DirectX 3D In Delphi

by Andrew Kern and Noel Rice

The problem with most "minimal" examples of programming in DirectX 3D is that they are not minimal. When I want to learn a complex new technology, I want the smallest possible example that still runs. We just happen to have such an example here...

Though this article attempts to keep the code size to a bare minimum for an example of 3D graphics with Microsoft's Direct3D, you will still need to install Direct3D, and have access to at least one *.X file, and DirectX.pas. The simplest way to access to the first two pieces is to download the Game SDK from Microsoft's website, currently at:

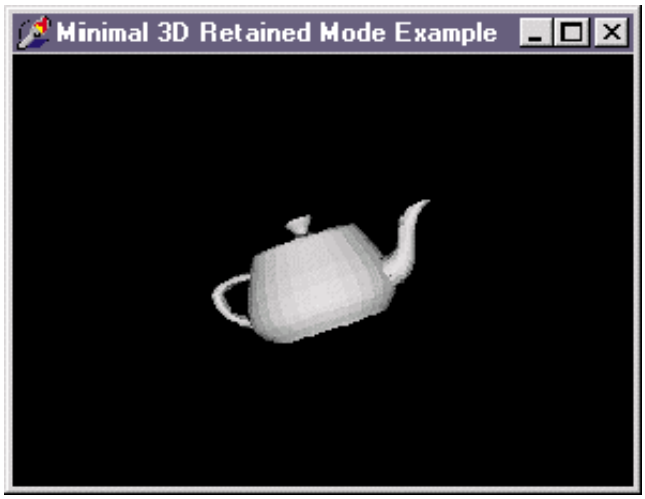
```
www.microsoft.com/directx/
  resources/downloads/dx5dl.htm
```

The final piece, DirectX.pas is on the accompanying disk. This Pascal file contains prototypes for all interfaces used in the article example. Special thanks to Blake Stone for all his work on the original conversion of Microsoft's header files to DirectX.pas and to Danny Thorpe for making the final interface conversions.

To program a minimal example that displays a rotating object, we can use the DirectX 3D "Retained Mode" that handles the gritty work of organizing buffers, matrix math, rendering and screen swapping. Retained Mode allows us to focus on Spielberg-esqe tasks such as placing visible objects, cameras and lights within a 3D space. Before getting too comfortable sitting in the director's chair and shouting "Action!" you must first create and initialize seven Direct3D Retained Mode COM interfaces that together perform the work of building and displaying moving 3D images. These interfaces are:

```
IDirect3DRM;
IDirect3DRMFrame;
IDirectDrawClipper;
IDirect3DRMDevice;
IDirect3DRMMeshBuilder;
IDirect3DRMLight;
IDirect3DRMViewPort;
```

A word on the order of variable declaration. The `IDirect3DXXX` variables represent interfaces managed by the VCL. In most cases we won't want or need to have direct control over interface creation or destruction. In other words, we shouldn't have occasion to call `AddRef` or `Release`. However, we need to be aware that Delphi *will* call interface `Release` methods and that this can cause problems if we're not aware of the destruction order. All interfaces will be destroyed in the reverse order that they are created. This holds true for interfaces declared in procedures as well as globally declared interfaces. To avoid grief you must not destroy interfaces that are still being used by other interfaces.



➤ *Figure 1*

Case in point: if `IDirect3DRMDevice` is declared after `IDirect3DRMViewPort`, then the device will be destroyed before the viewport. The function `CreateViewPort` instantiates the viewport interface based on a given device. If the device is yanked out from under the viewport bad things happen. Access violation city. So, the moral of the story is: declare interfaces in the same order that they are used. If access violations appear on termination of the application, try reordering your interfaces.

First use the `Direct3DRMCreate` function to create a `IDirect3DRM` interface. `IDirect3DRM` acts as a manager for the 3D environment and allows us to create other retained mode interfaces:

```
Var D3DRM: IDirect3DRM;
...
Direct3DRMCreate(D3DRM);
```

Next, *frames* are created to provide position, orientation and velocity for each entity in the 3D space. Frames in themselves are not visual, but can be thought of as places to hang visual objects. The `IDirect3DRM.CreateFrame` method takes two `IDirect3DRMFrame` parameters: a parent frame and the frame to be created. The first frame created in the example, "Scene," acts as a container for the entire 3D space. This frame has no parent, so we pass `Nil` as the first parameter to `CreateFrame`. The following three frame interfaces will provide position, velocity and orientation information for lights, camera and the visual object being displayed. `Scene` is the parent frame passed in the first parameter for the creation of lights, camera and `MyvisualObject` (Listing 1).

The next steps are necessary to make DirectX3D activity visible in our Delphi form. We'll use an `IDirectDrawClipper` interface to manage the visible area of our window, allow DirectX to know

what window we're drawing in and to create an output device. The first and last parameters to DirectDrawCreateClipper function below are currently reserved by Microsoft and unused. The SetHWnd function sets the window handle used to access clipping information. The first SetHWnd parameter is unused and should be zero. In the call to CreateDeviceFromClipper pass the clipper interface, allow the system to select a device by passing Nil in the second parameter, use the form's ClientWidth and ClientHeight for device dimensions, and finally pass the IDirect3DRMDevice to be created (Listing 2).

So far we have a 3D environment, locations for various objects and a mechanism to provide drawing capabilities, but we still need a visual object to look at, lights to illuminate the object and a camera to see the object with.

To create visual 3D objects, DirectX3D uses the concept of a *mesh*, which is a collection of polygonal faces represented by a IDirect3DRMMesh interface. We can also use a IDirect3DRMMeshBuilder interface to create an implicit mesh interface and to load the mesh from an existing *.x file containing data describing the object's faces. Once the mesh is loaded we have a visual object with a location that defaults to the origin. Place the mesh into the scene by calling AddVisual to attach the visual object to a frame. An effective 3D demo requires motion. The SetRotation function causes the MyVisualObject frame to spin in relation to the scene frame along horizontal, vertical or depth axis, at a speed given by the last parameter (Listing 3).

We now have an visual object in the scene, but we still can't see it because it's dark. We need to create a IDirect3DRMLight and attach it to a frame to illuminate the visual object. Though there are five different varieties of lights, I suggest using D3DRMLIGHT_DIRECTIONAL for most reliable results. D3DRMLIGHT_DIRECTIONAL simulates a faraway light source like the sun, and as such is not as sensitive to position as some of the other settings (Listing 4).

```
Var
  Scene, Camera, Lights, MyVisualObject: IDirect3DRMFrame;
...
D3DRM.CreateFrame(Nil, Scene);
D3DRM.CreateFrame(Scene, Lights);
D3DRM.CreateFrame(Scene, Camera);
D3DRM.CreateFrame(Scene, MyVisualObject);
```

➤ *Listing 1*

```
Var
  DDClipper: IDirectDrawClipper;
  Dev: IDirect3DRMDevice;
...
DirectDrawCreateClipper(0, DDClipper, Nil);
DDClipper.SetHWnd(0, Handle);
D3DRM.CreateDeviceFromClipper(DDClipper, Nil, ClientWidth, ClientHeight, Dev);
```

➤ *Listing 2*

```
Var
 MeshBuilder: IDirect3DRMMeshBuilder;
 MyVisualObject: IDirect3DRMFrame;
...
D3DRM.CreateMeshBuilder(MeshBuilder);
MeshBuilder.Load(PChar('egg.x'), Nil, D3DRMLOAD_FROMFILE, Nil, Nil);
MyVisualObject.AddVisual(IDirect3DRMVisual(MeshBuilder));
MyVisualObject.SetRotation(Scene, 1, 1, 1, 0.01);
```

➤ *Listing 3*

```
Var Light1: IDirect3DRMLight;
...
D3DRM.CreateLightRGB(D3DRMLIGHT_DIRECTIONAL, 1.0, 1.0, 1.0, Light1);
Lights.AddLight(Light1);
```

➤ *Listing 4*

```
Var View: Idirect3DRMViewPort;
...
D3DRM.CreateViewPort(Dev, Camera, 0, 0, ClientWidth, ClientHeight, View);
```

➤ *Listing 5*

The point of view of the user looking at a visual 3D object is represented by a viewport. 3D objects are rendered to a 2D rectangular area of a device. This process is handled by the IDirect3DRMViewPort interface (Listing 5).

Since all objects have locations that default to the origin, we will want to move the camera back a few paces so that our camera and object are not crowded together in the same space. The light does not need to be repositioned, because we're only interested in the light as a vector that represents the direction it is pointing, not it's actual location. Since we are trying to keep this example simple, we don't bother to reposition it. We only need to move the camera back in order to view the visible object:

```
Camera.SetPosition(Scene,0,0,-10);
```

As the director of this process you have created the 3D environment complete with lights, camera and visible objects. The call to "Action!" takes the form of the Tick function. IDirect3DRM.Tick can be placed in a TTimer's OnTimer event and will update the position of frames in the scene and render the new results to the output device. The single parameter taken by Tick applies to rotations and velocities by all frames in the scene. The larger the number, the faster the apparent speed of object movement in the scene.

The DirectX 3D engine is susceptible to floating-point processor stack overflow errors under Windows 95, but seems to work fine under NT4 with Service Pack 3. For those of us on 95, the try..except block will trap for these errors and ignore them:

```
try
  D3DRM.Tick(1.0);
except
  on EInvalidOp do
    ; //ignore floating point overflow
end;
```

Listing 6 shows a no-frills working program in its most minimal form. This does not show any of the niceties, like error trapping or resizing of windows. We won't get into window resizing, but to make the minimal program a bit more robust we can add helper functions to examine and react to DirectX return codes. Add DDUTILS.PAS to your uses clause (Listing 7) and wrap each function call with a xxxCheck function. You can use the D3DRMCheck function for methods of retained mode interfaces:

```
D3DRMCheck(
  Direct3DRMCreate(D3DRM));
```

and DDCheck for the IDirectDrawClipper interface methods:

```
DDCheck(DirectDrawCreateClipper
  (0, DDClipper, Nil));
DDCheck(DDClipper.SetHWnd(
  0, Handle));
```

DirectX3D retained mode allows us to jump over the low-level tedium of having to build our own 3D engine while Delphi 3 simplifies the process of working with these COM interfaces.

The combination of DirectX3D and Delphi 3 makes the formidable task of 3D display a relatively simple matter of about eighteen lines of code. From here you can dig into DirectX.pas; it's filled with possibilities for adding textures, creating new mesh objects, and lighting effects.

Andrew Kern (email akern@corp.borland.com and Noel Rice (email nrice@corp.borland.com) both work for Borland in the USA.

➤ *Listing 7*

```
unit DDUtils;
interface
uses Windows, Classes, SysUtils, DirectX, ComObj;
type
  EDDError = Exception;
  ED3DRMError = Exception;
procedure DDCheck(Result: HResult);
procedure D3DRMCheck(Result: HResult);
implementation
procedure D3DRMError(ErrorCode: HResult);
var ErrMsg: String;
begin
  case ErrorCode of
    D3DRMERR_BADALLOC:        ErrMsg := 'Out of memory.';
    D3DRMERR_BADDEVICE:       ErrMsg :=
      'Device is not compatible with renderer.';
    D3DRMERR_BADFILE:         ErrMsg := 'Data file is corrupt.';
    D3DRMERR_BADMAJORVERSION: ErrMsg := 'Bad DLL major version.';
    D3DRMERR_BADMINORVERSION: ErrMsg := 'Bad DLL minor version.';
    D3DRMERR_BADOBJECT:       ErrMsg := 'Object expected in argument.';
    D3DRMERR_BADTYPE:         ErrMsg := 'Bad argument type passed.';
    D3DRMERR_BADVALUE:        ErrMsg := 'Bad argument value passed.';
    D3DRMERR_FACEUSED:        ErrMsg := 'Face already used in a mesh.';
    D3DRMERR_FILENOTFOUND:    ErrMsg := 'File cannot be opened.';
    D3DRMERR_NOTDONEYET:      ErrMsg := 'Unimplemented.';
    D3DRMERR_NOTFOUND:        ErrMsg := 'Object not found in specified place.';
    D3DRMERR_UNABLETOEXECUTE: ErrMsg := 'Unable to carry out procedure.';
  end;
  ErrMsg := 'DirectDraw 3D Retained Mode error: ' + ErrMsg;
  raise ED3DRMError.Create(ErrMsg);
end;
procedure D3DRMCheck(Result: HResult);
begin
  if Result <> D3DRM_OK then D3DRMError(Result);
end;
procedure DDError(ErrorCode: HResult);
begin
  raise EDDError.Create('DirectDraw error');
end;
procedure DDCheck(Result: HResult);
begin
  if Result <> DD_OK then DDError(Result);
end;
end.
```

➤ *Listing 6*

```
unit MinDirX1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DirectX, StdCtrls, ExtCtrls, ComObj;
type
  TfrmMain = class(TForm)
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  public
    D3DRM: IDirect3DRM;
    Scene, Camera, Lights, MyVisualObject: IDirect3DRMFrame;
    MeshBuilder: IDirect3DRMMeshBuilder;
    Light1: IDirect3DRMLight;
    View: IDirect3DRMViewPort;
    DDClipper: IDirectDrawClipper;
  end;
var frmMain: TfrmMain;
    Dev: IDirect3DRMDevice;
implementation
{$R *.DFM}
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  { Create the IDirect3DRM interface from which other
    interfaces are created and managed }
  Direct3DRMCreate(D3DRM);
  { Create frames to provide location and velocity for
    each visual element in the 3D landscape }
  D3DRM.CreateFrame(Nil, Scene);
  D3DRM.CreateFrame(Scene, Lights);
  D3DRM.CreateFrame(Scene, Camera);
  D3DRM.CreateFrame(Scene, MyVisualObject);
  { Next steps create an IDirect3DRMDevice to keep track of
    hard/software capabilities & window area being drawn to}
  DirectDrawCreateClipper(0, DDClipper, Nil);
  DDClipper.SetHWnd(0, Handle);
  D3DRM.CreateDeviceFromClipper(DDClipper, Nil, ClientWidth,
   ClientHeight, Dev);
  { Create 3D visible object, add to scene and give it a spin }
  D3DRM.CreateMeshBuilder(MeshBuilder);
  MeshBuilder.Load(PChar('Egg.x'), Nil, D3DRMLOAD_FROMFILE,
    Nil, Nil);
  MyVisualObject.AddVisual(IDirect3DRMVisual(MeshBuilder));
  MyVisualObject.SetRotation(Scene, 1, 1, 1, 0.01);
  { Add a light and attach to a frame. Forgetting this makes
    for a fully functioning but extremely boring display }
  D3DRM.CreateLightRGB(D3DRMLIGHT_DIRECTIONAL, 0.03, 0.9,
    0.9, Light1);
  Lights.AddLight(Light1);
  { viewport defines how 3D scene is rendered to 2D window.
    Position & orientation of Camera define user point-of-view }
  D3DRM.CreateViewPort(Dev, Camera, 0, 0, ClientWidth,
    ClientHeight, View);
  Camera.SetPosition(Scene, 0, 0, -7);
  Timer1.Interval := 1;
end;
procedure TfrmMain.Timer1Timer(Sender: TObject);
begin
  D3DRM.Tick(1.0);
  { The Tick method actually encapsulates four other tasks:
    IDirect3DRMFrame.Move: Reposition frames according to
      rotation and velocity.
    IDirect3DRMViewport.Clear: Clear the viewport to the
      current background color.
    IDirect3DRMViewport.Render: Draw 3D scene to memory.
    IDirect3DRMDevice.Update: Copy rendered scene to window}
end;
end.
```