Александр Чиртик

ТРЮКИ И ЭФФЕКТЫ

# ПРОГРАММИРОВАНИЕ В DELPHI

ПИТЕР

Александр Чиртик

# Программирование в Delphi. Трюки и эффекты

«Питер»
2010

**Чиртик А. А.**

Программирование в Delphi. Трюки и эффекты / А. А. Чиртик — «Питер», 2010

Как и все издания данной серии, эта книга адресована тем, кто хочет научиться делать с помощью уже знакомых программных пакетов новые интересные вещи. Издание будет полезно и новичкам, и опытным программистам. Автор описывает удивительные возможности, скрытые в языке, и на примерах учит читателя программистским фокусам – от «мышек-невидимок» и «непослушных окон» до воспроизведения MP3 и управления офисными программами Word и Excel из приложений Delphi. Купив эту книгу, вы пройдете непростой путь к вершинам программистского мастерства весело и интересно.

# Содержание

# Александр Анатольевич Чиртик Программирование в Delphi. Трюки и эффекты

## Введение

В настоящее время количество книг, посвященных различным языкам программирования, настолько велико, что иногда просто не знаешь, какую выбрать. Цель этой книги – не просто тривиальное изложение материала о Delphi. Она поможет вам получить опыт в решении многих задач. В итоге вы получите необходимый базис знаний, который даст возможность легко и быстро усваивать что-то новое. Здесь вы найдете ответы на вопросы, которые возникают у большинства людей при разработке своих собственных приложений. Вам больше не придется задумываться над тем, как решать мелкие задачи, которые составляют значительную часть повседневной работы большинства программистов. У вас появится возможность тратить больше времени именно на основную цель, поставленную перед вами, а не на второстепенную.

Данная книга рассчитана на читателей, которые уже имеют некий опыт в программировании, причем достаточный, чтобы не излагать тривиальные вещи заново. Однако сразу отмечу, пусть даже вы делаете свои первые шаги на пути к написанию приложений на высоком уровне, – книга окажет вам неоценимую помощь. Она построена так, чтобы вы смогли с высокой степенью эффективности узнавать новый материал. В конце книги есть приложения в удобном для восприятия виде. В них вы найдете информацию, которая часто используется при написании программ.

Зачастую люди выбирают Delphi за его простоту. Она подкупает начинающих, которые хотят почти сразу писать программы, а не разбираться в особенностях синтаксиса языка. Простота в совокупности с мощью дают вам целый набор инструментов для воплощения задуманного. Однако запомните: чтобы научиться хорошо программировать, недостаточно иметь огромный объем теоретических знаний, хотя и он немаловажен. Следует научиться думать в концепции выбранного вами языка, и тогда вас ждет успех. Ведь не понимая, зачем все это нужно, вы не сможете эффективно воспользоваться ресурсами языка для наиболее удачного решения поставленных задач.

В этой книге описано множество примеров. Есть как относительно простые, так и довольно сложные, но пусть последнее вас не пугает: к тому моменту, когда вы начнете их рассматривать, они не покажутся вам особенно трудными.

# От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты gromakovski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все примеры, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

# Глава 1
# Окна

- **Привлечение внимания к приложению**
- **Окно приложения**
- **Полупрозрачные окна**
- **Окна и кнопки нестандартной формы**
- **Немного о перемещении окон**
- **Масштабирование окон**
- **Добавление команды в системное меню окна**
- **Отображение формы поверх других окон**

Почему было решено начать книгу именно с необычных приемов использования оконного интерфейса? Причиной стало то, что при работе с операционной системой Windows мы видим окна постоянно и повсюду (отсюда, собственно, и название этой операционной системы). Речь идет не только об окнах приложений, сообщений, свойств: понятие о таких окнах есть у любого начинающего пользователя Windows.

В своих собственных окнах рисуются и элементы управления (текстовые поля, панели инструментов, таблицы, полосы прокрутки, раскрывающиеся списки и т. д.). Взгляните на интерфейс, например, Microsoft Word. Здесь вы увидите, что даже содержимое документа находится в своем собственном окне с полосами прокрутки (правда, это необязательно элемент управления). Окна элементов управления отличаются от «самостоятельных» окон (упрощенно) отсутствием стиля, позволяющего им иметь заголовок, а также тем, что они являются дочерними по отношению к другим окнам. Понимание этого является важным, так как на нем основана часть примеров данной главы.

Рассматриваемые примеры частично используют средства, предусмотренные в Borland Delphi, а частично – возможности «чистого» API (см. гл. 2). Практически все API-функции работы с окнами требуют задания параметра с типом значения HWND – дескриптора окна. Это уникальное значение, идентифицирующее каждое существующее в текущем сеансе Windows окно. В Delphi дескриптор окна формы и элемента управления хранится в параметре Handle соответствующего объекта.

Нужно также уточнить, что в этой главе термины «окно» и «форма» употребляются как синонимы, когда речь идет о форме. Когда же речь идет об элементах управления, то так и говорится: «окно элемента управления».

# Привлечение внимания к приложению

Начнем с простых примеров, позволяющих привлечь внимание пользователя к определенному окну приложения. Это может пригодиться в различных ситуациях: начиная от необходимости уведомления пользователя об ошибке программы и заканчивая простой сигнализацией ему, какое окно в данный момент времени ожидает пользовательского ввода.

## Инверсия заголовка окна

Вероятно, вы не раз могли наблюдать, как некоторые приложения после выполнения длительной операции или при возникновении ошибки как бы подмигивают. При этом меняется цвет кнопки приложения на Панели задач, а состояние открытого окна меняется с активного на неактивное. Такой эффект легко достижим при использовании API-функции FlashWindow или ее усовершенствованного, но более сложного варианта – функции FlashWindowEx.

> **Примечание**
>
> Здесь сказано, что функции изменяют цвет кнопки приложения на Панели задач. Однако этого не происходит при выполнении приведенных ниже примеров. Почему так получается и как это изменить, рассказано в следующем разделе этой главы (стр. 15).

Первая из этих функций позволяет один раз изменить состояние заголовка окна и кнопки на Панели задач (листинг 1.1).

### Листинг 1.1. Простая инверсия заголовка окна

```
procedure TForm1.cmbFlashOnceClick(Sender: TObject);
begin
FlashWindow(Handle, True);
end;
```

Как видите, функция принимает дескриптор нужного окна и параметр (тип BOOL) инверсии. Если значение флага равно T rue, то состояние заголовка окна изменяется на противоположное (из активного становится неактивным и наоборот). Если значение флага равно False, то состояние заголовка окна дважды меняет свое состояние, то есть восстанавливает свое первоначальное значение (активно или неактивно).

Более сложная функция FlashWindowEx в качестве дополнительного параметра (кроме дескриптора окна) принимает структуру FLASHWINFO, заполняя поля которой можно настроить параметры мигания кнопки приложения и/или заголовка окна.

В табл. 1.1 приведено описание полей структуры FLASHWINFO.

### Таблица 1.1. Поля структуры FLASHWINFO

| Поле | Тип | Назначение |
|---|---|---|
| cbSize | UINT | Размер структуры FLASHWINFO (для отслеживания версий) |
| hwnd | HWND | Дескриптор окна |
| dwFlags | DWORD | Набор флагов, задающий режим использования функции FlashWindowEx. Значения этих флагов и их описания приведены после таблицы |
| uCount | UINT | Количество инверсий заголовка окна и/или кнопки на Панели задач |
| dwTimeout | DWORD | Время между изменениями состояния заголовка окна и/или кнопки на Панели задач. Если значение равно нулю, используется системное значение таймаута |

Значение параметра dwFlags формируется из приведенных ниже флагов с использованием операции побитового ИЛИ:

• FLASHW_CAPTION – инвертирует состояние заголовка окна;

• FLASHW_TRAY – заставляет мигать кнопку на Панели задач;

• FLASHW_ALL – сочетание FLASHW_CAPTION и FLASHW_TRAY;

• FLASHW_TIMER – периодически измененяет состояния заголовка окна и/или кнопки на Панели задач до того момента, пока функция FlashWindowEx не будет вызвана с флагом FLASHW_STOP;

• FLASHW_TIMERNOFG – периодически измененяет состояния заголовка окна и/или кнопки на Панели задач до тех пор, пока окно не станет активным;

• FLASHW_STOP – восстанавливает исходное состояние окна и кнопки на Панели задач.

Далее приведены два примера использования функции FlashWindowEx.

В первом примере состояние заголовка окна и кнопки на Панели задач изменяется десять раз в течение двух секунд (листинг 1.2).

### Листинг 1.2. Десятикратная инверсия заголовка окна

```
procedure TForm1.cmbInverse10TimesClick(Sender: TObject);
var
fl: FLASHWINFO;
begin
fl.cbSize:= SizeOf(fl);
fl.hwnd:= Handle;
fl.dwFlags:= FLASHW_CAPTION or FLASHW_TRAY; //аналогично FLASHW_ALL
fl.uCount:= 10;
fl.dwTimeout:= 200;
FlashWindowEx(fl);
end;
```

Второй пример демонстрирует использование флагов FLASHW_TIMER и FLASHW_STOP для инверсии заголовка окна в течение заданного промежутка времени (листинг 1.3).

### Листинг 1.3. Инверсия заголовка окна в течение определенного промежутка времени

```
//Запуск процесса периодической инверсии заголовка
procedure TForm1.cmbFlashFor4SecClick(Sender: TObject);
var
fl: FLASHWINFO;
begin
fl.cbSize:= SizeOf(fl);
fl.hwnd:= Handle;
fl.dwTimeout:= 200;
fl.dwFlags:= FLASHW_ALL or FLASHW_TIMER;
fl.uCount:= 0;
FlashWindowEx(fl);
Timer1.Enabled:= True;
end;
//Остановка инверсии и заголовка
procedure TForm1.Timer1Timer(Sender: TObject);
var
fl: FLASHWINFO;
begin
fl.cbSize:= SizeOf(fl);
fl.hwnd:= Handle;
fl.dwFlags:= FLASHW_STOP;
FlashWindowEx(fl);
Timer1.Enabled:= False;
end;
```

В данном примере используется таймер, срабатывающий каждые четыре секунды. Таймер первоначально неактивен. Конечно, можно было бы не использовать его, а просто посчитать количество инверсий, совершаемых в течение требуемого интервала времени (в данном случае четырех секунд) и задать его в поле uCount. Но приведенный пример предназначен именно для демонстрации использования флагов FLASHW_TIMER и FLASHW_STOP.

## Активизация окна

Теперь рассмотрим другой, гораздо более гибкий способ привлечения внимания к окну приложения. Он базируется на использовании API-функции SetForegroundWindow. Данная функция принимает один единственный параметр – дескриптор окна. Если выполняется ряд условий, то окно в заданным дескриптором будет выведено на передний план, и пользовательский ввод будет направлен в это окно. Функция возвращает нулевое значение, если не удалось сделать окно активным.

В приведенном ниже примере окно активизируется при каждом срабатывании таймера (листинг 1.4).

## Листинг 1.4. Активизация окна

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
SetForegroundWindow(Handle);
end;
```

В операционных системах старше Windows 95 и Windows NT 4.0 введен ряд ограничений на действие функции SetForegroundWindow. Приведенный выше пример как раз и является одним из случаев недружественного использования активизации окна – но это всего лишь пример.

Чтобы активизировать окно, процесс не должен быть фоновым либо должен иметь право устанавливать активное окно, назначенное ему другим процессом с таким правом, и т. д. Все возможные нюансы в пределах одного трюка рассматривать не имеет смысла. Стоит отметить, что в случае, когда окно не может быть активизировано, автоматически вызывается функция FlashWindow для окна приложения (эта функция заставляет мигать кнопку приложения на Панели задач). Поэтому даже при возникновении ошибки при вызове функции SetForegroundWindow приложение, нуждающееся во внимании, не останется незамеченным.

# Окно приложения

Обратите внимание на то, что название приложения, указанное на кнопке, расположенной на Панели задач, совпадает в названием проекта (можно установить на вкладке Application окна Project options, вызываемого командой меню Project ► Options). Но это название не совпадает с заголовком главной формы приложения. Взгляните на приведенный ниже код, который можно найти в DPR-файле (несущественная часть опущена).

```
program …
begin
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.Run;
end;
```

В конструкторе класса TApplication, экземпляром которого является глобальная переменная Application (ее объявление находится в модуле Forms), происходит неявное создание главного окна приложения. Заголовок именно этого окна отображается на Панели задач (кстати, этот заголовок можно также изменить с помощью свойства Title объекта Application). Дескриптор главного окна приложения можно получить с помощью свойства Handle объекта Application.

Главное окно приложения делается невидимым (оно имеет нулевую высоту и ширину), чтобы создавалась иллюзия его отсутствия и можно было считать, что главной является именно форма, создаваемая первой.

Для подтверждения вышесказанного можно отобразить главное окно приложения, используя следующий код (листинг 1.5).

### Листинг 1.5. Отображение окна приложения

```
procedure TForm1.Button1Click(Sender: TObject);
begin
SetWindowPos(Application.Handle, 0, 0, 0, 200, 100,
SWP_NOZORDER or SWP_NOMOVE);
end;
```

В результате использования этого кода ширина окна станет равной 200, а высота 100, и вы сможете посмотреть на главное окно. Кстати, можно заметить, что при активизации этого окна (например, щелчке кнопкой мыши на заголовке) фокус ввода немедленно передается созданной первой, то есть главной, форме.

Теперь должно стать понятно, почему не мигала кнопка приложения при применении функций FlashWindow или FlashWindowEx к главной форме приложения. Недостаток этот теперь можно легко устранить, например, следующим образом (листинг 1.6).

### Листинг 1.6. Мигание кнопки приложения на Панели задач

```
procedure TForm1.Button2Click(Sender: TObject);
var
```

```
fl: FLASHWINFO;
begin
fl.cbSize:= SizeOf(fl);
fl.hwnd:= Application.Handle;
fl.dwFlags:= FLASHW_ALL;
fl.uCount:= 10;
fl.dwTimeout:= 200;
FlashWindowEx(fl);
end;
```

В данном случае одновременно инвертируется и заголовок окна приложения. Убедиться в этом можно, предварительно выполнив код листинга 1.5. Наконец, чтобы добиться одновременного мигания кнопки приложения на Панели задач и заголовка формы (произвольной, а не только главной), можно выполнить следующий код (листинг 1.7).

### Листинг 1.7. Мигание кнопки приложения и инверсия заголовка формы

```
procedure TForm1.Button3Click(Sender: TObject);
var
fl: FLASHWINFO;
begin
//Мигание кнопки
fl.cbSize:= SizeOf(fl);
fl.hwnd:= Application.Handle;
fl.dwFlags:= FLASHW_TRAY;
fl.uCount:= 10;
fl.dwTimeout:= 200;
FlashWindowEx(fl);
//Инверсия заголовка
fl.cbSize:= SizeOf(fl);
fl.hwnd:= Handle;
fl.dwFlags:= FLASHW_CAPTION;
fl.uCount:= 10;
fl.dwTimeout:= 200;
FlashWindowEx(fl);
end;
```

В данном случае инвертируется заголовок формы Form1. Кнопка на Панели задач может не только мигать, но и, например, быть скрыта или показана, когда в этом есть необходимость. Так, для скрытия кнопки приложения можно применить API-функцию ShowWindow:

```
ShowWindow(Application.Handle, SW_HIDE);
```

Чтобы показать кнопку приложения, можно функцию ShowWindow вызвать с равным SW_NORMAL вторым параметром.

# Полупрозрачные окна

В Windows 2000 впервые появилась возможность использовать прозрачность окон (в англоязычной документации такие полупрозрачные окна называются Layered windows). Сделать это можно, задав дополнительный стиль окна (о назначении и использовании оконных стилей вы можете узнать из материалов, представленных в гл. 2). Здесь не будет рассматриваться использование API-функций для работы с полупрозрачными окнами, так как их поддержка реализована для форм Delphi. Соответствующие свойства включены в состав класса TForm.

• AlphaBlend – включение или выключение прозрачности. Если параметр имеет значение True, то прозрачность включена, если False – выключена.

• AlphaBlendValue – значение, обратное прозрачности окна (от 0 до 255). Если параметр имеет значение 0, то окно полностью прозрачно, если 255 – непрозрачно.

Значения перечисленных свойств можно изменять как с помощью окна Object Inspector, так и во время выполнения программы (рис. 1.1).

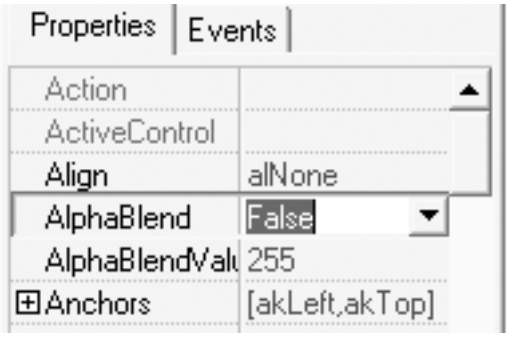



Рис. 1.1. Свойства для настройки прозрачности в окне Object Inspector

На рис. 1.2 наглядно продемонстрировано, как может выглядеть полупрозрачное окно (форма Delphi).

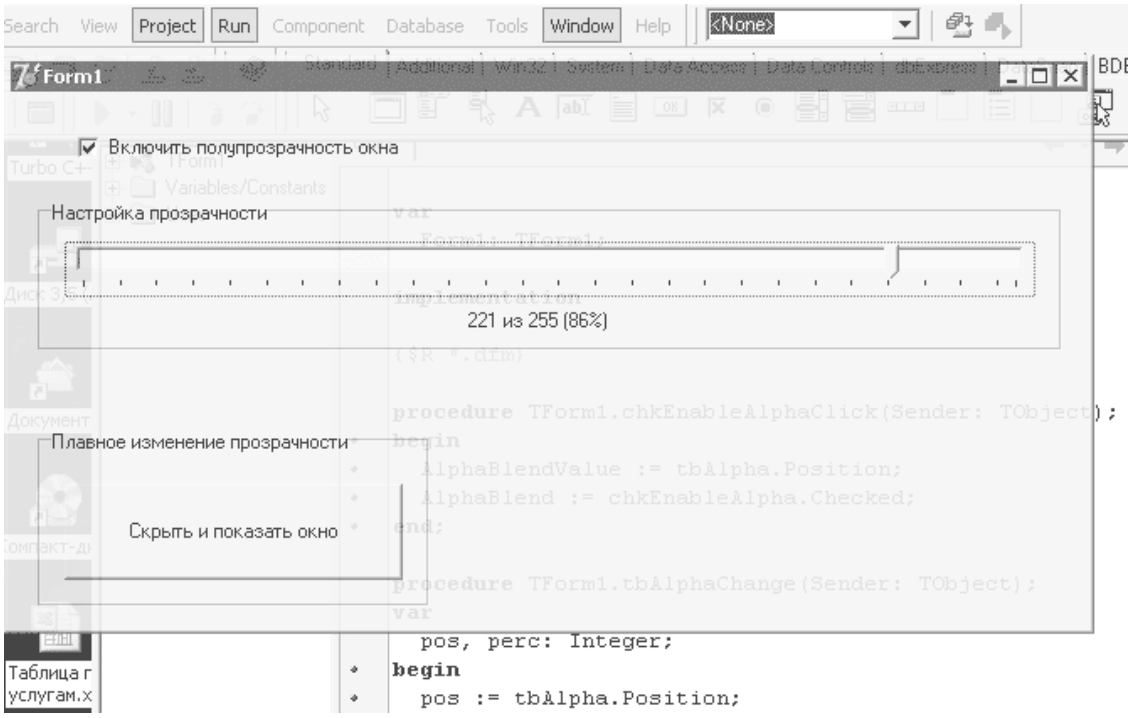


Рис. 1.2. Форма с коэффициентом прозрачности, равным 14 %

В качестве примера ниже показано, как используются свойства AlphaBlend и AlphaBlendValue для задания прозрачности окна во время выполнения программы (сочетание положения ползунка tbAlpha, состояния флажка chkEnableAlpha и подписи lblCurAlpha на форме, представленной на рис. 1.2) (листинг 1.8).

### Листинг 1.8. Динамическое изменение прозрачности окна

```
procedure TForm1.chkEnableAlphaClick(Sender: TObject);
begin
AlphaBlendValue:= tbAlpha.Position;
AlphaBlend:= chkEnableAlpha.Checked;
end;
procedure TForm1.tbAlphaChange(Sender: TObject);
var
pos, perc: Integer;
begin
pos:= tbAlpha.Position;
//Новое значение прозрачности
AlphaBlendValue:= pos;
//Обновим подпись под ползунком
perc:= pos * 100 div 255;
lblCurAlpha.Caption:= IntToStr(pos) + 'из 255 ('+IntToStr(perc) + '%)';
end;
```

Применив следующий код, можно реализовать довольно интересный эффект постепенного исчезновения, а затем появления формы (листинг 1.9).

### Листинг 1.9. Исчезновение и появление формы

```
implementation
var
isInc: Boolean; //Если True, то значение AlphaBlend формы
//увеличивается, если False, то уменьшается
//(форма скрывается)
procedure TForm1.cmbHideAndShowClick(Sender: TObject);
begin
if AlphaBlend then chkEnableAlpha.Checked:= False;
//Включаем прозрачность (подготовка к плавному скрытию)
AlphaBlendValue:= 255;
AlphaBlend:= True;
Refresh;
//Запускаем процесс скрытия формы
isInc:= False;
Timer1.Enabled:= True;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
var val: Integer;
```

```
begin
if not isInc then
begin
//"Растворение " окна
val:= AlphaBlendValue;
Dec(val, 10);
if val <= 0 then
begin
//Окно полностью прозрачно
val:= 0;
isInc:= True;
end
end
else begin
//Появление окна
val:= AlphaBlendValue;
Inc(val, 10);
if val >= 255 then
begin
//Окно полностью непрозрачно
val:= 255;
Timer1.Enabled:= False; //Процесс закончен
AlphaBlend:= False;
end
end;
AlphaBlendValue:= val;
end;
```

Единственная сложность (если это можно назвать сложностью) приведенного в листинге 1.9 алгоритма кроется в использовании таймера (Timerl) для инициирования изменения прозрачности окна. Так сделано для того, чтобы окно могло принимать пользовательский ввод, даже когда оно скрывается или постепенно показывается, и чтобы приложение не «съедало» все ресурсы на относительно слабой машине. Попробуйте сделать плавное изменение прозрачности в простом цикле, запустите его на каком-нибудь Pentium III 600 МГц без навороченной видеокарты – и сами увидите, что станет с бедной машиной.

Грамотное, а главное, уместное использование прозрачности окон может значительно повысить привлекательность интерфейса приложения (взгляните хотя бы на Winamp 5 при включенном параметре прозрачности окон).

# Окна и кнопки нестандартной формы

Далее будут рассмотрены некоторые стандартные возможности Windows, которые можно использовать для достижения большего разнообразия и привлекательности элементов оконного интерфейса. Приведенные здесь примеры измененяют формы элементов управления и, естественно, формы самих окон приложений.

## Регионы. Создание и использование

Рассматриваемые эффекты по изменению формы окон базируются на использовании регионов (областей) отсечения – в общем случае сложных геометрических фигур, ограничивающих область рисования окна. По умолчанию окна (в том числе и окна элементов управления) имеют область отсечения, заданную прямоугольным регионом с высотой и шириной, равной высоте и ширине самого окна.

Однако использование прямоугольных регионов для указания областей отсечения совсем не обязательно. Использование отсечения по заданному непрямоугольному региону при рисовании произвольного окна наглядно представлено на рис. 1.3: а – исходный прямоугольный вид формы; б – используемый регион, формирующий область отсечения; в – вид формы, полученный в результате рисования с отсечением по границам заданного региона.

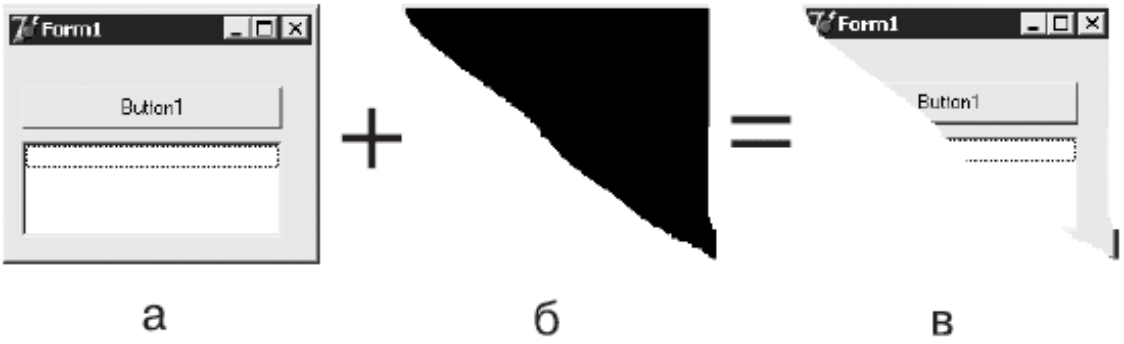



Рис. 1.3. Использование области отсечения при рисовании окна

Рассмотрим операции, позволяющие создавать, удалять и модифицировать регионы.

### Создание и удаление регионов

Создать несложные регионы различной формы можно с помощью следующих API-функций:

```
function CreateRectRgn(p1, p2, p3, p4: Integer): HRGN;
function CreateEllipticRgn(p1, p2, p3, p4: Integer): HRGN;
function CreateRoundRectRgn(p1, p2, p3, p4, p5, p6: Integer): HRGN;
```

Все перечисленные здесь и ниже функции создания регионов возвращают дескриптор GDI-объекта «регион». Он впоследствии и передается в различные функции, работающие с регионами.

Первая из приведенных функций (CreateRectRgn) предназначена для создания регионов прямоугольной формы. Параметры этой функции необходимо толковать следующим образом:

• p1 и p2 – горизонтальная и вертикальная координаты левой верхней точки прямоугольника;

• p3 и p4 – горизонтальная и вертикальная координаты правой нижней точки прямоугольника.

Следующая функция (CreateEllipticRgn) предназначена для создания региона в форме эллипса. Параметры этой функции – координаты прямоугольника (аналогично функции CreateRectRgn), в который вписывается требуемый эллипс.

Третья функция (CreateRoundRectRgn) создает регион в виде прямоугольника с округленными углами. При этом первые четыре параметра функции аналогичны соответствующим параметрам функции CreateRectRgn. Параметры p5 и p6 – ширина и высота сглаживающих углы эллипсов (рис. 1.4).

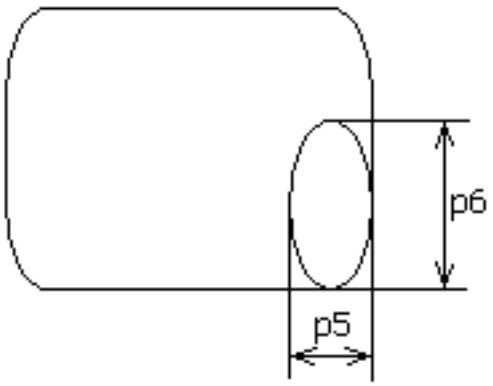



Рис. 1.4. Округление прямоугольника функцией CreateRoundRectRgn

Трех приведенных функций достаточно даже в том случае, если нужно создавать регионы очень сложной формы. Это достигается с помощью применения многочисленных операций над простыми регионами, как в приведенном далее примере создания региона по битовому шаблону. Однако рассмотрим еще одну несложную функцию, которая позволяет сразу создавать регионы-многоугольники по координатам точек вершин многоугольников:

```
function CreatePolygonRgn(const Points; Count, FillMode: Integer): HRGN;
```

Функция CreatePolygonRgn использует следующие параметры:

• Points – указатель на массив записей типа TPoint, каждый элемент которого описывает одну вершину многоугольника (координаты не должны повторяться);

• Count – количество записей в массиве, на который указывает параметр Points;

• FillMode – способ заливки региона (в данном случае определяет, попадает ли внутренняя область многоугольника в регион).

Параметр FillMode принимает значения WINDING (попадает любая внутренняя область) и ALTERNATE (попадает внутренняя область, если она находится между нечетной и следующей четной сторонами многоугольника).

**Примечание**

При создании регионов с помощью любой из указанных выше функций координаты точек задаются в системе координат того окна, в котором предполагается использовать регион. Так, если у вас есть кнопка размером 40 x 30 пикселов, левый верхний угол которой расположен на форме в точке (100; 100), то для того, чтобы создать для кнопки прямоугольный регион 20 x 15 пикселов с левой верхней точкой (0;0) относительно начала координат кнопки,

следует вызвать функцию CreateRectRgn с параметрами (0, 0, 19, 14), а не (100, 100, 119, 114).

Поскольку регион является GDI-объектом (подробнее в гл. 6), то для его удаления, если он не используется системой, применяется функция удаления GDI-объектов DeleteObject:

```
function DeleteObject(p1: HGDIOBJ): BOOL;
```

## Регион как область отсечения при рисовании окна

Обычно регион нужно удалять в том случае, если он не используется системой, однако после того, как регион назначен окну в качестве области отсечения, удалять его не следует. Функция назначения региона окну имеет следующий вид:

```
function SetWindowRgn(hWnd: HWND; hRgn: HRGN; bRedraw: BOOL): Integer;
```

Функция возвращает 0, если произвести операцию не удалось, и ненулевое значение в случае успешного выполнения операции. Параметры функции SetWindowRgn следующие:

- hWnd – дескриптор окна, для которого устанавливается область отсечения (свойство Handle формы или элемента управления);
- hRgn – дескриптор региона, назначаемого в качестве области отсечения (в простейшем случае является значением, возвращенным одной из функций создания региона);
- bRedraw – флаг перерисовки окна после назначения новой области отсечения (для видимых окон обычно используется значение True, для невидимых – False).

Чтобы получить копию региона, формирующего область отсечения окна, можно использовать API-функцию GetWindowRgn:

```
function GetWindowRgn(hWnd: HWND; hRgn: HRGN): Integer;
```

Первый параметр функции – дескриптор (Handle) интересующего окна. Второй параметр – дескриптор предварительно созданного региона, который в случае успеха модифицируется функцией GetWindowRgn так, что становится копией региона, формирующего область отсечения окна. Значения целочисленных констант – возможных возвращаемых значений функции – следующие:

- NULLREGION – пустой регион;
- SIMPLEREGION – регион в форме прямоугольника;
- COMPLEXREGION – регион сложнее, чем прямоугольник;
- ERROR – при выполнении функции возникла ошибка либо окну задана область отсечения.

Далее приведен пример использования функции GetWindowRgn (предполагается, что приведенный ниже код является телом одного из методов класса формы).

```
var rgn: HRGN;
begin
rgn:= CreateRectRgn(0,0,0,0); //Первоначальная форма региона не важна
if GetWindowRgn(Handle, rgn) <> ERROR then
begin
//Операции с копией региона, формирующего область отсечения окна…
```

```
end;
DeleteObject(rgn); //Мы пользовались копией региона, которую должны
//удалить (здесь или в ином месте, но сами)
end;
```

## Операции над регионами

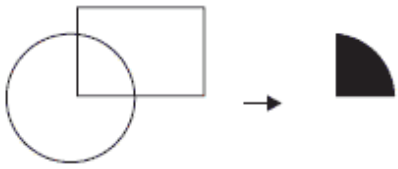
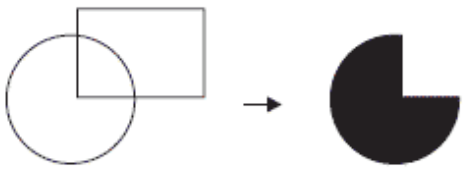
При рассказе о функциях создания регионов неоднократно упоминалось о возможности комбинирования регионов для получения сложных форм. Пришло время кратко рассмотреть операции над регионами. Все операции по комбинированию регионов осуществляются с помощью функции CombineRgn:

```
function CombineRgn(p1, p2, p3: HRGN; p4: Integer): Integer;
```

Параметры этой функции следующие:
- p1 – регион (предварительно созданный), предназначенный для сохранения результата;
- p2, p3 – регионы-аргументы операции;
- p4 – тип операции над регионами.

Более подробно действие функции CombineRgn при различных значениях параметра p4 поясняется в табл. 1.2.

### Таблица 1.2. Операции функции CombineRgn

| Значение p4 | Операция | Пример |
|---|---|---|
| RGN_AND | Пересечение регионов |  |
| RGN_OR | Объединение регионов |  |
| RGN_DIFF | Разность регионов (часть региона p2, не являющаяся частью p3) |  |
| RGN_XOR | Так называемое исключающее ИЛИ (объединение непересекающихся частей регионов p2 и p3) |  |

Кроме приведенных в табл. 1.2 констант, в качестве параметра p4 функции CombineRgn можно использовать параметр RGN_COPY. При его использовании копируется регион, заданный параметром p2, в регион, заданный параметром p1.

Тщательно рассчитывая координаты точек регионов-аргументов, можно с использованием функции CombineRgn создавать регионы самых причудливых форм, в чем вы сможете убедиться ниже.

Наконец, после теоретического отступления можно рассмотреть несколько примеров создания и преобразования регионов, предназначенных для формирования области отсечения окон (форм и элементов управления на формах).

## Закругленные окна и многоугольники

Сначала самые простые примеры: создание регионов без операций над ними. Формы всех трех приведенных здесь примеров содержат по три кнопки различной ширины и высоты, которым также задаются области отсечения.

> **Примечание**
>
> В приведенных далее примерах регионы для области отсечения окна создаются при обработке события FormCreate. Однако это сделано только для удобства отладки и тестирования примеров и ни в коем случае не должно наталкивать вас на мысль, что этот способ является единственно правильным. На самом деле, если в приложении много окон, использующих области отсечения сложной формы, то запуск приложения (время от момента запуска до показа первой формы) может длиться, по крайней мере, несколько секунд. Так происходит потому, что все формы создаются перед показом первой (главной) формы (см. DPR-файл проекта). Исправить ситуацию можно, создавая формы вручную в нужный момент времени либо создавая регионы для областей отсечения, например, перед первым отображением каждой конкретной формы.

В приведенном ниже обработчике события FormCreate создается окно в форме эллипса с тремя кнопками такой же формы (листинг 1.10).

### Листинг 1.10. Окно и кнопки в форме эллипсов

```
procedure TfrmElliptic.FormCreate(Sender: TObject);
var
formRgn, but1Rgn, but2Rgn, but3Rgn: HRGN;
begin
//Создаем регионы кнопок
but1Rgn:= CreateEllipticRgn(0, 0, Button1.Width–1, Button1.Height–1);
SetWindowRgn(Button1.Handle, but1Rgn, False);
but2Rgn:= CreateEllipticRgn(0, 0, Button2.Width–1, Button2.Height–1);
SetWindowRgn(Button2.Handle, but2Rgn, False);
but3Rgn:= CreateEllipticRgn(0, 0, Button3.Width–1, Button3.Height–1);
SetWindowRgn(Button3.Handle, but3Rgn, False);
//Регион для окна
formRgn:= CreateEllipticRgn(0, 0, Width–1, Height–1);
SetWindowRgn(Handle, formRgn, True);
end;
```

Ширина и высота эллипсов в приведенном примере равна, соответственно, ширине и высоте тех окон, для которых создаются регионы. При необходимости это можно изменить,

например, если требуется, чтобы все кнопки были одной величины независимо от размера, установленного во время проектирования формы.

Результат выполнения кода листинга 1.10 можно увидеть на рис. 1.5.

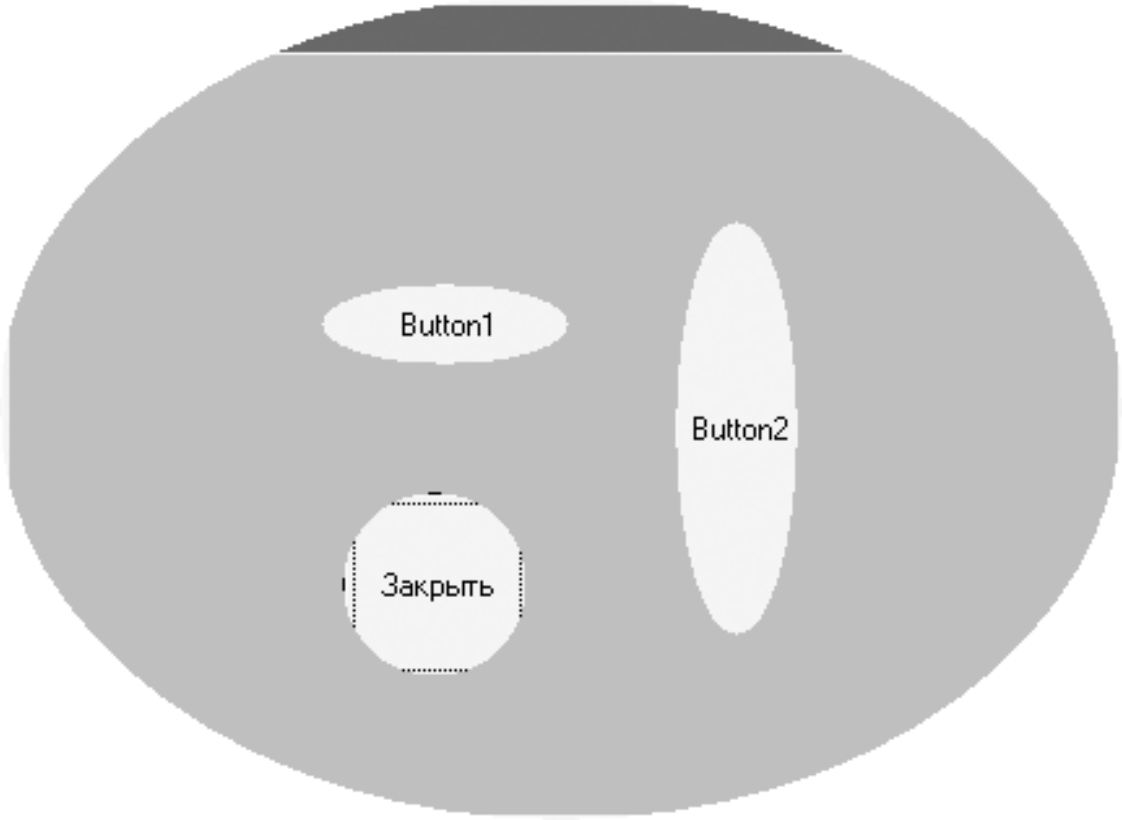



Рис. 1.5. Окно и кнопки в форме эллипсов

Далее рассмотрим не менее интересный (возможно, даже более полезный на практике) пример – округление углов формы и кнопок на ней, то есть применение области отсечения в форме прямоугольника с округленными углами. Ниже приведен код реализации соответствующего обработчика события FormCreate (листинг 1.11).

### Листинг 1.11. Окно и кнопки с округленными краями

```
procedure TfrmRoundRect.FormCreate(Sender: TObject);
var
formRgn, but1Rgn, but2Rgn, but3Rgn: HRGN;
begin
//Создаем регионы для кнопок
but1Rgn:= CreateRoundRectRgn(0, 0, Button1.Width–1, Button1.Height–1,
Button1.Width div 5, Button1.Height div 5);
SetWindowRgn(Button1.Handle, but1Rgn, False);
but2Rgn:= CreateRoundRectRgn(0, 0, Button2.Width–1, Button2.Height–1,
Button2.Width div 5, Button2.Height div 5);
SetWindowRgn(Button2.Handle, but2Rgn, False);
but3Rgn:= CreateRoundRectRgn(0, 0, Button3.Width–1, Button3.Height–1,
Button3.Width div 5, Button3.Height div 5);
SetWindowRgn(Button3.Handle, but3Rgn, False);
//Регион для окна
```

```
formRgn:= CreateRoundRectRgn(0, 0, Width–1, Height–1,
Width div 5, Height div 5);
SetWindowRgn(Handle, formRgn, False);
end;
```

В листинге 1.11 размеры округляющих эллипсов вычисляются в расчете из размеров конкретного окна (20 % от его ширины и 20 % от высоты). Это смотрится не всегда красиво. В качестве альтернативы для ширины и высоты скругляющих эллипсов можно использовать фиксированные небольшие значения.

Результат выполнения кода листинга 1.11 можно увидеть на рис. 1.6.

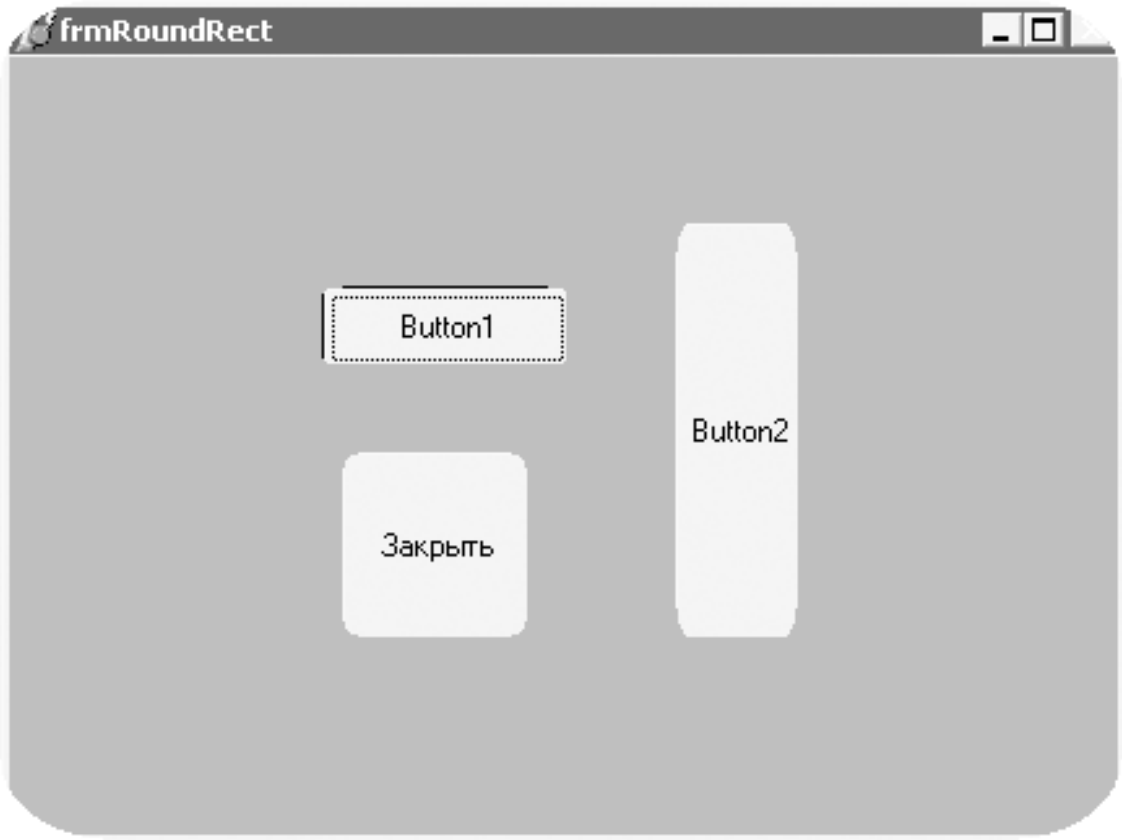



Рис. 1.6. Окно и кнопки с округленными краями

Теперь самый интересный из предусмотренных примеров – создание окна и кнопок в форме многоугольников, а именно: окна в форме звезды, кнопок в форме треугольника, пяти– и шестиугольника (рис. 1.7).

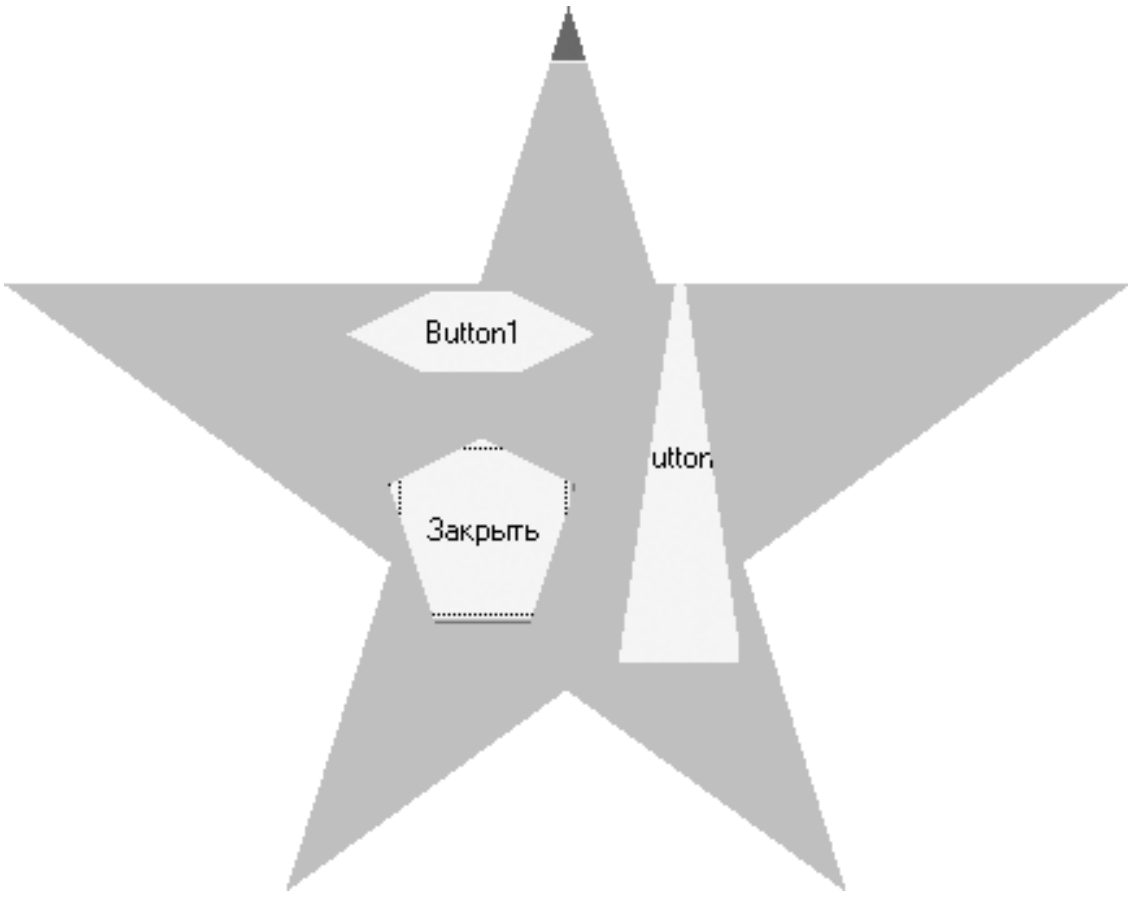



Рис. 1.7. Окно и кнопки в форме многоугольников

Код создания регионов для областей отсечения данного примера приведен в листинге 1.12.

### Листинг 1.12. Окно и кнопки в форме многоугольников

```
procedure TfrmPoly.FormCreate(Sender: TObject);
var
points: array [0..5] of TPoint;
formRgn, but1Rgn, but2Rgn, but3Rgn: HRGN;
begin
//Создаем регионы для окна и кнопок
//..шестиугольная кнопка
Make6Angle(Button1.Width, Button1.Height, points);
but1Rgn:= CreatePolygonRgn(points, 6, WINDING);
SetWindowRgn(Button1.Handle, but1Rgn, False);
//..треугольная кнопка
Make3Angle(Button2.Width, Button2.Height, points);
but2Rgn:= CreatePolygonRgn(points, 3, WINDING);
SetWindowRgn(Button2.Handle, but2Rgn, False);
//..пятиугольная кнопка
Make5Angle(Button3.Width, Button3.Height, points);
but3Rgn:= CreatePolygonRgn(points, 5, WINDING);
SetWindowRgn(Button3.Handle, but3Rgn, False);
```

```
//..форма в виде звезды
MakeStar(Width, Height, points);
formRgn:= CreatePolygonRgn(points, 5, WINDING);
SetWindowRgn(Handle, formRgn, False);
end;
```

Особенностью создания регионов в приведенном листинге является использование дополнительных процедур для заполнения массива points координатами точек-вершин многоугольников определенного вида. Все эти процедуры принимают, помимо ссылки на сам массив points, ширину и высоту прямоугольника, в который должен быть вписан многоугольник. Описание процедуры создания треугольника приведено в листинге 1.13.

### Листинг 1.13. Создание треугольника

```
procedure Make3Angle(width, height: Integer; var points: array of TPoint);
begin
points[0].X:= 0;
points[0].Y:= height – 1;
points[1].X:= width div 2;
points[1].Y:= 0;
points[2].X:= width – 1;
points[2].Y:= height – 1;
end;
```

В листинге 1.14 приведено описание процедуры создания шестиугольника.

### Листинг 1.14. Создание шестиугольника

```
procedure Make6Angle(width, height: Integer; var points: array of TPoint);
begin
points[0].X:= 0;
points[0].Y:= height div 2;
points[1].X:= width div 3;
points[1].Y:= 0;
points[2].X:= 2 * (width div 3);
points[2].Y:= 0;
points[3].X:= width – 1;
points[3].Y:= height div 2;
points[4].X:= 2 * (width div 3);
points[4].Y:= height – 1;
points[5].X:= width div 3;
points[5].Y:= height – 1;
end;
```

Листинг 1.15 содержит описание процедуры создания пятиугольника (неправильного).

### Листинг 1.15. Создание пятиугольника

```
procedure Make5Angle(width, height: Integer; var points: array of TPoint);
var a: Integer; //Сторона пятиугольника
begin
a:= width div 2;
points[0].X:= a;
points[0].Y:= 0;
points[1].X:= width – 1;
points[1].Y:= a div 2;
points[2].X:= 3 * (a div 2);
points[2].Y:= height – 1;
points[3].X:= a div 2;
points[3].Y:= height – 1;
points[4].X:= 0;
points[4].Y:= a div 2;
end;
```

Пятиугольная звезда, используемая как область отсечения формы, создается с помощью описанной в листинге 1.15 процедуры Make5Angle. После ее создания изменяется порядок следования вершин пятиугольника, чтобы их обход при построении региона выполнялся в той же последовательности, как рисуется звезда карандашом на бумаге (например, 1-3-5-2-4) (листинг 1.16).

### Листинг 1.16. Создание пятиугольной звезды

```
procedure MakeStar(width, height: Integer; var points: array of TPoint);
begin
Make5Angle(width, height, points);
//При построении звезды точки пятиугольника обходятся не по порядку,
//а через одну
Swap(points[1], points[2]);
Swap(points[2], points[4]);
Swap(points[3], points[4]);
end;
```

Процедура MakeStart (листинг 1.16) использует дополнительную процедуру Swap, меняющую местами значения двух передаваемых в нее аргументов. Процедура Swap реализуется чрезвычайно просто и потому в тексте книги не приводится.

## Комбинированные регионы

Вы уже научились создавать и использовать простые регионы. Однако многим может показаться недостаточным тех форм окон, которые можно получить с использованием в качестве области отсечения лишь одного несложного региона. Пришло время заняться созданием окон более сложной формы, применяя рассмотренные ранее операции над регионами.

## «Дырявая» форма

Этот простейший пример сомнительной полезности предназначен для первого знакомства с операциями над регионами. Здесь применяется только одна из возможных операций – операция XOR для формирования «дырок» в форме (рис. 1.8).

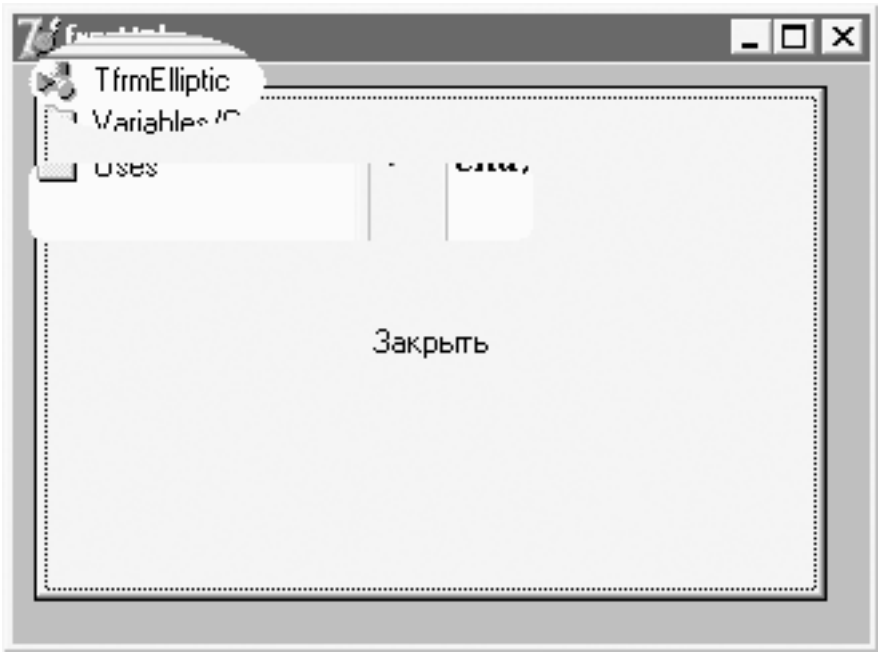



Рис. 1.8. «Дырки» в форме

На рис. 1.8 явно видно, как в «дырках» формы просвечивает одно из окон среды разработки Delphi. При этом, когда указатель находится над «дыркой», сообщения от мыши получают те окна, части которых видны в «дырке».

Программный код, приводящий к созданию формы столь необычного вида, приведен в листинге 1.17.

### Листинг 1.17. Создание «дырок» в форме

```
procedure TfrmHole.FormCreate(Sender: TObject);
var
rgn1, rgn2: HRGN; //"Регионы-дырки" в форме
formRgn: HRGN;
begin
//Создание региона для формы
formRgn:= CreateRectRgn(0, 0, Width – 1, Height – 1);
//Создание регионов для "дырок"
rgn1:= CreateEllipticRgn(10, 10, 100, 50);
rgn2:= CreateRoundRectRgn(10, 60, 200, 90, 10, 10);
//Создание "дырок" в регионе формы
CombineRgn(formRgn, formRgn, rgn1, RGN_XOR);
CombineRgn(formRgn, formRgn, rgn2, RGN_XOR);
SetWindowRgn(Handle, formRgn, True);
//Регионы для "дырок" больше не нужны
```

```
DeleteObject(rgn1);
DeleteObject(rgn2);
end;
```

## Сложная комбинация регионов

Теперь пришла очередь рассмотреть более сложный, но и гораздо более интересный пример. Последовательное применение нескольких операций над регионами приводит к созданию формы, показанной на рис. 1.9 (белое пространство – «вырезанные» части формы).

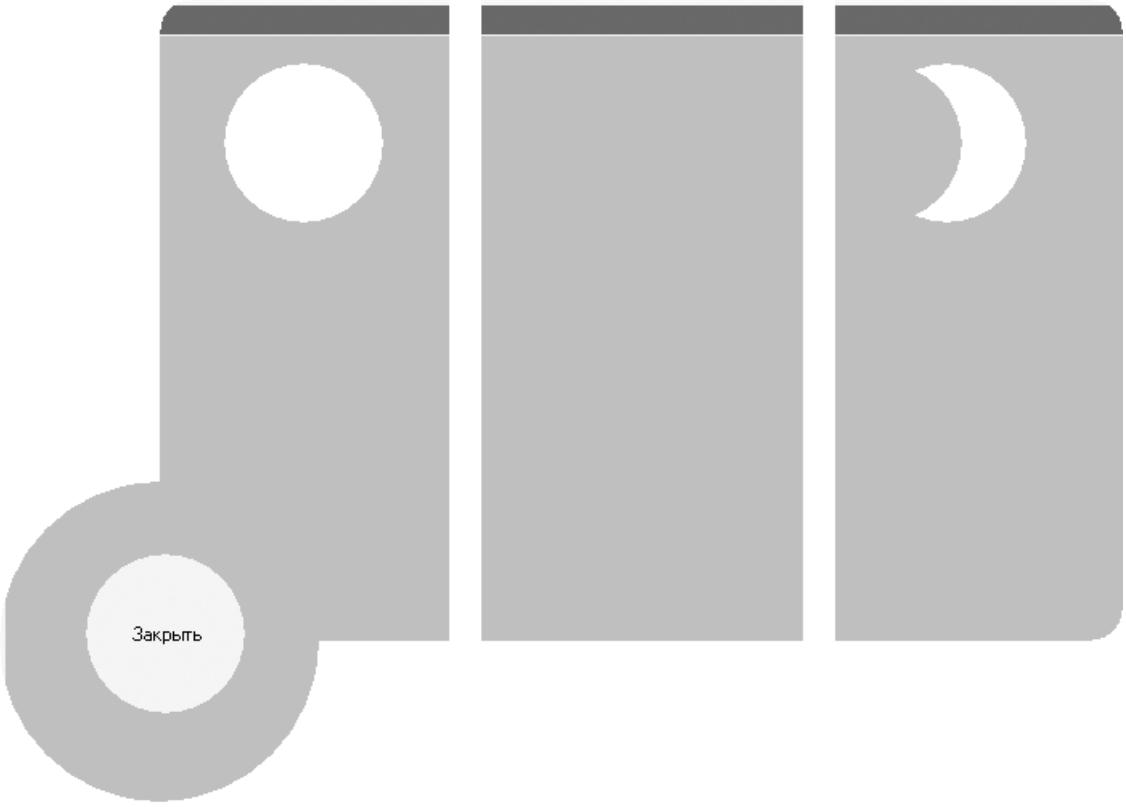



Рис. 1.9. Сложная комбинация регионов

Описание процедуры, выполняющей операции над регионами, приведено в листинге 1.18.

## Листинг 1.18. Сложная комбинация регионов

```
procedure TfrmManyRgn.FormCreate(Sender: TObject);
var
r1, r2, r3, r4, r5, r6, r7: HRGN;
formRgn: HRGN;
butRgn: HRGN;
begin
//Создание регионов
r1:= CreateRoundRectRgn(100, 0, 700, 400, 40, 40);
r2:= CreateRectRgn(280, 0, 300, 399);
r3:= CreateRectRgn(500, 0, 520, 399);
```

```
r4:= CreateEllipticRgn(140, 40, 240, 140);
r5:= CreateEllipticRgn(0, 300, 200, 500);
r6:= CreateEllipticRgn(500, 40, 600, 140);
r7:= CreateEllipticRgn(540, 40, 640, 140);
//Комбинирование
//..разрезы в основном регионе
CombineRgn(r1, r1, r2, RGN_XOR);
CombineRgn(r1, r1, r3, RGN_XOR);
//..круглая "дырка" в левой стороне
CombineRgn(r1, r1, r4, RGN_XOR);
//..присоединение круга в левой нижней части
CombineRgn(r1, r1, r5, RGN_OR);
//..создание "дырки" в форме полумесяца
CombineRgn(r7, r7, r6, RGN_DIFF);
CombineRgn(r1, r1, r7, RGN_XOR);
formRgn:= CreateRectRgn(0, 0, 0, 0);
CombineRgn(formRgn, r1, 0, RGN_COPY);
DeleteObject(r1);
DeleteObject(r2);
DeleteObject(r3);
DeleteObject(r4);
DeleteObject(r5);
DeleteObject(r6);
DeleteObject(r7);
//Создание круглой кнопки закрытия
butRgn:= CreateEllipticRgn(50, 50, 150, 150);
SetWindowRgn(Button1.Handle, butRgn, False);
SetWindowRgn(Handle, formRgn, True);
end;
```

В этом листинге подписано, какие операции предназначены для создания каких элементов итогового региона. В операциях участвуют семь регионов. Расположение используемых в операциях регионов показано на рис. 1.10.

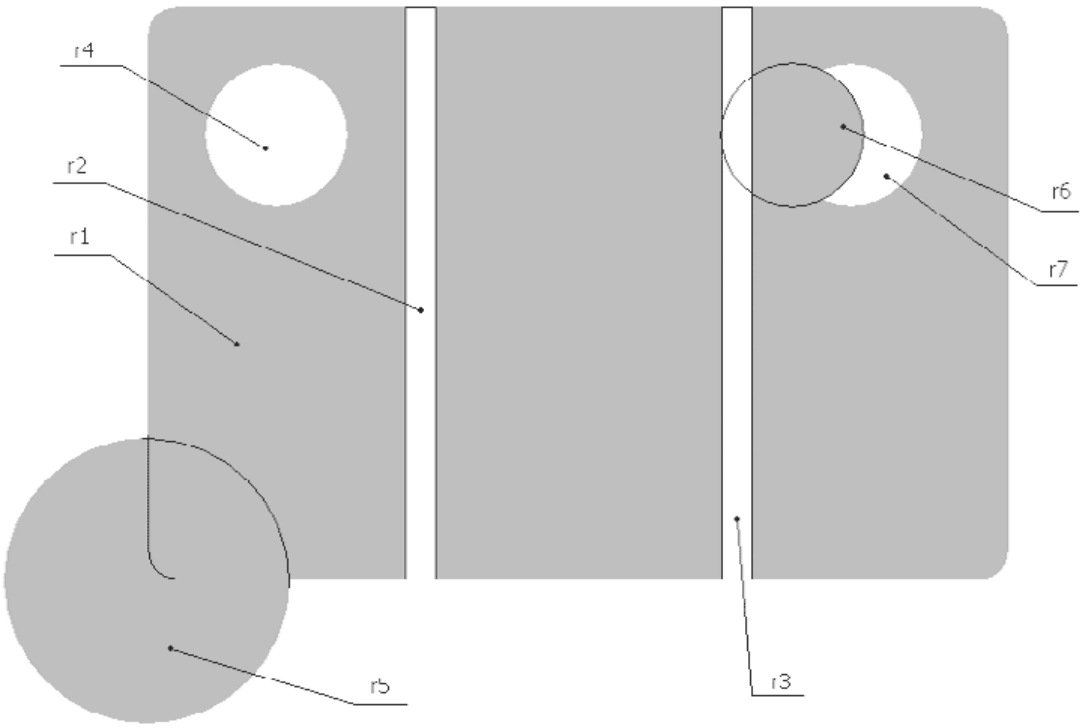



Рис. 1.10. Элементарные регионы, используемые для получения формы, представленной на рис. 1.9

## Использование шаблона

Предыдущий пример наглядно демонстрирует мощь функции CombineRgn при построении регионов сложной формы. Однако существует огромное количество предметов, контуры которых крайне сложно повторить, комбинируя простые регионы. Построение многоугольных регионов с большим количеством точек может в этом случае выручить, но ведь это крайне нудно и утомительно.

Если есть изображение предмета, контуры которого должны совпадать с контурами региона, то гораздо проще при построении региона обрабатывать само изображение, выбирая все точки, для которых выполняется определенное условие. Используемое изображение и будет тем шаблоном, по которому «вырезается» регион нужной формы.

Рассмотрим простейший пример: есть изображение, каждая точка которого должна попасть в результирующий регион, если ее цвет не совпадает с заданным цветом фона. При этом изображение анализируется по так называемым «скан-линиям», то есть построчно. Из подряд идущих точек не фонового цвета формируются прямоугольные регионы, которые объединяются с результирующим регионом. Пример возможного используемого шаблона приведен на рис. 1.11.

Рис. 1.11. Пример растрового изображения-шаблона

Код функции построения региона указанным способом приведен в листинге 1.19.

### Листинг 1.19. Построение региона по шаблону

```
function RegionFromPicture(pict:TPicture; backcolor: TColor): HRGN;
var
rgn, resRgn: HRGN;
x, y, xFirst: Integer;
begin
resRgn:= CreateRectRgn(0, 0, 0, 0); //Результирующий регион
//Анализируем каждую скан-линию рисунка (по горизонтали)
for y:= 0 to pict.Height – 1 do
begin
x:= 0;
while x < pict.Width do
begin
if (pict.Bitmap.Canvas.Pixels[x, y] <> backcolor) then
begin
xFirst:= x;
Inc(x);
//Определим часть линии, окрашенной не цветом фона
while (x < pict.Width) and
(pict.Bitmap.Canvas.Pixels[x, y] <> backcolor) do Inc(x);
//Создаем регион для части скан-линии и добавляем его к
//результирующему региону
```

```
rgn:= CreateRectRgn(xFirst, y, x–1, y+1);
CombineRgn(resRgn, resRgn, rgn, RGN_OR);
DeleteObject(rgn);
end;
Inc(x);
end;
end;
RegionFromPicture:= resRgn;
end;
```

Загрузка изображения-шаблона и создание региона могут происходить, например, при создании формы (листинг 1.20).

### Листинг 1.20. Создание региона для области отсечения формы

```
procedure TfrmTemplate.FormCreate(Sender: TObject);
var
pict: TPicture;
begin
//Загрузка изображения и создание региона
//(считаем, что цвет фона – белый)
pict:= TPicture.Create;
pict.LoadFromFile('back.bmp');
SetWindowRgn(Handle, RegionFromPicture(pict, RGB(255,255,255)), True);
end;
```

В листинге 1.20 подразумевается, что используется файл back.bmp, находящийся в той же папке, что и файл приложения. Цвет фона – белый. Таким образом, если шаблон, показанный на рис. 1.11, хранится в файле back.bmp, то в результате получается форма, показанная на рис. 1.12.

Рис. 1.12. Результат построения региона по шаблону

# Немного о перемещении окон

Кроме придания необычного вида окнам способами, рассмотренными выше, можно также несколько разнообразить интерфейс за счет оригинального использования перемещения окон. Ниже показано, как можно самостоятельно назначать области, позволяющие перетаскивать форму. Еще один пример демонстрирует один из способов дать пользователю возможность самому определять расположение элементов управления на форме.

# Перемещение за клиентскую область

Здесь на конкретном примере (перемещение формы за любую точку клиентской области) продемонстрировано, как можно самостоятельно определять положение некоторых важных элементов окна. Под элементами окна здесь подразумеваются:

• строка заголовка (предназначена не только для отображения текста заголовка, но и служит областью захвата при перемещении окна мышью);

• границы окна (при щелчке кнопкой мыши на верхней, нижней, правой и левой границе можно изменять размер окна, правда, если стиль окна это допускает);

• четыре угла окна (предназначены для изменения размера окна с помощью мыши);

• системные кнопки закрытия, разворачивания, сворачивания, контекстной справки (обычно расположены в строке заголовка окна);

• горизонтальная и вертикальная полосы прокрутки;

• системное меню (раскрывается щелчком кнопкой мыши на значке окна);

• меню – полоса меню (обычно расположена вверху окна);

• клиентская область – по умолчанию все пространство окна, кроме строки заголовка, меню и полос прокрутки.

Каждый раз, когда над окном перемещается указатель мыши либо происходит нажатие кнопки мыши, система посылает соответствующему окну сообщение WM_ NCHITTEST для определения того, над которой из перечисленных выше областей окна находится указатель. Обработчик этого сообщения, вызываемый по умолчанию, информирует систему о расположении элементов окна в привычных для пользователя местах: заголовка – сверху, правой границы – справа и т. д.

Как вы, наверное, уже догадались, реализовав свой обработчик сообщения WM_ NCHITTEST, можно изменить назначение элементов окна. Этот прием как раз и реализован в листинге 1.21.

### Листинг 1.21. Перемещение окна за клиентскую область

```
procedure TfrmMoveClient.WMNCHitTest(var Message: TWMNCHitTest);
var
rc: TRect;
p: TPoint;
begin
//Если точка приходится на клиентскую область, то заставим систему
//считать эту область частью строки заголовка
rc:= GetClientRect();
p.X:= Message.XPos;
p.Y:= Message.YPos;
p:= ScreenToClient(p);
```

```
if PtInRect(rc, p) then
Message.Result:= HTCAPTION
else
//Обработка по умолчанию
Message.Result:= DefWindowProc(Handle, Message.Msg, 0, 65536 * Message.YPos + Message.XPos);
end;
```

Приведенный в листинге 1.21 обработчик переопределяет положение только строки заголовка, возвращая значение HTCAPTION. Этот обработчик может возвращать следующие значения (целочисленные константы, возвращаемые функцией DefWindowProc):

• HTBORDER – указатель мыши находится над границей окна (размер окна не изменяется);

• HTBOTTOM, HTTOP, HTLEFT, HTRIGHT – указатель мыши находится над нижней, верхней, левой или правой границей окна соответственно (размер окна можно изменить, «потянув» за границу);

• HTBOTTOMLEFT, HTBOTTOMRIGHT, HTTOPLEFT, HTTOPRIGHT – указатель мыши находится в левом нижнем, правом нижнем, левом верхнем или правом верхнем углу окна (размер окна можно изменять по диагонали);

• HTSIZE, HTGROWBOX – указатель мыши находится над областью, предназначенной для изменения размера окна по диагонали (обычно в правом нижнем углу окна);

• HTCAPTION – указатель мыши находится над строкой заголовка окна (за это место окно перемещается);

• HTCLIENT – указатель мыши находится над клиентской областью окна;

• HTCLOSE – указатель мыши находится над кнопкой закрытия окна;

• HTHELP – указатель мыши находится над кнопкой вызова контекстной справки;

• HTREDUCE, HTMINBUTTON – указатель мыши находится над кнопкой минимизации окна;

• HTZ OOM, HTMAXBUTTON – указатель мыши находится над кнопкой максимизации окна;

• HTMENU – указатель мыши находится над полосой меню окна;

• HTSYSMENU – указатель мыши находится над значком окна (используется для вызова системного меню);

• HTHSCROLL, HTVSCROLL – указатель находится над вертикальной или горизонтальной полосой прокрутки, соответственно;

• HTTRANSPARENT – если возвращается это значение, то сообщение пересылается окну, находящемуся под данным окном (окна должны принадлежать одному потоку);

• HTNOWHERE – указатель не находится над какой-либо из областей окна (например, на границе между окнами);

• HTERROR – то же, что и HTNOWHERE, только при возврате этого значения обработчик по умолчанию (DefWindowProc) воспроизводит системный сигнал, сигнализирующий об ошибке.

## Перемещаемые элементы управления

В завершение материала о перемещении окон приведу один совсем несложный, но довольно интересный пример, позволяющий прямо «на лету» изменять внешний вид приложения. Достигается это благодаря возможности перемещения и изменения размера элементов управления так, будто это обычные перекрывающиеся окна.

Чтобы вас заинтересовать, сразу приведу результат работы примера. На рис. 1.13 показан внешний вид формы в начале работы примера.

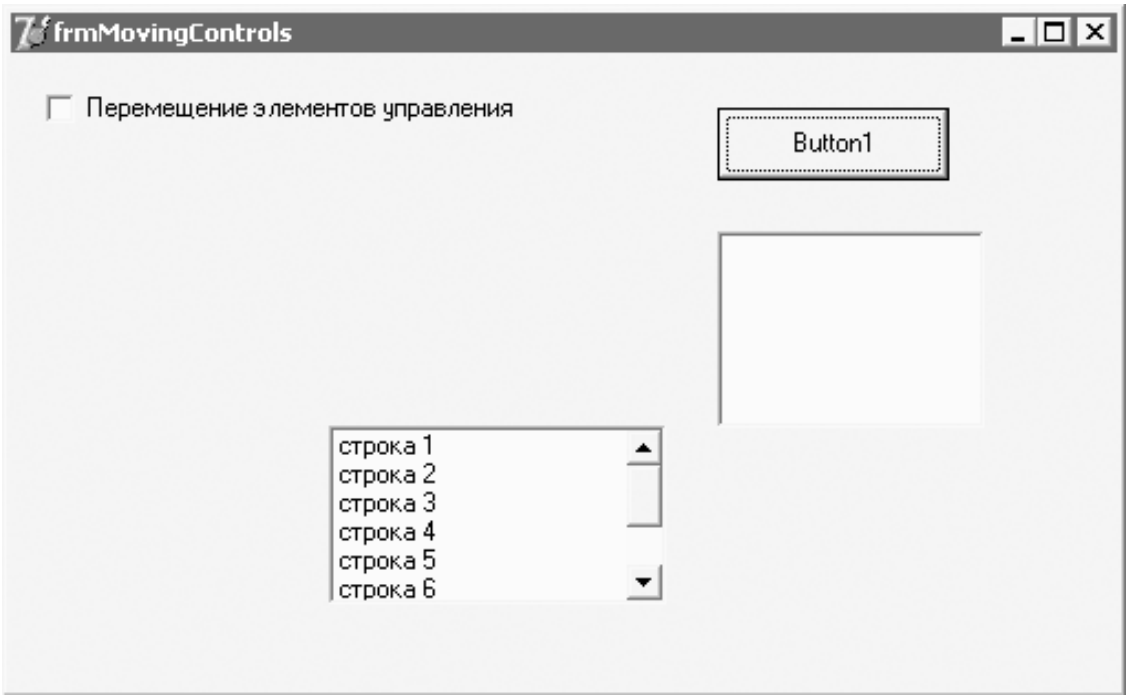



Рис. 1.13. Первоначальный вид формы

После установки флажка Перемещение элементов управления получается результат, показанный на рис. 1.14.

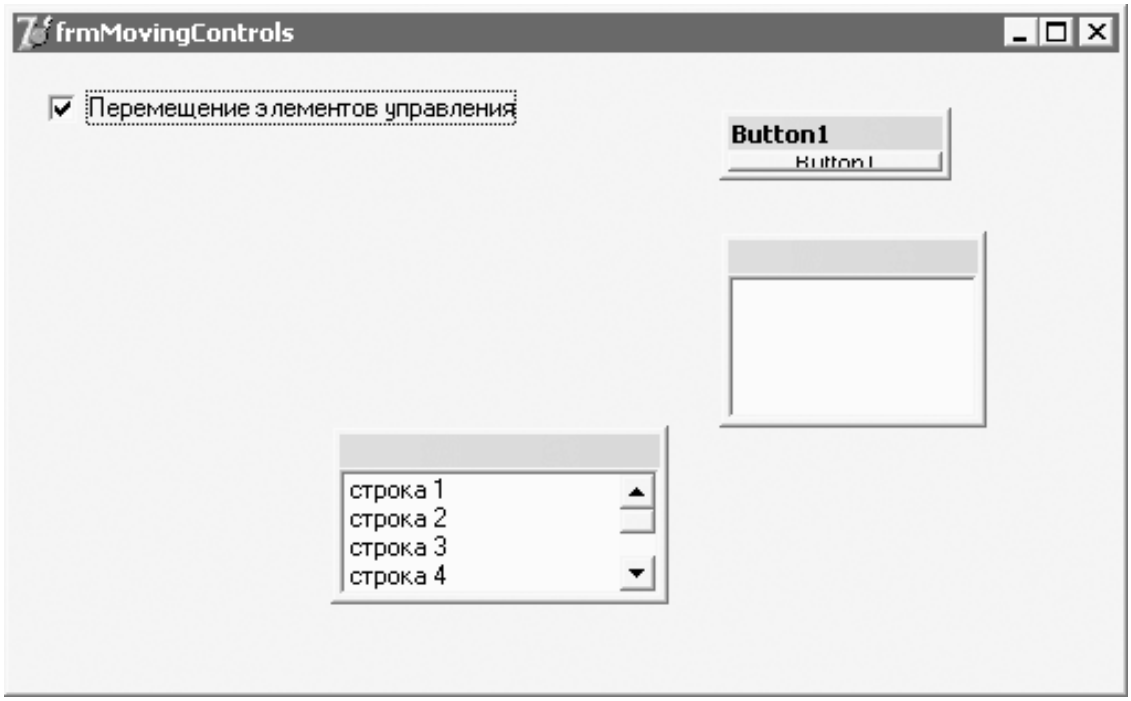



Рис. 1.14. Элементы управления можно перемещать (флажок не учитывается)

В результате выполнеия произвольных перемещений, изменения размера окон, занявших место элементов управления, снятия флажка получаем измененный интерфейс формы (рис. 1.15).

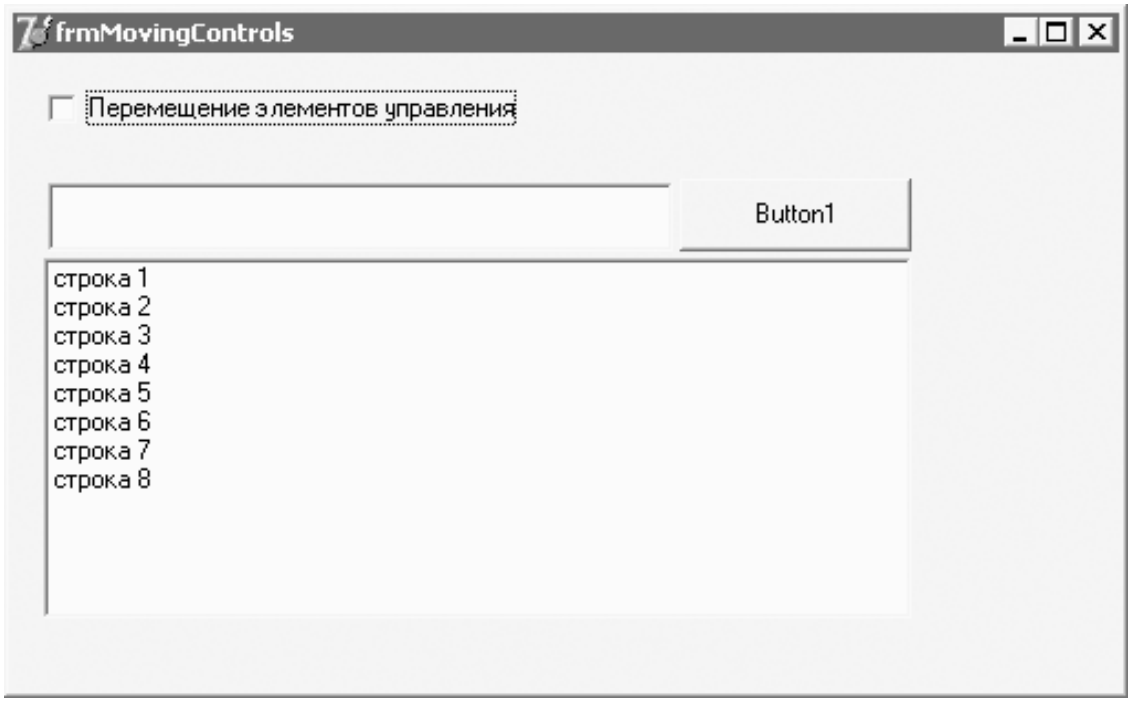



Рис. 1.15. Внешний вид формы после перемещения элементов управления

Как же достигнут подобный эффект? Очень просто. Вы уже знаете, что элементы управления рисуются внутри своих собственных окон (дочерних по отношению к окну формы). Окна элементов управления отличает отсутствие в их стиле флагов (подробнее в гл. 2), позволяющих отображать рамку и изменять размер окна. Это легко изменить, самостоятельно задав нужные флаги в стиле окна с помощью API-функции SetWindowLong. Для удобства можно написать отдельную процедуру, которая будет дополнять стиль окна флагами, необходимыми для перемещения и изменения размера (как, собственно, и сделано в примере) (листинг 1.22).

### Листинг 1.22. Разрешение перемещения и изменения размера

```
procedure MakeMovable(Handle: HWND);
var
style: LongInt;
flags: UINT;
begin
//Разрешаем перемещение элемента управления
style:= GetWindowLong(Handle, GWL_STYLE);
style:= style or WS_OVERLAPPED or WS_THICKFRAME or WS_CAPTION;
SetWindowLong(Handle, GWL_STYLE, style);
style:= GetWindowLong(Handle, GWL_EXSTYLE);
style:= style or WS_EX_TOOLWINDOW;
SetWindowLong(Handle, GWL_EXSTYLE, style);
//Перерисуем в новом состоянии
flags:= SWP_NOMOVE or SWP_NOSIZE or SWP_DRAWFRAME or SWP_NOZORDER;
SetWindowPos(Handle, 0, 0, 0, 0, 0, flags);
end;
```

Как можно увидеть, дополнительные флаги задаются в два этапа. Сначала считывается старое значение стиля окна. Потом с помощью двоичной операции ИЛИ стиль (задается целочисленным значением) дополняется новыми флагами. Это делается для того, чтобы не пропали ранее установленные значения стиля окна.

Вообще, процедура MakeMovable изменяет два стиля окна: обычный и расширенный. Расширенный стиль окна изменяется лишь для того, чтобы строка заголовка получившегося окна занимала меньше места (получаем так называемое окно панели инструментов). Полный перечень как обычных, так и расширенных стилей можно просмотреть в приложении 2.

Логично также реализовать процедуру, обратную MakeMovable, запрещающую перемещение окон элементов управления (листинг 1.23).

### Листинг 1.23. Запрещение перемещения и изменения размера

```
procedure MakeUnmovable(Handle: HWND);
var
style: LongInt;
flags: UINT;
begin
//Запрещаем перемещение элемента управления
style:= GetWindowLong(Handle, GWL_STYLE);
style:= style and not WS_OVERLAPPED and not WS_THICKFRAME
and not WS_CAPTION;
SetWindowLong(Handle, GWL_STYLE, style);
style:= GetWindowLong(Handle, GWL_EXSTYLE);
style:= style and not WS_EX_TOOLWINDOW;
SetWindowLong(Handle, GWL_EXSTYLE, style);
//Перерисуем в новом состоянии
flags:= SWP_NOMOVE or SWP_NOSIZE or SWP_DRAWFRAME or SWP_NOZORDER;
SetWindowPos(Handle, 0, 0, 0, 0, 0, flags);
end;
```

Осталось только реализовать вызовы процедур MakeMovable и MakeUnmovable в нужном месте программы. В рассматриваемом примере вызовы заключены внутри обработчика изменения состояния флажка на форме (листинг 1.24).

### Листинг 1.24. Управление перемещаемостью элементов управления

```
procedure TfrmMovingControls.chkSetMovableClick(Sender: TObject);
begin
if chkSetMovable.Checked then
begin
//Разрешаем перемещение элементов управления
MakeMovable(Memo1.Handle);
MakeMovable(ListBox1.Handle);
MakeMovable(Button1.Handle);
end
else
begin
//Запрещаем перемещение элементов управления
```

```
MakeUnmovable(Memo1.Handle);
MakeUnmovable(ListBox1.Handle);
MakeUnmovable(Button1.Handle);
end;
end;
```

# Масштабирование окон

Возможность масштабирования окон (форм) является интересным приемом, который может быть заложен в дизайн приложения. При этом имеется в виду масштабирование в буквальном смысле этого слова: как пропорциональное изменение размера элементов управления формы, так и изменение размера шрифта.

Использовать масштабирование при работе с Delphi крайне просто, ведь в класс TWinControl, от которого наследуются классы форм, встроены методы масштабирования. Вот некоторые из них:

• ScaleControls – пропорциональное изменение размера элементов управления на форме;

• ChangeScale – пропорциональное изменение размера элементов управления с изменением шрифта, который используется для отображения текста в них.

Оба приведенных метода принимают два целочисленных параметра: числитель и знаменатель нового масштаба формы. Пример задания параметров для методов масштабирования приведен в листинге 1.25.

### Листинг 1.25. Масштабирование формы с изменением шрифта

```
procedure TfrmScaleBy.cmbSmallerClick(Sender: TObject);
begin
ChangeScale(80, 100); //Уменьшение на 20 % (новый масштаб – 80 %)
end;
procedure TfrmScaleBy.cmbBiggerClick(Sender: TObject);
begin
ChangeScale(120, 100); //Увеличение на 20 % (новый масштаб – 120 %)
end;
```

Чтобы размер шрифта правильно устанавливался, для элементов управления нужно использовать шрифты семейства TrueType (в данном примере это шрифт Times New Roman).

На рис. 1.16 показан внешний вид формы до изменения масштаба.

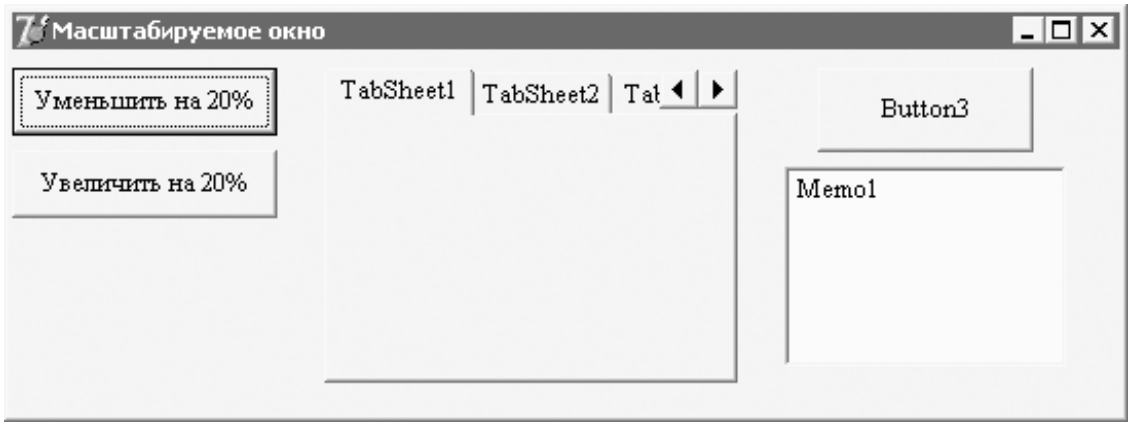



Рис. 1.16. Форма в оригинальном масштабе

Внешний вид формы после уменьшения масштаба в 1,25 раза (новый масштаб составляет 80 % от первоначального) показан на рис. 1.17.

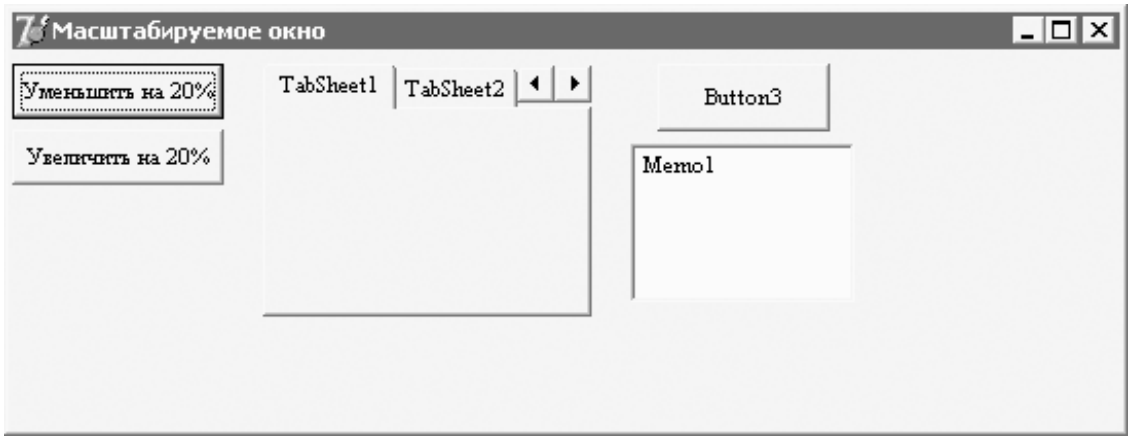



Рис. 1.17. Форма в масштабе 80 %

То, что форма не изменяет свой размер при масштабировании, можно легко исправить, установив, например, свойство AutoSize в значение True с помощью редактора свойств объектов (Object **Inspector).**

Если по каким-либо причинам использование свойства AutoSize вас не устраивает, то можно рассчитать новый размер формы самостоятельно. Только пересчитывать нужно не размер всего окна, а его клиентской области, ведь строка заголовка при масштабировании не изменяется. Расчет размера окна можно выполнить следующим образом.

1. Получить прямоугольник клиентской области окна (GetClientRect).
2. Вычислить новый размер клиентской области.
3. Рассчитать разницу между новой и первоначальной шириной, новой и первоначальной высотой клиентской области; сложить полученные значения с первоначальными размерами самой формы.

Пример расчета для увеличения размера клиентской области в 1,2 раза приведен ниже:

```
GetClientRect(Handle, rc);
newWidth:= (rc.Right – rc.Left) * 120 div 100;
newHeight:= (rc.Bottom – rc.Top) * 120 div 100;
Width:= Width + newWidth – (rc.Right – rc.Left);
Height:= Height + newHeight – (rc.Bottom – rc.Top);
```

**Примечание**

Чтобы после изменения масштаба формы можно было вернуться в точности к исходному масштабу (с помощью соответствующей обратной операции), нужно для уменьшения и увеличения использовать коэффициенты, произведение которых равно единице. Например, при уменьшении масштаба на 20 % (в 0,8 раз) его нужно увеличивать при обратной операции на 25 % (в 1/0,8 = 1,25 раза).

# Добавление команды в системное меню окна

Обратите внимание на меню, раскрывающееся при щелчке кнопкой мыши на значке окна. В этом системном меню обычно присутствуют пункты, выполняющие стандартные действия над окном, такие как закрытие, минимизация, максимизация и др. Для доступа к этому меню предусмотрены специальные функции, что дает возможность использовать его в своих целях.

Для получения дескриптора (HMENU) системного меню окна используется API-функция GetSystemMenu, а для добавления пункта в меню – функция AppendMenu. Пример процедуры, добавляющей пункты в системное меню, приведен в листинге 1.26.

### Листинг 1.26. Добавление пунктов в системное меню окна

```
procedure TForm1.FormCreate(Sender: TObject);
var hSysMenu: HMENU;
begin
hSysMenu:= GetSystemMenu(Handle, False);
AppendMenu(hSysMenu, MF_SEPARATOR, 0, '');
AppendMenu(hSysMenu, MF_STRING, 10001, 'Увеличить на 20%');
AppendMenu(hSysMenu, MF_STRING, 10002, 'Уменьшить на 20 %');
end;
```

В результате выполнения этого кода системное меню формы Form1 станет похожим на меню, показанное на рис. 1.18.

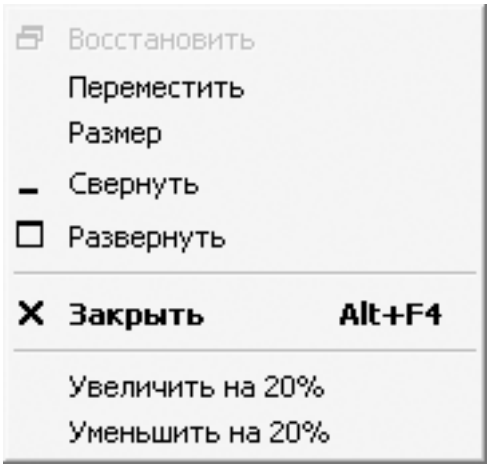



Рис. 1.18. Пользовательские команды в системном меню

Однако недостаточно просто создать команды меню – нужно предусмотреть обработку их выбора. Это делается в обработчике сообщения WM_SYSCOMMAND (листинг 1.27).

### Листинг 1.27. Обработка выбора пользовательских пунктов в системном меню

```
procedure TForm1.WMSysCommand(var Message: TWMSysCommand);
begin
if Message.CmdType = 10001 then
```

```
//Увеличение масштаба
ChangeScale(120, 100)
else if Message.CmdType = 10002 then
ChangeScale(80, 100)
else
//Обработка по умолчанию
DefWindowProc(Handle, Message.Msg, Message.CmdType, 65536 * Message.YPos+ Message.XPos);
end;
```

Обратите внимание на то, что числовые значения, переданные в функцию Append-Menu, используются для определения, какой именно пунктменю выбран. Чтобы меню работало стандартным образом, все поступающие от него команды должны быть обработаны. Поэтому для всех команд, реакция на которые не заложена в реализованном обработчике, вызывается обработчик по умолчанию (функция DefWindowProc).

# Отображение формы поверх других окон

Иногда вам может пригодиться возможность отображения формы поверх всех окон. За примером далеко ходить не надо: посмотрите на окно Диспетчера задач Windows. Теперь вспомните, терялось ли хоть раз окно Свойства: Экран среди других открытых окон. Это происходит благодаря тому, что это окно перекрывается другими окнами и при этом не имеют никакого значка на Панели задач (правда, это окно все же можно найти с помощью Диспетчера задач).

# Конец ознакомительного фрагмента.