**Теоретический материал.**

Векторная графика – это способ представления изображений, основанный на использовании элементарных геометрических объектов: точек, линий, сплайнов и многоугольников.

Эту графику часто называют объектно-ориентированной, так как в ее основе лежат примитивные объекты, такие как окружности, линии, сферы, кубы и тому подобное, и на их основе создаются более сложные объекты, образуя своего рода иерархическое дерево.

Любая векторная иллюстрация, то есть сама картинка, – это верхний уровень иерархии.

Ниже идут объекты, представляющие собой разнообразные векторные формы. Каждый из этих объектов, в свою очередь, состоит из открытых или замкнутых контуров. Контуром в данном случае называется кривая, представляющая собой очертания некоторого графического объекта. Если начальная и конечная точки кривой совпадают, то такой контур называется замкнутым. Если же контур имеет четко обозначенные концевые точки, то он называется открытым.

Каждый контур состоит из сегментов, занимающих в контуре определенное положение, фиксируемое опорными точками или узлами – началом и концом сегмента. Замкнутые контуры могут иметь свойство заливки – цвета или узора, выводимого в замкнутой области, ограниченной кривой.

Самый нижний уровень иерархии образуют основные элементы векторных изображений: узлы и отрезки линий, их соединяющие.

Таким образом, упрощенную иерархию векторного изображения можно представить следующим образом (рис. 1.1).



Рис. 1 1 Иерархия векторного изображения

Информация о любом объекте в векторной графике хранится в описательной форме. Например, окружность будет храниться следующим образом:

* число, соответствующее координате х центра окружности;
* число, соответствующее координате у центра окружности;
* величина радиуса окружности;
* цвет и толщина контура;
* цвет и характер заливки (при ее наличии).

Векторная графика широко используется в рекламных и дизайнерских агентствах, редакциях и издательствах, так как выполняемые ими оформительские работы основаны на применении шрифтов и простейших геометрических элементов. Применение именно векторной графики позволяет более эффективно решать задачи подобного рода. Это обуславливается следующими достоинствами данного способа представления изображений.

* Так как размер описательной части не зависит от реальной величины объекта, то можно описать сколь угодно большой объект минимальным объемом информации и, следовательно, сохранить его в файл минимального размера.
* Так как информация об объекте хранится в описательной форме, то можно легко изменять параметры объекта, и такие операции, как перемещение, масштабирование, вращение, заполнение и т. д., не будут ухудшать качества рисунка.
* Наряду с достоинствами, векторная графика обладает и недостатками:
* Некоторые объекты не могут быть представлены в векторном виде, так как для их изображения может потребоваться большое количество объектов высокой сложности, что, в свою очередь, повлечет рост объемов требуемой памяти и времени отображения картинки.
* Сложность преобразования растрового изображения в векторное.
* Сложность масштабирования картинок с низким разрешением.

Множество технологий обладает возможностями работы с векторной графикой. В рамках данной лабораторной работы мы познакомимся с возможностями технологии GDI+. Но сначала рассмотрим, что это такое.

GDI (Graphics Device Interface, Graphical Device Interface) является интерфейсом Windows, предназначенным для представления графических объектов и вывод их на монитор или принтер. Он отвечает за отрисовку линий, кривых, отображение шрифтов и обработку палитры и обеспечивает унификацию работы с различными устройствами вместо прямого доступа к оборудованию, что позволяет одними и теми же функциями рисовать на различных устройствах, получая при этом практически одинаковые изображения.

Данная технология предоставляет богатые возможности для работы с векторной графикой. Основной ее элемент (линия) имеет такие атрибуты, как толщина, рисунок пунктира, цвет, которые собраны в одном объекте. Кроме того, существуют различные способы заливки областей, также собранные в одном объекте, и поддержка матрицы поворотов и растяжений изображений векторной графики.

На базе технологии GDI была разработана GDI+. Это улучшенная среда для 2D-графики, расширенная возможностями сглаживания линий, использования координат с плавающей точкой, градиентной заливки, использованием ARGB-цветов и т. п. Рассмотрим основные возможности для работы с этой технологией, предоставляемые C#.

В рамках данной лабораторной работы мы освоим технику рисования картинок в оперативной памяти с последующим выводом их на форму. Для рисования мы будем использовать объект класса Bitmap, а для вывода на форму – элемент управления PictureBox.

Рассмотрим написание простейшего приложения, когда по нажатию на кнопку на форме будет отображаться красный кружок. Для этого создаем новый проект типа WindowsFormsApplication: ***Файл – Создать – Проект – Приложение Windows Form***.

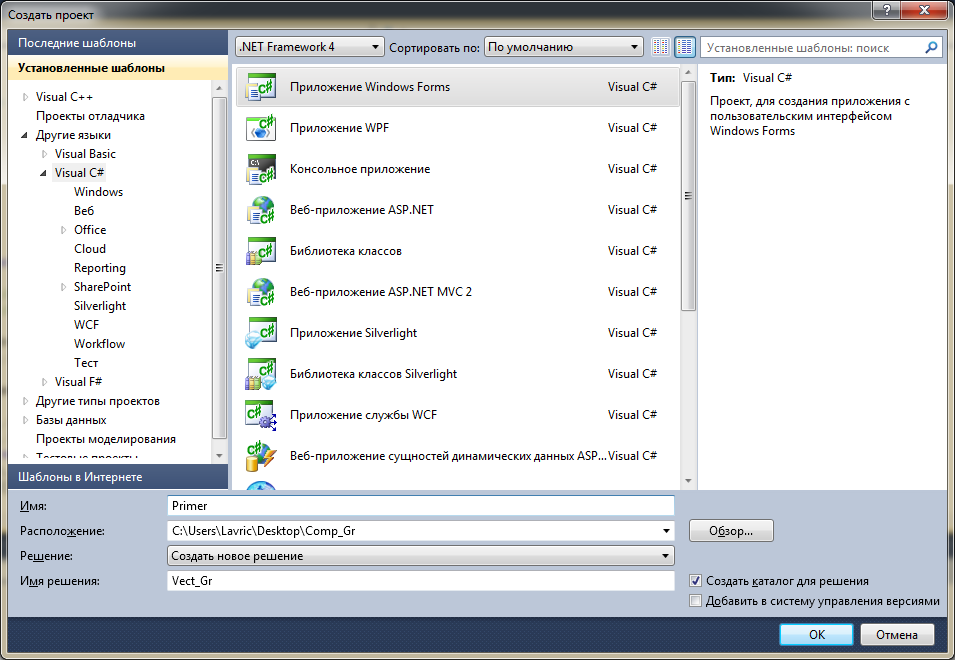


Рис. 1 2 Окно создания проекта Приложения Windows Forms

В окне создания проекта ввести имя проекта, имя решения и указать место расположения решения. Для завершения этапа создания проекта нажать на кнопку ОК.

Размещаем на форме компоненты PictureBox и Button. Для добавления данных элементов воспользуемся **Панелью элементов** (если ее нет, выполним последовательность команд **Вид – Панель элементов**), а затем выполнить двойной щелчок левой кнопкой мыши, предварительно установив курсормыши на соответствующем названии компонента на Панели элементов.

Форма примет следующий вид Рис. 1.3.

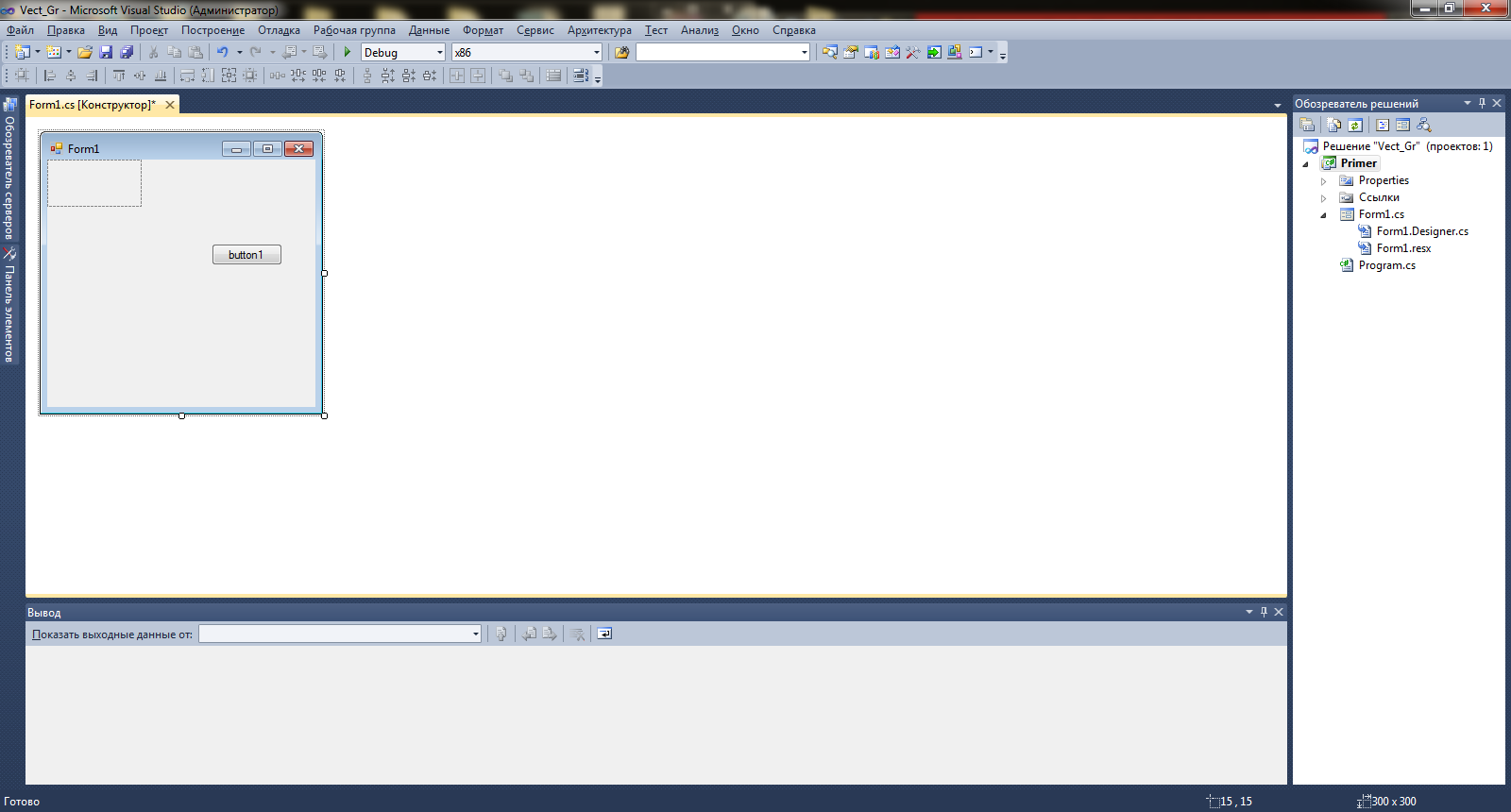


Рис. 1 3 Вид формы

Переходим к методу обработки нажатия на кнопку. Для этого выполним войной щелчок левой кнопки мыши, предварительно установив ее курсор мыши на компоненте Button1. В открывшейся вкладке кода программы получим метод:

private void button1\_Click(object sender, EventArgs e)

В нем и будем писать следующий код. Во-первых, создадим объект класса Bitmap, с размерами, совпадающими с нашим PictureBox. В данном случае Bitmap выполняет роль холста, а PictureBox – рамки, в которую этот холст повесят. Поэтому размеры должны совпадать. В итоге получаем код:

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Теперь создадим объект класса Graphics – основного класса, предоставляющего доступ к возможностям GDI+. Для данного класса не определено ни одного конструктора. Его объект создается в ходе выполнения ряда методов применительно к конкретным объектам, у которых есть поверхность для рисования. Одним из таких объектов и является Bitmap. Поэтому создаем объект класса Graphics следующим образом:

Graphics gr = Graphics.FromImage(bmp);

Теперь все вызовы методов отображения фигур будут отрабатывать на нашей битовой карте. Класс Graphics содержит множество методов рисования вида Fill\* или Draw\*, отвечающих за отображение закрашенных или не закрашенных фигур. Первая группа методов в качестве одного из параметров принимает объект типа Brush (кисть), а вторая – объект типа Pen (карандаш). Исключение – метод DrawString, который отображает текст. Этот метод в качестве одного из параметров принимает объект Brush. Для того чтобы отобразить красный круг, как мы задумали ранее, необходимо написать следующий код:

gr.FillEllipse(Brushes.Red,10,10,40,40);

Обратите внимание, что координаты 10,10 – это не координаты центра эллипса, а координаты левого верхнего угла прямоугольника, в который будет вписан эллипс. Brushes.Red – стандартная красная кисточка. 40,40 – высота и ширина эллипса. Если вы сейчас запустите приложение и нажмете на кнопку, то ничего не произойдет. Вернее, эллипс будет нарисован, но мы его не увидим, так как сейчас он находится в оперативной памяти. Для того чтобы отобразить нарисованную картинку на форме, необходимо добавить следующую строчку кода:

pictureBox1.Image = bmp;

В итоге на экране вы должны увидеть примерно следующее (рис. 1.4):

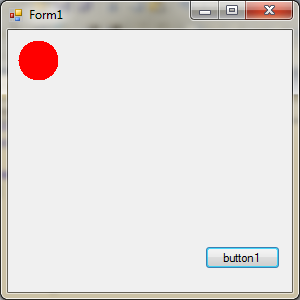


Рис. 1 4 Красный круг

Рисование в технологии GDI+ осуществляется слоями, которые перекрывают друг друга, поэтому если мы напишем вот такой код:

gr.FillEllipse(Brushes.Red,10,10,40,40);

gr.FillEllipse(Brushes.Green,10,10,40,40); ,

то на форме увидим только зеленый круг.

Теперь рассмотрим способы создания различных кисточек. Во-первых, создадим кисточку случайного цвета. Для этого напишем следующий код:

Random r=new Random();

SolidBrush sb=new SolidBrush(Color.FromArgb(r.Next(0,255), r.Next(0,255), r.Next(0,255), r.Next(0,255)));

Метод FromArgb принимает в качестве параметров 4 числа, кодирующих цвет в системе ARGB: A (альфа) – прозрачность; R (red) – красный цвет; G (green) – зеленый цвет; В (blue) – синий цвет.

При использовании этой кисти в рисовании нашего круга:

SolidBrush sb = new SolidBrush(Color.FromArgb(r.Next(0, 255), r.Next(0, 255), r.Next(0, 255), r.Next(0, 255)));

gr.FillEllipse(sb, 10, 10, 40, 40);

мы можем получить следующий результат (рис. 1.5):

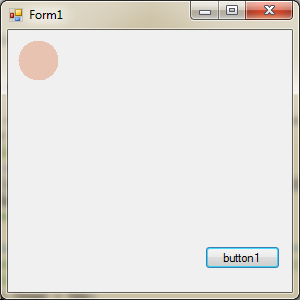


Рис. 1 5 Использование кисточки случайного цвета

Для создания кисточек с более интересными характеристиками необходимо подключить ***namespace System.Drawing.Drawing2D***.

Рассмотрим создание градиентной кисти. Напишем следующий код:

LinearGradientBrush gradientBrush = new LinearGradientBrush(new Point(0, 0),new Point(40, 40), Color.Green, Color.Blue);

При использовании этой кисти в рисовании нашего круга:

LinearGradientBrush gradientBrush = new LinearGradientBrush(new Point(0, 0), new Point(40, 40), Color.Green, Color.Blue);

gr.FillEllipse(gradientBrush, 10, 10, 40, 40);

мы получим следующий результат (рис. 1.6):

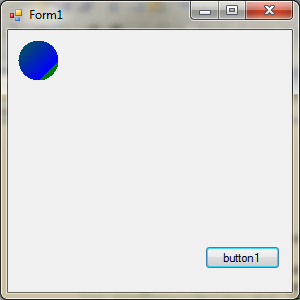


Рис. 1 6 Использование градиентной кисти

Как мы знаем, линейная градиентная заливка – это вид заливки, которой необходимо задать цвет и координаты двух ключевых точек, расположенных на одной прямой, а цвет точек между ними рассчитывается относительно них по определенным математическим алгоритмам. В итоге получают плавные переходы из одного цвета в другой.

Следующая кисть, которую мы рассмотрим, – штриховая.

Создается она следующим образом:

HatchBrush hatchBrush = new HatchBrush(HatchStyle.Cross, Color.Red, Color.Yellow);

Первый параметр конструктора задает вид штриховки. Это системное перечисление с множеством вариантов. Второй параметр отвечает за цвет штрихов, а третий – цвет фона.

При использовании этой кисти в рисовании нашего круга:

HatchBrush hatchBrush = new HatchBrush(HatchStyle.Cross, Color.Red, Color.Yellow);

gr.FillEllipse(hatchBrush, 10, 10, 40, 40);

мы получим следующий результат (рис. 1.7):

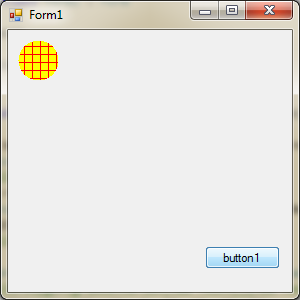


Рис. 1 7 Штрихованный круг

Теперь рассмотрим создание карандашей. Их можно создавать на основе цвета или на основе ранее созданной кисти. Можно указывать толщину линии, а также форму ее конечной точки (только для незамкнутых контуров) и прочее. Ниже приведен код для создания

различных карандашей:

Pen p1=new Pen(Color.Red); // Обычное красное перо

Pen p2=new Pen(Color.Green, 4); // Ширина 4 пикселя

p2.EndCap = LineCap.ArrowAnchor; //Стрелочка на конце

Pen p3=new Pen(gradientBrush, 6); // Градиент

Pen p4=new Pen(hatchBrush, 6); // Штриховка

Для вывода текста необходимо использовать метод DrawString, принимающий в качестве параметров строку, которую нужно вывести, объект класса Font, кисть и координаты вывода. Конструктор класса Font, в свою очередь, принимает название и размер шрифта. Приведем код вывода надписи «Hello, world!»:

gr.DrawString(“Hello, world!”, new Fonr(“Arial”,15),hatchBrush,20,20);

Рассмотрим некоторые элементы, описанные выше более подробно.

**Методы и свойства класса Graphics**

**Графическая поверхность**

Графическая поверхность состоит из отдельных точек – пикселов. Положение точки на графической поверхности характеризуется горизонтальной (X) и вертикальной (Y) координатами (рис. 1.8). Координаты точек отсчитываются от левого верхнего угла и возрастают слева направо (координата X) и сверху вниз (координата Y). Левая верхняя точка графической поверхности имеет координаты (0, 0). Размер графической поверхности формы соответствует размеру *клиентской* области (т. е. без учета высоты области заголовка и ширины границ) формы (свойство ClientSize), а размер графической поверхности компонента PictureBox – размеру компонента.

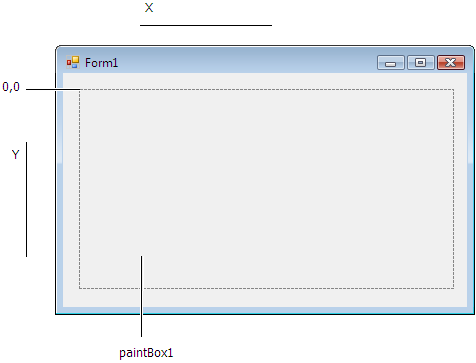


Рис. 1 8 Координаты точек графической поверхности формы

**Карандаши и кисти**

Методы рисования графических примитивов (например, DrawLine – линия, DrawRectangle – прямоугольник, FillRectangle – область) используют *карандаши* и *кисти*. Карандаш (объект Pen) определяет вид линии, кисть (объект Brush) - вид закраски области. Например, метод

**DrawLine(Pens.Black, 10,20, 100, 20);**

рисует из точки (10, 20) горизонтальную линию длиной 100 пикселов, ис-пользуя черный (Black) карандаш из стандартного набора карандашей (Pens), а метод

**FillRectangle(Brushes.Green, 5, 10, 20, 20);**

при помощи стандартной кисти зеленого (Green) цвета рисует зеленый квадрат, левый верхний угол которого находится в точке (5, 10).

Карандаши и кисти определены в пространстве имен System.Drawing.

**Карандаш**

Карандаш определяет вид линии – цвет, толщину и стиль. В распоряжении программиста есть два набора карандашей: стандартный и системный. Также программист может создать собственный карандаш.

*Стандартный набор карандашей* – это цветные карандаши (всего их 141), которые рисуют непрерывную линию толщиной в *один пиксел*. Некоторые карандаши из стандартного набора приведены в табл. 1.1.

Табл.1.1 Некоторые карандаши из стандартного набора

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Карандаш** | | **Цвет** |
| Pens.Red | Красный |
| Pens.Orange | Оранжевый |
| Pens.Yellow | Желтый |
| Pens.Green | Зеленый |
| Pens.LightBlue | Голубой |
| Pens.Blue | Синий |
| Pens.Purple | Пурпурный |
| Pens.Black | Черный |
| Pens.LightGray | Серый |
| Pens.White | Белый |
| Pens.Transparent | Прозрачный |

*Системный набор карандашей* – это карандаши, цвет которых определяется текущей цветовой схемой операционной системы и совпадает с цветом какого-либо элемента интерфейса пользователя. Например, цвет карандаша SystemPens.ControlText совпадает с цветом, который в текущей цветовой схеме используется для отображения текста на элементах управления (командных кнопках и др.), а цвет карандаша SystemPens.WindowText – с цветом текста в окнах сообщений.

Карандаш из стандартного (Pens) и системного (SystemPens) наборов рисует непрерывную линию толщиной в один пиксел. Если надо нарисовать пунктирную линию или линию толщиной больше единицы, то следует использовать карандаш программиста.

*Карандаш программиста* – это объект типа Pen, свойства которого (табл. 1.2) определяют вид линии, рисуемой карандашом.

Табл.1.2 Свойства объекта Pen

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Color | Цвет линии |
| Width | Толщина линии (задается в пикселах) |
| DashStyle | Стиль линии (DashStyle.Solid – сплошная; DashStyle.Dash – пунктирная, длинные штрихи; DashStyle.Dot – пунктирная, короткие штрихи; DashStyle.DashDot – пунктирная, чередование длинного и короткого штрихов; DashStyle.DashDotDot – пунктирная, чередование одного длинного и двух коротких штрихов; DashStyle.Custom – пунктирная линия, вид которой определяет свойство DashPattern) |
| DashPattern | Длина штрихов и промежутков пунктирной линии DashStyle.Custom |

Для того чтобы использовать карандаш программиста, его надо создать. Создает карандаш конструктор объекта Pen. Конструктор *перегружаемый*, т. е. для объекта Pen определено несколько конструкторов, которые различаются количеством параметров. Например, конструктор Pen(*Цвет*) создает карандаш указанного цвета толщиной в один пиксел, а Pen(*Цвет*, *Толщина*) – карандаш указанного цвета и толщины. В качестве параметра *Цвет* можно использовать константу типа Color (табл. 1.3). Другие константы типа Color (а их более 100) можно найти в справочной системе.

Табл.1.2 Константы типа Color

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Константа** | | **Цвет** |
| Color.Red | Красный |
| Color.Orange | Оранжевый |
| Color.Yellow | Желтый |
| Color.Green | Зеленый |
| Color.LightBlue | Голубой |
| Color.Blue | Синий |
| Color.Purple | Пурпурный |
| Color.Black | Черный |
| Color.LightGray | Серый |
| Color.White | Белый |
| Color.Transparent | Прозрачный |

Цвет, ширину линии и стиль карандаша, созданного программистом, можно изменить. Чтобы это сделать, надо изменить значение соответствующего свойства.

В листинге 1.1 приведена процедура, которая демонстрирует создание и ис-пользование карандаша программиста.

**Листинг 1.1 Создание карандаша**

private void button1\_Click(object sender, EventArgs e)

{

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics gr = Graphics.FromImage(bmp);

Pen p1 = new Pen(Color.Red,2);// Красный карандаш тощиной 2 пиксела

//рисуем контур прямоугольника

gr.DrawRectangle(p1, 10, 10, 100, 100);

//теперь меняем карандаш на зеленый толщиной 4 пиксела

p1.Width = 4;

p1.Color = Color.Green;

gr.DrawRectangle(p1, 20, 20, 100, 100);

//меняем линию на пунктирную толщиной 5 пикселов

p1.Width = 5;

p1.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;

//рисуем прямоугольник пунктиром

gr.DrawRectangle(p1,30,30,100, 100);

// показываем изображение

pictureBox1.Image = bmp;

}

Результат:

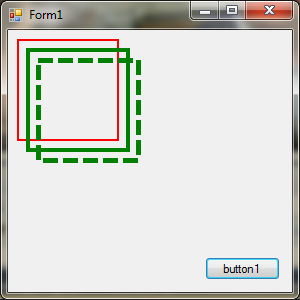


Рис. 1 9 Результат листинга 1.1

Программист может создать карандаш, который рисует пунктирную линию, отличную от стандартной. Чтобы это сделать, надо в свойство DashPattern поместить ссылку на массив описания отрезка линии, а свойству DashStyle присвоить значение DashStyle.Custom.

Массив описания отрезка линии — это состоящий из двух элементов одномерный массив (типа float), который содержит информацию об отрезке пунктирной линии (цепочке "штрих — пропуск"). Первый элемент массива задает длину штриха, второй — пропуска. Приведенная в листинге 1.2 функция демонстрирует процесс создания карандаша, который рисует пунктирную линию, определенную программистом.

**Листинг 1.2 Линия определенная программистом**

private void button1\_Click(object sender, EventArgs e)

{

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics gr = Graphics.FromImage(bmp);

Pen p1 = new Pen(Color.Red,2);// Красный карандаш тощиной 2 пиксела

// Создаем массив типа float, длиной 2

float[] myPattern = new float[2];

myPattern[0] = 15; // штрих

myPattern[1] = 5; // пропуск

p1.Color = Color.Olive;

p1.DashStyle = System.Drawing.Drawing2D.DashStyle.Custom;

p1.DashPattern = myPattern;

gr.DrawRectangle(p1,40,40,100,100); // прямоугольник

pictureBox1.Image = bmp;

}

Результат:

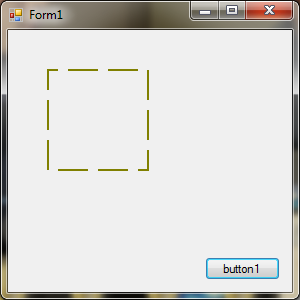


Рис. 1 10 Результат листинга 1.2

**Кисть**

Кисти используются для закраски внутренних областей геометрических фигур. Например, инструкция

gr.FillRectangle(Brushes.DeepSkyBlue, x, y, w, h);

рисует закрашенный прямоугольник.

В приведенном примере Brushes.DeepSkyBlue – это стандартная кисть тем-но-небесного цвета. Параметры x, y определяют положение прямоугольника; w, h — его размер.

В распоряжении программиста есть три типа кистей: стандартные, штриховые и текстурные.

*Стандартная кисть* закрашивает область одним цветом (сплошная закраска). Всего есть 140 кистей, некоторые из них приведены в табл. 1.4. Полный список кистей можно найти в справочной системе.

Табл. 1.4 Некоторые кисти из стандартного набора

|  |  |
| --- | --- |
| **Кисть** | **Цвет** |
| Brushes.Red | Красный |
| Brushes.Orange | Оранжевый |
| Brushes.Yellow | Желтый |
| Brushes.Green | Зеленый |
| Brushes.LightBlue | Голубой |
| Brushes.Blue | Синий |
| Brushes.Purple | Пурпурный |
| Brushes.Black | Черный |
| Brushes.LightGray | Серый |
| Brushes.White | Белый |
| Brushes.Transparent | Прозрачный |

*Штриховая кисть* (HatchBrush) закрашивает область путем штриховки. Область может быть заштрихована горизонтальными, вертикальными или наклонными линиями разного стиля и толщины. В табл. 1.5 перечислены некоторые из возможных стилей штриховки. Полный список стилей штриховки можно найти в справочной системе.

Табл. 1.5 Некоторые стили штриховки областей

|  |  |
| --- | --- |
| **Стиль** | **Штриховка** |
| HatchStyle.LightHorizontal | Редкая горизонтальная |
| HatchStyle.Horizontal | Средняя горизонтальная |
| HatchStyle.NarrowHorizontal | Частая горизонтальная |
| HatchStyle.LightVertical | Редкая вертикальная |
| HatchStyle.Vertical | Средняя вертикальная |
| HatchStyle.NarrowVertical | Частая вертикальная |
| HatchStyle.LageGrid | Крупная сетка из горизонтальных и вертикальных линий |
| HatchStyle.SmallGrid | Мелкая сетка из горизонтальных и вертикальных линий |
| HatchStyle.DottedGrid | Сетка из горизонтальных и вертикальных линий, составленных из точек |
| HatchStyle.ForwardDiagonal | Диагональная штриховка "вперед" |
| HatchStyle.BackwardDiagonal | Диагональная штриховка "назад" |
| HatchStyle.Percent05 – HatchStyle.Percent90 | Точки (степень заполнения 5%, 10%, ..., 90%) |
| HatchStyle.HorizontalBrick | "Кирпичная стена" |
| HatchStyle.LargeCheckerBoard | "Шахматная доска" |
| HatchStyle.SolidDiamond | "Бриллиант" ("Шахматная доска", повернутая на 45) |
| HatchStyle.Sphere | "Пузырьки" |
| HatchStyle.ZigZag | "Зигзаг" |

В листинге 1.3 приведена функция, демонстрирующая процесс создания и использования штриховой кисти. При создании кисти конструктору передаются: константа HatchStyle, которая задает вид штриховки, и две константы типа Color, первая из которых определяет цвет штрихов, вторая – цвет фона.

**Листинг 1.3. Пример создания и использования штриховой кисти**

private void button1\_Click(object sender, EventArgs e)

{

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics gr = Graphics.FromImage(bmp);

HatchBrush p1 = new HatchBrush(HatchStyle.DottedGrid,Color.Black, Color.SkyBlue);

gr.FillRectangle(p1,20,20,90,60);

pictureBox1.Image = bmp;

}

*Замечание*: для нормальной роботы программы необходимо в код добавить подключение пространства имен

**using namespace System.Drawing.Drawing2D;**

Результат:

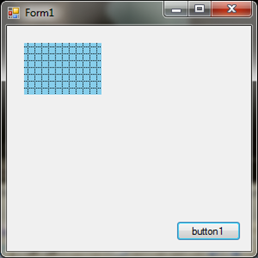


Рис. 1 11 Результат работы листинга программы 1.3

*Текстурная кисть* (TextureBrush) представляет собой рисунок, который обычно загружается во время работы программы из файла (bmp, jpg или gif) или из ресурса. Закраска области текстурной кистью выполняется путем дублирования рисунка внутри области.

Листинг 1.4 демонстрирует процесс создания и использования текстурной кисти. Кисть создает конструктор, которому в качестве параметра передается текстура – имя графического файла.

Возможна градиентная, в простейшем случае двухцветная, закраска области, при которой цвет закраски области меняется от одного к другому вдоль линии градиента.

Чтобы закрасить область градиентом, надо создать *градиентную* кисть. При создании кисти конструктору передаются линия градиента и две константы, определяющие цвет градиента. Например:

myBrush = gcnew System.Drawing.Drawing2D.LinearGradientBrush

(Point(10,10),Point(110,10), Color.Red, Color.White);

После того как градиентная кисть будет создана, ее можно использовать для закраски области:

e->Graphics->FillRectangle(myBrush,10,10,100,40);

Следует обратить внимание, для равномерной закраски области координаты точек линии градиента должны находиться на границе закрашиваемой области или быть вне ее.

Иногда возникает необходимость, используя один и тот же градиент, закрасить несколько областей разного размера. Если попытаться закрасить область большего размера градиентной кистью меньшего размера, то область большего размера будет выглядеть так, как показано на рис. 1.6.

**Листинг 1.4. Пример создания и использования текстурной кисти**

private void button1\_Click(object sender, EventArgs e)

{

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics gr = Graphics.FromImage(bmp);

LinearGradientBrush p1 = new LinearGradientBrush(new Point(10, 10), new Point(110, 10), Color.Red, Color.White);

gr.FillRectangle(p1,20,20,90,60);

pictureBox1.Image = bmp;

}

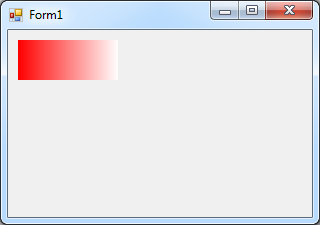


Рис. 1 12 Размер градиентной кисти соответствует размеру области

Чтобы область была закрашена равномерно, для нее надо создать свою (новую) градиентную кисть или трансформировать существующую. Приведенная в листинге 1.5 функция показывает, как это сделать. Она же показывает, как можно изменить цвет градиентной кисти.

**Листинг 1.5. Пример создания и изменения градиентной кисти**

private void button1\_Click(object sender, EventArgs e)

{

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics gr = Graphics.FromImage(bmp);

Rectangle myRect = new Rectangle(20,10,200,40);

LinearGradientBrush myBrush = new LinearGradientBrush(myRect,Color.Red,Color.White,0.0f,true);

gr.FillRectangle(myBrush, myRect);

// изменить цвет

Color[] nc=new Color[2]; //nc — NewColor

nc[0] = Color.Green;

nc[1] = Color.Yellow;

myBrush.LinearColors = nc;

// Так как будем закрашивать область другого размера,

// то надо изменить размер градиентной кисти

Point[]transformArray ={new Point(20,60), new Point(420,60), new Point(20,100)};

Matrix myMatrix=new Matrix(myRect,transformArray);

myBrush.MultiplyTransform(myMatrix, MatrixOrder.Prepend);

// рисуем новой кистью

gr.FillRectangle(myBrush, 20, 60, 400, 40);

// восстановить размер кисти

myBrush.ResetTransform();

// рисуем область такого же размера, как и первую

gr.FillRectangle(myBrush, 20, 120, 200, 40);

pictureBox1.Image = bmp;

}

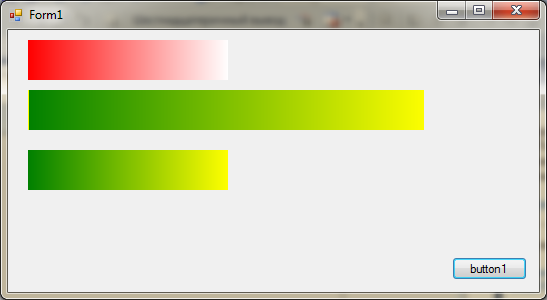


Рис. 1 13 Результат создания и изменения градиентной кисти

**Графические примитивы**

Любая картинка, чертеж, схема представляет собой совокупность графических *примитивов*: точек, линий, окружностей, дуг, текста и др.

Вычерчивание графических примитивов на графической поверхности (Graphics) выполняют соответствующие методы (табл. 1.6).

Табл.1.6 Некоторые методы вычерчивания графических примитивов

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| DrawLine(Pen, x1, y1, x2, y2), DrawLine(Pen, p1, p2) | Рисует линию. Параметр Pen определяет цвет, толщину и стиль линии; параметры x1, y1, x2, y2 или p1 и p2 — координаты точек начала и конца линии |
| DrawRectangle(Pen, x, y, w, h) | Рисует контур прямоугольника. Параметр Pen определяет цвет, толщину и стиль границы прямоугольника: параметры x, y– координаты левого верхнего угла; параметры w и h за-дают размер прямоугольника |
| FillRectangle(Brush, x, y, w, h) | Рисует закрашенный прямоугольник. Параметр Brush определяет цвет и стиль закраски прямоугольника; параметры x, y – координаты левого верхнего угла; параметры w и h задают размер прямоугольника |
| DrawEllipse(Pen, x, y, w, h) | Рисует эллипс (контур). Параметр Pen определяет цвет, толщину и стиль линии эллипса; параметры x, y, w, h – координаты левого верхнего угла и размер прямоугольника, внутри которого вычерчивается эллипс |
| FillEllipse(Brush, x, y, w, h) | Рисует закрашенный эллипс. Параметр Brush определяет цвет и стиль закраски внутренней области эллипса; параметры x, y, w, h – координаты левого верхнего угла и размер прямоугольника, внутри которого вычерчивается эллипс |
| DrawPolygon(Pen, P) | Рисует контур многоугольника. Параметр Pen определяет цвет, толщину и стиль линии границы многоугольника; параметр P (массив типа Point) — координаты углов многоугольника |
| FillPolygon(Brush, P) | Рисует закрашенный многоугольник. Пара-метр Brush определяет цвет и стиль закраски внутренней области многоугольника; пара-метр P (массив типа Point) – координаты углов многоугольника |
| DrawString(str, Font, Brush, x, y) | Выводит на графическую поверхность строку текста. Параметр Font определяет шрифт; Brush – цвет символов; x и y – точку, от ко-торой будет выведен текст |
| DrawImage(Image, x, y) | Выводит на графическую поверхность иллюстрацию. Параметр Image определяет иллюстрацию; x и y – координату левого верхнего угла области вывода иллюстрации |

Один и тот же элемент можно нарисовать при помощи *разных*, но имеющих *одинаковые* имена методов (вспомните: возможность объявления функций, имеющих одинаковые имена, но разные параметры, называется *перегрузкой).*

Например, прямоугольник можно нарисовать методом DrawRectangle, которому в качестве параметров передаются координаты левого верхнего угла и размеры прямоугольника:

gr.DrawRectangle(Pens.Black, x, x, w, h)

Эту же задачу может решить метод DrawRectangle, которому в качестве параметра передается структура типа Rectangle, поля которой определяют прямоугольник (положение и размер):

Rectangle aRect = Rectangle(20,100,50,50);

gr.DrawRectangle(Pens.Blue, aRect);

Существование нескольких методов, выполняющих одну и ту же задачу, позволяет программисту выбрать метод, наиболее подходящий для решения конкретной задачи.

В качестве параметров методов вычерчивания графических примитивов часто используется структура Point. Ее поля X и Y определяют положение (координаты) точки графической поверхности. Например:

Point p1 = new Point(10, 10);

Point p2 = new Point(100, 10);

// рисуем линию из p1 в p2

gr.DrawLine(Pens.Green, p1, p2);

**Линия**

Метод DrawLine рисует прямую линию. В инструкции вызова метода следует указать карандаш, которым надо нарисовать линию, и координаты точек начала и конца линии:

DrawLine(aPen, x1, y1, x2, y2)

или

DrawLine(aPen, p1, p2)

Параметр aPen (типа Brush) задает карандаш, которым рисуется линия, x1 и y1 или p1 — точку начала линии, а x2 и y2 или p2 — точку конца линии. Параметры x1, y1, x2 и y2 должны быть одного типа (Integer или Single). Тип параметров p1 и p2 — Point.

Например, инструкция

gr.DrawLine(Pens.Green,10,10,300,10

рисует зеленую линию толщиной в один пиксел из точки (10, 10) в точку (300, 10).

Эту же линию можно нарисовать и так:

Point p1 = new Point(10,10);

Point p2 = new Point(300,10);

gr.DrawLine(Pens.Green, p1, p2);

**Ломаная линия**

Метод DrawLines рисует ломаную линию. В качестве параметров методу передается карандаш (Pen) и массив типа Point, элементы которого содержат координаты узловых точек линии. Метод рисует ломаную линию, последовательно соединяя точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т. д. Например, следующий фрагмент кода рисует ломаную линию, состоящую из четырех звеньев.

**array**<Point>^ p; *// массив точек*

p = gcnew array<Point>(5);

p[0].X = 10; p[0].Y =50;

p[1].X = 20; p[1].Y =20;

p[2].X = 30; p[2].Y =50;

p[3].X = 40; p[3].Y =20;

p[4].X = 50; p[4].Y =50;

e->Graphics->DrawLines(Pens.Green,p);

Метод DrawLines можно использовать для вычерчивания замкнутых контуров. Для этого первый и последний элементы массива должны содержать координаты одной и той же точки.

**Прямоугольник**

Метод DrawRectangle чертит прямоугольник (рис. 1.14). В качестве параметров метода надо указать карандаш, координаты левого верхнего угла и размер прямоугольника:

DrawRectangle(aPen, x, y, w, h);

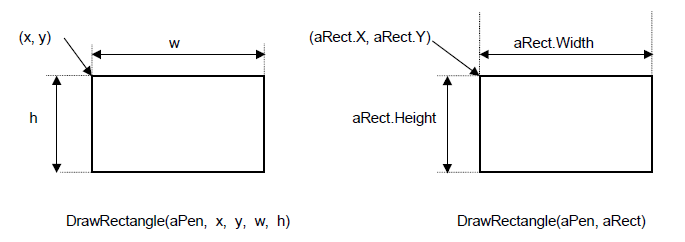


Рис. 1 14 Метод DrawRectangle рисует прямоугольник

Вместо четырех параметров, определяющих прямоугольник, можно указать структуру типа Rectangle:

DrawRectangle(aPen, aRect);

Поля X и Y структуры aRect задают координаты левого верхнего угла прямоугольника, а Width и Height — размер (ширину и высоту).

Вид линии границы прямоугольника (цвет, стиль и ширину) определяет параметр aPen, в качестве которого можно использовать один из стандартных карандашей или карандаш, созданный программистом.

Метод FillRectangle рисует закрашенный прямоугольник. В качестве пара-метров методу надо передать кисть, координаты левого верхнего угла и размер прямоугольника:

FillRectangle(aBrush, x, y, w, h);

Вместо x, y, w и h можно указать структуру типа Rectangle:

FillRectangle(aBrush, aRect);

Параметр aBrush, в качестве которого можно использовать стандартную или созданную программистом штриховую (HatchBrush), градиентную (LineadGradientBrush) или текстурную (TextureBrush) кисть, определяет цвет и стиль закраски области.

Далее приведен фрагмент кода, который демонстрирует использование методов DrawRectangle и FillRectangle.

// Зеленый прямоугольник размером 60х30,

// левый верхний угол которого в точке (10, 10)

Rectangle aRect = new Rectangle(10, 10, 60, 30); // положение и размер прямоугольника

gr.FillRectangle(Brushes.ForestGreen, aRect);

// Желтый прямоугольник с черной границей размером 60х30,

// левый верхний угол которого в точке (100, 10)

aRect.X = 100;

gr.FillRectangle(Brushes.Gold, aRect); // прямоугольник

gr.DrawRectangle(Pens.Black, aRect); // граница

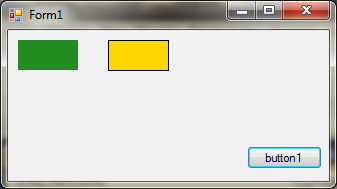


Рис. 1 15 использование методов DrawRectangle и FillRectangle

**Точка**

Казалось бы, чего проще — нарисовать на графической поверхности точку.Но у объекта Graphics нет метода, который позволяет это сделать. Метод SetPixel есть у объекта Bitmap. Поэтому если действительно необходимо сформировать картинку из точек, придется создать объект Bitmap, сформиро вать на его поверхности изображение, а затем это изображение при помощи метода DrawImage вывести на графическую поверхность. Но можно поступить проще — вместо точки вывести квадрат размером в один пиксел.

Например, инструкция

gr.FillRectangle(Brushes.Red,x,y,1,1);

рисует на графической поверхности красную точку.

**Многоугольник**

Метод DrawPolygon чертит многоугольник (контур). Инструкция вызова метода в общем виде выглядит так:

DrawPolygon(aPen, p)

Параметр p — массив типа Point, определяет координаты вершин многоугольника. Метод DrawPolygon чертит многоугольник, соединяя прямыми линиями точки, координаты которых находятся в массиве: первую со второй, вторую с третьей и т. д. Последняя точка соединяется с первой. Вид границы многоугольника определяет параметр aPen, в качестве которого можно использовать стандартный или созданный программистом карандаш.

Закрашенный многоугольник рисует метод FillPolygon. Инструкция вызова метода в общем виде выглядит так:

FillPolygon(aBrush, p)

Параметр aBrush, в качестве которого можно использовать стандартную или созданную программистом штриховую (HatchBrush), градиентную (LineadGradientBrush) или текстурную (TextureBrush) кисть, определяет цвет и стиль закраски внутренней области многоугольника.

Далее приведен фрагмент кода, который демонстрирует использование методов DrawPolygon и FillPolygon – рисует корону.

Point[] p=new Point [5];

p[0].X = 10; p[0].Y =30;

p[1].X = 10; p[1].Y =10;

p[2].X = 30; p[2].Y =20;

p[3].X = 50; p[3].Y =10;

p[4].X = 50; p[4].Y =30;

gr.FillPolygon(Brushes.Gold, p);

gr.DrawPolygon(Pens.Black,p);

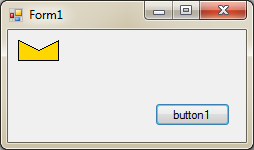


Рис. 1 16 использование методов DrawPolygon и FillPolygon

**Эллипс и окружность**

Метод DrawEllipse чертит эллипс внутри прямоугольной области (рис. 1.17). Если прямоугольник является квадратом, то метод рисует окружность.

Инструкция вызова метода DrawEllipse в общем виде выглядит так:

DrawEllipse(aPen, x, y, w, h);

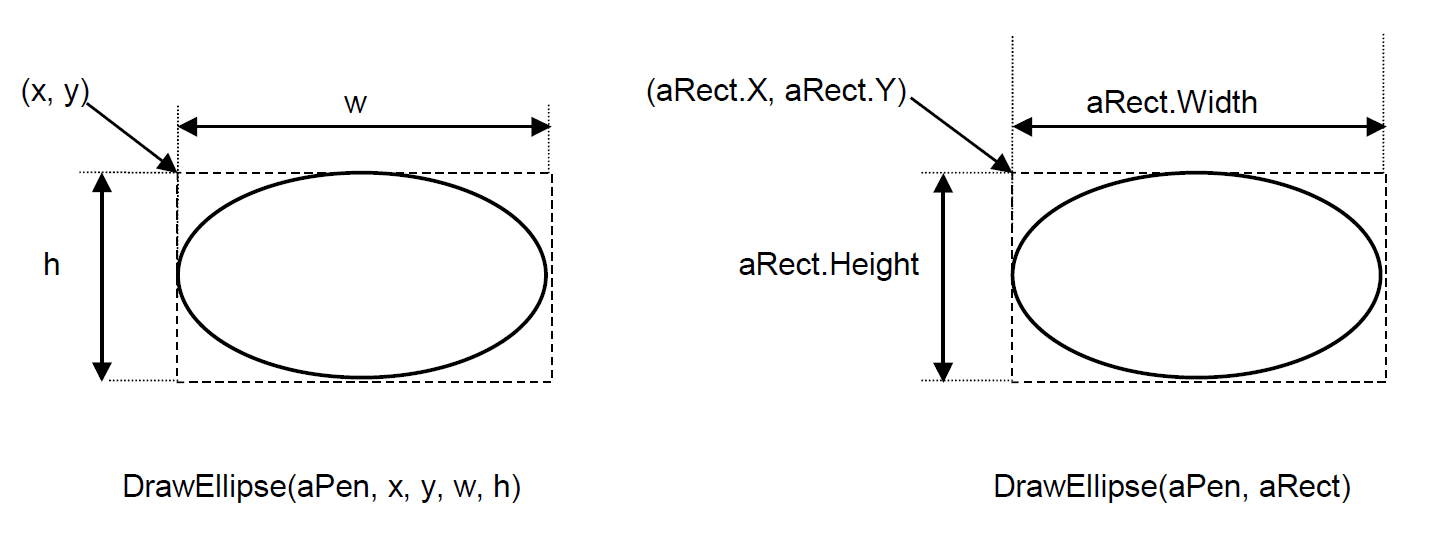


Рис. 1 17 Метод DrawEllipse рисует эллипс

Параметр aPen, в качестве которого можно использовать один из стандартных карандашей или карандаш, созданный программистом, определяет вид (цвет, толщину, стиль) границы эллипса. Параметры x, y, w и h задают координаты левого верхнего угла и размер прямоугольника, внутри которого метод рисует эллипс.

В инструкции вызова метода DrawEllipse вместо параметров x, y, w и h можно указать структуру типа Rectangle:

DrawEllipse(aPen, aRect);

Поля X и Y структуры aRect задают координаты левого верхнего угла области, внутри которой метод рисует эллипс, а поля Width и Height – размер.

Метод FillEllipse рисует закрашенный эллипс. В инструкции вызова метода следует указать кисть (стандартную или созданную программистом), координаты и размер прямоугольника, внутри которого надо нарисовать эллипс:

FillEllipse(aBrush, x, y, w, h);

Кисть определяет цвет и способ закраски внутренней области эллипса. Вместо параметров x, y, w и h можно указать структуру типа Rectangle:

FillEllipse(aBrush, aRect);

**Дуга**

Метод DrawArc рисует дугу – часть эллипса (рис. 2.2). Инструкция вызова метода в общем виде выглядит так:

DrawArc(aPen, x, y, w, h, startAngle, sweepAngle)

Параметры x, y, w и h определяют эллипс (окружность), частью которого является дуга. Параметр startAngle задает начальную точку дуги – пересечение эллипса и прямой, проведенной из центра эллипса и образующей угол startAngle с горизонтальной осью эллипса (угловая координата возрастает по часовой стрелке). Параметр sweepAngle задает длину дуги (в градусах). Если значение sweepAngle положительное, то дуга рисуется от начальной точки по часовой стрелке, если отрицательное – то против. Величины углов задаются в градусах.

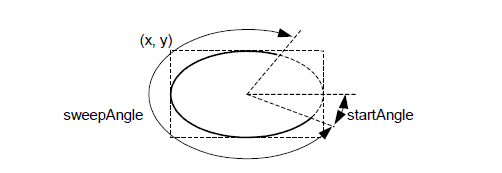


Рис. 1 18 Значения параметров метода DrawArc определяют дугу как часть эллипса

В инструкции вызова метода DrawArc вместо параметров x, y, w и h можно указать структуру типа Rectangle:

DrawArc(aPen, aRect, startAngle, sweepAngle)

**Сектор**

Метод DrawPie рисует границу сектора (рис. 2.3). Инструкция вызова метода выглядит так:

DrawPie(aPen, x, y, w, h, startAngle, sweepAngle);

Параметры x, y, w и h определяют эллипс, частью которого является сектор. Параметр startAngle задает начальную точку дуги сектора – пересечение эллипса и прямой, проведенной из центра эллипса и образующей угол startAngle с горизонтальной осью эллипса (угловая координата возрастает по часовой стрелке). Параметр sweepAngle — длину дуги сектора (в градусах). Если значение sweepAngle положительное, то дуга сектора рисуется от начальной точки по часовой стрелке, если отрицательное – против. Величины углов задаются в градусах.

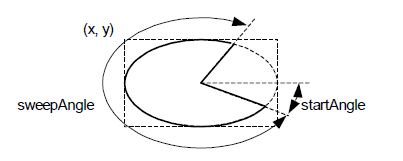


Рис. 1 19 Значения параметров метода DrawPie определяют сектор как часть эллипса

В инструкции вызова метода DrawPie вместо параметров x, y, w и h можно указать структуру типа Rectangle:

**DrawPie(aPen, aRect, startAngle, sweepAngle)**

Метод FillPie рисует сектор. Параметры у метода FillPie, за исключением первого, вместо которого надо указать кисть, такие же, как и у метода DrawPie.

**Текст**

Вывод текста на графическую поверхность выполняет метод DrawString. В инструкции вызова метода указывается строка, шрифт, кисть и координаты точки, от которой надо вывести текст:

DrawString(st, aFont, aBrush, x, y);

Параметр st задает текст, параметр aFont – шрифт, который используется для отображения текста, a aBrush — цвет текста. Параметры x и y определяют координаты левого верхнего угла области отображения текста (рис. 2.5).



Рис. 1 20 Координаты и размер области отображения текста

В программе, окно которой показано, текст выводит функция обработчик события Click.

private void button1\_Click(object sender, EventArgs e)

{

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics gr = Graphics.FromImage(bmp);

String st1 = "Microsoft Visual Studio 2010";

gr.DrawString(st1,this.Font, Brushes.Black, 20,20);

pictureBox1.Image = bmp;

}

В приведенном примере для вывода текста используется шрифт формы, заданный свойством Font. Если текст надо вывести шрифтом, отличным от шрифта, заданного для формы, то этот шрифт следует создать – объявить и инициализировать объект типа Font.

Инструкция создания шрифта (вызова конструктора) выглядит так:

Font aFont = new System.Font(FontFamily,Size,FontStyle);

Параметр FontFamily (строкового типа) задает шрифт, на основе которого создается новый (определяет семейство, к которому относится создаваемый шрифт). В качестве значения параметра FontFamily можно использовать название шрифта, зарегистрированного в системе (Arial, Times New Roman, Tahoma). Параметр Size задает размер (в пунктах) шрифта. Параметр FontStyle определяет стиль символов шрифта (FontStyle::Bold — полужир-ный; FontStyle::Italic — курсив; FontStyle::UnderLine — подчеркнутый). Параметр FontStyle можно не указывать. В этом случае будет создан шрифт обычного начертания (FontStyle::Regular).

Следует обратить внимание, что изменить характеристики созданного шрифта нельзя (свойства FontFamily, Size и Style объекта Font определены "только для чтения"). Поэтому если в программе предполагается использовать разные шрифты, их необходимо создать.

В листинге 1.6 приведена функция обработки события Click, демонстрирующая создание и использование шрифтов для вывода текста на поверхность формы (рис. 1.21).

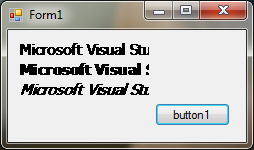


Рис. 1 21 Вывод текста на поверхность формы методом DrawString

**Листинг 1.6 Создание и использование шрифта**

private void button1\_Click(object sender, EventArgs e)

{

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics gr = Graphics.FromImage(bmp);

int x,y;

x = 10;

y = 10;

String st = "Microsoft Visual Studio 2010";

Font rFont = new Font ("Tahoma", 11, FontStyle.Regular);

Font bFont = new Font ("Tahoma", 11, FontStyle.Bold);

Font iFont = new Font ("Tahoma", 11, FontStyle.Italic);

gr.DrawString(st, rFont, Brushes.Black, x, y);

gr.DrawString(st, bFont, Brushes.Black, x, y+20);

gr.DrawString(st, iFont, Brushes.Black, x, y+40);

pictureBox1.Image = bmp;

}

Метод DrawString позволяет вывести текст в прямоугольную область. Причем, если длина текста такая, что вывести его в одну строку нельзя, он будет выведен в несколько строк.

Инструкция вызова метода DrawString, обеспечивающая вывод текста в область, выглядит так:

DrawString(st, aFont, aBrush, aRec);

Параметр aRec задает положение и размер области вывода текста.

В листинге 2.3 приведена функция обработки события Paint, которая де-монстрирует вывод текста в область. Окно программы приведено на рис. 2.7 (границы области вывода текста показаны для наглядности).

Листинг 1.7. **Вывод текста в область**

private void button1\_Click(object sender, EventArgs e)

{

Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics gr = Graphics.FromImage(bmp);

String st1 = "У лукоморья дуб зеленый;\nЗлатая цепь на дубе том:\n И днем и ночью кот ученый\nВсе ходит по цепи кругом;";

// положение и размер области вывода текста

Rectangle aRect = new Rectangle(10,10,200,90);

// вывести текст

gr.DrawString(st1, this.Font, Brushes.Black, aRect);

// показать область отображения текста

gr.DrawRectangle(Pens.Gray, aRect);

pictureBox1.Image = bmp;

}

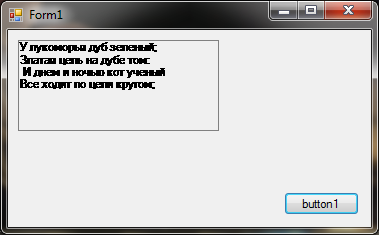


Рис. 1 22 Метод DrawString позволяет вывести текст в область

Часто надо знать, сколько места займет текст, например, для того, чтобы правильно разместить его на графической поверхности. Так, чтобы разместить текст по центру формы, надо знать ширину области вывода. Очевидно, что размер области вывода зависит от шрифта, который используется для отображения текста.

Информацию о размере области вывода текста возвращает метод MeasureString. В инструкции вызова метода надо указать строку и шрифт, который будет использован для ее отображения. Значением метода MeasureString является объект типа SizeF, свойства Width и Height которого содержат информацию о размере области вывода текста.

В листинге 1.8 приведен фрагмент программы, которая демонстрирует использование метода MeasureString. Функция обработки события Paint выводит текст в центре формы (рис. 1.23). Кроме обработки события Paint про-грамма выполняет обработку события Resize, которое возникнет, если пользователь изменит размер окна. Обработка заключается в вызове метода Refresh, который информирует систему о необходимости обновить (перери-совать) окно, что приводит к возникновению события Paint.

Листинг 1.8 **Использование метода MeasureString**

private void Form1\_Paint(object sender, PaintEventArgs e)

{

String st1 = "Microsoft Visual Studio 2010";

Font h1 = new Font("Tahoma",12);

Font h2 = new Font("Tahoma",10);

Font h3 = new Font("Tahoma",9);

float x,y;

float w,h;

w =e.Graphics.MeasureString(st1, h1).Width;

h = e.Graphics.MeasureString(st1, h1).Height;

x = ((float)this.ClientSize.Width-w)/2;

y = 10;

e.Graphics.DrawString(st1, h1, Brushes.Black, x, y);

w = e.Graphics.MeasureString(st1, h2).Width;

h = e.Graphics.MeasureString(st1, h2).Height;

x = ((float)this.ClientSize.Width-w)/2;

y += h;

e.Graphics.DrawString(st1,h2, Brushes.Black, x, y);

w = e.Graphics.MeasureString(st1, h3).Width;

h = e.Graphics.MeasureString(st1, h3).Height;

x = ((float)this.ClientSize.Width-w)/2;

y += h;

e.Graphics.DrawString(st1,h3, Brushes.Black, x, y);

}

private void Form1\_Resize(object sender, EventArgs e)

{

this.Refresh();

}

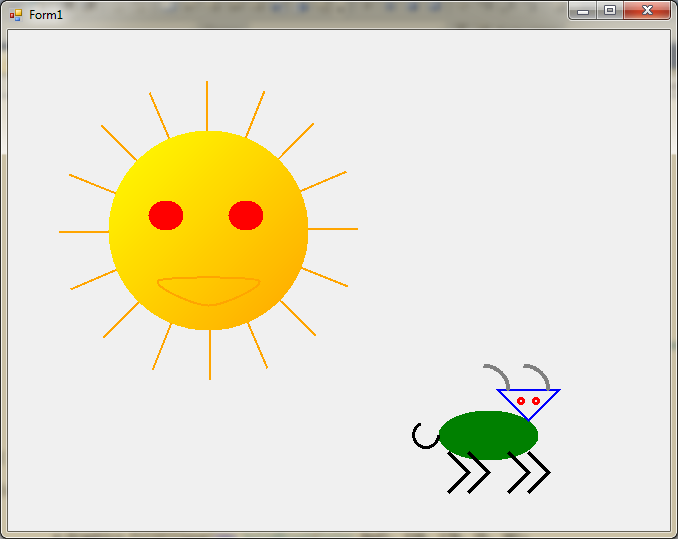


Рис. 1 23 Метод MeasureString позволяет получить информацию о размере области вывода текста, чтобы расположить текст по центру формы

Примеры создания примитивов.

**Пример 1.**

Результат:



Код:

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;

using System.Drawing.Drawing2D;

namespace l1

{

public partial class Form1 : Form

{

float x0 = 430;

float y0 = 380;

float t = 0;

public Form1()

{

InitializeComponent();

}

private void Form1\_Paint(object sender, PaintEventArgs e)

{

Graphics g = e.Graphics; // Сохранить Graphics

DrawSheep(g, x0, y0, t); // Нарисовать барана

int r = 150;

Pen pen = new Pen(Color.Orange, 2);

for (double i = -Math.PI; i < Math.PI; i += Math.PI / 8)

{

e.Graphics.DrawLine(pen, 200, 200, (float)(r \* Math.Cos(i + r) + 200), (float)(r \* Math.Sin(i + r) + 200));

}

Point p1 = new Point(100, 100);

Point p2 = new Point(300, 300);

LinearGradientBrush br = new LinearGradientBrush(p1, p2, Color.Yellow, Color.Orange);

e.Graphics.FillEllipse(br, 100, 100, 200, 200);

Point[] mouth = new Point[]

{

new Point (150,250),new Point(200,275),new Point(250,250)

};

e.Graphics.DrawClosedCurve(pen, mouth);

e.Graphics.FillEllipse(new SolidBrush(Color.Red), 140, 170, 35, 30);

e.Graphics.FillEllipse(new SolidBrush(Color.Red), 220, 170, 35, 30);

r++;

}

void DrawSheep(Graphics g, float x0, float y0, float t)

{

Pen p1 = new Pen(Color.Blue, 2); // заготавливаем перья и кисти

Pen p2 = new Pen(Color.Red, 3); // для рисования барана

Pen p3 = new Pen(Color.Gray, 4);

Pen p4 = new Pen(Color.Black, 3);

SolidBrush b1 = new SolidBrush(Color.Green);

g.FillEllipse(b1, x0, y0, 100, 50); // туловище

float xhead = x0 + 60; // вычисляем координаты головы

float yhead = y0 - 20;

g.DrawPolygon(p1, new PointF[] { new PointF(xhead, yhead), new PointF(xhead + 30, yhead + 30),

new PointF(xhead + 60, yhead) }); // рисуем голову

g.DrawEllipse(p2, x0 + 80, y0 - 12, 5, 5); // рисуем глаза

g.DrawEllipse(p2, x0 + 95, y0 - 12, 5, 5);

g.DrawArc(p3, x0 + 20, y0 - 45, 50, 50, 0, -90); // рисуем рога

g.DrawArc(p3, x0 + 60, y0 - 45, 50, 50, 0, -90);

float xlegs = x0 + 10; // вычисляем координаты ног

float ylegs = y0 + 42; // и рисуем четыре ноги

g.DrawLines(p4, new PointF[] { new PointF(xlegs, ylegs), new PointF(xlegs + 20 \* (1 - t), ylegs + 20), new PointF(xlegs, ylegs + 40) });

xlegs = xlegs + 20;

g.DrawLines(p4, new PointF[] { new PointF(xlegs, ylegs), new PointF(xlegs + 20 \* (1 - t), ylegs + 20), new PointF(xlegs, ylegs + 40) });

xlegs = xlegs + 40;

g.DrawLines(p4, new PointF[] { new PointF(xlegs, ylegs), new PointF(xlegs + 20 \* (1 - t), ylegs + 20), new PointF(xlegs, ylegs + 40) });

xlegs = xlegs + 20;

g.DrawLines(p4, new PointF[] { new PointF(xlegs, ylegs), new PointF(xlegs + 20 \* (1 - t), ylegs + 20), new PointF(xlegs, ylegs + 40) });

g.DrawArc(p4, x0 - 25, y0 + 12, 25, 25, 0, 245); // рисуем хвост

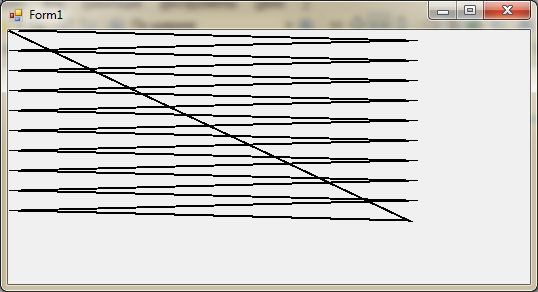
}

}

}

**Пример 2.**

Результат:



Код:

public partial class Form1 : Form

{

Point[] points = new Point[50];

Pen pen = new Pen(Color.Black, 2);

public Form1()

{

InitializeComponent();

}

private void Form1\_Paint(object sender, PaintEventArgs e)

{

Graphics g = e.Graphics;

g.DrawLines(pen, points);

}

private void Form1\_Load(object sender, EventArgs e)

{

for (int i = 0; i < 20; i++)

{

int xPos;

if (i % 2 == 0)

{

xPos = 10;

}

else

{

xPos = 400;

}

points[i] = new Point(xPos, 10 \* i);

}

}

}

**Задания.**

Изучите с помощью справки MSDN методы и свойства классов Graphics, Color, Pen и SolidBrush.

1. Уметь читать коды программ, представленных в лабораторной работе.
2. Прокомментировать тексты кодов программ Пример1 и Пример 2.
3. Изучите с помощью справки MSDN методы и свойства классов Graphics, Color, Pen и SolidBrush. Создайте собственное приложение выводящий на форму рисунок, соответствующий Вашему варианту и состоящий из различных объектов (линий, многоугольников, эллипсов, прямоугольников и пр.), не закрашенных и закрашенных полностью. Используйте разные цвета и стили линий (сплошные, штриховые, штрих-пунктирные).

|  |  |
| --- | --- |
| Вариант | Рисунок |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |