Тема 9.
Программирование графики с использованием GDI+

Что такое GDI+?

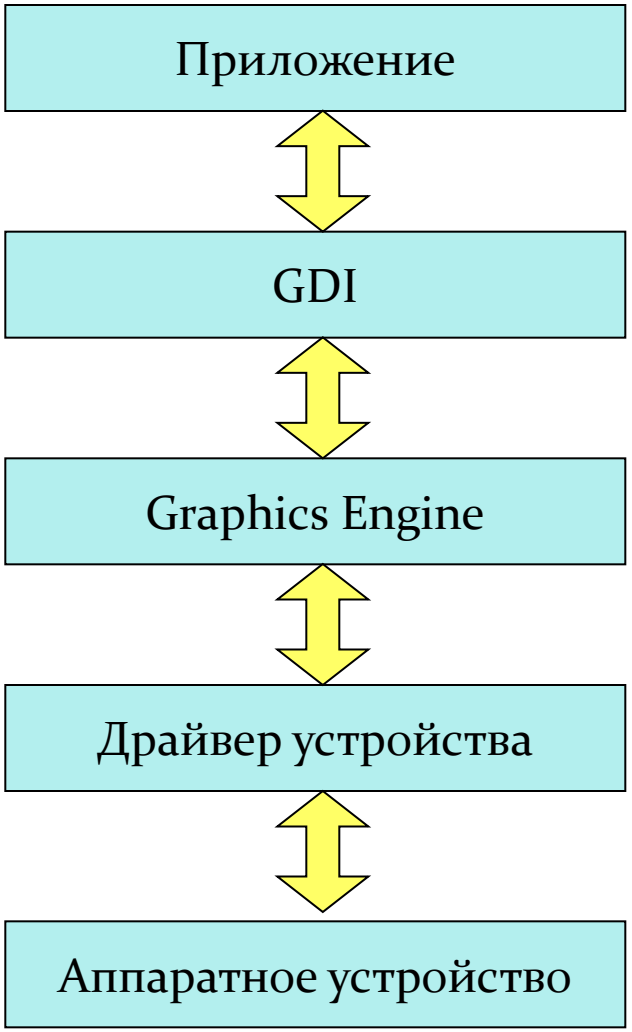
Все возможности для работы с графикой компиляторы C++ представляют в виде дополнительных библиотек графических функций.

Под Windows компиляторы используют библиотеку GDI+ (**Graphic Device Interface),** то есть графические функции API (Application Program Interface) Windows, под Mac OS библиотеку **QuickDraw,** то есть графические функции API Mac OS.

GDI+ позволяют создателям приложений выводить данные на экран или на принтер без необходимости обеспечивать работу с определенными типами устройств отображения. Для отображения информации программисту достаточно вызывать нужные методы классов GDI+. При этом автоматически учитываются типы определенных устройств и выполняются вызовы

Многоуровневая архитектура графической подсистемы

- Верхний слой – клиентские API
  - GDI, GDI+,DirectDraw, Direct3D, OpenGL, используются прикладными программами
    - Находятся в адресном пространстве приложения
- Средний слой – т.н. Graphics Engine
  - Часть ядра ОС
    - Содержит сотни функций, используемых верхним слоем
- Нижний слой – драйвер устройства
  - Осуществляет непосредственное взаимодействие с графическим устройством
    - Используется средним слоем для доступа к устройству



Три части GDI+…

- Существуют *три основных направления* применения функций GDI+:
  - *Текст*
  - *Растровая графика*
  - *Векторная графика*
- ***Текст***
  - Данная область применения охватывает задачи вывода текста на экран или принтер с настройкой отображения текста путем использования различных шрифтов, размеров и стилей. Интерфейс GDI+ предоставляет большое количество средств для поддержки данного круга задач

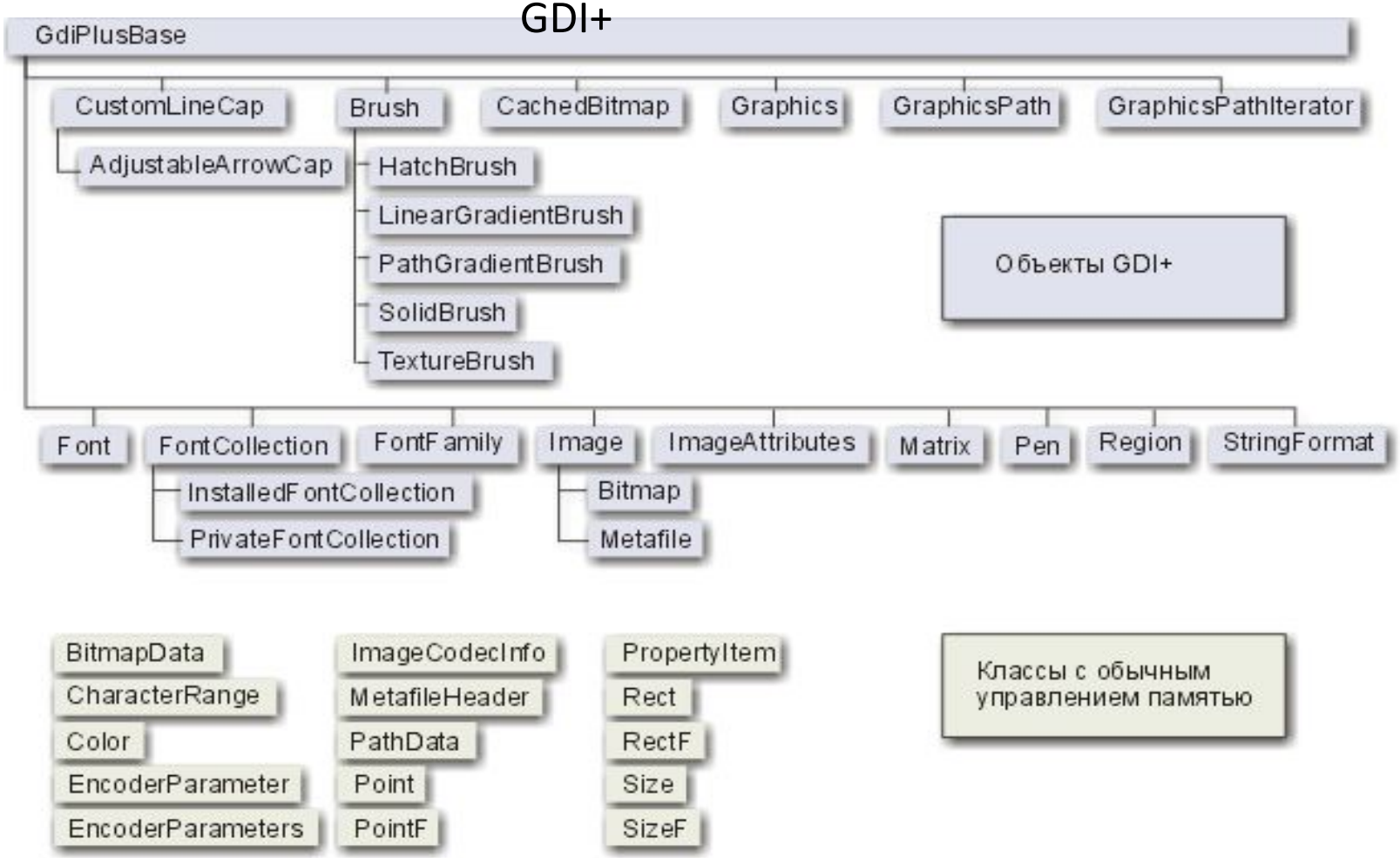
Три части GDI+…

- ***Векторная графика***
  - *Векторная графика состоит в рисовании примитивов, заданных набором точек в системе координат*. Например, прямую линию можно задать двумя крайними точками, а прямоугольник – его положением, шириной и высотой
  - GDI+ предоставляет *классы и структуры* для хранения данных *о примитивах*, *о способе их рисования* и *для их отрисовки*. Например, в структуре **`Rectangle`** хранится положение и размер прямоугольника, в классе **`Pen`** – данные о цвете, толщине и стиле линии, а класс **`Graphics`** содержит методы для вывода графики
  - *Векторный рисунок можно записать в* ***метафайл***. Интерфейс GDI+ предоставляет класс **`Metafile`**, позволяющий записывать, отображать или сохранять метафайлы

Три части GDI+

- ***Растровая графика***
  - *Некоторые рисунки сложно или невозможно отображать с использованием векторной графики*. Попробуйте сохранить с помощью векторной графики высококачественную фотографию переполненного стадиона!
  - Рисунки такого типа хранятся в виде ***точечных рисунков*** — массивов чисел, каждое из которых определяет цвет определенной точки на рисунке
  - Интерфейс GDI+ предоставляет класс **`Bitmap`**, позволяющий изменять, отображать или сохранять точечные рисунки

Иерархия классов в GDI+



Интерфейс управляемых классов GDI+ содержит около 60 классов, 50 перечислений и 8 структур.

Ключевым классом в GDI+ является **Graphics** . Именно он содержит почти две сотни методов, отвечающих за рисование, отсечение и параметры устройства вывода. Многие классы работают совместно с классом Graphics.

Плоская графика на базе GDI+

Применение интерфейса GDI+ будем рассматривать на примере программных модулей, созданных в Visual Studio на языке программирования C++ для CLR в режиме Windows Forms Application

Common Language Runtime

Изначально среда Microsoft Visual Studio для языка C++ была ориентирована на разработку Win32 приложений. С появлением технологии Net и платформы Microsoft .NET Framework for Windows, обеспечивающей поддержку этой технологии в Microsoft Windows, в Microsoft Visual Studio были интегрированы возможности, обеспечивающие разработку .NET приложений на языке C++ , а также на языках C#,VB,F#.

Технология Microsoft .NET основана на идее универсального программного кода, который может быть выполнен любым компьютером, вне зависимости от используемой операционной системы. Универсальность программного кода обеспечивается за счёт предварительной (выполняемой на этапе разработки) компиляции исходной программы в универсальный промежуточный код (CIL-код, Common Intermediate Language), который во время запуска (загрузки) программы транслируется в выполняемый.

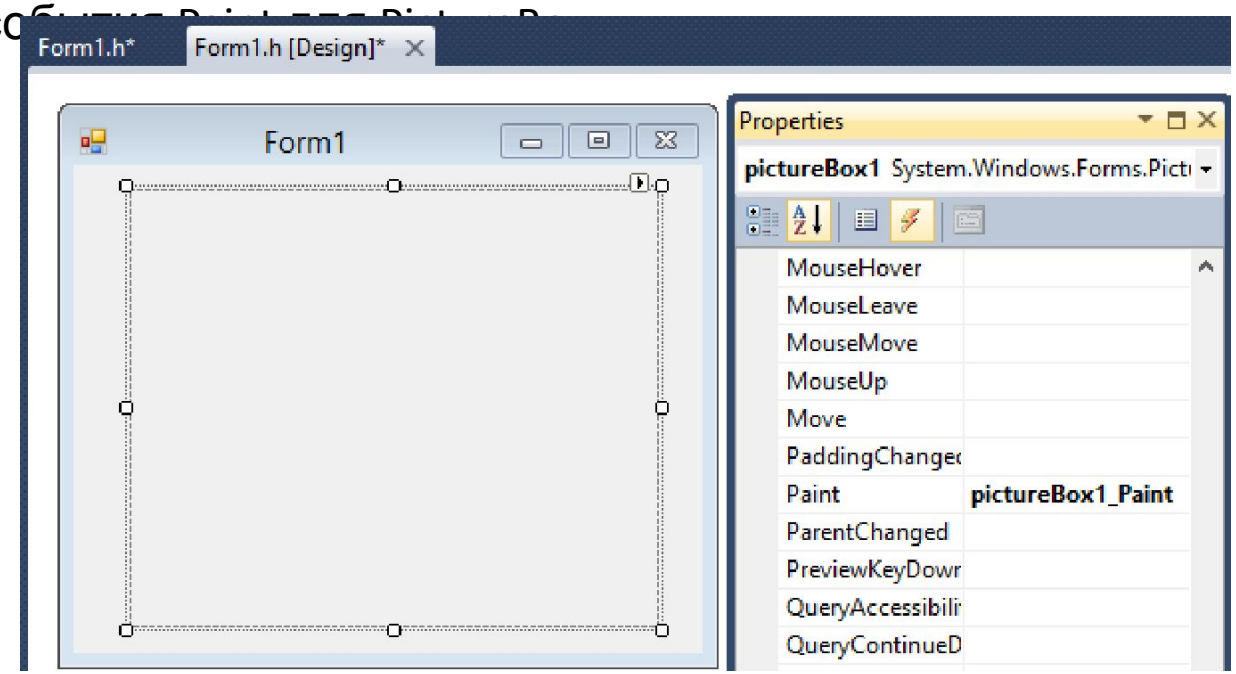
Выполнение .NET - приложений в операционной системе Microsoft Windows обеспечивает Common Language Runtime(CLR,общеязыковая исполняющая

Пространство имён GDI++ в NET.Framework

Пространство имен	Описание
System::Drawing	Классы для 2d графики, а так же основной класс Graphics
System::Drawing::Drawing2d	Расширенные возможности 2d графики, векторная графика
System::Drawing::Imaging	Классы для работы с графическими изображениями
System::Drawing::Printing	Печать
System::Drawing::Text	Работа со шрифтами

Класс/структура внутри System::Drawing	Описание
Bitmap	Базовая обработка изображений в форматах bmp, gif, jpg
Brush	Класс кисти для определения цвета и текстуры при заливке прямоугольников, эллипсов/окружностей и полигонов.
Brushes	Класс со статичным, заранее определенным набором кистей.
Color	Структура, определяющая цвет.
Font	Класс шрифта.
FontFamily	Класс для группы шрифтов с одинаковым дизайном
Graphics	Класс, представляющий собой поверхность для рисования.
Icon	Класс для стандартной иконки Windows
Image	Абстрактный класс для классов Bitmap и Icon.
Pen	Класс ручки, определяющий цвет, толщину и текстуру обводки фигур.
Pens	Класс с набором заранее определенных классов Pen.
Point, PointF	Структура, содержащая координаты (x,y) в целых числах типа Int32 или с плавающей точкой типа Single.
Rectangle, RectangleF	Структура, описывающая размер и расположение прямоугольника с помощью типов Int32 или Single.
Region	Класс, определяющий геометрическую фигуру посредством прямоугольников.
Size, SizeF	Структура, определяющая размер в числах типа Int32 или Single.
SolidBrushes	Класс-кисть для сплошного заполнения заданным цветом.
TextureBrush	Класс-кисть, использующий для заполнения заданное изображение

Рисовать векторные графические изображения можно непосредственно в клиентской части формы или на поверхности компонента PictureBox. Графическая поверхность формы или компонент PictureBox представляет собой объект Graphics. Таким образом, чтобы на графической поверхности появилась линия, прямоугольник, окружность или другой графический примитив, необходимо вызвать соответствующий метод объекта Graphics. Но доступ к графической поверхности объекта (его свойству Graphics) есть только у функции обработки события Paint. Поэтому, отобразить графику может только она. Создаём функцию обработки со




```
private: System::Void pictureBox1_Paint (System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {    }
```

Пример обработки события Paint

```
private: System::Void pictureBox1_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
 int x,y;//левый верхний угол
 int w,h;//ширина и высота полосы
 x=10;
 y=10;
 w=28;
 h=60;
 //зелёная полоса
 e->Graphics->FillRectangle(System::Drawing::Brushes::Green,x,y,w,h);
 //белая полоса
 x+=w;
 e->Graphics->FillRectangle(System::Drawing::Brushes::White,x,y,w,h);
 //красная полоса
  x+=w;
 e->Graphics->FillRectangle(System::Drawing::Brushes::Red,x,y,w,h);

 //Подпись
 //Используется шрифт, заданный свойством Font формы
 e->Graphics->DrawString("Италия",this->Font,System::Drawing::Brushes::Black,10,80);
 }
```

Графическая поверхность состоит из отдельных точек – пикселов. Левая верхняя точка графической поверхности имеет координаты (0,0). Размер графической поверхности формы соответствует размеру клиентской области формы, а размер графической поверхности компонента PictureBox – размеру компонента.

Некоторые методы рисования 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` задают 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. Параметр `Brush` определяет цвет и стиль закраски внутренней области многоугольника; параметр `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 которой 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

Карандаши и кисти. Стандартный и системный наборы.

Методы рисования графических примитивов используют *Карандаши* и *Кисти.*
Карандаш (объект Pen класса Pen) определяет вид линии, а **Кисть** (объект Brush класса Brush) - вид закраски области.

```
DrawLine(System::Drawing::Pens::Black,10,20,100,20)
FillRectangle(System::Drawing::Brushes::Green,x,y,w,h)
```

Могут использоваться карандаши и кисти из стандартного (как в примере выше) и системного набора или можно создать свой собственный карандаш и кисть.
Карандаш определяет вид линии - цвет, толщину и стиль.
Системный набор карандашей, цвет которых определяется текущей цветовой схемой операционной системы и совпадает с цветом какого-либо элемента интерфейса пользователя. Например, цвет карандаша SystemPens::WindowText совпадает с цветом текста в окне сообщений.
Карандаш из стандартного (Pens) и системного (SystemPens) наборов рисует непрерывную линию толщиной в 1 пиксел.

Во всех остальных случаях надо использовать карандаш программиста.

Карандаши и кисти. Карандаш программиста.

Карандаш программиста – объект класса Pen, свойства которого определяют вид линии, рисуемой карандашом.

Свойства объекта Pen

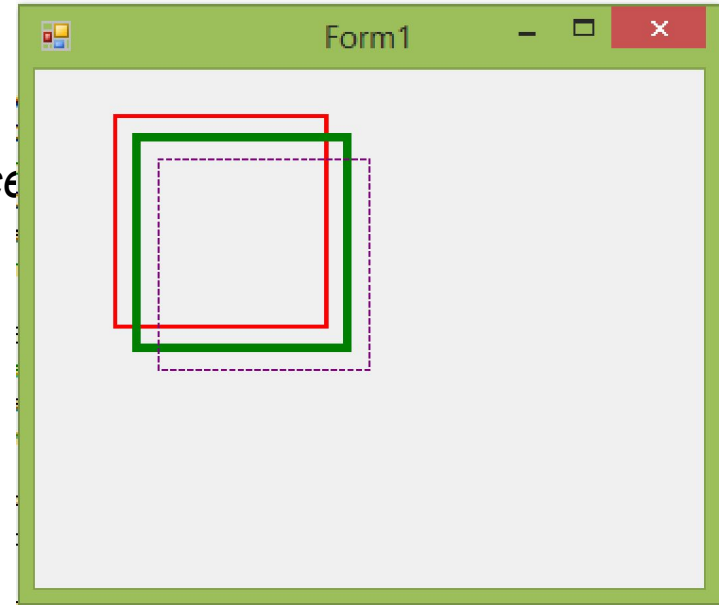
Свойство	Описание
Color	Цвет линии
Width	Толщина линии (задается в пикселах)
DashStyle	Стиль линии (DashStyle::Solid — сплошная; DashStyle::Dash — пунктирная, длинные штрихи; DashStyle::Dot — пунктирная, короткие штрихи; DashStyle::DashDot — пунктирная, чередование длинного и короткого штрихов; DashStyle::DashDotDot — 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.
Цвет, ширину линии и стиль карандаша, созданного программистом, можно изменить. Чтобы это сделать, надо изменить значение соответствующего свойства.

Карандаш программиста.

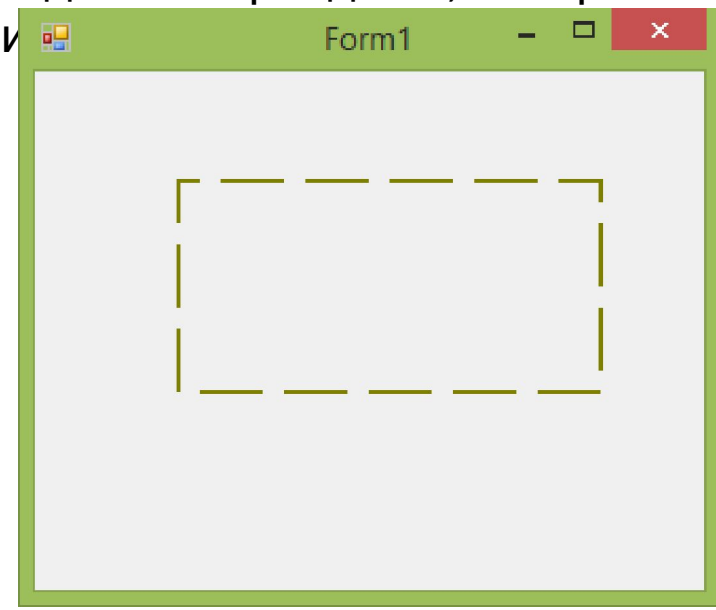
Пример создания и использования карандаша программиста

```
private: System::Void pictureBox1_Paint(System::Object^ sender,
System::Windows::Forms::PaintEventArgs^  e)
{
System::Drawing::Pen^  aPen; // карандаш
// создать красный "толстый" карандаш
aPen = gcnew System::Drawing::Pen(Color::Red,2);
e->Graphics->DrawRectangle(aPen,10,10,100,100);
// теперь карандаш зеленый и его толщина 4 пиксе
aPen->Width = 4;
aPen->Color = Color::Green;
// рисуем зеленым карандашом
e->Graphics->DrawRectangle(aPen,20,20,100,100);
// теперь линия пунктирная
aPen->Width = 1;
aPen->Color = Color::Purple;
aPen->DashStyle = System::Drawing::Drawing2D::DashStyle::Dash;
// рисуем пунктиром
e->Graphics->DrawRectangle(aPen,30,30,100,100);
}
```



Карандаш программиста

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 ниже функция демонстрирует процесс создания карандаша, который рисует пунктирную линию, определенную программи




```
private: System::Void pictureBox1_Paint(System::Object^ sender,
System::Windows::Forms::PaintEventArgs^ e)
{
System::Drawing::Pen^  aPen; // карандаш
aPen = gcnew System::Drawing::Pen(Color::Red,2);
// стиль линии, определенный программистом
array<float>  ^myPattern;
myPattern = gcnew array<float>(2);
myPattern[0] = 15; // штрих
myPattern[1] = 5; // пропуск
aPen->Color = Color::Olive;
aPen->DashStyle = System::Drawing::Drawing2D::DashStyle::Custom;
aPen->DashPattern = myPattern;
e->Graphics->DrawRectangle(aPen,40,40,200,100); // прямоугольник
}
```

Кисть

Кисти используются для закраски внутренних областей геометрических фигур. Например, инструкция e->Graphics->FillRectangle(Brushes::DeepSkyBlue, x, y, w, h); рисует закрашенный прямоугольник.

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

Стандартная кисть закрашивает область одним цветом (сплошная закраска).

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

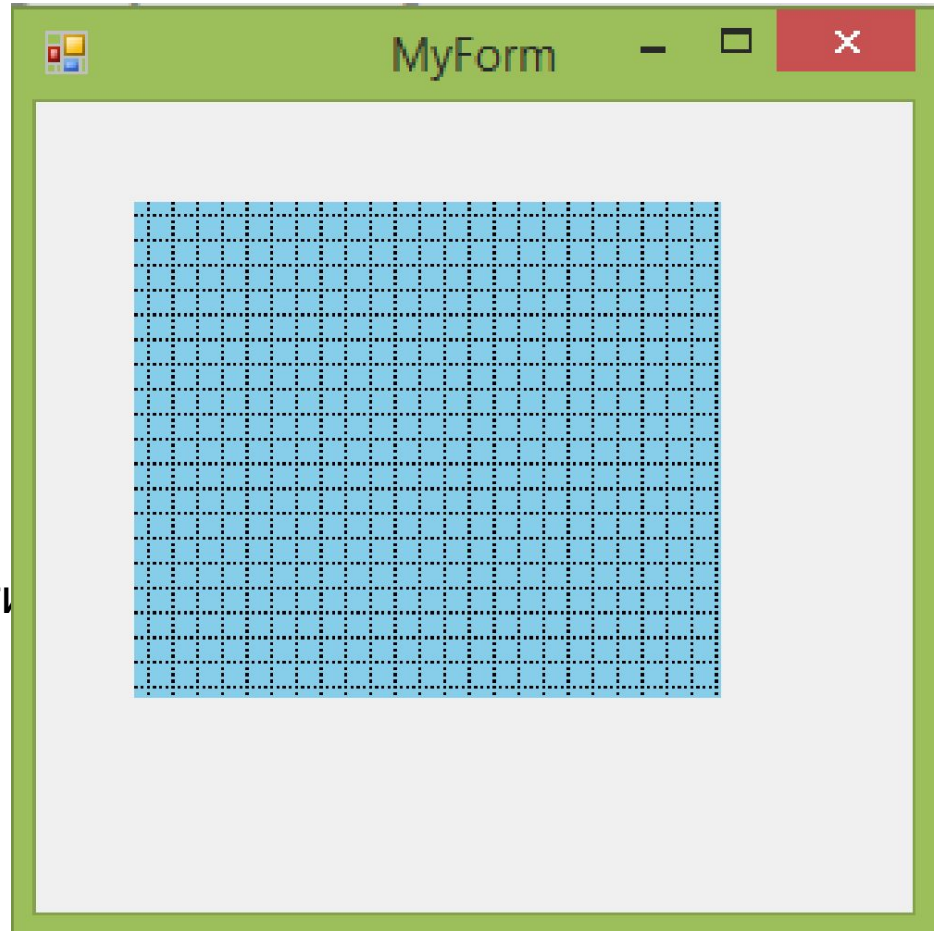
Штриховая кисть

Штриховая кисть (HatchBrush) закрашивает область путем штриховки. Область может быть заштрихована горизонтальными, вертикальными или наклонными линиями разного стиля и толщины. 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"

Штриховая кисть (продолжение)

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.
В листинге приведена функция,
Демонстрирующая процесс создания и использования штриховой кисти




```
private: System::Void pictureBox1_Paint(System::Object^ sender,
System::Windows::Forms::PaintEventArgs^ e) {
// штриховка (HatchBrush-кисть)
HatchBrush^  hBrush = gcnew HatchBrush(HatchStyle::DottedGrid, Color::Black,Color::SkyBlue);
e->Graphics->FillRectangle(hBrush, 20, 20, 90, 60); }
```

Градиентная кисть

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

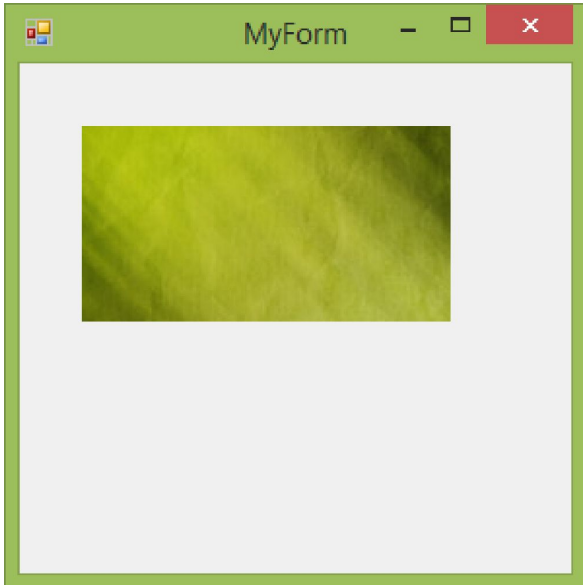
```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {

System::Drawing::Drawing2D::LinearGradientBrush^ myBrush;
myBrush = gcnew System::Drawing::Drawing2D::LinearGradientBrush(Point(5,10),Point(110,10),
Color::Red, Color::White);
e->Graphics->FillRectangle(myBrush,5,5,100,40);
```


```
Rectangle myRect = Rectangle(20,60,200,40);
System::Drawing::Drawing2D::LinearGradientBrush^ myBrush1;
myBrush1 =gcnew
System::Drawing::Drawing2D::LinearGradientBrush(myRect,Color::Red,Color::White,0.0f,true);
e->Graphics->FillRectangle(myBrush1, myRect);
 }
```

Текстурная кисть

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.

```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
float x, y, w, h;
x = 20; y = 20; w = 190; h = 100;
TextureBrush^ myBrush; // текстурная кисть
try
{// загрузить текстуру из файла
myBrush = gcnew TextureBrush(Image::FromFile(
"C:\\Users\\Виктор\\Documents\\Visual Studio 2015\\Projects\\Project1\\zelenaya-tekstura.bmp"));
// рисуем текстурной кистью
e->Graphics->FillRectangle(myBrush, x, y, w, h);}
catch (System::IO::FileNotFoundException^ ex)
{
e->Graphics->DrawRectangle(Pens::Black, x, y, w, h);
e->Graphics->DrawString("Source image",
this->Font, Brushes::Black, x, y + 5);
e->Graphics->DrawString(" not found",
this->Font, Brushes::Black, x, y + 20);
}
}
```



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.

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 задают размер прямоугольника
FillRectangle(Brush,x,y,w,h)	Рисует закрашенный прямоугольник. Пара-метр Brush определяет цвет и стиль закраски прямоугольника; параметры x, y — координаты левого верхнего угла; параметры w и h за-дают 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

Графические примитивы

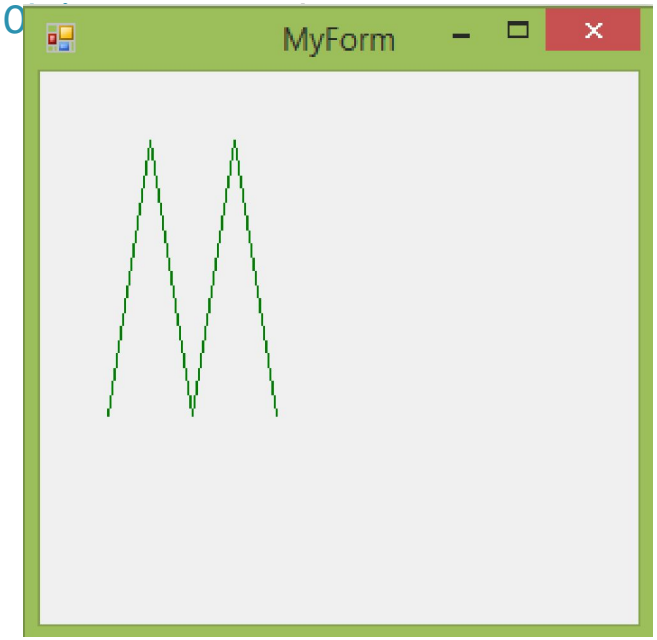
Метод	Действие
FillEllipse(Brush, x, y, w, h)	Рисует закрашенный эллипс. Параметр Brush определяет цвет и стиль закраски внутренней области эллипса; параметры x, y, w, h — координаты левого верхнего угла и размер прямоугольника, внутри которого вычерчивается эллипс.
DrawPolygon(Pen, P)	Рисует контур 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. Параметр Brush определяет цвет и стиль закраски внутренней области многоугольника; параметр 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 которой 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

Рисование линий

Метод **DrawLine** рисует прямую линию. В инструкции вызова метода следует указать карандаш, которым надо нарисовать линию, и координаты точек начала и конца линии:
DrawLine(Pen, x1, y1, x2, y2) или DrawLine(Pen, p1, p2).
Например, инструкция e->Graphics->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 = Point(10,10); Point p2 = Point(300,10); e->Graphics->DrawLine(Pens::Green, p1, p2);

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.

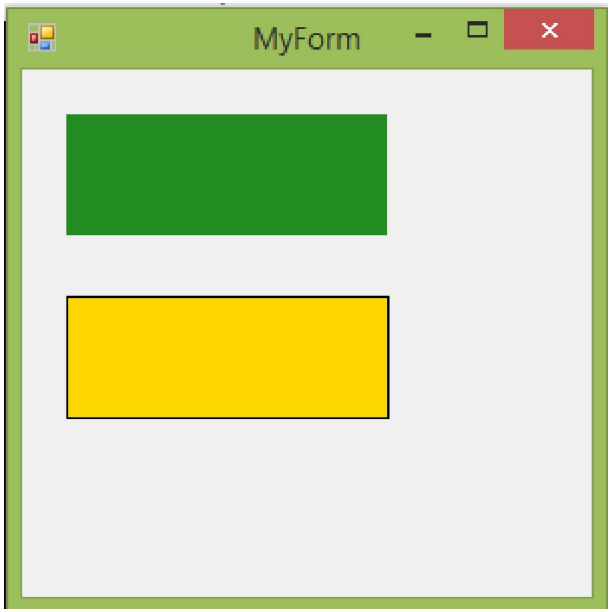
```
private: System::Void pictureBox1_Paint(System::O
System::Windows::Forms::PaintEventArgs^  e) {
array<Point>^ p; // массив точек
p = gcnew array<Point>(5);
p[0].X = 20; p[0].Y = 150;
p[1].X = 40; p[1].Y = 20;
p[2].X = 60; p[2].Y = 150;
p[3].X = 80; p[3].Y = 20;
p[4].X = 100; p[4].Y = 150;
e->Graphics->DrawLines(Pens::Green, p);
}
```



Прямоугольник

Метод **DrawRectangle** чертит прямоугольник. В качестве параметров метода надо указать карандаш, координаты левого верхнего угла и размер прямоугольника:
DrawRectangle(aPen, x, y, w, h); 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 параметров 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);



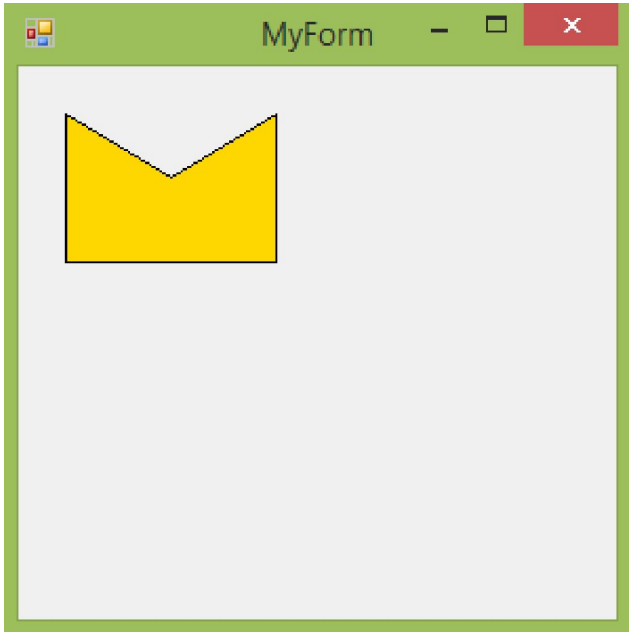

```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
Rectangle aRect;
// положение и размер прямоугольника
// Зеленый прямоугольник размером 160x60,
// левый верхний угол которого в точке (10, 10)
aRect = Rectangle(10, 10, 160, 60); // положение и размер
e->Graphics->FillRectangle(Brushes::ForestGreen, aRect);
// Желтый прямоугольник с черной границей размером 160x60,
// левый верхний угол которого в точке (10, 100)
aRect.Y = 100;
e->Graphics->FillRectangle(Brushes::Gold, aRect); // прямоугольник
e->Graphics->DrawRectangle(Pens::Black, aRect); // граница }
```

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.

Метод FillPolygon рисует закрашенный многоугольник. 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.

```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
array<Point>^ p;
p = gcnew array<Point>(5);
p[0].X = 10; p[0].Y = 80;
p[1].X = 10; p[1].Y = 10;
p[2].X = 60; p[2].Y = 40;
p[3].X = 110; p[3].Y = 10;
p[4].X = 110; p[4].Y = 80;
e->Graphics->FillPolygon(Brushes::Gold, p);
e->Graphics->DrawPolygon(Pens::Black, p);
}
```

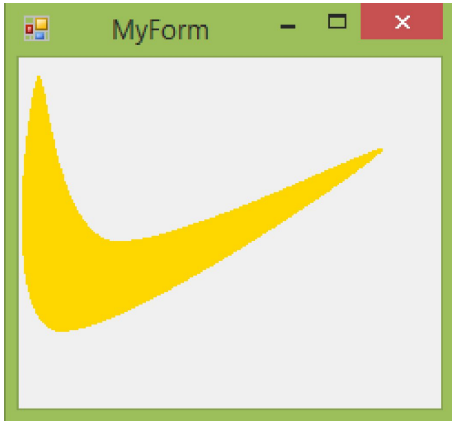


Многоугольник (продолжение)

Когда точки соединяются не прямыми линиями, получается фигура, которую называют «замкнутая кривая».

```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^
e) {

array<Point>^ p = {Point(10,80),Point(10,10),Point(60,40),Point(110,10),Point(110,80)};

e->Graphics->FillClosedCurve(Brushes::Gold, p);
}
```


```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^
e) {

array<Point>^ p = {Point(10,10),Point(50,100),Point(200,50),Point(20,150) };

e->Graphics->FillClosedCurve(Brushes::Gold, p);
}
```



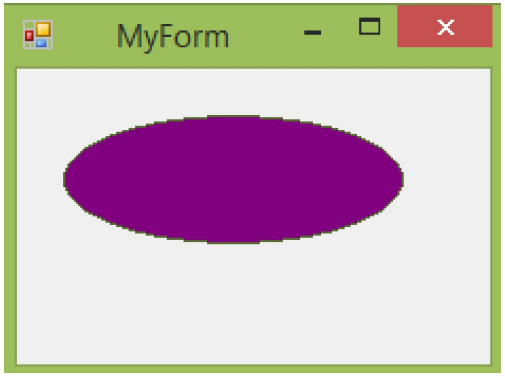
Эллипс и окружность

Метод **DrawEllipse** чертит эллипс внутри прямоугольной области. Если прямоугольник является квадратом, то метод рисует окружность. Инструкция вызова метода DrawEllipse в общем виде выглядит так: DrawEllipse(aPen, x, y, w, h);

В инструкции вызова метода DrawEllipse вместо параметров x, y, w и h можно указать структуру типа Rectangle: DrawEllipse(aPen, aRect);

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);

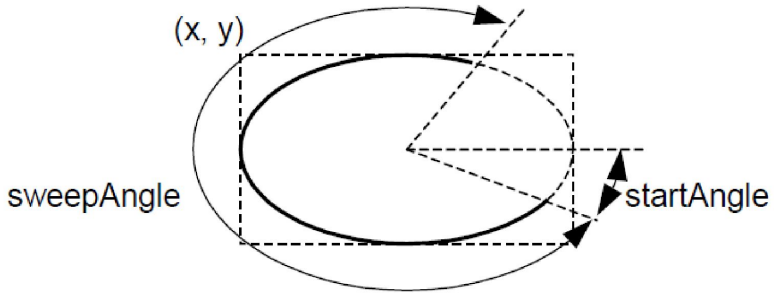
```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
Rectangle aRect;
// положение и размер прямоугольника
// левый верхний угол которого в точке (10, 10)
aRect = Rectangle(10, 10, 160, 60); // положение и размер
// Пурпурный эллипс, вписанный в прямоугольник, размером 160x60
e->Graphics->FillEllipse(Brushes::Purple, aRect); // эллипс
e->Graphics->DrawEllipse(Pens::DarkOliveGreen, aRect); // граница эллипса }
}
```



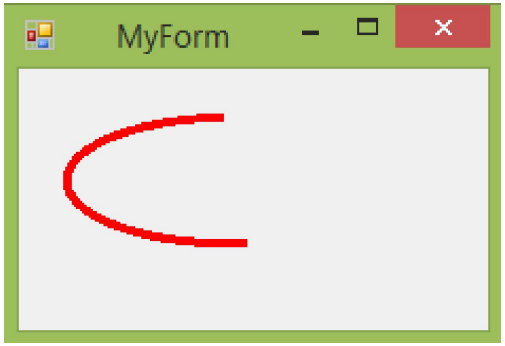
Дуга

Метод **DrawArc** рисует дугу — часть эллипса. 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.



В инструкции вызова метода DrawArc вместо параметров x, y, w и h можно указать структуру типа Rectangle: DrawArc(aPen, aRect, startAngle, sweepAngle)

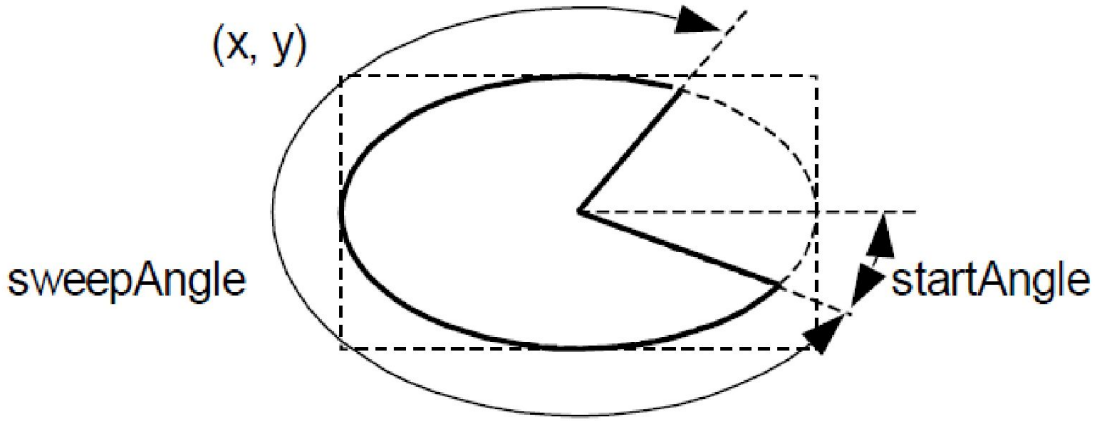



```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
Rectangle aRect;
aRect = Rectangle(10, 10, 160, 60); // положение и размер
System::Drawing::Pen^  aPen; // карандаш
aPen = gcnew System::Drawing::Pen(Color::Red, 4); // создать красный "толстый" карандаш
e->Graphics->DrawArc(aPen, aRect, 80,180); // дуга эллипса }
}
```

Сектор

Метод **DrawPie** рисует границу сектора. Инструкция вызова метода выглядит так:
DrawPie(aPen, x, y, w, h, startAngle, sweepAngle);

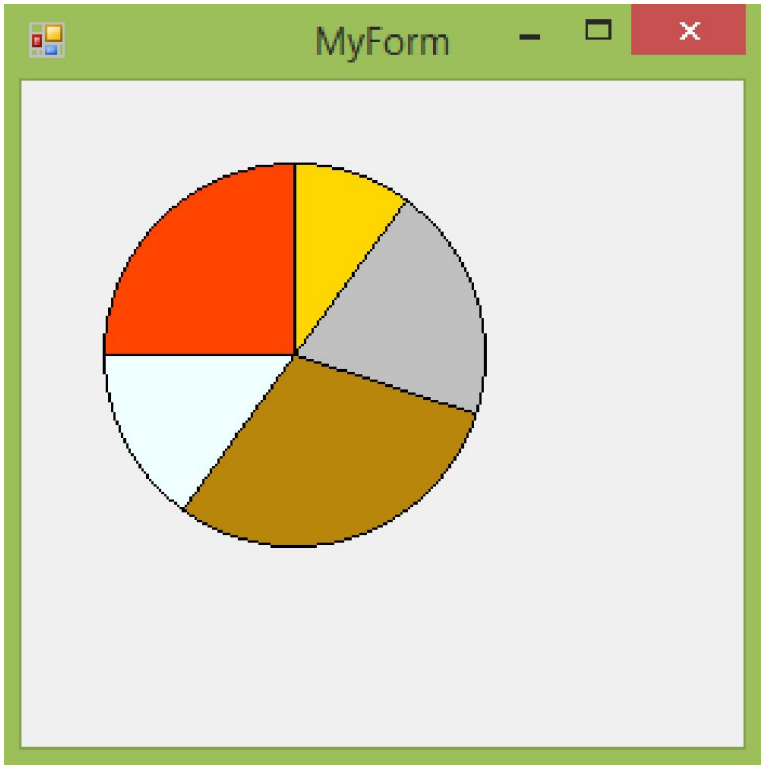
Параметры x, y, w и h определяют эллипс, частью которого является сектор. Параметр startAngle задает начальную точку дуги сектора — пересечение эллипса и прямой, проведенной из центра эллипса и образующей угол startAngle с горизонтальной осью эллипса (угловая координата возрастает по часовой стрелке). 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. 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.

Сектор (продолжение)

```
private: System::Void pictureBox1_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
int N;// количество секторов
array<double>^ p; // доля каждого сектора
p = gcnew array<double>(N);//  инициализируем массив
p[0] = 0.1; p[1] = 0.2; p[2] = 0.3; p[3] = 0.15; p[4] = 0.25;
int X = 20; int Y = 20;//Начальная точка квадрата, в который будет вписан круг
int d = 150;//размер сторон квадрата
int swe; // длина дуги сектора
System::Drawing::Brush^  fBrush = Brushes::White; // кисть для заливки сектора диаграммы
System::Drawing::Graphics^ g = e->Graphics;// графическая поверхность
int j = 0;
N=5;//количество секторов
int sta = -90;// начальная точка дуги сектора
// рисуем диаграмму
for (int i = 0; i < N; i++){
// длина дуги
swe = (int)(360 * p[i]);
// задать цвет сектора
switch (i){
case 0: fBrush = Brushes::Gold;break;
case 1: fBrush = Brushes::Silver;break;
case 2: fBrush = Brushes::DarkGoldenrod;break;
case 3: fBrush = Brushes::Azure;break;
case 4: fBrush = Brushes::OrangeRed;break;}
// рисуем сектор
g->FillPie(fBrush, X, Y, d, d, sta, swe);
// рисуем границу сектора
g->DrawPie(Pens::Black, X,Y, d, d, sta, swe);
sta = sta + swe;// начальная точка дуги для следующего сектора
   }
```



Комбинирование фигур

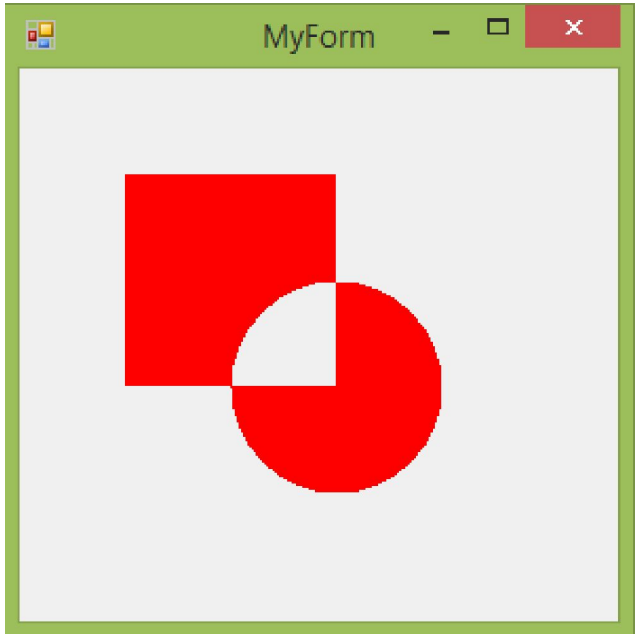
В пространстве имён System::Drawing::Drawing2D существует специальный класс GraphicsPath, который описывает сложные контуры, полученные несколькими фигурами.

Класс хранит список различных фигур, для добавления которых существует ряд функций, например, AddRectangle () или AddEllipse().

Для рисования таких фигур существует функция DrawPath() и FillPath() контекста устройства Graphics.

```
private: System::Void MyForm_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {

//создание контура из прямоугольника и круга
GraphicsPath^ path = gcnew GraphicsPath();

path->AddRectangle(Rectangle(50,50,100,100));

path->AddEllipse(Rectangle(100, 100, 100, 100));

e->Graphics->FillPath(Brushes::Red,path);
}
```

GraphicsPath определяет некоторые границы, которые закрашиваются.

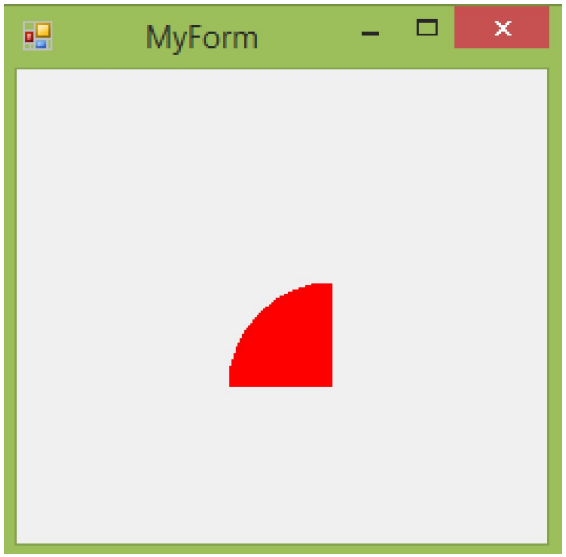


Комбинирование фигур (продолжение)

Используя специальный класс Region при комбинировании доступны режимы пересечения, объединения, исключающего объединения, разности фигур.

Пересечение

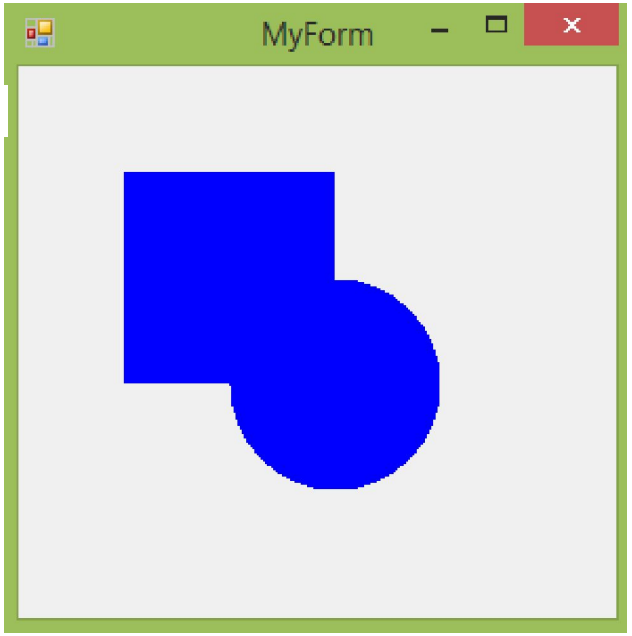
```
private: System::Void MyForm_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {

//создание  GraphicsPath с прямоугольником
GraphicsPath^ path1 = gcnew GraphicsPath();
path1->AddRectangle(Rectangle(50, 50, 100, 100));

//создание  GraphicsPath с кругом;
GraphicsPath^ path2 = gcnew GraphicsPath();
path2->AddEllipse(Rectangle(100, 100, 100, 100));

//создание Region на базе path1
System::Drawing::Region^ r;
r = gcnew System::Drawing::Region(path1);

//пересечение с path2
r->Intersect(path2);

//рисование полученного объекта Region
e->Graphics->FillRegion(Brushes::Red, r);
}
```



Комбинирование фигур (продолжение)

Объединение

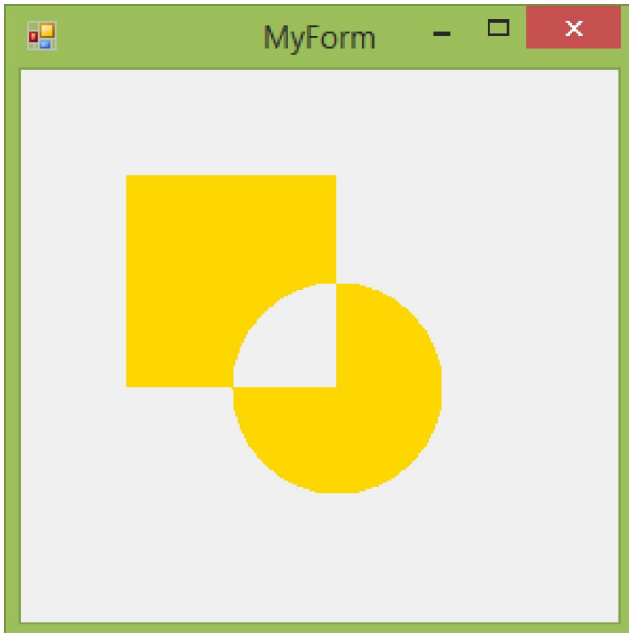
```
private: System::Void MyForm_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {

//создание  GraphicsPath с прямоугольником
GraphicsPath^ path1 = gcnew GraphicsPath();
path1->AddRectangle(Rectangle(50, 50, 100, 100));

//создание  GraphicsPath с кругом;
GraphicsPath^ path2 = gcnew GraphicsPath();
path2->AddEllipse(Rectangle(100, 100, 100, 100));

//создание Region на базе path1
System::Drawing::Region^ r;
r = gcnew System::Drawing::Region(path1);

//пересечение с path2
r->Union(path2);

//рисование полученного объекта Region
e->Graphics->FillRegion(Brushes::Blue, r);
}
```



Комбинирование фигур (продолжение)

Исключающее объединение

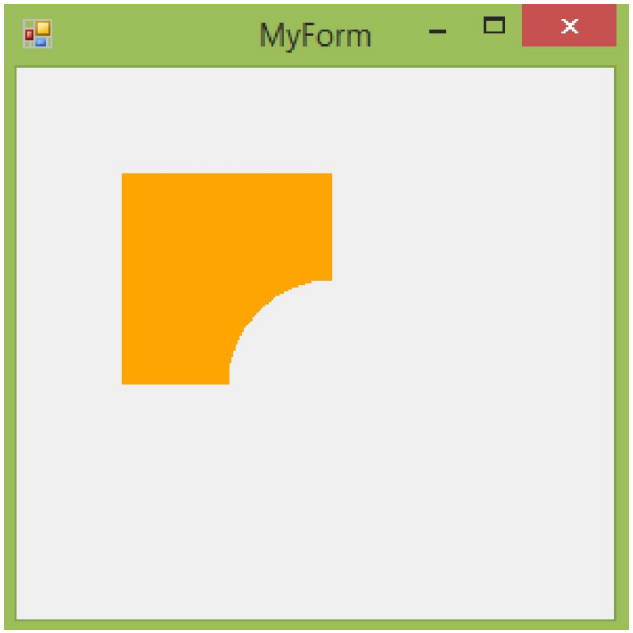
```
private: System::Void MyForm_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {

//создание  GraphicsPath с прямоугольником
GraphicsPath^ path1 = gcnew GraphicsPath();
path1->AddRectangle(Rectangle(50, 50, 100, 100));

//создание  GraphicsPath с кругом;
GraphicsPath^ path2 = gcnew GraphicsPath();
path2->AddEllipse(Rectangle(100, 100, 100, 100));

//создание Region на базе path1
System::Drawing::Region^ r;
r = gcnew System::Drawing::Region(path1);

//пересечение с path2
r->Xor(path2);

//рисование полученного объекта Region
e->Graphics->FillRegion(Brushes::Gold, r);
}
```



Комбинирование фигур (продолжение)

Разность

```
private: System::Void MyForm_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {

//создание  GraphicsPath с прямоугольником
GraphicsPath^ path1 = gcnew GraphicsPath();
path1->AddRectangle(Rectangle(50, 50, 100, 100));

//создание  GraphicsPath с кругом;
GraphicsPath^ path2 = gcnew GraphicsPath();
path2->AddEllipse(Rectangle(100, 100, 100, 100));

//создание Region на базе path1
System::Drawing::Region^ r;
r = gcnew System::Drawing::Region(path1);

//пересечение с path2
r->Exclude(path2);

//рисование полученного объекта Region
e->Graphics->FillRegion(Brushes::Gold, r);
}
```

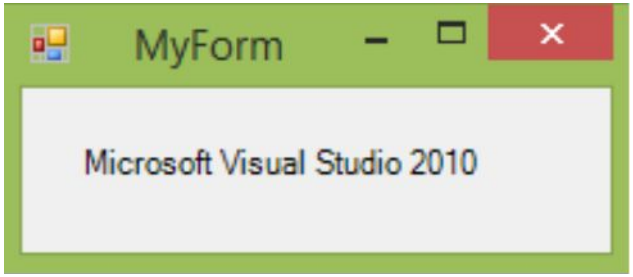


Текст
продолжение

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, а aBrush — цвет текста. Параметры x и y определяют координаты левого верхнего угла области отображения текста.

```
private: System::Void MyForm_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
String^ st1 = "Microsoft Visual Studio 2010";
e->Graphics->DrawString(st1, this->Font, Brushes::Black, 20, 20);}
```



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:
System::Drawing::Font^ aFont = **gcnew** System::Drawing::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 - полужирный; FontStyle::Italic - курсив; FontStyle::UnderLine - подчеркнутый). Параметр FontStyle можно не указывать. В этом случае будет создан шрифт обычного начертания (FontStyle::Regular).

Текст продолжение

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.

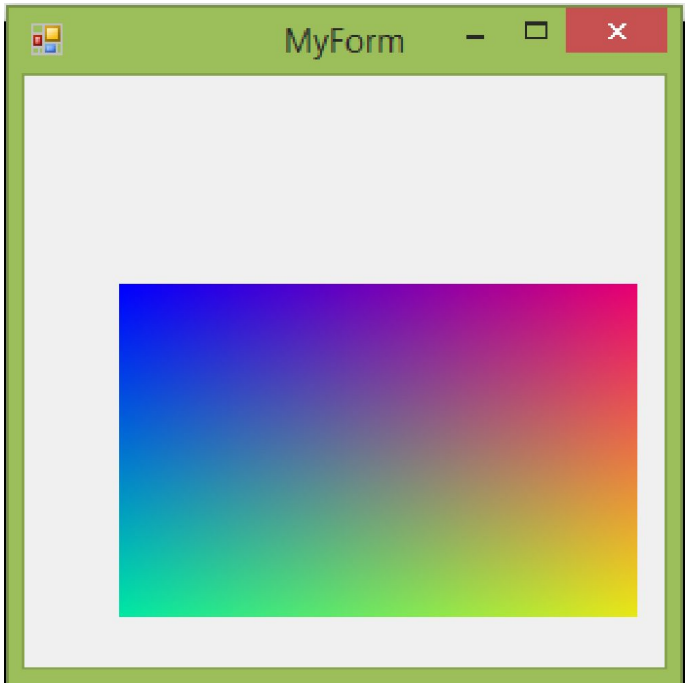
```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^ e) {
int x, y;
x = 10;
y = 10;
String^ st = "Microsoft Visual Studio 2015";
System::Drawing::Font^ rFont = gcnew System::Drawing::Font ("Tahoma", 16, FontStyle::Regular);
System::Drawing::Font^ bFont = gcnew System::Drawing::Font ("Tahoma", 16, FontStyle::Bold);
System::Drawing::Font^ iFont = gcnew System::Drawing::Font ("Tahoma", 16, FontStyle::Italic);
e->Graphics->DrawString(st, rFont, Brushes::Black, x, y);
e->Graphics->DrawString(st, bFont, Brushes::Black, x, y + 20);
e->Graphics->DrawString(st, iFont, Brushes::Black, x, y + 40);
```

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.

Попиксельный вывод

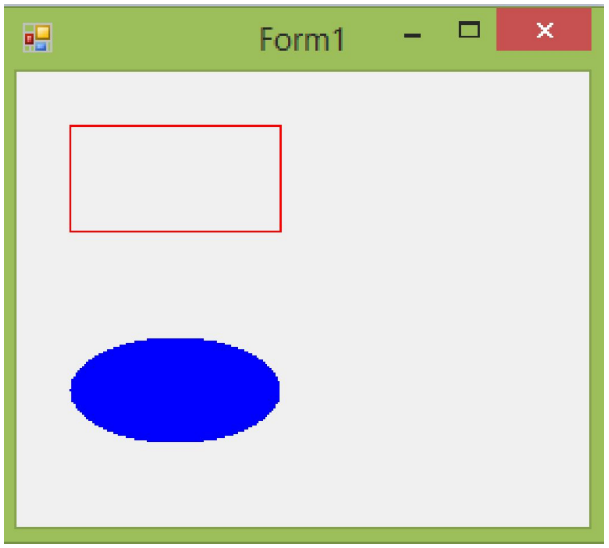
```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
Refresh();
int xmin = 20, ymin = 70;
int xmax = Width - 30, ymax = Height - 70;
int width = xmax - xmin;
int height = ymax - ymin;
Bitmap^ bmp = gcnew Bitmap(500, 300);
for (int y = ymin; y<ymax; y++)
{
for (int x = xmin; x<xmax; x++)
{
int r = 255 * (x - xmin) / width;
int g = 255 * (y - ymin) / height;
int b = 255 - 255 * (x - xmin + y - ymin) / (width + height);
// создание изображения размером 500 на 300
Color f = Color::FromArgb(r, g, b);
bmp->SetPixel(x, y, f);}}
// создания контекста устройства рисования в изображении
Graphics^ image = Graphics::FromImage(bmp);
// рисование в bmp
e->Graphics->DrawImage(bmp, 10, 10);
}
```



Трансформации

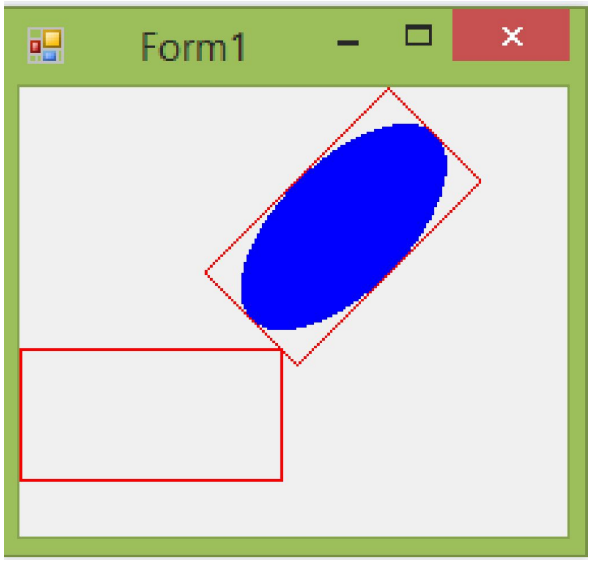
Перенос- линейное перемещение объекта.

```
private: System::Void Form1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
Rectangle ellRect;
ellRect=Rectangle(25,25,100,50);
// Перенести объект на 100 пикселов вниз
 e->Graphics->TranslateTransform(0,100);
// Нарисовать Эллипс, используя заданный
прямоугольник
 e->Graphics->FillEllipse(Brushes::Blue,ellRect);
//Отменить смещение координат
 e->Graphics->ResetTransform();
//Нарисовать прямоугольник
 e->Graphics->DrawRectangle(Pens::Red,ellRect); }
```



Поворот: угол указывается в градусах

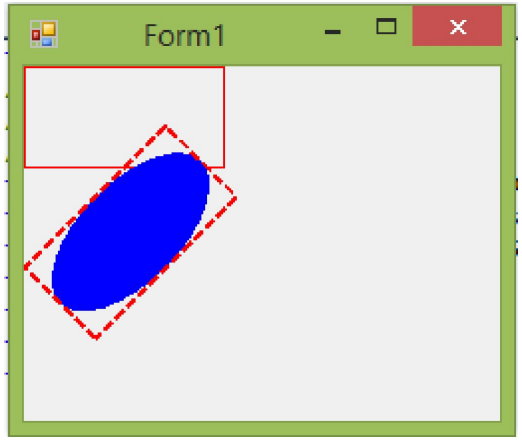
```
private: System::Void Form1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
 Rectangle ellRect;
 ellRect=Rectangle(0,100,100,50);
 e->Graphics->DrawRectangle(Pens::Red,ellRect);
 e->Graphics->RotateTransform(-45);
 e->Graphics->FillEllipse(Brushes::Blue,ellRect);
 e->Graphics->DrawRectangle(Pens::Red,ellRect);
 e->Graphics->ResetTransform(); }
```



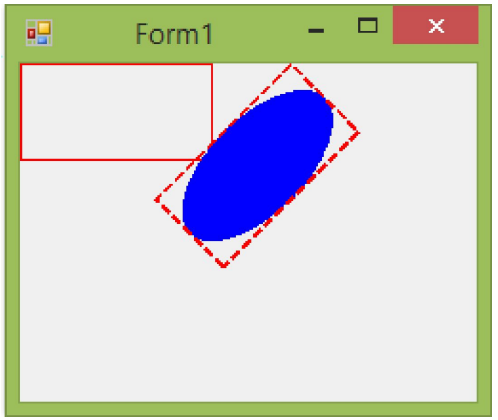
Трансформация (продолжение)

Перенос с поворотом (При объединении преобразований их порядок очень важен)

```
Rectangle ellRect;
 ellRect=Rectangle(0,0,100,50);
 System::Drawing::Pen^  myPen; // карандаш
 myPen = gcnew System::Drawing::Pen(Color::Red,2);
 myPen->DashStyle = System::Drawing::Drawing2D::DashStyle::Dash;
 e->Graphics->DrawRectangle(Pens::Red,ellRect);
 e->Graphics->TranslateTransform(0,100);
 e->Graphics->RotateTransform(-45);
 e->Graphics->FillEllipse(Brushes::Blue,ellRect);
 e->Graphics->DrawRectangle(myPen,ellRect);
 e->Graphics->ResetTransform();
```



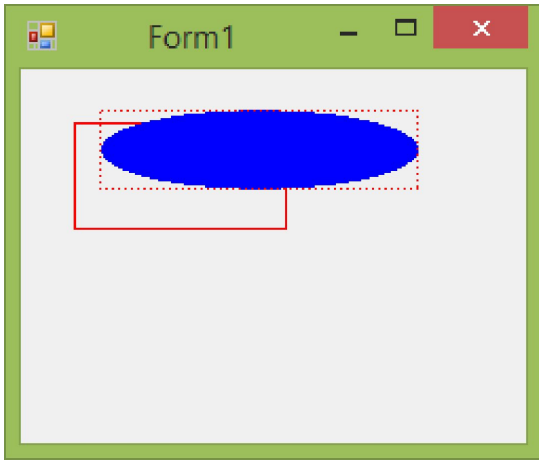

```
 Rectangle ellRect;
 ellRect=Rectangle(0,0,100,50);
 System::Drawing::Pen^  myPen; // карандаш
 myPen = gcnew System::Drawing::Pen(Color::Red,2);
 myPen->DashStyle = System::Drawing::Drawing2D::DashStyle::Dash;
 e->Graphics->DrawRectangle(Pens::Red,ellRect);
 e->Graphics->RotateTransform(-45);
 e->Graphics->TranslateTransform(0,100);
 e->Graphics->FillEllipse(Brushes::Blue,ellRect);
 e->Graphics->DrawRectangle(myPen,ellRect);
 e->Graphics->ResetTransform();}
```



Трансформация (продолжение)

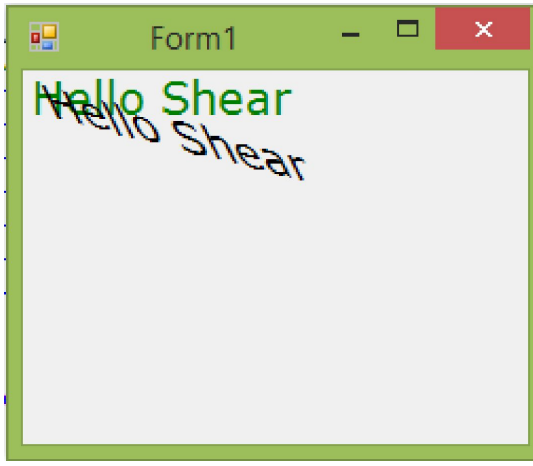
Масштабирование

```
Rectangle ellRect;
 ellRect=Rectangle(25,25,100,50);
 System::Drawing::Pen^  myPen; // карандаш
 myPen = gcnew System::Drawing::Pen(Color::Red,1);
 myPen->DashStyle = System::Drawing::Drawing2D::DashStyle::Dot;
 e->Graphics->DrawRectangle(Pens::Red,ellRect);
 e->Graphics->ScaleTransform(1.5f,0.75f);
 e->Graphics->FillEllipse(Brushes::Blue,ellRect);
 e->Graphics->DrawRectangle(myPen,ellRect);
 e->Graphics->ResetTransform();
```



Сдвиг (При сдвиге фрагмента изображения необходимо непосредственно определять матрицу преобразований)

```
 System::Drawing::Font^ font;
 font=gcnew System::Drawing::Font("Verdana",16,FontStyle::Regular);
 e->Graphics->DrawString ("Hello Shear",font,Brushes::Green,0,0);
 Matrix^ matrix;
 matrix=gcnew Matrix();
 matrix->Shear(0.5f,0.25f);
 e->Graphics->Transform=matrix;
 e->Graphics->DrawString ("Hello Shear",font,Brushes::Black,0,0);
 e->Graphics->ResetTransform();
```



Анимаци

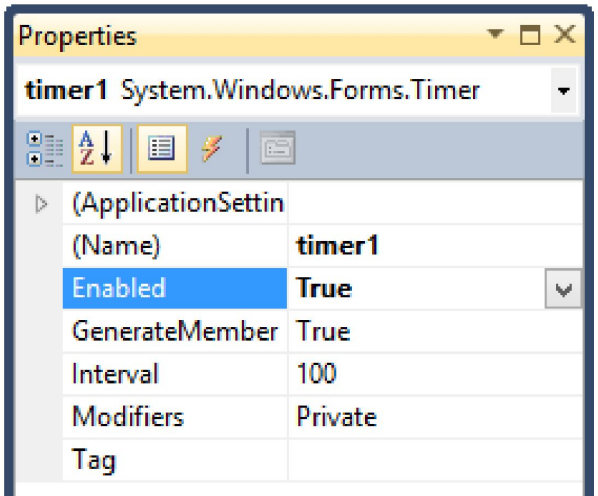
Наиболее просто создать эффект меняющейся картинки можно путем вывода на экран (графическую поверхность) последовательности заранее подготовленных картинок (кадров), то есть методом, так называемой, классической анимации. Кадры анимации обычно помещают в один файл (так удобнее их создавать), в виде изображения, состоящего из отдельных изображений, предназначенных к показу.

Рассмотрим вывод анимации бегущей зебры, состоящей из 6 фрагментов, находящихся в файле БегЗебры.jpg.

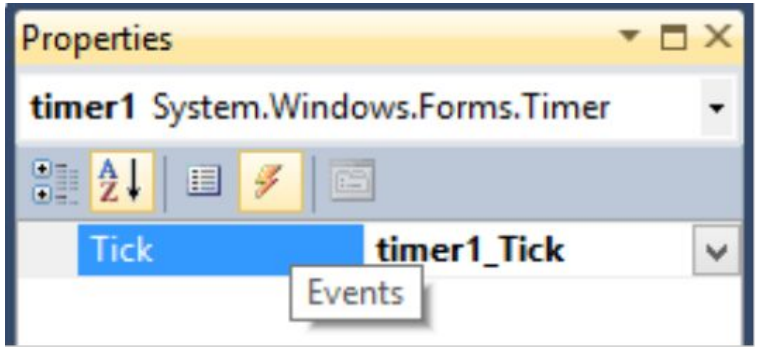
Для управления выводом кадров во времени необходимо поставить на форму компонент Timer. Компонент Timer генерирует последовательность событий Tick. Обычно он используется для активизации с заданным периодом некоторых действий. Компонент является невизуальным (во время работы программы в окне не отображается). Свойства компонента приведены в таблице.

Свойство	Описание
`Interval`	Период генерации события `Tick`. Задается в миллисекундах
`Enabled`	Разрешение работы. Разрешает (значение `True`) или запрещает (значение `False`) генерацию события `Tick`

Примем, чтобы цикл движений бега зебры составлял 600 миллисекунд. Так как у нас 6 кадров, то период генерации события Tick (`Interval`) задаём 100 миллисекунд. Включаем таймер по-умолчанию, установив свойство Enabled в True.



В поле Events в свойствах компонента Timer задаём для события Tick функцию обработки этого события. Если два раза щёлкнуть мышкой по свойству, то будет задана функция , имеющая название по-умолчанию timer1_Tick.

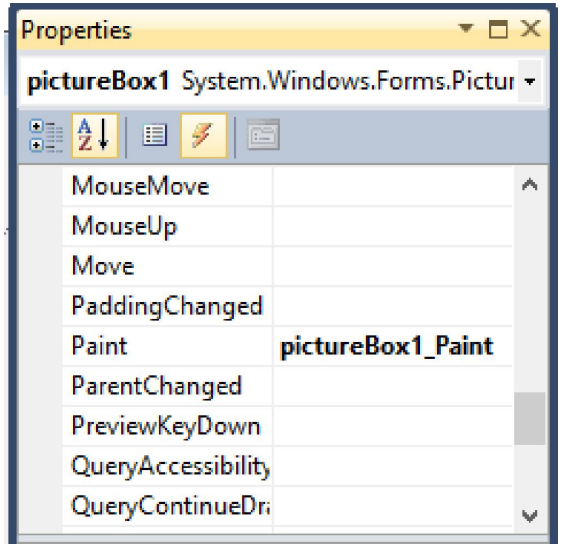


В файл Form1.h будет добавлена при этом заготовка для написания функции:

```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {

 }
```

Для вывода графики ставим на форму компонент PictureBox и задаём функцию обработки события pictureBox 1_Paint

```
private: System::Void pictureBox1_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {

}
```



В начале работы программы составное изображение загружаем в буфер (объект типа Bitmap). Для этого после создания формы, компонентов PictureBox и Timer дописываем в начале заголовочного файла Form1.h выделенные жирным текстом строки кода

```
public ref class Form1 : public System::Windows::Forms::Form
{
private:
Bitmap^ zebra;//Объявляем объект типа Bitmap, в котором будем хранить исходное изображение анимации
int hz,wz;//объявляем переменные, в которых будем хранить размеры окна вывода анимации
int s;//Объявляем переменную, хранящую номер выводимого фрагмента анимации
private: System::Windows::Forms::Timer^  timer1;
private: System::Windows::Forms::PictureBox^  pictureBox1;
public:
Form1(void)
{
InitializeComponent();
zebra=gcnew Bitmap (Application::StartupPath + "\\БегЗебры.jpg");//Создаём экземпляр объекта типа  Bitmap
hz=(zebra->Height)/2; //Инициализируем размеры окна показа анимации: по вертикали - 1/2 высоты,
wz=(zebra->Width)/3; //по горизонтали 1/3 ширины изображения
s=0; //Задаём первоначальный номер фрагмента, равный 0
}
protected:
/// <summary>
/// Clean up any resources being used.
/// </summary>
~Form1()
{
if (components)
{
delete components;
}
}
```

Вывести на графическую поверхность фрагмент битового образа (загруженный битовый образ содержит все кадры, а нам нужен отдельный кадр) можно при помощи метода DrawImage.

Инструкция вызова метода DrawImage для отображения фрагмента битового образа в общем виде выглядит так:

e->Graphics->DrawImage(*bitmap*, *r1*, *r2*, GraphicsUnit::Pixel);

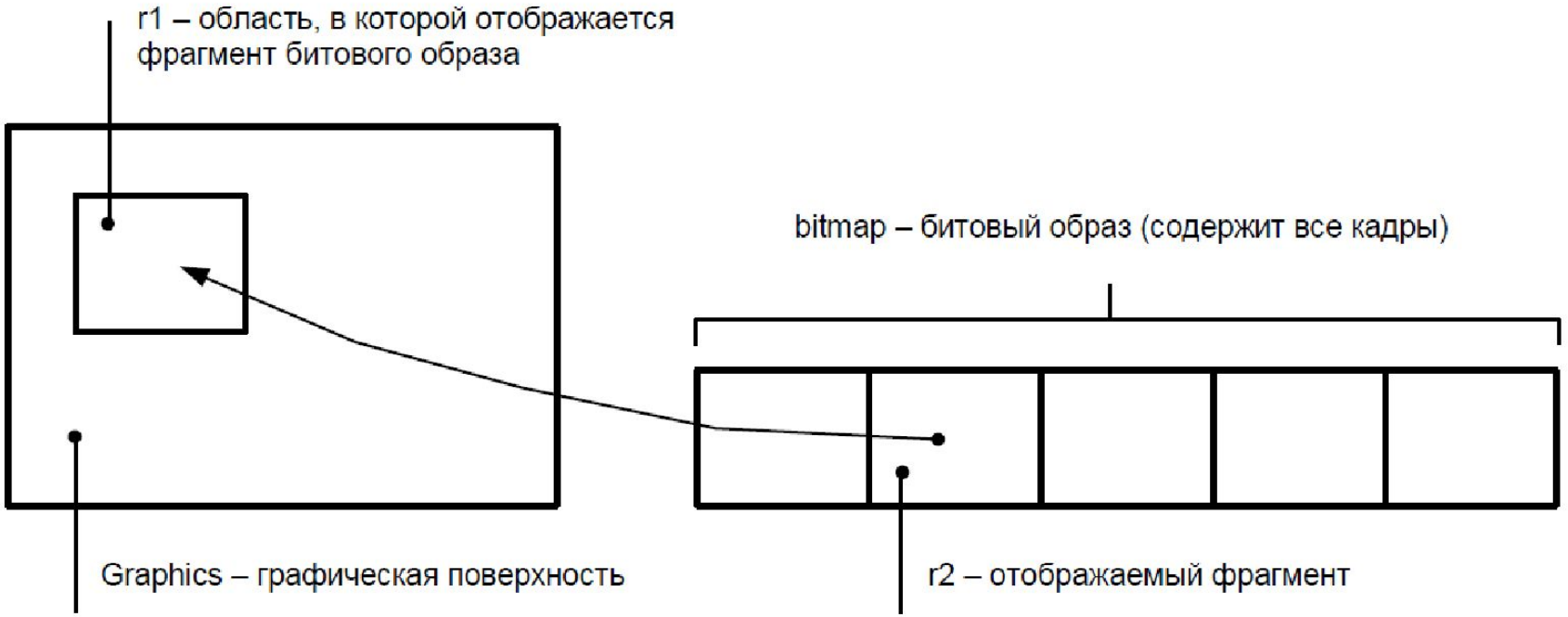
где:

Graphics — поверхность, на которую выполняется вывод фрагмента битового образа bitmap;

bitmap — битовый образ, фрагмент которого выводится на поверхность Graphics;

r1 — область на поверхности Graphics, в которую выполняется вывод фрагмента битового образа (объект Rectangle);

r2 — фрагмент битового образа bitmap (объект Rectangle), который выводится на поверхность Graphics.



Заполняем тело функций:

```
private: System::Void pictureBox1_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
//Задаём размеры pictureBox равными размеру кадра анимации
pictureBox1->Height=zh;
pictureBox1->Width=zw;
//задаём положение и размер поля вывода фрагментов анимации,
//а также положение и размер фрагмента в исходном изображении
Rectangle r1;
Rectangle r2;
r1.X=0; r1.Y=0;
r1.Width=zw;
r1.Height=zh;
r2.Width=zw;
r2.Height=zh;
if (s<3) {r2.X=s*zw;r2.Y=0;}
else {r2.X=(s-3)*zw;r2.Y=zh;}
if (zebra != nullptr)
{
// вывести фрагмент анимации
e->Graphics->DrawImage(zebra,r1,r2,GraphicsUnit::Pixel);
} }

private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
 if (s<5) {s++;}  else s=0;

 pictureBox1->Refresh(); //метод Refresh()  создаёт  для  pictureBox1 событие Paint,  что перезапускает функцию
                  // pictureBox1_Paint

 }
```

Результат