



maXbox Starter 7

Start with Game Programming

1.1 Play a Game

Today we spend another time in programming with the internet (called social network) and multi user games. But in this primer I just introduce some basics without net and cloud. Hope you did already work with the Starter 1 to 6 available at:

<http://www.softwareschule.ch/maxbox.htm>

This lesson will introduce you to a simple game called Arcade like the famous Pong. The first popular "arcade games" were early amusement park midway games such as shooting galleries, ball toss games, and the earliest coin-operated machines, such as those that claim to tell a person one's fortune or played mechanical music.

Games today have become so complex that they require large numbers of developers, graphic artists, testers and managerial overhead to develop. Also the multi user capabilities improved. They rival large commercial enterprise application in their complexity and cost many millions of euros to develop and market. So the rest of this manual provides few technical details on developing games in a general-purpose manner.


Let's begin with the Application Structure Process in General of a Game:

1. Configure and open window
2. Initialize resources or state (background colour, light positions and texture maps).
3. Register input call-back functions
 - render, resize, repaint
 - Input: keyboard, mouse, joystick etc.
 - screen, avatar or data glove
4. Enter event processing loop (e.g. OnIdle() or timer objects)

The final ingredient of a game is performance and the idea. Most of us don't want to play a slow and boring game. But I still remember an adventure game on my Atari that took 10 minutes to render each scene. I still played it, because the game idea was great and if it is intelligent enough, no one mind the slow motion. Our game can be slow and has to be simple, because it serves as a demo to explain the important 4 points of the Application Structure Process above-mentioned.

1.2 Code it

As you already know the tool is split up into the toolbar across the top, the editor or code part in the centre and the output window at the bottom.

 Before this starter code will work you will need to download maXbox from the website. It can be down-loaded from <http://sourceforge.net/projects/maxbox> site. Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click `maxbox3.exe`

the box opens a default program. Test it with F9 or press **Compile** and you should hear a sound a browser will open. So far so good now we'll open the example:

```
183_playearth_first.pas
```

If you can't find the file use the link:

<http://www.softwareschule.ch/download/exampleedition.zip>

Or you use the `Save Page as...` function of your browser¹ and load it from `examples` (or wherever you stored it). One important thing: You need some bitmaps and other resources found in the directory `\earthplay` and by open the above zip you'll find it. But you must adapt the following `RESPATH` in relation to the executable. Now let's take a look at the code of this project. Our first line is

```
04 Program PlayEarth;
```

We have to name the program it's called `PlayEarth`.

```
06 const RESPATH = '\examples\earthplay';
07     BSIZE = 32;
08     ANGLE = 5;
09
10 var
11     Form1: TForm;
12     bgroundI, spriteI, paddleI: TImage;
13     //;
```

First in line 06 we have the above path of the resources, initially our sound and vision.

☞ In cross-platform applications (maXbox runs on Linux or Mac), you should replace any hard-coded pathnames with the correct pathname or set it relative for the system or use environment variables or at least define a single `const` to change.;

I mention this for Linux developers to change the backslash to slash. This ingredient called resources to a successful game is the set of graphics and sound. They need to be good enough to compliment the game idea and game play but not so resource intensive or flashy that they disturb from it. These local files you want to load for the first time are found in a directory relative to the exe. We need a background image called `maxearth.bmp` and two other bitmaps with one sound file for the first. In line 11 and 12 we declare our objects which contain the graphics.

☞ This example requires 4 objects of the classes: `TForm`, `TImage`, `TPicture` and `TBitmap` and the bitmap passes our images with the help of a picture to the screen. That needs an explanation: You use image controls to display graphical images on a form. A picture is used hold any graphic image. Use this to handle arbitrary files such as displaying images in an image control. And the bitmap is a powerful graphics object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk.

Creating copies of a bitmap is fast since the handle is copied, not the image. That's how we load any bitmap to an image object:

```
32 bgroundI.Picture.Bitmap.LoadFromFile(b_bitmap);
```

And that's how to replace a graphic with a new bitmap!:

```
//bgroundI.Picture.Graphic:= spriteI.Picture.bitmap;
```

This routine can be found in our function `InitResources` which is called by the `FormCreate`, so we jump now to the beginning of the app where all *****magic***** starts.

¹ Or copy & paste

The app generates first a form with the create method by passing self pointer.

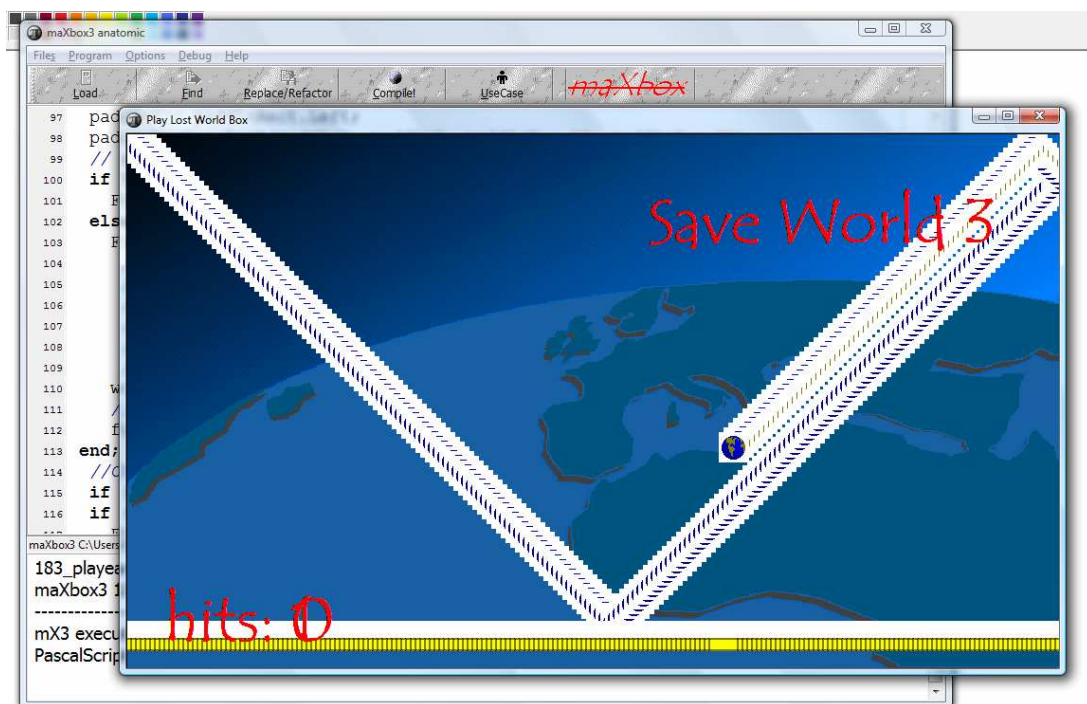
```
158 Form1 := TForm.Create(self);
```

So the object has to build first a window where the game runs up. TForm has some methods (in this case an event handler) named `onMouseMove`, `onKeyPress`, `onClose()` you can find in the `Forms.pas` unit or library. Instances of the TForm class are the building blocks of your application's user interface. The windows and dialog boxes in your application are all based on TForm.

As all win forms programming, even game coding, is event-based, understanding the principles of events and event handlers is critical. Although the event is triggered automatically, we need to create a special method called an event handler like `onMouseMove()` to be able to intercept the event and do something in response of it.

```
166 onMouseMove := @FormMouseMove
```

```
167 onKeyPress := @FormKeyPress;
```



1: The Gamer saves the World

Almost all the code you write is executed, directly or indirectly, in response to events.

👉 An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event--called an event handler--is a method (procedure). Now we have a win form that we can run, but it doesn't do anything. The form has no other controls on it such as buttons you can click or grids to display any information. In a regular form application we would now add such controls to the form to create our final application, but for our game we are going draw everything using the GDI API rather than the win form components. In line 71 ff you see an extract of the mouse move event handler. An `OnMouseMove` event occurs periodically when the user moves the mouse (what else). The event goes to the object that was under the mouse pointer when the user pressed a button. This allows you to give some intermediate feedback by drawing lines or in our case to move the paddle from left to right and vice versa.

```
71 paddleCenter := X;  
72 //paddle.Picture.Width = 32  
73 if (paddleCenter < BSIZE div 2) then //just first  
74   paddleCenter := BSIZE div 2;
```

In our interest is the parameter `x` in line 71 which sets the paddle (or a line) to the current position tracked by our mouse move. Next we come to the question how images live on the screen (animation and virtualisation is another topic). How graphic images appear in your application depends on the type of object whose canvas you draw on.

If you are drawing directly onto the canvas of a control, the picture is displayed immediately. However, if you draw on an off-screen image such as a `TBitmap` canvas, the image is not displayed until a control component copies from the bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its `OnPaint` message (VCL applications) or event (CLX applications). First we define a procedure called `FormActivate` which aims to the topic:

```
45 procedure FormActivate;
46 begin
47   //left/.top/.right(width)/.bottom(height)
48   bgroundRect:= Rect(0,0,Form1.ClientWidth,Form1.ClientHeight)
49   spriteRect:= Rect(0,0,BSIZE,BSIZE);
50   //WindowState:= wsMaximized;
51   Form1.Canvas.StretchDraw(bgroundRect, bgroundI.Picture.Bitmap);
52   Form1.Canvas.Draw(0,0, spriteI.Picture.bitmap);
```

So the one way that this can happen is to draw any graphic by passing through a method `Draw` of its own in the context of a canvas. That's the reason I made a few lines more at line 101 ff in case you'll need that for testing or research the code.

In line 51 we draw the background picture on the canvas. In most cases we use 3 methods:

- `CopyRect`: Copies part of an image from another canvas into the canvas.
- `Draw`: Renders the graphic object specified by the `Graphic` parameter on the canvas at the location given by the coordinates (X, Y).
- `StretchDraw`: Draws a graphic on the canvas so that the image fits in the specified rectangle. The graphic image may need to change its magnitude or aspect ratio to fit.

In `maXbox` there's also a paint box (`TPaintBox`) which allows your app to draw on a form. Write an `OnPaint` event handler to render an image directly on the paint box's canvas.

1.3 The Main Loop

Games are amazing different. As you know, smooth movement in games requires the screen to be updated many times per second. A "flicker fusion threshold" at which static images begin to fuse is generally taken to be 1/16 of a second, although it actually varies depending on illumination (brighter lights like computer monitors require higher frame rates) and where on the retina the image falls. Although movies are shown at 24 frames per second (FPS), 30 FPS is often considered the lowest-acceptable rate (render loop) for video games, and most action game players tune their graphics for no less than 60 FPS. Later on we learn how to compute the frame rate.

So, having a Pascal backend for a compiler brings a very strong advantage - portability and speed. But the box does have a very drawback - runtime speed mostly, because the byte code is executed line by line so we can't compete with a real compiler like `FreePascal` or `Delphi` but in most cases this does not compare with the advantages we get of a dynamic script language.

I believe, it's impossible to get something and not lose another thing.

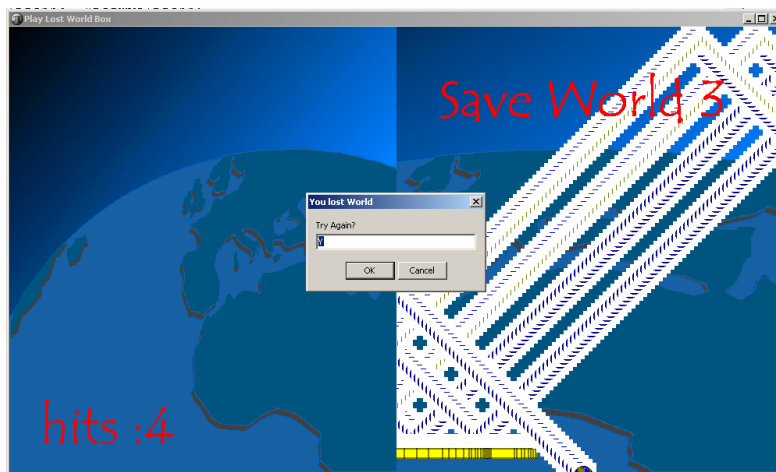
```
186 begin
187   if FormCreate then begin
188     FormActivate;
189     //Assign idle time function
190     Application.OnIdle:= @IdleLoop;
191   end;
```

Our main loop starts at line 186 and the so called render loop is line 190 (heart beat goes on). Because this render loop is called dozens of times per second and runs nonstop, game programming almost always uses the render loop as the “stopwatch or heartbeat” of the game, calculating everything inside the loop, not just graphics, but also physics, artificial intelligence, checking for user input, and of course scores and once again scores.

So how do we get the computer to run this loop? The form we added has an event called the `OnIdle` event. This event for a form object is called whenever the operating system has some spare time. Another well known event is the `OnPaint` event. This normally occurs only when you maximize a form or when a form is covered by another form that is moved.

At certain times, the operating system determines that objects onscreen need to refresh their appearance, so it generates `WM_PAINT` messages on Windows, which the VCL routes to `OnPaint` events. (In CLX applications, a paint event is generated, and routed to `OnPaint` events.) If you have written that event handler for that object, it is called when you use the `Refresh` method.

The default name generated for the `OnPaint` event handler in a form is `FormPaint`. You may want to use the refresh method at times to refresh a component or form. For example, you might call `refresh` in the form's `OnResize` event handler to redisplay any graphics or if using the VCL, you want to paint a background on a form.




2: With or without refresh screen

In our demo we don't use for instance this event; we handle all in the `OnIdle` loop. But like the paint event also the idle event (or in a certain manner the time event) depends on the operating system. On Linux for example the idle event didn't produce a valuable event so we can or must change the event producer to a timer or produce our own events to keep the game running. Now take a look at the picture above with the 2 screens of the game and you'll get the dependence between line 107 and 108!

```
107 worldRect:= Rect(0,0,Form1.clientWidth/2, Form1.clientHeight)
108 Form1.Canvas.StretchDraw(worldRect, bgroundI.Picture.Bitmap);
```

In line 107 the client width is divided by 2 into the result `worldRect`, so this region is only the left half of the window that will be refreshed by line 108 with the corresponding draw method! It also includes the update of the score panel on the left side. The player will receive a score that is based on the hits at which the world is saved.

So each refresh points to a canvas which is only available at runtime. So what's a canvas in the meaning of coding, more of this later on; first I want to explain what a canvas is.

 **Canvas** or the class `TCanvas` is a wrapper resource manager around the Windows device context, so you can also use all windows GDI functions on the canvas. The handle property of the canvas is called the device context.

The `TCanvas` object defined in the Graphics unit also protects you against common win graphics errors, such as restoring device contexts, pens, brushes, and so on to the value they had before the drawing operation. A canvas is used everywhere that drawing is required or possible, and makes drawing graphics both fail-safe and easy.

Let's say a few things about texting on canvas. For our example, `TextOut` writes to a canvas:

```
145 Form1.Canvas.TextOut(550, 50, 'Save World 3');
146 Form1.Canvas.TextOut(40, 470, 'hits: ' + IntToStr(hitcount));
```

`TextOut` simply writes a string on the canvas, starting at the point (X,Y), and then updates the `PenPos` to the end of the string. But how works transparent font on the canvas, her it is: The brush property of canvas named `style` is set on line 55:

```
36 with Form1.Canvas do begin
37   Font.Color:= clRed;
38   Brush.Style:= bsClear;
39   Font.Name:= 'Tempus Sans ITC';
40   Font.Size:= 50;
41 end;
```

The brush property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way. It allows you in a direct kind to visualize in two lines a bitmap from a file to the canvas:

```
Bitmap.LoadFromFile('MyBitmap.bmp');
Form1.Canvas.Brush.Bitmap:= Bitmap;
```

A brush's `Bitmap` property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas.



To draw graphics in an application, you draw on an object's canvas, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and it manages the device context for you, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or metafiles!



3: An ambient for the Game




Canvases are available only at runtime, so you do all your work with canvases by writing code. When working with graphics, you often encounter the terms drawing and painting:

- Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.
- Painting is the creation of the entire appearance of an object. Painting usually involves drawing. That is, in response to `OnPaint` events, an object generally draws some graphics. An edit box, for example, paints itself by drawing a rectangle and then drawing some text inside. A shape control, on the other hand, paints itself by drawing a single graphic.

By the way: If you use the `TImage` control to display a graphical image on a form, the painting and refreshing of the graphic contained in the image is handled automatically. The `Picture` property specifies the actual bitmap, drawing, or other graphic object that an image displays. You can also set the `Proportional` property to ensure that the image can be fully displayed in the image control without any distortion. Drawing on an image creates a persistent image. Consequently, you do not need to do anything to redraw the contained image.


In contrast, `TPaintBox`'s canvas maps directly onto the screen device (VCL) or the painter (CLX), so that anything drawn to the `PaintBox`'s canvas is transitory. This is true of nearly all controls, including the form itself. Therefore, if you draw or paint on a `TPaintBox` in its constructor, you will need to add that code to your `OnPaint` event handler in order for the image to be repainted each time the client area is invalidated.

 So far we have learned a lot about game coding, let's make a conclusion to the abstract structure process at the beginning of our demo:

- | | |
|--|---|
| 1. Configure and open window | → function <code>FormCreate</code> |
| 2. Initialize resources or state | → function <code>InitResources, FormActivate</code> |
| 3. Register input callback functions | → procedure <code>FormMouseMove</code> |
| 4. Enter event processing loop (e.g. <code>onIdle()</code>) | → procedure <code>IdleLoop()</code> |

```
function InitResources: boolean;
procedure FormActivate;
procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
procedure IdleLoop(Sender: TObject; var Done: Boolean);
function FormCreate: boolean;
```



 Try to change the text out parameters in line 145 to scale the form for full screen.

Some notes at last about the calculation of the frame rate:

```
FCurrentFrameTime:= Integer(GetTickCount);
FrameDiffTime:= FCurrentFrameTime - FLastFrameTime;
FLastFrameTime:= FCurrentFrameTime;
```

```
begin //in a loop
UpdateFrameTimeDiff;
if FrameDiffTime > 2 then
    Form1.Canvas.TextOut(40, 370, 'FPS: '+ IntToStr((1000 div FrameDiffTime)));
//Eg: 1000/50 = 20 FPS or 1000/500 = 2
```

max@kleiner.com

Links of `maxbox`, `OpenGL` and `Pascal/Delphi Games`:

<http://www.softwareschule.ch/maxbox.htm>

<http://sourceforge.net/projects/maxbox>

<http://sourceforge.net/apps/mediawiki/maxbox/>

http://www.softwareschule.ch/download/opengl_delphi_report.pdf

http://www.softwareschule.ch/download/terrain_delphi.zip

<http://www.delphigamer.com/main.php>

<http://www.pascalgamer.com/>