ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

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

В.В. Махно, С.С. Михалкович, М.В. Пучкин

Основы программирования графики

Часть 1. Базовые возможности

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

для преподавателей

факультета математики, механики и компьютерных наук,

ведущих курсы по основам программирования

Ростов-на-Дону

2007

Введение

Настоящие методические указания предназначены для преподавателей, использующих для обучения программированию элементарные графические средства. Дан краткий обзор таких средств в современных языках программирования, их сравнительная характеристика, после чего подробно представлена графическая библиотека GraphABC системы программирования PascalABC.NET (сайт http://pascalabc.net).

Обзор современных графических библиотек

Кратко рассмотрим графические возможности стандартных библиотек языков Delphi, C#. Отметим, что все, относящееся к C#, имеет отношение также ко всем .NET-языкам, поскольку они пользуются единой библиотекой классов FCL – Foundation Class Library).

В Delphi возможности графики сосредоточены в классе `TCanvas` и являются оберткой над функциями GDI (Graphics Device Interface) операционной системы Windows. Данный интерфейс – старый, не содержит множества современных возможностей. При создании платформы .NET (на которую ориентирован язык C#) фирма Microsoft разработала новую графическую библиотеку – GDI+, содержащую множество новых возможностей. В числе этих возможностей – расширенные перья и кисти (градиентные, с расширенными возможностями штриховок), возможность работы с рисунками всех современных форматов, прозрачность, сглаживание.

Следует отметить, что для того чтобы воспользоваться возможностями графики, необходимо создать оконное приложение с главной формой. Для начинающего программиста, привыкшего писать последовательные программы, такой способ сложен. Именно поэтому для языка PascalABC.NET, также основанного на .NET, была создана графическая библиотека GraphABC, опирающаяся на возможности GDI+ и позволяющая использовать графику в последовательных программах.

Вот как выглядит простейшая программа для GraphABC:

```
uses GraphABC;
begin
  Brush.Color := clYellow;
  Pen.Width := 3;
  Rectangle(10,10,WindowWidth-10,WindowHeight-10);
end.
```

Отметим, что, несомненно «за кадром» создается главная форма, однако, для разработчика программа выглядит как консольная, графические команды при этом сосредоточены внутри блока begin/end основной программы.

Для сравнения приведем отрезок программы на Delphi, выполняющий то же самое:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Brush.Color := RGB(200,100,50);
  Canvas.Pen.Width := 3;
  Canvas.Rectangle(10,10,ClientWidth-10,ClientHeight-10);
end;
```

Данный код требует знания основных понятий объектно-ориентированного, событийного программирования и содержит упоминание классов `TForm1` и `TObject`, а также свойства `Canvas`, в котором сосредоточены все методы рисования. Для начинающего программиста подобный код является сложным.

Аналогичный код программы на C# выглядит еще сложнее:

```
private void Form1_Paint(object snd,PaintEventArgs e)
{
  Brush b=new SolidBrush(Color.FromArgb(200,100,50));
  e.Graphics.FillRectangle(b,10,10,
    ClientSize.Width-10,ClientSize.Height-10);
  Pen p = new Pen(Color.Black,3);
  e.Graphics.DrawRectangle(p,10,10,
    ClientSize.Width-10,ClientSize.Height-10);
}
```

Данный код вводит еще большее количество классов: `PaintEventArgs`, `Pens`, `ClientSize` и свойства `Graphics`, в котором сосредоточены все методы рисования. Кроме того, код на C# иллюстрирует еще одну особенность: в библиотеке GDI+ графика является графикой *без сохранения состояния*. Это оз-

начает, что рисование графических примитивов производится не текущими пером, кистью и шрифтом, а перо, кисть или шрифт передаются всякий раз как параметр при вызове метода рисования. В некоторых ситуациях это облегчает восприятие: в месте вызова сразу видны все атрибуты рисования. Но в основном код усложняется.

Положение усугубляется тем, что при рисовании замкнутых фигур надо использовать как перо, так и кисть. Поэтому в библиотеке GDI+ принято решение разделить рисование границы и внутренности замкнутых областей. Методы рисования границы при этом имеют приставку `Draw`, а методы рисования внутренности – приставку `Fill`. Первым параметром в методы рисования границы передается перо, а в методы рисования внутренности – кисть.

Вторая проблема библиотеки GDI+ состоит в том, что перья, кисти и шрифты являются неуправляемыми ресурсами, работа с которыми в .NET затруднена тем, что момент их освобождения в .NET непредсказуем. Поэтому создание многочисленных перьев и кистей при каждом рисовании интенсивно расходует память.

Две указанные проблемы непосильны для начинающего программиста. Именно поэтому на основе GDI+ разработан модуль GraphABC, в котором данные проблемы решаются внутри, а на уровне интерфейса пользователь не знает о их существовании.

Кроме того, модуль GraphABC решает еще одну проблему – нарисованное в любом месте программы изображение автоматически восстанавливается на экране при перерисовке окна. В Delphi и C# для этого приходится прилагать дополнительные усилия.

Графические примитивы

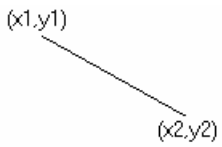
Простейшие объекты, которые можно рисовать с помощью графических процедур, называются *графическими примитивами*. К ним относятся точка, отрезок, прямоугольник, окружность, эллипс, дуга, сектор, сегмент и текст. По умолчанию все незамкнутые графические примитивы и граница замкнутых ри-

суются черным пером толщиной в 1 пиксел, а внутренность замкнутых – белой сплошной кистью.

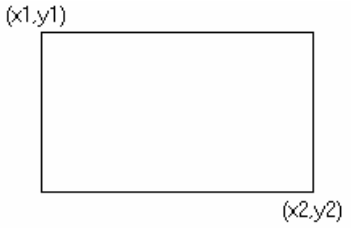
Приведем все процедуры рисования графических примитивов из модуля `GraphABC`.

`SetPixel(x,y,c)` – рисование пиксела цветом `c` точке `(x1,y1)` и концом в точке `(x2,y2)`;

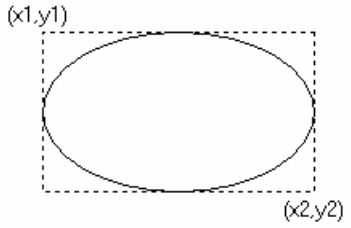
`Line(x1,y1,x2,y2)` – рисование отрезка с началом в точке `(x1,y1)` и концом в точке `(x2,y2)`;



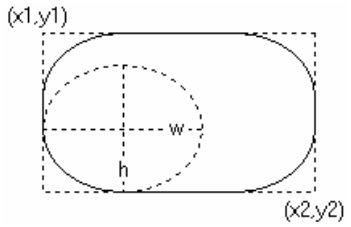
`Rectangle(x1,y1,x2,y2)` – рисование прямоугольника, заданного координатами противоположных вершин `(x1,y1)` и `(x2,y2)`;



`Ellipse(x1,y1,x2,y2)` – рисование эллипса, заданного своим описанным прямоугольником с координатами противоположных вершин `(x1,y1)` и `(x2,y2)`;

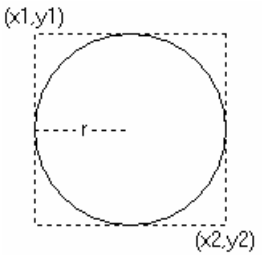


`RoundRect(x1,y1,x2,y2,w,h)` – рисование прямоугольника со скругленными краями; координаты `(x1,y1)` и `(x2,y2)` задают пару противоположных вершин, а числа `w` и `h` — ширину и высоту эллипса, используемого для скругления



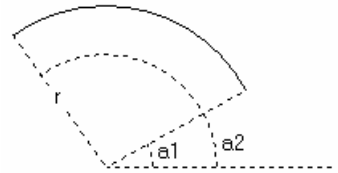
краев;

`Circle(x,y,r)`



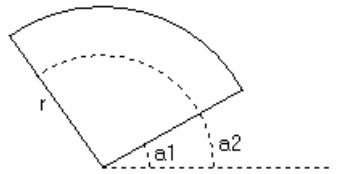
– рисование окружности с центром в точке `(x,y)` и радиусом `r`; результат выполнения этой процедуры аналогичен результату выполнения процедуры `Ellipse(x-r, y-r,x+r,y+r)`;

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



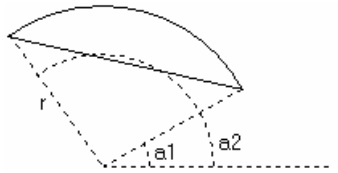
– рисование дуги окружности с центром в точке `(x,y)` и радиусом `r`, заключенной между двумя лучами, образующими углы `a1` и `a2` с осью *OX* (`a1` и `a2` — вещественные, задаются в *градусах* и отсчитываются против часовой стрелки);

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



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

`Chord(x,y,r,a1,a2)`



– рисование сегмента окружности, ограниченного дугой окружности и отрезком, соединяющим ее концы; смысл параметров — тот же, что и для процедуры `Arc`;

`TextOut(x,y,s)`

– вывод строки `s` в графическое окно; текст выводится текущим *шрифтом*; точ-

	ка `(x,y)` определяет левый верхний угол прямоугольной области, содержащей выведенный текст.
`Polygon(a: array of Point)`	– вывод многоугольника, заданного массивом точек `a`
`PolyLine(a: array of Point)`	– вывод ломаной, заданной массивом точек `a`

Цвет. Прозрачность

Цвет задается типом `Color` из пространства имен `System.Drawing`. В GraphABC он переопределен:

```
type Color = System.Drawing.Color;
```

Произвольный *цвет* формируется с помощью функции `RGB(r,g,b)`. Она возвращает целое значение, являющееся кодом цвета, который содержит красную (*red*), зеленую (*green*) и синюю (*blue*) составляющие 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(0,255,255)` — желтый. Таким образом, в нашем распоряжении имеется $256^3 = 16777216$ различных цветов.

Имеется также возможность задать прозрачную компоненту цвета, воспользовавшись функцией `ARGB(a,r,g,b)`: здесь `a` – целое в диапазоне 0..255, 255 означает полностью непрозрачный цвет, 0 – полностью прозрачный.

Из цвета `color` можно выделить красную, зеленую и синюю составляющие, используя функции `GetRed(color)`, `GetGreen(color)`, `GetBlue(color)`, а также прозрачную составляющую, используя функцию `GetAlpha(color)`. Например, если цвет `c` создан с помощью оператора

`c:=RGB(50,100,150)`, то `GetRed(c)` возвратит число `50`, `GetGreen(c)` возвратит `100` и `GetBlue(c)` возвратит `150`.

В модуле `GraphABC` имеется ряд предопределенных констант для обозначения стандартных цветов. Их более сотни, поэтому ниже приводится их неполный список:

```
const
  clBlack = Color.Black;
  clBlue = Color.Blue;
  clBrown = Color.Brown;
  clCyan = Color.Cyan;
  clDarkBlue = Color.DarkBlue;
  clDarkCyan = Color.DarkCyan;
  clDarkGray = Color.DarkGray;
  clDarkGreen = Color.DarkGreen;
  clDarkMagenta = Color.DarkMagenta;
  clDarkOrange = Color.DarkOrange;
  clDarkRed = Color.DarkRed;
  clGreen = Color.Green;
  clLightBlue = Color.LightBlue;
  clLightCyan = Color.LightCyan;
  clLightGray = Color.LightGray;
  clLightGreen = Color.LightGreen;
  clMagenta = Color.Magenta;
  clMaroon = Color.Maroon;
  clNavy = Color.Navy;
  clOlive = Color.Olive;
  clOrange = Color.Orange;
  clRed = Color.Red;
  clSilver = Color.Silver;
  clWhite = Color.White;
  clYellow = Color.Yellow;
```

Имеется также функция без параметров `clRandom` целого типа, возвращающая случайный непрозрачный цвет.

Перья

Рисование графических примитивов осуществляется *пером* и *кистью*. 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.

Перейдем к рассмотрению свойств пера. К ним относятся *координаты*, *цвет*, *ширина* и *стиль*.

Текущие *координаты* пера возвращаются функциями `PenX` и `PenY`. Для изменения текущих координат пера предназначены следующие процедуры:

`MoveTo(x,y)` – перемещение пера к точке с координатами `(x,y)`;

`LineTo(x,y)` – рисование отрезка от текущего положения пера до точки `(x,y)`; координаты пера при этом становятся равными `(x,y)`.

Имеется два способа работы со свойствами пера. Первый способ – процедурный. Рассмотрим его подробнее.

Цвет пера устанавливается с помощью процедуры `SetPenColor(c)`, а его текущее значение возвращается функцией `PenColor`. Процедура `SetPenWidth(w)` устанавливает *ширину* пера, равную `w` пикселам, а функция `PenWidth` целого типа возвращает текущую ширину пера. По умолчанию перо имеет черный цвет и ширину, равную 1. *Стиль* пера устанавливается процедурой `SetPenStyle(ps)`, его текущее значение возвращается функцией `PenStyle`. Имеются следующие стили пера, определяемые именованными константами целого типа:

`psSolid`	———————————
`psDash`	— — — — — — — — — — — — — — —
`psDot`	.....
`psDashDot`	_ . _ . _ . _ . _ . _ . _ . _ . _
`psDashDotDot`	_ . . _ . . _ . . _ . . _ . . _ . . _
`psClear`	(линия не рисуется)

Второй способ работы со свойствами пера – с помощью объекта `Pen`, имеющего свойства `Color`, `Width` и `Style`. Так, чтобы установить пунктирное перо красного цвета, достаточно написать

```
Pen.Color := clRed;
Pen.Width := 3;
```

Такой способ является более современным и предпочтительным для обучения, однако, нуждается в минимальном представлении об объектах, их свойствах и методах на уровне понимания точечной нотации. К счастью, точечная нотация легко усваивается даже младшими школьниками.

Кисти

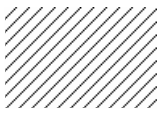
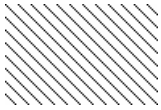
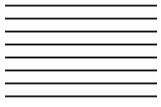
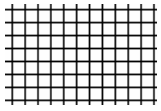
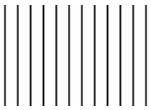
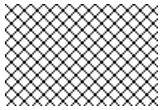
Начнем с рассмотрения свойств кисти. К ним относятся *цвет*, *стиль*, *штриховка* и второй цвет для градиентной кисти.

Для установки *цвета* кисти служит процедура `SetBrushColor(color)`. Функция `BrushColor` возвращает текущий цвет кисти. По умолчанию кисть имеет белый цвет.

Имеется четыре *стиля* кисти: сплошная, пустая, штриховая и градиентная, задаваемые константами `bsSolid`, `bsClear`, `bsHatch` и `bsGradient` соответственно. Для установки *стиля* кисти служит процедура `SetBrushStyle(bs)`. Функция `BrushStyle` возвращает текущий стиль кисти.

Пустая кисть используется для рисования замкнутых графических примитивов с пустой внутренностью. Аналогичную работу можно выполнить, используя графические примитивы с приставкой `Draw`.

Вид штриховки штриховой кисти задается более чем 50 именованными константами, основные из которых приведены ниже:

`bhBDiagonal` 
`bhFDiagonal` 
`bhHorizontal` 
`bhCross` 
`bhVertical` 
`bhDiagCross` 

Штриховка кисти устанавливается процедурой `SetBrushHatch(bh)`, а возвращается функцией `BrushHatch`. Штриховка работает только для кисти,

имеющей стиль bsHatch. Штриховка закрашивается текущим цветом кисти, а область вне штриховки – цветом, устанавливаемым процедурой SetHatchBrushBackgroundColor(c) и возвращаемым функцией HatchBrushBackgroundColor (по умолчанию – белый цвет).

Градиентная кисть в настоящей реализации GraphABC поддерживает только линейный градиент от левого верхнего до правого нижнего угла. Цвет левого верхнего угла определяется текущим цветом кисти, а цвет правого нижнего угла задается процедурой SetGradientBrushSecondColor(c), а возвращается функцией GradientBrushSecondColor.

Второй способ работы со свойствами кисти – с помощью объекта Brush, имеющего свойства Color, Style, Hatch, HatchBackgroundColor и GradientSecondColor. Так, задание штриховой кисти с горизонтальной штриховкой осуществляется следующим образом:

```
Brush.Style := bsHatch;
Brush.Hatch := bhHorizontal;
Brush.GradientSecondColor := clRed;
```

Шрифты, их свойства

Кроме пера и кисти с графическим окном связывается *шрифт*. Шрифт имеет следующие свойства: *цвет*, *размер*, *наименование* и *стиль*. Текущие свойства шрифта используются при выводе текста в графическое окно с помощью процедуры TextOut. Приведем список процедур и функций для работы со шрифтом графического окна:

SetFontColor(color)	– процедура, устанавливающая цвет шрифта;
FontColor	– функция, возвращающая текущий цвет шрифта;
SetFontSize(sz)	– процедура, устанавливающая размер шрифта в пунктах;
FontSize	– функция, возвращающая текущий размер шрифта в пунктах;
SetFontName(s)	– процедура, устанавливающая наименование

шрифта;

`FontName` – функция, возвращающая текущее наименование шрифта;

`SetFontStyle(fs)` – процедура, устанавливающая стиль шрифта;

`FontStyle` – функция, возвращающая текущий стиль шрифта;

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

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

По умолчанию установлен черный шрифт Arial размера 10 пунктов. Наиболее распространенные шрифты — это «Times New Roman», «Arial» и «Courier New». Наименование шрифта можно набирать без учета регистра.

Для установки *стиля* шрифта удобно пользоваться следующими именованными константами:

`fsNormal` – обычный;

`fsBold` – жирный;

`fsItalic` – наклонный;

`fsBoldItalic` – жирный наклонный;

`fsUnderline` – подчеркнутый;

`fsBoldUnderline` – жирный подчеркнутый;

`fsItalicUnderline` – наклонный подчеркнутый;

`fsBoldItalicUnderline` – жирный наклонный подчеркнутый.

Второй способ работы со свойствами шрифта – с помощью объекта `Font`, имеющего свойства `Color`, `Style`, `Size`, `Name`. Так, задание жирного наклонного синего шрифта Times в 12 пунктов осуществляется следующим образом:

```
Font.Name := 'Times New Roman';
Font.Size := 12;
Font.Style := fsBoldItalic;
Font.Color := clBlue;
```

Управление графическим окном

Следующие процедуры и функции в GraphABC позволяют управлять графическим окном:

`ClearWindow`	– процедура, закрашивающая окно белым цветом
`ClearWindow(c)`	– процедура, закрашивающая окно цветом `c`
`WindowWidth`	– функция, возвращающая ширину графического окна;
`WindowHeight`	– функция, возвращающая высоту графического окна;
`WindowLeft`	– функция, возвращающая отступ графического окна от левого края экрана;
`WindowTop`	– функция, возвращающая отступ графического окна от верхнего края экрана;
`WindowCaption`	– функция, возвращающая текст заголовка графического окна;
`SetWindowWidth(h)`	– процедура, устанавливающая ширину графического окна;
`SetWindowHeight(h)`	– процедура, устанавливающая высоту графического окна;
`SetWindowLeft(l)`	– процедура, устанавливающая отступ графического окна от левого края экрана (то есть экранную координату *x*);
`SetWindowTop(t)`	– процедура, устанавливающая отступ графического окна от верхнего края экрана (то есть экранную координату *y*);
`SetWindowSize(w,h)`	– процедура, устанавливающая ширину и высоту графического окна;
`SetWindowPos(l,t)`	– процедура, устанавливающая обе экранные коорди-

наты графического окна (то есть координаты его левого верхнего угла);

`SetWindowCaption(s)` – процедуру, устанавливающая новый заголовок окна;

`SaveWindow(fname)` – процедура, сохраняющая содержимое графического окна в файле с именем `fname`;

`LoadWindow(fname)` – процедура, загружающая в окно изображение из файла с именем `fname`; при этом размер окна подстраивается под размер загруженного изображения.

`CenterWindow` – процедура, центрирующая окно по центру экрана

Все размеры устанавливаются и возвращаются в пикселах.

Класс Picture

Для работы с изображениями в GraphABC имеется класс `Picture`, инкапсулирующий действия с рисунками. Ниже приводится его интерфейс:

```
type
  Picture = class
    constructor Create(w,h: integer);
```

Создает пустой рисунок размера w на h.

```
    constructor Create(fname: string);
```

Создает рисунок из файла с именем `fname`.

```
    constructor Create(r: Rectangle);
```

Создает рисунок из прямоугольника r графического окна

```
    procedure Load(fname: string);
```

Загружает рисунок из файла с именем `fname`.

```
    procedure Save(fname: string);
```

Сохраняет рисунок в файл с именем `fname`.

```
    procedure SetSize(w,h: integer);
```

Устанавливает размер рисунка w на h.

```
    property Width: integer;
```

```
property Height: integer;
```

Свойства, определяющие ширину и высоту рисунка

```
property Transparent: boolean;
```

Свойство, определяющее, имеет ли рисунок прозрачный цвет. При выводе рисунка методом Draw прозрачный цвет не рисуется.

```
property TransparentColor: Color;
```

Свойство, определяющее прозрачный цвет. По умолчанию – цвет левого нижнего пиксела.

```
function Intersect(p: Picture): boolean;
```

Пересекается ли данный рисунок с рисунком p. Для определения пересечения производится попиксельное сравнение. Прозрачным считается `TransparentColor`.

```
procedure Draw(x,y: integer);
```

Выводит рисунок в позиции `x,y`.

```
procedure Draw(x,y,w,h: integer);
```

Выводит рисунок в позицию `x,y`, масштабируя его к размеру `w,h`.

```
procedure Draw(x,y: integer; r: Rectangle);
```

Выводит часть рисунка из прямоугольника r в позицию `x,y`.

```
procedure Draw(x,y,w,h: integer; r: Rectangle);
```

Выводит часть рисунка из прямоугольника r в позицию `x,y`, масштабируя его к размеру `w,h`.

```
procedure CopyRect(dst: Rectangle; p: Picture;
  src: Rectangle);
```

Копирует часть рисунка `p` из прямоугольника `src` в прямоугольник `dst` текущего рисунка.

```
procedure FlipHorizontal;
```

Отражает рисунок относительно горизонтальной оси симметрии.

```
procedure FlipVertical;
```

Отражает рисунок относительно вертикальной оси симметрии.

Следующая группа методов является графическими примитивами рисования на рисунке. В качестве пера, кисти и шрифта используются те же, что и при выводе на графический экран.

```
    procedure PutPixel(x,y: integer; c: Color);
    function GetPixel(x,y: integer): Color;
    procedure Line(x1,y1,x2,y2: integer);
    procedure Line(x1,y1,x2,y2: integer; c: Color);
    procedure FillCircle(x,y,r: integer);
    procedure DrawCircle(x,y,r: integer);
    procedure FillEllipse(x1,y1,x2,y2: integer);
    procedure DrawEllipse(x1,y1,x2,y2: integer);
    procedure FillRectangle(x1,y1,x2,y2: integer);
    procedure FillRect(x1,y1,x2,y2: integer);
    procedure DrawRectangle(x1,y1,x2,y2: integer);
    procedure Circle(x,y,r: integer);
    procedure Ellipse(x1,y1,x2,y2: integer);
    procedure Rectangle(x1,y1,x2,y2: integer);
    procedure Arc(x,y,r,a1,a2: integer);
    procedure FillPie(x,y,r,a1,a2: integer);
    procedure DrawPie(x,y,r,a1,a2: integer);
    procedure Pie(x,y,r,a1,a2: integer);
    procedure TextOut(x,y: integer; s: string);
    procedure Clear;
  end;
```

Заметим, что GDI+ может работать с рисунками форматов .bmp, .jpg, .gif, .png, .wmf. В частности, используя класс Picture, можно легко преобразовывать форматы графических файлов:

```
    uses GraphABC;
    var p: Picture;
    begin
      p := Picture.Create('a.bmp');
      p.Save('a.jpg');
    end.
```

Анимация

Многие технические вопросы разработки приложений для работы с графикой рекомендуется рассматривать в рамках различных игровых проектов. При

этом можно эффективно совмещать обучение объектно-ориентированному программированию, основам работы с графикой и техническим особенностям языка. Рекомендуется рассматривать проекты параллельно со вводом простейших методов и подходов к работе с графикой, отмечая недостатки таких подходов, и указывая способы их решения.

Наиболее интересным представляется создание анимации. Рассмотрим пример простейшего приложения – «часы». Отличительной особенностью является то, что в данном случае эффект анимации достигается за счет полной перерисовки картинки.

```
Program Clock;
uses GraphABC;
const secPause = 20;
      wndRad = 250;
      radPercents = 80;

var     clkRad : integer;   //  Радиус циферблата в точках
minutes, seconds : integer; //  Текущее время (без часов)

procedure Init;
  //  Задает начальные значения
begin
  SetWindowSize(2*wndRad,2*wndRad);
  //  Радиус циферблата
  clkRad := Round(radPercents/100*wndRad);
end;

procedure DrawMarks;
  //  Рисует 12 отметок в виде кружочков на циферблате
  var i : integer;
begin
  SetBrushColor(clLightBlue);
  for i:=0 to 11 do
    Circle(wndRad + Round(clkRad*cos(2*Pi*i/12)),
           wndRad + Round(clkRad*sin(2*Pi*i/12)),10);
end;
procedure Tick;
begin
  //  Увеличивает время на 1 секунду
  inc(seconds);
  if seconds = 60 then
    begin
```

```
      seconds := 0;
      inc(minutes);
      if minutes = 60 then minutes := 0;
    end;
end;

procedure DrawArrows;
  //  Рисование стрелок
  var x,y : integer;
begin
  //  Определяем координаты конца секундной стрелки
  x := wndRad+Round(clkRad*sin(2*Pi*seconds/60));
  y := wndRad-Round(clkRad*cos(2*Pi*seconds/60));
  Line(wndRad, wndRad, x, y);

  //  Определяем координаты конца минутной стрелки
  x := wndRad+Round(0.8*clkRad*sin(2*Pi*minutes/60));
  y := wndRad-Round(0.8*clkRad*cos(2*Pi*minutes/60));
Line(wndRad, wndRad, x, y);
end;
 var i : integer;
begin
  Init;
  while True do
    begin
      Tick;
      ClearWindow;
      DrawMarks;
      DrawArrows;
      sleep(secPause);
    end;
end.
```

Данная программа наглядно демонстрирует проблему, часто возникающую при реализации анимации – мерцание. Объясняется это тем, что при выводе в графическое окно автоматически выполняется процесс перерисовки. При этом, если мы создаем какое-то изображение, последовательно рисуя различные элементы, то происходит наложение двух процессов – вывода в графическое окно и перерисовки. При рисовании статичного изображения это несущественно, а при выводе анимации возможно появление различных нежелательных эффектов.

Существует несколько способов борьбы с этим, обычно используются два подхода – управление процессом перерисовки, либо использование для построения изображения вместо графического окна некоторой области в памяти, а потом вывод построенного изображения в графическое окно.

Для управления процессом перерисовки в модуле GraphABC определены следующие процедуры:

```
procedure LockDrawing;
```

Блокирует вывод в графическое окно. Любые изменения графического окна сохраняются в памяти, но не отражаются на экране до тех пор, пока не будет снята блокировка либо окно не будет перерисовано принудительно.

```
procedure  UnlockDrawing;
```

Снимает блокировку и выполняет перерисовку графического окна.

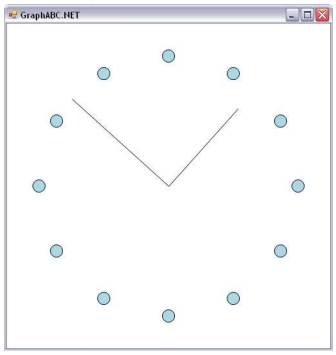
```
procedure Redraw;
```

Принудительная перерисовка графического окна, применяется для вывода графики в том случае, если окно заблокировано.

Рассмотрим пример использования этих процедур для рисования без мерцания. В программе «часы» тело программы изменяем следующим образом:

```
begin
  Init;
  LockDrawing;
  while True do
    begin
      Tick;
      ClearWindow;
      DrawMarks;
      DrawArrows;
      Redraw;
      sleep(secPause);
    end;
end.
```

В данном случае рисование выполняется без мерцания. Аналогичный эффект достигается комбинированием методов LockDrawing для блокировки и UnlockDrawing для снятия блокировки и перерисовки.



Следует обратить внимание, что блокировка графического окна не многоуровневая – количество вызовов процедуры LockDrawing значения не имеет, после однократного вызова метода UnlockDrawing блокировка будет снята независимо от того, сколько раз до этого был вызван метод LockDrawing.

В некоторых случаях удобно сначала построить изображение, а потом уже выводить его в графическое окно. Например, такая техника может применяться в тех случаях, когда процесс построения нового кадра изображения требует довольно много времени.

Для этого можно использовать объекты класса Picture, интерфейс которых предоставляет практические такие же возможности для работы с графикой, как и графическое окно.

```
Program Collage;
uses GraphABC;

procedure Collage(P1,P2,P : Picture);
  var     x, y : integer;
    maxX, maxY : integer;
  mnoj1, mnoj2 : double;
         R,G,B : integer;  //  составляющие цвета
         col : color;
begin
  //  Определяем размеры области пересечения рисунков
  if P1.Width<P2.Width then
    maxX := P1.Width-1
   else
    maxX := P2.Width-1;
  if P1.Height<P2.Height then
    maxY := P1.Height-1
   else
    maxY := P2.Height-1;
  //  Выводим сообщение
```

```
  SetFontSize(16);
  SetBrushStyle(bsClear);  //  Стиль заливки - невидимый
  TextOut((maxX - TextWidth('Обработка...')) div 2 ,
                      maxY - 40, 'Обработка...');
  SetPenColor(clRed);

  for x := 0 to maxX do
    begin
      mnoj1 := (maxX-x)/maxX;
      mnoj2 := x/maxX;
      for y := 0 to maxY do
        begin
          R := Round( mnoj1*GetRed(P1.GetPixel(x,y)) +
                      mnoj2*GetRed(P2.GetPixel(x,y)));
          G := Round( mnoj1*GetGreen(P1.GetPixel(x,y)) +
                      mnoj2*GetGreen(P2.GetPixel(x,y)));
          B := Round( mnoj1*GetBlue(P1.GetPixel(x,y)) +
                      mnoj2*GetBlue(P2.GetPixel(x,y)));
          col := RGB(R,G,B);
          P.PutPixel(x,y,col);
          //  Показываем прогресс обработки
          Line(x,maxY-10,x,maxY-2);
        end;
    end;
end;
  var Pict1, Pict2, Pict3 : Picture;
begin
  //  читаем изображения из файлов
  Pict1 := Picture.Create('2.jpg');
  Pict2 := Picture.Create('5.jpg');
  //  изменяем размеры графического окна
  SetWindowSize(Pict1.Width,Pict1.Height);

  Pict2.Draw(0,0);
  //  создаем картинку
  Pict3 := Picture.Create(Pict1.Width, Pict1.Height);
  //  Строим коллаж
  Collage(Pict1, Pict2, Pict3);
  Pict3.Draw(0,0);
end.
```

Архимедова спираль

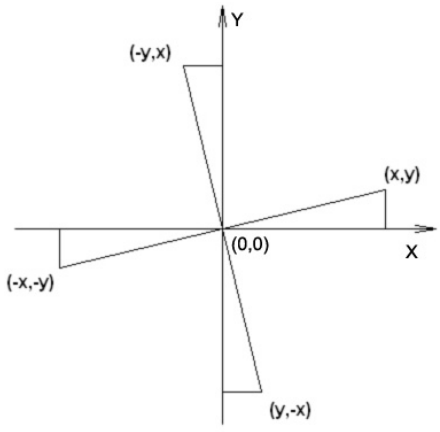
Рассмотрим пример построения спирали Архимеда – кривой, названной в честь великого древнегреческого математика Архимеда (3 в. до н.э.). Пусть пря-

мая линия выходит из начала координат, и вращается вокруг начала координат с постоянной скоростью. Из начала координат по этой прямой движется с равномерной скорость точка. Траектория, описываемая этой точкой, и будет представлять собой спираль Архимеда. Форма спирали зависит от скорости вращения и скорости движения точки. Сформулируем алгоритм рисования этой кривой. Пусть точка должна удалиться от начала координат на расстояние R (измеряемое в пикселях). Шаг точки по прямой равен одному пикселю, и после каждого шага будем рисовать текущее положение точки на экране. Предположим, что за каждый шаг угол поворота прямой вокруг начала координат увеличивается на некоторый угол и составляет *phi* радиан. Тогда формулы для вычисления текущих координат точки выглядят следующим образом:

$$X = R*cos(phi)$$

$$Y = R*sin(phi)$$

При этом можно немного усложнить задачу, и нарисовать на одном рисунке сразу четыре кривые, три из которых получаются поворотом точки вокруг начала координат на угол, кратный 90°. Получить координаты этих точек несложно, достаточно взглянуть на рисунок:



Если исходная точка имеет координаты *(x,y)*, то после поворота на 90° против часовой стрелки вокруг начала координат получаем точку с координатами *(-y, x)*. Аналогично, координаты двух других точек будут равны *(-x,-y)* и *(y,-x)*.

При составлении алгоритма необходимо учитывать еще одну особенность. Точка графического окна (пиксель) с координатами *(0,0)* расположена в левом верхнем углу графического окна, в то время как в графическом окне точка с эк-

ранными координатами *(0,0)* – это точка, расположенная в левом верхнем углу окна. Координаты центра графического окна будут равны:

$$Cx = WindowWidth \text{ div } 2$$

$$Cy = WindowHeight \text{ div } 2$$

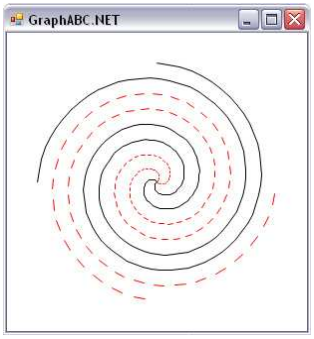
При этом ширина и высота окна должны быть нечетными значениями, иначе центр не будет совпадать с одним пикселем. Например, при ширине окна равной 101 пикселю центр будет иметь координату по X равную 50, при этом будет делить окно на две половины с координатами по X: [0..49] и [51..100]. Если ширина окна равна 100 пикселям, то диапазон координат точек по X будет [0..99], и центром могут считаться точки с координатами X равными 49 и 50.

В данном случае можно добиться при рисовании эффекта анимации, если рисовать спираль последовательно, с разумной величиной задержки между рисованием сегментов спирали. Это иллюстрация механизма анимации, при котором вся сцена не перерисовывается, а просто дополняется новыми элементами. При этом следует обратить внимание на то, что в данном случае эффект мерцания не возникает!

```
program spiral;
 uses GraphABC;
 const       WSize = 301;
        spiralSize = 200;
 var x_old, x, y_old, y, i, Center : integer;

begin
  SetWindowSize(WSize,WSize);
  //  Координата центра (x и y одинаковые в квадратном ок-
не)
  Center := WSize div 2;
  for i:=spiralsize-1 downto 0 do
    begin
      //  Запоминаем старые координаты
      //  Это координаты точек в декартовой системе
      x_old := x;
      y_old := y;
      //  Считаем координаты следующей точки
      x := Round((spiralSize-i)*cos(i/10));
      y := Round((spiralSize-i)*sin(i/10));
```

```
      Line(Center+x_old, Center+y_old,
               Center+x, Center+y, clRed);
      if odd(i) then
        Line(Center-x_old, Center-y_old,
               Center-x, Center-y);
      Line(Center-y_old, Center+x_old, Center-y,
               Center+x, clRed);
      if odd(i) then
        Line(Center + y_old, Center - x_old,
               Center + y, Center - x);
      //  Останавливаемся ненадолго
      Sleep(20);
    end;
  //  Сохраняем изображение в файл
  SaveWindow('Spiral.jpg');
end.
```

Спираль в процессе рисования показана на рисунке:



Фильтр – размытие по Гауссу

Рассмотрим еще один простой пример – преобразование изображения, которое в графических редакторах называется фильтром. Это простейший пример размытия, в котором цвет точки вычисляется как среднее арифметическое точек, окружающих данную (включая и саму эту точку). Такой алгоритм сложно реализовать с использованием только изображения в графическом окне по двум причинам. Прежде всего, цвет каждой точки вычисляется на основе цвета соседних точек, а цвета соседних – на основе этой. Поэтому простые формулы по вычислению среднего арифметического не подходят, нужно составлять более сложные и неочевидные формулы для получения цвета точки. Во-вторых, проблемы с граничными точками – для них формулы вычисления будут более сложными, необходимо проверять существование соседних точек рисунка. Алгоритм стано-

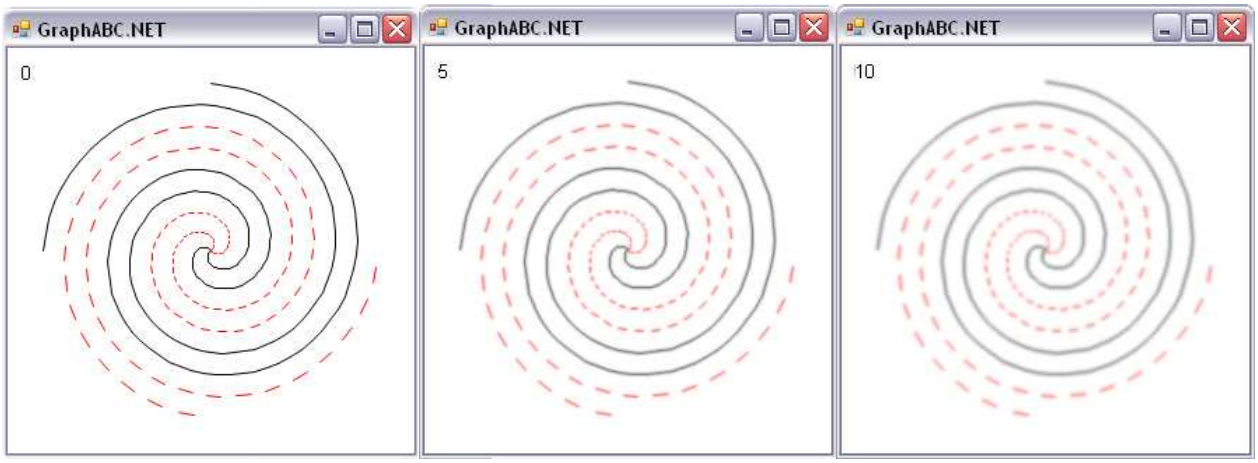
вится весьма сложным. Вместо этого можно воспользоваться построением нового («размытого») изображения в объекте класса *Picture*, а проблему с граничными точками решить просто – выводить в графическое окно не весь рисунок, а только его часть. Как правило, этого будет достаточно для хорошего эффекта размытия. Введем параметр *smoothing*, значение которого меняется от 0 до 1 и задает степень размытия. Значение 0 соответствует максимуму, при котором новое значение цвета в точке задается только цветами окружающих точек, значение 1 соответствует отсутствию размытия. Приведенная ниже программа иллюстрирует последовательное применение 10 операций размытия к изображению спирали Архимеда.

```
program Gauss;
 uses GraphABC;
  const border = 5;   // Не отображаемая граница рисунка
     smoothing = 1/9; // Коэффициент размытия
  var         P1, P2 : Picture;
procedure CopyPicture(P1,P2 : Picture);
  var x,y : integer;
begin
  for x := 0 to P1.Width-1 do
    for y := 0 to P2.Height-1 do
      P2.PutPixel(x,y,P1.GetPixel(x,y));
end;

procedure Smooth(P1,P2 : Picture);
   var x, y, i, j  : integer;
           r, g, b : integer;
               col : color;
begin

  for x := 1 to P1.Width-2 do
    for y := 1 to P2.Height-2 do
      begin
        //  Сбрасываем текущий цвет
        r := 0; g := 0; b := 0;
        //  Суммируем цвета окружающих точек
        for i := -1 to 1 do
          for j := -1 to 1 do
            if (i<>0)or(j<>0) then
              begin
                col := P1.GetPixel(x+i, y+j);
```

```
                r := r + GetRed(col);
                g := g + GetGreen(col);
                b := b + GetBlue(col);
              end;
        col := P1.GetPixel(x, y);
        r := Round(r*(1-smoothing)/8 +
                   GetRed(col)*smoothing);
        g := Round(g*(1-smoothing)/8 +
                   GetGreen(col)*smoothing);
        b := Round(b*(1-smoothing)/8 +
                   GetBlue(col)*smoothing);
        P2.PutPixel(x,y,RGB(r,g,b));
      end;
end;
 var k : integer;
     r : System.Drawing.Rectangle;
begin
  P1 := Picture.Create('Spiral.jpg');
  SetWindowSize(P1.Width-2*border, P1.Height-2*border);

  r := new System.Drawing.Rectangle(0,0,P1.Width-1,
                                    P1.Height-1);
  P1.Draw(-border,-border);
  P2 := Picture.Create(P1.Width, P1.Height);
  for k:=1 to 10 do
    begin
      LockDrawing;
      Smooth(p1,P2);
      ClearWindow;
      P2.Draw(-border,-border);
      //  CopyPicture(P2,P1);
      P1.CopyRect(r,P2,r);
      UnlockDrawing;
      Redraw;
      sleep(10);
    end;
end.
```

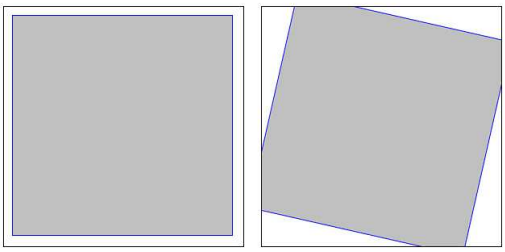
Следует сделать два замечания. В данной программе демонстрируются два способа копирования изображения из одного объекта класса *Picture* в другой – с помощью процедуры *CopyPicture*, и с помощью метода *CopyRect*. Предпочтительным является использование метода *CopyRect*, однако техника использования этого метода требует дополнительных пояснений. Второе замечание - неэффективность процедуры *Smooth*, в которой выполняется большое число сравне-

ний на совпадение с текущей точкой, оптимизацию этого процесса можно предлагать в качестве самостоятельного задания. Результат работы программы приведен на рисунке.



Алгоритм вращения

Рассмотрим пример приложения, в котором использование класса Picture может быть весьма интересным: вращение изображения вокруг центральной точки. Вращение прямоугольного изображения имеет одну особенность – при повороте на некоторый угол, как правило, часть точек исходного изображения оказывается за границами области рисования, а для части точек графического окна нет исходных точек для рисования.



Эту проблему можно решить, взяв в качестве исходного изображения объект класса *Picture*, размеры которого больше графического окна. Для того, чтобы не возникало «белых пятен», необходимо, чтобы исходное изображение имело размер, в $\sqrt{2}$ раз превышающий размер графического окна. Еще одна тонкость данного алгоритма заключается в следующем: теоретически необходимо взять каждую точку исходного изображения и, повернув ее на заданный угол, нарисо-

вать в результирующем изображении. Однако, из-за ошибок округления возможны ситуации, когда две точки исходного изображения проецируются в одну, а некоторые точки результирующего изображения оказываются не закрашенными. Чтобы избежать этого, необходимо выполнять обратное преобразование – брать каждый пиксель результирующего изображения, и для него находить соответствующий элемент исходного изображения по формулам:

$$X' = X*\cos(phi) + Y*\sin(phi)$$

$$Y' = -X*\sin(phi) + Y*\cos(phi)$$

где *phi* – угол поворота в радианах. При этом желательно для получения более качественного изображения использовать размытие, рассмотренное в предыдущем примере. Промежуточные изображения сохраняются в файлы с именами вида ‘1.jpg’, ‘2.jpg’ и т.д.

```
program Rotation;
  uses GraphABC;
  const border = 60;
     smoothing = 1/9;
  var P1, P2, P3 : Picture;
        x, y, x1, y1, shft, z : integer;
                          phi : double;
{Тут необходимо описать процедуру Smooth из предыдущего примера}
…
begin
  //  Читаем изображение спирали из файла
  P1 := Picture.Create('Spiral.jpg');
  P1.Draw(0,0);
  //  Задаем размеры графического окна
  SetWindowSize(P1.Width-2*border,P1.Height-2*border);
  shft := P1.Width div 2;
  //  Создаем еще два объекта Picture такого же размера
  P2 := Picture.Create(P1.Width,P1.Height);
  P3 := Picture.Create(P1.Width,P1.Height);
  //  Цикл - количество повторов
  for z := 0 to 35 do
    begin
      phi := z*Pi/18;
      //  Для каждой точки графического окна
      for x:=border to P1.Width-border-1 do
```

```
        for y := border to P1.Height-border-1 do
          begin
            //  Вычисляем координаты исходной точки
            x1 := Round((x-shft)*cos(phi)+
                                  (y-shft)*sin(phi));
            y1 := Round(-(x-shft)*sin(phi)+
                                  (y-shft)*cos(phi));
            //  Переносим исходную точку
            P2.PutPixel(x,y,P1.GetPixel(x1+shft,y1+shft));
          end;
      //  Размываем изображение
      Smooth(P2,P3);
      //  Рисуем результат поворота
      P3.Draw(-border,-border);
      //  Сохраняем результат в файл
      SaveWindow(IntToStr(z)+'.jpg');
  end;
end.
```

Построение спрайтов спирали Архимеда

Несмотря на постоянное увеличение вычислительной мощности компьютеров, реализация анимации с полной или частичной перерисовкой сцены в каждом кадре – трудоемкая задача. При этом иногда можно получить вполне реалистичную анимацию другим способом – с помощью спрайтов. Под термином «спрайт» обычно понимают последовательность заранее построенных и сохраненных двумерных изображений, которые последовательно выводятся на экран, создавая эффект анимации. При этом следует помнить, что для того, чтобы человек не замечал смены кадров, необходимо, чтобы они сменялись со скоростью примерно 25 кадров в секунду или более. Однако в компьютерных играх, к примеру, иногда используются и спрайты с меньшей частотой смены кадров в секунду.

В системе PascalABC.Net реализована библиотека ABCSprates, которая позволяет работать с объектами класса SpriteABC, реализующими основные операции со спрайтами. Рассмотрим пример создания спрайта, показывающего вращение спирали Архимеда (для этого в папке программы должны быть сохранены рисунки спирали, созданные в предыдущем примере).

```
program Sprites;
 uses GraphABC,ABCSprites,ABCObjects,Events;
 var  s: SpriteABC;
      i : integer;
begin
  s := new SpriteABC(0,0,'0.jpg');
  s.speed := 10;
  SetWindowSize(s.Width,s.Height);
  CenterWindow;
  for i:= 1 to 35 do
    s.Add(IntToStr(i)+'.jpg');
end.
```

В этом примере показан процесс создания спрайта на основе нескольких сохраненных изображений. Один из вариантов создания спрайта – использование конструктора *Create*, который создает спрайт на основе одного рисунка. После этого для объекта вызывается метод *Add*, который добавляет последовательность рисунков – с именами от ‘1.jpg’ до ’35.jpg’. Никаких вызовов дополнительных методов для начала анимации не требуется. Скоростью анимации можно управлять, меняя свойство *speed* для спрайта. Диапазон изменения значений этого свойства – от 1 до 10. Чем больше значение, тем выше скорость смены изображений (кадров) спрайта.

Управление состоянием спрайтов

Объекты класса `SpriteABC` могут иметь несколько состояний, которыми можно управлять. Каждому состоянию соответствует своя последовательность кадров, которая циклически воспроизводится. Например, пусть есть спрайт, в котором заданы 12 кадров. При этом спрайт имеет 3 состояния, каждому из которых соответствует своя последовательность кадров. Эта последовательность может быть, к примеру, такой:

Состояние	1	2	3
Номера кадров	с 1 по 3	с 4 по 10	с 11 по 12

В каждый момент времени спрайт может находиться в одном из состояний, которое определяется, какие кадры будут составлять анимацию. В данном при-

мере если спрайт находится в состоянии 2, то циклически будут показываться кадры с 4 по 10.

Для управления состоянием спрайтов сначала необходимо задать эти состояния. Каждое состояние имеет свое название (строковое значение), которое необходимо указывать при задании состояний. К примеру, задать три состояния, указанные выше, можно, последовательно вызвав метод *AddState*:

```
  s.AddState('start',3);
  s.AddState('middle',7);
  s.AddState('end',2);
```

где s – имя спрайта. При этом следует помнить, что при добавлении информации о состоянии методом *AddState* спрайт уже должен содержать достаточное число кадров! Определение состояний спрайта следует делать либо после того, как заданы все кадры, либо последовательно – добавить кадры, соответствующие

Текущему количеству состояний спрайта соответствует значение свойства *StateCount* (только для чтения). Для управления состоянием спрайта достаточно задать значение свойства *State* из диапазона [*1*..*StateCount*]. Пример работы с состояниями спрайтов выглядит следующим образом:

```
Program Sprites2;
 uses ABCSprites, GraphABC, Events;
 const side = 200; //  Сторона спрайта

 var    Spr : SpriteABC;

procedure Rotate90(P1,P2 : Picture);
  var x,y : integer;
begin
  if (P2.Width<P1.Height)or(P2.Height<P1.Width) then
    exit;

  for x := 0 to P1.Width-1 do
    for y := 0 to P1.Height-1 do
      P2.PutPixel(y,x,P1.GetPixel(x,y));
end;

procedure CreateSpritePictures;
  var  P1, P2 : Picture;
           St : string;
            i : integer;
```

```
begin
  SetFontSize(60);
  SetFontColor(clRed);
  //  Создаем две картинку размерами side*side
  P1 := Picture.Create(side,side);
  P2 := Picture.Create(side,side);

  St := '';
  for i := 1 to 3 do
    begin
      P1.Clear;
      St := St + '>';
      P1.TextOut((P1.Width - TextWidth(St)) div 2,
                 (P1.Height - TextHeight(St)) div 2, St);
      P1.Save(IntToStr(i+3)+'.bmp');
      Rotate90(P1,P2);
      P2.Save(IntToStr(i+6)+'.bmp');
      P1.FlipHorizontal;
      P1.Save(IntToStr(i+9)+'.bmp');
      P2.FlipVertical;
      P2.Save(IntToStr(i)+'.bmp');
    end;
end;

function CreateSprite : SpriteABC;
  var i : integer;
      S : SpriteABC;
begin
  S := SpriteABC.Create(0,0,'1.bmp');
  for i := 2 to 12 do
      S.Add(IntToStr(i)+'.bmp');

  //  Размечаем состояния
  for i:=1 to 4 do
    S.AddState(IntToStr(i),3);
  //  Проверка корректности
  S.CheckStates;
  Result := S;
end;

procedure KeyDown(key : integer);
   var newState :integer;
 begin
   case key of
     VK_Up    : NewState := 1;
```

```
      VK_Right : NewState := 2;
      VK_Down  : NewState := 3;
      VK_Left  : NewState := 4;
    end;

    if newState <> Spr.State then
      begin
        Spr.State := newState;
        Spr.Speed := 5;
      end
    else
      if Spr.Speed < 10 then
        Spr.Speed := Spr.SPeed + 1;
 end;

begin
  SetWindowSize(side,side);
  CenterWindow;
  CreateSpritePictures;
  Spr := CreateSprite;
  OnKeyDown := KeyDown;
end.
```

Данный пример демонстрирует основные принципы работы со спрайтами. Создается спрайт с четырьмя состояниями, каждое состояние содержит три кадра. Пользователь может управлять как скоростью смены кадров текущего состояния, так и выбирать текущее состояние. Все это выполняется с помощью клавиш с соответствующими стрелками на клавиатуре. Кадры спрайта представляют собой последовательности из трех рисунков, содержащих стрелочки, направленные в одном из четырех направлений:

Номер кадра	1	2	3	4	5	6	7	8	9	10	11	12
Состояние	1			2			3			4		
Кадры	^	^ ^	^ ^ ^	>	>>	>>>	v	v v	v v v	<	<<	<<<

Создается спрайт с помощью объекта класса Picture. Это удобнее, так как в этом примере, по сути, есть лишь 3 различных изображения (например, 1 – 3), остальные можно получить отражением и поворотом на 90°. Отражение для объекта *Picture* выполняется следующими методами:

```
procedure FlipHorizontal;
procedure FlipVertical;
```

Для выполнения вращения изображения на 90° требуется написать процедуру, которая выполняет вращение:

```
procedure Rotate90(P1,P2 : Picture);
```

Эта процедура создает в объекте P2 такое же изображение, как и в объекте P1, но повернутое на 90°. Процедура *CreateSpritePictures* создает набор из 12 изображений, и сохраняет их в файлы с именами ‘1.jpg’, ‘2.jpg’, … ,’12.jpg’. Из этих изображений будет строиться спрайт.

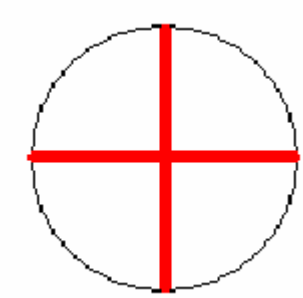
Функция *CreateSprite* создает спрайт из сохраненных изображений, возвращая ссылку на созданный спрайт. При этом стилистика работы соответствует ссылочной объектной модели *Delphi*, которая весьма сложна для понимания. Поэтому рекомендуется в процессе обучения периодически обращать внимание обучаемых на некоторые особенности работы, соответствующие этой модели представления объектов.

Управление состоянием спрайта выполняется в процедуре *KeyDown*, которая является обработчиком нажатия на клавишу. Эта процедура обрабатывает нажатия на клавиши со стрелками по следующему принципу: если нажатая клавиша совпадает с текущим изображением спрайта (направлением стрелок в кадрах), то увеличивается скорость анимации спрайта (если это возможно – если текущая скорость меньше максимальной). Иначе меняется состояние спрайта таким образом, чтобы стрелки в кадрах совпадали с нажатой клавишей, а скорость анимации устанавливается равной 5 – это половина от максимальной скорости. Таким образом, можно управлять как состоянием спрайта, так и скоростью его анимации.

Примеры графических приложений

Пример 1. Построение окружности

Построить окружность с двумя взаимно-перпендикулярными диаметрами, если заданы центр и радиус окружности.



Данный пример можно рассмотреть сразу же после изучения стандартных объектов модуля graphABC и свойств пера: функций SetPenColor и SetPenWidth. Далее приведен код процедуры для построения заданной фигуры.

```
uses graphABC;
procedure CircleWithCross(x,y,r:integer);
begin
Circle(x,y,r);
SetPenColor(clRed);
SetPenWidth(3);
Line(x,y-r,x,y+r);
Line(x-r,y,x+r,y);
end;
```

В качестве заданий можно рассмотреть следующие случаи:

1. Случайным образом изменять цвет диаметров с помощью функции `clRandom`.

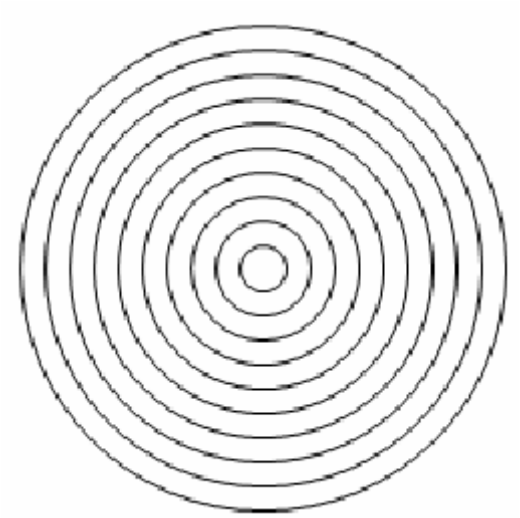
2. Изменять стиль пера самой окружности.

3. Построить прямоугольник, по заданным координатам верхнего левого верхнего угла (x_1; y_1) и размерам (a; b). Провести диагонали. Изменять стиль и цвет пера.

4. Построить прямоугольник, по заданным координатам верхнего левого угла и точки пересечения диагоналей. Изменять стиль и цвет пера.

Пример 2. Построение концентрических кругов

Для построения серии окружностей необходимо задать общий центр – точку (x,y), количество окружностей – N, радиус наименьшей окружности и шаг изменения по радиусу. Во избежание того, чтобы каждая новая окружность не «затирала» предыдущую, необходимо стиль кисти настроить как *SetBrushStyle(bsClear)* – прозрачный фон. Рассмотрим процедуру, реализующую поставленную задачу.



```
procedure InnerCircles(x,y,minr,step,n:integer);
{x, y - координаты центра
minR - радиус минимальной из окружностей
step - шаг изменения радиуса окружностей
N - количество окружностей}
var i:integer;
begin
SetBrushStyle(bsClear);
Circle(x,y,minr);
for i:=2 to n do
begin
minr:=minr+step;
Circle(x,y,minr);
end;
```

```
end.
```

Данную задачу можно модернизировать, изменяя в цикле толщину окружностей с помощью функции `SetPenWidth(w)` и цвет каждой новой окружности случайным образом с использованием функции `SetBrushColor(clRandom)`.

Пример 3. Построить график заданной функции (на примере $y=x^2$)

Рассмотрим одну из типичных задач – построение графика произвольной функции в графическом окне на некотором отрезке $[x_{left},x_{rigth}]$ с автоматическим масштабированием. Для построения необходимо знать масштаб, который вычисляется на основе минимального и максимального значения функции на данном отрезке. Для этого необходимо сначала найти экстремумы функции, а потом уже приступать к построению графика. График строится по точкам $[x_i,y_i]$, $1\le i\le N$, N – количество точек, поэтому экстремумы достаточно определить для $\{y_i\}$. При этом необходимо сначала построить массив точек графика с вещественными координатами, потом выполнить его масштабирование и округление так, чтобы получить массив точек экранных координат, и по ним построить график.

```
uses GraphABC, Events;
type FUN = function (x: real): real;
function f(x: real): real;
begin
  Result:=x*x;
end;
```

Процедура drawGraph, выполняющая построение графика заданной функции, координаты xl,xl0,wl,yl,yl0,hll –логические координаты, xs0,ws,ys0,hs – экранные координаты.

```
procedure drawGraph(x1,x2: real; f: FUN);
 var
```

```
  xl,xl0,wl,yl,yl0,hl: real;
  xs0,ws,ys0,hs: integer;
```

Функции LtoSx и LtoSy переводят логические координаты в экранные, сначала масштабируя, а затем вычисляя их.

```
function LtoSx(xl: real): integer;
 begin
   Result:=round(ws/wl*(xl-xl0)+xs0);
 end;
 function LtoSy(yl: real): integer;
 begin
   Result:=round(hs/hl*(yl-yl0)+ys0);
 end;
 function StoLx(xs: integer): real;
 begin
   Result:=wl/ws*(xs-xs0)+xl0;
 end;
 var xi: integer;
     yi: array [0..N] of real;
     min,max,y: real;
begin // drawGraph
  xs0:=0;
  ys0:=WindowHeight-1;
  ws:=WindowWidth;
  hs:=WindowHeight-1;
  xl0:=x1;
  wl:=x2-x1;
  min:=1e300;
  max:=-1e300;
  for xi:=0 to ws do
  begin
```

```
    yi[xi]:=f(StoLx(xi+xs0));
    if yi[xi]<min then min:=yi[xi];
    if yi[xi]>max then max:=yi[xi];
  end;
  yl0:=min;
  hl:=-(max-min);
  MoveTo(0,LtoSy(0));  LineTo(ws,LtoSy(0));
  MoveTo(LtoSx(0),0);  LineTo(LtoSx(0),hs);
  SetPenColor(clBlue);
  MoveTo(xs0,LtoSy(yi[0]));
  for xi:=xs0+1 to xs0+ws do
    LineTo(xi,LtoSy(yi[xi-xs0]));
end;
```

Процедура Resize необходима для построения графика функции, так как при получении новой точки происходит изменение размеров графического изображения, которое требует изменение размеров графического окна:

```
procedure Resize;
begin
  ClearWindow;
  drawGraph(0,60,f);
  Redraw; //перерисовка окна
end;
```

Пример 4. Случайное движение геометрических тел

При изменении размеров окна, траектория соответственно масштабируется. Использован модуль *ABCObjects*. Случайно сгенерированные объекты функцией **CreateRandomABC** перемещаются по экрану процедурой **Move.**

```
uses ABCObjects;
function CreateRandomABC: ObjectABC;
```

```
begin
  case Random(3) of
0: Result:=CircleABC.Create(Random(WindowWidth-30)+10,
   Random(WindowHeight-30)+10,Random(20)+10,clRandom);
1: Result:=RectangleABC.Create(Random(WindowWidth-30)+10,
           Random(WindowHeight-30)+10, Random(20)+10,
           Random(20)+10, clRandom);
2: Result:=StarABC.Create(Random(WindowWidth-30)+10,
            Random(WindowHeight-30)+10, Random(20)+10,
            Random(10)+5, Random(4)+4, clRandom);
  end;
end;
procedure Move(o: ObjectABC);
begin
  o.MoveOn(o.dx,o.dy);
  if (o.Left<0) or (o.Left+o.Width>WindowWidth) then
    o.dx:=-o.dx;
  if (o.Top<0) or (o.Top+o.Height>WindowHeight) then
    o.dy:=-o.dy;
end;
```

При выводе объектов на экран используется блокировка графического окна `LockDrawingObjects` с возможностью перерисовки всего графического окна с объектами `RedrawObjects`.

```
const n=200;
var     m: ObjectABC;
        i: integer;
begin
  SetWindowCaption('Движущиеся объекты');
  LockDrawingObjects;
  for i:=1 to n do
  begin
```

```
    m:=CreateRandomABC;
    repeat
      m.dx:=Random(7)-3;
      m.dy:=Random(7)-3;
    until (m.dx<>0) and (m.dy<>0);
  end;
  while True do
  begin
    for i:=1 to ObjectsCount do
      Move(Objects[i]);
    RedrawObjects;
  end;
end.
```

Пример 5. Игра «Составь слово»

Рассмотрим обработку событий клавиатуры и мыши на примере игры «Составь слово»: случайным образом генерируется таблица букв русского алфавита. далее в табло с помощью мыши составляется слово, если оно является правильным русским словом, то игрок получает очки. Набор самых употребительных русских слов содержится в текстовом файле.

Рассмотрим основные идеи решения данной задачи. Задаются два объекта типа «доска» (b,b1) - это само графическое окно табло, на которое будут заноситься выбранные буквы из случайным образом сгенерированной таблицы на основе массива символов (freqcharstr). На основной доске находятся три кнопки типа ButtonABC: «сказать» (btword), «заново» (btnew), «подсказка»(btplease), и объект типа RectangleABC (r), на котором ведется подсчет очков и ходов. Для каждой из указанных кнопок написан, соответственно, обработчик нажатой клавиши мыши **BtWordClick, BtNewClick, BtPleaseClick.** Процедура MyMouseDown – обработчик мыши, проверяет существует ли собранное по буквам слово

с помощью функции WordExists. Используются такие методы объектов как Visible, Text, Color.

```
    uses GraphABC,ABCObjects,ABCButtons,Events;
    const freqcharstr = 'аааааааааааа-
ааббббвввввгггггггдддддддддеееееееееееежжжжжжжжжзззззз
зззззззиииииииииииииииийййккккккккккккклллллллллллллммммммм
мммммммнннннннннннноооооооооооооооопппппппппппппрррррррррр
ррррррррсссссссссссссссссссссттттттттттттттттттуууууууу
ууууууфффффффффхххххххцццццццццццчччччччччччшшшшщщщщщээээъъю
ююьяяяяяяя';
    const maxwordlen=12;
      scorehits : array [1..maxwordlen] of integer =
(0,1,2,4,7,11,16,22,29,37,46,56);
    type
      MySquareABC=class(SquareABC) end;
    var
      b,b1: ObjectBoardABC;
      x,y: integer;
      cur,moves,score: integer;
      r: RectangleABC;
      btnew,btword,btplease: ButtonABC;
    function WordExists(s: string): boolean;
    //проверяем, есть ли собранное слово во входном тек-
стовом файле
    var
      f: text;
      str: string;
    begin
      s:=LowerCase(s);
```

```
  Result:=False;
  assign(f,'words.txt');
  reset(f);
  while not eof(f) do
  begin
    readln(f,str);
    if s=str then
    begin
      Result:=True; break;
    end;
  end;
  close(f); end;

procedure MyMouseDown(x,y,mb: integer);
var
  ob,ob1: ObjectABC;
  xx,yy,i: integer;
  s: string;
begin
  if mb=1 then
  begin
    if cur>b1.DimX then
      exit;
    ob:=ObjectUnderPoint(x,y);
//считываем объект под точкой с координатами (x,y)
    if ob is MySquareABC then
    begin
      ob1:=b1.GetObject(cur,1);
//помещаем выбранный объект в табло для сборки слова
      ob1.Visible:=True;
```

```
        ob1.Text:=ob.Text;
        Inc(cur);
        ob.Visible:=False;
    //в таблице букв перемещенный объект становится невидимым
        s:='';
        for i:=1 to cur-1 do
          s:=s+b1.GetObject(i,1).Text;
    //собираем слово
        if WordExists(s) then
          b1.Color:=clYellow
    //если данное слово существует, табло закрашивается желтым цветом
        else b1.Color:=clSkyBlue
      end;
     end
    else
     begin
      for xx:=1 to cur-1 do
        b1.GetObject(xx,1).Visible:=False;
      for xx:=1 to b.DimX do
      for yy:=1 to b.DimY do
        b.GetObject(xx,yy).Visible:=True;
      cur:=1;
      b1.Color:=clSkyBlue
     end;  end;

    procedure BtNewClick;
    //помещение новой буквы в табло
    var xx,yy: integer;
```

```
begin
  score:=0;
  moves:=0;
  r.Text:='Ходов: '+IntToStr(moves)+'   Очков: '
                                  +IntToStr(score);
  MyMouseDown(1,1,2);
  for xx:=1 to b.DimX do
  for yy:=1 to b.DimY do
    b.GetObject(xx,yy).Text:=
              UpCase(freqcharstr[Random(255)+1]);
end;

procedure BtWordClick;
//обработка слова, подсчет очков, ходов  и обновление
таблицы букв
var xx,yy: integer;
begin
  if b1.Color<>clYellow then
    exit;
  Inc(score,scorehits[cur-1]);
  Inc(moves);
  for xx:=1 to cur-1 do
    b1.GetObject(xx,1).Visible:=False;
  for xx:=1 to b.DimX do
  for yy:=1 to b.DimY do
    if not b.GetObject(xx,yy).Visible then
    begin
      b.GetObject(xx,yy).Visible:=True;
      b.GetObject(xx,yy).Text :=
               UpCase( freqcharstr[Random(255)+1]);
```

```
    end;
  cur:=1;
  b1.Color:=clSkyBlue;
  r.Text:='Ходов: '+IntToStr(moves)+
                    '   Очков: '+IntToStr(score);
end;

procedure BtPleaseClick;
//процедура подсказки: формирует слово из букв текущей таблицы и находящееся в словаре программы
var
  f: text;
  str,maxstr: string;
  arr,work: array ['а'..'я'] of integer;
  c: char;
  xx,yy,maxlen: integer;

  function CanConstructWord(s: string): boolean;
  var
    c: char;
    i: integer;
  begin
    work:=arr;
    Result:=True;
    for i:=1 to Length(s) do
    begin
      Dec(work[s[i]]);
      if work[s[i]]<0 then
      begin
        Result:=False;
```

```
        break;
      end;
    end;
  end;

begin
  maxlen:=0;
  maxstr:='';
  for c:='à' to 'ÿ' do
    arr[c]:=0;
  for xx:=1 to b.DimX do
  for yy:=1 to b.DimY do
    Inc(arr[LowCase(b.GetObject(xx,yy).Text[1])]);
  assign(f,'words.txt');
  reset(f);
  while not eof(f) do
  begin
    readln(f,str);
    if CanConstructWord(str) and (Length(str)>maxlen)
            and (Length(str)<=maxwordlen) then
    begin
      maxlen:=Length(str);
      maxstr:=str;
    end;
 end;
  close(f);
  writeln(maxstr);
end;

procedure MyKeyPress(c: char);
```

```
begin
  // Esc не закрывает окно !
end;
```

При задании необходимых параметров графического окна и инициализации объектов, необходимо учесть вызов OnMouseDown при нажатии мыши и OnKeyPress при нажатии символьной клавиши.

```
begin
  cls;
  SetWindowSize(640,480);
  SetWindowCaption('Собери слово!');
  SetBrushColor(clMoneyGreen);
  FillRect(0,0,WindowWidth,WindowHeight);
  cur:=1;
    btword:=ButtonABC.Create(70,410,180,30,
                          'Слово собрано',clLtGray);
  btnew:=ButtonABC.Create(280,410,100,30,'Начать
                                    заново', clLtGray);
  btplease:=ButtonABC.Create(410,410,160,30,'Подсказка',
                                              clLtGray);
  btnew.OnClick:=BtNewClick;
  btword.OnClick:=BtWordClick;
  btplease.OnClick:=BtPleaseClick;

  r:=RectangleABC.Create(70,350,500,30,clSkyBlue);
  r.Text:='Ходов: 0   Очков: 0';

  b1:=ObjectBoardABC.Create(20,40,maxwordlen,
                                       1,50,clSkyBlue);
  b:=ObjectBoardABC.Create(220,120,4,4,50,clMoneyGreen);
  b.BorderColor:=clGreen;
  b.Bordered:=False;
```

```
  for x:=1 to b1.DimX do
  begin
    b1.SetObject(x,1,SquareABC.Create(0,0,
                            b1.CellSize-6,clWhite));
    b1.GetObject(x,1).Visible:=False;
//начальное состояние табло для слова
  end;
  for x:=1 to b.DimX do
  for y:=1 to b.DimY do
  begin
    b.SetObject(x,y,MySquareABC.Create(0,0,
                            b.CellSize-6,clWhite));
    b.GetObject(x,y).Text:=
                UpCase(freqcharstr[Random(255)+1]);
  end;
//генерация таблицы букв
  OnMouseDown:=MyMouseDown;
  OnKeyPress:=MyKeyPress;
end.
```