графики / пер. со 2-го англ. изд. П. А. Монахова [и др.]; под ред. Ю.М. Баяковского [и др.]. – М. : Мир, 2001. – 604 с.
36. Роджерс Д. Алгоритмические основы машинной графики / пер. с англ. – М. : Мир, 1989. – 512 с.
37. Рост Р.Дж. OpenGL: трехмерная графика и язык программирования шейдеров / Р.Дж. Рост [и др.]. – СПб. : Питер, 2005. – 427 с.
38. Русскоязычное сообщество пользователей пакета Softimage|XSI. – Электронный ресурс: http://www.softimage.ru.
39. Скворцов А.В. Алгоритмы построения и анализа триангуляции / А.В. Скворцов, Н.С. Мирза. – Томск : Изд-во Томского ун-та, 2006. – 168 с.
40. Тихомиров Ю. Программирование трехмерной графики. – СПб. : BHV, 2000. – 245 с.
41. Томпсон Н. Секреты программирования трехмерной графики для Windows 95 / пер. с англ. Е. Матвеева. – СПб. : Питер, 1997. – 340 с.
42. Тюкачев Н.А. Компьютерная графика и мультимедиа: учебник / Н.А. Тюкачев, И.В. Илларионов, В.Г. Хлебостроев. – Воронеж : Издат.-полиграф. центр ВГУ, 2008. – 797 с.
43. Тюкачев Н.А. Программирование графики в Delphi. / Н.А. Тюкачев, И.В. Илларионов, В.Г. Хлебостроев. – СПб. : БХВ-Петербург. 2008. – 766 с.
44. Тюкачев Н.А. Разработка алгоритмов вычислительной геометрии и их реализация в трехмерной геоинформационной системе: учеб. пособие. – Воронеж : Издат.-полиграф. центр ВГУ, 2009.
45. Тюкачев Н.А. Сглаживание триангуляции трехмерных поверхностей // Информатика: проблемы, методология, технологии: мат. VI междунар. науч.-метод. конф. – Воронеж : ВГУ, 2006. – С. 9–11.
46. Тюкачев Н.А. Структура данных для топологически связанных трехмерных тел // Вестник ВГУ. Серия: Системный анализ и информационные технологии. – 2006. – № 1. – С. 141–144.
47. Фролов А.В. Программирование видеоадаптеров CGA, EGA и VGA. – М. : ДИАЛОГ-МИФИ, 1994. – 287 с.
48. Хоу А. LightWave 3D 8 / А. Хоу, Б.И. Маршалл. – М. : NT Press, 2004. – 420 с.

49. Шикин Е.В. Компьютерная графика. Полигональные модели: учеб. пособие / Е.В. Шикин, А. В. Боресков. – М. : ДИАЛОГ-МИФИ, 2005. – 461 с.
50. Шикин Е.В. Компьютерная графика: Динамика, реалистические изображения / под ред. О.А. Голубева. – М. : ДИАЛОГ-МИФИ, 1995. – 287 с.
51. Шикин Е.В. Начала компьютерной графики / А.М. Швайгер, А.Л. Хейфец, В.А. Краснов; под общ. ред. Е.В. Шикина – М. : Диалог-МИФИ, 1993. – 138 с.
52. Юань Фень. Программирование графики для Windows. – СПб. : Питер, 2002. – 1070 с.
53. Янг М. Программирование графики в Windows 95: Векторная графика на языке C+ / пер. с англ.; под ред. В. Тимофеева. – М. : БИНОМ, 1997. – 366 с.

Оглавление

Николай Аркадиевич ТЮКАЧЕВ,
Виктор Григорьевич ХЛЕБОСТРОЕВ

C#. ПРОГРАММИРОВАНИЕ 2D И 3D ВЕКТОРНОЙ ГРАФИКИ

Учебное пособие

Издание четвертое, стереотипное

Зав. редакцией литературы
по информационным технологиям и системам связи *О. Е. Гайнутдинова*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com;
196105, Санкт-Петербург, пр. Юрия Гагарина, 1, лит. А.
Тел.: (812) 412-92-72, 336-25-09.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 01.10.19.
Бумага офсетная. Гарнитура Школьная. Формат 60×90 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 20,00. Тираж 100 экз.

Заказ № 691-19.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.