Введение в GDI +. Графическая подсистема Windows

**Цель:** Научится использовать графическую подсистему Windows с помощью программных интерфейсов С#.

# 1. Теоретическая часть

GDI расшифровывается как Graphic Device Interface (интерфейс графических устройств). Этим словом обозначается подсистема Windows, предназначенная для вывода графических изображений (Windows полностью основана на использовании графики) на экран и на принтер. GDI+ — это новый набор программных интерфейсов, используемый в .NET.

В .NET предусмотрено множество пространств имен, предназначенных для вывода двумерных графических изображений. Помимо ожидаемых стандартных типов (например, для работы с цветом, со шрифтами, с пером и кистью, с изображениями) в этих пространствах имен предусмотрены типы для выполнения достаточно изощренных операций, таких как геометрические преобразования, сглаживание неровностей, подготовка палитры, поддержка вывода на принтер и многие другие. Перечень наиболее важных пространств имен для работы с графическими изображениями представлен в таблице 1

**Таблица 1.** Наиболее важные пространства имен GDI+

|  |  |
| --- | --- |
| **Пространство имен** | **Специализация** |
| System.Drawing | Важнейшее пространство имен GDI+, которое содержит основный типы для вывода графики (для работы со шрифтами, перьями, кистью и т. п.), а также исключительно важный тип Graphics. |
| System.Drawing.Drawing2D | В этом пространстве имен предусмотрены типы для выполнения более сложных операций с двумерной графикой (градиентная заливка, геометрические преобразования и т. п.) |
| System.Drawing.Imaging | Здесь определены типы, которые позволяют напрямую работатьс графическими изображениями (менять палитру, извлекать метаданные изображений, выполнять операции с метафайлами и т. п.) |
| System.Drawing.Printing | Это пространство имен определяет типы для вывода графики на принтер и взаимодействия с принтером в целом. |
| System.Drawing.Text | Это пространство имен позволяет работать с системными шрифтами. Например, тип FontColiection позволяет получать список всех установленных в системе шрифтов. |

Чтобы обеспечить возможность работы с графикой в приложении, необходимо добавить в него ссылку на сборку System.Drawing.dll. Если воспользоваться шаблоном Windows Application, то эта ссылка будет добавлена автоматически. После этого необходимо добавить в список используемых пространств имен строку

using System.Drawing;

и можно приступать к работе с графикой.

## 1.1 Пространство имен System.Drawing

Подавляющее большинство типов для работы с графикой, которые потребуются, находятся именно в пространстве имен System.Drawing. Некоторые наиболее важные типы этого пространства имен представлены в табл. 2.

**Таблица 2.** Типы пространства имен System.Drawing

|  |  |
| --- | --- |
| **Тип** | **Назначение** |
| Bitmap | Инкапсулирует файл изображения и определяет набор методов  для выполнения различных операций с этим изображением. |
| Brush  Brushes  SolidBrush  SystemBrushes  TextureBrash | Объекты Brush (кисть) используются для заполнения пространства внутри геометрических фигур (например, прямоугольников, эллипсов или многоугольников). Тип Brush — это абстрактный базовый класс, остальные типы являются производными от Brush и определяют разные наборы возможностей. Дополнительные типы Brush определены в пространстве имен System. Drawing. Drawing2D |
| Color  SystemColors  ColorTranslator | Структура Color определяет набор статических полей, которые могут быть использованы для настройки цвета. Тип ColorTranslator позволяет создавать новый цвет (объект Color) на основе внешнего представления этого цвета (цвета Win32, типа OLE\_COLOR, констант цвета в HTML и т. п.) |
| Font  FontFamiy | Тип Font инкапсулирует характеристики шрифта (имя, размер, начертание и т. п.). FontFamily представляет набор шрифтов, которые относятся к одному семейству, но имеют некоторые небольшие отличия. |
| Graphics | Этот важнейший класс определяет набор методов для вывода текста, изображений и геометрических фигур. Можно считать этот тип эквивалентом типа HDC в Win32 |
| Icon  SystemIcons | Эти классы предназначены для работы с пользовательскими и системными значками |
| Image  ImageAnimator | Image ~ это абстрактный базовый класс, который обеспечивает возможности типов Bitmap, Icon и Cursor. ImageAnimator позволяет производить показ изображений (типов, производных от Image) через указанные вами интервалы времени |
| Pen  Pens  SystemPens | Pen (перо) — это класс, при помощи которого можно рисовать прямые и кривые линии. В классе Реп определен набор статических свойств, при помощи которых можно получить объект Реп с заданными свойствами (например, с установленным цветом) |
| Point  PointF | Эти структуры обеспечивают работу с координатами точки. Point работает со значениями типа int, a PointF — со значениями типа float |

**Таблица 2.** Типы пространства имен System.Drawing (продожение)

|  |  |
| --- | --- |
| Rectangle  RectangleF | Эти структуры предназначены для работы с прямоугольными областями (int/noat) |
| Size  SizeF | Эти структуры обеспечивают работу с размерами (высотой и шириной).  Size использует значения типа int, a SizeF — типа float. |
| StringFormat | Этот тип используется для форматирования текста (определения выравнивания, междустрочного интервала и т. п.) |
| Region | Определяет область, занятую геометрической фигурой. |

## 1.2 Система координат по умолчанию в GDI+

Перед тем как начать знакомиться с тонкостями вывода графических объектов, необходимо разобраться в используемой в GDI+ системе координат. Как и в API Win32, в GDI+ можно выбирать из множества систем координат. Система, принятая по умолчанию, использует в качестве единицы измерения пикселы, а в качестве исходной точки — верхний левый угол. Координата X определяет смещение вправо, а координата Y — смещение вниз (рис. 1)

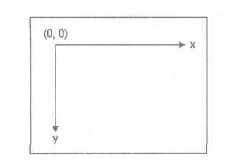


Рис. 1 Система координат, используемая по умолчанию

При помощи свойства PageUnit объекта Graphics мы можем выбрать другую единицу измерения, которая будет применяться к этому объекту. Для свойства PageUnit используются значения из перечисления GraphicsUnit (табл. 3).

**Таблица 3** Перечисление GraphicsUnit

|  |  |
| --- | --- |
| **Значение** | **Единица измерения** |
| Display | 1/75 часть дюйма |
| Document | 1/300 часть дюйма |
| Inch | Дюйм |
| Millimeter | Миллиметр |
| Pixel | Пиксель |
| Point | 1/72 часть дюйма |

В GDI+ для изменения точки отсчета системы координат используется метод TranslateTransform(), определенный в классе Graphics. Например, установить точку отсчета в положение 100.100 относительно системы координат по умолчанию можно следующим образом:

protected override void OnPaint(PaintEventArgs e)

{

e.Graphics.TranslateTransform(100, 100);

}

## 1.3 Работа с цветом

При использовании многих методов вывода, которые определены в классе Graphics, необходимо указать используемый цвет. Как правило, для этого используется структура Color. Эта структура позволяет задать цвет в системе ARGB (aplha-red-green-blue, альфа-канал (отвечающий за прозрачность) – красный – зеленый – синий). Чаще всего для выбора цвета используются статические свойства этой структуры, которые возвращают объект типа Color:

// Один из множества предопределенных цветов

Color c = Color.PapayaWhip;

Как видно из табл. 4, существуют и другие способы, при помощи которых мы можем создать объект Color. Вне зависимости от того, какой из них исспользуется, методы структуры Color позволят получить полную информацию о выбранном цвете (табл. 4)

**Таблица 4** Методы структуры Color

|  |  |
| --- | --- |
| **Метод** | **Назначение** |
| FromArgb() | Возвращает объект типа Color. Для этого метода указываются числовые значения прозрачности, красного, зеленого и синего цветов. |
| FromKnownColor() | Возвращает объект типа Color. Используются значения из перечисления KnownColor. |
| FromName() | Возвращает объект типа Color. Используются строковые значения (например, Red). |
| A,R,G,B | Эти свойства возвращают значения, присвоенные параметрам  прозрачности (А), красного (R), зеленого (G) и синего (В) цветов. |
| IsNamedColor()  Name | Применяются к текущему объекту Color. Они позволяют определить, соответствует ли он какому-либо из именованных цветов (например, Red), и, если соответствует, позволяют вернуть имя цвета. |
| GetBrightness()  GetHue()  GetSaturation() | Помимо самой распространенной системы RGB существует и другая система цветовоспроизведения — HSB (Hue-Saturation-Brightness, отгенок-насыщенность-яркость). Эта методы позволяют получать для текущего объекта Color соответственно значения яркости, оттенка и насыщенности. |
| ToArgb() | Возвращает числовые значения ARGB для объекта Color. |
| ToKnownColor() | Возвращает значение из перечисления KnownColor для объекта Color. |

Также цвет удобно выбирать при помощи специального класса ColorDialog.

## 1.4 Работа со шрифтами

Главный класс, который используется для работы со шрифтами в GDI+, — это класс ***System.Drawing.Font***. Объекты этого класса представляют конкретные шрифты, установленные на компьютере. В этом классе предусмотрено множество перегруженных конструкторов, но вот два наиболее часто используемых варианта:

// Создаем объект Font, указывая имя шрифта и его размер

Font f = new Font("Times New Roman", 12);

// Создаем объект Font, указывая имя, размер и стиль

Font f2 = new Font("Verdana", 50,

FontStyle.Bold|FontStyle.Underline);

При создании ***f2*** использовались стили из перечисления ***FontStyle***. Как видно, можно задавать сразу несколько стилей одновременно. Значения из перечисления ***FontStyle*** представлены в табл. 5.

**Таблица 5.** Доступные стили из перечисления FontStyle

|  |  |
| --- | --- |
| **Член перечисления FontStyle** | **Стиль** |
| Bold | Полужирный |
| Italic | Курсив |
| Regular | Обычный текст |
| Strikeout | Зачеркнутый |
| Underline | Подчеркнутый |

После того как были настроили необходимые параметры объекта Font, следующая задача — передать их методу ***Graphics.DrawString().*** Несмотря на то, что этот метод многократно перегружен, как правило, приходится указывать стандартный набор информации: текстовую строку, которая будет выводиться, используемые шрифт и кисть (толщина линии), а также область вывода. Например:

// public void DrawStr1ng(Str1ng, Font, Brush, Point);

g.DrawString("My string", new Font("Pop", 25),

new SolidBrush(Color.Black), new Point(0.0));

// public void DrawStringtString, Font, Brush, float, float);

g.DrawString("Another string", new Font("Times New Roman", 16),

new SolidBrush(Color.Red), 40, 40);

В обоих примерах использовался тип ***SolidBrush*** (с конкретным цветом). Вполне возможно одновременно использовать несколько типов кистей. В этой ситуации вполне подходит сплошная кисть (solid brush), более экзотические разновидности логических кистей будут рассмотрены ниже.

После того как объект ***Font*** создан, можно использовать его многочисленные свойства, такие как Bold, Italic, Unit, Height, Size, FontFamily и многие другие.

## 1.5 Работа с перьями

Обычное применение объектов ***Реn*** (перьев) заключается в рисовании линий. Как правило, объект ***Реn*** используется не сам по себе: он передается в качестве параметра многочисленным методам вывода, определенным в классе ***Graphics***. Как правило, названия всех этих методов, использующих ***Pen***, начинаются с Draw (например, ***DrawArc()***, ***DrawCurve()***, ***DrawLine()*** и т.д.) В классе Pen предусмотрено несколько перегруженных конструкторов, при помощи которых можно задать исходный цвет и толщину пера (объект ***Реn*** можно также создать на основе существующего объекта ***Brush***). Большая часть возможностей ***Реn*** определяется свойствами этого класса. Перечень наиболее важных свойств представлен в табл. 6.

**Таблица 6.** Свойства класса Реn

|  |  |
| --- | --- |
| Свойство | Назначение |
| Brush | Определяет кисть, используемую данным объектом Реn |
| Color | Определяет цвет создаваемых объектом Реn линий |
| CompoundObject | Позволяет получить или создать массив пользовательских вариантов штрихов и пустого пространства между штрихами |
| CustomStartCup  CustomEndCup | Позволяют получить или установить стиль “наконечника” пера,  который будет показан в начале линии (StartCap) и в конце линии (EndCap) |
| DashCap | Позволяет получить или установить стиль “наконечника” для перьев, рисующих пунктирные линии |
| DashOffset | Устанавливает смещение начала пунктира относительно исходной точки пунктирной линии |
| DashStyle | Позволяет получить или установить массив штрихов и пробелов между ними для пунктирных линий |
| DashPattern | Позволяет получить или установить стиль для пунктирных линий, создаваемых при помощи данного объекта Реn |
| LineJoin | Позволяет получить или установить стиль объединения при пересечении двух линий, выводимых данным объектом Реn |
| PenType | Позволяет получить стиль линий, выводимых при помощи данного объекта Реn |
| StartCup  EndCup | Позволяет получить или установить один из заранее готовых стилей “наконечника” пера. Используются значения из перечисления LineCap, определенного в пространстве имен System.Drawlng.Drawing2D |
| Width | Позволяет получить или установить ширину данного пера |

## 1.6 Работа с кистью

Кисти предназначены для закрашивания пространства между линиями. Можно определить для кисти цвет, текстуру или даже изображение. Сам класс ***Brush*** является абстрактным, и создавать объекты этого класса нельзя. Вместо этого в существуют классы, производные от ***Brush***, такие как ***SolidBrush***, ***HatchBrush***, ***LinearGradientBrush*** и т. п. Кроме того, создавать объекты кистей (выбрав из заранее готового набора) можно при помощи типов-коллекций ***Brushes*** и ***System.Brushes***, также определенных в пространстве имен ***System.Drawing***. Создание объектов из этих типов-коллекции производится при помощи их статических свойств. Далее можно передать созданный объект кисти в качестве параметра соответствующему методу объекта ***Graphics***. Выбранную кисть можно использовать для создания объекта Pen (перо). Это перо сможет рисовать линии, используя все возможности кисти (например, линия будет покрыта текстурами или изображением).

## 1.7 Вывод изображений

Осталось рассмотреть последний из основных типов: тип ***System.Drawing.Imagе***, который используется для вывода изображений. Класс ***Image*** определяет множество свойств и методов, которые можно использовать для настройки параметров выводимого изображения.

К примеру, при помощи свойств *Width*, *Height* и *Size* можно получить или установить размеры изображения. Кроме того, в пространстве имен ***System.Drawing.Imaging*** определено множество типов для проведения сложных преобразований изображений.

Класс ***Image*** является абстрактным, и создавать объекты этого класса нельзя, обычно объявленные переменные ***Image*** присваиваются объектам класса ***Bitmap***. Кроме того, мы можем создавать объекты класса ***Bitmap*** напрямую. Он поддерживает файлы с расширением \*.bmp, \*.png, \*.gif, а также файлы TIFF.

# 2. Практическая часть

## 2.1 Сеансы вывода графики

Для вывода графики и прорисовки содержимого окна в классе ***Control*** определен виртуальный метод ***OnPaint()***. Если нужно, чтобы на главной форме приложения (или элементе управления, или любом другом классе, производном от ***Control***) выводилась графика, потребуется заместить этот метод и извлечь объект ***Graphics*** из передаваемых этому методу параметров. Для более полного понимания создайте проект C# Windows Application. В файле *Form1.cs* проведите следующие изменения:

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

CenterToScreen();

this.Text = "Basic Paint Form";

}

protected override void OnPaint(PaintEventArgs e)

{

//получение объекта Graphics

Graphics g = e.Graphics;

g.DrawString("HelloGDI+", new Font("Times New Roman", 20),

new SolidBrush(Color.Black), 0, 0);

}

Откомпилируйте и запустите приложение. Вы получите форму вида:



**Рис. 2** Простейший пример работы метода OnPaint()

Для вывода изображения на форму есть два способа. Первый (который приведен выше) заключается в том, что напрямую дописывается необходимая логику в метод ***OnPaint()***. Второй (более предпочтительный) предусматривает перехват события Paint. Для реализации второго способа содержимое файла *Form1.cs* можно изменить следующим образом:

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

CenterToScreen();

this.Text = "Basic Paint Form";

//Добавляем обработчик события

//Обратите внимание на сигнатуру обработчика!!

public void Form1\_Paint(Object sender, PaintEventArgs e)

{

Graphics g = e.Graphics;

g.DrawString("HelloGDI+", new Font("Times New Roman", 20),

new SolidBrush(Color.Black), 0, 0);

}

}

Для создания обработчика события *Paint* можно воспользоватся стандартными средствами Visual Studio. Для этого выберите форму в дизайнере форм и в перечне событий для нее найдите событие Paint. Создайте обработчик двойным кликом по надписи.

Вне зависимости от того, какой метод выбирается, механизм вывода графики будет одним и тем же. Как только наше приложение станет «грязным», в очереди сообщений для этого приложения появится специальное сообщение, инициирующее сеанс вывода графики (перерисовки). Приложение становится «грязным» тогда, когда изменяются его размеры, когда его окно полностью или частично перекрывается окном другого приложения, когда оно восстанавливается из минимизированного состояния. В результате либо вызывается метод ***OnPaint()***, либо срабатывает обработчик события Paint.

## 2.2 Вывод графических объектов без события Paint

Очень часто бывает так, что вывод графического объекта необходимо произвести не в стандартных ситуациях, когда возникает событие Paint, а в ответ на другие события. Создадим приложение, задача которого — вывести маленький кружок в том месте, где на форме был сделан щелчок мышью. Первая наша задача, как всегда при выводе изображений, — получить объект ***Graphics***, а затем выполнить с этим объектом необходимые манипуляции. Объект ***Graphics*** можно получить при помощи метода ***Graphics.FormHwnd().*** Обратите внимание, что единственный параметр, передаваемый этому методу, — это значение свойства *Handle*. Свойство *Handle*, определено в классе ***Control*** и наследуется всеми классами, производными от ***Control***. Работа с объектом Graphics может выглядеть в примере так:

private void Form1\_MouseDown(object sender, MouseEventArgs e)

{

// Получаем объект Graphics

Graphics g = Graphics.FromHwnd(this.Handle);

// Теперь в месте щелчка рисуем кружок диаметром 10 пикселов

g.DrawEllipse(new Pen(Color.Green), e.X, e.Y, 10, 10);

}

Легко убедиться на практике, что при любой перерисовке формы (возникновении события Paint) все кружки, выведенные на форме после щелчков мышью, исчезнут. Это не удивительно, поскольку, как мы видим, вывод графических объектов (кружков) в нашем случае происходит только в результате щелчка мышью, но не при выводе формы на экран.

Скорее всего, в реальном приложении потребуется не только выводить какие-либо графические объекты на форму, но и сохранять их при перерисовке формы — обычно не нужно, чтобы они пропадали, например, при перекрытии формой окном другого приложения. Поэтому необходимо позаботиться о том, чтобы информация о графических объектах каким-то образом сохранялась и использовалась при перерисовке формы. Самый простой способ — создать внутреннюю коллекцию, например, при помощи класса ***ArrayLis***t (для его использования необходимо подключить *System.Collections*) и помещать туда нужные объекты. Затем к этой коллекции будет обращаться метод ***OnPaint()***. В нашей ситуации проще всего использовать коллекцию объектов Point, поскольку для рисования кружков ничего, кроме координат их центра, не нужно:

public partial class Form1 : Form

{

// Коллекция для хранения координат всех кружков

private ArrayList myPts = new ArrayList();

… … …

private void Form1\_MouseDown(object sender, MouseEventArgs e)

{

// Получаем объект Graphics

Graphics g = Graphics.FromHwnd(this.Handle);

// Раньше мы рисовали кружки по щелчку иыши:

//g.DrawEllipse(new Pen(Color.Green),е.Х,е.У,10, 10);

//Теперь добавляем координаты указателя при щелчке

// в коллекцию и вызываем Invalidate()

myPts.Add(new Point(e.X,e.Y));

Invalidate();

}

public void Form1\_Paint(Object sender, PaintEventArgs e)

{

Graphics g = e.Graphics;

g.DrawString("Hello GDI+", new Font("Times New Roman", 20), new SolidBrush(Color.Black), 0, 0);

// Выводим все точки

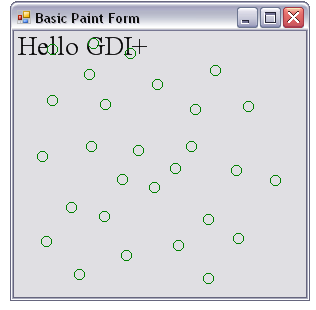
foreach(Point p in myPts)

g.DrawEllipse(new Pen(Color.Green),p.X,p.Y,10,10);

}

}

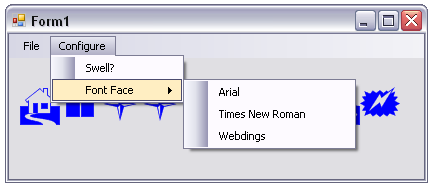
Таким образом, использование метода G***raphics.FromHwnd()*** — это очень удобный способ получить объект ***Graphics*** без использования события Paint. Результат работы программы представлен на рис. 3.



**Рис. 3** Результат работы программы

## 2.3 Создание приложения с возможностью выбора шрифта

Чтобы лучше освоить работу со шрифтами, создадим приложение, которое позволит пользователю выбирать нужный ему тип шрифта при помощи меню Configure (Настроить) -> Font Face (Гарнитура шрифта). То, что должно получиться, представлено на рис. 4.



**Рис. 4** Внешний вид создаваемого приложения

Кроме того, чтобы было интереснее, мы разрешим пользователю косвенно управлять размером шрифта. Пусть при выборе пользователем пункта меню Configure(Настроить) > Swell? (Разбухать?) шрифт начнет увеличиваться скачками через равные промежутки времени (при помощи объекта Timer), пока не достигнет установленного нами верхнего предела.

Первое, что нужно сделать, это объявить глобальные переменный, определяющие начальный шрифт и его размер. Также необходимо создать таймер:

public partial class Form1 : Form

{

private int swellValue;

// Этот шрифт будет использоваться по умолчанию:

private string fontFace = "Webdings";

public Form1()

{

InitializeComponent();

}

Далее создайте обработчик таймера двойным кликом по событию *Tick*. Обработчик события *Tick* должен увеличивать значение переменной ***swellValue*** и перерисовывать клиентскую область нашей формы. Чтобы шрифт при разбухании не превысил все разумные размеры, мы установим для ***swellValue*** максимум, равный 50. И еще один момент: чтобы уменьшить мерцание, связанное с перерисовкой всей клиентской части формы, будем обновлять только определенный «грязный» прямоугольник на форме:

private void timer1\_Tick(object sender, EventArgs e)

{

// При каждом "тике" размер шрифта увеличивается на 5

swellValue += 5;

// При достижении максимального размера уменьшаем его до 0

if (swellValue >= 50)

swellValue = 0;

// Перерисовываем нужною нам прямоугольную область формы

Invalidate(new Rectangle(0,0,ClientRectangle.Width, 120));

}

Теперь каждые 100 миллисекунд значение ***swellValue*** будет изменяться, а верхние 120 пикселов клиентской части формы – перерисовываться. Осталось связать ***swellValue*** и размер шрифта, а также обеспечить вывод текстовой строки на форму. Выглядеть это может так:

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

{

Graphics g = e.Graphics;

// Размер шрифта будет меняться между 12 и 62 в зависимости

// от текущего значения swell Value

Font theFont = new Font(fontFace, 30 + swellValue);

string message = "Hello GDI+";

// Выводим сообщение по центру формы

float windowCenter = this.DisplayRectangle.Width/2;

SizeF stringSize = g.MeasureString(message, theFont);

float startPos = windowCenter - (stringSize.Width/2);

g.DrawString(message, theFont, new SolidBrush(Color.Blue), startPos, 20);

}

Последнее, что нужно сделать, — настроить включение и отключение разбухания, шрифта при выборе пользователем пункта меню Configure (Настроить) -*>*Swell? (Разбухать?). Для этого необходимо настроить обработчик *swellToolStripMenuItem\_Click* для включения и отключения объекта ***Timer***:

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

{

if (timer1.Enabled)

{

timer1.Enabled = false;

swellToolStripMenuItem.Checked = false;

}

else

{

timer1.Enabled = true;

swellToolStripMenuItem.Checked = true;

}

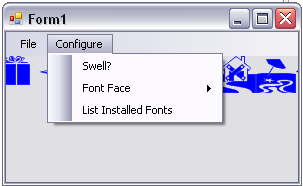
}

Смену шрифта отображаемого текста сделайте самостоятельно.

## 2.4 Вывод информации об установленных шрифтах

Давайте наделим приложение FontApp дополнительными возможностями. Пусть это приложение будет программным образом получать информацию об установленных в системе шрифтах и выводить эту информацию на форму. Для этой цели нам потребуются типы из еще одного пространства имен GDI+ — *System.Drawlng.Text*.

Вывод списка установленных в системе шрифтов будет производиться при выборе пользователем пункта меню Configure (Настроить) -> List Installed Fonts (Вывести установленные шрифты) — рис. 5.



**Рис 5** Вид создаваемого приложения

Механизм работы будет таким: при выборе пользователем этого пункта меню активируется соответствующий обработчик события (*listInstalledFontsToolStripMenuItem\_Click()*). Первое, что делает этот метод — создает объект класса ***InstalledFontCollection***. Этот класс содержит массив ***FontFamily***, представляющий набор всех шрифтов, установленных на данном компьютере. Получить доступ к объектам этого массива можно при помощи свойства ***InstalledFontCollection.Families***. Последнее, что нам осталось сделать,— извлечь имя каждой гарнитуры шрифта при помощи свойства *FontFamily.Name*. Полученные значения записываются в переменную типа ***string***, которую назовем ***installedFonts***:

private string installedFonts;

...

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

{

InstalledFontCollection fonts = new InstalledFontCollection();

for(int i=0; i < fonts.Families.Length; i++)

installedFonts += fonts.Families[i].Name;

// На этот раз потребуется перерисовать всю клиентскую область формы,

// поскольку вывод полученных значений будет производиться в

// нижнюю часть формы

Invalidate();

}

Последнее, что мы должны сделать, — вывести значение переменной ***installedFonts*** на форму. При этом нам необходимо учесть, что в верхнюю часть формы уже производится вывод «разбухающей» строки Hello GDI+, и сделать так, чтобы выводимые строки не мешали друг другу:

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

{

Graphics g = e.Graphics;

Font theFont = new Font(fontFace, 30 + swellValue);

string message = "Hello GDI+";

// Выводим сообщение по центру формы

float windowCenter = this.DisplayRectangle.Width/2;

SizeF stringSize = g.MeasureString(message, theFont);

float startPos = windowCenter - (stringSize.Width/2);

g.DrawString(message, theFont, new SolidBrush(Color.Blue), startPos, 20);

// Выводим информацию об установленных шрифтах под занятой областью

Rectangle myRect = new Rectangle(0, 120, ClientRectangle.Width,ClientRectangle.Height);

// Будем рисовать в этой области белый по черному

g.FillRectangle(new SolidBrush(Color.Black), myRect);

g.DrawString(installedFonts, new Font("Arial", 12), new SolidBrush(Color.White),myRect);

}

И для порядка сделаем так, чтобы при возникновении события Resize (то есть при изменении размера формы пользователем) нижняя часть формы перерисовывалась правильно:

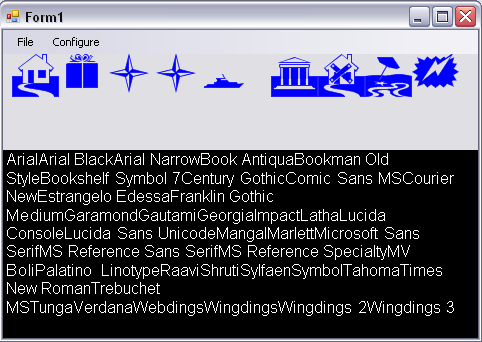
private void FontForm\_Resize(object sender, EventArgs e)

{

Invalidate();

}

Думаю, этот код в пояснениях не нуждается. Все, программа готова. Запустите ее. Вы должны получить нечто, похожее на это:



**Рис. 6.** Внешний вид готового приложения

## 2.5 Работа с перьями

Создадим приложение, иллюстрирующее работу с перьями. Для этого создайте Windows Application и назовите его PenDemo, установите необходимые свойства формы (имя, размеры (300х370) и т.д.). Вывод графических изображений при помощи класса Реn на нее будет производиться следующим образом:

using System.Drawing.Drawing2D;

... ...

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

{

Graphics g = e.Graphics;

// Создаем большое перо синего цвета

Pen bluePen = new Pen(Color.Blue, 20);

// Создаем еще одно перо при помощи заготовок из коллекции Pens

Pen pen2 = Pens.Firebrick;

//Выводим при помощи созданных перьев геометрические фигуры

g.DrawEllipse(bluePen, 10, 10, 100, 100);

g.DrawLine(pen2, 10, 130, 110, 130);

g.DrawPie(Pens.Black, 150, 10, 120, 150, 90, 80);

// Выводим многоугольник пурпурного цвета

Pen pen3 = new Pen(Color.Purple, 5);

pen3.DashStyle = DashStyle.DashDotDot;

g.DrawPolygon(pen3, new Point[] {

new Point(30, 140),

new Point(265, 200),

new Point(100, 225),

new Point(190, 190),

new Point(80, 330),

new Point(20, 180) });

// Добавляем прямоугольник со вписанным нами текстои

Rectangle r = new Rectangle(150, 10, 130, 60);

g.DrawRectangle(Pens.Blue, r);

g.DrawString("Hello out there...\n How are you?",

new Font("Arial", 12), Brushes.Black, r);

//Выводим пунктирную линию собственного стиля по периметру формы

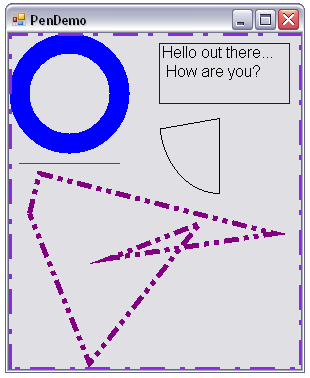
Pen customDashPen = new Pen(Color.BlueViolet, 5);

float[] myDashes = {5.0f, 2.0f, 1.0f, 3.0f};

customDashPen.DashPattern = myDashes;

g.DrawRectangle(customDashPen, ClientRectangle);

}



**Рис. 7** Внешний вод готового прилодения

## 2.6 Работа с «наконечниками» перьев

При рассмотрении предыдущей формы можно обнаружить, что концы всех нарисованных на ней линий обрезаны стандартным образом — под прямым углом. Если необходимы более изысканные окончания линий, в нашем распоряжении — перечисление LineCap.

В качестве примера создадим приложение, в котором при помощи класса Реn будет нарисовано множество линий с разными окончаниями. Программа выведет для каждого из значений перечисления LineCap его имя, а затем нарисует рядом линию с использованием этого значения:

Для этого снова создадим Windows Application и назовем его PenDemoCap. Добавим в него следующий код:

using System.Drawing.Drawing2D;

... ...

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

{

Graphics g = e.Graphics;

Pen thePen = new Pen(Color.Black, 10);

int yOffSet = 10;

// Получаем все значения перечисления LineCap

Array obj = Enum.GetValues(typeof(LineCap));

// Выводим линию с использованиен значения из LineCap

for(int x = 0; x < obj.Length; x++)

{

// Настраиваем "наконечник" пера

LineCap temp = (LineCap)obj.GetValue(x);

thePen.StartCap = temp;

thePen.EndCap = temp;

// Выводим имя значения перечисления

g.DrawString(temp.ToString(),new Font("Times New Roman",10),

new SolidBrush(Color.Black), 0, yOffSet);

// Выводим линию с выбранным наконечником

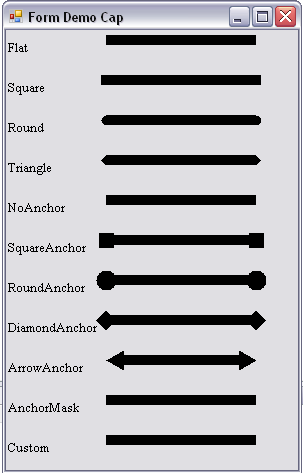
g.DrawLine(thePen, 100, yOffSet, this.Width-50, yOffSet);

yOffSet += 40;

}

}

В результате должно получиться:



**Рис. 8** Внешний вид приложения

## 2.7 Работа с кистью

В качестве примера приведем программу, которая будет использовать типы ***SolidBrush*** и ***Brushes*** для закрашивания пространства внутри геометрических фигур. Также будет показан пример использования градиентных, штрихованных и текстурных кистей. Для этого создайте Windows Application и назовите его BrushesDemo. Код обработки события Paint должен содержать такой код:

using System.Drawing.Drawing2D;

... ...

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

{

Graphics g = e.Graphics;

// Создаем "плотную кисть" - SolidBrush синего цвета

SolidBrush blueBrush = new SolidBrush(Color.Blue);

g.FillEllipse(blueBrush, 40, 40, 100, 100);

// Создаем еще одну кисть при помощи заранее готовой

коллекции Brushes

g.FillPie(Brushes.Aqua, 150, 10, 120, 150, 90, 80);

// Создаем штрихованую кисть и закрашиваем ею полигон

HatchBrush brush2 = new

HatchBrush(HatchStyle.DashedUpwardDiagonal,Color.Coral);

g.FillPolygon(brush2, new Point[] {

new Point(30, 140),

new Point(265, 200),

new Point(100, 225),

new Point(190, 190),

new Point(80, 330),

new Point(20, 180) });

// Добавляем прямоугольник со вписанным нами текстои

Rectangle r = new Rectangle(150, 10, 130, 60);

// Заполняем его градиентной кистью

Brush grBrush = new LinearGradientBrush(new Rectangle(10,

60, 50, 50), Color.BurlyWood, Color.BlueViolet, 45);

g.FillRectangle(grBrush, r);

g.DrawString("Hello out there...\n How are you?",

new Font("Arial", 12), Brushes.White, r);

//Загружаем некоторое изображение

Bitmap BrushImage = new Bitmap("Bitmap1.bmp");

//Создаем текстурную кисть с выбранным изображением

Brush TexturedBrush = new TextureBrush(BrushImage);

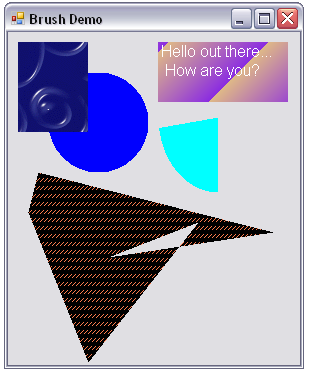
//Закрашиваем ею прямоугольник

g.FillRectangle(TexturedBrush, new Rectangle(10,10,70,90));

}

Для правильной работы данного приложения необходимо добавить к проекту какую-нибудь картинку \*.bmp, назвать ее Bitmap1.bmp и скопировать в папку …\bin\Debug или изменить параметр конструктора для ***BrushImage,*** указав реальный путь к картинке.

Готовое приложение должно выглядеть так:



***Рис.9***  Внешний вид готового приложения

## 2.8 Вывод изображений

Напишем приложение, в котором выведем три изображения на форму. Для этого создайте Windows Application. В нем необходимо объявить три переменные Image, а затем использовать для каждой из них объекты Вitmap:

private Image bMapImageA;

private Image bMapImageB;

private Image bMapImageC;

public Form1()

{

InitializeComponent();

bMapImageA = new Bitmap("ImageA.bmp");

bMapImageB = new Bitmap("ImageB.bmp");

bMaplmageC = new Bitmap("ImageC.bmp");

}

Файлы картинок должны находится в катологе \bin\Debug пректа. Вывод полученных изображений производится очень просто, поскольку в классе Graphics предусмотрен специальный метод, который так и называется — ***DrawImage().*** Этот метод многократно перегружен, поэтому в нашем распоряжении множество вариантов того, как поместить изображение в нужное нам место на форме. Кроме того, для настройки параметров выводимого изображения можно использовать с этим методом значения перечислений ***ImageAttributes*** и ***GraphicsUnit.*** В нашей ситуации нужно всего лишь указать место, на Koiqpoe будут помещены изображения, чтобы они не перекрывали друг друга. Координаты можно указать при помощи объектов Point (точка), Rectangle (прямоугольник), целочисленными значениями или значениями с плавающей запятой. В нашем случае решение может быть таким:

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

{

Graphics g = e.Graphics;

// Выводим изображения

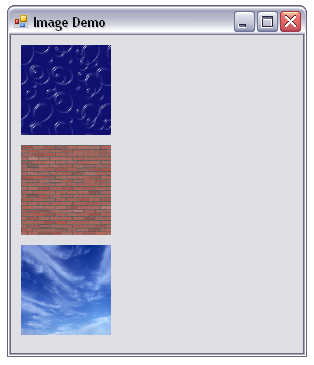
g.DrawImage(bMapImageA, 10, 10, 90, 90);

g.DrawImage(bMapImageB, 10, 110, 90, 90);

g.DrawImage(bMapImageC, 10, 210, 90, 90);

}

Откомпилируйте и запустите прилодение. Вы должны получить нечто, похожее на это:



***Рис. 10*** Внешний вид готового приложения

## 2.9 Графический редактор

Теперь применим все полученные знания и создадим программу «Графический редактор». Создаваемая программа будет иметь основные базовые возможности, присущие любому графическому редактору – это рисование произвольной линии (карандаш), рисование прямой линии, вывод текста и рисование эллипса. Для начала создайте Windows Application, назовите проект «GraphEditorApp». Задайте имя для формы, ее размеры и т.д. Параметр BackColor установите в White для того, чтобы фон окна был белым. Начнем с того, что создадим меню для этого приложения. Оно должно иметь вид:

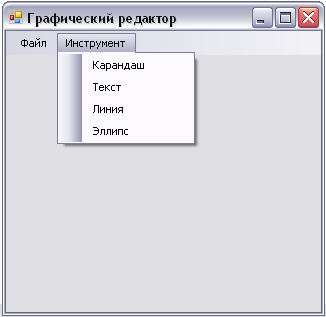


Рис 10 Меню для приложения «Графический редактор»

Измените имена пунктов меню на ToolsMenuItem, PenMenuItem, TextMenuItem, LineMenuItem, EllipseMenuItem соответственно. Установите свойство Checked пункта меню PenMenuItem в true. Этот инструмент будет выбран по умолчанию.

Далее добавьте в программу перечисление (enum), которое будет содержать режимы работы программы (инструменты). Для этого выше описания класса ***GraphApp*** добавьте описание перечисления и объявите его.

public enum Tools

{

PEN = 1, TEXT, LINE, ELLIPSE, NULL = 0

}

private Tools curentTool=Tools.PEN;

Далее создаем обработчики событий для каждого пункта меню:

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

{

//Циклом сбрасываем все флажки

foreach (ToolStripMenuItem menu in

ToolsMenuItem.DropDownItems)

menu.Checked = false;

//Устанавливаем текущий пункт меню в Checked

PenMenuItem.Checked = true;

//Устанавливаем как текущий инструмент

curentTool = Tools.PEN;

}

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

{

foreach (ToolStripMenuItem menu in

ToolsMenuItem.DropDownItems)

menu.Checked = false;

TextMenuItem.Checked = true;

curentTool = Tools.TEXT;

}

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

{

foreach (ToolStripMenuItem menu in

ToolsMenuItem.DropDownItems)

menu.Checked = false;

LineMenuItem.Checked = true;

curentTool = Tools.LINE;

}

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

{

foreach (ToolStripMenuItem menu in

ToolsMenuItem.DropDownItems)

menu.Checked = false;

EllipseMenuItem.Checked = true;

curentTool = Tools.ELLIPSE;

}

Таким образом мы сделали так, чтобы при выборе какого-нибудь инструмента, соответствующий пункт меню был тоже выбран.

Далее необходимо обрабатывать сообщения от мыши при щелчке ею в области окна. Для этого создаем обработчик и добавляем в него следующий код:

private void GraphApp\_MouseDown(object sender, MouseEventArgs e)

{

switch (curentTool)

{

case Tools.LINE:

//DrawLine(new Point (e.X, e.Y));

break;

case Tools.ELLIPSE:

//DrawEllipse(new Point (e.X, e.Y));

break;

case Tools.TEXT:

//DrawText(new Point(e.X, e.Y));

break;

case Tools.PEN:

// устанавливаем флаг для рисования карандашом

drawPen = true;

break;

// запоминаем первую точку для рисования

}

PreviousPoint.X = e.X;

PreviousPoint.Y = e.Y;

}

Для работы этого кода также необходимо объявить глобальные переменные:

public bool drawPen = false;

public Point PreviousPoint;

Пока что опише работу программы в случае, если выбран инструмент карандаш, все остальное пока закоментируем. У нас уже есть код, который обрабатывает нажатие мыши, осталось добавить код, обрабатывающий ее передвижение и отпускание.

private void GraphApp\_MouseUp(object sender, MouseEventArgs e)

{

//Если отпустили курсор, не рисуем

drawPen = false;

}

private void GraphApp\_MouseMove(object sender, MouseEventArgs e)

{

// если курсов еще не отпущен

if (drawPen)

{

// создаем объект Pen

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

// получаем текущее положение курсора

Point point = new Point(e.X, e.Y);

// создаем объект Graphics

Graphics g = this.CreateGraphics();

// рисуем линию

g.DrawLine(blackPen, PreviousPoint, point);

// сохраняем текущую позицию курсора

PreviousPoint = point;

}

}

***drawPen*** — это двоичная переменная, которая будет определять, рисовать карандашу или нет. Обработчик *GraphApp\_MouseDown* выставляет значение переменной ***drawPen*** в true, если в момент нажатия кнопки мыши установлен инструмент «Карандаш». Обработчик *GraphApp\_MouseUp* выставляет значение переменной назад в false. Таким образом, при нажатии кнопки мыши флаг ***drawPen*** устанавливается в true и не сбрасывается в false до тех пор, пока пользователь не отпустит кнопку вверх. Обработчик *GraphApp\_MouseMove* рисует на экране линии только тогда, когда переменная drawPen находится в true. Рисование происходит следующим образом:

1. Создается объект ***Реn***, который инициализируется черным цветом (Color.Black) и толщиной линии, равной 3.

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

2. Получаются текущие координаты курсора мыши.

Point point = new Point(e.X, e.Y);

3. Создается объект ***Graphics*** на базе текущей формы.

Graphics g = this.CreateGraphics();

4. Вызывается метод ***DrawLine*** объекта Graphics. В качестве координат линии передаются предыдущая и текущая координаты курсора мыши. После прорисовки линии текущая координата курсора мыши запоминается в переменную, которая хранит предыдущую координату.

PreviousPoint = point;

Таким образом, каждое движение мыши с нажатой кнопкой будет заставлять программу рисовать линию между двумя соседними точками. В итоге получится кривая пути, по которому двигался курсор мыши. Запустите приложение. Выберите режим рисования карандашом. Попробуйте нарисовать в окне любую кривую линию, удерживая курсор мыши в нажатом состоянии (рис. 11). Если у вас получилось нарисовать кривую — вы все сделали правильно, если нет — внимательно перечитайте приведенное описание.

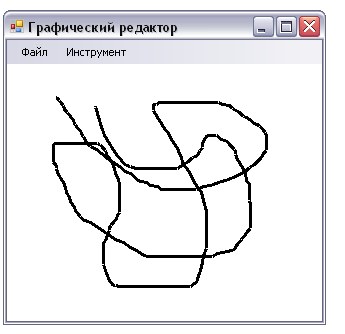


Рис 11 Рисование карандашом

Осталось реализовать функцию рисования текста, линии и эллипса. Для этого добавьте в класс GraphApp описание приведенных ниже функций:

void DrawLine(Point point)

{

// если один раз уже щелкнули

if (FirstClick == true)

{

// создаем объект Реn

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

// создаем объект Graphics

Graphics g = this.CreateGraphics();

// рисуем линию

g.DrawLine(BackPen, PreviousPoint, point);

FirstClick = false;

}

else

{

FirstClick = true;

}

}

void DrawEllipse(Point point)

{

// если один раз уже щелкнули

if (FirstClick == true)

{

// создаем объект Pen

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

// создаем объект Graphics

Graphics g = this.CreateGraphics();

// рисуем эллипс

g.DrawEllipse(BаскРеn,

PreviousPoint.X, PreviousPoint.Y,

point.X - PreviousPoint.X, point.Y - PreviousPoint.Y);

FirstClick = false;

}

else

{

FirstClick = true;

}

}

void DrawText(Point point)

{

// создаем объект Graphics

Graphics g = this.CreateGraphics();

// создаем объект Font

Font titleFont = new Font("Lucida Sans Unicode", 15);

// рисуем текст красным цветом

g.DrawString("Программирование на С#",

titleFont, new SolidBrush(Color.Red), point.X, point.Y);

}

Методы ***DrawLine*** и ***DrawEllipse*** используют переменную ***FirstClick***. Добавьте объявление этой переменной в класс, как глобальную:

public bool FirstClick = false;

Для того, чтобы эти функции исспользовались, расскоментируйте их вызовы в функции *GraphApp\_MouseDown*.

Методы ***DrawLine*** и ***DrawEllipse*** рисуют объекты по двум координатам точек на экране. При первом щелчке мыши запоминается первая координата и выставляется значение переменной ***FirstClick*** в true. При втором щелчке мыши происходит рисование линии, и значение ***FirstClick*** сбрасывается в false.

Для рисования текста используется метод ***DrawText***. Для прорисовки сначала создается объект Font со шрифтом типа Lucida Sans Unicode размером 15 единиц. Затем при помощи метода DrawString строка текста «Программирование на С#» выводится на экран. Запустите программу. Выбирая различные режимы работы программы, вы можете создавать простейшие графические изображения (рис. 12).

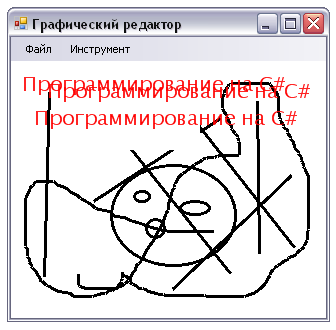


Рис. 12 готовое приложение «Графический редактор»

# 3. Домашние задания:

1. Реализуете все приложения, описанные выше.
2. Добавьте возможность выбора типа карандаша через меню (используйте тип пера и наконечники перьев)
3. Добавьте возможность именно ввода текста на форму, а не его создания, как было описано в работе.
4. Сделайте графический редактор и добавьте в него возможность заливки:
   1. просто выбранным цветом
   2. градиентной кистью
   3. текстурной кистью

**Контрольные вопросы**

1. Что такое GDI+ ?
2. Какая система координат используется в GDI+? Можно ли ее менять?
3. Что такое перо? Примеры использования пера
4. Какие виды перьев вы знаете?
5. Для чего используется кисть?
6. Какие виды кисти вы знаете?
7. Можно ли в качестве основы для кисти использоваться изображение?