МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
ДОНЕЦКОЙ НАРОДНОЙ РЕСПУБЛИКИ
ГОУ ВПО «ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ»
ФИЗИКО-ТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ
КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

К ВЫПОЛНЕНИЮ И ОФОРМЛЕНИЮ ЛАБОРАТОРНЫХ РАБОТ К КУРСУ

«ИНЖЕНЕРНАЯ И КОМПЬЮТЕРНАЯ ГРАФИКА»

(ЧАСТЬ 1)

ДЛЯ СТУДЕНТОВ НАПРАВЛЕНИЯ ПОДГОТОВКИ

09.03.01 «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

КВАЛИФИКАЦИОННОГО УРОВНЯ «БАКАЛАВР»

Донецк
ГОУ ВПО «ДонНУ»
2016

УДК: 004.92(072)
ББК: Ж11р30-252.43 + З973.2-018.3р30-252.43
К 731

Автор-составитель:
В. Н. Котенко, старший преподаватель кафедры

Ответственный за выпуск:
В. К. Толстых, д-р техн. наук, проф.

Утверждено на заседании ученого совета
физико-технического факультета ГОУ ВПО «ДонНУ»
Протокол № 18 от 20.05.2016

Методические указания к выполнению и оформлению лабораторных работ по курсу «Инженерная и компьютерная графика» для студентов укрупненной группы направлений подготовки 09.00.00 «Информатика и вычислительная техника» направления подготовки 09.03.01 «Информатика и вычислительная техника» квалификационного уровня «Бакалавр» / В. Н. Котенко. – Донецк: ГОУ ВПО «ДонНУ», 2016. – Ч.1 – 98 с.

Методические указания содержат теоретический материал по каждой теме, примеры, его разъясняющие, и варианты заданий.

Предназначены для студентов всех направлений подготовки, изучающих инженерную и компьютерную графику.

УДК: 004.92(072)
ББК: Ж11р30-252.43 + З973.2-018.3р30-252.43

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Методические указания предназначены для изучения студентами общих принципов хранения, отображения и преобразования графической информации; содержат сведения о методах, средствах и технологиях графического моделирования, о фундаментальных методах в графике, что предполагает наличие у обучающихся знаний об организации и функционировании ЭВМ и современных языках программирования. В данных указаниях использованы таблицы, схемы, графики, а также примеры программных кодов, которые разъясняют и закрепляют теоретический материал.

Методические указания построены таким образом, что вначале приводится теоретическая часть, знание которой необходимо для выполнения последующих задач; после нее формулируются задания в виде лабораторных работ. Каждая работа содержит цель, варианты заданий и порядок их выполнения, контрольные вопросы по соответствующей теме, ответы на которые помогают студентам закрепить теоретический материал.

В конце методических указаний приводится список рекомендованной литературы для изучения дисциплины.

Выполнение лабораторной работы состоит из нескольких этапов: получение студентом у преподавателя задания, изучение теоретической части и самостоятельный анализ литературных источников, проектирование и написание программы, тестирование и отладка программного продукта, оформление и защита отчета по работе.

Отчет по работе должен содержать название темы, цель, ответы на контрольные вопросы, распечатку программы и результатов.

1 Использование графики в приложениях *Windows*

В лабораторных работах мы изучим основы графической системы *GDI+* (Graphics Device Interface) в контексте платформы .NET и языка C#.

1.1 Использование *GDI* + в .NET-программах

В языке C# существует библиотека для создания графики *System.Drawing*. Она автоматически подключается при создании проекта.

Чтобы использовать *GDI* + в программах на основе .NET, необходимо объявить в верхней части каждого файла кода, какие пространства имен он использует. В нашем случае объявить, что используется пространство имен *System.Drawing*. В дальнейшем понадобится и пространство имен *System.Drawing.Drawing2D*. Поэтому, целесообразно сразу добавить и его в решение. Это делается с помощью объявления, показанного в листинге:

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
// Другие using объявления располагаются здесь
namespace MyNamespace
{
     // Здесь располагаются классы...
}
```

1.2 Объект *Graphics*

Вся работа в *GDI+* происходит на объекте *Graphics* – классе имитации поверхности *рисования в GDI+*. Класс находится в пространстве имен *Drawing*.

Класс не имеет конструкторов, потому что его объекты зависят от контекста конкретных устройств вывода. Создаются объекты специальными методами разных классов, например, из объекта *Bitmap* или к нему можно получить доступ как к некоторому объекту, инкапсулированному в некоторые элементы управления, в том числе и в объект формы приложения. Метод *CreateGraphics* класса *Control*, наследника класса *Form,* возвращает объект, ассоциированный с выводом графики на форму.

Во время цикла *Paint* объект *Graphics* будет предоставлен в объекте *PaintEventArgs*, который передается вашему коду в методах *OnPaint* и *OnPaintBackground* объектов управления *Windows Forms*. Этот же аргумент передается обработчикам, обслуживающим событие *Paint*, генерируемое методами *OnPaint*. При печати *PrintPageEventArgs*, предоставляемый в событии *PrintPage*, будет содержать объект *Graphics* для принтера и можно будет получить объект *Graphics* для определенных типов изображения, так что станет возможным рисовать прямо на изображении в памяти, как если бы это был экран.

1.3 Получение объекта *Graphics*

Когда пишете программы, которые выводят графику на экран, объект *Graphics* будет передан, завернутый в объекте *PaintEventArgs*.

Есть два способа, которыми можно получить объект *Graphics.*

Можно или переопределить защищенные методы *OnPaint* или *OnPaintBackground,* или добавить обработчик события *Paint*. Во всех этих случаях объект *Graphics* передается в объекте *PaintEventArgs*.

Первый способ – переопределение защищенного метода *OnPaint:*

```
protected override void OnPaint(PaintEventArgs e)
{
    // Получаем объект Graphics из объекта PaintEventArgs
```

```
    Graphics g = e.Graphics;
    g.DrawLine(...);
    // Или используем объект Graphics напрямую
    e.Graphics.DrawLine(...);
    // Вызвать базовый класс, иначе событие Paint не сработает
    base.OnPaint(e);
}
```

Второй способ – добавление обработчика события *Paint*:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e) {
    // Получаем объект Graphics из объекта PaintEventArgs
    Graphics g = e.Graphics;
    g.DrawLine(...);
    // Или используем объект Graphics напрямую
    e.Graphics.DrawLine(...);
}
```

Из листинга видно, что можно получить ссылку на объект *Graphics* и сохранить ее для использования в коде программы или использовать объект *Graphics* непосредственно из *PaintEventArgs*.

1.4 Операции с объектом *Graphics*

С объектом *Graphics* можно выполнить пять основных операций:

1) нарисовать фигуру. Фигуры, такие как прямоугольники, эллипсы и линии, рисуются с помощью объекта *Pen* (Перо). Перо может иметь различную толщину, цвет и многие другие атрибуты;

2) закрасить фигуру. Фигуры могут быть закрашены с помощью объекта *Brush* (кисть). Кисти имеют много сложных настроек в *GDI +*;

3) нарисовать строку. Текст может быть размещен на поверхности объекта *Graphics* с помощью метода *DrawString*;

4) нарисовать изображение. Используя один из методов *DrawImage*, можно создать изображения в любом масштабе;

5) модифицировать объект *Graphics*: получить высококачественную графику (в ущерб скорости), изменить способ вывода графики с помощью вращения, масштабирования или искажающих эффектов.

Для демонстрации использования объекта *Graphics* в листинге приведен пример кода обработчика события *Paint* формы для демонстрации рисования и закраски фигур.

```
protected override void OnPaint(PaintEventArgs e) {
      // Получаем объект Graphics
      Graphics g = e.Graphics;
      // Рисуем линию
      g.DrawLine(Pens.Red,10,5,110,15);
      // Рисуем эллипс
      g.DrawEllipse(Pens.Blue,10,20,110,45);
      // Рисуем прямоугольник
      g.DrawRectangle(Pens.Green,10,70,110,45);
      // Рисуем закрашенный эллипс
      g.FillEllipse(Brushes.Blue,130,20,110,45);
      // Рисуем закрашенный прямоугольник
      FillRectangle(Brushes.Green,130,70,110,45);
      base.OnPaint(e);
}
```

На рисунке 1.1 показана работа этого приложения.

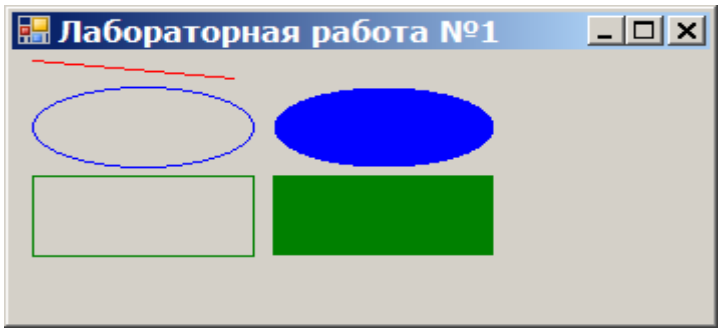

Рис. 1.1 – Рисование и закраска фигур

Не следует хранить объект *Graphics* вне рамок метода, где он задан. Когда рисование завершено, объект уничтожается.

Лабораторная работа № 1

Создание графического приложения *Windows*

Цель работы:

1. Изучить основы графической системы *GDI+*.

2. Получить навыки создания простейших графических приложений *Windows*.

Контрольные вопросы по теме:

1) Какая библиотека в C# используется для создания графики?

2) Какие пространства имен необходимо объявить, чтобы использовать *GDI+* в программе? Приведите соответствующие объявления.

3) Что представляет собой объект *Graphics*?

4) В каком объекте предоставляется объект *Graphics* при выводе графики на экран?

5) Приведите описание способа получения объекта *Graphics* переопределением защищенного метода *OnPaint* или *OnPaintBackground*.

6) Приведите описание способа получения объекта *Graphics* при добавлении обработчика события *Paint*.

7) Какой объект при печати, предоставляемый в событии *PrintPage*, будет содержать объект *Graphics* для принтера?

8) Какие операции можно выполнять с объектом *Graphics*?

9) В какой части программы следует хранить объект *Graphics*?

Задания:

а) Создайте простейшее графическое windows-приложение, которое рисует красный прямоугольник на объекте *PictureBox* при обработке события «Нажатие на кнопку». Для этого:

1) запустите *Microsoft Visual Studio*;

2) чтобы создать новый проект, зайдите в меню «Файл» в левом верхнем углу и выберите «Создать» → «Проект»;

3) в появившемся окне выберите в левой части окна *Visual C#*, а в правой части окна из предложенных создаваемых объектов – приложение *Windows Forms*;

4) после создания проекта появится «Форма» *(Form1)* и «Панель Элементов» *(ToolBox)*. Выберите из этой панели объекты *Button* (кнопка), *PictureBox* (поле для рисования) и расположите их на созданной форме;

5) выберите на форме объект *Button1*. В окне «Свойства» найдите кнопку «События», во вкладке «Действие» напротив *Click* введите название метода, отвечающего за событие при нажатии на кнопку, или дважды кликните мышкой, – название методу присвоится автоматически;

6) в методе *Button1_Click* между фигурными скобками напишите код, который будет выполняться на событие клика по кнопке:

```
// Выбираем перо myPen красного цвета толщиной в 1 пиксель:
Pen myPen = new Pen(Color.Red, 1);
// Объявляем объект «g» класса Graphics
Graphics g = pictureBox1.CreateGraphics();
g.DrawRectangle(myPen, 0, 0, pictureBox1.Size.Width-1,
pictureBox1.Size.Height-1);
// Рисуем прямоугольник
g.Dispose();
```

7) для запуска проекта нажмите на кнопку «Запуск» в виде зеленого треугольника на панели действий или используйте «горячую клавишу» *F5*.

б) Создайте графическое приложение, которое рисует красную линию, синий эллипс, синий закрашенный эллипс, зеленый

прямоугольник, зеленый закрашенный прямоугольник, переопределив метод *OnPaint*. Для этого:

1) повторите шаги 1, 2, 3 задания а);

2) нажмите клавишу *F7*, чтобы перейти к коду приложения;

3) после строк `public Form1() { InitializeComponent();}` вставьте

```
protected override void OnPaint(PaintEventArgs e){
Graphics g = e.Graphics;  // Получаем объект Graphics
// Рисуем линию
g.DrawLine(Pens.Red, 10, 5, 110, 15);
// Рисуем эллипс
g.DrawEllipse(Pens.Blue, 10, 20, 110, 45);
// Рисуем прямоугольник
g.DrawRectangle(Pens.Green, 10, 70, 110, 45);
// Рисуем закрашенный эллипс
g.FillEllipse(Brushes.Blue, 130, 20, 110, 45);
// Рисуем закрашенный прямоугольник
g.FillRectangle(Brushes.Green, 130, 70, 110, 45);
base.OnPaint(e);
}
```

Порядок выполнения лабораторной работы:

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

2 Системы координат, цвета

Положение элементов на поверхности объекта *Graphics* задается *x* и *y* координатами. Одиночная координата представляет собой пару чисел, которые определяют расстояние по горизонтали на поверхности слева направо и расстояние по вертикали сверху вниз. Например, линию можно провести из позиции 0,0 в позицию 100,200, используя код

```
g.DrawLine(Pens.Black, 0,0,100,200);
```

Координаты определены в последних четырех параметрах команды. Нарисованная линия показана на рисунке 2.1.

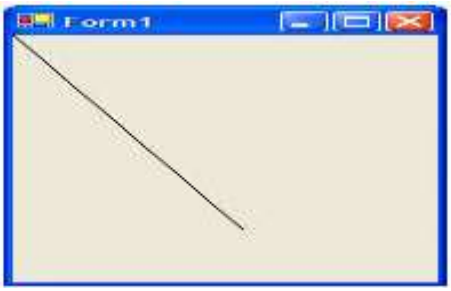

Рис. 2.1 – Линия, проведенная из позиции 0,0 в позицию 100,200

По умолчанию координаты, используемые в *GDI+*, ссылаются на позиции пикселей, но *GDI+* является независимой от разрешения системой рисования, т.е. можно задавать такие единицы рисования, как пиксели или единицы реального мира, такие как дюймы и миллиметры.

Многие графические системы, в частности, старый *Windows GDI*, использовали целочисленные значения для координат. Это означало, что нельзя было задать дюйм или вещественное число, например, 1,5. В *GDI+* координаты, используемые для размещения цвета на поверхности рисования, являются числами с плавающей точкой. Если нужно создать программу для рисования чертежа и показать конструкции в точных дюймовых или миллиметровых размерах, то можно это сделать, используя *GDI+*.

2.1 Координатные пространства

В *GDI+* есть три основных координатных пространства.

1. *Мировое координатное пространство.* В этом случае устанавливают координаты, которые определяют линии, фигуры и точки в двумерном пространстве графической системы. Мировые координаты – абстрактные величины, выраженные в виде чисел с плавающей точкой. По сути, каждый раз, когда рисуете, это происходит в данном пространстве координат.

2. *Страничное координатное пространство.* Страничное пространство – это пространство, где мировые координаты преобразуются в некоторые реальные значения. Можно создать страничное пространство, представляющее пиксели, дюймы, миллиметры и т. п. Это то, что делает *GDI+* системой независимого разрешения. В этом случае пользователь управляет тем, как страничное пространство интерпретирует мировое пространство, указывая объекту *Graphics,* какой *PageUnit* используется, и регулируя *PageScale*.

3. *Координатное пространство устройства*. Это пространство управляется системой и позволяет отобразить реальные значения в страничном пространстве на экран или принтер. Пространство устройства гарантирует, что линия длиной 1 дюйм выглядит длиной в дюйм на экране и на принтере, даже если эти два устройства имеют очень разные пиксельные разрешения. Нельзя напрямую управлять этим пространством.

Правильный термин для этого типа систем, где координаты преобразуются из одной системы в другую за несколько шагов, – *Graphics Pipeline* (графический конвейер). Графический конвейер *GDI+* принимает значения, которые используются, и превращает их из абстрактного числа с плавающей точкой в реальные значения, а затем – в аппаратные пиксели монитора или принтера.

Ниже перечислены фактические, реальные, доступные стандарты страничного пространства.

1) *Pixel*. Каждый элемент в мировом пространстве представляет собой 1 пиксель на экране или принтере. Это значение по умолчанию для страничных элементов. Объекты, нарисованные с этой установкой, будут иметь различные размеры на различных устройствах, таких как экраны и принтеры.

2) *Millimeter*. Каждый элемент в мировом пространстве представляет собой один миллиметр. При этой установке для страничного элемента он будет выглядеть одинаково как на принтере, так и на экране.

3) *Inch*. Каждый элемент в мировом пространстве представляет собой один дюйм. При этой установке для страничного элемента он будет выглядеть одинаково как на принтере, так и на экране.

4) *Point*. Каждый элемент в мировом пространстве представляет собой одну точку принтера размером 1/72 дюйма. *Point* – предпочтительная единица измерения для приложений интенсивного типа. При этой установке страничный элемент будет выглядеть одинаково и на принтере, и на экране.

5) *Display*. Элемент мирового пространства будет представлять собой

1/75 дюйма. Это пережиток тех времен, когда общепринятый шаг точки электронно-лучевой трубки был 75 точек на дюйм (DPI). При этой установке страничный элемент будет выглядеть одинаково и на принтере, и на экране.

6) *Document*. Элементы мирового пространства будут равны 1/300 дюйма. Это пережиток времен, когда лазерные принтеры имели разрешение 300 dpi.

На рисунке 2.2 показана работа приложения, демонстрирующего, как могут быть использованы действительные значения.

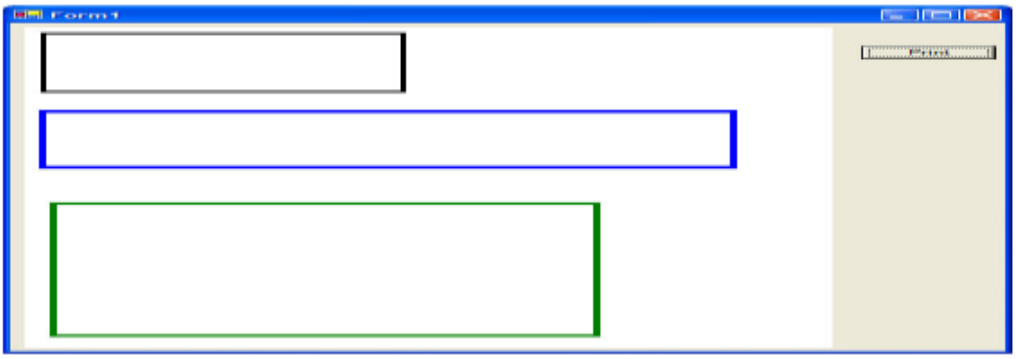

Рис. 2.2 – Системы рисования в мировом пространстве

Приложение рисует несколько прямоугольников в дюймовых, миллиметровых и пиксельных размерах. Оно также осуществляет печать. Если сравнить размеры на печатной копии и на экране, то будет видно, что печатные и экранные размеры реальных значений идентичны, а пиксельные – нет.

Фрагмент кода, который осуществляет рисование прямоугольников, показан в листинге:

```
private void DrawRectangles(Graphics g) {
     g.PageUnit = GraphicsUnit.Pixel;
     Pen p = new Pen(Color.Black, 3);   // перо толщиной 3
пикселя
     g.DrawRectangle(p,10,10,200,100);  // прямоугольник
     p.Dispose();
     g.PageUnit = GraphicsUnit.Inch;
     p = new Pen(Color.Blue,0.05f);     // перо толщиной в 1/20
дюйма
     g.DrawRectangle(p,0.1f,1.5f,4f,1f); // прямоугольник 4 к 1
     p.Dispose();
     g.PageUnit = GraphicsUnit.Millimeter;
     p = new Pen(Color.Green,1f);  // перо толщиной в 1
миллиметр
     g.DrawRectangle(p,4f,80f,80f,60f); // прямоугольник
80 x 60 мм
     p.Dispose(); }
```

В методе *DrawRectangles*, который приведен ниже, один и тот же код используется как для печати, так и для рисования на экране. Обратите внимание, как для рисования, основанного на дюймах, используются значения с плавающей точкой (дробные числа).

Полный листинг кода, приведенный ниже, содержит обработчик события нажатия на кнопку, чтобы напечатать страницу с нарисованными прямоугольниками. Если разрешение принтера не идентично разрешению экрана, то появятся два разных результата одного и того же кода.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Drawing.Printing;

namespace RealWorld {
public partial class Form1 : Form
{ public Form1() { InitializeComponent();    }
private void DrawRectangles(Graphics g) {
     g.PageUnit = GraphicsUnit.Pixel;
     Pen p = new Pen(Color.Black, 3);  // перо толщиной 3 пикселя
     g.DrawRectangle(p,10,10,200,100); // прямоугольник
     p.Dispose();
     g.PageUnit = GraphicsUnit.Inch;
     p = new Pen(Color.Blue,0.05f); // перо толщиной в 1/20 дюйма
     g.DrawRectangle(p,0.1f,1.5f,4f,1f); // прямоугольник 4 к 1
     p.Dispose();
     g.PageUnit = GraphicsUnit.Millimeter;
     p = new Pen(Color.Green,1f);   // перо толщиной в 1 миллиметр
     g.DrawRectangle(p,4f,80f,80f,60f); // прямоугольник 80 х 60 мм
     p.Dispose();
}
```

```
 private void panel1_Paint(object sender,
                           System.Windows.Forms.PaintEventArgs
     e){ DrawRectangles(e.Graphics); }
private void button1_Click(object sender, System.EventArgs e)
{
     PrintDocument pd = new PrintDocument();
     pd.PrintPage+ = new PrintPageEventHandler(pd_PrintPage);
     pd.Print();
}
private void pd_PrintPage(object sender, PrintPageEventArgs e)
{
     DrawRectangles(e.Graphics);
     e.HasMorePages = false;
  }
 }
}
```

2.2 Точки, размеры и прямоугольные области

Фундаментальное понятие для любой графической системы – способ представления позиций и размеров. *GDI+* использует специальные объекты, которые позволяют определить точку в двумерном пространстве, размер или прямоугольную область.

Координаты в *GDI+* представлены структурами *Point* или *PointF*. Обе структуры содержат члены *x* и *y*, которые хранят координаты для двух осей поверхности рисования. Структуры *Point* хранят значения как целые числа, а структуры *PointF* хранят их как значения с плавающей точкой. Необходимо помнить, что система *GDI+* внутри себя использует значения с плавающей точкой, таким образом, значения в структуре *Point* будут интерпретироваться системой рисования как типы *float* или *Single* с целью создания графики.

Некоторые методы рисования, такие как *DrawLine*, перегружаемы для того, чтобы принять либо значения координат *x*, *y*, либо две структуры

Point или *PointF* в качестве параметров. Следующий листинг показывает две команды рисования линии, которые функционально идентичны.

```
1)myGraphics.DrawLine(Pens.Black,10,20,210,50);
2)Point p1 = new Point(10,20);
  Point p2 = new Point(210,50);
  myGraphics.DrawLine(Pens.Black,p1,p2);
```

Как можно заметить, первая команда рисования линии принимает дискретные значения, а вторая берет точки, которые были созданы ранее.

Размеры в *GDI+* задаются шириной и высотой. Так же, как в *Point* и *PointF*, размер предоставляется в двух вариантах: *Size* и *SizeF*, где *Size* хранит ширину и высоту как целые числа, а *SizeF* хранит их как значения типа *float* или *Single*.

Фигуры, рисуемые на *GDI+* - поверхностях, такие как эллипсы или прямоугольники, определяются их местоположением, которое соответствует верхнему левом углу фигуры, и шириной и высотой, определяемых структурой *Size* или *SizeF*. Эти два параметра (расположение и размер) наиболее часто используются для создания определения *прямоугольной области*. Подобно *Points* и *Sizes*, прямоугольная область тоже имеет версии для целых чисел и чисел с плавающей точкой *Rectangle* и *RectangleF*.

Внутри прямоугольная область хранит позиции *x* и *y* верхнего левого угла, ширину и высоту. Они могут быть прочитаны или обработаны как *Location* и *Size*. Прямоугольная область также возвращает полезные параметры, такие как *Left, Right, Top* и *Bottom* в виде отдельных значений.

Данный код создает эффект, показанный на рисунке 2.3.

```
private void Form1_Paint(object sender, PaintEventArgs e){
    // Делим клиентскую область на маленькие квадраты
    SizeF smallSquareSize = new
    SizeF(0.1f*this.ClientRectangle.Width,
```

```
      0.1f*this.ClientRectangle.Height);
      // Создаем кисть
      SolidBrush sb = new SolidBrush(Color.White);
      // toggle – переключатель между черными и белыми
квадратами
      bool toggle = false;
      // Цикл из десяти итераций по оси y
      for(int y = 0; y < 10; y++) {
            // Десять шагов по горизонтали по оси x
            for(int x = 0; x < 10; x++) {
            // Выбираем цвет кисти
            if(toggle) sb.Color=Color.Black;
            else sb.Color = Color.White;
            // Создаем квадрат
         RectangleF rc = new RectangleF(x*smallSquareSize.Width,
                                        y*smallSquareSize.Height,
                                        smallSquareSize.Width,
                                        smallSquareSize.Height);
            // Заливаем квадрат выбранным цветом
            e.Graphics.FillRectangle(sb,rc);
            // Изменяем цвет
            toggle = !toggle;
            }
      // Изменяем цвет при достижении конца строки квадратов
      toggle = !toggle;
      }
      sb.Dispose();// Утилизируем кисть
}
```

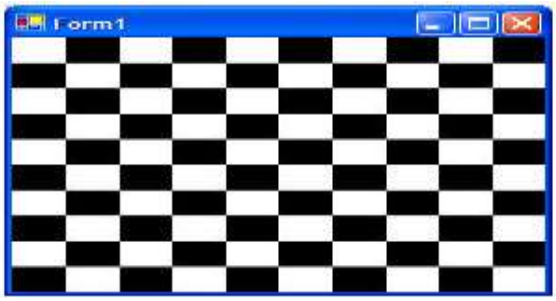

Рис. 2.3 – Эффект шахматной доски

Методы *GDI+* везде используют структуры *Point, PointF, Size, SizeF, Rectangle* и *RectangleF*. Массивы структур *Point* могут быть использованы для определения многоугольников; прямоугольные области используются для фигур, границ объектов или определения областей, где возможно рисование.

Используя метод *Contains, Rectangle* и *RectangleF* позволяют проверить, находится ли заданная точка внутри прямоугольника. Это полезно для тестирования на попадание в границы объектов. Можно получить объединение (Union) двух прямоугольных областей – и появится одна большая прямоугольная. С помощью метода *Intersect* можно получить прямоугольную область, представляющую собой пересечение двух других прямоугольных областей, а с помощью метода *IntersectsWith* – определить, перекрывает ли одна прямоугольная область другую. Прямоугольная область может быть перемещена физически с использованием метода *Offset,* увеличена или уменьшена в размере с помощью метода *Inflate*.

2.3 Цвета

Любая современная графическая система должна иметь возможности определения и использования цвета. Традиционно цвета определяются как смесь красного, зеленого и синего, на основе которых создаются основные аддитивные цвета. Цвета всегда должны рассматриваться как смесь этих трех в определенном процентном соотношении. Например, 0 % от всех трех цветов – черный. 100 % от всех трех – белый.

Значения каждого из основных цветов могут быть выражены по-разному. Иногда можно увидеть значение цвета, которое является значением с плавающей точкой в диапазоне от 0 до 1, где 0 – это 0 %, а 1 – это 100 %. Иногда можно увидеть основной цвет, выраженный в виде целого значения, в диапазоне от 0 до 255. Это происходит потому, что многие системы визуализации используют набор байт, содержащий

цветовые значения, и потому что максимальное число, которое может содержать байт, равно 255. В этом случае 0 % = 0, а 100 % = 255.

2.3.1 *Alpha*-прозрачность

Цветовая система в *GDI+* добавляет еще один уровень сложности – *Alpha*. *Alpha* – уровень прозрачности цвета. Его следует принимать за процент, где 0 % *Alpha* – непрозрачность, а 100 % *Alpha* – полная прозрачность. Всякий раз, когда создаете цвет, есть возможность использовать *Alpha*-настройки, которые сделают, если захотите, кисти и перья полупрозрачными.

2.3.2 Структура *Color*

Структура *Color* содержит полный набор разделяемых или статических свойств, определяющих большое количество цветов, которые можно легко использовать в своих программах без необходимости определения цвета. На рисунке 2.4 показаны эти цвета и их названия.

Рис. 2.4 – Именованные цвета

В дополнение к этому набору система определяет цвета, которые зависят от предпочтений пользователей, такие как цвет заголовка окна или фона окна. Например, системная таблица цветов на рисунке 2.5. На другом компьютере рисунок может выглядеть иначе.

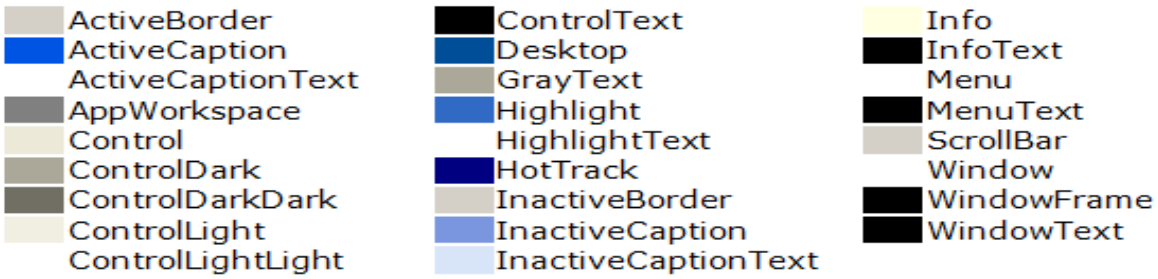

Рис. 2.5 – Системные цвета

Структура *Color* содержит четыре основных и пять дополнительных свойств для управления цветом или получения информации о нем:

1) *A* – Alpha-компонент, который определяет прозрачность цвета;

2) R – красная составляющая цвета;

3) G – зеленая составляющая цвета;

4) B – синяя составляющая цвета;

5) *Name* – одно из известных названий цветов: red, tan и т. п.;

6) *IsEmpty* – равен *true*, если *Color* создана, но не инициализирована;

7) *IsKnownColor* – имеет значение *true*, если этот цвет был создан из предопределенного цвета с помощью либо метода *FromName*, либо метода *FromKnownColor* из списка цветов, предоставляемых системой;

8) *IsNamedColor* – имеет значение true, если этот цвет был создан с помощью либо метода *FromName*, либо метода *FromKnownColor* из списка именованных цветов, предоставляемых системой;

9) *IsSystemColor* – равен true, когда цвет был создан из одного из специальных системных цветов, таких как *ActiveBorder* или *HotTrack*.

2.3.3 Создание цвета

Используя метод *FromArgb* структуры *Color*, можно создавать свои собственные цвета. При помощи одной из перегрузок этого метода можно создать любую комбинацию цвета и прозрачности, указав точные значения желаемого цвета. Если нет необходимости использовать прозрачность, то можно применить *Color.FromArgb (<r>, <g>, <b>)*, где цветовые компоненты находятся в диапазоне от 0 до 255. При необходимости, можно использовать *Color.FromArgb (<a>, <some-colour>)*. Перегрузка *Color.FromArgb (128, Color.Pink)* создает полупрозрачный розовый цвет. Кроме того, есть возможность задать весь цветовой набор ARGB, используя *Color.FromArgb (<a>, <r>, <g>, <b>)*.

2.3.4 Использование цвета

Самый распространенный способ использования цветов – это использование их в качестве параметра при создании кистей и перьев. Чтобы заполнить область с применением именованных цветов, нужно выбрать одно из известных значений, таких как *Color.Red* или *Color.DarkGreen*. Также можно брать собственные созданные цвета, используя один из методов заливки и объект *Brush*. На рисунке 2.6 показано приложение, которое случайными цветами заполняет фигуры.

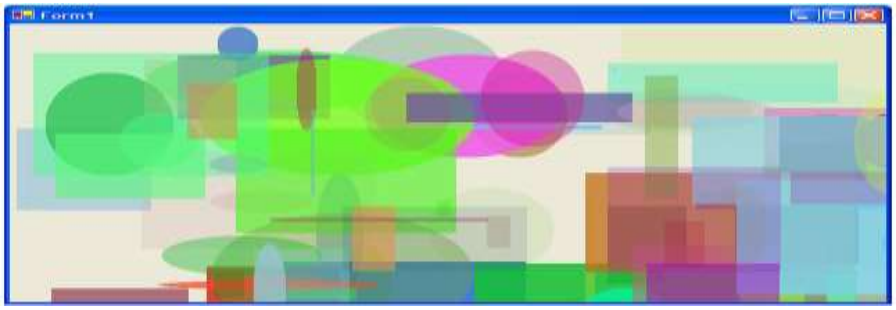

Рис. 2.6 – Фигуры, закрашенные случайными цветами и прозрачностями

Код, создающий этот эффект, приведен в листинге:

```
private void Form1_Paint(object sender, PaintEventArgs e)  {
     Random r = new Random(1);
     for(int c = 0; c < 100; c++) {
     SolidBrush sb = new
SolidBrush(Color.FromArgb(r.Next(8)*32,
     r.Next(255),r.Next(255),r.Next(255)));
     if(r.Next(2) == 1) {
          e.Graphics.FillRectangle(sb,
          r.Next(this.ClientRectangle.Width),
          r.Next(this.ClientRectangle.Height),
          1+r.Next(200), 1+r.Next(200));
     } else {
          e.Graphics.FillEllipse(sb,
          r.Next(this.ClientRectangle.Width),
          r.Next(this.ClientRectangle.Height),
          1+r.Next(200), 1+r.Next(200));
     }
     sb.Dispose();
     }
}
```

Лабораторная работа № 2

Системы координат, цвета. Построение графиков функций

Цель работы:

1. Получить навыки работы с координатными пространствами.
2. Получить навыки работы с цветом.
3. Научиться строить графики математических функций.

Контрольные вопросы по теме:

1) Что такое одиночная координата на поверхности объекта Graphics?

2) Что это означает независимость GDI+ от разрешения?

3) Перечислите и опишите координатные пространства GDI+.

4) Перечислите стандарты страничного пространства GDI+.

5) Какие структуры используются GDI+ для определения позиций и размеров? Опишите внутреннее содержание этих структур.

6) Как определяются цвета?

7) Что такое Alpha-прозрачность?

8) Приведите описание структуры Color.

9) Как создать свой собственный цвет? Как использовать именованные, системные и созданные собственные цвета?

Задания:

Разместите на форме *PictureBox* три кнопки для вывода графика по стандартам *Pixel*, *Millimeter*, *Inch* и кнопку очистки компонента *PictureBox*.

Выберите самостоятельно математический интервал по оси *x*, на котором будет строиться график функции. Например, для sin(x) возьмите интервал от -2π до 2π. Центр координат расположите в центре *PictureBox*.

При нажатии на кнопки, используя метод *DrawLine,* нарисуйте обрамляющую рамку для *PictureBox* и координатные оси. Цвет рамки и осей установите в один из *именованных* цветов.

Свойство *BackColor* для *PictureBox* установить методами *Color.*FromKnownColor и *Color.FromName*, например, *Color.FromKnownColor (KnownColor.ControlLightLight)* и *Color.FromName ("Info")* в один из *системных* цветов.

По нажатию на каждую из трех кнопок выведите график функции, указанной в варианте задания, используя соответствующий стандарт страничного пространства: *Pixel*, *Millimeter* и *Inch*. Цвет графика задает пользователь, используя метод *Color.FromArgb*, например, *Color.FromArgb*(210,224,200).

Очистку *Graphics* выполните цветом, созданным пользователем.

Пример вывода графика функции sin(x) показан на рисунке 2.7.

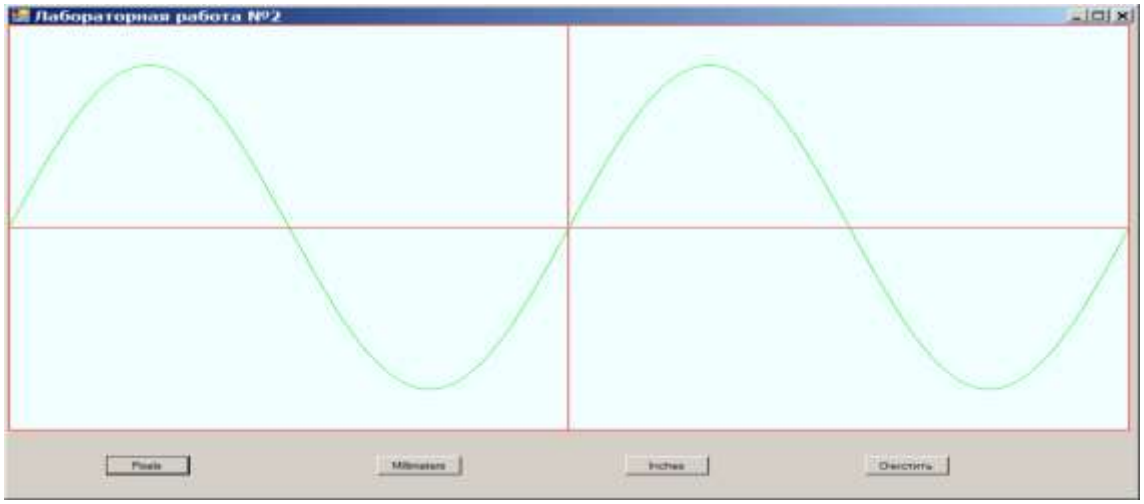

Рис. 2.7 – График функции sin(x)

Варианты индивидуальных заданий:

1. Постройте графики линейной функции y = k * x для k = 0.1, 0.2, 0.3. Свойство *Size* элемента управления *PictureBox* установите в (1000, 500).

2. Постройте график функции $y = 3 * x^2 + 1$. Свойство *Size* элемента управления *PictureBox* установите в (900, 450).

3. Постройте график функции $y = x^3 + 2 * x^2 + x$. Свойство *Size* элемента управления *PictureBox* установите в (800, 400).

4. Постройте график функции $y = \sin(x)$. Свойство *Size* элемента управления *PictureBox* установите в (1000, 500).

5. Постройте график функции $y = -6 * x^2 + 3 * x$. Свойство *Size* элемента управления *PictureBox* установите в (1000, 500).

6. Постройте график функции y = x5. Свойство *Size* элемента управления *PictureBox* установите в (900, 450).

7. Постройте график функции $y = \cos(x-1) + |x|$. Свойство *Size* элемента управления *PictureBox* установите в (800, 400).

8. Постройте график функции $y = 4 * x^2 + 3 * x$. Свойство *Size* элемента управления *PictureBox* установите в (1000, 500).

9. Постройте график функции $y = 2 * x^3 + 2 * x$. Свойство *Size* элемента управления *PictureBox* установите в (900, 450).

10. Постройте график функции $y = \cos(x)$. Свойство *Size* элемента управления *PictureBox* установите в (800, 400).

11. Постройте график функции $y = -3 * x + 6 * x^2$. Свойство *Size* элемента управления *PictureBox* установите в (1000, 500).

12. Постройте график функции $y = x^3 + x - 4$. Свойство *Size* элемента управления *PictureBox* установите в (900, 450).

13. Постройте график функции $y = \sin(x - 1) - |x|$. Свойство *Size* элемента управления *PictureBox* установите в (800, 400).

14. Постройте график функции $y = x^2 - x + 12$. Свойство *Size* элемента управления *PictureBox* установите в (1000,500).

15. Постройте график функции $y = -10 * x^3 - 5 * x + 2$. Свойство *Size* элемента управления *PictureBox* установите в (900, 450).

Порядок выполнения лабораторной работы:

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

3 Работа с текстом

Текст в *GDI+* легко использовать. *GDI+* обладает такими опциями визуализации, которые обеспечивают четкий и читаемый вид, а также предлагают различные варианты расположения текста на экране.

Есть три основных понятия, которые важны для понимания элементарной типографики для приложений *GDI+* и *Windows Forms*. Это используемые размеры, способ использования шрифтов и способ форматирования вывода. На рисунке 3.1 показаны некоторые общие термины типографики по отношению к различным шрифтам.

Рис. 3.1 – Типографический пример

3.1 Размеры

Размер шрифта обычно выражается в пикселях. Размер шрифта – это его высота, ограниченная «М-квадрат». Традиционно буква М в верхнем регистре была самым крупным символом шрифта. Сейчас это не совсем так, но название осталось. Объект *Font* – это *GDI+* контейнер для шрифта, созданный из *FontFamily*, который, содержит описание шрифта.

Расположение текста на экране выполняется в точках или каким-то из методов размещения текста в мировых координатах.

Важно помнить, что задание размещения текста в пикселях нежелательно, так как размер пикселя на экране отличается от размера печатного пикселя. Поэтому лучше использовать систему измерения в мировых координатах. Так как *Points* – единицы измерения для принтеров, то это очевидный выбор для координат размещения текста. Ничто не мешает использовать другие координаты, но все же стоит быть в курсе всех тонкостей отношений между этими системами и размерами.

Шрифты идентифицируются не только по названиям, но и по вертикальным размерам (кеглям), измеряемым в пунктах. В традиционной типографике 1 пункт (пт) равен 0,01384 дюйма. Это около 1/72 дюйма, поэтому в компьютерной типографике принято считать, что 1 пт в точности равен 1/72 дюйма.

Рендеринг шрифтов – очень сложная тема. В общем случае, фонт-файл – это совокупное описание коллекции *глифов* (*глиф* – конкретное графическое представление графемы), которые описывают *графемы* (*графема* – единица представления письменной речи) в шрифте. Некоторые символы используют более одного глифа, и даже возможно, что глиф, используемый для каждого символа, может меняться в зависимости от символа, который предшествует ему. Когда создается шрифт с указанием *Font Family* и *Size*, система создает описания всех глифов шрифта, адаптированных под размер и разрешение устройства вывода.

Именно поэтому *true-type*-шрифт может сохранять свое качество при кегле 10 пт или 1/7 дюйма в высоту и хорошо выглядеть при кегле 144 пт или 2 дюйма в высоту. На рисунке 3.2 это показано разницей между кеглями 10 пт, 18 пт и 36 пт.

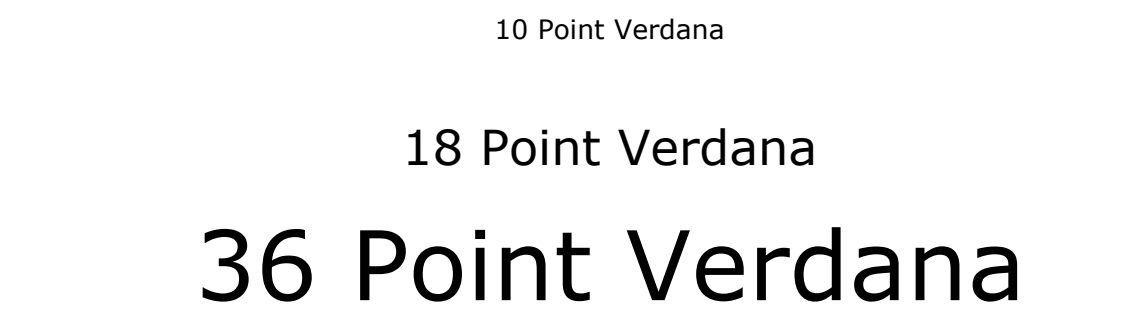

Рис. 3.2 – Рендеринг шрифтов различных размеров

Заметно, что качество типа 10 пт хорошее, невзирая на то, что количество пикселей, используемых для отображения типа, небольшое. Когда имеется в наличии больше пикселей, визуализатор использует набор построенных глифов, чтобы воспользоваться преимуществом дополнительного разрешения, и качество шрифта остается хорошим.

3.2 Использование шрифтов

Вначале необходимо создать объект *Font* с указанием нужного шрифта. Это можно осуществить несколькими способами, однако простейший конструктор шрифта не требует ничего, кроме имени семейства шрифтов и размера.

```
Font fn = new Font("Times New Roman",10);
```

Эта строка кода создает шрифт с параметрами по умолчанию и устанавливает размер 10 пт в высоту. Когда шрифт создан, можно использовать его, чтобы нарисовать строку на любом объекте *Graphics* с помощью метода *DrawString*.

DrawString имеет несколько перегрузок, которые позволяют поместить текст в определенную позицию или в целевую прямоугольную область с различными опциями форматирования. Простейший вариант заключается в использовании мировых координат и выбранных в текущий

момент страничных единиц, чтобы расположить текст. Следующий листинг печатает «Hello World!» на форме:

```
private void Form1_Paint(object sender, PaintEventArgs e) {
    Font fn = new Font("Times New Roman",10);
    e.Graphics.DrawString("HelloWorld!",fn,Brushes.Black,10,10);
    fn.Dispose();
}
```

На рисунке 3.3 показан результат работы приложения.

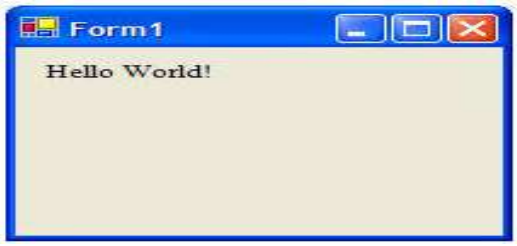

Рис. 3.3 – Простое отображение текста

Текст в примере выводится в левом верхнем углу в позиции (10,10). Форматирование текста не предусмотрено. *GDI+* предоставляет несколько способов форматирования абзаца текста в заданной области.

3.3 Форматирование

Есть два объекта, которые управляют размещением форматированного текста. Для задания параметров форматирования используются прямоугольная область вывода и объект *StringFormat*. Простой абзац текста может быть отформатирован в заданной прямоугольной области довольно просто, а объект *StringFormat* позволяет выбрать горизонтальное и вертикальное выравнивание плюс некоторые другие параметры, которые влияют на способ отображения текста. Есть три основных формата: *Near, Center* и *Far*. Они называются так потому, что некоторые языки требуют выравнивания по правой границе и

ориентации чтения справа налево для нормального текста. Таким образом, эквивалентом выравнивания по левому краю абзаца в западноевропейских языках будет выравнивание по правому краю как, например, в иврите. Поэтому выравнивания называют *Near* (для выравнивания по начальной части страницы) и *Far* – по конечной части страницы.

На рисунке 3.4 показан абзац текста, отформатированный по центру клиентской прямоугольной области формы.

Рис. 3.4 – Форматированный текст

Когда приложение, работа которого показана на рис. 3.4, изменяет размеры, текст автоматически обновляется, чтобы соответствовать прямоугольнику, который на 20 пикселей меньше по высоте и ширине, чем клиентская прямоугольная область формы. Текст центрируется горизонтально с помощью объекта *StringFormat* со свойством *Alignment*, установленным в *StringAlignment.Center*. Он также центрируется по вертикали установкой свойства *LineAlignment* объекта *StringFormat* в значение *StringAlignment.Center*.

В дополнение к физическому выравниванию, объект *StringFormat* может быть настроен для отображения символа многоточие "..." каждый раз, когда не может поместить строку в область. Это контролируется свойством *StringFormat.Trimming* и членами перечисления *StringTrimming*.

Код, показанный в следующем листинге, – обработчик события *Paint* для формы, показанной на рисунке 3.4. Обратите внимание, что для того

чтобы заставить это приложение перерисовать свой текст при изменении размеров окна, форма определяет настройку стиля управления *ControlStyles.ResizeRedraw* в конструкторе формы.

```
private void Form1_Load(object sender, EventArgs e) {
      SetStyle(ControlStyles.ResizeRedraw, true);
}
private void Form1_Paint(object sender, PaintEventArgs e) {
      Font fn = new Font("Times New Roman",10);
      string str =
      "Lorem ipsum dolor sit amet, consectetuer adipiscing elit."+
      "Quisque dolor leo, sollicitudin a, porta vel, faucibus id,
nunc...";
      StringFormat sf =
      (StringFormat)StringFormat.GenericTypographic.Clone();
      sf.Alignment = StringAlignment.Center;
      sf.LineAlignment = StringAlignment.Center;
      sf.Trimming = StringTrimming.EllipsisWord;
      e.Graphics.DrawString(str, fn, Brushes.Black,
      new RectangleF(10,10,this.ClientRectangle.Width-20,
                     this.ClientRectangle.Height-20), sf);
      fn.Dispose()
}
```

Для вертикального вывода строки текста необходимо внести в программу указанную строку кода:

```
sf.FormatFlags = StringFormatFlags.DirectionVertical;
```

Итак, из этого видно, что шрифты управляются в объекте *Font*, что они, как правило, измеряются в пунктах и могут быть нарисованы с форматированием или без форматирования с использованием метода *DrawString* и необязательных объектов *StringFormat*.

Лабораторная работа №3

Работа с текстом

Цель работы:

1. Изучить управление шрифтами.

2. Получить навыки вывода текстовых сообщений различными шрифтами.

Контрольные вопросы по теме:

1) Что такое размер шрифта и в чем он выражается?
2) Что представляет собой объект *Font*?
3) Почему нежелательно задавать размещение текста в пикселях?
4) Что такое кегль и в чем он измеряется?
5) Что представляет собой фонт-файл?
6) Приведите пример простейшего конструктора шрифта.
7) Какие объекты управляют размещением форматированного текста?
8) Перечислите основные свойства объекта *StringFormat*.
9) Что задают форматы Near, Center и Far?
10) Что необходимо сделать, чтобы заставить приложение перерисовать свой текст при изменении размеров окна?

Задание:

Напишите программу, которая создавала бы на диске текстовый файл и записывала в него указанное количество строк. Открывала бы существующий файл, считывала строки и выводила три группы строк с указанными в варианте номерами строк каждой группы, шрифтами, стилем, размером, направлением вывода и выравниванием. Пример вывода показан на рисунке 3.5.

Рис. 3.5 – Вывод трех групп строк

Во всех вариантах заданий подберите размер символов таким образом, чтобы все строки помещались в объекте *Graphics* и не перекрывали друг друга.

Варианты индивидуальных заданий:

№	Строки	Шрифт	Стиль	Размер	Направление	Выравнивание
1	1–9	Times New Roman	Bold	24	Горизонтально	Center, Near
	10–14	Courier New	Regular	36	Вертикально	Center, Near
	15	Arial	Italic	1 inch	Горизонтально	Far, Near
2	1–6	Calibri	Strikeout	36	Вертикально	Near, Near
	7–11	Consolas	Bold	24	Горизонтально	Far, Near
	12	Corbel	Underline	0,5 inch	Горизонтально	Center, Near
3	1–6	Imprint MT Shadow	Regular	24	Горизонтально	Near, Far
	7–13	Arial Black	Bold	18	Горизонтально	Center, Far
	14	Corbel	Italic	1,5 inch	Вертикально	Near, Center
4	1–8	Magneto	Bold	18	Горизонтально	Center, Far
	9–10	Perpetua	Italic	24	Вертикально	Near, Far
	11	Cambria	Regular	2 inch	Горизонтально	Center, Near
5	1–6	Arial	Underline	36	Вертикально	Center, Center
	7–9	Broadway	Regular	24	Горизонтально	Far, Near
	10	Times New Roman	Strikeout	0,5 inch	Горизонтально	Near, Far
6	1–9	Corbel	Italic	24	Горизонтально	Center, Near
	10–14	Imprint MT Shadow	Regular	36	Вертикально	Center, Near
	15	Arial Black	Bold	1 inch	Горизонтально	Far, Near
7	1–6	Cambria	Regular	36	Вертикально	Near, Near
	7–11	Magneto	Bold	24	Горизонтально	Far, Near
	12	Perpetua	Italic	0,5 inch	Горизонтально	Center, Near
8	1–6	Arial	Underline	24	Горизонтально	Near, Far
	7–13	Times New Roman	Strikeout	18	Горизонтально	Center, Far
	14	Broadway	Regular	1,5 inch	Вертикально	Near, Center

Окончание таблицы

9	1–8	Courier New	Regular	18	Горизонтально	Center, Far
	9–10	Arial	Italic	24	Вертикально	Near, Far
	11	Times New Roman	Bold	2 inch	Горизонтально	Center, Near
10	1–6	Consolas	Bold	36	Вертикально	Center, Center
	7–9	Calibri	Strikeout	24	Горизонтально	Far, Near
	10	Corbel	Underline	0,5 inch	Горизонтально	Near, Far
11	1–9	Broadway	Regular	24	Горизонтально	Center, Near
	10–14	Arial	Underline	36	Вертикально	Center, Near
	15	Times New Roman	Strikeout	1 inch	Горизонтально	Far, Near
12	1–6	Perpetua	Italic	36	Вертикально	Near, Near
	7–11	Magneto	Bold	24	Горизонтально	Far, Near
	12	Cambria	Regular	0,5 inch	Горизонтально	Center, Near
13	1–6	Arial Black	Bold	24	Горизонтально	Near, Far
	7–13	Corbel	Italic	18	Горизонтально	Center, Far
	14	Imprint MT Shadow	Regular	1,5 inch	Вертикально	Near, Center
14	1–8	Calibri	Strikeout	18	Горизонтально	Center, Far
	9–10	Consolas	Bold	24	Вертикально	Near, Far
	11	Arial	Underline	2 inch	Горизонтально	Center, Near
15	1–6	Courier New	Regular	36	Вертикально	Center, Center
	7–9	Times New Roman	Bold	24	Горизонтально	Far, Near
	10	Arial	Italic	0,5 inch	Горизонтально	Near, Far

Порядок выполнения лабораторной работы:

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

4 Перья

Перья используются для рисования линий, многоугольников, кривых и таких фигур, как прямоугольники и эллипсы. Они могут быть и очень простыми, и достаточно сложными. Во многих вышеприведенных примерах простые перья были использованы для иллюстрации рисования фигур.

Кажется, что не трудно рисовать линии на графической поверхности. Однако для *GDI+* это далеко не так. «Простой» процесс рисования линий на самом деле очень усложнен богатством возможностей из-за огромного количества свойств, включающих ширину пера, стиль, тип и форму окончания линии, способ ее заполнения.

4.1 Стандартные перья и их ширина

Пространство имен *System.Drawing* обеспечивает стандартный набор перьев, которые доступны для использования. Они могут быть найдены в массиве *Pens*. Для каждого системного цвета предназначено одно доступное перо. Все эти перья шириной в одну единицу. Рисование линии одним из имеющихся в наличии пером осуществляется таким образом:

```
graphics.DrawLine(Pens.Black, 0,0,100,100);
```

Следует помнить, что *GDI+* является системой независимого разрешения, поэтому единицей ширины пера может быть дюйм, миллиметр, точка или пиксель в зависимости от настроек свойства *Graphics.PageUnit*. Если нужно перо шириной не в одну единицу, то необходимо самому его создать.

Это делается путем построения пера с цветом или кистью и значением с плавающей точкой, которое определяет его ширину. Не стоит забывать, что единицей ширины пера будет 1 дюйм, если свойство

Graphics.PageUnit установлено в дюймах, когда потребуется перо, скажем, 1/20 дюйма ширины, чтобы создать нужную линию. Поэтому свойство *Width* пера – значение с плавающей точкой.

Как конструктор *Pen* принимает настройку ширины, показано в следующем листинге. В этом случае перо будет шириной 5 единиц:

```
Pen p = new Pen(Color.Red,5);
```

Чтобы заставить *Pen* нарисовать линию шириной в 1 пиксель, независимо от страничных единиц, надо ширину пера установить в –1.

4.2 Линии в *GDI+*

Все линии, на самом деле, – сложные фигуры, которые создаются и заполняются конкретной кистью. Даже когда отображается линия только в 1 пиксель, линии рисуются сложным образом. Чтобы проиллюстрировать это, на рисунке 4.1 показана одна и та же линия пикселей при нескольких увеличениях.

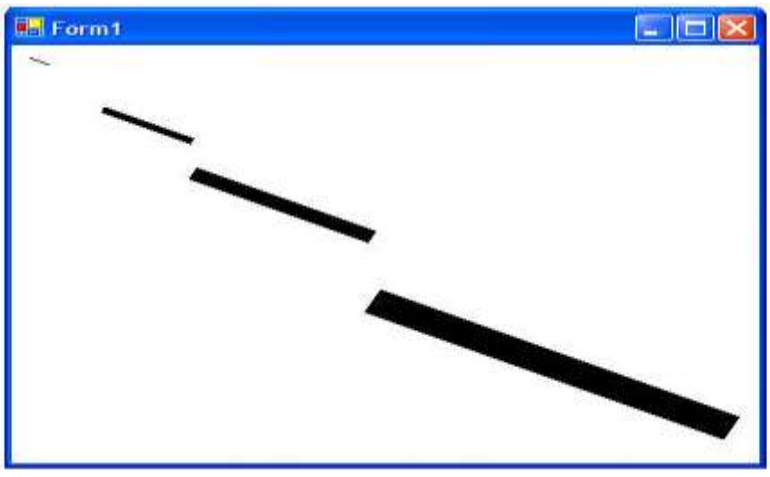

Рис. 4.1 – Одно и то же перо при разных увеличениях

Хорошо видно, что по мере увеличения от 1 до 20-ти раз, форма линии видится как прямоугольник. Так, линия – это прямоугольник? Нет, не совсем. Линия – фигурный объект, который может в некоторых случаях

быть прямоугольным, но форма концов линии зависит от типа используемых *наконечников*.

4.3 Наконечники линии

Почему наконечники важны? *Наконечники* позволяют привести в порядок конец линии, чтобы сделать его аккуратнее или красиво соединить с другими линиями. На рисунке 4.2 показан вывод с закругленными наконечниками.

Рис. 4.2 – Закругленные наконечники линии

На рисунке 4.3 показаны несколько линий, концы которых соединены квадратными наконечниками.

Рис. 4.3 – Квадратные наконечники линии

На рисунке 4.4 показаны те же линии, нарисованные с закругленными наконечниками. Преимущество очевидно.

Рис. 4.4 – Закругленные наконечники линии

Линия – это не просто ряд пикселей. В функцию перьев заложено больше, чем вывод строки точек по прямой линии. Они создают сложные области, которые могут быть заполнены любым цветом, градиентом или текстурой, сделанной из растрового изображения.

Помимо закругленных наконечников линий существует выбор и других форм, а также есть возможность создать свой собственный наконечник. Стандартные наконечники линий представлены на рисунке 4.5

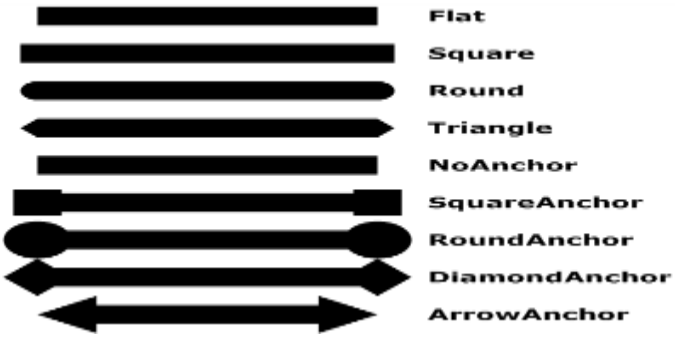

Рис. 4.5 – Линии и наконечники

Код, выводящий рисунок 4.5, указан ниже. Обратите внимание, что два члена перечисления: *LineCaps*, *Custom* и *AnchorMask* были исключены из иллюстрации.

```
// Этот файл использует директиву using System.Drawing.Drawing2D;
private void Form1_Paint(object sender, PaintEventArgs e) {
     Pen p = new Pen(Color.Black,20);
     int y = 20;
     foreach(string s in
     Enum.GetNames(typeof(System.Drawing.Drawing2D.LineCap))){
```

```
            p.StartCap = p.EndCap = (LineCap)Enum.Parse
            (typeof(System.DrawingDrawing2D.LineCap),s);
            e.Graphics.DrawLine(p,30,y,230,y);
            e.Graphics.DrawString(s,Font,Brushes.Black,260,y-10,
            StringFormat.GenericTypographic);
            y+ = 40;
      }
}
```

4.4 Точки и пунктирные линии

Перья могут быть оформлены по длине пунктирными линиями и замысловатыми перьевыми наконечниками. На рисунке 4.6 показан набор стандартных стилей линии, определенных в перечислении *DashStyle*.

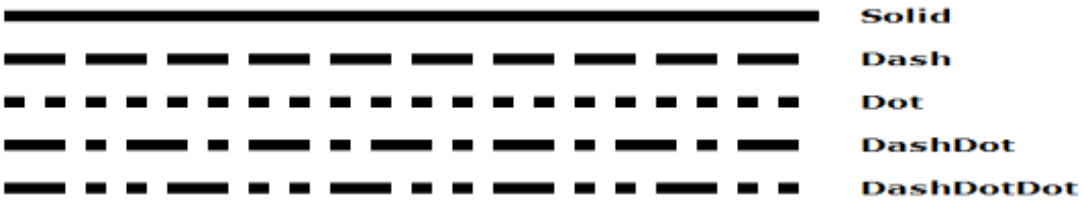

Рис. 4.6 – Стандартные штрих-стили

Код, рисующий это изображение:

```
// Это файл использует директиву using System.Drawing.Drawing2D;
private void Form1_Paint(object sender, PaintEventArgs e) {
      Pen p = new Pen(Color.Black,10);
      int y = 20;
      foreach(string s in
      Enum.GetNames(typeof(System.Drawing.Drawing2D.DashStyle)){
      p.DashStyle = (DashStyle)Enum.Parse(typeof(DashStyle),s);
      e.Graphics.DrawLine(p,20,y,420,y);
      e.Graphics.DrawString(s,Font,Brushes.Black,440,y-10,
                              StringFormat.GenericTypographic);
            y += 40;
      }     }
```

4.5 Комбинированные перья

Комбинированные перья позволяют легко рисовать причудливые границы и линии. Представьте себе ручку с таким широким пером, что можно разрезать его на кусочки и выбрать некоторые из них. Перо следует рассматривать как имеющее одну, неважно какую, единицу ширины. Параллельные линии создаются делением пера на доли, как показано на рисунке 4.7.

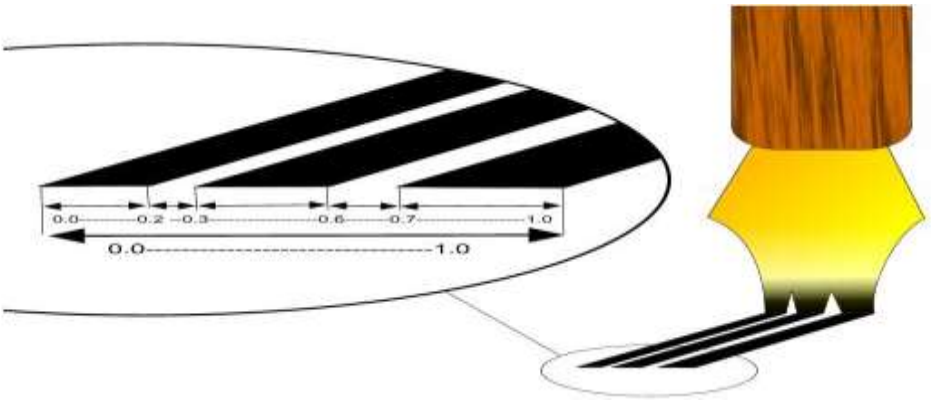

Рис. 4.7 – Комбинированное перо

Определение этого пользовательского пера обеспечивается массивом *float* или *Single*-значений, которые описывают расстояния между линиями и их толщину. После создания комбинированное перо может быть использовано для рисования причудливых рамок, как показано на рисунке 4.8.

Рис. 4.8 – Комбинированные перья в действии

Код, выводящий рисунок, приведен в нижеследующем листинге.

```
private void Form1_Paint(object sender, PaintEventArgs e) {
     Pen p = new Pen(Color.Black,20);
```

```
        p.CompoundArray = new float[]{0.0f,0.2f,0.3f,0.6f,0.7f,1.0f};
        e.Graphics.DrawRectangle(p,20,20,200,150);
        e.Graphics.DrawEllipse(p,250,20,430,150);
}
```

Из приведенных примеров становится очевидным, что перья более сложны, чем простые линии.

Лабораторная работа № 4

Рисование линий

Цель работы:

1. Изучить объекты для рисования линий.

2. Получить навыки изображения линий различных типов.

Контрольные вопросы по теме:

1) В каком массиве хранится стандартный набор перьев? Приведите пример рисования линии одним из имеющихся в наличии пером.

2) Какая величина может быть единицей ширины пера? Приведите пример построения пера заданной ширины и цвета.

3) Что представляют собой линии?

4) Перечислите виды стандартных наконечников линий.

5) Перечислите набор стандартных стилей линии, которые определены в перечислении *DashStyle*.

6) Что такое комбинированное перо?

7) Приведите пример рисования линии комбинированным пером.

Задание:

Для вариантов 1–6:

Получите кривую дракона n-го порядка. Каждой кривой ставится в соответствие последовательность, состоящая из нулей и единиц, где единица соответствует повороту кривой налево, а ноль – повороту направо. Кривая дракона первого порядка имеет двоичную формулу 1. Для того чтобы получить двоичную формулу кривой дракона каждого следующего порядка, необходимо приписать справа к формуле кривой предыдущего порядка единицу. Полученная последовательность дает половину искомой формулы. Затем в последовательности цифр,

предшествующих приписанной единице, замените нулем единицу, стоящую в ее середине, после чего припишите полученную последовательность справа от уже построенной части формулы.

Для кривой дракона 2-го порядка это выглядит таким образом:

а) 1; б) **1**1; в) 110.

Для кривой дракона 3-го порядка:

а) 110; б) **110**1; в) 1101100.

Для кривой дракона 4-го порядка:

а) 1101100; б) **1101100**1; в) 110110011100100.

Жирным шрифтом выделена последовательность цифр, предшествующая приписанной единице, в которой затем средняя единица меняется на ноль, и эта последовательность приписывается справа.

Для вариантов 7–15:

Даны целые числа: t1, t2, ..., t31. Последовательность значений t1, t2, ..., t31 задает график температур за март месяц. Постройте график температур. Отрезки прямых линий, лежащие выше горизонтальной прямой, соответствующей нулевой температуре, изображаются комбинированными линиями. Каждый отрезок задается своим массивом. Отрезки, превосходящие температуру 15° С, рисуются сплошной широкой линией. Отрезки прямых линий, лежащие ниже горизонтальной прямой, соответствующей нулевой температуре, изображаются сплошной линией нормальной толщины. Отрезки, соответствующие температуре ниже – 5° С, изображаются пунктирной широкой линией.

Варианты индивидуальных заданий:

1. Получите кривую дракона 3-го порядка. Кривая изображается комбинированной линией. Хвост рисуется точечной толстой линией.

2. Получите кривую дракона 4-го порядка. Кривая изображается прерывистой линией. Голова дракона рисуется комбинированной линией.

3. Получите кривую дракона 5-го порядка. Кривая изображается точечной линией. Хвост дракона рисуется комбинированной линией.

4. Получите кривую дракона 6-го порядка. Кривая изображается комбинированной линией. Голова дракона рисуется точечной линией.

5. Получите кривую дракона 7-го порядка. Кривая изображается прерывистой линией. Голова дракона рисуется комбинированной линией.

6. Получите кривую дракона 8-го порядка. Кривая изображается сплошной линией. Хвост дракона рисуется линией, определенной пользователем.

7. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
5	10	12	15	–2	–6	–10	2	8	20	21	18	10	11	4	–6
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	
3	5	9	0	–1	–7	0	5	7	8	6	12	16	20	20	

8. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
5	11	12	18	2	–6	–8	5	8	16	21	17	0	–1	4	–6
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	
3	3	3	0	–2	–8	0	17	7	9	16	10	16	19	12	

9. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
–6	1	2	5	–1	–8	–9	2	8	12	21	12	7	4	4	–3
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	
6	4	4	0	–1	–9	0	7	7	16	6	0	–6	15	18	

10. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
–8	–1	1	5	–2	–6	–10	2	9	0	18	18	10	12	4	–8
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	
3	3	19	10	–2	–6	0	4	5	4	12	16	8	5	7	

11. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
10	15	12	18	2	–2	–10	–2	0	8	12	14	17	11	4	–2
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	

3	–6	–9	0	1	7	0	8	17	18	16	10	6	5	0

12. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
–1	1	10	18	2	6	10	2	–1	–2	–8	0	3	7	9	16
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	
18	12	9	5	0	–6	–4	5	3	8	17	12	12	12	10	

13. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
–10	3	11	8	–2	9	15	–2	1	–3	–10	5	0	8	4	12
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	
–1	2	15	–1	5	0	–3	7	6	4	14	10	8	8	7	

14. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
1	–11	0	8	12	16	11	–2	–4	2	7	3	6	–5	7	10
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	
8	2	9	–5	1	0	–6	–5	–3	7	10	14	15	7	–1	

15. Постройте график температур:

t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**t9**	**t10**	**t11**	**t12**	**t13**	**t14**	**t15**	**t16**
–3	10	1	7	2	–6	10	–3	–8	–10	–8	1	0	9	–9	8
t17	**t18**	**t19**	**t20**	**t21**	**t22**	**t23**	**t24**	**t25**	**t26**	**t27**	**t28**	**t29**	**t30**	**t31**	
–3	4	10	0	–5	–3	4	–5	–3	7	12	–5	–2	0	7	

Порядок выполнения лабораторной работы:

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

5 Кисти и заполнения областей

Объект *Brush GDI+* используется для заливки областей цветом. Как и у *Pens*, у кистей есть довольно сложные модели поведения. Так же, как и у перьев, у кистей есть набор шаблонных объектов. Набор *Brushes* содержит одну сплошную кисть для каждого из стандартных цветов системы. К ним можно получить доступ с помощью, например, *Brushes.Red.*

5.1 Сплошная кисть

Базовый класс *Brush* является абстрактным. То есть он не может быть использован в изначальном виде, потому что не имеет некоторых деталей реализации, хотя способ работы интерфейсов класса определен. Конкретные примеры работающих кистей могут быть получены от базового класса *Brush,* и простейшим из них является объект *SolidBrush*.

Этот объект используется для рисования областей сплошной закраски и довольно прост в использовании. Все, что нужно сделать, это выбрать нужный цвет:

```
SolidBrush SB = new SolidBrush(Color.Blue);
```

На рисунке 5.1 показаны фигуры, заполненные сплошными кистями.

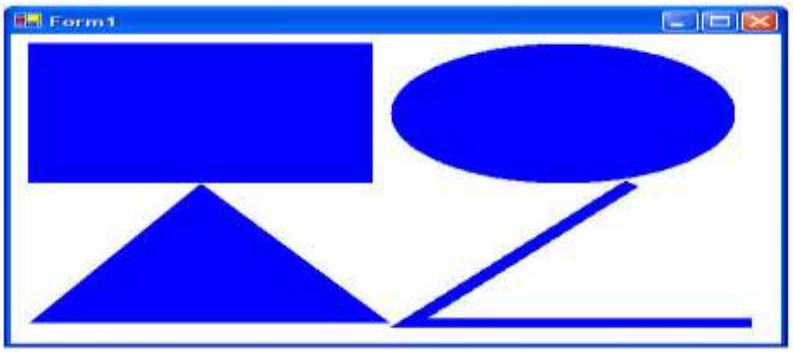

Рис. 5.1 – Фигуры, закрашенные с помощью *SolidBrush*

Код приведен в листинге:

```
private void Form1_Paint(object sender, PaintEventArgs e) {
     SolidBrush sb = new SolidBrush(Color.Blue);
     e.Graphics.FillRectangle(sb,10,10,200,150);
     e.Graphics.FillEllipse(sb,220,10,200,150);
     e.Graphics.FillPolygon(sb,new Point[]{
                    new Point(110,160), new Point(10,310),
                    new Point(220,310)});
     Pen p = new Pen(sb,10);
     e.Graphics.DrawLines(p,new Point[]{
     new Point(360,160),new Point(230,310),new Point(430,310)});
     p.Dispose();
     sb.Dispose();
}
```

Кисть может быть использована для заполнения контура, созданного объектом *Pen*. Перья могут создавать сложные формы, и эти формы могут быть заполнены любой кистью, созданной комбинацией *Pen-Brush*.

5.2 Штриховая кисть

Другой тип кисти получается от *HatchBrush*. *HatchBrush* может быть построен на основе одного из штриховых шаблонов, показанных на рисунке 5.2.

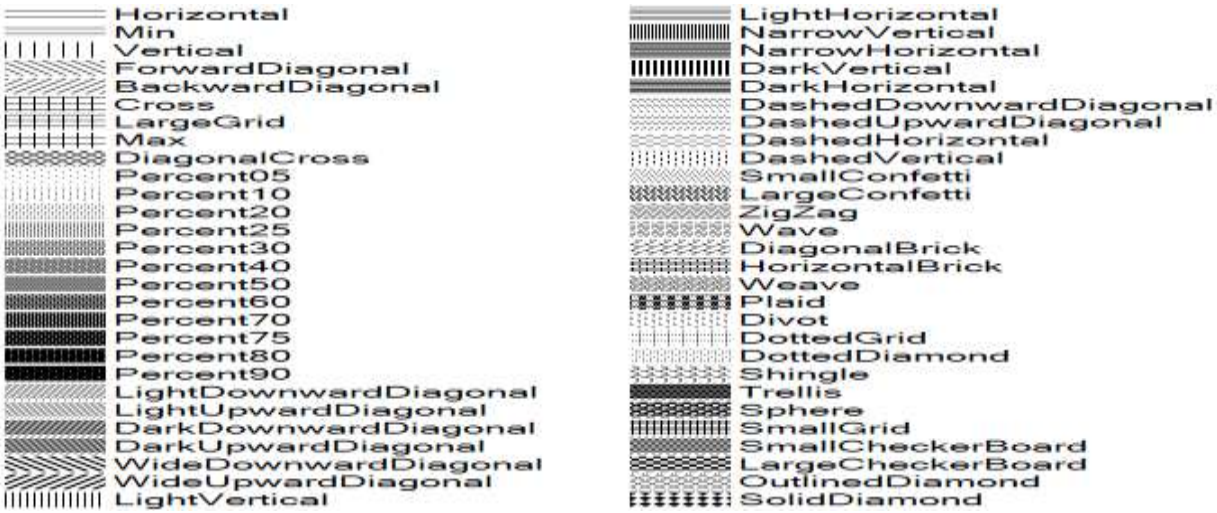

Рис. 5.2 – Стили штриховки

Эта кисть позволяет заполнить фигуры одним из стандартных стилей штриховки, предоставляемых системой. *Hatch*-шаблоны – это повторяющиеся квадратные картинки размером 8 x 8 пикселей, которые укладываются в мозаику, чтобы покрыть плоскость бесшовной текстурой.

Чтобы использовать штриховую кисть в коде, нужно создать кисть и выбрать цвет, в который будут окрашены передний план и фон. На рисунке 5.2 фон белый, а передний план – черный. На рисунке 5.3 показан вывод, идентичный использованному для рисования в примере с *SolidBrush*, но на этот раз для заполнения фигур используется *HatchBrush*.

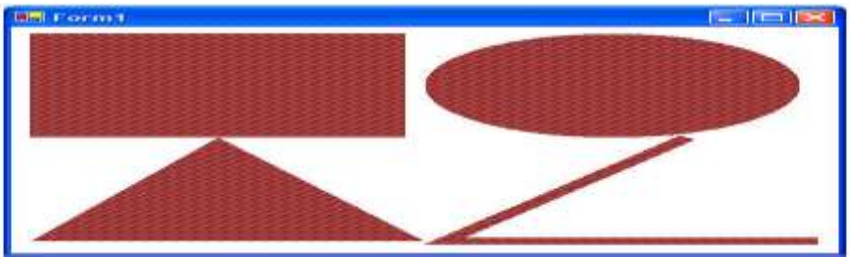

Рис. 5.3 – Те же фигуры, но другие кисти

Кисть использует стиль штриховки *DiagonalBrick* и применяет коричневый и серый цвет для фона и переднего плана соответственно. На листинге показан код генерирующий это изображение:

```
private void Form1_Paint(object sender, PaintEventArgs e) {
     HatchBrush hb = new HatchBrush(HatchStyle.DiagonalBrick,
                                    Color.Gray,Color.Brown);
     e.Graphics.FillRectangle(hb,10,10,200,150);
     e.Graphics.FillEllipse(hb,220,10,200,150);
     e.Graphics.FillPolygon(hb,new Point[]{
     new Point(110,160),new Point(10,310), new Point(220,310)});
     Pen p = new Pen(hb,10);
     e.Graphics.DrawLines(p,new Point[]{
     new Point(360,160),new Point(230,310),new Point(430,310)});
     p.Dispose(); hb.Dispose();
}
```

5.3 Рисование с изображениями

Последний тип кисти, который мы рассмотрим, – *TextureBrush*. Объект позволяет заполнить область содержимым какого-либо изображения. Это полезно, если есть возможность использовать бесшовные плитки текстуры, такие как изображения мрамора или кирпичной кладки. Текстура, показанная на рисунке 5.4, будет плиткой для покрытия любой области.

Рис. 5.4 – Мраморная плитка

Для *TextureBrush* требуется изображение для построения текстуры, которое можно загрузить с диска, как показано ниже.

```
Image img = Image.FromFile("imagefile.bmp");
TextureBrush tb = new TextureBrush(img);
```

Эта кисть может быть использована для рисования приведенного набора фигур, которые заполнены изображением вместо цвета или шаблона. На рисунке 5.5 показана работа приложения.

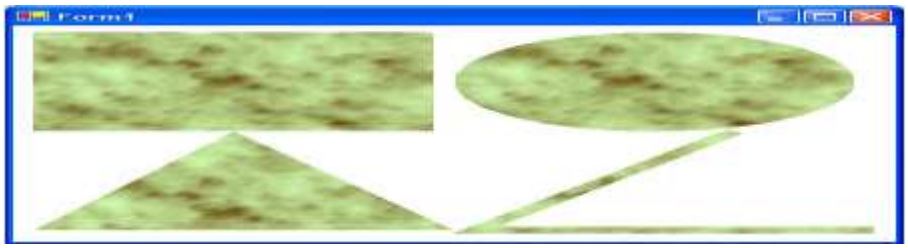

Рис. 5.5 – Плиточные текстуры

Если ни одна из встроенных кистей не подходит, можно создать собственную, определив ее как изображение bitmap размером 8 x 8 пикселей. Такая кисть может иметь любой внешний вид и любой цвет.

Лабораторная работа № 5

Кисти и заполнения областей. Построение коммерческих диаграмм

Цель работы:

1. Изучить объекты для заполнения областей.

2. Получить навыки закраски фигур.

Контрольные вопросы по теме:

1) Какой объект используется для заливки областей цветом?

2) Можно ли использовать базовый класс *Brush* в изначальном виде?

3) Какой объект используется для рисования областей сплошной закраски? Приведите пример его создания.

4) Для чего используется кисть *HatchBrush*?

5) Что такое *Hatch*-шаблон?

6) Приведите примеры нескольких штриховых шаблонов и пример создания штриховой кисти.

7) Для чего используется кисть *TextureBrush*? Приведите пример создания кисти *TextureBrush*.

8) Как создать свою собственную кисть?

Задания:

1. Для приведенной ниже задачи, выведите на экран поясняющую надпись внутри закрашенного определенным пользователем образцом закраски многоугольника (количество углов = **номер варианта * 2 + 1** для вариантов 1–7, количество углов = **номер варианта** для вариантов 8–15).

2. Выведите данные, приведенные в номере варианта, в виде столбчатой диаграммы. Рисунок должен включать в себя линию нулевого уровня, метки элементов диаграмм, вспомогательные линии (горизонтальные штриховые полосы для сравнения высот стержней

диаграмм). Прямоугольники столбчатой диаграммы заполните, используя разные виды кистей: *SolidBrush*, *HatchBrash* и *TextureBrush*.

Варианты индивидуальных заданий:

1. Средняя стоимость акций некоторой корпорации за восемь лет.
2. Средняя температура воздуха в г. Донецке за шесть месяцев.
3. Зарплата семи рабочих строительной бригады.
4. Результаты восьми экспериментов.
5. Оценки по программированию девяти студентов группы.
6. Минимальный уровень заработной платы в стране за восемь лет.
7. Стоимость основных фондов шести фирм.
8. Средний доход бюджета семи городов.
9. Цена автомобиля «ВАЗ» в восьми городах страны.
10. Грузоподъемность семи марок автомобилей.
11. Рейтинговые баллы девяти абитуриентов.
12. Количество студентов, поступивших в ДонНУ, за семь лет.
13. Количество студентов факультета девяти годов рождения.
14. Количество гаражей в каждом из семи гаражных кооперативов.
15. Количество сотрудников на каждом из семи факультетов вуза.

Порядок выполнения лабораторной работы:

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

6 Вычерчивание фигур. Управление изображениями

6.1 Рисование графических примитивов

Техника рисования линий на поверхности *Graphics* называется вычерчивание. Объект *Pen* будет следовать по указанной линии, чтобы создать контур фигуры. В случае открытой фигуры, у которой конечная точка не совпадает с начальной, линия будет иметь разные концы. Для замкнутой фигуры линия будет проведена от начальной точки по всей фигуре, пока она снова не достигнет начала и не закроет ее.

Для закрытых фигур (прямоугольники, эллипсы, многоугольники), созданных из массивов координат, область, ограниченная заданным контуром, может быть заполнена цветом, комбинированным цветом или картинкой. Заполнение осуществляется с помощью объекта *Brush*.

Объект *Graphics GDI+* включает набор методов, с помощью которых можно создать такие виды изображений:

1) *DrawLine* – линию из одной координаты в другую;
2) *DrawLines* – серию линий, определенных в массиве *x, y* координат;
3) *DrawRectangle* – прямоугольник (все прямоугольники определяются положением, шириной и высотой);
4) *DrawRectangles* – массив прямоугольников;
5) *FillRectangle* – заполненный прямоугольник;
6) *FillRectangles* – массив заполненных прямоугольников;
7) *DrawEllipse* – эллипс, заданный положением, шириной и высотой;
8) *FillEllipse* – заполненный эллипс;
9) *DrawPolygon* – многоугольник, заданный массивом *x, y* координат;
10) *FillPolygon* – заполненный многоугольник;
11) *DrawCurve* – cardinal-сплайн (последовательность отдельных кривых, объединенных в большую кривую), заданный массивом *x, y* координат;

12) *DrawClosedCurve* – замкнутый cardinal-сплайн;

13) *DrawBezier* – кривую Безье, определенную *x, y* координатами для узлов и контрольных точек;

14) *DrawPath* – объект *Path*. *Path* – набор линий и фигур в одном, простом для использования контейнере;

15) *FillPath* – заполненный объект *Path*;

16) *DrawArc* – сегмент дуги (дуга – это часть эллипса);

17) *DrawPie* – сектор, определенный эллипсом;

18) *FillPie* – заполненный сектор, определенный эллипсом;

19) *FillRegion* – заполнение внутренней области объекта *Region*, который описывает внутреннюю часть графической фигуры, состоящей из прямоугольников и контуров.

Изображение на рисунке 6.1 показывает некоторые из основных операций вычерчивания контуров и заполнения в *GDI+*.

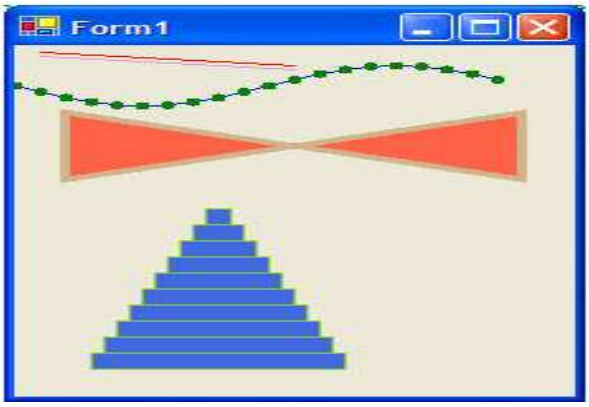

Рис. 6.1 – Некоторые из вычерчиваний и заполнений

На листинге показан код, который генерирует это изображение.

```
private void Form1_Paint(object sender, PaintEventArgs e) {
     Graphics g = e.Graphics; //Получить объект Graphics
     // Нарисовать линию, используя дискретные координаты
     g.DrawLine(Pens.Red,10,5,110,15);
     // Нарисовать линию, используя точки
     Point p1 = new Point(10,8); Point p2 = new Point(110,18);
```

```
        e.Graphics.DrawLine(Pens.Plum,p1,p2);
        // Нарисовать многоугольник, используя массив из PointF
        PointF[] pts = new PointF[20];
        float angle = 0;
        for(int x=0; x<20; x++) {
        pts[x] = new PointF(x*10,(float)(30+(15*Math.Sin(angle))));
        angle+=(float)Math.PI/10;
        }
        e.Graphics.DrawLines(Pens.Blue,pts);
        // Заполнить эллипс над каждой из точек синусоиды
        foreach(PointF p in pts)
        e.Graphics.FillEllipse(Brushes.Green,new RectangleF(p.X-3,
                                                    p.Y-3,6f,6f));
        // Нарисовать заполненный замкнутый многоугольник
        Point[] poly = new Point[5]{new Point(20,50),
             new Point(200,100),new Point(200,50),
             new Point(20,100),new Point(20,50)};
        e.Graphics.FillPolygon(Brushes.Tomato,poly);
        Pen pen = new Pen(Color.Tan,4);
        e.Graphics.DrawPolygon(pen,poly);
        pen.Dispose();
        // Создать, заполнить и нарисовать массив прямоугольников
        Rectangle[] rcs = new Rectangle[10];
        for(int x=1; x<11; x++)
        rcs[x-1] = new Rectangle(80-(x*5),110+(x*12),x*10,12);
        e.Graphics.FillRectangles(Brushes.RoyalBlue,rcs);
        e.Graphics.DrawRectangles(Pens.YellowGreen,rcs);
}
```

На рисунке 6.2 показано заполнение контейнера *Path*, включающего в себя сегмент дуги, cardinal-сплайн, строку текста и сектора эллипса.

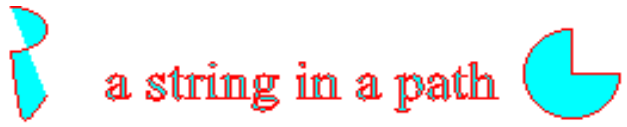

Рис. 6.2 – Пример использования объекта *Path*

На листинге показан код, который генерирует это изображение.

```
private void Form1_Paint(object sender, PaintEventArgs e) {
      Graphics g = e.Graphics; // Получить объект Graphics
      Pen p = new Pen(Brushes.Red, 1);
      SolidBrush mySolidBrush = new SolidBrush(Color.Aqua);
      // Создание объекта Path
      GraphicsPath myGraphicsPath = new GraphicsPath();
      Point[] myPointArray = {new Point(15,20),new Point(20,50),
                              new Point(30,40)};
      FontFamily myFontFamily = new FontFamily("Times New Roman");
      PointF myPointF = new PointF(50,20);
      StringFormat myStringFormat = new StringFormat();
      // Добавление в Path объектов Arc, Curve, String и Pie
      myGraphicsPath.AddArc(0,0,30,20,-90,180);
      myGraphicsPath.AddCurve(myPointArray);
      myGraphicsPath.AddString("a string in a path",
                myFontFamily,0,24,myPointF,myStringFormat);
      myGraphicsPath.AddPie(230,10,40,40,0,270);
      // Рисование и заливка объекта Path
      g.FillPath(mySolidBrush, myGraphicsPath);
      g.DrawPath(p, myGraphicsPath);
}
```

На рисунке 6.3 показан еще один пример использования объекта *Path*, демонстрирующий отличие между *Curve* и *ClosedCurve*.

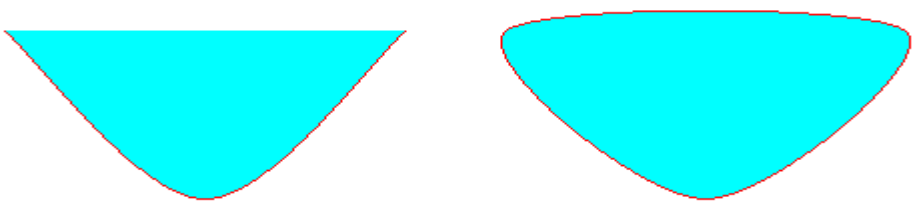

Рис. 6.3 – Откртый и замкнутый cardinal-сплайн

Код, генерирующий это изображение:

```
private void Form1_Paint(object sender, PaintEventArgs e) {
     Graphics g = e.Graphics; // Получить объект Graphics
     Pen p = new Pen(Brushes.Red, 1);
     SolidBrush mySolidBrush = new SolidBrush(Color.Aqua);
     // Создание объекта Path
     GraphicsPath myGraphicsPath = new GraphicsPath();
     // Массивы координат cardinal-сплайнов
     Point[] myPointArray_1 = { new Point(100,300),
                   new Point(200,400), new Point(300,300)};
     Point[] myPointArray_2 = { new Point(350,300),
                      new Point(450,400), new Point(550,300)};
     // Добавление в Path объектов Curve и ClosedCurve
     myGraphicsPath.AddCurve(myPointArray_1);
     myGraphicsPath.AddClosedCurve(myPointArray_2);
     // Рисование и заливка объекта Path
     g.FillPath(mySolidBrush, myGraphicsPath);
     g.DrawPath(p, myGraphicsPath);
}
```

На рисунке 6.4 показан еще один пример использования методов рисования и заливки.

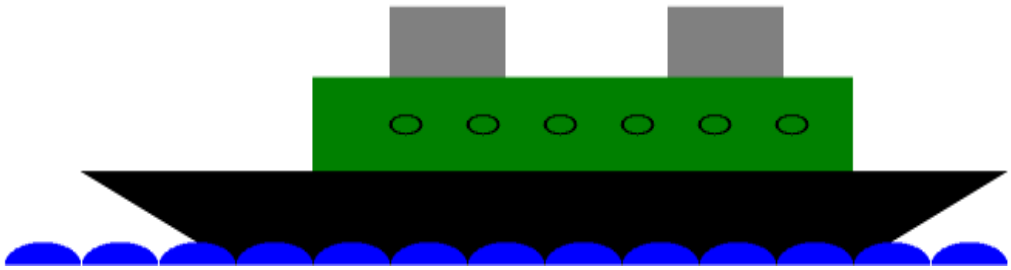

Рис. 6.4 – Изображение корабля, созданное из графических примитивов

Код, генерирующий это изображение:

```
private void Form1_Paint(object sender, PaintEventArgs e) {
     Graphics g = e.Graphics; // Получить объект Graphics
     Pen myOkna = new Pen(Color.Black, 2);
     // Определение кистей
     SolidBrush myCorp = new SolidBrush(Color.Black);
```

```
        SolidBrush myTrum = new SolidBrush(Color.Green);
        SolidBrush myTrub = new SolidBrush(Color.Gray);
        SolidBrush mySe = new SolidBrush(Color.Blue);
        // Рисование и закраска труб, корпуса корабля и трюма
        g.FillRectangle(myTrub, 300, 125, 75, 75);
        g.FillRectangle(myTrub, 480, 125, 75, 75);
        g.FillPolygon(myCorp, new Point[] {
                    new Point(100,300),new Point(700,300),
                    new Point(700,300),new Point(600,400),
                    new Point(600,400),new Point(200,400),
                    new Point(200,400),new Point(100,300)
                    });
        g.FillRectangle(myTrum, 250, 200, 350, 100);
        // Рисование и закраска моря
        int x = 50; int Radius = 50;
        while (x<=pictureBox1.Width-Radius) {
             dc.FillPie(mySe,0+x,375,50,50,0,-180);
             x+=50;
        }
        // Рисование окошек корпуса корабля
        for (int y=300; y<=550; y+=50) {
            dc.DrawEllipse(myOkna,y,240,20,20);
        }
}
```

6.2 Управление изображениями

Управление изображениями в *GDI+* простое и мощное. Класс *Image* предоставляет контейнер для разнообразных форматов изображений, а объект *Graphics* обеспечивает все основные функции, необходимые для отображения картинок. Как и многие из объектов .NET Framework, *Image* – абстрактный базовый класс, производными от которого являются конкретные классы. Основной класс, используемый в *GDI+*, – *Bitmap*.

Характеристики изображения:

1. Размеры в пикселях. Логический размер изображения определяется его размерами в пикселях, выраженными шириной и высотой. Эти значения представляют собой количество отдельных изображений-ячеек по горизонтали и вертикали от начала координат.

2. Глубина пикселя. Это количество бит информации, которое используется для описания одной ячейки изображения. Чем больше количество бит, тем больше оттенков цвета изображение может отображать. Большинство изображений содержит минимум 24 бита на пиксель: по 8 бит на красную, зеленую и синюю составляющую цвета. Это означает, что отдельный пиксель может показать любой из 16 777 216 различных цветов. Есть также индексированные типы изображений, которые используют палитру для обеспечения выбора цвета.

3. Разрешение в пикселях. Это мера того, сколько пикселей изображения выводится на линейный дюйм экрана или области принтера. Физический размер изображения зависит от разрешения в пикселях и размера изображения.

Изображение можно получить с диска методом *Image.FromFile*. Это позволяет указать файл, например, *.JPG* или *.TIFF*, и загрузить его в объект в памяти. Декодер изображения заботится о различных файловых форматах, так что об этом беспокоиться не придется. Имеющееся изображение можно отобразить с помощью метода *Graphics.DrawImage*.

На рисунке 6.5 изображение с диска *C*: отображено на форме.

Рис. 6.5 – Рисование изображения

Код приложения показан в листинге:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace Load_Image {
     public partial class Form1 : Form  {
     Image myImage;
     public Form1() { InitializeComponent(); }
     private void Form1_Load(object sender, EventArgs e) {
          myImage = Image.FromFile(@"flower.jpg");}
     private void Form1_Paint(object sender, PaintEventArgs e){
     if (myImage != null) e.Graphics.DrawImage(myImage, 0, 0);
} } }
```

Приложение загружает изображение в обработчике *OnLoad* и рисует его в обработчике *OnPaint*.

Видно, что команда *DrawImage* рисует изображение, начиная с левого верхнего угла с указанных *x* и *y* координат. Этот способ отображения без излишеств удобен в том случае, если изображение имеет правильный размер и в нем ничего не надо менять.

6.3 Приемник и источник

Команда *DrawImage* часто используется для вывода изображения в указанное место, определяемое в виде целевого прямоугольника. Объявление положения и размера получаемого изображения осуществляется с использованием структуры *Rectangle*. Код, показанный выше в листинге, может быть модифицирован для вставки изображения в конкретный целевой прямоугольник, как показано на рисунке 6.6.

Рис. 6.6 – Несколько искаженное изображение

Изменение кода, который выводит в этом случае изображение, показано в листинге:

```
private void Form1_Paint(object sender, PaintEventArgs e){
     if (myImage != null) e.Graphics.DrawImage(myImage,
                                new RectangleF(20,20,50,130));
}
```

Можно заметить, что определение приемника заставляет пиксели растягиваться или сжиматься, чтобы вместиться в требуемую область.

Приведенный выше пример использует весь образ в качестве источника. Это установка по умолчанию. Если будет указана исходная область, тогда можно взять любую часть исходного изображения и отобразить его в любом месте и с любым коэффициентом искажения. На рисунке 6.7 показано выводимое изображение после того, как область-источник выбрана и скопирована в область-приемник.

Рис. 6.7 – Результат вывода источника в приемник

Код *DrawImage* в *OnPaint* будет иметь такой вид:

```
e.Graphics.DrawImage(myImage,new RectangleF(20,20,50,130),
     new RectangleF(206,46,62,73), GraphicsUnit.Pixel);
```

Лабораторная работа № 6

Построение статических изображений

Цель работы:

1. Изучить методы *Graphics* для рисования графических примитивов.
2. Научиться создавать изображения из графических примитивов.
3. Изучить методы получения с диска изображений и их отображения.
4. Получить навыки отображения изображений с диска.

Контрольные вопросы по теме:

1) Чем отличаются открытые и замкнутые фигуры?
2) Какие существуют методы объекта *Graphics* для рисования или заполнения графических примитивов?
3) Приведите пример рисования замкнутого многоугольника.
4) Как создать, заполнить и нарисовать массив прямоугольников?
5) Приведите пример рисования и заполнения контейнера *Path*.
6) Какой класс предоставляет контейнер для разнообразных форматов изображений?
7) Какими основными характеристиками обладают изображения?
8) Каким методом получают изображение с диска и каким отображают?
9) Приведите пример вывода изображения в конкретный целевой прямоугольник.
10) Как отобразить часть исходного изображения в любом месте и с любым коэффициентом искажения?

Задание

Создайте перечисленные ниже изображения (при рисовании используйте весь набор методов построения графических примитивов).

Варианты индивидуальных заданий:

1. Восход солнца над городом.
2. Корабль, плывущий по волнам моря.
3. Ракета, стартующая с космодрома.
4. Легковой автомобиль, совершающий обгон грузового автомобиля.
5. Летающая тарелка инопланетян, приземляющаяся на поляне.
6. Паровоз с клубами дыма, движущийся по железнодорожным путям.
7. Автобус, движущийся по автомагистрали.
8. Птица, приземляющаяся на опушке леса.
9. Тучи, начинающие закрывать солнце над городом.
10. Всплывающая из глубин океана подводная лодка.
11. Рыбка, плавающая в аквариуме.
12. Самолет, отрывающийся от взлетной полосы.
13. Человек, идущий по улице города.
14. Военный корабль, выпускающий боевую ракету.
15. Легковой автомобиль, въезжающий в гараж.
16. Лист, падающий с дерева.
17. Цветок, уносимый ветром.
18. Лодка, плывущая по реке.
19. Танк, движущийся по минному полю.
20. Биллиардный шар, катящийся по столу.

Порядок выполнения лабораторной работы:

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

7 Создание динамических изображений. Двойная буферизация

7.1 Создание динамических изображений

Алгоритм создания и перемещения графического объекта по статическому изображению состоит из нескольких шагов.

В методе обработки события загрузки формы *Form_Load*:

1. Создать статическое изображение, по которому будет двигаться объект.

Для этого в свойстве *Image* элемента *PictureBox* задайте изображение, отображаемое элементом управления, взяв его из файла на диске.

```
pictureBox1.Image = Image.FromFile(@"fon.jpg");
```

Создайте экземпляр класса *Graphics*, связанный с поверхностью рисования *PictureBox1.Image*:

```
Graphics g_pictureBox = Graphics.FromImage(pictureBox1.Image);
```

Создайте экземпляр класса *Bitmap* от *pictureBox1.Image*:

```
Bitmap pictureBoxBitMap = new Bitmap(pictureBox1.Image);
```

2. Создать изображение движущегося объекта.

Для этого инициализируйте новый экземпляр класса *Bitmap* с размером нашего движущегося графического объекта:

```
Bitmap spriteBitMap = new Bitmap(width, height);
```

Создайте экземпляр класса *Graphics*, связанный с поверхностью рисования *spriteBitMap*:

```
Graphics g_sprite = Graphics.FromImage(spriteBitMap);
```

Используя методы рисования графических примитивов, нарисуйте на нем изображение, которое будет перемещаться.

3. Создать буфер для временного хранения части статического изображения, которое будет затираться перемещающимся объектом.

Для этого инициализируйте новый экземпляр класса *Bitmap* с размером нашего движущегося графического объекта:

```
Bitmap cloneBitMap = new Bitmap(width, height);
```

4. Вывести изображение объекта на статическое изображение.

Для этого задайте начальные координаты левого верхнего угла выводимого объекта (x,y). Сохраните в *cloneBitMap* участок статического изображения с этими координатами, используя собственный метод *SavePart(x,y)*, и выведите движущийся объект в эти координаты:

```
g_pictureBox.DrawImage(spriteBitMap, x, y);
```

Вызовите метод перерисовки элемента управления *pictureBox1*:

```
pictureBox1.Invalidate();
```

5. *Создать экземпляр класса Timer*:

```
Timer timer1 = new Timer();
```

Укажите интервал и обработчик события *Tick* от таймера:

```
timer1.Interval = 100;
timer1.Tick += new EventHandler(timer1_Tick);
```

В обработчике события *Tick*-таймера:

1. Восстановите затертую на предыдущем шаге область *PictureBox1.Image*:

g_pictureBox.DrawImage(cloneBitMap, x, y);

2. Измените координаты для вывода движущегося объекта. Например, при движении объекта по горизонтали нарастите координату *x*.

x = x + 5;

Проведите контроль на выход координат объекта за границы *pictureBox1*. Например, контроль по x-координате на выход за правую границу:

if (x + width > pictureBox1.Width) {x = pictureBox1.Location.X;}

3. Сохраните область экрана с новыми координатами перед выводом движущегося объекта, используя собственный метод *SavePart(x,y)*, в *cloneBitMap*.

4. Выведите изображение в новых координатах:

g_pictureBox.DrawImage(spriteBitMap, x, y);

5. Вызовите метод перерисовки элемента управления *pictureBox1*:

pictureBox1.Invalidate();

Разместите на форме кнопку для старта анимации движения и в обработчике события нажатия на кнопку *Button_Click* запустите таймер:

timer1.Enabled = true;

Код приложения, реализующий данный алгоритм, показан в листинге:

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Lesson_Animation
{
  public partial class Form1 : Form
  {
   // битовая картинка pictureBox
```

```
Bitmap pictureBoxBitMap;
// битовая картинка динамического изображения
Bitmap spriteBitMap;
// битовый образ для временного хранения области экрана
Bitmap cloneBitMap;
// объект Graphics, на котором будет двигаться объект
Graphics g_pictureBox;
// объект Graphics для динамического изображения
Graphics g_sprite;
int diametr = 40; // диаметр рисуемого круга
int x, y;       // координаты динамического изображения
Timer timer1;  // таймер

public Form1()   { InitializeComponent(); }
// Метод для рисования динамического изображения
void DrawSprite() {
 g_sprite.FillEllipse(Brushes.Lime,0,0,diametr-1,diametr-1);
 g_sprite.DrawEllipse(Pens.Purple,0,0,diametr-1, diametr-1);
}

 // Метод для сохранения части изображения экрана
void SavePart(int xt, int yt) {
 Rectangle cloneRect = new Rectangle(xt,yt,diametr,diametr);
 System.Drawing.Imaging.PixelFormat format =
                              pictureBoxBitMap.PixelFormat;
 cloneBitMap = pictureBoxBitMap.Clone(cloneRect, format);
}

// Обработчик события загрузки формы
private void Form1_Load(object sender, EventArgs e){
 // Создаем Bitmap для pictureBox1 и графическую поверхность
 pictureBox1.Image = Image.FromFile(@"fon.jpg");
 pictureBoxBitMap = new Bitmap(pictureBox1.Image);
 g_pictureBox = Graphics.FromImage(pictureBox1.Image);
 // Создаем Bitmap для объекта и графическую поверхность
```

```
 spriteBitMap = new Bitmap(diametr, diametr);
 g_sprite = Graphics.FromImage(spriteBitMap);
 // Создаем изображение движущегося объекта
 DrawSprite();
 // Создаем Bitmap для временного хранения части изображения
 cloneBitMap = new Bitmap(diametr, diametr);
 // Начальные координаты вывода движущегося объекта
 x = 0; y = pictureBox1.Height / 2;
 // Сохраняем область экрана перед первым выводом объекта
 SavePart(x, y);
 // Выводим объект
 g_pictureBox.DrawImage(spriteBitMap, x, y);
 // Перерисовываем pictureBox1
 pictureBox1.Invalidate();
 // Создаем таймер
 timer1 = new Timer();
 timer1.Interval = 100;
 timer1.Tick += new EventHandler(timer1_Tick);
}

// Включаем таймер по нажатию на кнопку
private void button1_Click(object sender, EventArgs e)
{ timer1.Enabled = true; }

// Движение изображения по событию Tick-таймера
private void timer1_Tick(object sender, EventArgs e)
{
 // Восстанавливаем затертую область статического изображения
 g_pictureBox.DrawImage(cloneBitMap, x, y);
 // Изменяем координаты для следующего вывода объекта
 x = x + 5;
 // Проверяем на выход изображения объекта за правую границу
 if(x > pictureBox1.Width-diametr)
 {x = pictureBox1.Location.X;}
 // Сохраняем часть статического изображения перед выводом
```

```
            SavePart(x, y);
            // Выводим движущийся объект
            g_pictureBox.DrawImage(spriteBitMap, x, y);
            // Перерисовываем pictureBox1
            pictureBox1.Invalidate();
        }
    }
}
```

На рисунке 7.1 показана работа приложения. Шарик цвета lime с контуром цвета purple движется слева направо по статическому изображению.

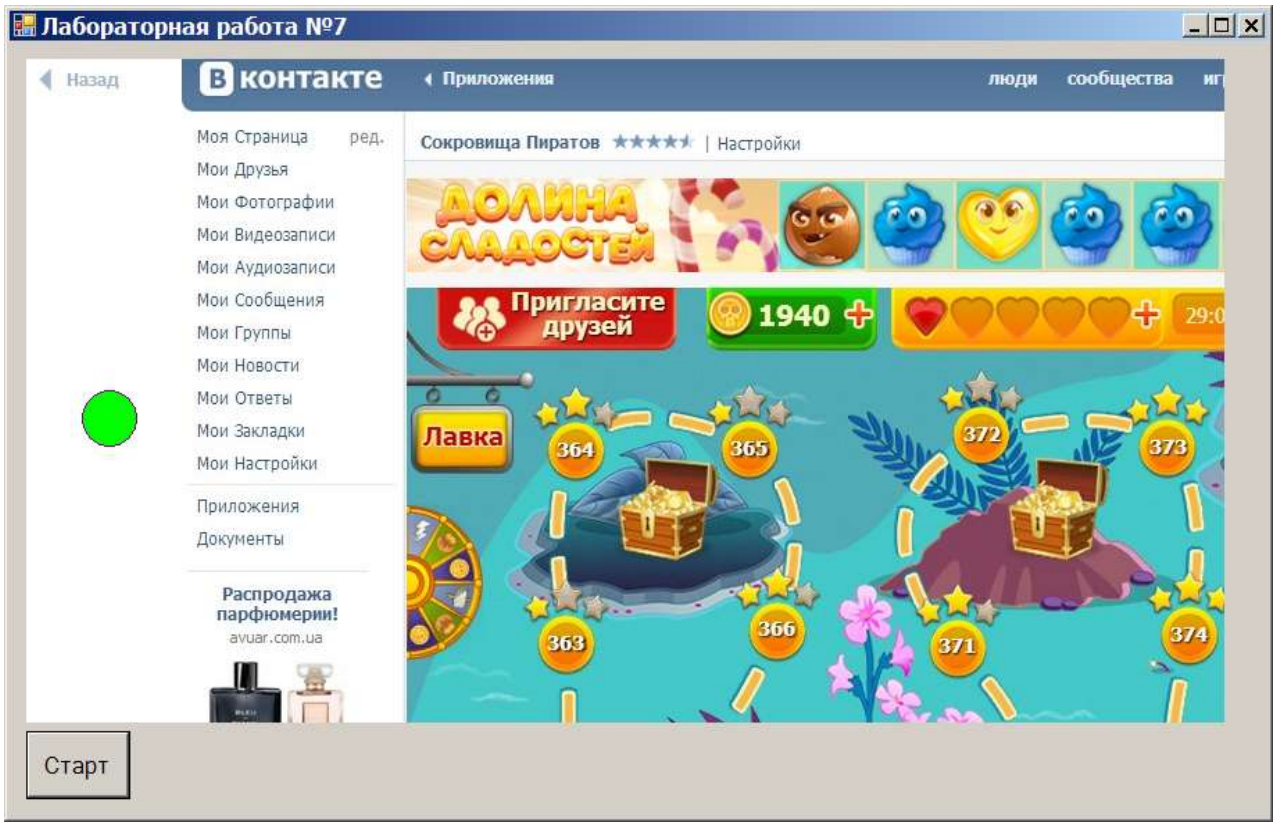

Рис. 7.1 – Шарик, движущийся по статическому изображению

7.2 Двойная буферизация графики

Графические действия, требующие нескольких сложных операций рисования, могут привести к тому, что визуализируемые изображения

будут мерцать или иметь неприемлемый внешний вид. Чтобы это устранить, *.NET Framework* предоставляет доступ к двойной буферизации.

При двойной буферизации для решения проблем, связанных с многократным повторением операций рисования, используется буфер в памяти. Если двойная буферизация включена, все операции рисования сначала выполняются в памяти, а лишь затем – на экране компьютера. После завершения всех операций рисования содержимое буфера копируется из памяти непосредственно на связанную с ним область экрана. Поскольку на экране выполняется лишь одна графическая операция, то мерцание, которое часто возникает в сложных операциях рисования, исчезает.

7.2.1 Двойная буферизация по умолчанию

Для большинства приложений наилучших результатов можно достичь при использовании двойной буферизации *.NET Framework* по умолчанию. Двойная буферизация включена для стандартных элементов управления *Windows Forms* по умолчанию.

Включить двойную буферизацию в собственных формах и элементах управления можно двумя способами. Для этого нужно либо присвоить свойству *DoubleBuffered* значение *true – DoubleBuffered = true;*, либо вызвать метод *SetStyle*, чтобы сделать флаг *OptimizedDoubleBuffer* равным *true – SetStyle(ControlStyles.OptimizedDoubleBuffer, true);*.

Вызывать метод *SetStyle* рекомендуется лишь в тех случаях, если пользователем был написан весь код отрисовки элемента управления.

7.2.2 Управление буферизацией графики вручную

В сложных случаях буферизации, например, для отображения анимации или сложном управлении памятью, можно воспользоваться для

реализации собственной логики двойной буферизации классами *.NET Framework*.

За выделение отдельных буферов графики и управление ими отвечает класс *BufferedGraphicsContext*. У каждого приложения есть собственный экземпляр *BufferedGraphicsContext* по умолчанию, который управляет всей двойной буферизацией для данного приложения.

Экземпляры *BufferedGraphicsContext* по умолчанию управляются классом *BufferedGraphicsManager*. Ссылку на экземпляр *BufferedGraphicsContext* по умолчанию можно получить, обратившись к свойству *BufferedGraphicsManager.Current*.

Получение ссылки на объект *BufferedGraphicsContext* по умолчанию:

```
BufferedGraphicsContext myContext;
myContext = BufferedGraphicsManager.Current;
```

Не рекомендуется вызывать метод *Dispose* по ссылке *BufferedGraphicsContext*, полученной у класса *BufferedGraphicsManager*, так как объект *BufferedGraphicsManager* обрабатывает все выделение и распределение памяти для экземпляров *BufferedGraphicsContext* по умолчанию.

Можно повысить производительность приложений, активно использующих графические возможности, создав *выделенный* экземпляр *BufferedGraphicsContext* вместо объекта *BufferedGraphicsContext*, предоставляемого классом *BufferedGraphicsManager*.

Это дает возможность создавать графические буфера и управлять ими по отдельности, но объем потребляемой приложением памяти в этом случае возрастет.

Создание выделенного объекта *BufferedGraphicsContext*:

```
BufferedGraphicsContext myContext;
myContext = new BufferedGraphicsContext();
// Здесь расположен код создания графики.
// Для экземпляра объекта BufferedGraphicsContext
```

```
// Не по умолчанию при завершении вызвать метод Dispose
myContext.Dispose();
```

7.2.3 Вывод буферизованной графики вручную

Чтобы использовать экземпляр класса *BufferedGraphicsContext* для создания графических буферов, можно вызвать метод *BufferedGraphicsContext.Allocate*, который возвращает экземпляр класса *BufferedGraphics*. Объект *BufferedGraphics* служит для управления буфером памяти, связанным с областью отрисовки, например, с формой или элементом управления.

После создания объекта класса *BufferedGraphics* этот объект будет управлять отрисовкой изображения непосредственно в буфере памяти. Для передачи туда изображения, можно использовать свойство *BufferedGraphics.Graphics*, которое служит для доступа к объекту *Graphics*, представляющему собой содержимое этого буфера памяти. Передача изображений этому объекту ничем не отличается от передачи изображений объекту *Graphics*, который представляет собой область экрана.

После записи всей информации об изображении в буфер можно воспользоваться методом *BufferedGraphics.Render*, чтобы скопировать содержимое буфера в нужную область экрана.

Для вывода буферизированной графики вручную:

1. Получите ссылку на экземпляр класса *BufferedGraphicsContext*:

```
// Пример предполагает наличие формы под названием Form1
BufferedGraphicsContext currentContext;
BufferedGraphics myBuffer;
// Получаем ссылку на текущий BufferedGraphicsContext
currentContext = BufferedGraphicsManager.Current;
```

2. Создайте экземпляр класса *BufferedGraphics*, вызвав метод *Allocate*:

```
// Создаем экземпляр BufferedGraphics, связанный с Form1,
```

```
// и с такими же размерами, как размер поверхности Form1.
myBuffer = currentContext.Allocate(this.CreateGraphics(),
                                   this.DisplayRectangle);
```

3. С помощью свойства *Graphics* поместите в буфер графические объекты:

```
// Рисуем эллипс в графическом буфере
myBuffer.Graphics.DrawEllipse(Pens.Blue,this.DisplayRectangle);
```

4. Завершив операции рисования в графическом буфере, вызовите метод *Render*, чтобы отрисовать содержимое буфера на указанной или связанной с ним поверхности рисования:

```
// Отрисовываем содержимое буфера на связанную с ним
// поверхность рисования
myBuffer.Render();
// Отрисовываем содержимое буфера на указанную поверхность
// рисования
myBuffer.Render(this.CreateGraphics());
```

5. Выполнив отрисовку графики, вызовите метод *Dispose* экземпляра *BufferedGraphics*, чтобы освободить ресурсы системы:

```
myBuffer.Dispose();
```

На рисунке 7.2 показан результат вывода буферизированной графики вручную.

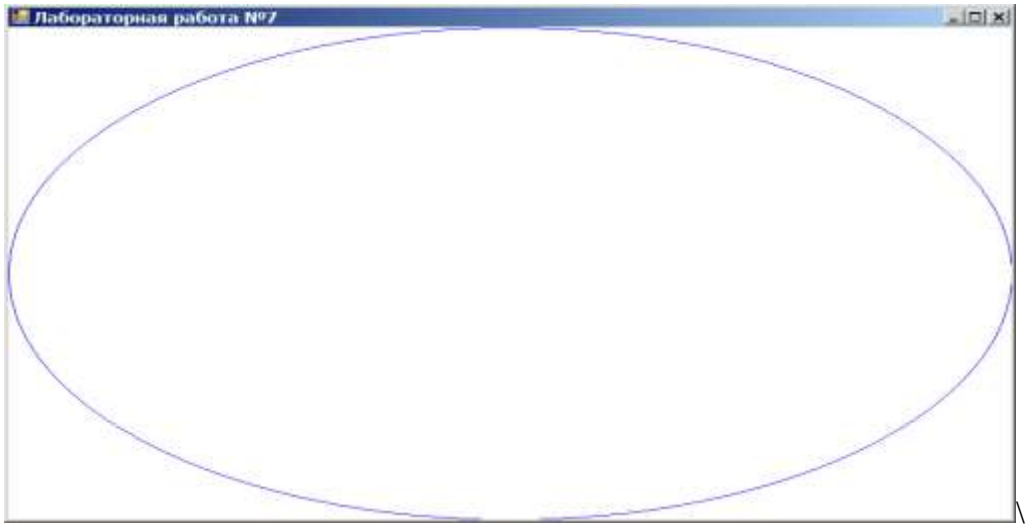

Рис. 7.2 – Эллипс, нарисованный с использованием двойной буферизации

Лабораторная работа № 7

Построение динамических изображений

Цель работы:

1. Изучить методы объекта *Graphics* для создания динамических изображений и алгоритмы создания.

2. Изучить принципы двойной буферизации графики.

3. Получить навыки создания динамических изображений.

Контрольные вопросы по теме:

1) Как создать статическое изображение, по которому будет перемещаться движущийся объект?

2) Как создать изображение объекта, который будет перемещаться?

3) Для чего создается временный буфер?

4) Как скопировать часть статического изображения во временный буфер?

5) Какие действия необходимо произвести в обработчике события *Tick*-таймера, чтобы объект перемещался по статическому изображению?

6) Как в общих чертах можно описать алгоритм получения динамического изображения?

7) Для чего используется двойная буферизация графики?

8) Как включить двойную буферизацию в собственных формах и элементах управления?

9) Когда необходимо управление буферизацией графики вручную?

10) За что отвечает класс *BufferedGraphicsContext*?

11) Приведите пример создания выделенного экземпляра *BufferedGraphicsContext*.

12) Какие шаги необходимо выполнить для вывода буферизированной графики вручную?

Задание:

Создайте перечисленные ниже динамические изображения.

Варианты индивидуальных заданий:

1. Всплывающая из глубин океана подводная лодка.
2. Летающая тарелка инопланетян, приземляющаяся на поляне.
3. Цветок, уносимый ветром.
4. Военный корабль, выпускающий боевую ракету.
5. Ракета, стартующая с космодрома.
6. Самолет, отрывающийся от взлетной полосы.
7. Паровоз с клубами дыма, движущийся по железнодорожным путям.
8. Танк, движущийся по минному полю.
9. Корабль, плывущий по волнам моря.
10. Автобус, движущийся по автомагистрали.
11. Легковой автомобиль, въезжающий в гараж.
12. Восход солнца над городом.
13. Лодка, плывущая по реке.
14. Человек, идущий по улице города.
15. Легковой автомобиль, совершающий обгон грузового.
16. Биллиардный шар, катящийся по столу.
17. Лист, падающий с дерева.
18. Рыбка, плавающая в аквариуме.
19. Туча, начинающие закрывать солнце над городом.
20. Птица, приземляющаяся на опушке леса.

Порядок выполнения лабораторной работы:

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

8 Спрайты. Мультипликация

8.1 Спрайты

Многие видеоигры, в которых игрок управляет объектами, атакующими другие объекты, управляемые программой, или защищающимися от них, включают два класса активных объектов:

1) среда, представляющая собой мало меняющееся поле игры;

2) спрайты.

Спрайт – это небольшой подвижный объект, который движется по полю видеоигры по определенным правилам с заданной целью.

Когда космический корабль стреляет фотонными торпедами, то изображение торпеды реализуется спрайтом.

Под спрайтом можно понимать фигуру, определенную некоторыми замкнутыми отрезками, т.е. многоугольник.

8.2 Поле игры

В большинстве видеоигр поле игры представляет собой неменяющееся или медленно меняющееся изображение, на котором происходит основное действие. Поле игры изображается отдельными программами, загружаемыми в начале, поэтому нет необходимости загружать все программы верхнего уровня для динамической генерации игрового поля.

8.3 Мультипликация на экране

Ключевым и наиболее впечатляющим моментом видеоигр является мультипликация. Мультипликация – их основной отличительный признак.

Основной метод мультипликации прост: уничтожить изображение объекта и создать его вновь, но с некоторым небольшим смещением. Скорость этого процесса должна быть очень высокой. Это может быть обеспечено путем непосредственного доступа к видеопамяти. Для повышения качества отображения спрайта, быстродействия операций уничтожения и повторного изображения объекта используется графический буфер, в котором изначально рисуется изображение спрайта. В дополнение к нему используется вспомогательный графический буфер, в котором сохраняется участок экрана, затираемый спрайтом на очередном шаге перемещения по экрану. Этот способ обеспечивает возможность быстрого перемещения спрайта по экрану, не меняя его цвет и размеры, и, не уничтожая в памяти данные о его изображении.

При разработке видеоигр необходимо так подбирать размеры спрайта, чтобы возможности компьютера и видеокарты реализовывались оптимальным образом.

8.4 Мультипликация спрайта

Передвижение спрайта по экрану составляет только половину возможностей его «оживления».

В основном спрайт используется на экране для того, чтобы создавать иллюзию движения. Например, спрайт, который выглядит подобно человеку, может передвигать ногами, как будто он идет. Этот тип «оживления» является наиболее впечатляющим и наиболее легким. Для обеспечения такой возможности разрабатываются два или более варианта спрайта, разница между которыми заключается в том, что некоторые из его частей отличаются от первоначального варианта. Программа последовательно меняет варианты спрайта в процессе его движения по экрану.

8.5 Организация данных в видеоиграх

Подобно остальным программам, программы видеоигр включают как операторы, так и данные.

К данным относятся счет игры, статус различных расходуемых в процессе игры ресурсов (например, количество запущенных фотонных торпед) и т. п. Большинство данных, используемых в видеоиграх, представляют собой позиции экрана для различных объектов.

Координаты позиций экрана для движущихся объектов должны храниться в установленных переменных.

Информацию о фиксированных объектах игрового поля целесообразно хранить непосредственно в видеопамяти. Потому что, если в процессе игры потребуется информация для изменения поля, то осуществляется доступ к видеопамяти и оттуда считываются массивы с информацией об измененном объекте.

8.6 Контроль границ

В большинстве видеоигр существуют спрайты, которые находятся под управлением пользователя. Обычно игроку не разрешается перемещать спрайт через некоторые объекты игрового поля или через другой спрайт.

Есть два способа *ограничения местонахождения спрайта*.

В первом способе в установленных переменных хранятся граничные точки области, где разрешено движение спрайта. При перемещении спрайта по экрану осуществляется контроль выхода за пределы этих допустимых значений. Однако этот метод обладает довольно малой реактивностью и для игр с большим количеством объектов неэффективен.

Более удобным способом является простая проверка области экрана на предмет нахождения в ней какого-либо объекта путем контроля

соответствующей области видеопамяти. Это обеспечивается тем, что информация об игровом поле уже находится в видеопамяти и бессмысленно где-либо ее дублировать.

8.7 Изменение цвета

В процессе игры удобно оперировать объектами разного цвета. Например, красный может отображать границы не пересекаемых областей, зеленый используется для вашего спрайта, а желтый – для спрайта противника. Хотя это можно сделать с использованием переменных, описывающих эти объекты, часто бывает удобнее заранее определять для объекта его цвет. Это не только упростит процесс программирования видеоигры, но и сделает ее более быстродействующей.

Например, если в пурпурный цвет окрашена мина, то считается, что на ней подорвались лишь в том случае, если одна из точек спрайта окрашивается в пурпурный цвет. В игре «пинг-понг» белый цвет обычно несовместим с белым (они отталкиваются), но можно двигаться по черному игровому полю. Таким образом, белый шарик может перемещаться по черному полю, если ударяется белой ракеткой или отражается от белой стены (линии) позади ракетки.

8.8 Табло счета активного противника

Роль компьютера в игре во многом зависит от того, является ли эта игра для одного человека или для двоих. Если в игре участвуют два человека, компьютеру отводится роль арбитра и функция табло для отображения счета. Однако в игре, где участвует один игрок, компьютер становится активным противником. С точки зрения программирования разработка игр, где компьютер выступает в роли противника, значительно интересней.

8.9 Разработка видеоигры «Воздушный бой»

Для создания видеоигры понадобятся такие элементы как спрайты и поле игры. Спрайтами игры будут:

1. Самолет, летящий по полю игры слева направо

Рис. 8.1 – Самолет, летящий слева направо

2. Самолет, летящий по полю игры справа налево

Рис. 8.2 – Самолет, летящий справа налево

3. Зенитная пушка

Рис. 8.3 – Зенитная пушка

4. Снаряд

Рис. 8.4 – Снаряд

Изображения спрайтов скачиваются из сети Internet или создаются с использованием графических редакторов *Paint.NET* или *Adobe Photoshop*.

Одним из важных свойств изображений спрайтов является наличие у картинки прозрачного фона. Поэтому изображения необходимо

конвертировать или сохранить в формате png, который предоставляет такую возможность.

На рисунке 8.5 показано изображение самолета на светло-сером фоне.

Рис. 8.5 – Самолет на светло-сером фоне

Для того чтобы сделать фон прозрачным, откройте изображение в редакторе *Paint.NET*. Выберите инструмент «Волшебная палочка» и щелкните ею по фону. Он будет выделен. Для удаления фона нажмите клавишу Del.

Если расширение файла изображения не .png, то выберите пункт меню «Файл» => «Сохранить как» и сохраните картинку в формате png.

Чтобы подготовить поле игры, скачиваем из сети Internet фотографию неба и сохраняем на своем жестком диске. Пример фотографии неба показан на рисунке 8.6.

Рис. 8.6 – Фотография неба

Вверху главной формы приложения размещаем элементы управления *label* для отображения количества сбитых и упущенных самолетов, ниже размещаем элемент управления *pictureBox* (графическое окно для отображения поля игры) и еще ниже – два элемента управления *button* для запуска и приостановки игры и сброса счетчиков. В свойство *Image* элемента *pictureBox* загружаем файл изображения поля игры.

Добавляем на форму два компонента *timer1* и *timer2*. Устанавливаем для них свойство *Enabled* в значение *True*. Свойство *Interval* компонента *timer1* устанавливаем в 40 (с интервалом времени 0,04 секунды будет предприниматься попытка создания нового самолета). Свойство *Interval* компонента *timer2* устанавливаем в 10 (с интервалом 0,01 секунды будет происходить перемещение самолетов и снарядов по экрану).

На рисунке 8.7 показана работа приложения.

Рис. 8.7 – Игра «Воздушный бой»

В начале программы задаются константы и переменные, создаются два вспомогательных метода для удаления самолета и снаряда.

По событию загрузки формы устанавливаются флаги программы и загружается на поле игры зенитная пушка с прозрачным фоном (свойство *BackColor* элемента *pictureBox* пушки устанавливается в *Color.Transparent*). Таким образом, при перемещении пушки ее фоном будет фон поля игры.

При тиках компонента *timer1* генерируется случайное число от 0 до 900. Если оно больше или равно нулю и меньше заданного порогового числа, регулирующего частоту появления самолетов на экране, и количество самолетов на экране меньше числа, определяющего

максимальное количество самолетов на экране, то создается новый самолет с прозрачным фоном (свойство *BackColor* элемента *pictureBox* самолета устанавливается в *Color.Transparent*). Таким образом, при перемещении самолета его фоном будет фон поля игры. Самолет добавляется на поле игры.

При тиках компонента *timer2* проверяется, установлен ли флаг игры, уменьшается на единицу счетчик интервалов времени до готовности пушки к стрельбе. Этот счетчик необходим, чтобы снаряды не могли лететь непрерывно. Затем проверяется, установлен ли флаг выстрела и готовность пушки. Если эти два условия выполнены, то на поле игры добавляется новый снаряд. Далее в цикле осуществляется перемещение самолетов и их контроль на выход за границы поля. Если самолет вышел за границы игрового поля, то он удаляется и увеличивается счетчик пропущенных самолетов.

В следующем цикле происходит перемещение снарядов и контроль на выход за границы поля игры и попадания в самолет. Если снаряд вышел за границы поля игры, он уничтожается. Если прямоугольные области самолета и снаряда пересекаются, то они уничтожаются и увеличивается счетчик сбитых самолетов. Измененные счетчики сбитых и пропущенных самолетов выводятся на экран.

По нажатию на кнопку начала и паузы игры в зависимости от надписи на ней, устанавливается или сбрасывается флаг игры. Если флаг игры установлен, то фокус устанавливается на поле игры, чтобы можно было перемещать зенитную пушку. Если сброшен, то фокус устанавливается на саму кнопку.

По нажатию на вторую кнопку счетчики игры сбрасываются в ноль и выводятся на экран.

В методе обработки события «нажатие на клавишу» осуществляется перемещение пушки влево или вправо при нажатии на клавиши «Стрелка

влево» и «Стрелка вправо», установка флага выстрела при нажатии на клавишу «Пробел» и запуск или пауза игры при нажатии на Enter.

В методе обработки события «отпускание клавиши» осуществляется сброс флага выстрела при отпускании клавиши «Пробел».

Код видеоигры «Воздушный бой» с подобными комментариями приведен в листинге:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Graph_game
{
  public partial class Form1 : Form {
  const int AirplaneMax = 5;   // Максимум самолетов на экране
  const int BulletsMax = 5;    // Максимум снарядов на экране
  PictureBox[] Airplanes = new PictureBox[AirplaneMax]; // Самолеты
  PictureBox[] Bullets = new PictureBox[BulletsMax]; // Снаряды
  PictureBox Gun = new PictureBox();     // Картинка пушки
  bool StartFlag;              // Флаг игры
  bool FireFlag;               // Флаг выстрела
  int Cooldown = 30;      // Скорость зарядки пушки (3/10 секунды)
  int Shooted = 0;        // Количество тиков до готовности пушки
  int AirplaneShootedCount = 0;     // Счетчик сбитых самолетов
  int AirplaneMissedCount = 0; // Счетчик упущенных самолетов
  int GunSpeed = 5;            // Скорость пушки
  int AirplaneSpeed = 3;       // Скорость самолета
  int BulletSpeed = 7;         // Скорость снаряда
  int AirplaneCount = 0;       // Количество самолетов на экране
  int BulletCount = 0;         // Количество снарядов на экране
  int AirplaneFrequency = 20;  // Частота появления самолетов
```

```
  public Form1() { InitializeComponent(); }
  // *** Удаление самолета ***
  private void DeleteAirplane(int i) {
   Airplanes[i].Dispose(); // освобождение ресурсов Airplanes[i]
   for (int j = i; j<AirplaneCount-1; j++)
   { Airplanes[j] = Airplanes[j + 1]; }
   AirplaneCount--;
  }
  // *** Удаление снаряда ***
  private void DeleteBullet(int i) {
   Bullets[i].Dispose(); // освобождение ресурсов Bullets[i]
   for (int j = i; j<BulletCount-1; j++)
   {Bullets[j] = Bullets[j + 1];}
   BulletCount--;
  }
  // *** Обработка события загрузки формы ***
  private void Form1_Load(object sender, EventArgs e) {
   // Устнавливаем все флажки при загрузке формы в false
   StartFlag = false; FireFlag = false;
   // Загружаем в pictureBox Gun изображение пушки
   Gun.Image = Image.FromFile(@"Пушка.png");
   Gun.BackColor = Color.Transparent;    // Прозрачный фон пушки
   Gun.Location = new Point(pictureBoxPole.Image.Width / 2,
   pictureBoxPole.Image.Height - Gun.Image.Height);
   Gun.Size = new Size(Gun.Image.Width, Gun.Image.Height);
   Gun.Name = "Game_Gun";
   pictureBoxPole.Controls.Add(Gun); // Добавляем пушку на поле игры
   Gun.BringToFront();
  }
// *** Таймер для создания самолетов ***
  private void timer1_Tick_1(object sender, EventArgs e){
  // Если флаг игры установлен
  if (StartFlag) {
   // Получаем случайное целое число меньше 900
   Random a = new Random();
   int RandomEvent = a.Next(900);
```

```
// Новый самолет
if ((RandomEvent >= 0) & (RandomEvent < AirplaneFrequency) &
(AirplaneCount < AirplaneMax)) {
// Увеличиваем счетчик количества самолетов на экране
AirplaneCount++;
// Переименовываем самолеты со старшими номерами
for (int i = AirplaneCount - 1; i > 0; i--) {
Airplanes[i] = Airplanes[i - 1];
}
// Создаем изображение нового самолета
Airplanes[0] = new PictureBox();
// который будет лететь слева направо
if ((RandomEvent > -1) & (RandomEvent <= AirplaneFrequency/2)){
Airplanes[0].Image = Image.FromFile(@"Самолет_ЛП.png");
Airplanes[0].Image.Tag = "ЛП";
Airplanes[0].Location = new Point(0,pictureBoxPole.Location.Y +
a.Next(100));
}
else {// который будет лететь справа налево
Airplanes[0].Image = Image.FromFile(@"Самолет_ПЛ.png");
Airplanes[0].Image.Tag = "ПЛ";
Airplanes[0].Location = new Point(pictureBoxPole.Width -
Airplanes[0].Image.Width,pictureBoxPole.Location.Y + a.Next(100));
}
// Устанавливаем прозрачный цвет фона самолета
Airplanes[0].BackColor = Color.Transparent;
Airplanes[0].Size = new Size(Airplanes[0].Image.Width,
Airplanes[0].Image.Height);
Airplanes[0].Name = "Airplane" + AirplaneCount.ToString();
// Добавляем самолет на поле игры
pictureBoxPole.Controls.Add(Airplanes[0]);
Airplanes[0].BringToFront();
}
}
}
// *** Таймер для движения объектов и стрельбы ***
private void timer2_Tick(object sender, EventArgs e) {
```

```
// Если игра запущена
if (StartFlag){
 // Уменьшаем тики времени до перезарядки пушки
 if (Shooted > 0) { Shooted--; }
 // Если установлен флаг выстрела
 if (FireFlag){
 // и если пушка готова к выстрелу
 if (Shooted == 0){
 // Увеличиваем количество выпущенных снарядов
 BulletCount++;
 // переименовываем снаряды со старшими номерами
 for (int i = BulletCount-1; i > 0; i--){
 Bullets[i] = Bullets[i - 1];
 }
 // Создаем изображение нового снаряда
 Bullets[0] = new PictureBox();
 Bullets[0].Image = Image.FromFile(@"Снаряд.png");
 Bullets[0].Location = new Point(Gun.Location.X +
 Gun.Image.Width /2, Gun.Location.Y);
 Bullets[0].Size = new Size(Bullets[0].Image.Width,
 Bullets[0].Image.Height);
 Bullets[0].Name = "Bullets" + BulletCount.ToString();
 // Добавляем снаряд на поле игры
 pictureBoxPole.Controls.Add(Bullets[0]);
 Bullets[0].BringToFront();
 // Устанавливаем количество тиков до нового выстрела
 Shooted = Cooldown;
 }
 }
 // Перемещение самолетов и контроль на выход за границы
 for (int i = 0; i < AirplaneCount; i++) {
 // Если самолет летит слева направо
 if (Airplanes[i].Image.Tag.ToString() == "ЛП")
 { // Смещаем самолет вправо на AirplaneSpeed точек
 Airplanes[i].Left = Airplanes[i].Left + AirplaneSpeed;
 // Если самолет долетел до правой границы
 if (Airplanes[i].Left > pictureBoxPole.Image.Width) {
```

```
DeleteAirplane(i);// Удаляем самолет
AirplaneMissedCount++;// Увеличиваем счетчик пропущенных
}
}
Else {
// Если самолет летит справа налево
// Смещаем самолет влево на AirplaneSpeed точек
Airplanes[i].Left = Airplanes[i].Left - AirplaneSpeed;
// Если самолет долетел до левой границы
if (Airplanes[i].Left < 0 - Airplanes[i].Width) {
DeleteAirplane(i);      // Удаляем самолет
AirplaneMissedCount++;// Увеличиваем счетчик пропущенных
}
}
}
//Перемещение снарядов, контроль на выход за границы и попадание
for (int i = 0; i < BulletCount; i++)
{ // Перемещаем снаряд вверх на BulletSpeed точек
Bullets[i].Location = new Point(Bullets[i].Location.X,
                        Bullets[i].Location.Y - BulletSpeed);
// Если снаряд долетел до верхней границы
if (Bullets[i].Location.Y < pictureBoxPole.Location.Y –
                                                this.Location.Y)
{ DeleteBullet(i);} // Удаляем снаряд
// Получаем координаты прямоугольной области снаряда
Rectangle r1 = Bullets[i].DisplayRectangle;
// Проверка на попадание в самолет
for (int j = 0; j < AirplaneCount; j++) {
// Получаем координаты прямоугольной области j-го самолета
Rectangle r2 = Airplanes[j].DisplayRectangle;
r1.Location = Bullets[i].Location;
r2.Location = Airplanes[j].Location;
// Если прямоугольные области пересекаются
if (r1.IntersectsWith(r2)) {
DeleteBullet(i);        // Удаляем снаряд
DeleteAirplane(j);      // Удаляем самолет
AirplaneShootedCount++; // Увеличиваем количество сбитых
```

```
 }
 }
 }
 // Изменяем значения счетчиков игры на экране
 label_AirplaneMissed.Text = AirplaneMissedCount.ToString();
 label_AirplaneShooted.Text = AirplaneShootedCount.ToString();
 }
}
// ***Запуск и остановка игры по нажатию на кнопку ***
private void button1_Click(object sender, EventArgs e) {
 if (button1.Text == "Старт") {
 button1.Text = "Пауза"; StartFlag = true;
 pictureBoxPole.Focus(); // Устанавливаем фокус на поле игры
 return;
 }
 if (button1.Text == "Пауза") {
 button1.Text = "Старт"; StartFlag = false;
 button1.Focus(); // Устанавливаем фокус на клавишу "Старт"
 }
}
// ***Очистка счетчиков игры по нажатию на кнопку ***
private void buttonClear_Click(object sender, EventArgs e) {
 // Обнуляем количество сбитых и пропущенных самолетов
 AirplaneShootedCount = 0;
 AirplaneMissedCount = 0;
 // Выводим на экран
 label_AirplaneMissed.Text = AirplaneMissedCount.ToString();
 label_AirplaneShooted.Text = AirplaneShootedCount.ToString();
}
// *** События клавиатуры будут проходить через обработчики
// pictureBoxPole ***
 private void pictureBoxPole_PreviewKeyDown(object sender,
 PreviewKeyDownEventArgs e) {
 // чтобы обрабатывались клавиши со стрелками "влево-вправо"
 e.IsInputKey = true;
}
```

```
  // *** Обработка нажатия клавиш ***
  private void Form1_KeyDown(object sender, KeyEventArgs e) {
   // Перемещение пушки вправо на GunSpeed точек
   if (e.KeyCode == Keys.Right) {
   if (Gun.Left + Gun.Width + GunSpeed < pictureBoxPole.Image.Width)
   { Gun.Left = Gun.Left + GunSpeed; }
   }
   // Перемещение пушки влево на GunSpeed-точек
   if (e.KeyCode == Keys.Left) {
   if (Gun.Left >= GunSpeed) {Gun.Left = Gun.Left - GunSpeed; }
   }
   // Установка флага FireFlag по нажатию клавиши "пробел"
   if (e.KeyCode == Keys.Space) { FireFlag = true; }
   // Остановка и возобновление игры по нажатию клавиши "Enter"
   if (e.KeyCode == Keys.Enter) { button1_Click(sender,e); }
  }
  // *** Обработка отпускания клавиш ***
  private void Form1_KeyUp(object sender, KeyEventArgs e)
   { // Сброс флага FireFlag по отпусканию клавиши "пробел"
   if (e.KeyCode == Keys.Space) { FireFlag = false; }}
  }
}
```

Лабораторная работа № 8

Разработка видеоигр

Цель работы:

1. Изучить основы техники программирования видеоигр.
2. Получить навыки создания видеоигр.

Контрольные вопросы по теме:

1) Что такое «спрайт»?
2) Какой основной метод мультипликации?
3) В чем заключается «оживление» спрайта?
4) Какие данные хранят программы видеоигр?
5) Как осуществляется ограничение местонахождения спрайта?
6) Почему в видеоиграх удобно оперировать объектами разного цвета?

Задание:

Создайте видеоигру. Для этого разработайте структуры данных, изображения спрайтов, нарисуйте поле игры, реализуйте табло счета.

Варианты индивидуальных заданий:

1. **Морской бой.** Выпущенная торпеда должна поражать появляющиеся в дальней части экрана корабли противника, имеющие разный вид, и, соответственно, за каждый уничтоженный корабль начисляется различное число очков. Торпедный аппарат управляется пользователем. Алгоритм движения кораблей противника определяется программой. Табло счета – общее количество объектов противника, количество не уничтоженных и общее количество очков за уничтоженные корабли.

2. **Воздушный бой.** Зенитный комплекс сбивает пролетающие самолеты и воздушные шары противника, за которые начисляется различное количество очков. Управляется он пользователем. Алгоритм движения самолетов и воздушных шаров определяется программой. Табло счета – общее количество летательных аппаратов противника, количество не уничтоженных и общее количество очков за уничтоженные летательные аппараты.

3. **Салочки.** Один человек догоняет другого. Первый управляется пользователем, второй – компьютером. Препятствиями являются деревья, кустарник, овраги. Табло счета – время погони.

4. **Змейка**. Змея гоняется за человеком, который собирает монеты, имеющие разную ценность, и после этого должен спрятаться в домик, расположенный в правой нижней части экрана. Препятствиями на поле игры являются кусты и деревья. Человек управляется пользователем. Алгоритм движения змейки определяется программой. Табло счета – общая ценность всех собранных монет.

5. **Бой летающих тарелок.** Одна из тарелок управляется пользователем, другая – компьютером. Тарелка противника имеет фиксированное количество «жизней». После каждого попадания количество «жизней» тарелки противника уменьшается. Препятствия на игровом поле – столбы. Табло счета – общее и оставшееся количество «жизней» тарелки противника.

6. **Arcanoid.** Шарик отскакивает от биты, расположенной в одном конце экрана, и разбивает несколько рядов разноцветных прямоугольников, расположенных в другом конце экрана. Прямоугольники разных цветов дают разное количество очков. Табло счета – общее количество очков и количество очков, заработанных за разбитые прямоугольники.

7. **Охрана дачного участка от птиц.** На дачном участке случайным образом приземляются и взлетают разные птицы. Человек, управляемый

пользователем, должен бросать по ним палку. Попадание в разных птиц оценивается различным количеством очков. Табло счета – общее количество очков и количество очков, заработанных за попадания в птиц.

8. **Попадание в движущуюся мишень**. Нужно попасть из лука, расположенного в одном конце экрана, в мишень, движущуюся в другом конце экрана. Попадание в различные круги мишени дает разное количество очков. Табло счета – общее количество стрел, оставшееся количество стрел, количество заработанных за попадания очков.

9. **Танк на минном поле**. Необходимо провести танк по минному полю из одного конца экрана в другой. Танк имеет фиксированное число «жизней». Помимо мин на игровом поле должны быть препятствия в виде деревьев. Наезд на мину уменьшает число «жизней» танка. Табло счета – общее и оставшееся количество «жизней».

10. **Сбор желудей и листьев, падающих с дуба.** Управляемый вами человек должен собирать желуди и листья, падающие с дуба, которые оцениваются различным количеством очков, а также опасаться прямого попадания желудей, которое приводит к потере одной «жизни». Табло счета – общее и оставшееся количество «жизней» человека, общее количество набранных очков.

11. **Бомбардировщик.** Управляемый игроком самолет должен бомбить появляющиеся и исчезающие на земле объекты. Разные объекты оцениваются различным количеством очков. Следует опасаться попадания в грозовые облака, приводящие к гибели самолета. Табло счета – общее количество набранных за попадания очков.

12. **Погоня за дичью**. Волк, управляемый пользователем, гоняется по лесу за животными. Разные животные оцениваются различным количеством очков. Табло счета – общее количество животных, количество оставшихся животных и количество заработанных очков.

13. **Охота за рыбами**. В пруду щука, управляемая пользователем, гоняется за рыбами. Препятствиями служат водоросли, коряги и т. п.

Различные рыбы оцениваются разным количеством очков. Табло счета – общее количество рыб, количество оставшихся рыб и количество заработанных очков.

14. **Бильярд**. На игровом столе находятся два шара и четыре лузы. Ударом кия, который управляется пользователем, необходимо попасть в один из шаров. Шары считаются забитыми в лузу, если до этого произошло их соударение. Табло счета – количество забитых шаров.

15. **Теннис.** Есть несколько шариков. Управляемая игроком ракетка ударяет по шарику. Тот попадает в стену, находящуюся в противоположной стороне экрана, и отскакивает в сторону игрока. В случае, если шарик не попадает на ракетку, он теряется. Попадания в различные части стены дают различное количество очков. Табло счета – общее количество шариков, количество оставшихся шариков, количество набранных очков.

Порядок выполнения лабораторной работы:

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

СПИСОК РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ

1. Корриган Дж. Компьютерная графика. Секреты и решения / Дж. Корриган. – М.: Энтроп, 1995. – 352 с.

2. Поляков А. Программирование графики: GDI+ и DirectX / А. Поляков, В. Брусенцев. – СПб.: БХВ-Петербург, 2005. – 360 с.

3. Рубанцев В. Занимательная графика на Си-шарпе. Подробный самоучитель / В. Рубанцев. – М.: RVGames, 2012. – 205 с.

4. Рубанцев В. Программирование на языке C# 5.0: Компьютерная графика. Средний уровень / В. Рубанцев. – М.: RVGames, 2014. – 957 с.

5. Фленов М. Е. Библия C# / М. Е. Фленов. – СПб.: БХВ-Петербург, 2011. – 560 с.

6. Нейгел Кр. C# 2005 для профессионалов / Кр. Нейгел, Б. Ивьен, Дж. Глинн и др. – М.: Издательский дом «Вильямс», 2006. – 1376 с.

СПИСОК ЭЛЕКТРОННЫХ РЕСУРСОВ

1. Вывод графики [Электрон. ресурс] – Режим доступа: http://www.pmbk.ru/lister/047/1/index.shtml. – Загл. с экрана.

2. Объекты Graphics и Drawing в Windows Forms [Электрон. ресурс] – Режим доступа: https://msdn.microsoft.com/ru-ru/library/a36fascx (v=vs.110).aspx. – Загл. с экрана.

3. Основы библиотеки System.Drawing Forms [Электрон. ресурс] – Режим доступа: http://grafika.me/node/24. – Загл. с экрана.

4. Фролов А. В. Визуальное проектирование приложений C# [Электрон. ресурс] / А. В. Фролов, Г. В. Фролов – Режим доступа: http://www.frolov-lib.ru/books/msnet/c_sharp2/ch10.html. – Загл. с экрана.

Учебное издание

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
К ВЫПОЛНЕНИЮ И ОФОРМЛЕНИЮ ЛАБОРАТОРНЫХ РАБОТ К КУРСУ
«ИНЖЕНЕРНАЯ И КОМПЬЮТЕРНАЯ ГРАФИКА»
(ЧАСТЬ 1)

Автор-составитель: **Котенко** Владислав Николаевич

Редактор М. В. Совпель
Технический редактор Н.Н. Гиюк

Подписано в печать 29.08.2016 г.
Формат 60х84/16. Бумага офсетная.
Печать – цифровая. Усл.-печ. Л. 4,71.
Тираж 100 экз. Заказ № 16-Ав169.

Донецкий национальный университет
83001, г. Донецк, ул. Университетская, 24
Свидетельство о внесении субъекта
Издательской деятельности в государственный реестр
Серия ДК № 1854 от 24.06.2004 г.