

Глава 12. Графические возможности Delphi.....	262
12.1 Графическая система Windows	263
12.2 Первый пример работы с графикой.....	264
12.3 Свойства карандаша.....	265
12.4 Свойства кисти	269
12.5 Работа с текстом в графическом режиме	273
12.6 Вывод текста под углом	275
12.7 Работа с цветом	279
12.8 Методы объекта TCanvas	282
12.9 Компонент работы с графическими файлами (TImage)	285
12.10 Рисование на стандартных компонентах	290
12.11 Работа с экраном.	293
12.12 Режимы рисования.	295



Глава 12. Графические возможности Delphi.



Вот мы уже постепенно добрались и до 12-й главы моей книги. Здесь я расскажу тебе про то, как Delphi помогает упростить работу с графикой Windows. Эту тему можно было раскрыть даже немного раньше, потому что мы уже немного познакомились с основами и немного рисовали. В этой же главе я постараюсь всё расписать как можно подробнее.

Windows – это графическая оболочка и всё, что ты в ней видишь – это графика. Но для программиста большинство вещей очень сильно упрощены, особенно в Delphi, поэтому мы пока ещё не сильно сталкивались с графическими средствами. Но если ты соберёшься писать какой-нибудь большой проект, то обязательно столкнешься с проблемой рисования при оформлении определённых частей программы.

Хотя я очень мало читал книг на русском языке по Delphi (моя знания идут из англоязычных источников, мануалов и исходников), но в мои руки попадало несколько экземпляров на русском языке. Во всех из них тема графики обсуждается в середине или ближе к концу (в любом случае после описания баз данных). И везде эта тема считалась сложным материалом. С одной стороны материал действительно сложнее, чем базы данных, но не на столько, чтобы пугать читателя. В моей же книге графика будет обсуждаться именно здесь, потому что дальнейшее изучение кодинга (в том числе и баз данных) невозможно без понимания графических возможностей Delphi/Windows.

Так как Windows – это графическая оболочка, то дальнейшее обучение программированию невозможно без прочтения этой главы, так что не пропусти и прочти её полностью, даже если ты думаешь, что графика тебе не нужна.



12.1 Графическая система Windows

В предыдущих главах мы уже встречались с графикой Delphi и немного рисовали. Когда мы делали это, то обращались к объекту *Canvas*. Практически у всех компонентов Delphi есть это объектное свойство. Почему объектное? Да потому что *Canvas* имеет тип объекта *TCanvas*. То есть в нашем компоненте за рисование отвечает объект *TCanvas*. Так что если компонент поддерживает рисование, то у него обязано быть такое свойство.

Canvas в переводе с английского означает холст. Получается, что каждый компонент – это холст, на котором нарисовано изображение компонента. Взглянем на кнопку. На самом деле это не кнопка, а холст, на котором нарисовано изображение кнопки и текст. Когда ты щёлкаешь на кнопку, изображение изменяется и приобретает вид нажатой кнопки.

Графика Windows действительно похожа на рисование на холсте. А для такого рисования необходимо две вещи – карандаш (*Pen*) и кисть (*Brush*). Именно такие свойства и присутствуют у объекта *Canvas*. Карандаш используется для рисования линий и контуров, а кисть используется для закраски. У обоих есть свои свойства (цвет, тип и т.д.), но чтобы было понятнее, посмотри на рисунок 12.1.1:

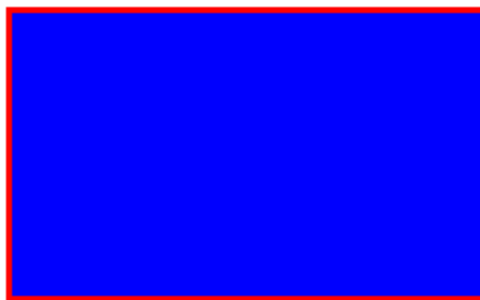


Рисунок 12.1.1 Простой прямоугольник

Это простой прямоугольник. Контур прямоугольника рисуется карандашом (в данном случае красного цвета). Центр прямоугольника закрашивается кистью (у нас синего цвета).

Но не надо думать, что всё приходится нудно рисовать по пикселям. В Windows для тебя уже заготовлено достаточно инструментов для облегчения процесса работы с графикой. Вообще, большое количество готовых инструментов – это самый большой козырь Windows. Благодаря этому Windows имеет такую популярность у программистов (легче кодить), а значит больше выходит программного обеспечения, а значит, пользователю удобно и есть выбор. Именно это сделало эту ОС популярной.

Графические инструменты Windows объединены под одним названием – GDI (Graphic Device Interface – интерфейс графических устройств). Все функции для работы с графикой находятся одной динамической библиотеке *gdi.dll*, но подробнее о библиотеках чуть позже.

Ещё одним плюсом GDI является то, что все функции аппаратно независимые. Это значит, что результат вывода графики будет одинаков вне зависимости от графического устройства (видео карты) установленной в компьютере. Ведь каждая карта имеет свои особенности и может работать специфично от других. Но для GDI всё это параллельно. Но тут же выскакивает и минусы GDI:

1. Он не использует ускорения;

2. Слишком медлителен;
3. Поддерживается только двумерная графика.

Все эти минусы отражаются на том, что GDI не предназначен для создания игр, за то хорошо подходит для офисных приложений. А вот игры хорошо создавать с помощью OpenGL или DirectX, но об этом тоже надо вести отдельный разговор, потому что тема эта слишком большая.

12.2 Первый пример работы с графикой

Давай попробуем написать простейший пример, в котором будет рисоваться простой квадрат. Но для усложнения дела, квадрат будем рисовать на форме и внутри компонента *TPaintBox*, который очень хорошо подходит для рисования.

Создай новое приложение, и помести на него компонент *PaintBox* с закладки *System*. Постарайся разместить этот компонент на нижней половине окна, как на рисунке 12.2.1.

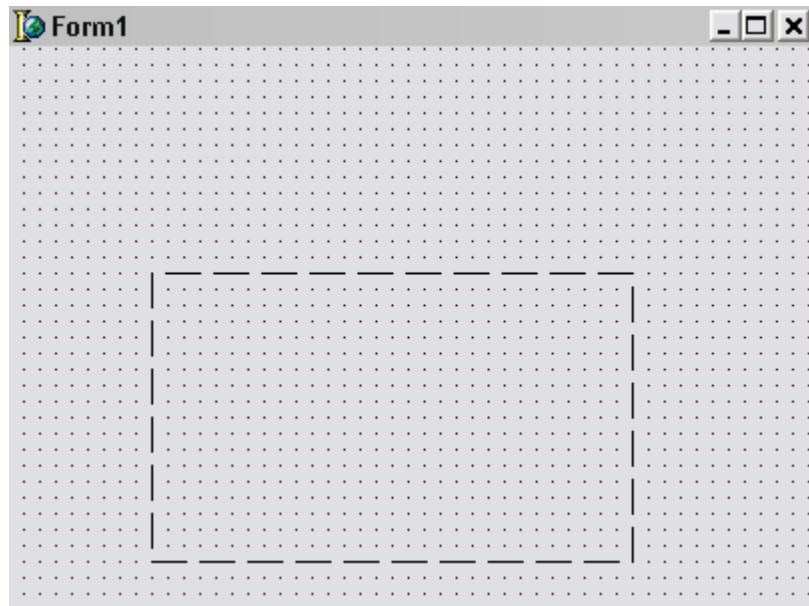


Рис 12.2.1. Форма будущей проги.

Что у формы, что у *PaintBox* есть свойство *Canvas*, значит, на них можно рисовать. Рисование лучше всего производить по событию *OnPaint*, которое так же есть у обоих компонентов. Итак, создадим обработчик события *OnPaint* для формы и напомним тут следующее:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle(10,10,100,100);
end;
```

Здесь я вызываю метод *Rectangle* объекта *Canvas* нашей главной формы. У этого метода четыре параметра:

1. Левая позиция квадрата;
2. Верхняя позиция квадрата;

3. Правая позиция;
4. Нижняя позиция.

Теперь выдели компонент *PaintBox* и создай такой же обработчик события *OnPaint* для этого компонента. В нём напишите следующее:

```
procedure TForm1.PaintBox1Paint(Sender: TObject);  
begin  
  PaintBox1.Canvas.Rectangle(10,10,100,100);  
end;
```

Здесь я вызываю тот же метод, с таким же параметрами, только для *PaintBox*. Это значит, что этот квадрат будет рисоваться уже внутри компонента *PaintBox*.

Попробуй запустить приложение, и ты увидишь два квадрата (см рисунок 12.2.2). Оба квадрата мы рисуем с помощью метода *Rectangle* с одними и теми же параметрами и по идее, они должны быть нарисованы в одном и том же месте. Но на деле это не так, потому что первый квадрат рисуется на форме, и координаты его отсчитываются относительно формы (10, 0, 100, 100), а второй внутри компонента и координаты отсчитываются относительно этого компонента (10, 0, 100, 100).

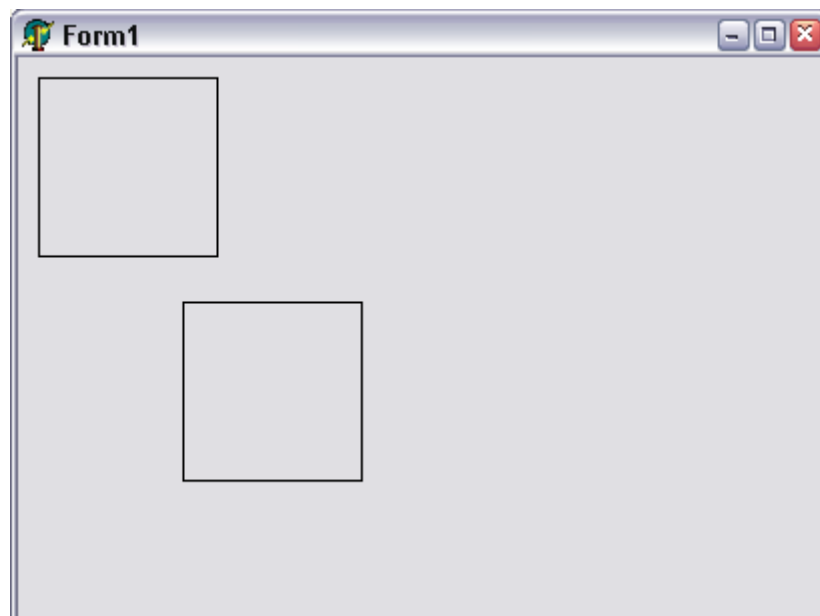



Рис 12.2.1. Форма будущей проги.

 На компакт диске, в директории \Примеры\Глава 12\Rectangle ты можешь увидеть пример этой программы.

12.3 Свойства карандаша

Теперь давай разберёмся с цветом. Как я уже сказал, для рисования используется два понятия цвета – цвет карандаша и цвет кисти. За стиль карандаша (в том числе и цвет) отвечает свойство *Pen* объекта *TCanvas*. За стиль кисти отвечает свойство *Brush*. И *Brush* и *Pen* – это тоже объекты, у которых есть свои свойства, о которых мы и поговорим в этой главе.

Для начала разберёмся с объектом **TPen**. Как я уже сказал, этот объект отвечает за свойства карандаша. У него есть следующие свойства:

Color – цвет карандаша.

Handle – здесь находится описание карандаша, которое можно использовать при обращении к WinAPI функциям. Вообще-то тебе пора уже запомнить, что у большинства объектов есть свойство *Handle*, которое нужно только для API функций и в повседневных программах мы его использовать не будем.

Mode – режим отображения показывает, как будет рисоваться линия.

Style – стиль карандаша. Существуют следующие стили (графическое отображение стилей линий ты можешь увидеть на рисунке 12.3.1):

- *psSolid* – сплошная линия;
- *psDash* – линия в виде пунктира (состоит из коротких линий);
- *psDot* – линия из точек;
- *psDashDot* – линия с чередующимися чёрточками и точками;
- *psDashDotDot* – линия с чередующимися чёрточками и двумя точками;
- *psClear* – невидимая линия;
- *psInsideFrame* – линия внутри формы. Внешне похожа на сплошную.

Width – ширина карандаша.

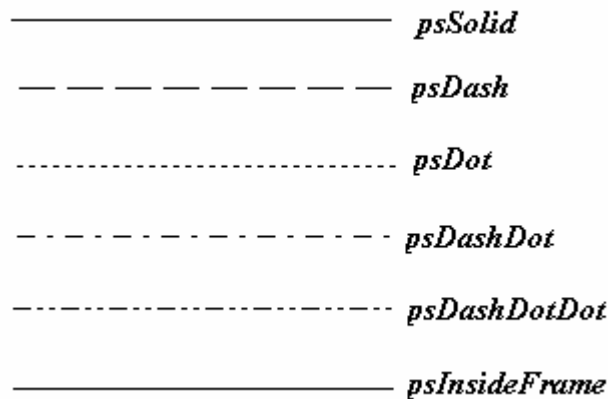


Рис 12.3.1 Стили линий

Теперь давай напишем пример, в котором увидим на практике свойства карандаша в действии. Создай новое приложение в Delphi. Создай обработчик события OnPaint и напиши в нём следующее:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  //Рисуем сплошную линию (psSolid)
  Canvas.Pen.Style:=psSolid;
  Canvas.MoveTo(10,20);
  Canvas.LineTo(200,20);

  //Рисуем psDash линию
  Canvas.Pen.Style:=psDash;
  Canvas.MoveTo(10,40);
  Canvas.LineTo(200,40);

  //Рисуем psDash линию
  Canvas.Pen.Style:=psDot;
  Canvas.MoveTo(10,60);
```

```
Canvas.LineTo(200,60);

//Рисуем psDashDot линию
Canvas.Pen.Style:=psDashDot;
Canvas.MoveTo(10,80);
Canvas.LineTo(200,80);

//Рисуем psDashDotDot линию
Canvas.Pen.Style:=psDashDotDot;
Canvas.MoveTo(10,100);
Canvas.LineTo(200,100);

//Рисуем psClear линию
Canvas.Pen.Style:=psClear;
Canvas.MoveTo(10,120);
Canvas.LineTo(200,120);

//Рисуем psInsideFrame линию
Canvas.Pen.Style:=psInsideFrame;
Canvas.MoveTo(10,140);
Canvas.LineTo(200,140);
end;
```

Результат работы программы ты можешь увидеть на рисунке 12.3.2.

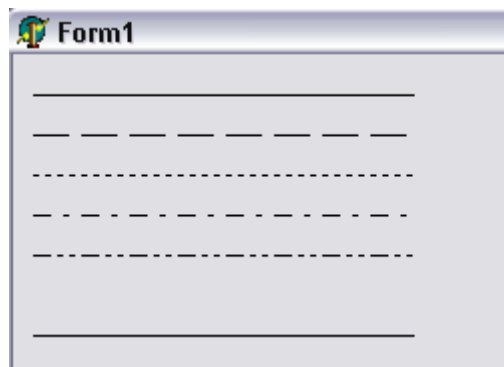


Рисунок 12.3.2 Результат работы программы.

В данном примере, по событию *OnPaint* (когда надо перерисовать форму) я поочерёдно рисую линии разного стиля. Для этого я сначала выбираю нужный стиль (например, *Canvas.Pen.Style:=psSolid* – выбирает стиль сплошной линии).

Потом я перемещаю карандаш в точку начала линии - *Canvas.MoveTo(X, Y)*. Метод *MoveTo* перемещает карандаш в позицию указанную в качестве параметров *X*, *Y*. При перемещении не происходит никакого рисования на холсте (*Canvas*). *X* и *Y* – это не сантиметры и не миллиметры, а количество пикселей (количество экранных точек).

Отсчёт координаты *X* идёт слева на право. Это значит, что левая сторона окна равна нулевой позиции *X*, а правая сторона окна – максимальное значение. Но это не значит, что *X* не может быть отрицательным или больше максимума. Ты без проблем можешь указывать любые значения, только нужно учитывать, что часть линии может уйти за пределы окна.

Отсчёт координаты *Y* идёт сверху вниз. Это значит, что верхнее обрамление окна является нулевой точкой *Y*. При этом заголовок окна (с названием формы и системными кнопками) не входит в пространство окна.

Теперь я должен нарисовать линию с помощью метода *LineTo(X, Y)*. В качестве параметров передаются координаты линии. Отрезок будет нарисован, начиная от текущей позиции карандаша, куда мы перешли с помощью метода *MoveTo* и до координат, указанных при вызове метода *LineTo*.

После прорисовки первой линии, я выбираю следующий стиль и перемещаюсь в позицию на 20 пикселей ниже уже нарисованной линии и рисую следующую линию.

Теперь добавим в нашу программу возможность смены цвета карандаша. Для этого бросим на форму кнопку с надписью «Изменить цвет» и компонент *ColorDialog* с закладки *Dialogs*. Компонент *ColorDialog* предназначен для отображения стандартного диалога выбора цвета. На форме он будет выглядеть в качестве простого квадратика с пиктограммой и при запуске не будет виден. Обновлённую форму ты можешь увидеть на рисунке 12.3.3.

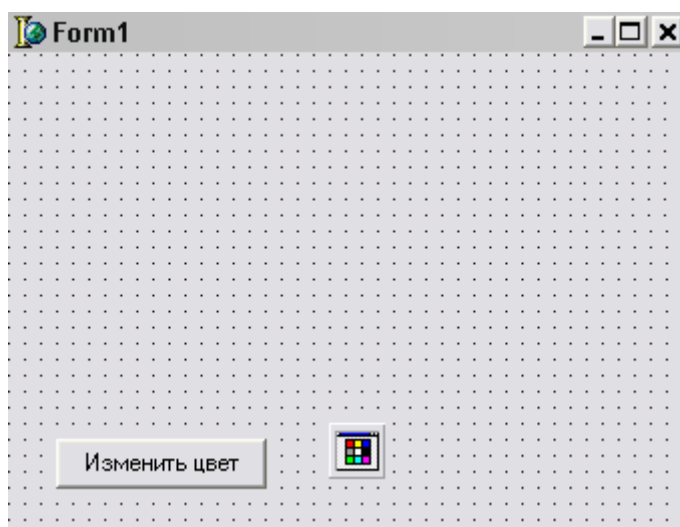


Рисунок 12.3.3 Обновлённая форма будущей программы

По событию *OnClick* для кнопки пишем:

```
if ColorDialog1.Execute then  
  Canvas.Pen.Color:=ColorDialog1.Color;  
  FormPaint(nil);
```

В первой строке я отображаю окно выбора цвета (*ColorDialog1.Execute*). Если пользователь выбрал цвет, а не нажал «Отмена», то окно возвращает значение *true*, поэтому я проверяю, если результат показа окна равен *true*, то изменить цвет:


```
if ColorDialog1.Execute then  
  Изменить цвет холста
```

Напоминаю, что по умолчанию конструкция **if** проверяет указанный код на равенство *true* если не указано обратное. Поэтому эту же конструкцию можно было бы записать так:


```
if ColorDialog1.Execute=true then  
    Изменить цвет холста
```

Результат выбранного цвета записывается в свойство *Color* компонента *ColorDialog1*. Именно его мы и присваиваем цвету карандаша *Canvas.Pen.Color*. После этого нужно только перерисовать рисунок. Для этого я явно вызываю процедуру обработчик события *OnPaint* формы. У нас обработчик называется *FormPaint*, именно его я и вызываю.


Можешь запустить программу и проверить результат работы смены цветов линий.

Теперь добавим возможность выбора толщины линии. Для этого бросим компонент *UpDown*  с закладки *Win32*. По событию *OnClick* по этому компоненту напишем следующий код:

```
procedure TForm1.UpDown1Click(Sender: TObject; Button: TUDBtnType);  
begin  
    Canvas.Pen.Width:=UpDown1.Position;  
    Repaint;  
end;
```

Напоминаю, что компонент *UpDown* состоит из двух кнопок – верхняя увеличивает внутренний счётчик, а нижняя уменьшает. Текущее значение счётчика можно прочитать в свойстве *Position*. Именно это значение я и присваиваю в свойство ширины карандаша *Canvas.Pen.Width*.

После этого я вызываю метод главной формы *Repaint*. Этот метод генерирует событие о том, что надо перерисовать содержимое окна. Это значит, что будет автоматически вызван обработчик события *OnPaint*. Результат – тот же, что и просто вызов напрямую обработчика, как мы это делали после смены цвета, но такой способ считается более правильным. Я в своих программах пользуюсь обоими способами, и отдать предпочтение одному из них не могу. Если говорить о том, какой способ более правильный, то оба они работают без проблем, просто второй способ более эстетичный и красивый, хотя и требует дополнительных затрат на генерацию сообщения о необходимости перерисовать окно.

 На компакт диске, в директории \Примеры\Глава 12\Pen ты можешь увидеть пример этой программы.

12.4 Свойства кисти

За параметры кисти отвечает свойство *Brush* объекта *TCanvas*. Как я уже говорил, кисть используется для закраски замкнутых пространств. Она тоже имеет объектный тип как и карандаш, а значит обладает своими свойствами и методами.

У объекта кисти *TBrush* есть несколько свойств влияющих на параметры кисти:

Bitmap – картинка, которая будет использоваться в качестве фона закрашки. Картинка должны быть формата 8x8 пикселей. Если будет больше, то задействованы будут только пиксели верхнего левого квадрата 8x8.

Мы пока не трогали картинки, поэтому я это свойство рассматривать пока не буду. Единственное, что я сделаю – приведу небольшой кусок кода, а потом ты сможешь вернуться к нему и разобраться самостоятельно:

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create; //Создаётся картинка
  try
    Bitmap.LoadFromFile('MyBitmap.bmp'); //Загружается картинка
    Form1.Canvas.Brush.Bitmap := Bitmap; //Присваивается в качестве фона
    Form1.Canvas.Rectangle(0,0,100,100); // Рисуется квадрат
  finally
    Form1.Canvas.Brush.Bitmap := nil; // Обнуляется фон
    Bitmap.Free; // Уничтожается картинка
  end;
end;
```

Color – так же как и у карандаша, у кисти тоже может быть свой цвет.

Handle – такой же указатель, как и у карандаша, но на кисть.

Style – стиль фона. Здесь могут быть следующие значения: bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross. На рисунке 12.4.1 ты можешь увидеть графическое отображение каждого из стилей.

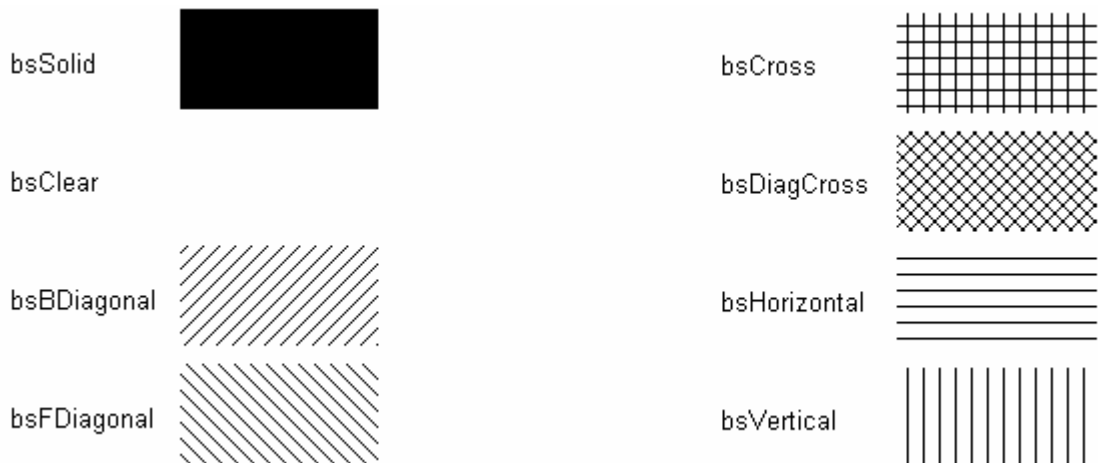


Рисунок 12.4.1 Стили фона

Теперь перейдём к практической части работы с кистью и напомним небольшой пример. Создай новый проект и давай приступим к кодированию.

Как и в прошлом примере, давай создадим обработчик события OnPaint для формы, чтобы по этому событию производить рисование. В обработчике напомним следующее:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Brush.Style:=bsSolid;
```

```
Canvas.Rectangle(10,10,50,50);

Canvas.Brush.Style:=bsBDiagonal;
Canvas.Rectangle(10,110,50,150);

Canvas.Brush.Style:=bsFDiagonal;
Canvas.Rectangle(10,160,50,200);

Canvas.Brush.Style:=bsCross;
Canvas.Rectangle(110,10,150,50);

Canvas.Brush.Style:=bsDiagCross;
Canvas.Rectangle(110,60,150,100);

Canvas.Brush.Style:=bsHorizontal;
Canvas.Rectangle(110,110,150,150);

Canvas.Brush.Style:=bsVertical;
Canvas.Rectangle(110,160,150,200);

Canvas.Brush.Style:=bsClear;
Canvas.Rectangle(10,60,50,100);
end;
```

Здесь код разбит на блоки по две строчки. В первой строчке я задаю стиль кисти, а во второй рисую прямоугольник с помощью метода `Rectangle(x, y, r, b)`, где:

x – левая сторона прямоугольника;
y – верхняя сторона прямоугольника;
r – правая сторона прямоугольника;
b – нижняя сторона прямоугольника.

Чтобы было более ясно, взгляни на рисунок 12.4.2.

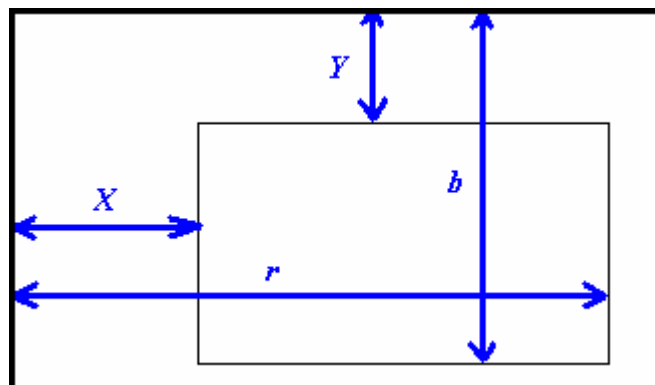


Рисунок 12.4.2 Размеры прямоугольника

Таким образом, я рисую восемь прямоугольников с разными стилями кисти. Если ты запустишь сейчас этот пример, то не заметишь никакой разницы, все прямоугольники будут одинаковыми. Это потому что сейчас цвет кисти имеет такой же цвет, что и форма, поэтому всё сливается. Чтобы увидеть разницу надо изменить цвет фона кисти.

Давай бросим на форму кнопку «Изменить цвет» и компонент *ColorDialog* и по нажатию на неё напишем:

```
if ColorDialog1.Execute then
  Canvas.Brush.Color:=ColorDialog1.Color;
```

FormPaint(nil);

Здесь я запускаю окно изменения цвета, и если цвет выбран, то присваиваю его кисти `Canvas.Brush.Color:=ColorDialog1.Color`.

Вот теперь можешь запускать программу и смотреть результат. Щёлкни по кнопке «Изменить цвет» и выбери что-нибудь из тёмных, например, синий. Результат смотри на рисунке 12.4.3.

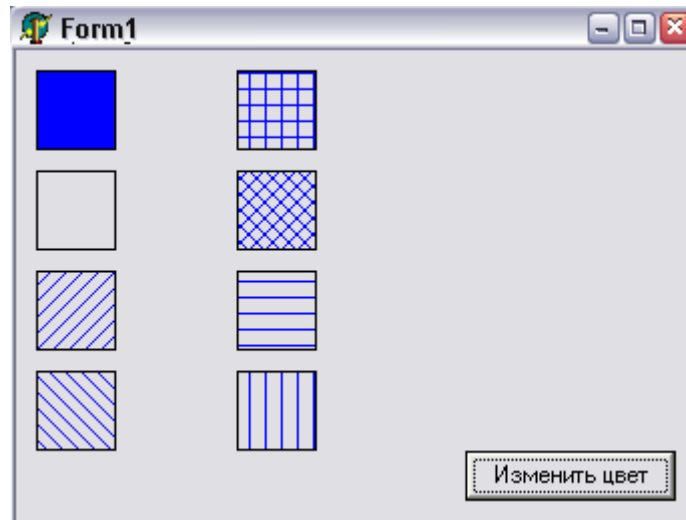


Рисунок 12.4.3 Результат работы программы

Теперь пару замечаний:

1. Заметь, что прямоугольник с невидимой кистью (`Style=bsClear`) я рисую последним, хотя на форме он расположен во второй строке первой колонки. Это связано с тем, что после рисования с невидимой кистью цвет теряется. Попробуй поставить рисование с невидимой кистью где-нибудь раньше, и ты увидишь, что до прямоугольника с прозрачной кистью фон будет того цвета, что ты выбрал, а после будет белого цвета (рисунок 12.4.4).
2. Если программу свернуть и потом развернуть, то цвет кисти опять же будет белого цвета. Это связано с тем, что когда мы нарисовали последний квадрат (который был с прозрачным фоном) цвет кисти всё же изменился на белый. Мы этого не увидели, потому что последний квадрат был с прозрачным фоном. При следующей прорисовке цвет уже изначально белый. Чтобы избавиться от этого эффекта, после каждого рисования с фигур прозрачной кистью надо устанавливать цвет. Для большей надёжности желательно устанавливать цвет непосредственно перед рисованием.

Так как у нас после рисования с прозрачным фоном больше не будет вывод на экран, то тут уже нет смысла устанавливать цвет. А вот перед началом рисования это делать всё же желательно:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  //Обязательно устанавливаю цвет кисти
  Canvas.Brush.Color:=ColorDialog1.Color;

  //Рисую первый квадрат
```

```
Canvas.Brush.Style:=bsSolid;  
Canvas.Rectangle(10,10,50,50);
```

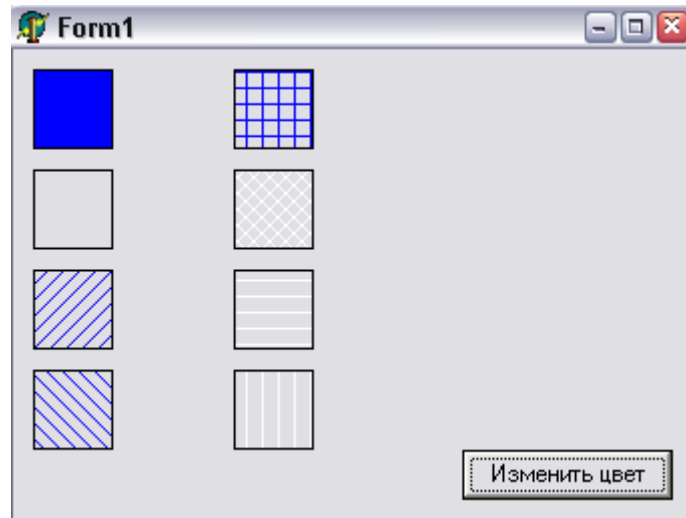


Рисунок 12.4.4 Последние три прямоугольника имеют белый цвет кисти, потому что прямо перед этим, я нарисовал прямоугольник с прозрачным фоном.

 На компакт диске, в директории \Примеры\Глава 12\Brush ты можешь увидеть пример этой программы.

12.5 Работа с текстом в графическом режиме

Конечно же, название этой части достаточно расплывчато и не точно, потому что Windows сам по себе графический и вся работа сама по себе уже графическая. Но всё же иногда мы работаем с текстом, воспринимая его как текст, а иногда мы прямо выводим его в виде графики.

Для вывода текста на экран у объекта **TCanvas** есть метод *TextOut*. У этого метода три параметра:

1. **X** позиция текста;
2. **Y** позиция текста;
3. Непосредственно строка текста, которую надо вывести.

Создай новое приложение и по событию *OnPaint* напиши:

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
  Canvas.TextOut(100,100, 'Привет всем!!!');  
end;
```

Здесь я просто вывожу на экран текст в координатах (100, 100).

За стиль шрифта отвечает свойство *Font* объекта **TCanvas**. Это свойство тоже имеет объектный тип (**TFont**), у которого очень много свойств. Среди них, конечно же, есть и свойство *Color*, так что давай бросим на форму кнопку и *ColorDialog*, чтобы можно было менять цвет текста.

По событию *OnClick* для кнопки пишем следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ColorDialog1.Execute then
    FormPaint(nil);
end;
```

Здесь я показываю окно выбора цвета и если цвет выбран, то просто перерисовываю окно. А где же изменение цвета шрифта? Я же уже сказал, когда мы работали с кистью, что цвет нужно менять непосредственно перед рисованием, и здесь это делать нет смысла. Поэтому корректируем событие *OnPaint*:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Font.Color:=ColorDialog1.Color;
  Canvas.TextOut(100,100, 'Привет всем!!!');
end;
```

Теперь бросим на форму компонент *FontDialog* с закладки *Dialogs*. Это почти такой же компонент, как мы использовали для смены цвета, только здесь будет появляться стандартное окно смены шрифта. Добавь на форму кнопку с надписью «Изменить шрифт» и по её нажатию напиши следующее:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if FontDialog1.Execute then
    Canvas.Font:=FontDialog1.Font;
    FormPaint(nil);
end;
```

В первой строчке я показываю окно смены шрифта так же, как это делалось для смены цвета. Если пользователь выбрал шрифт и нажал «ОК», то я устанавливаю его свойству *Font* объекта *Canvas*. После этого я заставляю форму обновить своё содержимое. При новой прорисовке содержимого формы, текст уже выводиться с новым шрифтом.

Единственный недостаток этого примера – если сначала выбрать большой шрифт, а затем маленький, то старое содержимое текста, написанного большим шрифтом, не уничтожается, а новый рисуется поверх старого (см рисунок 12.5.1).

Самый простейший способ избавиться от такого нежелательного эффекта – после смены шрифта вместо прямого вызова функции *FormPaint(nil)* использовать вызов метода формы *Repaint* или *Invalidate*. О первом из них я уже говорил, и мы его уже использовали. Второй метод *Invalidate* имеет практически тот же смысл, только заставляет прорисоваться полностью всё окно, а не только его содержимое.

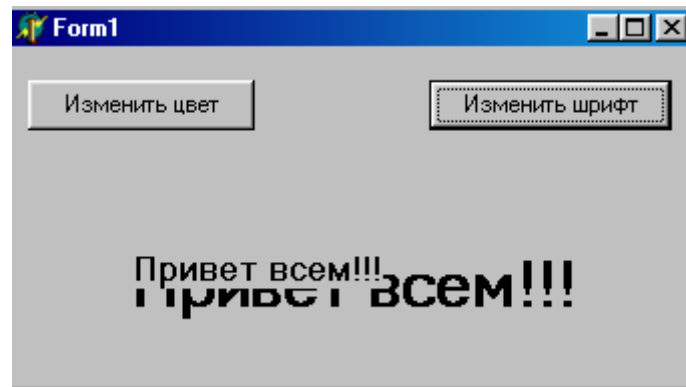


Рисунок 12.5.1 Ненужный эффект наложения старого текста на новый. Маленький новый текст рисуется поверх старого большого.

Итак, мы научились менять все параметры текста с помощью стандартного диалогового окна. Как менять отдельные свойства – это отдельная тема, ведь свойство *Font* имеет объектный тип **TFont**, с массой свойств – имя шрифта, стиль, цвет, размер. Но всё это отдельная тема и мы её пока опустим.

 На компакт диске, в директории \Примеры\Глава 12\Text ты можешь увидеть пример этой программы.

12.6 Вывод текста под углом

Сейчас я хочу показать тебе один трюк – как можно вывести текст под углом. Как ты мог заметить, пока мы увидели только функцию, которая умеет выводить текст горизонтально и никаких больше намёков на возможность развернуть текст и написать его например вертикально.

Создай новый проект. Теперь создай обработчик события *OnCreate* для главной формы. В этой процедуре напиши следующее:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  index:=0;
end;
```

Index - это у нас будет счётчик, но его ещё надо объявить, поэтому иди в объявления *private* и напиши следующее:

```
private
{ Private-Deklarationen }
index:Integer;
cl:Boolean;
```

Теперь мы познакомимся с типами данных Delphi. Мы объявили с тобой две переменные: *index* и *cl*. Первая из них - это целое, знаковое число. Вторая - это булево, и может принимать значения **true** или **false**.

Основные приготовления закончены, и мы можем переходить непосредственно к программированию. Создай обработчик события для главной формы *OnMouseDown*.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
Var
  A: Integer; // Объявление переменной A - целое число.
begin
  A := random(3600);
  CanvasSetAngle(Canvas, A / 10);
  Canvas.TextOut(X, Y, FormatFloat('##0.0', A/10)+'°');
end;
```

Давай рассмотрим текст процедуры. В первой строчке мы используем функцию *random*, она возвращает случайное значение, но не больше чем число, указанное в скобках. В нашем случае - это 3600.

Вторую строчку я опущу, а рассмотрим сразу третью. *Canvas.TextOut* - выводит текст на форме и мы с ней уже работали.

В качестве текста я опять использовал процедуру: *FormatFloat*. Эта процедура переводит число с запятой (вещественное, или так сказать дробное) с учётом формата.

```
function FormatFloat(
  const Format: string; // Строка формата
  Value: Extended // Число
): string;
```

В качестве формата я указал **##0.0**, что приводит указанное число к этому виду, т.е. отрезает все числа после запятой, оставляя только одно (об этом говорит один ноль после запятой в строке формата). Перед запятой может быть любое количество чисел, потому что стоит два знака решётки. В качестве числа я указываю переменную **A** делённую на 10.

Теперь возвращаемся ко второй строке. *CanvasSetAngle* - этой процедуры ещё нет, мы её должны написать. Я сейчас приведу весь текст программы, а потом мы рассмотрим эту процедуру отдельно.

```
unit Textrot1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls;
type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private-Deklarationen }
    index: Integer;
    cl: Boolean;
  public
```



```
{ Public-Deklarationen }
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure CanvasSetAngle(C: TCanvas; A: Single);
var
  LogRec: TLOGFONT; // Объявляем переменную логического шрифта
begin
  GetObject(C.Font.Handle, SizeOf(LogRec), Addr(LogRec));
  LogRec.lfEscapement := Trunc(A*10);
  LogRec.lfOrientation := Trunc((A+10) * 100);
  C.Font.Handle := CreateFontIndirect(LogRec);
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
Var A: Integer;
begin
  A := Random(3600);
  CanvasSetAngle(Canvas, A / 10);
  Canvas.TextOut(x, Y, FormatFloat('##0.0', A/10)+'°');
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  index:=0;
  Canvas.Brush.Style:=bsClear;
end;
end.
```

В качестве параметров для функции *CanvasSetAngle* мы передаём процедуре *Canvas* и угол разворота текста. Угол разворота - имеет значение *Single* - что означает вещественное (дробное). До этого имя процедур вместо нас писал *Delphi*. Эту процедуру тебе придётся вписывать своими руками, потому что она самостоятельная и не принадлежит никакому объекту

Теперь перейдём к содержимому процедуры. Рассмотрим по частям первую строчку.

GetObject // Это функция возвращает информацию о графическом объекте
C.Font.Handle // Объект на который нужно получить значение.
SizeOf(LogRec) // Передаем размер возвращаемого значения
Addr(LogRec) // Передаём адрес возвращаемого значения

С помощью этой функции, мы получаем информацию о шрифте, используемом нами для рисования. Вторая и третья строчки этой процедуры изменяют значения полученной информации. Четвёртая функция записывает изменённую информацию.

Запусти получившееся приложение, и пощёлкой по форме мышкой. В месте щелчка должен появиться текст под случайным углом. Когда наиграешься, закрой программу и я продолжу.

Теперь поставь на форме `Timer`, который находится в закладке `System`. Создай для таймера обработчик события `OnTimer` и напиши в нём следующее:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Canvas.SetAngle(Canvas, index);
  Canvas.TextOut(100, 100, 'CyD Soft');
  index:=index+45;
  if index>=360 then
  begin
    index:=0;
    if cl then
      Canvas.Font.Color:=clBlack
    else
      Canvas.Font.Color:=clRed;
    cl:=not cl;
  end;
end;
```

Эта процедура будет вызываться каждый раз, когда пройдёт интервал времени указанный в свойстве *Interval* компонента *Timer*. По умолчанию там указано 1000 (число в миллисекундах, что равно 1 секунде), значит, процедура будет вызываться через каждую секунду.

Внутри процедуры я выставляю угол, на который надо вывести текст (*Canvas.SetAngle*), потом вывожу текст (с помощью *Canvas.TextOut*).


После этого я прибавляю переменной *index* значение 45. В этой переменной храниться значение угла, под которым надо вывести текст и через каждую секунду это значение увеличивается на 45 градусов.

Далее идёт проверка - если *index* больше или равен 360, значит мы прошли полный круг, а если так, то выполняется следующий код:

```
index:=0;
if cl then
  Canvas.Font.Color:=clBlack
else
  Canvas.Font.Color:=clRed;
cl:=not cl;
```

Здесь в первой строке я обнуляю переменную *index*, чтобы начать вывод текста с 0 градусов (мы же уже прошли полный круг). Далее я проверяю значение переменной *cl* если она равна *true*, то значению цвета текста будет присвоено *clBlack*, что равно чёрному цвету. Иначе цвет смениться на красный *clRed*.

После этого я изменяю значение переменной *cl* на противоположное, о чём говорит конструкция *cl:=not cl*. Здесь я присваиваю переменной противоположное (*not*) значение её самой. Это значит, что если *cl* равнялась *true*, то после этого кода будет равняться *false*. Так что после прохождения текстом очередного круга, цвет будет меняться с чёрного, на красный и обратно.

 На компакт диске, в директории \Примеры\Глава 12\TextAngle ты можешь увидеть пример этой программы.

Попробуй запустить пример. Пускай он немного поработает, чтобы форма заполнилась текстом. Теперь попробуй перекрыть окно программы другим окном или свернуть его. Потом снова восстанови окно. Что ты видишь? Всё, что было нарисовано - исчезло. Это потому что Windows не сохраняет содержимое окна. Мы сами должны его восстанавливать.

Когда окно свернулось и восстановилось, то генерируется событие *OnPaint*, по которому нужно перерисовать содержимое окна. Поэтому все функции рисования стараются располагать именно в обработчике этого события. Так мы будем рисовать и реагировать на события когда надо перерисовать содержимое экрана одной и той же функцией.

В нашем примере использование события *OnPaint* неудобно. В таких случаях на форму ставят компонент *TImage* и рисуют в нём. Этот компонент, в отличие от формы сохраняет своё содержимое. Когда ты рисуешь на холсте компонента *TImage*, то ты рисуешь в специально отведённой памяти. А вот когда компонент *TImage* нуждается в прорисовке, то эта область памяти копируется на сам компонент. Таким образом не надо самостоятельно ни за чем следить. Подробнее с компонентом *TImage* мы познакомимся немного позже.

12.7 Работа с цветом

Мы уже научились менять цвет и даже в предыдущей части узнали, что константы `clBlack` равна чёрному цвету, а `clRed` красному. Но есть ещё много констант, которые определяют стандартные цвета для более удобного использования. Вот именно с ними нам предстоит сейчас познакомиться и узнать, как храниться цвет в памяти машины.

Цвет храниться в виде типа *TColor*. Хотя в названии типа в начале стоит буква **T**, этот тип не объектный, а просто число из 4-х байт, хотя реально нас будут интересовать только последние три.

Ты наверно должен знать, что в компьютерной графике цвет представляется тремя составляющими красного, зелёного и голубого (RGB). В разных пропорциях, из этих трёх цветов можно получить любой другой. Например, если взять красного и зелёного по максимуму, а синего вообще не брать, то получится жёлтый цвет.

Каждый из цветов представляется в виде одного байта, так что для хранения трёх цветов достаточно 3 байтов. Но зачем же тогда для *TColor* выделено 4-е байта? Да потому что в компьютере регистры чётные и могут хранить только 1, или 2, или 4 байта. Так что у переменной цвета один байт избыточен (первый) и чаще всего равен нулю. В играх и графических пакетах этому байту нашли применение – он часто указывает прозрачность, но в офисных приложениях его просто игнорируют.

Как ты уже знаешь, один байт может принимать значения от 0 до 255 (в десятичной форме) или от 0 до FF (в шестнадцатеричной). Так что в шестнадцатеричной форме цвет будет выглядеть как \$00FFFFFF. Только тут сразу надо отметить, что первые два нуля – это лишний байт, потом идут FF для голубого цвета, потом FF для зелёного и последние FF для красного. Получается, что в памяти цвет храниться как BGR (в обратном порядке). Абсолютно красный цвет будет равен \$000000FF, абсолютно зелёный = \$0000FF00, а голубой - \$00FF0000.

Давай попробуем научиться работать с цветом на практике, заодно и познакомимся с необходимыми функциями. Создай новое приложение и брось на него компоненты так, как показано у меня на рисунке 12.7.1.

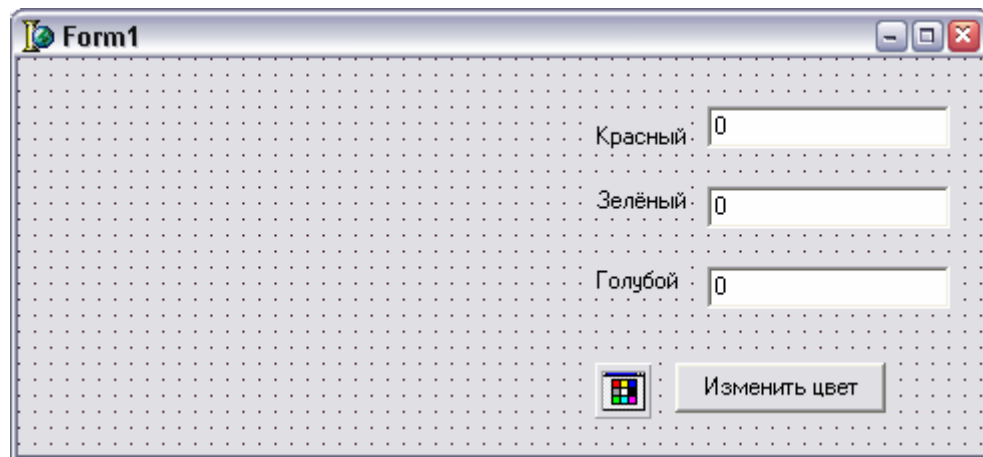


Рисунок 12.7.1 Форма будущей программы

Итак, у нас на форме три компонента *TEdit*. Для красного я компонент назвал *RedEdit*, для зелёного *GreenEdit*, ну и для синего *BlueEdit*. Так же на форме есть кнопка для смены цвета (её имя не имеет значения) и *ColorDialog*, для смены цвета.

Если ты сам создаёшь пример, а не пользуешься готовым с диска, то постарайся всё разместить так, как у меня на рисунке (ближе к правому краю), потому что слева мы будем рисовать квадрат.

По нажатию кнопки пишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  c:LongInt;
begin
  if not ColorDialog1.Execute then
    exit;

  C:=ColorToRGB(ColorDialog1.Color);
  RedEdit.Text:=IntToStr(GetRValue(C));
  GreenEdit.Text:=IntToStr(GetGValue(C));
  BlueEdit.Text:=IntToStr(GetBValue(C));

  Repaint;
end;
```

В разделе **var** объявлена одна переменная целого типа *LongInt*. Это целое число размером в 4-е байта и будет использоваться для хранения значения цвета.

В первой строчке я показываю окно смены цвета *ColorDialog1.Execute*. Если пользователь не выбрал цвет (об этом говорит конструкция **if not**), то мы прерываем выполнение процедуры с помощью выхода из неё - *exit*.

Дальше я преобразовываю выбранный цвет *ColorDialog1.Color* из типа *TColor* в простое число с помощью функции *ColorToRGB*. Этой функции надо передать цвет в виде *TColor* (мы передаём *ColorDialog1.Color*) и она вернёт целое 4-х байтное число, которое я записываю в переменную *C*.

В следующей строке я присваиваю строке ввода *RedEdit* значение красной составляющей цвета. Для этого я сначала использую функцию *GetRValue*. Ей я передаю значение цвета в виде целого числа (переменная *C*). Результат – однобайтное число, которое показывает значение красной составляющей. Этот результат – число и прежде

чем его присваивать в строку ввода, его надо преобразовать в строку. Для этого я превращаю его в текст с помощью знакомой нам функции *IntToStr*.

То же самое я проделываю и с зелёным цветом в следующей строке кода. Только для получения зелёной составляющей я использую функцию *GetGValue*.

Для получения синей составляющей, я использую функцию *GetBValue*. Таким образом, после выполнения таких действий, я разбил 4 байта цвета из переменной *C*, на отдельные байты по цветам и разнёс их в соответствующие строки ввода.

После этого я заставляю окно прорисоваться *Repaint*.

По событию *OnPaint* напишем следующий код:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Brush.Color:=RGB(StrToIntDef(RedEdit.Text, 0),
    StrToIntDef(GreenEdit.Text, 0), StrToIntDef(BlueEdit.Text,0));
  Canvas.Rectangle(10,10, 250, 150);
end;
```

Здесь мне надо проделать обратные действия – превратить три составляющих цвета из строк ввода в одно целое значение цвета. Для этого используется функция *RGB(R, G, B)*. У этой функции три параметра и все они целые числа:

R – значение красного цвета;

G – значение зелёного цвета;

B – значение синего цвета.

В качестве параметров я передаю значения указанные в соответствующих строках ввода, предварительно преобразовывая их из строк в числа. Чтобы было яснее, в приведённом выше коде я раскрасил код соответствующими цветами.

Результат преобразования цвета я записываю в цвет кисти. После этого я рисую прямоугольник, у которого цвет фона будет тот, что мы выбрали.

И последнее что мы сделаем – создадим обработчик события *OnChange* для всех строк ввода. Выдели сначала строку ввода для красного цвета, потом удерживая *Shift* щёлкни по остальным. У тебя должны быть выделены все строки ввода серыми рамками. Теперь перейди в объектном инспекторе на закладку *Events* и дважды щёлкни по *OnChange*, чтобы создать обработчик. В нём напиши:

```
procedure TForm1.RedEditChange(Sender: TObject);
begin
  Repaint;
end;
```

Попробуй запустить этот пример. Теперь выбери какой-нибудь цвет, и ты увидишь составляющие этого цвета. Можешь даже напрямую изменять значения в строках ввода, и результат моментально будет отражаться на цвете прямоугольника. Только помни, ни одна из составляющих не может быть больше 255!!!

На рисунке 12.7.2 ты можешь увидеть пример работы моей программы. Потренируйся с ним и возможно ты поймёшь то, что не понял из моих слов. Словами можно объяснить многое, а практика показывает ещё больше.

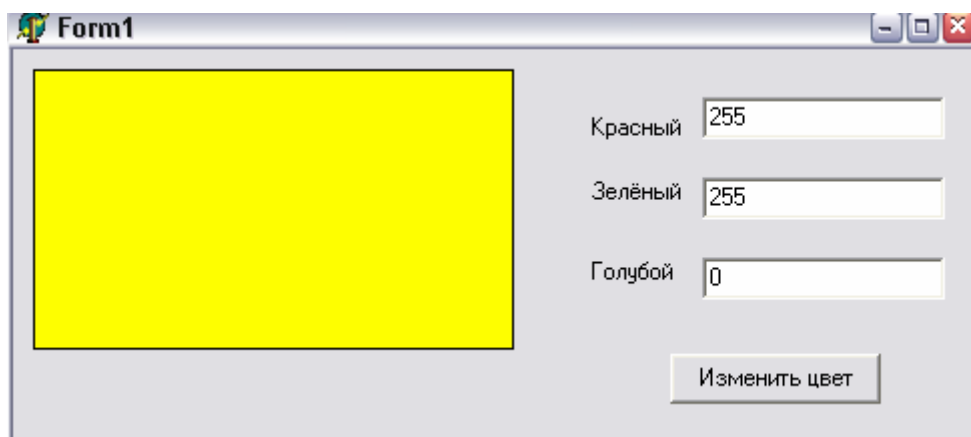



Рисунок 12.7.2 Результат работы программы

 На компакт диске, в директории **\Примеры\Глава 12\Color** ты можешь увидеть пример этой программы.

Ну а теперь познакомимся с константами, которые уже заготовлены для основных цветов. Ты можешь их реально использовать в своих программах и присваивать, как в примере из главы 12.6. Я не буду перечислять все константы, потому что ты можешь их сам в любой момент найти, если щёлкнешь в объектном инспекторе по свойству *Color* любого компонента. Всё, что ты там увидишь в списке, это и есть константы, которые можно использовать. Я сам всегда пользуюсь этим методом, потому что сразу вижу константу и цвет.

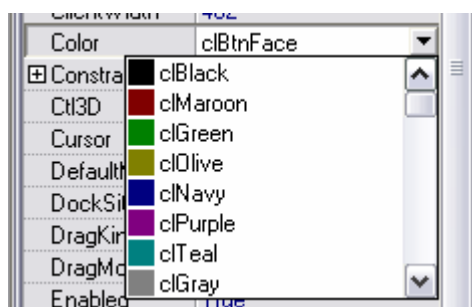


Рисунок 12.7.3 Цветовые константы

12.8 Методы объекта TCanvas

Последнее, что нам надо узнать в этой части книги — познакомиться с методами рисования. Пока что мы использовали только линии, прямоугольники и текст, но на этом возможности объекта **TCanvas** не заканчиваются.

Pixels

Первое, чем мы познакомимся, не будет методом — это свойство *Pixels*. Это свойство — двухмерный массив, указывающий на битовую матрицу изображения. Что это значит? Проще всего показать. Допустим, что тебе нужно поставить точку чёрного цвета в координатах (10,10). Для этого ты пишешь следующий текст:

```
Canvas.Pixels[10,10]:=Black;
```

С помощью этого же свойства можно узнать цвет в какой-нибудь точке. Например:

```
var  
  c:TColor;  
begin  
  c:=Canvas.Pixels[10,10];  
  if c=clBlack then  
    //Точка с координатами (10, 10) чёрного цвета  
  End;
```

TextWidth и TextHeight

Давай сразу познакомимся ещё с двумя методами для работы с текстом – *TextWidth* и *TextHeight*. Обоим методам нужно передать какой-нибудь текст и первый из них вернёт его ширину, а второй – высоту в пикселях. Эти метода очень удобны, когда тебе нужно выводить форматированный текст.

Допустим, что тебе надо вывести две строки текста. Ты можешь вывести одну из них, а потом чуть ниже вывести другую. А вот теперь самое интересное – на сколько ниже? Ведь если взять слишком много, то будет большой промежуток, а если слишком мало, то одна строка наедет на другую или попросту её перекроет. Эти методы позволяют точно узнать длину или высоту текста, в зависимости от используемого шрифта.

Используй их каждый раз, когда это нужно и не надейся на собственные расчёты, потому что шрифт на разных машинах может выглядеть по-разному.

Arc

Следующий метод объекта *TCanvas* – это метод *Arc*, который предназначен для рисования дуги. У него аж 8 параметров - X1, Y1, X2, Y2, X3, Y3, X4, Y4. Как видишь, это 4 пары координат X и Y, которые указывают 4 точки через которые надо провести дугу.

CopyRect

Этот метод предназначен для копирования указанного прямоугольника из одного объекта *TCanvas* в другой. У этого метода три параметра:

Dest: TRect – область, указывающая, куда надо копировать;

Canvas: TCanvas – это объект, из которого надо копировать;

Source: TRect - область, указывающая, откуда надо копировать.

Первый и последний параметр имеют тип *TRect*. Это простая структура из четырёх целых чисел - *Left*, *Top*, *Right*, *Bottom*. Не трудно догадаться, что это координаты прямоугольника. Для создания переменной такого типа лучше всего использовать функцию *Rect*. Ей нужно передать четыре этих параметра *Left*, *Top*, *Right*, *Bottom* и она вернёт вам готовую структуру.

Давай рассмотрим пример и увидим всё на практике. Допустим, что у нас есть две формы. Мы хотим из второй формы *Form2* скопировать всё её содержимое в первую форму *Form1*. При этом мы отобразим содержимое второй формы на первой в

прямоугольнике размером (10, 10, 110, 110), т.е. На рисунке 12.8.1 показано графически, куда мы будем копировать.

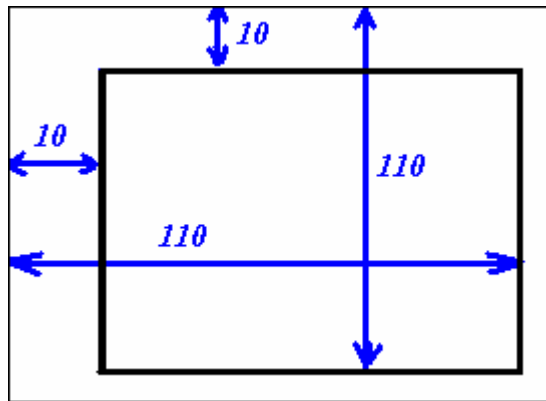


Рис 12.8.1 Область копирования

Этот код будет выглядеть так:

```
var
  SRect, DRect: TRect; // Объявляю две переменные типа TRect
begin
  SRect:=Rect(0, 0, Form2.Width, Form2.Height);
  DRect:=Rect(10, 10, 110, 110);
  Form1.Canvas1.CopyRect(DRect, Form2.Canvas, SRect);
end;
```

В первой строке я заполняю структуру *SRect* с помощью функции *Rect*. При этом я указываю полные координаты окна *Form2* т.е. (0, 0, ширина второй формы, высота второй формы).

Во второй строке я заполняю структуру *DRect* с помощью всё той же функции *Rect*. В принципе, ей можно не пользоваться и заполнять поля как и для любой другой структуры:

```
DRect.Left:=10;
DRect.Top:=10;
DRect.Right:=110;
DRect.Bottom:=110;
```

В этом случае код займёт аж четыре строчки, поэтому я не люблю такие вещи. Лучше всё записать одной строкой.

И последнее - я копирую холст второй формы в первую. Сразу хочу ответить, что если размер области источника больше приёмника, то область будет растянута/сжата до размеров приёмника. Это значит, что если размеры второй формы больше чем 100x100 (именно в такой квадрат на форме 1 мы хотим скопировать вторую форму), то изображения второй формы будет сжато до размеров 100x100.

Draw

Этот метод тоже предназначен для копирования изображений, но другого формата. У него три параметра – X и Y координаты куда копировать и объект типа TGraphic который надо копировать. Этот объект мы ещё не рассматривали и узнаем о нём в следующей главе.

Ellipse

Этот метод предназначен для рисования эллипса (овала). Есть две реализации этого метода:

```
procedure Ellipse(X1, Y1, X2, Y2: Integer);  
procedure Ellipse(const Rect: TRect);
```

В первом случае нужно передать четыре координаты прямоугольника, в который будет вписан эллипс. Во втором случае достаточно одного параметр типа TRect (как ты уже знаешь, у этой структуры есть все необходимые четыре поля). Какой ты будешь использовать – это твоё дело.

FillRect

У этого метода только один параметр – TRect, указывающий область, которую необходимо залить цветом кисти. В принципе, это то же самое, что и нарисовать прямоугольник.

FloodFill

Заливка. У этого метода четыре параметра – X и Y координаты точки, с которой нужно начинать заливку. Третий параметр – цвет. Последний параметр – способ заливки. Возможны два способа:

fsSurface – залить всю область, где цвет равен цвету указанному в третьем параметре.

fsBorder - залить всю область, где цвет не равен цвету указанному в третьем параметре.

Этих методов пока достаточно. Я не собираюсь переписывать весь файл помощи в своей книге, потому что это бесполезно. Но я показал тебе основные методы, которые тебе могут пригодиться. С ними, и многими другими методами мы познакомимся на практике чуть позже. Возможно уже в следующей главе.

12.9 Компонент работы с графическими файлами (TImage)

Как я уже сказал – первое, с чем мы познакомимся в этой главе, будет компонент для работы с графическими файлами – **TImage**. Этот компонент ты можешь найти на закладке *Additional* палитры компонентов.

С этим компонентом мы уже немного познакомились в главе 11.5, но тогда мы рассматривали его как компонент украшение, который просто располагает на форме красивое изображение. Тогда я не хотел затрагивать другие возможности компонента, потому что мы ещё не были готовы познакомиться с графикой. Сейчас, когда я уже рассказал всё необходимое, пора разобрать этот компонент по свойствам и методам.



- TImage

Компонент **TImage** достаточно универсальный и может отображать картинки разного формата. Но в начальной установке он может загружать только BMP, JPG, JPEG или WMF файлы. Давай посмотрим, как это делается. Создай новое приложение и брось на форму одну кнопку и компонент **TImage** с закладки *Additional*.

Теперь брось на форму компонент *OpenPictureDialog* с закладки *Dialogs*. Этот компонент предназначен для отображения на экране стандартного окна открытия картинки. Нам так же понадобится кнопка, по нажатию которой мы будем отображать окно открытия картинки и потом загружать выбранную.

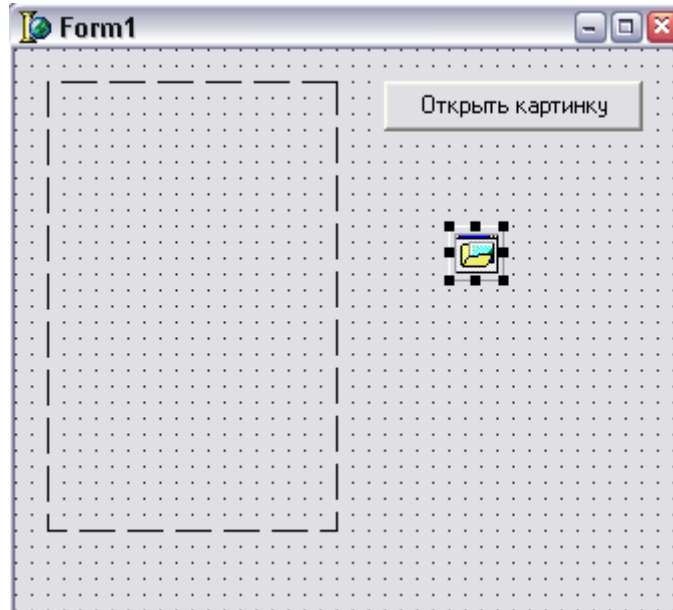


Рис. 12.9.1. Форма будущей программы

На рисунке 12.9.1 ты можешь увидеть форму будущей программы. Но пока готова только форма. Чтобы программа стала полноценной надо написать код загрузки картинки. По нажатию кнопки «Открыть картинку» пишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

В первой строке я отображаю стандартное окно открытия картинки. Для этого достаточно вызвать метод *Execute* компонента *OpenPictureDialog1*, т.е. написать *OpenPictureDialog1.Execute*. Этот метод возвращает логическое значение. Если оно равно *true*, то пользователь выбрал файл, иначе нажал отмену. Именно поэтому я проверяю результат вызова метода *Execute* с помощью *if OpenPictureDialog1.Execute then*.

Если файл выбран, то выполняется следующий код:

```
Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
```

Разберём эту конструкцию по частям. У компонента *Image1* есть свойство *Picture*. Это свойство имеет объектный тип (а значит и свои свойства и методы) *TPicture*. Этот объект предназначен для работы с изображениями практически любого типа. Он достаточно универсален, в чём мы убедимся достаточно скоро.

Для загрузки изображения я использую метод *LoadFromFile* (загрузить картинку из файла) объекта *Picture*. В качестве единственного параметра этого метода нужно указать имя открываемого файла или полный путь, если картинка находится не в той же директории, что и сама программа.

Мы выбираем имя файла с помощью стандартного окна и полный путь к файлу находится в свойстве *FileName* компонента *OpenPictureDialog1*.

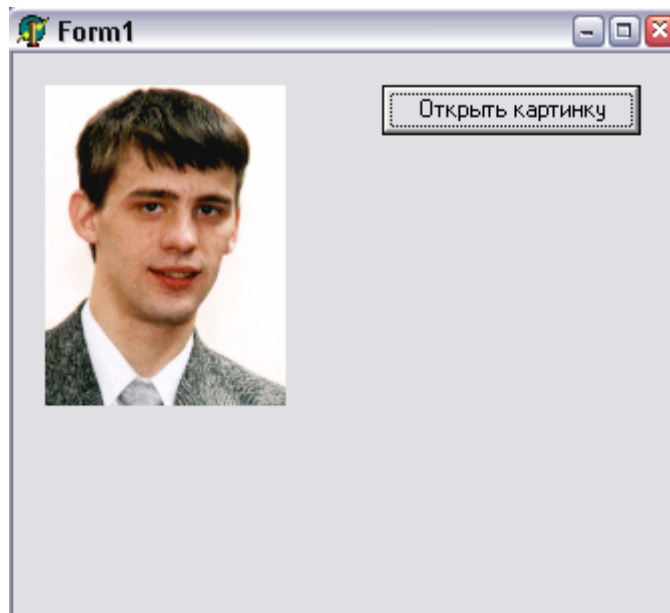


Рис. 12.9.2. Программа в действии

Всё достаточно просто. Попробуй теперь запустить программу и посмотреть на результат её работы. В окне открытия файла посмотри, какие типы файлов ты можешь открывать. В зависимости от версии и установленной комплектации количество типов может быть от 1 до 3 - *bmp*, *ico* и *wmf*.

Давай научим нашу программу работать с *jpeg* форматом файлов. Не волнуйся, это не сложно и нам не придётся писать сложный алгоритм распаковки изображения. В Delphi уже есть всё необходимое, надо только это необходимое подключить к проекту.

Для начала переместись в раздел **uses** проекта и подключи туда модуль *jpeg*. В этом модуле описано всё необходимое для работы с *jpeg* форматом изображений.

В принципе этого достаточно. Осталось только заставить окно открытия файлов показывать фильтр на данный тип файлов. Для этого выдели компонент *OpenPictureDialog1* и дважды щёлкни по свойству *Filter*.

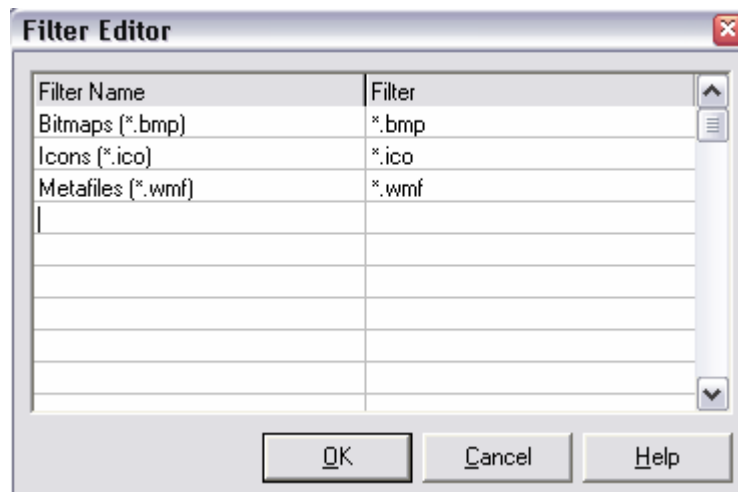



Рис. 12.9.3. Окно редактирования фильтра

В этом окне у тебя есть несколько заполненных строк. У меня (рисунок 12.9.3) только три. В четвёртой строке (первой пустой строке) напиши в первой колонке «*JPEG Files*», а во второй колонке напиши «**.jpg*». Можешь нажимать *OK* и запускать программу. Теперь в окне открытия графического файла можно выбрать тип *JPEG* и открыть нужный файл. Он так же будет загружен в компонент *Image1*, даже не смотря на свой сложный алгоритм сжатия и другой вид хранения данных.

 На компакт диске, в директории \Примеры\Глава 12>Loading Images ты можешь увидеть пример этой программы.

Теперь попробуем модернизировать пример и получить доступ к содержимому картинки. Для этого я буду копировать изображение используя прозрачность. Как? Сейчас узнаешь.

Создай для неё обработчик события *OnPaint* для главной формы в уже созданном нами примере. В созданной процедуре *FormPaint* напиши следующее:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.BrushCopy(Rect(200,16,200+Image1.Width,16+Image1.Height),
    Image1.Picture.Bitmap,
    Rect(0,0,Image1.Width,Image1.Height),
    Image1.Picture.Bitmap.Canvas.Pixels[1,1]);
end;
```

А в процедуре, где мы загружаем изображение нужно в конце добавить вызов метода *Repaint*, чтобы после открытия графического файла форма прорисовалась заново и вызвался обработчик *OnPaint*.

Теперь попробуй запустить программу и загрузить в неё bmp файл. Ты должен увидеть результат подобный рисунку 12.9.4. Слева у нас находится изображение картинки *Image1*, а справа мы делаем копию изображения на форму.

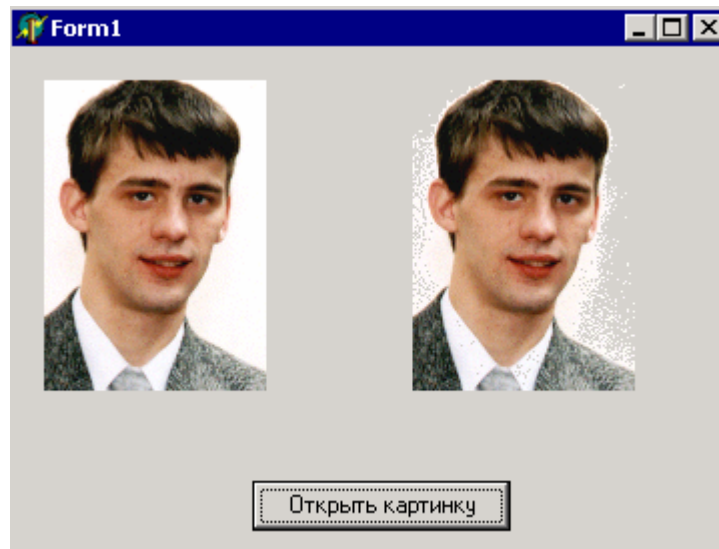


Рис 12.9.4. Результат работы программы


А вот теперь давай посмотрим, что тут происходит. Код кажется немного сложным, но это только на первый взгляд. Давай рассмотрим всё по частям. Мы используем процедуру `BrushCopy` у уже знакомого `Canvas`. Эта процедура копирует на `Canvas` картинку.

```
procedure BrushCopy(  
  const Dest: TRect; // Область приёмника  
  Bitmap: TBitmap; // Картинка которая будет копироваться  
  const Source: TRect; // Область источника  
  Color: TColor); // Прозрачный цвет
```

Область приёмника объявлена как `TRect`, который имеет вид `TRect = (Left, Top, Right, Bottom: Integer)`. Что находится в скобках, я думаю пояснять не надо. То же самое и с областью источника. В качестве картинки мы передаём `Bitmap` из `TImage`.

В качестве прозрачного цвета я использовал цвет пикселя в позиции `[1,1]` из картинки `TImage`. На это указывает запись `Image1.Picture.Bitmap.Canvas.Pixels[1,1]`. Попробую записать её немного по другому:

`TImage1.Его_картинка.Bitmap.Холст.Пиксел[1_по_оси_X, 1_по_оси_Y]`

 На компакт диске, в директории `\Примеры\Глава 12\Image1` ты можешь увидеть пример этой программы.

Обрати внимание, что если ты сейчас попытаешься открыть какой-нибудь не `bmp` файл, то программа ничего не отобразит. Это связано с тем, что только `BMP` файлы хранят свои изображения в свойстве `Bitmap`, все остальные хранят в свойстве `Graphic`. Так что получить доступ к `JPEG` изображению таким образом нельзя. Зато можно с помощью метода `Draw`. Для этого нужно подкорректировать обработчик события `OnPaint`:

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
  Canvas.Draw(200, 16, Image1.Picture.Graphic);
```

end;

Здесь я уже рисую не *Bitmap*, а *Graphic*, поэтому программа будет работать корректно.

Векторные файлы, такие как *wmf* хранят свои данные в свойстве *Metafile*. Для их отображения обработчик события *OnPaint* должен быть таким:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Draw(200, 16, Image1.Picture.Metafile);
end;
```

12.10 Рисование на стандартных компонентах

Очень часто, для лучшего представления данных, тебе будет нужно рисовать внутри компонента *TListBox*. Что я имею ввиду? Посмотри на рисунок 12.10.1, и ты всё поймёшь.

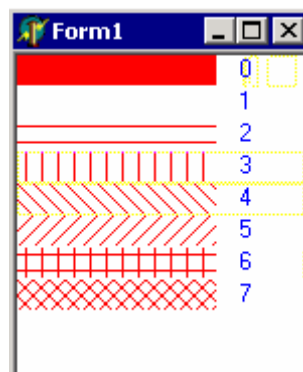


Рис 12.10.1. Пример

Может это покажется странным, но всё это делается за семь строчек кода. Конечно же, в одну строку можно записать и двадцать операций, но я этот случай не учитываю.

Секрет рисования заключается в том, что у компонента *TListBox1* свойство *Style* должен быть *lbOwnerDrawFixed* или *lbOwnerDrawVariable*.

После этого создаёшь обработчик события *OnDrawItem* для этого компонента и в нём пишешь:

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
  with ListBox1.Canvas do
  begin
    Brush.Color:=clRed; // Задаём красный цвет кисти.
    Brush.Style:=TBrushStyle(Index); // Выбираем стиль кисти
    Pen.Style:=psClear;
    Rectangle(Rect.Left,Rect.Top,Rect.Left+100,Rect.Bottom);
```

```
Brush.Style:=bsClear;  
Font.Color:=clBlue;  
TextOut(Rect.Left+110,Rect.Top,IntToStr(index));  
end;  
end;
```

Вот и всё. Твоя прога готова, жми на запуск и наслаждайся. Только давай посмотрим, что тут написано.

Первая строка:

with ListBox1.Canvas do

Оператор "With" говорит, что все последующие операции будут производиться с компонентом (объектом) *ListBox1.Canvas*. Для того, чтобы ты лучше понял я приведу код без этой строки:

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;  
  Rect: TRect; State: TOwnerDrawState);  
begin  
  ListBox1.Canvas.Brush.Color:=clRed;  
  ListBox1.Canvas.Brush.Style:=TBrushStyle(Index);  
  ListBox1.Canvas.Pen.Style:=psClear;  
  ListBox1.Canvas.Rectangle(Rect.Left,Rect.Top,Rect.Left+100,Rect.Bottom);  
  
  ListBox1.Canvas.Brush.Style:=bsClear;  
  ListBox1.Canvas.Font.Color:=clBlue;  
  ListBox1.Canvas.TextOut(Rect.Left+110,Rect.Top,IntToStr(index));  
end;
```

Как видишь, в каждой строке появились подписи *ListBox1.Canvas*. Код стал очень не красивым. Постоянно нужно говорить, что Brush или ещё что-нибудь нужно взять у *ListBox1.Canvas*.

Поэтому я и использовал оператор *With*


```
With "Объект" do  
Begin  
  // Всё, что находится здесь, будет относиться к объекту "Объект".  
  // Поэтому не надо писать имя объекта перед каждым используемым  
  // Свойством или методом.  
End;
```

Теперь ещё несколько подводных камней нашей проги. Конструкция *Brush.Style:=TBrushStyle(Index)* выбирает кисть в зависимости от рисуемого в данный момент элемента. Всего существует восемь стилей кисти. Когда вываливается сообщение *OnDrawItem* для первого элемента (об этом говорит параметр *index* передаваемый в процедуру *ListBox1DrawItem*), мы рисуем элемент с кистью первого стиля. Для второго элемента будет использоваться второй стиль кисти и т.д.

Карандаш я выбрал прозрачным *Pen.Style:=psClear*, это для того, чтобы не было никаких оборок. Попробуй убрать эту строку и посмотреть на результат.

Функция *Rectangle(x1,y2,x2,y2)* рисует прямоугольник с соответствующими координатами. Дальше я делал прозрачной кисть и задавал цвет фона. После этого я просто выводил текст строки с помощью функции *TextOut(x, y, текст)*.

Попробуй сделать тоже самое с компонентом *TComboBox*. Не забудь про свойство *Style* у этого компонента. А в остальном, весь код будет таким же.

 На компакт диске, в директории \Примеры\Глава 12\ListBox ты можешь увидеть пример этой программы.

Ещё один пример рисования в стандартных компонентах, который я использую с целью обучения работы с графикой – рисование графических подсказок в строке состояния. Что я имел ввиду под выражением "графические подсказки"? Всё очень просто. Ты каждый день встречаешь в программах строку состояния внизу экрана, в которой выскакивают подсказки. Сегодня я покажу тебе, как сделать текст в компоненте *StatusBar* трёхмерным.

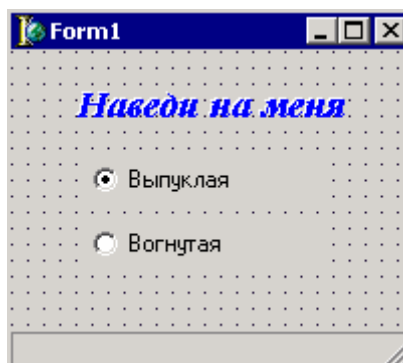


Рис 12.10.2. Форма будущей программы

На рисунке 12.10.2 показана форма, которая будет использоваться нами для вывода графической подсказки. Прежде чем мы приступим, я хочу напомнить, как вообще выводятся подсказки. Вот пример программы (точнее огрызок от программы), которая выводит подсказки:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := ShowHint;
end;


procedure TForm1.ShowHint(Sender: TObject);
begin
  StatusBar1.SimpleText:=Application.Hint;
end;
```

Вспомним, что здесь происходит. В процедуре *FormCreate* (обработчик события *OnCreate* для главной формы), мы устанавливаем в качестве обработчика события *Application.OnHint* свою процедуру *ShowHint*. Теперь, когда будет происходить событие *OnHint* (т.е. когда нужно вывести подсказку), будет вызываться процедура *ShowHint*. В этой процедуре я просто вывожу подсказку в *StatusBar1*.

Теперь можно переходить к графической подсказке. Вот полный исходник нового вида функции *ShowHint*:


```
procedure TForm1.ShowHint(Sender: TObject);
var
  l,t:Integer;
begin
  StatusBar1.Repaint;
  with StatusBar1.Canvas do
  begin
    Brush.Style:=bsClear;
    Font.Color:=clWhite;
    l:=10;
    t:=1;
    TextOut(l,t,Application.Hint);
    if RadioButton1.Checked then
    begin
      inc(l);
      inc(t);
    end
    else
    begin
      dec(l);
      dec(t);
    end;
    Font.Color:=clBlue;
    TextOut(l,t,Application.Hint);
  end;
end;
```

Здесь ничего сложного нет. Я просто вывожу два раза текст подсказки с разным цветом и небольшим смещением. Это происходит точно так же, как и расположение двух компонентов *TLabel* на форме с небольшим смещением и разным цветом текста.

 На компакт диске, в директории \Примеры\Глава 12\StatusBar ты можешь увидеть пример этой программы.

12.11 Работа с экраном.

Не знаю зачем, но очень часто меня спрашивают о том, как получить доступ к экрану. Такие люди хотят скопировать содержимое экрана в виде картинку и потом использовать это по своему усмотрению. Так как такой вопрос бывает не редко, то я решил добавить описание этого примера в книгу. В любом случае пример интересен и полезен в познавательных целях.

Для этого примера нам понадобится форма с двумя кнопками и одной картинкой. Создай новый проект и поставь на него две пимпы *TButton* и один штука *TImage*. Для первой кнопки напишем в событии *OnClick*:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ScreenDC:HDC;
begin
  ScreenDC := GetDC(0);
  Rectangle(ScreenDC, 10, 10, 200, 200);
  ReleaseDC(0,ScreenDC);
end;
```

С помощью этой процедуры я рисую прямо на экране вне области окна своей программы. Во время рисования, я не обращаю внимания на запущенные приложения. Если они попадают под руку, то рисование происходит поверх них.

Теперь о содержимом. Я объявляю переменную *ScreenDC* типа *HDC*. *HDC* - это тип контекста рисования в *windows*, и работает почти так же, как и *TCanvas* (чуть позже ты увидишь связь). С помощью функции *GetDC(0)* я возвращаю контекст окна указанного в скобках. Но в этих скобках стоит 0 (ноль), а не указатель на реальное устройство или окно. Это значит, что мне нужен глобальный контекст, то есть самого экрана.

Далее, я вызываю функцию *Rectangle*, она похожа на ту, что мы использовали раньше *TCanvas.Rectangle*. Есть только одно отличие - первый параметр теперь, это контекст устройства, а затем идут координаты прямоугольника. Это связано с тем, что раньше мы рисовали через объект *TCanvas*, а сейчас будем рисовать средствами GDI Windows. Если честно, то процедура *TCanvas.Rectangle* всего лишь вызывает *Rectangle* из Windows API и подставляет нужный контекст устройства и размеры, поэтому в ней на один параметр меньше. Сейчас мы сами сделаем это, без помощи *TCanvas*.

После рисования, я освобождаю больше не нужный мне контекст через функцию *ReleaseDC*. Такие вещи обязательно надо освобождать, чтобы не засорять память.

Если ты захочешь рисовать не на экране, а внутри определённого окна, то в этой процедуре нужно поправить только первую строчку. А именно, в качестве параметра *GetDC* передавать указатель на окно. Указатель на наше окно находится в свойстве *Handle* объекта *TForm*.

Сейчас можно запустить программу и посмотреть на результат, а я пока перейду ко второй кнопке. Для неё мы напишем следующий текст по событию *OnClick*:


```
procedure TForm1.Button2Click(Sender: TObject);
var
  Canvas:TCanvas;
  ScreenDC:HDC;
begin
  ScreenDC := GetDC(0);
  Canvas:=TCanvas.Create();
  Canvas.Handle:=ScreenDC;
  Image1.Canvas.Copyrect(Rect(0,0,Image1.Width,Image1.Height),
    Canvas, Rect(0,0,Screen.Width,Screen.Height));
  ReleaseDC(0,ScreenDC);
  Canvas.Free;
end;
```

Сразу скажу, что здесь я получаю копию экрана и сохраняю её в компоненте *Image1*.

Первая строка такая же, как и в предыдущей процедуре. Я точно также получаю контекст рисования экрана. Потом я создаю новую переменную *Canvas* типа *TCanvas* (знакомый нам контекст рисования). Потом я связываю их между собой с помощью простого присваивания в *Canvas.Handle:=ScreenDC*. Теперь мой *TCanvas* указывает на экран, и я могу рисовать на нём, привычными нам методами. Теперь видишь связь между холстом *Canvas* и контекстом рисования *HDC*. Объект холста всегда содержит указатель на контекст рисования *HDC* в свойстве *handle* и использует этот контекст при вызове всех своих методов (таких как *Rectangle*). Для компонентов Delphi это свойство заполняется автоматически и нам не надо о нём заботиться.

Далее, я получаю копию экрана и записываю её в картинку *TImage* с помощью функции *CopyRect* у контекста рисования картинки (*Image1.Canvas.CopyRect*).

После копирования я освобождаю контекст рисования *ScreenDC* и созданный холст *Canvas* чтобы освободить выделенную память.

 На компакт диске, в директории \Примеры\Глава 12\Screen ты можешь увидеть пример этой программы.

12.12 Режимы рисования.

У карандаша (свойство *Pen* холста) есть очень много режимов рисования. Благодаря им можно добиться очень интересных эффектов. Режимы рисования устанавливаются в свойстве *Mode* карандаша. Это свойство имеет тип *TPenMode* и может принимать следующие значения: *pmBlack*, *pmWhite*, *pmNop*, *pmNot*, *pmCopy* и так далее. Здесь я тебе покажу, как можно использовать это свойство в своих целях.

Я пишу не справочник, а книгу, по которой ты должен научиться программировать, познакомиться с некоторыми алгоритмами и научиться правильно мыслить. Именно поэтому я не буду расписывать все возможные варианты режимов, а опишу только один – *pmNotXor*, и пример его использования. Почему этот режим? Да потому что ему легко найти применения, а остальные мне ещё ни разу не понадобились за всю мою программистскую карьеру.

Итак, представим, что нам надо нарисовать пример, в котором пользователь должен иметь возможность рисования на форме прямоугольника. Кода в нашем примере будет немного и логика до предела проста:

1. По нажатию кнопки мыши мы запоминаем текущие координаты точки, где был произведён щелчок.
2. При движении мышки мы должны проверять: если кнопка мыши нажата, то пользователь растягивает прямоугольник и мы должны его нарисовать начиная от стартовой позиции до текущей.

Теперь реализуем это в программе. Создай новый проект и перейди в раздел **private** описания объекта. Там нужно добавить следующие переменные:

```
private
{ Private declarations }
StartX, StartY:Integer;
dragging:Boolean;
```

Переменные *StartX*, *StartY* будут использоваться для запоминания координат начала прямоугольника. Переменная *dragging* будет использоваться для признака растягивания. Если эта переменная равна *true*, то пользователь нажал кнопку мыши и перемещает кнопку мыши нажата и мы должны растягивать прямоугольник.

По событию *OnCreate* для формы мы должны задать переменной *dragging* значение по умолчанию – *false*, ведь после старта приложения мышь ничего не тянет и кнопки не нажата. Вот чтобы избежать случайного попадания в переменную значения *true* мы должны по этому событию написать следующий код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  dragging:=false;
end;
```

По нажатию кнопки мыши *OnMouseDown* пишем следующий код:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  
  Shift: TShiftState; X, Y: Integer);  
begin  
  StartX:=X;  
  StartY:=Y;  
  
  dragging:=true;  
end;
```

Сначала рассмотрим параметры, которые мы получили вместе с обработчиком. У нас тут пять параметров:

Sender – как всегда, здесь храниться объект, который сгенерировал событие.

Button – здесь храниться признак клавиши, которая была нажата. Этот параметр может быть равен: *mbLeft* (нажата левая кнопка), *mbRight* (нажата правая кнопка) или *mbMiddle* (нажата средняя кнопка).

Shift – состояние дополнительных клавиш клавиатуры. Это набор параметров типа *TShiftState*. Этот параметр может хранить любые из следующих значений *ssShift* - нажата клавиша *Shift*, *ssAlt* - нажата клавиша *Alt*, *ssCtrl* - нажата клавиша *Ctrl*, *ssLeft* - левая кнопка мыши нажата, *ssRight* - правая кнопка мыши нажата, *ssMiddle* – средняя кнопка нажата, *ssDouble* – был двойной щелчок мышкой.

Чтобы проверить, была ли нажата кнопка *Shift* во время нажатия мышкой можно написать следующий код:

```
if ssShift in Shift then  
  ....
```

Почему этот параметр – это набор? Да потому что одновременно на клавиатуре может быть нажато две клавиши *Ctrl* и *Shift* и даже три клавиши.

X, Y – последние два параметра – это координаты, в которых была нажата кнопка мыши.

Внутри самой процедуры я сохраняю координаты в переменных *StartX* и *StartY*, и изменяю переменную *dragging* на *true*.

Теперь напишем обработчик события *OnMouseMove*, который генерируется при каждом перемещении мышки:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,  
  Y: Integer);  
begin  
  if dragging=false then exit;  
  
  Canvas.Rectangle(StartX, StartY, X, Y);  
end;
```

Параметры этого обработчика похожи на обработчик события *OnMouseDown*.

В первой строчке происходит проверка, если переменная *dragging* равна *false*, то сейчас нет никакого перемещения и нужно выйти из процедуры. Иначе нужно нарисовать прямоугольник с координатами первой точки *StartX*, *StartY* и второй точки *X*, *Y*.

По событию *OnMouseUp* нужно присвоить переменной *dragging* значение *false*. Я думаю, что нет смысла показывать этот код, напиши его сам.

Теперь запусти программу. Попробуй щёлкнуть мышкой и потянуть её. Будет создаваться эффект, как будто ты растягиваешь прямоугольник. А теперь не отпуская мышку попробуй начать уменьшать прямоугольник. Вот тут начинается полный ужас рисунок 12.12.1. Пока мы растягивали прямоугольник, он спокойно накладывался сверху старого. Но как только мы начали уменьшать, новые прямоугольники становятся меньше старого и старый остаётся на экране, а новый оказывается как бы внутри.

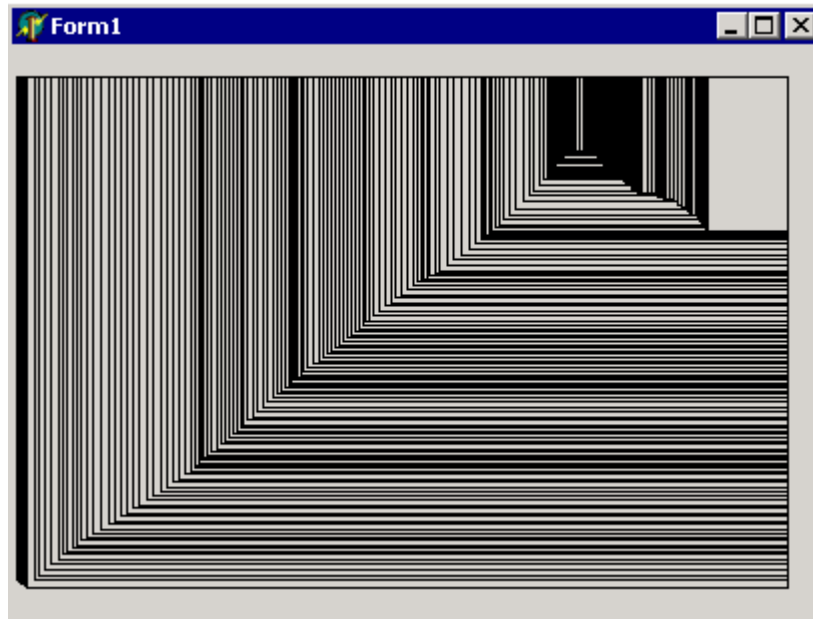



Рис 12.12.1. Пример работы программы

 На компакт диске, в директории \Примеры\Глава 12\CopyMode1 ты можешь увидеть пример этой программы.

Чтобы избавиться от этого эффекта есть два способа:

1. Перед каждым рисованием запоминать содержимое экрана и потом восстанавливать его. Такой способ связан с большими нагрузками на процессор и лишним расходом памяти.
2. Затирать только старые прямоугольники и восстанавливать то, что было под линиями.

Все почему-то боятся использовать второй способ, думая, что он сложен. Сейчас я покажу тебе обратное. Мало того, что этот способ абсолютно прост в программировании, но и очень быстр.

Для начала добавим в раздел **private** ещё несколько переменных:

```
private
{ Private declarations }
OldPenMode:TPenMode;
StartX, StartY, OldX, OldY:Integer;
dragging:Boolean;
```

Я добавил переменную *OldPenMode*, в которой будет сохраняться текущее значение режима рисования. Переменные *OldX*, *OldY* нужны для хранения конечных координат старого прямоугольника (начальные координаты *StartX* и *StartY*).

Теперь подправим обработчик события *OnMouseDown*:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.Brush.Color:=clWhite;
  OldPenMode:=Canvas.Pen.Mode;
  Canvas.Pen.Mode:=pmNotXor;

  StartX:=X;
  StartY:=Y;
  OldX:=X;
  OldY:=Y;

  dragging:=true;
end;
```

В самом начале я добавил изменение свойства кисти холста. Я сделал кисть белой, чтобы прямоугольники имели белый фон и ты лучше увидел эффект закраски.

Во второй строке я сохраняю текущий режим рисования в переменной *OldPenMode*. В следующей строке я меняю режим на *pmNotXor*. В таком режиме, когда мы рисуем первый раз какую-то фигуру, то она выводится в нормальном виде. Если нарисовать второй раз прямоугольник, то он просто стирается и старое изображение восстанавливается на экране.

После этого я просто заполняю текущие координаты *StartX*, *StartY*, *OldX* и *OldY*.

Теперь посмотрим обработчик события *OnMouseMove*:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if dragging=false then exit;

  Canvas.Rectangle(StartX, StartY, OldX, OldY);
  Canvas.Rectangle(StartX, StartY, X, Y);
  OldX:=X;
  OldY:=Y;
end;
```

Начало обработчика такое же. Потом я рисую прямоугольник в старой позиции, где он был нарисован на прошлом шаге. Так как используется режим *pmNotXor*, то повторное рисования прямоугольника в старой позиции просто восстанавливает старое значение. После этого я рисую фигуру в новой позиции и сохраняю текущую позицию X и Y в переменных *OldX* и *OldY*.

И наконец обработчик события *OnMouseUp*:


```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
dragging:=false;
Canvas.Pen.Mode:=OldPenMode;
Canvas.Rectangle(StartX, StartY, X, Y);
end;
```

В первой строке я присваиваю переменной *dragging* значение *false*. Во второй восстанавливаю старое значение режима рисования. И в самой последней строке я рисую уже окончательный вариант прямоугольника.

Запусти программу и попробуй нарисовать прямоугольник. Теперь, когда ты его рисуешь никаких накладок не происходит. Обрати внимание, что когда ты растягиваешь прямоугольник, то его фон прозрачный и только когда ты отпускаешь мышку (режим рисования восстанавливается) рисуется прямоугольник с фоном белого цвета.

Попробуй нарисовать сразу несколько прямоугольников на форме и убедиться, что всё работает нормально.

 На компакт диске, в директории \Примеры\Глава 12\CopyMode2 ты можешь увидеть пример этой программы.