

Федеральное государственное образовательное учреждение  
высшего профессионального образования  
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

---

**М. Э. Абрамян, С. С. Михалкович**

---

**РАБОТА С ГРАФИКОЙ  
В СИСТЕМЕ  
PASCALABC.NET**

---

Ростов-на-Дону 2009

## Оглавление

1. Модуль GraphABC.....	3
2. Цвета и пикселы .....	3
Задания .....	5
3. Графические примитивы .....	6
Задания .....	9
4. Перо и кисть.....	10
Задания .....	13
5. Процедура FloodFill.....	14
Задания .....	14
6. Вывод текста .....	15
Задания .....	16
7. Действия с графическим окном .....	17
Задания .....	19
8. Система координат .....	19
Задания .....	21
9. Анимация .....	22
Задания .....	23
10. Рисунки.....	24
Задания .....	28

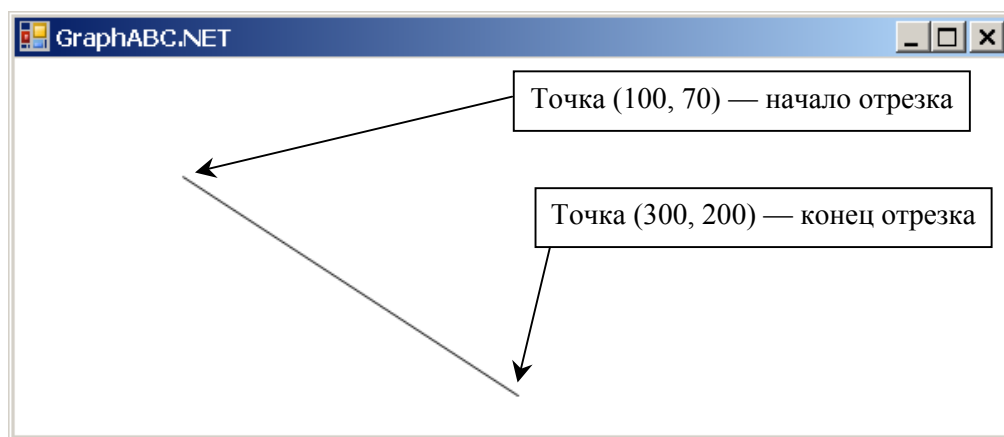
## 1. Модуль GraphABC

В системе PascalABC.NET имеется стандартный модуль GraphABC, предназначенный для работы с графикой. Он включает более 100 процедур, функций, переменных и констант и позволяет рисовать в специальном *графическом окне*.

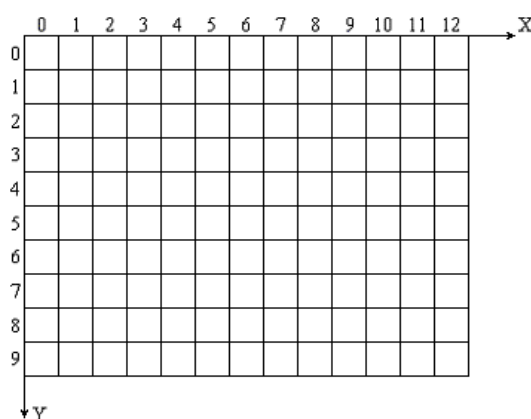
Рассмотрим простейшую графическую программу.

```
П uses GraphABC;  
begin  
    Line(100, 70, 300, 200);  
end.
```

После ее запуска мы увидим на экране графическое окно, в котором будет нарисован отрезок прямой с началом в точке (100, 70) и концом в точке (300, 200).



Графическое окно состоит из *пикселей* — точек, которые можно окрашивать в различные цвета, формируя тем самым требуемое изображение. Пиксели нумеруются слева направо и сверху вниз, начиная с нулевого пиксела. Изобразим рабочую область графического окна в увеличенном виде с нанесенной системой координат (каждый квадратик на рисунке соответствует отдельному пикселу).



Как видно из рисунка, ось ординат  $OY$  направлена вниз, и левый верхний пиксел имеет координаты (0, 0).

## 2. Цвета и пиксели

*Цвета* в библиотеке GraphABC имеют специальный тип Color. Любой непрозрачный цвет можно получить с помощью функции RGB( $r, g, b$ ), задав в каче-

стве параметров красную (*red*), зеленую (*green*) и синюю (*blue*) составляющие цвета с интенсивностями *r*, *g* и *b* соответственно (*r*, *g* и *b* — целые в диапазоне от 0 до 255, причем значение 0 соответствует минимальной интенсивности, а 255 — максимальной). Например, RGB (255, 0, 0) дает красный цвет, RGB (0, 0, 255) — синий, RGB (255, 255, 0) — желтый. Таким образом, в нашем распоряжении имеется  $256^3 = 16777216$  различных непрозрачных цветов.

Имеется также функция ARGB (*a*, *r*, *g*, *b*), позволяющая задать дополнительную составляющую цвета *a*, определяющую уровень его *прозрачности* (эта составляющая называется *альфа-компонентой*). Альфа-компонента, как и прочие цветовые составляющие, принимает значения в диапазоне 0..255, причем 255 означает непрозрачный цвет, а 0 — полностью прозрачный. Смысл остальных параметров функции ARGB — тот же, что и для функции RGB. Таким образом, цвет, возвращаемый функцией RGB(*r*, *g*, *b*), совпадает с цветом, возвращаемым функцией ARGB(255, *r*, *g*, *b*).

Из цвета *col* можно выделить красную, зеленую и синюю составляющие, используя функции GetRed(*col*), GetGreen(*col*), GetBlue(*col*). Например, если цвет *col* создан с помощью оператора *col* := RGB(50, 100, 150), то функция GetRed(*col*) возвратит число 50, GetGreen(*col*) возвратит 100 и GetBlue(*col*) возвратит 150. Для получения альфа-компоненты предназначена функция GetAlpha(*col*).

В модуле GraphABC имеется около 140 именованных констант для обозначения стандартных цветов (константы имеют префикс *cl*). Все стандартные цвета, за исключением полностью прозрачного цвета *clTransparent* с нулевой альфа-компонентой, являются непрозрачными. В приводимом ниже списке основных цветовых констант после имени константы указывается ее описание и (в скобках) соответствующие ей интенсивности красной, зеленой и синей составляющих.

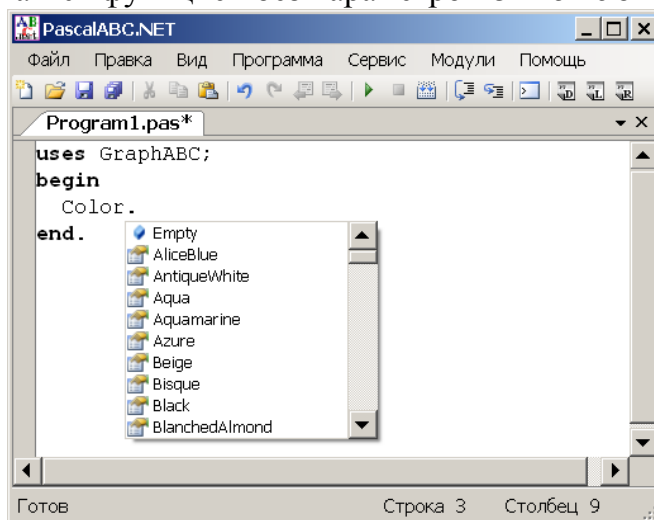
<i>clAqua</i>	– бирюзовый (0,255,255)	<i>clMaroon</i>	– темно-красный (128,0,0)
<i>clBlack</i>	– черный (0,0,0)	<i>clNavy</i>	– темно-синий (0,0,128)
<i>clBlue</i>	– синий (0,0,255)	<i>clOlive</i>	– оливковый (128,128,0)
<i>clBrown</i>	– коричневый (165,42,42)	<i>clPurple</i>	– фиолетовый (128,0,128)
<i>clDarkGray</i>	– темно-серый (169,169,169)	<i>clRed</i>	– красный (255,0,0)
<i>clFuchsia</i>	– сиреневый (255,0,255)	<i>clSilver</i>	– серебряный (192,192,192)
<i>clGray</i>	– серый (128,128,128)	<i>clSkyBlue</i>	– голубой (135,206,235)
<i>clGreen</i>	– зеленый (0,128,0)	<i>clTeal</i>	– сине-зеленый (0,128,128)
<i>clLightGray</i>	– светло-серый (211,211,211)	<i>clWhite</i>	– белый (255,255,255)
<i>clLime</i>	– ярко-зеленый (0,255,0)	<i>clYellow</i>	– желтый (255,255,0)

Кроме этого, для задания стандартного цвета можно воспользоваться тем, что тип *Color* является так называемым *перечислимый типом*, то есть его значения определяются как перечисление именованных констант. Чтобы обратиться к этим константам, надо использовать *точечную нотацию*: *Color.Blue*, *Color.Green* и т. п. Для всех констант с префиксом *cl* имеются их одноименные аналоги в типе *Color*, например, *clGreen* = *Color.Green*. Точечная нотация удобна тем, что редактор PascalABC.NET после набора слова *Color* и следующей за ним точки выводит подсказку по точке: список всех значений типа *Color* (см. рисунок).

Используя точечную нотацию, полностью прозрачный цвет можно задать двумя способами: *Color.Transparent* (прозрачный белый цвет) и

`Color.Empty` (прозрачный черный цвет). На экране эти цвета ничем не отличаются, поскольку они «одинаково не видны».

Если в программе требуется получить случайный непрозрачный цвет, то проще всего воспользоваться функцией без параметров `clRandom`.



Для закрашивания пиксела с координатами  $(x, y)$  цветом `col` предназначена процедура `SetPixel(x, y, col)`. Параметры  $x$  и  $y$  имеют целый тип, параметр `col` — тип `Color`. Функция `GetPixel(x, y)` возвращает текущее значение цвета (типа `Color`) для пиксела с координатами  $(x, y)$ .

**Пример 1.** Рассмотрим следующую программу:

```
П uses GraphABC;
begin
    for var x:=0 to 255 do
        for var y:=0 to 255 do
            SetPixel(x, y, RGB(x, y, 0));
        end.
end.
```

Она выполняет градиентное закрашивание квадрата размера 256 на 256 (*градиентным* называется закрашивание, при котором цвета плавно переходят один в другой). Левый верхний угол квадрата ( $x = 0, y = 0$ ) имеет черный цвет, левый нижний ( $x = 0, y = 255$ ) — зеленый, правый верхний ( $x = 255, y = 0$ ) — красный, а в правом нижнем углу ( $x = 255, y = 255$ ) смешиваются зеленый и красный цвета, в результате чего получается желтый цвет.

**Пример 2.** Рассмотрим следующую программу:

```
П uses GraphABC;
begin
    for var i:=1 to 100000 do
        SetPixel(Random(256), Random(256), clRandom);
    end.
end.
```

Она заполняет случайными цветами пикселы квадрата размера 256 на 256, причем выбор пикселей также производится случайным образом.

## Задания

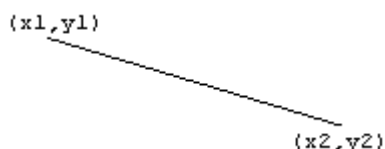
1. Изменить программу из примера 1 так, чтобы она:
  - а) иллюстрировала смешение красного и синего цветов;
  - б) иллюстрировала смешение синего и зеленого цветов.

2. Нарисовать квадрат с горизонтальным градиентным переходом от черного к красному.
3. Нарисовать квадрат с вертикальным градиентным переходом от белого к синему.
4. Нарисовать пикселями вертикальную линию.
5. Нарисовать пикселями периметр квадрата.

### 3. Графические примитивы

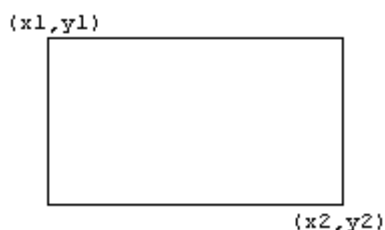
Простейшие геометрические фигуры, которые можно изобразить с помощью одной графической команды, называются *графическими примитивами*. В библиотеке GraphABC графическими примитивами являются отрезок, прямоугольник, эллипс, круг, дуга и сектор. Опишем связанные с ними процедуры рисования:

Line(x1, y1, x2, y2)



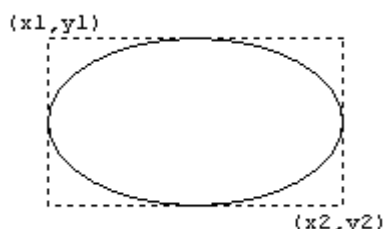
– рисование отрезка с началом в точке (x1, y1) и концом в точке (x2, y2);

Rectangle(x1, y1, x2, y2)



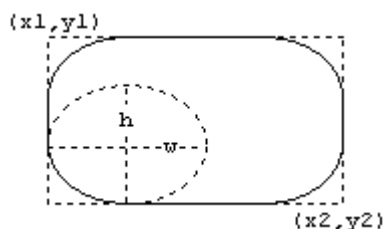
– рисование прямоугольника, заданного координатами противоположных вершин (x1, y1) и (x2, y2);

Ellipse(x1, y1, x2, y2)



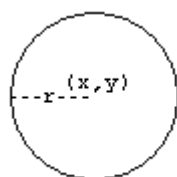
– рисование эллипса, вписанного в прямоугольник с координатами противоположных вершин (x1, y1) и (x2, y2);

RoundRect(x1, y1, x2, y2, w, h)



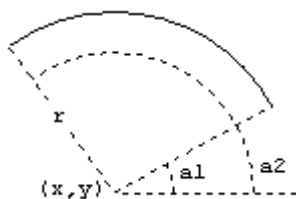
– рисование прямоугольника со скругленными краями; координаты (x1, y1) и (x2, y2) задают пару противоположных вершин прямоугольника, а числа w и h — ширину и высоту эллипса, используемого для скругления краев;

Circle(x, y, r)



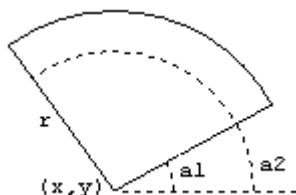
– рисование круга с центром в точке (x, y) и радиусом r; результат выполнения этой процедуры аналогичен результату выполнения процедуры Ellipse(x-r, y-r, x+r, y+r);

Arc(x, y, r, a1, a2)



– рисование дуги окружности с центром в точке  $(x, y)$  и радиусом  $r$ , заключенной между двумя лучами, образующими углы  $a1$  и  $a2$  с осью  $OX$  ( $a1$  и  $a2$  — вещественные, задаются в *градусах* и отсчитываются против часовой стрелки);

Pie(x, y, r, a1, a2)



– рисование сектора круга, ограниченного дугой; смысл параметров — тот же, что и для процедуры Arc.

### Пример 1. Рисование круга.

Введем координаты центра и радиус круга, а затем нарисуем этот круг.

```

П uses GraphABC;
begin
    var x, y, r: integer;
    write('Введите координаты центра: ');
    readln(x, y);
    write('Введите радиус круга: ');
    readln(r);
    Circle(x, y, r);
end.
```

Ввод чисел выполняется, как обычно, в поле ввода, появляющееся в нижней части окна PascalABC.NET. Если поле ввода заслонено графическим окном, то надо переместить это окно, зацепив мышью его заголовок. Перед набором чисел с клавиатуры необходимо активизировать поле ввода, щелкнув на нем мышью.

### Пример 2. Ряд из квадратов.

Начертим ряд из 8 квадратов со стороной 50 и расстоянием между ними 20. Очевидно, что левые края квадрата отстоят друг от друга на  $20+50=70$  пикселей. Поэтому на каждой итерации цикла будем увеличивать абсциссу  $x$  левой стороны квадрата на 70.

```

П uses GraphABC;
begin
    var x := 30;
    for var i:=1 to 8 do
        begin
            Rectangle(x, 30, x+50, 80);
            x += 70;
        end;
end.
```

### Пример 3. Круговая диаграмма.

Нарисуем круговую диаграмму, состоящую из четырех секторов с угловой величиной 30, 170, 90 и 70 градусов соответственно (в сумме 360 градусов). Центром круга будем считать точку (320, 240), а радиусом — 150 пикселей.

```

П uses GraphABC;
const
```

```

x = 320;
y = 240;
r = 150;
begin
  var a := 0;
  Pie(x,y,r,a,a+30);
  a += 30;
  Pie(x,y,r,a,a+170);
  a += 170;
  Pie(x,y,r,a,a+90);
  a += 90;
  Pie(x,y,r,a,a+70);
end.

```

#### Пример 4. Клеточное поле.

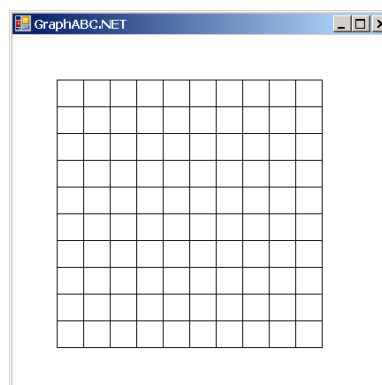
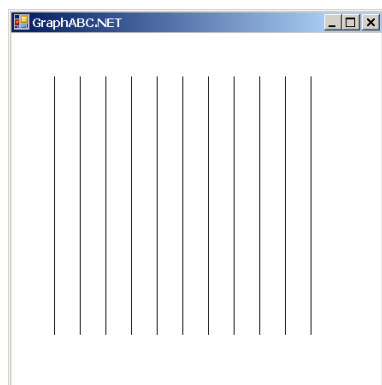
Нарисуем клеточное поле размера 10 на 10 клеток с размером каждой клетки 30 пикселей. Для этого нарисуем вначале 11 вертикальных отрезков длины  $30 \cdot 10 = 300$  на расстоянии 30 пикселей друг от друга:

```

П uses GraphABC;
begin
  var x := 50;
  for var i:=0 to 10 do
  begin
    Line(x,50,x,350);
    x += 30;
  end;
end.

```

Полученный результат изображен на левом рисунке.



Добавим в конец программы код, рисующий 11 горизонтальных отрезков:

```

var y := 50;
for var i:=0 to 10 do
begin
  Line(50,y,350,y);
  y += 30;
end;

```

Результат изображен на рисунке справа.

#### Пример 5. Правильный многоугольник.

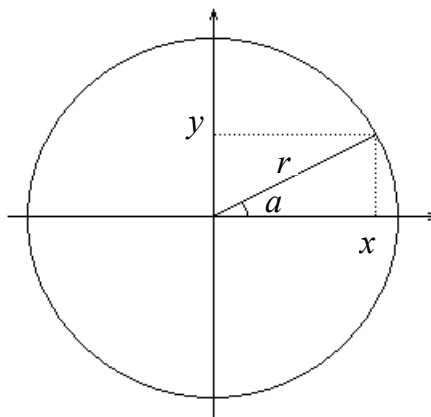
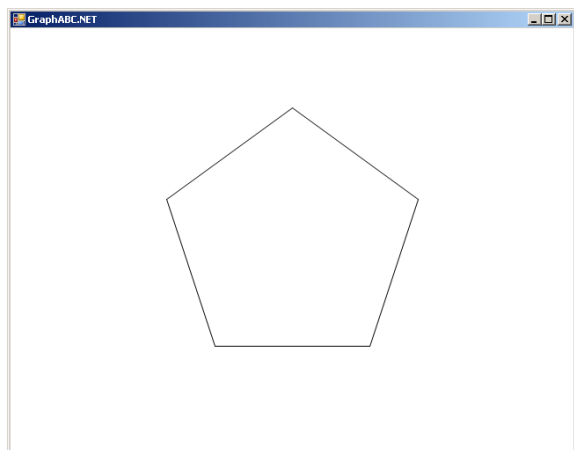
Нарисуем правильный пятиугольник, вписанный в окружность с центром (320, 240) и радиусом 150 (см. левый рисунок на следующей странице). Для рисования нам потребуются формулы, выражающие координаты точки  $(x, y)$  через расстояние



$r$  от точки до начала координат и угол  $a$  между радиус-вектором этой точки и положительной полуосью  $OX$  (см. правый рисунок на следующей странице):

$$x = r \cdot \cos a, \quad y = r \cdot \sin a.$$

В нашем случае роль начала координат играет точка (320, 240), поэтому ее координаты надо прибавить к правой части приведенных формул. Далее, в графическом окне ось  $OY$  направлена *вниз*, поэтому для первой вершины (которая на рисунке будет верхней точкой многоугольника) следует выбрать угол  $a$ , равный  $-\pi/2$ .



```
uses GraphABC;
const
  n = 5;
  x0 = 320;
  y0 = 240;
  r = 150;
begin
  var a := -Pi/2;
  var x1 := x0;
  var y1 := y0-r;
  for var i:=1 to n do
  begin
    a += 2*Pi/n;
    var x2 := x0+Round(r*Cos(a));
    var y2 := y0+Round(r*Sin(a));
    Line(x1,y1,x2,y2);
    x1 := x2;
    y1 := y2;
  end;
end.
```

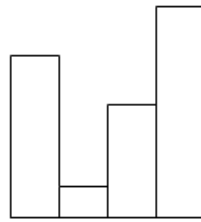
**Замечание.** Изменяя значение константы  $n$ , можно нарисовать правильный многоугольник с любым требуемым числом сторон.

### Задания

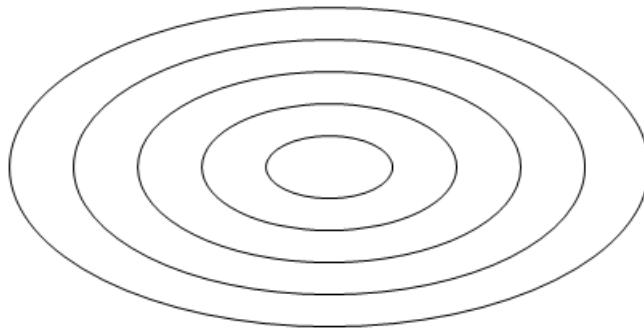
1. Ввести с клавиатуры координаты вершин треугольника  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$ . Нарисовать этот треугольник, используя графический примитив `Line`.
2. Нарисовать пилу из 8 зубцов следующего вида:



3. Нарисовать гистограмму из 4 столбцов (высота каждого столбца вводится с клавиатуры). Ниже приведен рисунок для столбцов высоты 100, 20, 70 и 130:



4. Нарисовать 5 эллипсов с общим центром следующего вида:



5. Нарисовать пятиконечную звезду.  
 6. Нарисовать солнце с лучами.  
 7. Нарисовать часы с секундной стрелкой, направленной на  $n$  секунд (значение  $n$  вводится с клавиатуры и должно лежать в диапазоне 0..59).

## 4. Перо и кисть

Рисование графических примитивов осуществляется графическими инструментами: *пером* Pen и *кистью* Brush. При этом линии рисуются с помощью пера, а если получаемая фигура имеет внутренность, то эта внутренность закрашивается кистью.

Перо и кисть имеют различные *свойства*, которые можно изменять в процессе рисования. Для доступа к свойствам надо использовать точечную нотацию.

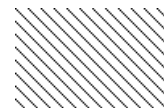
Начнем с рассмотрения свойств кисти, среди которых следует прежде всего отметить *цвет* Brush.Color и *стиль* Brush.Style. Основными стилями кисти являются bsSolid (однотонная кисть) и bsHatch (штриховая кисть). По умолчанию кисть является однотонной и имеет белый цвет.

Штриховые кисти закрашивают внутренность фигуры, нанося на нее штриховку. Если установлен штриховой стиль кисти, то вид штриховки задается свойством кисти Brush.Hatch. Его значения определяются более чем 50 именованными константами, основные из которых приведены ниже:

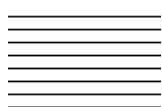
bhBDiagonal



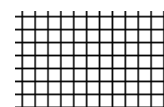
bhFDiagonal



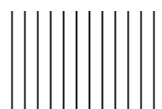
bhHorizontal



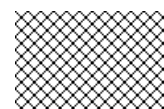
bhCross



bhVertical



bhDiagCross



Область вне штриховки закрашивается фоновым цветом штриховки (по умолчанию используется белый фоновый цвет), а сама штриховка наносится текущим цветом кисти `Brush.Color`. При этом следует иметь в виду, что при установке стиля `bsHatch` любой цвет кисти меняется на черный, а при установке стиля `bsSolid` — на белый. Например, операторы

```
Brush.Style := bsHatch;  
Brush.Hatch := bhVertical;
```

устанавливают штриховую кисть с вертикальной штриховкой; текущий цвет кисти — черный, фоновый цвет кисти — белый.

Вид штриховки `Brush.Hatch` имеет перечислимый тип `HatchStyle`. Поэтому достаточно в тексте программы ввести имя типа `HatchStyle` и точку, чтобы увидеть список всех констант типа `HatchStyle` и выбрать нужный вариант.

Для установки фонового цвета штриховки следует воспользоваться свойством `Brush.HatchBackgroundColor`. Например, следующие операторы обеспечивают настройку вертикальной штриховки зеленым цветом по желтому фону (обратите внимание на то, что свойство `Color` устанавливается *после* свойства `Style`):


```
Brush.Style := bsHatch;  
Brush.Color := Color.Green;  
Brush.Hatch := HatchStyle.Vertical;  
Brush.HatchBackgroundColor := Color.Yellow;
```

Если требуется сделать прозрачной область вне штриховки, то достаточно использовать цвет `Color.Empty`:

```
Brush.HatchBackgroundColor := Color.Empty;
```


Приведем два примера, связанных с использованием кисти.

#### Пример 1. Светофор.

```
 uses GraphABC;  
const  
    r = 50;  
    x = 100;  
begin  
    var y := 100;  
    Brush.Color := Color.Red;  
    Circle(x, y, r);  
    y += 120;  
    Brush.Color := Color.Yellow;  
    Circle(x, y, r);  
    y += 120;  
    Brush.Color := Color.Green;  
    Circle(x, y, r);  
end.
```

Программа рисует три круга, центры которых отстоят по вертикали на расстоянии 120 пикселей. Радиус и абсцисса центра у кругов одинаковые, поэтому они заданы в виде констант `r` и `x` соответственно.

#### Пример 2. Перебор штриховок.

```
 uses GraphABC;  
begin  
    Brush.Style := bsHatch;  
    Brush.Hatch := HatchStyle.Horizontal;  
    Rectangle(10, 10, 400, 400);
```

```

Sleep(1000);
Brush.Hatch := HatchStyle.DiagonalBrick;
Rectangle(10, 10, 400, 400);
Sleep(1000);
Brush.Hatch := HatchStyle.DashedVertical;
Rectangle(10, 10, 400, 400);
end.

```

В данной программе прямоугольник заполняется различными штриховками. Пауза между рисованием различных штриховок обеспечивается вызовом процедуры `Sleep(ms)`, где `ms` — количество миллисекунд (в данном случае пауза составляет 1000 миллисекунд, то есть 1 секунду).

Теперь обратимся к свойствам пера `Pen`, основными из которых являются *текущие координаты* `Pen.X` и `Pen.Y`, *цвет* `Pen.Color`, *ширина* `Pen.Width` и *стиль* `Pen.Style`. По умолчанию перо имеет черный цвет и ширину 1 пиксел.

С текущими координатами пера связаны две процедуры, с помощью которых легко рисовать ломаные линии; в частности, графики функций (см. п. 6.8, пример 3):

`MoveTo(x, y)` перемещает перо к точке с координатами  $(x, y)$ ;

`LineTo(x, y)` рисует отрезок от текущего положения пера до точки с координатами  $(x, y)$ .

После выполнения любой из этих процедур текущие координаты пера становятся равными  $(x, y)$ .

Изменяя стиль пера, можно изображать различные штриховые линии. Имеются следующие стили пера:

<code>psSolid</code>	
<code>psDash</code>	
<code>psDot</code>	
<code>psDashDot</code>	
<code>psDashDotDot</code>	
<code>psClear</code>	(линия не рисуется)

По умолчанию установлен стиль `psSolid`. Стиль можно также задавать с помощью перечислимого типа `DashStyle` с константами `Solid`, `Dash`, `Dot`, `DashDot`, `DashDotDot`.

### Пример 3. Штриховые линии.

Рассмотрим следующую программу.



```

uses GraphABC;
begin
  Pen.Width := 3;
  var y := 60;
  Pen.Color := Color.Red;
  Pen.Style := DashStyle.Dash;
  Line(50, y, 400, y);
  y += 30;
  Pen.Color := Color.Green;
  Pen.Style := DashStyle.Dot;
  Line(50, y, 400, y);

```

```

y += 30;
Pen.Color := Color.Blue;
Pen.Style := DashStyle.DashDot;
Line(50, y, 400, y);
end.

```

Данная программа рисует три горизонтальные линии шириной 3 пиксела, имеющие различные цвета и штриховые стили:



Как мы знаем, внутренность замкнутой фигуры рисуется кистью, а ее граница — пером. Иногда необходимо нарисовать только внутренность или только границу фигуры. Для каждого замкнутого графического примитива в GraphABC определены еще две процедуры: процедура с приставкой `Fill` рисует внутренность замкнутой фигуры без границы, а процедура с приставкой `Draw` рисует границу без внутренности. Например, для рисования круга можно использовать, кроме процедуры `Circle`, процедуры `FillCircle` и `DrawCircle` с теми же параметрами.

**Пример 4.** Олимпийские кольца.



В данном случае важно, чтобы внутренность колец была прозрачной, поэтому будем рисовать кольца процедурой `DrawCircle`:

```

П uses GraphABC;
const r = 27;
begin
  Pen.Width := 3;
  Pen.Color := Color.Blue;
  DrawCircle(100, 100, r);
  Pen.Color := Color.Black;
  DrawCircle(160, 100, r);
  Pen.Color := Color.Red;
  DrawCircle(220, 100, r);
  Pen.Color := Color.Yellow;
  DrawCircle(130, 130, r);
  Pen.Color := Color.Green;
  DrawCircle(190, 130, r);
end.

```

Константа `r` подобрана таким образом, чтобы кольца немного пересекались.

## Задания

1. Нарисовать шахматную доску  $n$  на  $n$  клеток с размером одной клетки  $sz$  пикселей.
2. Ввести составляющие цвета  $r$ ,  $g$ ,  $b$  и нарисовать прямоугольник заданного цвета.

3. Выбрать 4 штриховки кисти и пронумеровать их от 1 до 4. Ввести номер штриховки внутренности прямоугольника (число от 1 до 4) и вывести прямоугольник с заданной штриховкой.
4. Используя вертикальные линии, нарисовать градиент от черного к белому.

## 5. Процедура FloodFill

В растровых графических редакторах имеется инструмент, называемый *заливкой*. При его применении к некоторой точке изображения выполняется закрашивание («заливка») текущим цветом всей непрерывной области, которая содержит эту точку и имеет ее цвет. Например, если применить инструмент внутри круга, то вся его внутренность будет залита текущим цветом.

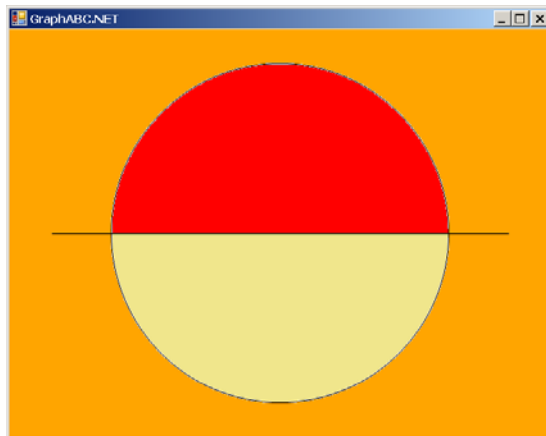
В библиотеке GraphABC для заливки областей предусмотрена процедура FloodFill. Вызов FloodFill(x, y, col) заливает цветом col непрерывную одноцветную область, содержащую точку (x, y).

**Пример.** Рассмотрим программу:



```
uses GraphABC;
begin
  Circle(320, 240, 200);
  Line(50, 240, 590, 240);
  FloodFill(320, 100, Color.Red);
  FloodFill(320, 340, Color.Khaki);
  FloodFill(0, 0, Color.Orange);
end.
```

Результатом работы программы является следующее изображение:



Первая команда FloodFill заливает красным цветом верхнюю половину пересеченного круга, вторая — цветом хаки — нижнюю половину и, наконец, внешняя область заливается оранжевым цветом с помощью третьей команды FloodFill.

### Задания

1. Дополните программу из примера 3 (п. 6.3), закрасив части круговой диаграммы и внешнюю область случайными цветами.
2. Дополните программу из примера 4 (п. 6.3), закрасив клетки поля черным цветом в шахматном порядке (начиная с левой нижней клетки).
3. В задании 4 из п. 6.3 дополнительно закрасьте центральный эллипс и кольца, ограниченные соседними эллипсами, оттенками красного цвета (интенсив-

ность цвета должна убывать в направлении от центрального эллипса к внешнему кольцу).

## 6. Вывод текста

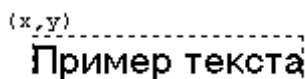
Для вывода текста используется третий графический инструмент — *Font* (*шрифт*), имеющий следующие свойства, доступ к которым, как обычно, осуществляется с помощью точечной нотации: *цвет* `Font.Color`, *размер* в пунктах `Font.Size` (10 пунктов примерно равны 3 миллиметрам), *наименование* `Font.Name` и *стиль* `Font.Style`.

По умолчанию устанавливается обычный черный шрифт размера 8 пунктов, имеющий наименование «Arial». Наиболее распространенными шрифтами, кроме «Arial», являются шрифты «Times New Roman» и «Courier New». Наименование шрифта можно набирать без учета регистра. Если указано наименование несуществующего шрифта, то выбирается шрифт «Arial».

Установка стиля шрифта производится с помощью следующих констант:

<code>fsNormal</code>	— обычный;
<code>fsBold</code>	— жирный;
<code>fsItalic</code>	— наклонный;
<code>fsBoldItalic</code>	— жирный наклонный;
<code>fsUnderline</code>	— подчеркнутый;
<code>fsBoldUnderline</code>	— жирный подчеркнутый;
<code>fsItalicUnderline</code>	— наклонный подчеркнутый;
<code>fsBoldItalicUnderline</code>	— жирный наклонный подчеркнутый.

Вывод текстовой строки в графическое окно осуществляется вызовом процедуры `TextOut(x, y, s)`. Строка `s` выводится текущим шрифтом; точка `(x, y)` определяет левый верхний угол прямоугольной области, содержащей выведенный текст:



**Пример 1.** Выведем строку 'PascalABC.NET' жирным наклонным шрифтом «Arial», используя разные размеры и случайные цвета:

```

П
uses GraphABC;
begin
    Font.Name := 'Arial';
    Font.Style := fsBoldItalic;
    var y := 10;
    for var i:=8 to 20 do
        begin
            Font.Size := i;
            Font.Color := clRandom;
            TextOut(150,y,i.ToString+' PascalABC.NET');
            y += 35;
        end;
    end.

```

Текст выводится шрифтами размера от 8 до 20 пунктов на экранных строках, отстоящих друг от друга на расстоянии 35 пикселей. В начале каждой текстовой

строки выводится размер шрифта. Для преобразования размера шрифта из целого числа в строку используется конструкция `i.ToString`, возвращающая строковое представление целого числа (эта возможность `PascalABC.NET` отсутствует в традиционном Паскале, однако имеется во всех языках платформы .NET).

Иногда в программе требуется определить размер в пикселах, который будет иметь выведенный текст. Для этого используются следующие функции:

`TextWidth(s)` — функция, возвращающая ширину строки `s` в пикселах при текущих настройках шрифта;

`TextHeight(s)` — функция, возвращающая высоту строки `s` в пикселах при текущих настройках шрифта.

**Пример 2.** Выведем строку `'PascalABC.NET'` в окаймляющем прямоугольнике:



```
uses GraphABC;
begin
    Font.Name := 'Times New Roman';
    Font.Size := 40;
    var s := 'PascalABC.NET';
    var w := TextWidth(s);
    var h := TextHeight(s);
    TextOut(90, 150, s);
    DrawRectangle(90, 150, 90+w, 150+h);
end.
```

Для окаймления строки вычислим вначале ее ширину и высоту, предварительно установив все свойства шрифта. После этого выведем текст с помощью `TextOut`, а затем нарисуем контур прямоугольника, используя `DrawRectangle`.

## Задания

1. Ввести в диалоге текстовую строку (в переменную типа `string`), размер шрифта, а также номер имени шрифта (1 — Times New Roman, 2 — Arial, 3 — Courier New), после чего вывести эту строку в левом верхнем углу графического окна, используя шрифт черного цвета с указанными свойствами.
2. Определить, каким символам на клавиатуре соответствуют следующие символы шрифта «Symbol»:

⊗ ⊕ ⊘  
α β γ δ

**Подсказка:** после установки шрифта с наименованием «Symbol» выведите большие и маленькие буквы русского и латинского алфавита.

3. Написать свое имя разноцветными буквами.
4. Секундомер. Последовательно выводить в одной и той же позиции графического окна числа от 0 до 59, отображая каждое число в течение 1 секунды. Для вывода использовать шрифт большого размера, для преобразования числа `n` в строку использовать конструкцию `n.ToString`.



## 7. Действия с графическим окном

Для настройки *графического окна* предназначена переменная `Window`, которая имеет следующие свойства:

`Window.Width` — ширина *клиентской части* графического окна (клиентская часть окна не включает его заголовок и рамку);

`Window.Height` — высота клиентской части графического окна;

`Window.Left` — отступ графического окна от левого края экрана;

`Window.Top` — отступ графического окна от верхнего края экрана;

`Window.Title` — текст, отображаемый в заголовке графического окна (по умолчанию в заголовке выводится текст 'GraphABC.NET').

Кроме того, для переменной `Window` с помощью точечной нотации можно вызывать так называемые *методы* — процедуры, связанные с этой переменной:

`Window.Clear` очищает графическое окно, закрашивая его белым цветом;

`Window.Clear(col)` очищает графическое окно, закрашивая его цветом `col`;

`Window.SetSize(w,h)` устанавливает новые размеры клиентской части графического окна в пикселах (`w` — ширина клиентской части окна, `h` — ее высота);

`Window.SetPos(l,t)` устанавливает отступ графического окна от левого верхнего края экрана в пикселах (`l` — отступ слева, `t` — отступ сверху);

`Window.Save(fname)` сохраняет содержимое графического окна в файле с именем `fname` (независимо от расширения файла используется графический формат PNG);

`Window.Load(fname)` отображает в графическом окне содержимое файла с именем `fname` (файл может иметь любой из распространенных графических форматов, в частности, BMP, JPEG, PNG, GIF); после загрузки изображения графическое окно изменяет размеры в соответствии с размером изображения;

`Window.Close` закрывает графическое окно и завершает приложение;

`Window.CenterOnScreen` размещает графическое окно по центру экрана;

`Window.Minimize` сворачивает графическое окно;


`Window.Maximize` максимизирует графическое окно;

`Window.Normalize` восстанавливает исходное положение и размеры графического окна, если оно ранее было свернуто или максимизировано.

**Пример 1.** Нарисуем две диагонали графического окна и окружность с центром в центре окна, касающуюся ближайших противоположных сторон окна.

Одна из диагоналей должна соединять точки  $(0, 0)$  и  $(\text{Window.Width}, \text{Window.Height})$ , а вторая — точки  $(0, \text{Window.Height})$  и  $(\text{Window.Width}, 0)$ . Окружность имеет центр  $(\text{Window.Width} \div 2, \text{Window.Height} \div 2)$  и радиус, равный минимальному из значений  $\text{Window.Width} \div 2$  и  $\text{Window.Height} \div 2$ . Для того чтобы внутренность окружности не заслоняла часть диагоналей, следует вначале нарисовать окружность, а затем — диагонали.

Приведем текст программы:

```
 uses GraphABC;  
begin  
    var w := Window.Width;  
    var h := Window.Height;
```

```

Circle(w div 2,h div 2,Min(w div 2,h div 2));
Line(0,0,w,h);
Line(0,h,w,0);

```

**end.**

Для нахождения минимального значения мы воспользовались функцией `Min`.

**Пример 2.** Плавное изменение размеров окна.

Будем увеличивать в цикле высоту окна и уменьшать ширину:



```

uses GraphABC;
begin
  for var i:=1 to 200 do
    begin
      SetWindowSize(Window.Width-1,Window.Height+1);
      Sleep(10);
    end;
  end.

```

Заметим, что даже если увеличить количество итераций цикла, сделать ширину окна нулевой не удастся, поскольку в заголовке окна обязательно должна отображаться иконка и кнопки минимизации, максимизации/восстановления и закрытия окна. Высоту клиентской части окна можно сделать нулевой.

**Пример 3.** Заполнение окна мозаикой из квадратов.

Пусть квадрат имеет сторону `sz`, задаваемую константой. Очевидно, по горизонтали можно расположить `Window.Width div sz` полных квадратов, а по вертикали — `Window.Height div sz`. Нарисуем по горизонтали и вертикали на один квадрат больше, начав нумерацию с 0, чтобы заполнить оставшуюся часть окна:



```

uses GraphABC;
const sz = 30;
begin
  for var x:=0 to Window.Width div sz do
    for var y:=0 to Window.Height div sz do
      begin
        Brush.Color := clRandom;
        Rectangle(x*sz,y*sz,x*sz+sz+1,y*sz+sz+1);
      end;
    end.

```

Каждый квадрат рисуется процедурой `Rectangle`, при этом он закрашивается случайным цветом. Левый верхний угол задается координатами  $(x*sz, y*sz)$ , где  $x$  и  $y$  меняются от нуля. Координаты правого нижнего угла больше координат левого верхнего угла на величину `sz+1` (благодаря добавке в 1 пиксел границы смежных квадратов совмещаются).

Интересно отметить, что если увеличить размеры окна, зацепив мышью его правый нижний угол и перетаскив его вправо и вниз, то в окне появятся ранее скрытые части квадратов, находящихся в последнем ряду по горизонтали и вертикали. Это означает, что область рисования не ограничивается размерами графического окна.

**Пример 4.** Сохранение содержимого окна.

Нарисуем 1000 случайных прямоугольников и сохраним содержимое окна в файле. После этого очистим окно и через 1 секунду восстановим его прежнее содержимое.



```
uses GraphABC;
begin
  for var i:=1 to 1000 do
  begin
    Brush.Color := clRandom;
    var x := Random(Window.Width);
    var y := Random(Window.Height);
    var w := Random(50)+10;
    var h := Random(50)+10;
    Rectangle(x,y,x+w,y+h);
  end;
  Window.Save('pic.png');
  Sleep(1000);
  Window.Clear;
  Sleep(1000);
  Window.Load('pic.png');
end.
```

**Задания**

1. Составьте программу, перемещающую графическое окно по экрану.
2. Заполните графическое окно мозаикой из эллипсов.
3. В цикле максимизируйте и минимизируйте окно 10 раз с интервалом 1 с.
4. Организуйте «цветомузыку» графического окна, очищая его с использованием случайного цвета (чтобы предотвратить слишком быстрое мелькание цветов, применяйте процедуру Sleep).
5. Нарисуйте российский флаг, занимающий всю клиентскую часть графического окна. Для этого достаточно нарисовать три прямоугольника одинакового размера с требуемыми фоновыми цветами (сверху вниз: белый, синий, красный).

**8. Система координат**

В графическом окне можно изменять систему координат, получая при этом интересные визуальные эффекты. Текущая система координат определяется переменной `Coordinate`, имеющей следующие свойства и методы:

`Coordinate.OriginX` –  $X$ -координата начала координат относительно левого верхнего угла клиентской области графического окна;

`Coordinate.OriginY` –  $Y$ -координата начала координат относительно левого верхнего угла клиентской области графического окна;

`Coordinate.Angle` – угол поворота системы координат (в градусах);

`Coordinate.Scale` – масштаб системы координат;

`Coordinate.SetOrigin(x0,y0)` – устанавливает начало системы координат относительно левого верхнего угла клиентской области графического окна;

`Coordinate.SetMathematic` – устанавливает оси системы координат так, что ось  $OY$  направлена вверх, а ось  $OX$  – вправо;

`Coordinate.SetStandard` – устанавливает оси системы координат так, что ось *OY* направлена вниз, а ось *OX* – вправо.

**Пример 1.** Прямоугольник и окружность в центре окна.

Установим начало координат в центре графического окна — точке с координатами (`Window.Width div 2, Window.Height div 2`). Прямоугольник в центре экрана теперь будет симметричен относительно начала координат, поэтому координаты его левой и правой сторон, а также координаты верхней и нижней сторон будут различаться только знаком, например,  $-200, +200$  и  $-100, +100$  соответственно. Окружность, размещенная в центре окна, теперь будет иметь нулевые координаты центра; положим ее радиус равным 50:

```

П1 uses GraphABC;
begin
  Coordinate.SetOrigin(Window.Width div 2,
    Window.Height div 2);
  Rectangle(-200, -100, 200, 100);
  Circle(0, 0, 50);
end.
```

Заметим, что для получения той же самой картинке в другом месте экрана достаточно изменить параметры метода `SetOrigin`, не меняя параметры процедур, выполняющих непосредственное рисование.

Что произойдет, если убрать в параметрах `SetOrigin` добавки `div 2`? А если положить эти параметры равными 0? В каком из этих случаев удастся увидеть всю нарисованную картинку (изменив предварительно размеры окна)?

**Пример 2.** Поворот с рисованием: циферблат с делениями.

Поместим начало координат в центр экрана и нарисуем большую окружность-циферблат по центру экрана, а также маленькую окружность в правой части циферблата — деление на цифре «3»:

```

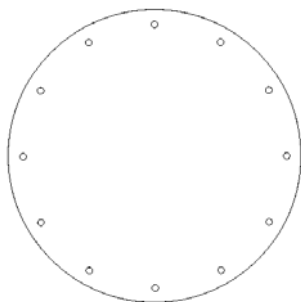
П1 uses GraphABC;
begin
  Coordinate.SetOrigin(Window.Width div 2,
    Window.Height div 2);
  Circle(0, 0, 200);
  Circle(180, 0, 5);
end.
```

Убедившись, что деление на циферблате рисуется правильно, поместим процедуру рисования деления в цикл **for**, который будет выполняться 12 раз, и добавим в этот цикл команду поворота системы координат на  $360/12$  градусов:

```

П2 uses GraphABC;
begin
  Coordinate.SetOrigin(Window.Width div 2,
    Window.Height div 2);
  Circle(0, 0, 200);
  for var i:=1 to 12 do
  begin
    Circle(180, 0, 5);
    Coordinate.Angle += 360/12;
  end;
end.
```

После запуска нового варианта программы получим следующее изображение:



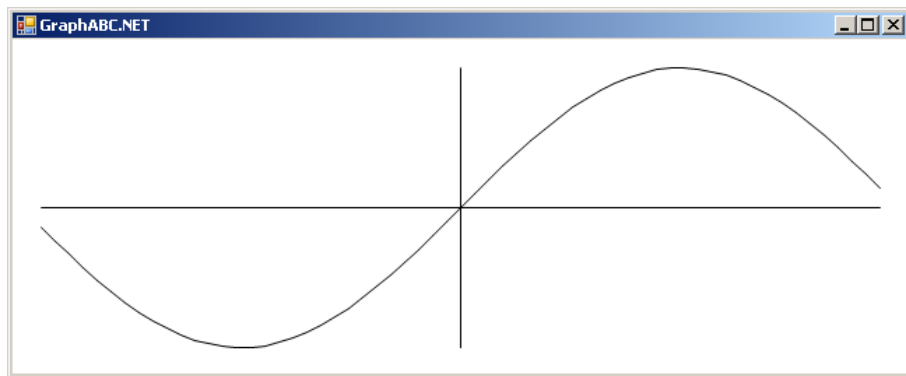
### Пример 3. Рисование графика функции.

Нарисуем график функции  $\sin x$ , изменяя значение аргумента  $x$  на промежутке от  $-3$  до  $3$  с шагом  $0.1$ . Поскольку координаты графического окна измеряются в пикселах, перед рисованием очередной линии графика будем умножать аргумент  $x$  функции и ее значение  $y = \sin x$  на масштабный множитель  $100$ , сохранив его в константе  $m$ .



```
uses GraphABC;
const m = 100;
begin
    Coordinate.SetMathematic;
    Coordinate.SetOrigin(Window.Width div 2, m+20);
    Line(-3*m, 0, 3*m, 0);
    Line(0, -m, 0, m);
    var x := -3.0;
    var y := Sin(x);
    MoveTo(Round(m*x), Round(m*y));
    while x < 3.01 do
        begin
            LineTo(Round(m*x), Round(m*y));
            x += 0.1;
            y := Sin(x);
        end;
    end.
```

В данном случае для получения «правильного» графика необходимо установить математическую систему координат (ось  $OY$  направлена вверх).



### Задания

1. В центре графического окна нарисовать квадрат, повернутый на  $45$  градусов.
2. Дополнить программу из примера 2, нарисовав циферблат с секундной стрелкой, направленной на  $n$  минут (целочисленное значение  $n$  вводится с клавиатуры).

- туры, оно должно принадлежать диапазону 0..59). Использовать поворот координат.
- Используя преобразование системы координат (сдвиг центра и изменение ориентации оси  $OY$ ), а также процедуры `MoveTo` и `LineTo`, упростить программу рисования правильного многоугольника (см. п. 6.3, пример 5).
  - Вывести строку 'PascalABC.NET' разными цветами и под разными углами:



## 9. Анимация

Под *анимацией* в графических приложениях понимается изменение содержимого графического окна с течением времени. В этом случае в графическом окне располагается один или несколько объектов, свойства которых меняются. Обычно объект меняет свое положение, но может менять также форму, цвет и размер. Ранее мы уже разрабатывали программы с анимацией, хотя и не использовали этот термин (см. пример 2 из п. 6.4, а также пример 2 из п. 6.7, в котором изменялись свойства не графических объектов, а самого графического окна).

Если при изменении объекта его размер и положение не меняются, то для реализации анимации достаточно изменять нужные свойства объекта и перерисовывать его, делая после этого паузу. Однако при изменении положения или размера объекта обычно требуется предварительно стереть его прежнее изображение, что может приводить к возникновению такого эффекта, как *мерцание*. Продемонстрируем это на простом примере.

**Пример 1.** Перемещение круга (с мерцанием).

Нарисуем зеленый круг и будем перемещать его в цикле в направлении слева направо (то есть в направлении увеличения координаты  $x$ ). При этом необходимо стирать прежнее изображение, так как иначе за кругом будет оставаться «след». Простейший способ стереть старое изображение — это очистить все графическое окно, вызвав его метод `Clear` (заметим, что этот способ особенно удобен при одновременной анимации нескольких объектов):



```
uses GraphABC;
begin
  Brush.Color := clGreen;
  for var x:=70 to 500 do
    begin
      Window.Clear;
      Circle(x, 200, 50);
      Sleep(3);
    end;
  end.
```


При выполнении данной программы происходит заметное мерцание движущегося круга. Причина мерцания заключается в том, что в момент стирания круг пол-

ностью исчезает, и, таким образом, та область, которую он занимает на экране, попеременно закрашивается то зеленым, то белым цветом. Отметим, что мерцание будет проявляться даже при анимации *белого* круга, поскольку и в этом случае его граница будет то стираться, то рисоваться заново.

В библиотеке GraphABC имеются процедуры LockDrawing и Redraw, специально предназначенные для подавления мерцания в анимированных графических приложениях. Вызов процедуры LockDrawing блокирует дальнейшее рисование непосредственно на экране, но рисование, тем не менее, осуществляется в специальном *внеэкранный буфере* (то есть в некоторой области оперативной памяти). Чтобы вывести содержимое внеэкранный буфера на экран, следует вызвать процедуру Redraw. При этом прежнее изображение будет заменено на новое *без промежуточной очистки экрана*, и поэтому мерцания не возникнет (подчеркнем, что очистка изображения на каждом шаге анимации по-прежнему производится, однако это делается во внеэкранный буфере).

### Пример 2. Перемещение круга без мерцания.

Для подавления мерцания достаточно добавить в программу из предыдущего примера два оператора: вызов процедуры LockDrawing непосредственно перед циклом **for** и вызов процедуры Redraw в цикле перед процедурой Sleep:

```
 uses GraphABC;  
begin  
  Brush.Color := clGreen;  
  LockDrawing;  
  for var x:=70 to 500 do  
    begin  
      Window.Clear;  
      Circle(x,200,50);  
      Redraw;  
      Sleep(3);  
    end;  
end.
```

Таким образом, теперь в цикле вначале выполняется очистка изображения во внеэкранный буфере, затем в этом же буфере рисуется круг (в новой позиции), после чего полученное изображение копируется на экран процедурой Redraw, и выполнение программы приостанавливается на указанный промежуток времени.

## Задания

При выполнении всех заданий необходимо устранять мерцание.

1. Реализовать пульсацию круга, расположенного в центре графического окна: вначале круг увеличивается в размере, пока не достигнет ближайших границ окна, затем сжимается до точки в центре окна.
2. Дополнить пульсацию круга, описанную в задании 1, постепенным изменением цвета его фона: вначале круг имеет красный фон максимальной интенсивности (255), затем, по мере увеличения его размера, интенсивность красного цвета уменьшается до 0, а при сжатии круга — опять увеличивается.
3. Организовать перемещение круга по периметру графического окна.
4. Организовать перемещение круга в произвольном прямолинейном направлении с отражением от границ графического окна.

## 10. Рисунки

Для работы с рисунками в библиотеке GraphABC необходимо использовать переменные типа `Picture`:

```
var p: Picture;
```

После описания такой переменной ее обязательно следует *инициализировать*, связав или с каким-либо существующим рисунком, или с пустым рисунком указанного размера. Для этого достаточно вызвать один из двух вариантов процедуры `CreatePicture`. Вариант данной процедуры с параметрами `p` типа `Picture` и `fname` типа `string` позволяет связать с переменной `p` существующий рисунок, содержащийся в файле с именем `fname`, например:

```
CreatePicture(p, 'pic.bmp');
```

Если указанный файл отсутствует или его формат не соответствует ни одному из допустимых графических форматов, то возникнет ошибка времени выполнения.

Вариант процедуры `CreatePicture` с параметрами `p` типа `Picture` и `w, h` типа `integer` позволяет создать пустой рисунок шириной `w` и высотой `h` пикселей, например:

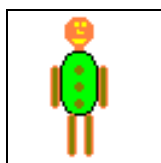
```
CreatePicture(p, 30, 30);
```

После инициализации переменной типа `Picture` для нее можно вызывать различные свойства и методы, используя точечную нотацию.

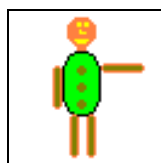
Основными свойствами рисунка `p` являются его *размеры*: ширина `p.Width` и высота `p.Height`. С помощью этих свойств можно не только определять текущий размер рисунка, но и изменять его, причем уменьшение размеров приводит к тому, что часть рисунка пропадает.

Основным методом для рисунка `p` является метод `p.Draw(x, y)`, который выводит рисунок `p` в графическое окно; при этом параметры `x, y` задают координаты левого верхнего угла рисунка.

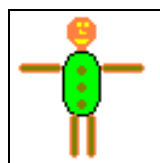
В примерах к данному пункту мы будем использовать набор из четырех рисунков в формате PNG размера 64 на 64 пиксела:



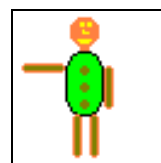
man0.png



man1.png



man2.png



man3.png

Файлы с этими рисунками можно скопировать в рабочий каталог из подкаталога `Images` системного каталога `PascalABC.NET` (этот каталог обычно имеет имя `PascalABC.NET` и находится в каталоге `C:\Program Files`).

**Пример 1.** Голова человечка.



```
uses GraphABC;
begin
    var p: Picture;
    CreatePicture(p, 'man0.png');
    p.Height := 17;
    p.Draw(0, 0);
end.
```

На экране появится изображение головы без туловища:





Два оставшихся свойства типа `Picture` связаны с *прозрачностью*. Для любого рисунка `p` можно задать режим прозрачности, установив его свойство `p.Transparent` равным `True`. В результате те части рисунка, которые имеют белый цвет, будут считаться прозрачными, и «сквозь них» будет просвечивать прежнее содержимое графического окна.

**Пример 2.** Человечки, стоящие рядом.

```

П uses GraphABC;
begin
    var p: Picture;
    CreatePicture(p, 'man0.png');
    p.Transparent := True;
    p.Draw(0, 0);
    p.Draw(26, 0);
end.

```

На экране появится изображение двух человечков (см. рисунок слева). Заметим, что если закомментировать оператор, изменяющий свойство `Transparent`, то мы получим картинку, изображенную справа.



Иногда требуется сделать прозрачным другой цвет рисунка `p`. В этом случае надо присвоить значение этого цвета свойству `p.TransparentColor` (по умолчанию свойство `TransparentColor` равно `clWhite`).

Кроме уже рассмотренного метода `Draw` к часто используемым методам для рисунков `p` типа `Picture` можно отнести следующие:

`p.Load(fname)` – загрузить в рисунок `p` изображение из файла `fname`;

`p.Save(fname)` – сохранить рисунок `p` в файле `fname`;

`p.FlipHorizontal` – зеркально отразить рисунок `p` относительно горизонтальной оси симметрии;

`p.FlipVertical` – зеркально отразить рисунок `p` относительно вертикальной оси симметрии.

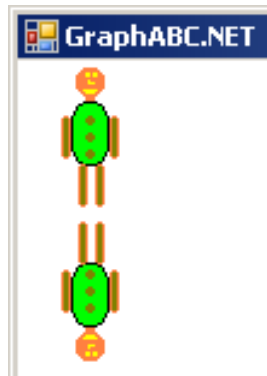
**Пример 3.** Отражение в воде.

```

П uses GraphABC;
begin
    var p: Picture;
    CreatePicture(p, 'man0.png');
    p.Draw(0, 0);
    p.FlipVertical;
    p.Draw(0, 64);
end.

```

На экране появится изображение человечка вместе с его отражением.



Последняя, самая многочисленная группа методов типа `Picture` дублирует все графические примитивы (см. п. 6.3) и другие процедуры, связанные с рисованием (например, `SetPixel` и `TextOut`). Если вызвать эти процедуры для переменной `p` типа `Picture` (например, `p.Circle(10,10,5)`), то соответствующее изображение будет создано не в графическом окне, а на рисунке, связанном с этой переменной.

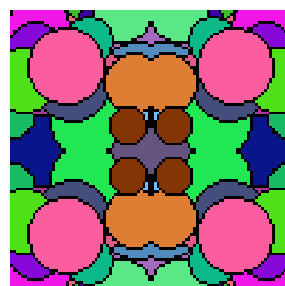
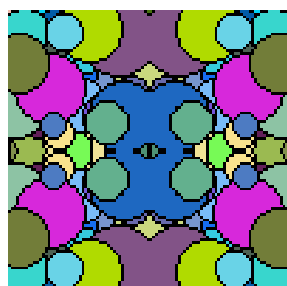
#### Пример 4. Калейдоскоп.

Если случайным образом расположить на рисунке достаточно большое количество разноцветных кругов, а затем зеркально отразить его по вертикали, по горизонтали и одновременно по вертикали и горизонтали, то получится симметричная картинка, подобная тем, которые мы видим в калейдоскопе. Приведем программу, генерирующую такие изображения:



```
uses GraphABC;
begin
  var p: Picture;
  CreatePicture(p, 50, 50);
  for var i:=1 to 100 do
    begin
      Brush.Color := clRandom;
      p.Circle(Random(50), Random(50), Random(10)+5);
    end;
    p.Draw(0, 0);
    p.FlipVertical;
    p.Draw(0, 50);
    p.FlipHorizontal;
    p.Draw(50, 50);
    p.FlipVertical;
    p.Draw(50, 0);
  end.
```

Итоговое изображение состоит из четырех квадратных частей размера 50 на 50 пикселей — исходного рисунка и трех его отражений. Приведем два образца полученных изображений:




Рисунки можно с успехом применять при разработке анимационных приложений. Рассмотрим два простых примера.

#### **Пример 5.** Зарядка.

Если последовательно изображать в одной и той же позиции графического окна рисунки человечков, приведенные на с. 170, то возникнет впечатление, что это один человечек, «размахивающий руками».

В начале программы инициализируем все четыре рисунка. В цикле будем находить остаток от деления параметра цикла на 4 и, в зависимости от результата (равного 0, 1, 2 или 3), будем определять картинку, которую требуется нарисовать в данный момент (для этого удобно использовать оператор выбора). Останется вызвать для выбранного рисунка метод Draw и сделать паузу:

```
 uses GraphABC;  
begin  
  var p,p0,p1,p2,p3: Picture;  
  CreatePicture(p0, 'man0.png');  
  CreatePicture(p1, 'man1.png');  
  CreatePicture(p2, 'man2.png');  
  CreatePicture(p3, 'man3.png');  
  for var i:=1 to 100 do  
    begin  
      case i mod 4 of  
        0: p := p0;  
        1: p := p1;  
        2: p := p2;  
        3: p := p3;  
      end;  
      p.Draw(0,0);  
      Sleep(200);  
    end;  
end.
```

Обратите внимание на то, что переменной типа Picture можно присвоить другую переменную этого же типа; в этом случае обе переменные будут связаны с одним и тем же рисунком.

Заметим также, что мерцание в нашей программе возникать не будет, так как в ней не используются специальные процедуры для стирания изображений (при выводе очередного изображения предыдущее, находящееся на том же месте, стирается автоматически).

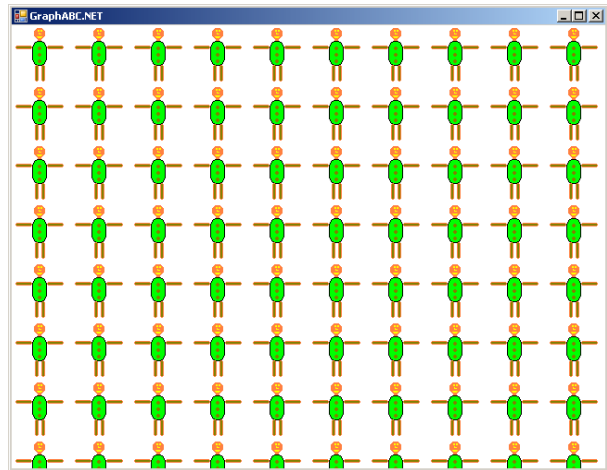
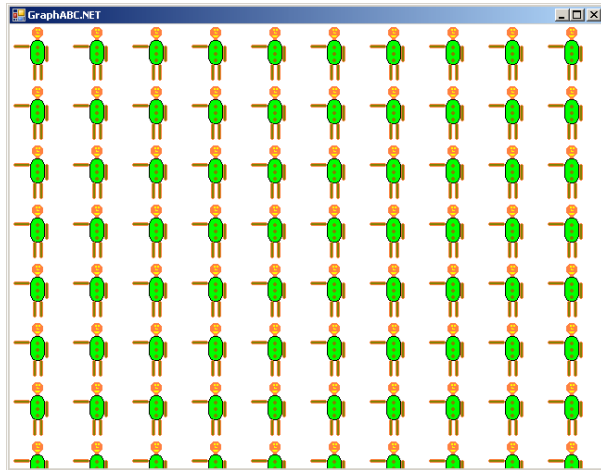
#### **Пример 6.** Зарядка в школе.

Впечатление от предыдущей анимационной картинки можно значительно усилить, если «размножить» ее по всему графическому окну. Для этого достаточно заменить в программе оператор p.Draw(0,0) на следующий двойной цикл:

```
for var w:=0 to Window.Width div 64 do  
  for var h:=0 to Window.Height div 64 do  
    p.Draw(w*64,h*64);
```

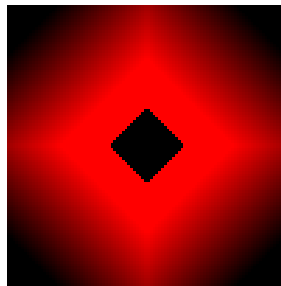
В этом фрагменте мы учли, что рисунки имеют размер 64 на 64 пиксела.

Приведем два «моментальных снимка» окна при выполнении программы.



### Задания

1. Модифицировать пример 4 («Калейдоскоп»), покрыв симметричным узором все графическое окно.
2. Используя прием программы «Калейдоскоп» и заполняя рисунок-образец с помощью метода `p.SetPixel`, создать следующее изображение, состоящее из красных оттенков разной интенсивности:



**Указание.** Для заполнения рисунка-образца размера 50 на 50 пикселей организуйте двойной цикл по  $x$  и  $y$ , изменяя их от 0 до 49, и на каждой итерации задавайте цвет пиксела  $(x, y)$  по следующей формуле:  $RGB(3 * (x+y), 0, 0)$ . Как будет изменяться картинка при изменении множителя 3 на 4, 5, 10, 50?

3. Модифицировать пример 6 «Зарядка в школе», сделав так, чтобы человечки маршировали, перемещаясь по графическому окну сверху вниз («Военный парад»). Для этого достаточно изменить второй параметр метода `Draw`.
4. Модифицировать пример 6 «Зарядка в школе», сделав так, чтобы человечки размахивали руками как кому вздумается («Непослушные ученики»). Для этого достаточно перенести оператор выбора внутрь двойного цикла и в качестве его переключателя использовать функцию `Random(4)`.