

# Основы программирования

## Занятие 1. Реализация изображений. Графические методы в Delphi.

**Задание:** Изучите теорию, выполните предложенные задания. Код отправьте скриншотом или текстовым файлом педагогу «ВКонтакте».

### Пиксели и цвет

У вас наверняка возник вопрос: «А как в программах для *Windows* выполняется рисование?»

Каждый компонент, на котором можно нарисовать, имеет свойство `Canvas` (*холст*), с помощью его свойств и методов можно нарисовать все, что угодно.

Создайте новый проект с формой (*VCL Forms Application*) и сохраните его в папке *Пиксели* под именем `Pixels.bdsproj`.

Определить или изменить цвет отдельных пикселей можно с помощью свойства-массива `Pixels`. У каждого пикселя две стандартных координаты:  $x$  — расстояние от левой границы холста,  $y$  — расстояние от верхней границы. Точка  $(0, 0)$  — это левый верхний угол.

Чтобы узнать или поменять цвет пикселя с координатами  $(x, y)$ , используют обращение `Canvas.Pixels[x, y]`.

Некоторые элементы формы (например, заголовок и рамка) недоступны для рисования («обычными» методами). Область, с которой можно работать, называется *клиентской*, ее ширина и высота определяются через свойства формы `ClientWidth` и `ClientHeight`. Поскольку отсчет координат начинается с нуля, нижний правый пиксель клиентской области имеет координаты  $(ClientWidth-1, ClientHeight-1)$ .

Добавьте обработчик события `OnClick` для формы:

```
var x, y: integer;
begin
  for x:=0 to ClientWidth-1 do
    for y:=0 to ClientHeight-1 do
      Canvas.Pixels[x,y] := clRed;
    end;
  end;
```

Запустите программу и щелкните мышью по форме.

Легко понять, что в этой программе в двойном цикле перебираются все пиксели, и для каждого устанавливается красный цвет, который обозначается как `clRed`.

Поскольку этот обработчик является методом формы, здесь запись `Canvas` обозначает то же самое, что `Form1.Canvas`, то есть, мы рисуем прямо на форме.

Даже на быстродействующем компьютере работа с отдельными пикселями выполняется медленно, нужно этого избегать: рисовать с помощью линий, прямоугольников и др.

Цвет можно задать и по-другому, а формате *RGB* (*red* — красный, *green* — зеленый, *blue* — синий). Каждая из этих составляющих цвета — целое число в интервале от 0 до 255.

Измените строчку, где записывается новый цвет:

```
Canvas.Pixels[x, y] := RGB(0, 255, 255);
```

Запустите программу и проверьте ее.

Цвет в *Delphi* — это целое число, занимающее 4 байта. В принципе, можно прямо записывать это число в массив `Pixels`, при этом удобнее всего использовать шестнадцатеричную запись, где каждая составляющая кодируется двумя шестнадцатеричными цифрами (и занимает 1 байт). Перед шестнадцатеричными числами в Паскале ставится знак `$`.

Измените строчку, где записывается новый цвет:

```
Canvas.Pixels[x, y] := $00FFFF;
```

Запустите программу и проверьте ее.

Возможно, вы ожидали снова увидеть бирюзовый цвет (смесь зеленого с синим), а получился — желтый, то есть, смесь зеленого с красным. Дело в том, что в такой записи старший байт определяет синий цвет, следующий — зеленый и младший — красный.

Функции `GetRValue`, `GetGValue` и `GetBValue` позволяют выделить из цвета красную, зеленую и синюю составляющие. Все они возвращают байтовое значение.

В начале обработчика `OnClick` добавьте объявление переменных

```
var R, G, B: byte;  
    color: TColor;
```

В конец процедуры добавьте код, который определяет цвет пикселя (10, 10), раскладывает его на составляющие и выводит их значения в окне сообщения:

```
color := Canvas.Pixels[10, 10];  
R := GetRValue(color);  
G := GetGValue(color);  
B := GetBValue(color);  
ShowMessage('Цвет (' + IntToStr(R) + ', ' + IntToStr(G)  
            + ', ' + IntToStr(B) + ')');
```

Проверьте работу программы.

Переменная `color`, в которую сначала записывается цвет пикселя, объявлена как `TColor` — специальный тип данных для кодирования цвета. По сути, это целое число, но именно такое объявление более грамотно, потому что явно показывает назначение переменной.

Цвет пикселей можно получать в результате вычислений, при этом могут получиться весьма интересные узоры.

Измените строчку, где записывается новый цвет:

```
Canvas.Pixels[x, y] := x*y;
```

Запустите программу, разверните окно на полный экран и щелкните по форме. После этого закройте проект.

## Геометрические фигуры

В *Delphi*, так же, как и в других системах программирования с графическими возможностями, существуют так называемые **графические примитивы** — простейшие объекты, из которых составляется рисунок.

Цвет и стиль линий определяется свойством `Canvas.Pen` (перо), которое содержит несколько подсвойств. Из них наиболее важны три следующих:

`Color` — цвет линий;

Width — толщина линий

Style — стиль линии: *psClear* — нет линий, *psSolid* — сплошная, *psDash* — штриховая, *psDot* — точечная и другие.

По умолчанию линия сплошная, имеет черный цвет и толщину 1 пиксель. Вот эта команда устанавливает красный цвет для линий:

```
Canvas.Pen.Color := clRed;
```

Замкнутые фигуры можно залить сплошным цветом или узором прямо при рисовании. Тип заливки определяется кистью — свойством `Canvas.Brush`, которое имеет два основных подсвойства:

Color — цвет кисти;

Style — стиль кисти: *bsClear* — нет заливки, *bsSolid* — сплошная, *bsCross* — в клеточку и другие.

По умолчанию устанавливается сплошная заливка белым цветом.

При рисовании линий используется понятие *текущей позиции рисования* (графического курсора). Это невидимая точка, из которой начинается рисование очередного отрезка. Метод `MoveTo(x, y)` перемещает курсор (текущую позицию) в точку с координатами  $(x, y)$ , а команда `LineTo(x, y)` рисует отрезок из текущей позиции в точку  $(x, y)$ . Например, эти команды рисуют отрезок из точки  $(10, 10)$  в точку  $(100, 100)$ :

```
Canvas.MoveTo ( 10, 10 );  
Canvas.LineTo ( 100, 100 );
```

После выполнения метода `LineTo(x, y)` текущая позиция смещается в точку  $(x, y)$ .

Создайте новый проект и сохраните его в папке *Графика* под именем `Graph.bdsproj`.

Добавьте кнопку `TButton` с надписью «Нарисовать».

Добавьте в описание класса формы процедуру `Draw`:

```
public  
  procedure Draw;
```

Добавьте в секцию кода `implementation` реализацию процедуры:

```
procedure TForm1.Draw;  
begin  
  Canvas.Pen.Color := clBlue;  
  Canvas.MoveTo(0, 0);  
  Canvas.LineTo(100, 100);  
end;
```

Создайте обработчик события `OnClick`, который вызывает процедуру `Draw`. Запустите программу и проверьте ее работу.

Хотелось бы избавиться от утомительного написания символов «`Canvas.`» перед каждой командой рисования. Для этого есть простой способ: использовать оператор `with`:

```
with Canvas do begin  
  ...  
end;
```

Программа будет считать, что все обращения внутри этого блока — это обращения к свойствам и методам объекта `Canvas`.

Измените код процедуры таким образом:

```
with Canvas do begin
  Pen.Color := clBlue;
  MoveTo(0,0);
  LineTo(100,100);
end;
```

Для рисования прямоугольника используется метод

```
Canvas.Rectangle ( x1, y1, x2, y2 );
```

Два противоположных угла этого прямоугольника находятся в точках  $(x_1, y_1)$  и  $(x_2, y_2)$ , для контура используются настройки пера (`Canvas.Pen`), для заливки внутренней части — кисть (`Canvas.Brush`). Следующий код рисует красный прямоугольник с синей заливкой:

```
with Canvas do begin
  Pen.Color := clRed;
  Brush.Color := clBlue;
end;
```

Чтобы отключит заливку, достаточно установить прозрачную кисть, то есть

```
Canvas.Brush.Style := bsClear;
```

Эллипсы (и круги) рисуются с помощью метода `Canvas.Ellipse`:

```
Canvas.Ellipse( x1, y1, x2, y2 );
```

Точки  $(x_1, y_1)$  и  $(x_2, y_2)$  — это противоположные углы прямоугольника, в который вписан этот эллипс.

## Заливка

Чтобы залить прямоугольник каким-то цветом или узором (не рисуя рамку), используют метод `Canvas.FillRect`. Его параметр — объект типа `TRect`, который можно построить с помощью функции `Rect` прямо при вызове метода:

```
Canvas.FillRect( Rect(x1, y1, x2, y2) );
```

Здесь  $x_1, y_1, x_2$  и  $y_2$  — числа или арифметические выражения. Как и в методе `Rectangle`, точки  $(x_1, y_1)$  и  $(x_2, y_2)$  — это противоположные углы прямоугольника.

Иногда нужно залить область со сложной границей. В этом случае применяют метод `FloodFill`. Например,

```
Canvas.FloodFill ( 50, 100, clBlack, fsBorder );
```

Цвет и стиль заливки определяется текущими установками кисти.

Здесь первые два параметра — координаты точки, откуда начинается заливка. Третий параметр — цвет границы или внутренней области, в зависимости от значения четвертого параметра. Если он равен `fsBorder`, мы заливаем все, начиная от исходной точки до границы заданного цвета (в данном примере — черного, `clBlack`). Черная граница должна быть сплошной, если она будет разорвана, заливается весь холст.

Возможен второй вариант, когда нужно залить область одного цвета, и эта область не имеет одноцветной границы. Так работает, например, заливка в редакторе *Paint*. В этом случае заливка, начинающаяся с точки  $(50, 100)$ , выполняется так:

```
Canvas.FloodFill ( 50, 100, Canvas.Pixels[50,100], fsSurface );
```

Обратите внимание, что для надежности мы определили цвет точки (50,100) с помощью массива `Pixels`

## Вывод текста

Для того, чтобы вывести текст на холст, используется метод `Canvas.TextOut`. Например,

```
Canvas.TextOut ( 50, 100, 'Дом, который построил Джек' );
```

Первые два параметра определяют левый верхний угол текста (здесь — точка (50,100)), третий — сам текст (символьная строка).

Используемый шрифт задается в свойстве `Canvas.Font`, которое имеет несколько подсвойств (название шрифта, цвет, размер, стиль и др.). В этом примере текст набран шрифтом *Verdana*, размером 10 пт, жирным, курсивным:

```
with Canvas do begin
  Font.Name := 'Verdana'; // гарнитура
  Font.Color := clRed;    // цвет
  Font.Size := 10;       // размер в пунктах
  Font.Style := [fsBold,fsItalic]; // стиль
  TextOut ( 50, 100, 'Дом, который построил Джек' );
end;
```

Напомним, что свойство `Font.Style` — это множество, которое может включать элементы *fsBold* (жирный), *fsItalic* (курсив), *fsUnderline* (подчеркивание) и *fsStrikeOut* (вычеркивание) в любых комбинациях.

У объекта `Canvas` есть и другие методы, про которые можно прочитать в справочной системе или в Интернете. Некоторые из них мы изучим далее.

## Практикум

Измените программу так, чтобы при нажатии на кнопку на форме появлялся домик с подписью, примерно такой, как на рисунке:



Для закраски двери используйте коричневый цвет (R=140, G=70, B=20).

## 2. Сохранение рисунка

Еще раз запустите программу, и выведите рисунок на форму. После этого сверните окно программы (кнопка  в заголовке окна программы) и снова раскройте его. Сохранился ли рисунок?

А рисунок не сохранился. Почему?

Дело в том, что мы рисовали домик прямо на экране, холст `Canvas` нигде не хранит это изображение, поэтому при сворачивании окна оно было потеряно. Где же выход?

Чтобы ответить на этот вопрос, нужно понять, как строится изображение в окне программы. Дело в том, что *Windows* не хранит изображения окон программ. Каждая программа обязана уметь перерисовать свое окно, когда получит от операционной системы специальное сообщение `WM_PAINT`. Например, это случается при разворачивании окна из свернутого состояния. Перехватить это сообщение можно с помощью события `OnPaint`.

Создайте обработчик события `OnPaint` для формы, вызвав в нем процедуру `Draw`. Проверьте, решило ли это проблему.

Теперь рисунок на форме появляется сразу, хотя и сохраняется при любых действиях с окном. Для того, чтобы мы видели рисунок только после нажатия на кнопку, введем логическую переменную `IsReady`, которой будем устанавливать значение `True` в обработчике `OnClick` кнопки. Напомним, что во все глобальные переменные при создании записываются нулевые значения, для логической переменной это соответствует значению `False`.

Объявите глобальную логическую переменную `IsReady`. В обработчике `OnClick` присвойте ей значение `True`. Измените обработчик `OnPaint` чтобы процедура `Draw` вызывалась только при истинном значении переменной `IsReady`. Проверьте работу программы.

Показанный прием (перерисовка в обработчике `OnPaint`) хорошо работает, когда рисунок несложный и перерисовать его — дело сотых долей секунды. Теперь представьте, что рисунок строится очень долго (минуты или даже часы), например, в ходе сложных расчетов или эксперимента. Перерисовать его быстро невозможно, значит, изображение нужно где-то хранить, конкретнее — в памяти.

Для хранения изображений в памяти служит класс `TBitmap` (*bitmap* — битовая карта). При запуске программы рисунок в памяти нужно создать (вызвав конструктор `TBitmap.Create`), а при завершении работы — удалить из памяти с помощью метода-деструктора `Free`.

Объявите глобальную переменную `bmp` типа `TBitmap`:

```
var bmp: TBitmap;
```

В обработчике `OnCreate` формы создайте рисунок в памяти:

```
bmp := TBitmap.Create;  
bmp.Width := 300;  
bmp.Height := 300;
```

Свойства `Width` и `Height` задают ширину и высоту рисунка в пикселях.

В обработчике `OnDestroy`, который вызывается при уничтожении формы, освободите память:

```
bmp.Free;
```

Теперь в процедуре `Draw` нам нужно рисовать не на форме, а на холсте объекта `bmp`, поэтому она должна выглядеть так:

```
with bmp.Canvas do begin
  ...
end;
```

Измените процедуру `Draw`, чтобы рисование выполнялось в памяти на холсте объекта `bmp`. Запустите программу и проверьте ее работу, попытайтесь понять, почему рисунка не видно.

Теперь процедура `Draw` рисует только в памяти, поэтому мы не видим картинку. Чтобы это исправить, в обработчике `OnPaint` формы нужно выводить на холст формы битовую карту, что можно сделать с помощью метода `Canvas.Draw`:

```
Canvas.Draw ( 0, 0, bmp );
```

Первые два параметра — это координаты верхнего левого угла рисунка, третий — адрес рисунка в памяти.

Еще один метод, `Canvas.StretchDraw`, позволяет изменять размеры выводимого рисунка (растягивать или сжимать его). Его первый параметр — прямоугольник, в котором нужно вывести рисунок. Это объект типа `TRect`, его можно построить с помощью функции `Rect` (мы уже делали это при вызове метода `FillRect`).

Измените обработчик события `OnPaint` так, чтобы вывести на форму сам рисунок и его уменьшенную копию (справа):

```
if IsReady then begin
  Canvas.Draw ( 0, 0, bmp );
  Canvas.StretchDraw (Rect (300, 150, 390, 240), bmp);
end;
```

Запустите программу и щелкните по кнопке. Теперь сверните окно программы и разверните его заново. Попробуйте объяснить результат.

Итак, щелчок по кнопке не привел к появлению рисунка. Это наводит на мысль (и это действительно так!), что обработчик `OnPaint` не был вызван. Чтобы его вызвать, в обработчик `OnClick` кнопки нужно добавить вызов метода `Repaint` (перерисовать) для формы.

Добавьте в обработчик `OnClick` вызов метода

```
Repaint;
```

и проверьте работу программы. После этого можно закрыть проект.