

# DirectXGraphics For Visual Basic

Jack Hoxley

## **Introduction**

This document is a compilation of all 3 parts of the original series written for the website [www.GameDev.net](http://www.GameDev.net) . For ease of reading/use I've now compiled them into a single pdf document.

The article text remains the same as that originally posted, but with a few slight changes for known errata and some tidying up so they fit together better in one long document.

All content was written by myself, Jack Hoxley, and you can email me at [Jack.Hoxley@DirectX4VB.com](mailto:Jack.Hoxley@DirectX4VB.com) or visit my website at [www.DirectX4VB.com](http://www.DirectX4VB.com)

A constantly updating, a far more detailed set of tutorials is available at my website [www.DirectX4VB.com](http://www.DirectX4VB.com) (go to DirectX8->DirectXGraphics). This article is intended as an all-in-one guide to getting started, it is by no-means a complete guide to using Direct3D8.

*Enjoy!*

## **Table Of Contents:**

<b><u>Part 1</u></b>	<b>3</b>
<a href="#">Getting Started</a>	4
<a href="#">A Simple Application</a>	5
<a href="#">The Main Loop</a>	10
<a href="#">Render()</a>	12
<a href="#">Drawing Something - The Theory</a>	13
<a href="#">Drawing Something - The Practical Part</a>	16
<a href="#">Overview</a>	20
<b><u>Part 2</u></b>	<b>21</b>
<a href="#">Reconfiguring Our Direct3D Application</a>	22
<a href="#">Getting Started With Basic 3D Geometry</a>	29
<a href="#">Matrices And Transformations</a>	33
<a href="#">Vertex Buffers And Index Buffers</a>	40
<a href="#">Summary</a>	48
<b><u>Part 3</u></b>	<b>49</b>
<a href="#">Using Textures In Direct3D</a>	50
<a href="#">Loading 3D Models in to Direct3D</a>	57
<a href="#">Using Direct3D Lighting</a>	61
<a href="#">Conclusion</a>	69
<b><u>Recommended Reading and Links</u></b>	<b>70</b>
<b><u>Disclaimer and Copyright</u></b>	<b>71</b>

# DirectXGraphics For Visual Basic Part 1

Welcome to the first of a 3 part mini-series on the usage of DirectXGraphics (The graphical component of DirectX8), over these 3 articles I will cover everything that you need to know to be a competent programmer in this area. Whilst this series will not cover absolutely every aspect (that would require many more than 3 parts), by the end you will be able to do most things, and anything you cant do you should be able to work out for yourself, or read other tutorials and easily understand them.

Some people may well say that you cant write a proper 3D game in visual basic, I'm not here to argue about that, but I will tell you now – it is perfectly possible to write a moderate to advanced game in full 3D using pure visual basic, maybe not the next quake/half-life, but that doesn't mean you cant do any games.

In order to program with DirectX you are going to need a few things:

1. A general knowledge of the visual basic language, whilst complicated things will be explained I will assume that you can write a reasonably complex program.
2. A copy of Visual Basic 5 or later, earlier versions of visual basic do not support the Component Object Model (COM), and therefore will not be able to use DirectX8. The source code presented here is from VB6 – there may be a few compatibility issues with VB5, but these should be fixed fairly easily.
3. A copy of DirectX8, the runtimes are perfectly acceptable (the ones that you get from the Microsoft site or magazine CD's). If you're serious about learning and using DirectX getting the SDK (Software Development Kit) will be a huge advantage.

Whilst these articles are going to be in visual basic, the actual DirectX8 interfaces are almost identical to those used in C/C++ (except for the obvious language differences), so if you can use DirectX8 in C/C++ then you'll find this very easy...

## **Getting Started**

DirectXGraphics all come under the name of Direct3D, which will be the term used from now on (it's shorter), but the names are interchangeable. Direct3D when it gets hard gets very, very hard indeed; but luckily the basics are very simple, and a basic application can be set up in a 100 or so lines of code. So here goes:

First we need to attach DirectX8 to our VB program – so it knows how to use it; open up VB and create a new "Standard EXE" project. A single form should be added to the project view. Go to the *Project* menu, then click on *references* to display the library dialog. You should see a long list of objects and libraries that in the middle of the window – all of them with a small checkbox to the left. Scroll down until you see an entry called "DirectX 8 for Visual Basic Type Library", select the check box and click "Ok".

We have now referenced our project to the DirectX 8 runtime library; that is all we need to do in order to use DirectX 8 features in visual basic. Bare in mind that the end user will have to have DirectX 8 installed on their computer for your application to even begin execution – if it's not there your program will terminate as soon as it's started. I have a template set up for this type of application, so it appears in my "New Project" dialog box – something you may wish to do.

## A simple application

Now that we can use DirectX8 we're going to set up a very simple example – all it will do is create an instance of Direct3D and clear the screen, then terminate.

The first thing we need to do is put some variables into our (Declarations) section of the form:

```
Dim Dx As DirectX8 'The master Object, everything comes from here
Dim D3D As Direct3D8 'This controls all things 3D
Dim D3DDevice As Direct3DDevice8 'This actually represents the hardware doing the rendering
Dim bRunning As Boolean 'Controls whether the program is running or not...
```

The first 3 variables here (Dx, D3D, D3DDevice) are all classes – we'll need to initialise them and terminate them; the fourth variable, bRunning, is just a simple Yes/No flag that states if the application is running or not – more on that one later.

Now seems like a good time to explain what these different objects do. DirectX8 has a hierarchy of objects and interfaces, each one with a parent, and in this case a "DirectX8" object is as far back as they go. The "Direct3D8" object deals with creating devices and enumerating their capabilities. Finally the "Direct3DDevice8" object, this represents your 3D card – you tell it to do things and (within reason) it'll do it. We therefore create a "DirectX8" object, this then helps us create a "Direct3D8" object, which in turn will setup a "Direct3DDevice8" object for us to use.

There are several other interfaces/objects that we can create, but right now we don't really need to know much about them – you'll see them as we go. It is very useful to have a copy of the DirectX8 SDK help file when dealing with these objects, whilst VB's intellisense and object browser are very useful, the SDK help file explains and lists all the functions and features of each interface/object.

Now that we have the variables defined we can start to do something with them; for this we're going to create a function called "Initialise()" which does exactly what it says it will – when it's finished execution (and assuming no errors) we'll be able to use all the objects and start making things appear on screen.

```
Public Function Initialise() As Boolean
On Error GoTo ErrHandler:

    Initialise = True '//We succeeded
    Exit Function

ErrHandler:
    Debug.Print "Error Number Returned: " & Err.Number, Err.Description
    Initialise = False
End Function
```

Above is the basic framework for the function – and you'll be seeing that most of the functions are designed like this. Technically Initialise() does not need to be a function as it doesn't return any particular data. But I particularly like this layout because it allows me to design a good function that should never bring down the rest of the

application – if it fails all it will do is return false to whoever called it. When calling this function we should use code like this:

```
If Not (Initialise() = True) Then GoTo Error_Handler:
```

Which will execute the initialisation code, then if it succeeds it will carry on as normal, but if it fails it will go to the "Error\_Handler" for processing / correction. The above call is just a simplified (and easier to read) version of:

```
If Initialise() = False Then  
    GoTo Error_Handler:  
End If
```

Now that we've got the basic function structure laid out we'll put something in it. The first thing we need to do is define two structures and initialise the Direct3D objects:

```
Dim DispMode As D3DDISPLAYMODE '//Describes our Display Mode  
Dim D3DWindow As D3DPRESENT_PARAMETERS '//Describes our Viewport  
  
Set Dx = New DirectX8 '//Create our Master Object  
Set D3D = Dx.Direct3DCreate() '//Let our Master Object create the Direct3D Interface
```

The two structures are used in a minute to help create the final Direct3DDevice8 object, but right now all we do is define them. Next we create the DirectX8 object – you may well know that it would have been perfectly legal to have defined the object like "Dim Dx as New DirectX8", but this is bad for what we want to do. The method just mentioned is known as Early Binding, the method we're using is called Late Binding; the difference being that if you late bind it visual basic will not check if it's created when you try and use it (but will return errors), if you early bind it VB will compile the code with a statement around EVERY call to the object along the lines of "If the object is nothing, create it". Whilst that may well only be true the first time around, it's still something extra for the computer to think about, and being a game we want all the speed we can get, and these objects will be used 1000's of times a second – so you can imagine the sort of speed we'll be wasting. Secondly we make our master interface create the generic Direct3D interface. You will always be able to create a Direct3D interface – no matter what the hardware installed can do. Now we need to fill out the two structures we just defined:

```
D3D.GetAdapterDisplayMode D3DADAPTER_DEFAULT, DispMode '//Retrieve the current display Mode  
  
D3DWindow.Windowed = 1 '//Tell it we're using Windowed Mode  
D3DWindow.SwapEffect = D3DSWAPEFFECT_COPY_VSYNC '//We'll refresh when the monitor does  
D3DWindow.BackBufferFormat = DispMode.Format '//We'll use the format we just retrieved...
```

Not too complicated really – but it gets more complicated if we want to use fullscreen rendering (covered later on). Whilst it really won't matter for this sample, if you are using windowed mode it's a good idea to keep your window fairly small – 400x300 in

pixels is a good size for most resolutions. The next part actually involves creating an instance of a Direct3D device – this part can be slightly dangerous – if you send parameters that the end-users computer cant handle then it'll fail and cause an error. This mostly tends to happen when you're using fullscreen modes and you need to choose a resolution/colour depth that suits their hardware/monitor.

```
Set D3DDevice = D3D.CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, _  
    frmMain.hWnd, _  
    D3DCREATE_SOFTWARE_VERTEXPROCESSING, _  
    D3DWindow)
```

The actual code isn't too complicated, but it's what goes in the parameters that is. The parameters are as follows:

**Adapter As Long** : Whilst they don't seem to be making 3D cards with primary/secondary devices this is where you can change it. Almost all graphics cards will be on the default adapter (D3DADAPTER\_DEFAULT). Set it to 0 or 1 otherwise

**DeviceType As Const\_D3DDEVTYPE** : What type of device we want to use; there are 2 main types of device and an optional 3<sup>rd</sup>:

*D3DDEVTYPE\_HAL* - Hardware acceleration, where the actual 3D card does the rendering

*D3DDEVTYPE\_REF* - A reference device, purely for developers – you'll be lucky if you get more than 0.25 frames a second out of this. On the other hand you can do absolutely anything with it – full feature support.

*D3DDEVTYPE\_SW* - This cant be used unless you register a software renderer (a plugin for DirectX), but there aren't any bundled with Direct3D, so you'll have to make your own (very hard) or get a 3<sup>rd</sup> party one.

**hFocusWindow As Long** : This lets Direct3D know which window it needs to render to, mostly for windowed mode, so it can check if it's gone behind other windows or it's been closed and so on... always pass <FormName>.hWnd here, but make sure that the window is visible first.

**BehaviourFlags As Long** : How this device will behave, and what does what (processor and/or 3D card). This should be D3DCREATE\_SOFTWARE\_VERTEXPROCESSING on most computers – or computers where there is no hardware transform and lighting or better (almost every card except the GeForce cards); in which case you can put in D3DCREATE\_HARDWARE\_VERTEXPROCESSING, which will force the 3D card to do transform and lighting operations; alternatively you can use the D3DCREATE\_PUREDEVICE option – which is new to Direct3D8, and is only available on the £300+ GeForce2 Ultra chipsets (at time of writing anyway).

**PresentationParameters As D3DPRESENT\_PARAMETERS** : Just place the structure that we filled earlier in here...

That's our initialisation code complete – assuming that code runs through successfully then we'll have a fully initialised device attached to our form ready to play with. One word about DirectX errors – they always have the description "Automation Error" and a number in the negative 2 millions (-2001230 for example). Should you want to know what that means in english you'll need to check the Err.Number against a set of constants that the DirectX8 library provides us with. If you have the SDK you can check what error numbers each function might return and only check those, otherwise you'll need to check them all – and there are a lot of them! Look for them in the object browser, they all tend to begin with "D3DERR\_" or "E\_"...

One final thing that I want to cover before we move on further is the topic of enumeration; you may not have heard of this before – but it’s something you’ll become familiar with if you spend any length of time programming in DirectX. Enumeration is the process of analysing the hardware to see what it’s capable of (or not capable of). You’ll meet most of it as we go along – but there are a couple that are relevant to device creation that I need to cover here.

In the above initialisation code we specified D3DDEVTYPE\_HAL, which may or may not be available on the host computer, and if we want to jump to fullscreen mode we’ll need to know what resolutions and colour depths the hardware supports as well. Whilst software vertex processing works on all computers it would be nice to take advantage of any additional hardware features available. To do this we use the following code:

```
Dim DevCaps As D3DCAPS8
On Local Error Resume Next
    D3D.GetDeviceCaps D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, DevCaps
If Err.Number = D3DERR_INVALIDDEVICE Then
    'We couldn't get data from the hardware device - probably doesn't exist...
    D3D.GetDeviceCaps D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, DevCaps
    Err.Clear '//Remove the error value..
End If

'//For Hardware vertex processing:
If (DevCaps.DevCaps And D3DDEVCAPS_HWTRANSFORMANDLIGHT) Then
    Debug.Print "Hardware Transform and lighting supported"
Else
    Debug.Print "Hardware Transform and lighting is not supported"
End If

'//For Pure Device processing:
If (DevCaps.DevCaps And D3DDEVCAPS_PUREDEVICE) Then
    Debug.Print "Pure Device is supported."
Else
    Debug.Print "Pure device is not supported"
End If

'//To check the rest we use:
If D3D.CheckDeviceType(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, _
    DispMode.Format, DispMode.Format, 1) = D3D_OK Then
    Debug.Print "The selected device format is acceptable"
Else
    Debug.Print "The selected device format is not acceptable"
End If
```

At the moment the above code does nothing more than tell you, the developer, what the hardware can do – or can’t do. The first two lines retrieve all the enumeration data for the device we’ve specified; there’s a simple error handler in here that should stop us crashing if there is no hardware device present – if this is the case then the program gets data from the reference device (which will support almost everything). The D3DCAPS8 structure now holds all the information that we need to evaluate the device and it’s capabilities. If you know that your application requires other features you can enumerate them now and find out if there’s any point in using this device.

So we now have our structure filled with data, all we need to do now is extract the information we want. At this point it’s a good idea to use the SDK help file – there

are literally thousands of different flags and features we can check for, and the help file lists every one of them with a small description (which isn't always that helpful). If you are just going to be reading tutorials off the net, or not poking around much yourself then it's not too important – most tutorials will explain what enumerations you'll need to perform.

In the above example we first check for hardware transform and lighting capabilities, the transform part is to do with vertex manipulation, and if it's done in hardware then it would imply that we can set up a device that uses the `D3DCREATE_HARDWARE_VERTEXPROCESSING` flag. We then check for the presence of the pure device option, which is the next step up from hardware transform and lighting (therefore use this if it's present). If this returns true then we can specify `D3DCREATE_PUREDEVICE` when creating the device. On the other hand if both of these return false we'll just have to use `D3DCREATE_SOFTWARE_VERTEXPROCESSING`. Lastly we check if the device type can be created – we specify the type of device, the format and if it's in windowed mode or not; if this call evaluates to `D3D_OK` then we'll be able to create a device with the same parameters, if it doesn't then we need to find some other parameters – this will usually just mean changing to the Reference device – as the call we made earlier to get the current display mode will of told us the correct format, and if we're using fullscreen modes then we'll have enumerated the possibilities (more on that later) – which only leaves the possibility that there's no hardware device present. We could avoid this completely and just remember what happened when we got the enumeration data – and which device that came from.

## The Main Loop

This next part is fairly quick and simple, yet is important to any Direct3D application – how the thing runs. Anyone who has paid any attention to a commercial game will know about frame rates – how many times every second the computer is updating the game; high frame rate is good, low frame rate is bad.

We now need to set up our application so that it runs on a loop, event based (where we only update when something has changed) simply will not cut it here – you'll be wasting valuable time either doing nothing or trying to work out if something has changed; and on top of that it's almost always going to be changing. Secondly never ever, ever, ever use a timer control or similar for this job – they are inaccurate and slow, you may well be able to set them to 1ms, but in reality they're only accurate to about 50-100ms (maximum frame rate is therefore 10-20fps).

We're going to use a loop instead – in theory a never ending loop. This loop will execute as fast as possible, and will form the basis of our frame rate – the faster the loop goes the higher the frame rate. This loop will use a simple Boolean flag to determine if it's running, as soon as this variable goes false the loop terminates and we do something else (probably close the application down).

Here's the code that the sample program uses:

```
Private Sub Form_Load()
    Me.Show '//Make sure our window is visible

    bRunning = Initialise()
    Debug.Print "Device Creation Return Code : ", bRunning 'So you can see what happens...

    Do While bRunning
        Render '//Update the frame...
        DoEvents '//Allow windows time to think; otherwise you'll get into a really tight (and bad) loop...

    Loop '//Begin the next frame...

    '//If we've gotten to this point the loop must have been terminated
    ' So we need to clean up after ourselves. This isn't essential, but it's
    ' good coding practise.

    On Error Resume Next 'If the objects were never created;
    ' (the initialisation failed) we might get an
    ' error when freeing them... which we need to
    ' handle, but as we're closing anyway...

    Set D3DDevice = Nothing
    Set D3D = Nothing
    Set Dx = Nothing
    Debug.Print "All Objects Destroyed"

    '//Final termination:
    Unload Me
End
End Sub
```

As you can see straight away this code is all in the Form\_Load event, which isn't the optimal place to put it. Whenever I design a bigger project I always put the control loop as a Sub Main() in a separate control module and leave the form completely

empty – and then all the subsequent code goes in classes (one for graphics, utilities, maths, audio, physics, AI, Input, File handling and so on).

The first step is to make sure the form is visible, normally this wont happen until this procedure has finished (which in our case wont happen till the program terminates), as already mentioned, Direct3D wont function properly if the form isn't visible or isn't loaded.

Next we initialise Direct3D, we place it's return value in the Boolean on/off switch (instead of the original method I showed you); the beauty of this is that if it returns false on the first pass of the loop it'll terminate itself.

In the middle we have the main loop, a Do While ... Loop structure. At the moment it's made up of two statements, these will be the only two statements executed for the vast majority of runtime. Place any additional calls or statements that you want processed on a frame by frame basis. The key part here is to have a DoEvents call at the end of the loop, without it your program will go down the pan very quickly – and in most cases lock up the system. If this statement is not in here we wont receive any messages, no input (keyboard, mouse) and the chances are that pure VB language statements will not be executed properly. The DoEvents yields time for the system (windows In this case) to think about things and do whatever it sees fit – if other programs are running then they'll have their time now, and if you've asked windows to do anything it'll probably happen now.

Lastly we have the termination code, as noted in the extract this is not necessarily required – the DirectX library will see that objects are destroyed safely, but you can never be sure from computer to computer – so it's best to do it for yourself. As with all other COM based interfaces destroy them in the reverse order from which they were created.

## Render()

You should have noticed in the main loop code above that there was a call to the Render() function on every pass of the main loop. It's this code that actually presents the graphics on the screen – and processes anything relevant to how the graphics are displayed.

For this sample this code isn't going to do anything greatly exciting – this is our first DirectXGraphics application after all. The next couple of articles will explain the more interesting parts...

```
Public Sub Render()  
    '//1. We need to clear the render device before we can draw anything  
    '    This must always happen before you start rendering stuff..  
    D3DDevice.Clear 0, ByVal 0, D3DCLEAR_TARGET, &HCCCCFF, 1#, 0  
    'the hexadecimal value in the middle is the same as when you're using colours in HTML - if you're familiar  
    'with that.  
  
    '//2. Next we would render everything. This example doesn't do this, but if it did it'd look something  
    '    like this:  
  
    D3DDevice.BeginScene  
    'All rendering calls go between these two lines  
    D3DDevice.EndScene  
  
    '//3. Update the frame to the screen..  
    '    This is the same as the Primary.Flip method as used in DirectX 7  
    '    These values below should work for almost all cases..  
    D3DDevice.Present ByVal 0, ByVal 0, 0, ByVal 0  
End Sub
```

Not too complicated, but trust me, it gets bigger and bigger as we start adding new stuff. The render procedure always follows the same pattern – Clear, Draw, Display on screen. The clear part removes whatever was left in the frame buffers, then we draw everything – all our triangles, models and whatever else we fancy. Finally we update the screen – when this call is finished whatever we just drew will appear on the monitor.

As you learn more about DirectXGraphics, and as these articles go on this function will be adapted and altered quite dramatically – it can get very big and quite complicated. One thing to always bear in mind about this procedure (and any others that you put in the main loop) is that they have to be fine tuned and as smooth as possible – any untidy or slow code will have a massive impact on the overall speed of your application. If each call takes 6ms to process, but something you do makes it take 10ms instead your frame rate will drop from 167fps down to 100fps – it's still pretty fast, but assuming you have other things to do (AI, Audio, Physics and general gameplay) then this drop will be more significant.

## **Drawing Something – The theory**

Okay, so you've learnt how to initialise a Direct3D application in visual basic – wow. Hardly cutting edge visuals, and completely useless as well. So I'm pretty sure that you want to learn how to draw something.

Drawing In Direct3D is extremely simple when you get your head around it, but it requires a fair amount of work and memory before you can get to this point. The first part is understanding what the different words mean – right now we'll stick to the simple definitions and elaborate on them as we get more advanced later on.

### **1. Vertices (plural of vertex)**

A vertex can be thought of as a defining point – the corner of a triangle, square or other shape. Using vertices we can construct 2D and 3D shapes with various properties; a vertex will be described by a visual basic User Defined Type (we'll see these later) and are usually made up of a position, colour and texture coordinates.

### **2. Polygon**

Polygons are what you'll have heard about by "normal" people most of the time – so and so 3D card pumps out 101 million polygons a second (or whatever), in fact, Direct3D renders all of it's primitives using triangles. But then again, a triangle is the simplest possible polygon. Lists of triangles are stored as arrays of the vertex type that you are using.

### **3. Face**

A face is usually 3 vertices arranged in some sort of polygon, but it also has an orientation – you can tell which way it is facing. Direct3D interpolates vertex components across a face, in particular colour – as we'll be seeing later on. 3D models (imported from 3D modelling programs) tend to be made up of 100's or 1000's of faces.

### **4. Textures**

A texture is applied to a triangle to make it look more real, Textures are just 2D bitmaps/pictures loaded from the hard drive into memory and then mapped to relevant polygons during rendering. We will cover these in more detail later on.

### **5. Mesh**

A mesh is another word for a model – it is usually represented by one object in the program, and contains 100s or 1000s of vertices and faces, along with all relevant materials and textures. These will be covered later on.

Now you have those words floating around we can start doing some interesting things. Don't swear by the above definitions though – they are very loose and are only there to offer a basic introduction to the terms, more complex and meaningful definitions will be offered as and when they are necessary.

The first thing I want to cover is the 3 types of vertex that you can use. Whilst the structure of Direct3D allows for 100's of different combinations there are three types that will serve most situations – and all other situations will be adaptations or modification of these three.

### 1. Untransformed and Unlit vertex

These vertices are literally just points in 3D space with an orientation and a set of texture coordinates. Direct3D will do the lighting for you – you set up some lights, some vertices and it'll do the rest. These tend to be the most commonly used unless people opt for using lightmaps or pre-calculated lighting (as some commercial games do).

### 2. Untransformed and Lit vertex

These vertices are points in 3D space with texture coordinates like the first type, but these ones have a colour value. This allows us to make proper 3D geometry without having to worry about lighting. When we start doing 3D geometry these will be the first type to use – as they're the easiest.

### 3. Transformed and Lit vertex

These are vertices specified in two dimensions – screen coordinates, all Direct3D does is apply textures to them, clip them and draw them – you are expected to specify a colour value and a valid 2D position. Using these are the only way you will get Direct3D to do 2D graphics.

Yet more things to remember... But before we go through a simple demonstration of how to use these vertices there is one more thing you need to know. Flexible Vertex Formats. As I mentioned earlier, Direct3D allows for 100's of possible vertex types – it is through this system of Flexible Vertex Formats (FVF) that we achieve this. A flexible vertex format description is a variable of type Long that is a combination of flags that Direct3D can use to work out what format the data you pass it is in. If you pass invalid, or incorrect for the data you use one of two things will happen – Nothing will be rendered, something very strange will be rendered.

The vertex formats for the 3 main types look like this:

```
Const FVF_TLVERTEX = (D3DFVF_XYZRHW Or D3DFVF_TEX1 Or D3DFVF_DIFFUSE Or D3DFVF_SPECULAR)
Const FVF_LVERTEX = (D3DFVF_XYZ Or D3DFVF_DIFFUSE Or D3DFVF_SPECULAR Or D3DFVF_TEX1)
Const FVF_VERTEX = (D3DFVF_XYZ Or D3DFVF_NORMAL Or D3DFVF_TEX1)
```

And their UDT structures will look like this:

```
Private Type TLVERTEX
```

```
  X As Single
```

```
  Y As Single
```

```
  Z As Single
```

```
  rhw As Single
```

```
  color As Long
```

```
  Specular As Long
```

```
  tu As Single
```

```
  tv As Single
```

```
End Type
```

```
Private Type LITVERTEX
```

```
  X As Single
```

```
  Y As Single
```

```
  Z As Single
```

```
  color As Long
```

```
  Specular As Long
```

```
  tu As Single
```

```
  tv As Single
```

```
End Type
```

```
Private Type VERTEX
```

```
  X As Single
```

```
  Y As Single
```

```
  Z As Single
```

```
  nx As Single
```

```
  ny As Single
```

```
  nz As Single
```

```
  tu As Single
```

```
  tv As Single
```

```
End Type
```

Right this second you don't really need to know all of these – I put them in mainly for reference purposes; those of you familiar with DirectX7 in visual basic, or DirectX in another language may well want to jump ahead slightly.

In the next section we'll extend our sample program to render some basic 2D and 3D geometry in fullscreen mode. It may sound like a small task, but you've got the mountain to climb yet my friend.

## Drawing Something – The Practical part

As already mentioned we render all our geometry using triangles (I prefer to use the term triangles instead of polygon – but feel free to use whichever you prefer). These triangles will be made up of a set of vertices; and these vertices will have a specific set of properties depending on what they're for.

It would therefore make sense that we had an array of one vertex type filled with data – which is exactly what we're going to do. As you can imagine this will get very long – one line of code per vertex, 3 per triangle – even a simple cube can take up a hundred or so lines of code... which is the reason why I'm going to keep this simple.

### **Step 1: Setting things up.**

First we need to create an array of vertices:

```
Dim TriVert(0 To 2) As TLVERTEX '//we require 3 vertices to make a triangle...
```

Then we need to set a couple of parameters in the initialisation procedure:

```
D3DDevice.SetVertexShader FVF_TLVERTEX  
D3DDevice.SetRenderState D3DRS_LIGHTING, 0
```

The first line tells the rendering device what type of vertex we're going to be using – pass the constant that we defined earlier here. The second parameter tells Direct3D that we don't want it to do the lighting – by default it will.

### **Step 2: Making the triangle**

This is as simple as filling out the array with the required data, for clarity we'll stick it in a whole new procedure, which will be called at the end of the initialisation process:

```
Private Sub InitialiseGeometry()  
    TriVert(0) = CreateTLVertex(0, 0, 0, 1, &HFF0000, 0, 0, 0)  
    TriVert(1) = CreateTLVertex(175, 0, 0, 1, &HFF00&, 0, 0, 0)  
    TriVert(2) = CreateTLVertex(0, 175, 0, 1, &HFF&, 0, 0, 0)  
End Sub
```

Three things to note here; firstly we have a new function here – CreateTLVertex(), this is a little helper function that I wrote to help in filling the structures with the relevant data, it looks like this:

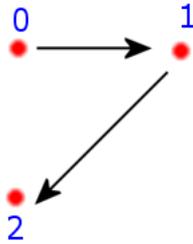
```

Private Function CreateTLVertex(X As Single, Y As Single, Z As Single, rhw As Single, _
                                Color As Long, Specular As Long, tu As Single, _
                                tv As Single) As TLVERTEX
    CreateTLVertex.X = X
    CreateTLVertex.Y = Y
    CreateTLVertex.Z = Z
    CreateTLVertex.rhw = rhw
    CreateTLVertex.Color = Color
    CreateTLVertex.Specular = Specular
    CreateTLVertex.tu = tu
    CreateTLVertex.tv = tv
End Function

```

Secondly we have to specify the colour as a long, using the RGB( ) function wont work properly here – it is possible to reverse the values and use it – RGB(B, G, R), but if you can use hexadecimal it's the preferred method. Think of it as the same as an HTML colour code and you'll be fine.

Thirdly, and more subtly is the order in which the vertices where created. This is actually extremely important – get this wrong and Direct3D will cull (remove) your triangles and not render them; It is often quite likely that this is the cause if you make a program and nothing is rendered (yet appears to be set up correctly). If we plot the triangle coordinates in order we will see the pattern – Clockwise:



You can set which type of triangle Direct3D will cull (Clockwise, Counter-Clockwise or none), but by default it will cull counter-clockwise triangles, therefore rendering only those that are in a clockwise order. You may well think that it's easy just to stop it from removing any triangles – whilst that is true it's bad practise. If you can get into the habit of generating your vertices in the correct order from the beginning then so much the better; but it's still useful to know how you specify the culling modes:

```

D3DDevice.SetRenderState D3DRS_CULLMODE, D3DCULL_NONE
D3DDevice.SetRenderState D3DRS_CULLMODE, D3DCULL_CW
D3DDevice.SetRenderState D3DRS_CULLMODE, D3DCULL_CCW

```

Fairly simple really, whichever you specify the opposite will be rendered; for example, our triangles are clockwise ordered, therefore you should specify the above as being D3DCULL\_CCW (but that's the default, so you don't need to).

The last thing to note before we move on is to do with transformed and lit vertices. As I told you, transformed and lit vertices are 2D – X and Y, so why is there a Z coordinate? The Z-Coordinate should be on a 0.0 to 1.0 scale, and when a depth buffer is attached (more on that later) triangles will be drawn over each other based

on this value, for example – a triangle with a Z of 0 will go over a triangle with a Z of 1. And a Triangle with 3 different Z values will go through any other triangles that it intersects.... If two triangles have the same Z value, whichever is rendered last will appear on top.

### Step 3: Rendering the triangle

We'll now go back to our Render() procedure – as discussed earlier – and update it to render our new triangle. At the moment we aren't doing anything clever, no textures, no transformations – so everything can be done with one call.

Later on we'll use a more optimised method of rendering, and some more clever tricks; but right now we'll stick to the basics:

```
D3DDevice.BeginScene
    'All rendering calls go between these two lines
    D3DDevice.DrawPrimitiveUP D3DPT_TRIANGLELIST, 1, TriVert(0), Len(TriVert(0))
D3DDevice.EndScene
```

Simple as that really. We use the function "DrawPrimitiveUP" to render a custom type of vertex straight from memory – it is of type TRIANGLELIST (more in a second), is made up of one triangle and uses the specified array with the specified size. In more detail:

The first parameter is the primitive type, whilst Direct3D does all solid rendering as triangles, but we can also render points and lines. The full range of options are listed here:

**D3DPT\_POINTLIST:** Direct3D draws each vertex as an unconnected point in 3D space, just a single pixel dot on the screen. Useful for particle effects (but there are better ways).

**D3DPT\_LINELIST:** Every pair of entries specifies a pair of coordinates for Direct3D to draw a line between (0 & 1, 2 & 3, 4 & 5 etc...)

**D3DPT\_LINESTRIP:** Same as a line list, but the beginning of one line joins up with the end of the previous – creating a continuous line through all the specified points.

**D3DPT\_TRIANGLELIST:** Every triplet of vertices defines a new triangle – this will be filled in solid, with colour components blended across it's surface; this is the method we currently use. (0-1-2 & 3-4-5 & 6-7-8-9 etc...).

**D3DPT\_TRIANGLESTRIP:** Same as a triangle list, but it creates a series of triangles all joining up; from the second triangle onwards each triangle uses the last vertex of the previous triangle as it's first vertex (0-1-2 & 2-3-4 & 4-5-6 etc...). Colours are blended across the surface of these in the same way as a triangle list.

**D3DPT\_TRIANGLEFAN:** draws a series of triangles all connected to the first vertex – perfect for octagonal, hexagonal or circular type shapes. 0-1-2 & 0-3-4 & 0-5-6 etc... These triangles are drawn solid, and colour is blended across them.

There are numerous advantages and disadvantages to all of these methods; some of more obvious than others. As a start you should already have guessed that the less triangles you use the faster it goes – so try to keep them to a low number (without looking nasty); more subtly, you should also keep the vertex count as low as possible when creating geometry. This is on the basis that Direct3D will send all of the data through the various cables, pipes and chips to the 3D card – and the more

data you have to send the longer it takes; so the less vertices you use the faster the data can be transmitted. When drawing a continuous line it would make perfect sense to use a line strip. When dealing with triangles decide on what you need to do – often it is faster and simpler to render something using a triangle strip, other times it makes it more complicated and you should use a triangle list; then there is the option of using a triangle fan for circular type objects. Experiment and see...

The second parameter is the primitive count, think of it as the number of triangles, or number of points (depending on the primitive type). In this example we only created one triangle, so have specified the fact that there is only 1 triangle.

The third parameter specifies where Direct3D should look for the vertex data, this must be an entry in the array; usually the first entry – but it doesn't have to be; just remember that there needs to be the correct number of vertices left for the specified primitive count.

The last parameter specifies how big (in memory bytes) our vertex structure is – this is for Direct3D's internal usage. You don't really need to understand how it works, but basically the third parameter points to the beginning of the memory to look for, and Direct3D knows how many entries there will be (primitive count value), using this parameter it can get the size of the structure, and therefore work out where all the individual bits of data are, and how much memory the whole lot should take up. Use `Len( )` on the first element in the array for this.

Now that you can render simple 2D geometry you should practise it – try making a square using the various methods, draw a circle-like object with a triangle fan; and try making an arch or rounded rectangle using a triangle strip. If you look at almost any basic geometric shape it will always break down into a series of triangles (sometimes not very nicely), but with some basic maths skills it is easy to write an algorithm that generates an arch (or whatever) from a series of triangles...

## **Overview**

You may well think that very little has been done in this article – you would be very wrong thinking this. In the next articles we will cover all of the major aspects of Direct3D programming – if there was anything that you didn't follow in this article (code wise) then it's going to catch you out later. Trust me. I have written enough DirectX applications that I can write a complete DirectX 7 and DirectX 8 program similar to this off the top of my head in very little time.

The next article will advance our knowledge of geometry into the 3<sup>rd</sup> dimension; along the way we'll learn about vertex buffers, index buffers, fullscreen mode, depth buffers, normals and some basic lighting – sound like fun? You bet ☺

Any feedback on this article is much appreciated, or if you have any questions I'd be happy to try and help you out (but remember, I'm good, but I don't know everything [yet]). Drop me a message: [Jack.Hoxley@DirectX4VB.com](mailto:Jack.Hoxley@DirectX4VB.com) , or visit my main programming site, [www.DirectX4VB.com](http://www.DirectX4VB.com) for a massive collection of over 100 tutorials and 25 articles...

## DirectX Graphics For Visual Basic Part 2

Welcome back to part 2 of this mini-series, hopefully you've read and learnt the information covered in the first part (thanks to all the complements I received about the first article), this article starts where the last one left off – things will not be covered twice, so make sure you know what happened in the first article...

By the time you've read and learnt the things I am about to cover you should be perfectly capable of creating a simple game/demo – which shows you how quickly you can get started in DirectX3D. Having said this, don't expect to finish this article (or this series) and go on to write the next big 3D engine – it won't happen, I've had many emails from people who've only read the first couple of tutorials on the basics of D3D and want to get straight on with a "Simple" quake clone... try something like pong/tetris/snakes first.

This article is going to be quite steep – the things covered may well not come to you easily, if not, re-read the article until it does or seek out other beginners guides to 3D graphics / theory. Today we'll be covering:

- 1) Setting up the sample application to go full-3D.
- 2) Extending the last example to use basic 3D geometry.
- 3) Extending this further to use vertex buffers and index buffers.

The above 3 things would usually be covered by several articles, as they are deceptively big topics, anyway, onwards and upwards!

## **Reconfiguring our Direct3D application**

The sample at the end of the last article was very simplistic – not much use for anything really, before we go into full-3D we'll need to add a few parameters and configure a few new things.

I also want to take the time to introduce full screen mode, this is the main display format used by games, where, funnily enough, your game occupies the entire screen. Full screen mode is much faster, and isn't held back by windows (which is effectively suspended in the background). Full screen mode requires you to pick a resolution that the hardware/monitor combination can handle – open up the windows display properties and see what settings you can set the resolution slider to – these (but not always) will be the display modes that Direct3D can use on your hardware. 800x600, 1024x768 are examples of full screen modes. The important thing about this is that the resolutions available will differ from one computer to another – 640x480, 800x600, 1024x768 all tend to be standard resolutions, but there is no guarantee that they will be available (only 1 of my 2 computers supports 1024x768 display modes). To solve this problem we must use enumeration.

```
Dim tmpDispMode As D3DDISPLAYMODE '//used during the enumeration of avail. modes
Dim I As Long '//so we can loop through the avail. display modes

For I = 0 To D3D.GetAdapterModeCount(0) - 1 'primary adapter
    D3D.EnumAdapterModes 0, I, tmpDispMode
    Debug.Print tmpDispMode.Width & "x" & tmpDispMode.Height
Next I
```

The previous piece of code will output a list of all the display modes supported by your hardware to VB's immediate (debug) window. The code will need to go at the start of the Initialise() function, but after the Dx and D3D objects have been initialised. The output of the above code, for my GeForce 256 + generic 15" monitor was:

```
320x200
320x240
400x300
480x360
512x384
640x400
640x480
800x600
960x720
1024x768
1152x864
1280x960
1280x1024
320x200
320x240
400x300
480x360
512x384
640x400
640x480
800x600
960x720
1024x768
1152x864
```

```
1280x960
1280x1024
```

So why are there two of each resolution? Its not a mistake, it's down to the format of the display mode. Anyone paying any attention to games will know that you can have 32 bit and 16 bit rendering (amongst various other formats) – the above list does not contain that data, but the first copy of the resolutions will be in 16 bit format, the second set will be in 32 bit format. This requires some discussion:

```
D3DFMT_A1R5G5B5
D3DFMT_A4R4G4B4
D3DFMT_A8R3G3B2
D3DFMT_A8R8G8B8
D3DFMT_DXT1
D3DFMT_DXT2
D3DFMT_DXT3
D3DFMT_DXT4
D3DFMT_DXT5
D3DFMT_R5G6B5
D3DFMT_R8G8B8
D3DFMT_X1R5G5B5
D3DFMT_X4R4G4B4
D3DFMT_X8R8G8B8
```

Above is a selection of members from the enumeration type "CONST\_D3DFORMAT" – which we'll be using later. All of the above describe a format that the display mode should be in, such as 32 bit/16 bit – but it's not as simple as saying 16 or 32 bit... you can work it out by counting the number of bits in the description:

### D3DFMT\_X8R8G8B8

Add up the 8's and you get 32 – which indicates that D3DFMT\_X8R8G8B8 is a 32 bit mode, all of the ones above are either 16 or 32 bit format (except the D3DFMT\_DXT\* ones). The next part to notice is the lettering – indicating the channel, all of them will have an RGB triplet – Red, Green and Blue – you should all know that colours on a computer screen are made up of these 3 colours, even using paintbrush you could probably see this in action. We also have an optional X or A channel, the X just means unused, there are bits allocated, but they wont have anything in them, and wont be used for anything. The A channel is alpha, something that will be covered later on. If you don't need alpha blending/transparencies then you'll be okay using an X\*\*\*\*\* format, if you need alpha but accidentally use an X\*\*\*\*\* format nothing will happen – or at least what you want to happen won't.

A word on accuracy, the more bits to a channel the more colours you can represent, which is why the 32 bit formats will look substantially better than the 16 bit formats. A channel can represent  $2^n$  (where n is the number of bits) colours. An 8 bit channel can therefore represent  $2^8$  colours = 256 colours, a 4 bit channel can only represent  $2^4$  channels = 16 colours. You may have noticed that there is an R5G5B5 format and an R5G6B5 format – this is down to the fact that our eyes are more sensitive to green light, so being able to represent more colours in the green channel is better. On a connected, but not particularly useful note – the total number of colours supported by a display mode will be  $2^n$  again, where n is the total number of bits being used, not including the X channel. I tend not to include the A

channel in my calculations, but you can if you want. Therefore a 16 bit colour will have  $2^{16}$  colours = 65536, and a 32 bit colour will have  $2^{32}$  colours = 4,294,967,296 – roughly 4.3 billion...

Now (hopefully) we have an understanding of display formats, we can go about setting one up. I've written a small function that will check for support of a specified display mode – it works fine for this tutorial, but you'll probably need a more rigid function for a proper project:

```
Private Function CheckDisplayMode(Width As Long, Height As Long, Depth As Long) As CONST_D3DFORMAT
//0. any variables
  Dim I As Long
  Dim DispMode As D3DDISPLAYMODE

//1. Scan through
  For I = 0 To D3D.GetAdapterModeCount(0) - 1
    D3D.EnumAdapterModes 0, I, DispMode
    If DispMode.Width = Width Then
      If DispMode.Height = Height Then
        If DispMode.Format = D3DFMT_R5G6B5 Or D3DFMT_X1R5G5B5 Or D3DFMT_X4R4G4B4 Then
          '16 bit mode
          If Depth = 16 Then CheckDisplayMode = DispMode.Format: Exit Function
        ElseIf DispMode.Format = D3DFMT_R8G8B8 Or D3DFMT_X8R8G8B8 Then
          '32bit mode
          If Depth = 32 Then CheckDisplayMode = DispMode.Format: Exit Function
        End If
      End If
    End If
  Next I
  CheckDisplayMode = D3DFMT_UNKNOWN
End Function
```

**ERRATA:** several lines in the above code do not work correctly (I have left the original source visible). Where it uses "DispMode.Format = \*\*\*\* or \*\*\*\* or \*\*\*\*\*" code, it will be evaluated incorrectly. Change this to be "DispMode.Format = \*\*\*\* or DispMode.Format = \*\*\*\* or DispMode.Format = \*\*\*\*\*" and it will work fine.

Fairly simple really, it assumes that the D3D object has been created, and will use the global copy of the object – if it hasn't been created this piece of code will error-out. The only slightly complicated part is the format selection, it defines formats as either 16 bit or 32 bit – whilst the 32 bit selection technically includes a 24 bit format, colour wise it's identical. I've not checked against formats with alpha components – the display mode wont use an alpha channel, that's for texturing – you can set up a display mode with an alpha channel, but there's little point, on top of that the formats listed here are more likely to be supported than the equivalent with an alpha channel. We can extend the usage of this function to help select the samples display mode – we'll hard code this part rather than spend time building a user interface in to allow the user to select the resolution – it's not hard to do that, and you can probably work it out as you need it.

```

DispMode.Format = CheckDisplayMode(640, 480, 32)
If DispMode.Format > D3DFMT_UNKNOWN Then
    '640x480x32 is supported
    DispMode.Width = 640: DispMode.Height = 480
Else
    DispMode.Format = CheckDisplayMode(640, 480, 16)
    If DispMode.Format > D3DFMT_UNKNOWN Then
        '640x480x16 is supported
        DispMode.Width = 640: DispMode.Height = 480
    Else
        'hmm, neither are supported. oh well...
        MsgBox "Your hardware does not appear to support" _
            & " 640x480 display modes in either 16 bit or 32 bit modes. Exiting" _
            , vbInformation, "Error"
        Unload Me
    end
End If
End If

```

Not too complicated really. We could check for higher resolutions, but for this sample it isn't really necessary. By the time this little segment of code has been executed we will have a properly initialised, valid D3DDISPLAYMODE structure that we can use when setting up our device. If it could not create the structure it will exit out of the program.

We aren't done yet though, we need to fill out the D3DPRESENT\_PARAMETERS structure differently from the windowed mode example. This is mostly down to the device creation requiring more data than in the last sample. The new configuration looks like this:

```

D3DWindow.BackBufferCount = 1
D3DWindow.BackBufferFormat = DispMode.Format
D3DWindow.BackBufferWidth = DispMode.Width
D3DWindow.BackBufferHeight = DispMode.Height
D3DWindow.hDeviceWindow = frmMain.hWnd
D3DWindow.AutoDepthStencilFormat = D3DFMT_D16
D3DWindow.EnableAutoDepthStencil = 1
D3DWindow.SwapEffect = D3DSWAPEFFECT_COPY_VSYNC

```

Notice that we're copying the display mode format information to the structure at this point, you could avoid this by writing straight to this structure – but for clarity I left it separate.

The first section deals with configuring a backbuffer. Anyone who's done any work with DirectDraw/Direct3D in DirectX7 will already know what one of these is (you should do). When Direct3D does the actual rendering of 3D geometry onto a 2D surface it will do it in parts, usually as each piece of geometry is rendered. If we rendered straight to the screen in all but the highest frame rate situations you would be able to see the screen being drawn piece by piece – even if it was going quite quickly you'd be able to pick up on strange artefacts appearing – a tree appearing and quickly being overwritten by the house in front of it (for example). To solve this problem we use a secondary buffer, the image is composed on this surface (an

identical size to the screen), and then the whole contents of the backbuffer are copied to the screen in one quick operation (its not actually copied, the addresses/pointers are switched around). This removes the possibility of any drawing artefacts appearing – or at least it should do... We configure our backbuffer here using our display mode, we never actually configure the screen surface, D3D will take the measurements specified in the backbuffer members and use those – saves on any simple data mis-matching errors.

The second part deals with configuring the depth buffer. This is another concept that you'll need to grasp when dealing with 3D environments. In 3D we, obviously, have 3 dimensions, XYZ, and in 2D we only have X and Y, when we want to project our 3D scene onto our 2D screen we need to know what happens to this 3<sup>rd</sup> dimension. As things are converted into 2D in any given order we will need to check if the current part we are drawing is in front of, or behind the current piece of scene in the frame buffer (the screen/backbuffer). Or in more technical language, when we draw a pixel in 2D we check it's depth coordinate against that stored for the same location in depth buffer (which will hold the depth for the pixel currently in the frame buffer), if the depth is greater (the new pixel is behind the old one) then it wont be drawn, if the depth is les (the new pixel is in front of the old one) then it will draw it over the top of existing pixel. The depth buffer, is therefore a surface identical in dimensions to the screen and backbuffer. The only involvement we'll ever have with it is telling D3D we want to use it, turning it on and clearing it before each frame (to remove the depth information from the previous frame). The only important part at this stage is specifying what format the depth buffer will be in – similar to the way we specified what format the screen/backbuffer will be in. The more bits allocated to each pixel the more accurate the depth testing will be, typically they come as standard at 16bits per pixel, newer hardware is allowing 24 bit or 32 bit depth buffers (usually as combinations with a stencil buffer). We can enumerate what depth buffer modes are available using the following code:

```

'In order of preference: 32, 24, 16
If D3D.CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, DispMode.Format, _
                        D3DUSAGE_DEPTHSTENCIL,
                        D3DRTYPE_SURFACE, D3DFMT_D32) = D3D_OK Then
    '//Enable a pure 32bit Depth buffer
    D3DWindow.AutoDepthStencilFormat = D3DFMT_D32
    D3DWindow.EnableAutoDepthStencil = 1
    Debug.Print "32 bit Depth buffer selected"
Else '//search for a 24 bit depth buffer

    If D3D.CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, _
                            DispMode.Format, D3DUSAGE_DEPTHSTENCIL,
                            D3DRTYPE_SURFACE, D3DFMT_D24X8) = D3D_OK Then
        '//Enable a 24 bit depth buffer
        D3DWindow.AutoDepthStencilFormat = D3DFMT_D24X8
        D3DWindow.EnableAutoDepthStencil = 1
        Debug.Print "24 bit Depth buffer selected"
    Else
        If D3D.CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, _
                                DispMode.Format, D3DUSAGE_DEPTHSTENCIL,
                                D3DRTYPE_SURFACE, D3DFMT_D16) = D3D_OK Then
            '//Enable a 16 bit depth buffer
            D3DWindow.AutoDepthStencilFormat = D3DFMT_D16
            D3DWindow.EnableAutoDepthStencil = 1
            Debug.Print "16 bit Depth buffer selected"
        Else
            '//hmm. No depth buffers available... Dont use one then :)
            D3DWindow.EnableAutoDepthStencil = 0
            Debug.Print "no Depth buffer selected"
        End If
    End If
End If
End If

```

But for clarity this sample just uses a 16 bit depth buffer, in about 99% of cases the hardware will support a 16 bit depth buffer – which is perfectly acceptable for most 3D environments. If one is not available in hardware you may find that Direct3D will emulate ones existence – which is much slower, but it will still function.

The last two things to discuss are the .hDeviceWindow member, and the .SwapEffect member. These are not complicated, the first parameter, hDeviceWindow needs to be the hWnd property of the form that you are using – this is so that Direct3D can keep track of your application/window, if the form is closed, minimised or moved Direct3D can find out. The SwapEffect member indicates how Direct3D should draw to the screen, there are two main choices here, V-Sync or not. Hopefully you are aware of the monitors refresh rate – or how it works, if you use V-Sync then Direct3D will wait until a vertical refresh event occurs before drawing the next frame, therefore, if you only have a 70hz monitor the maximum frame rate will be around 70fps (not always exactly) – using a v-sync is usually better quality than without as it performs the copy at the same time as the monitor and no artefacts should appear. Disabling V-Sync forces Direct3D to copy the frame buffers to the screen as soon as it's finished rendering – using this method you can achieve considerably higher frame rates (if it's being locked to the refresh rate that is...) at the cost of visual artefacts; whilst it doesn't affect some hardware (mine is fine), on others you can get a visible tear line across the screen – where one frame is above and one frame (usually the previous) is below, if this happens it will look pretty ugly! Also note that the drivers have the ability to override these settings, my drivers are setup

to ignore v-sync, and I cant, programmatically, force it to use the V-sync; vice-versa when I enable v-sync in the driver properties. The available choices for the SwapEffect member are:

**D3DSWAPEFFECT\_COPY**  
**D3DSWAPEFFECT\_COPY\_VSYNC**  
D3DSWAPEFFECT\_DISCARD  
**D3DSWAPEFFECT\_FLIP**  
D3DSWAPEFFECT\_FORCE\_DWORD

The 3 bolded entries are the ones that you should be using. The `_COPY` member is for a single backbuffer (like ours) that ignores V-Sync where possible, the `_COPY_VSYNC` option is the same, but will lock drawing to the monitors refresh rate. The `_FLIP` member is the same as the `_COPY` member except for multiple backbuffers (usually when you have 2).

Now we've covered the redesigned initialisation process. At last... There are two final lines that we should add after the device has been created:

```
D3DDevice.SetRenderState D3DRS_ZENABLE, 1
```

That will enable our depth buffer (AKA Z-Buffer) for rendering. And this next line goes in place of the current line in the `Render()` function:

```
D3DDevice.Clear 0, ByVal 0, D3DCLEAR_TARGET Or D3DCLEAR_ZBUFFER, &HCCCCFF, 1#, 0
```

All that's changed there is that it's clearing the depth buffer as well as the frame buffers. If you leave this line out you'll start getting some strange artefacts appearing (you can use it to your advantage) where the new scene is drawn according to the depth information of the previous scene...

If you now run the program (F5 or Ctrl+F5 in the IDE) you should be greeted with a 640x480 light blue screen and a small triangle in the top left corner – identical to the sample in the last example – but in fullscreen!

## **Getting Started With Basic 3D Geometry**

Now things start to get more fun. What we've done so far could be seen as being fairly boring, just setting up the foundations for bigger things really...

Luckily, now that the initialisation process is sorted out quite nicely adding the ability to use 3D geometry will be quite straight forward, if you read the first article properly you'll find that a lot of the ground work for geometry has already been laid down. To summarise the relevant key points brought up in the first article:

- 3D Geometry is still made up of vertices and triangles.
- Triangles must be created in a clockwise order (unless you change it to CCW/none).

Simple as that really, the only major change is going to be the switch from using 2D vertex coordinates into using 3D vertex coordinates, which can't be too hard can it? You also need to remember that it's all relative to the camera viewpoint rather than to the screen (2D) coordinates.

The next part of the sample program we're going to design will render a small 3D cube on the screen, then we'll use this as a base to learning more of the manipulation features that Direct3D exposes for us (Matrices). First, take the user defined type and FVF description from the last article:

```
Const FVF_LVERTEX = (D3DFVF_XYZ Or D3DFVF_DIFFUSE Or D3DFVF_SPECULAR Or D3DFVF_TEX1)

Private Type LITVERTEX
    X As Single
    Y As Single
    Z As Single
    Color As Long
    Specular As Long
    tu As Single
    tv As Single
End Type
```

This particular vertex is pre-lit by us, that means that we must specify the colour of the vertex rather than let Direct3D light it for us (lighting will be covered later). We'll also use a little helper function CreateLitVertex() to help us initialise the structures, this is very similar to the one in the previous article, if you are unclear as to the exact nature of it have a look at the sample code for this article...

Now that we have the vertex descriptor/UDT set up we need to define the vertex structures for our cube. For this sample we're going to use 36 vertices:

```
Dim CubeVerts(0 To 35) As LITVERTEX
```

Why are we using 36 vertices for a 6 sided/8 cornered object? A cube has 6 square faces, to make up a square we require 2 triangles, therefore we have 12 triangles in total. Each triangle requires 3 vertices, hence the 36 figure. Later on we'll learn how

to reduce the number of vertices in the cube to 8, but for a first example this will do fine...

```

Private Sub InitialiseGeometry()
  '//0. Any Variables

  '//1. Define the colours at each corner
  Const Corner000 As Long = &HFF0000 'red
  Const Corner001 As Long = &HFF00 'green
  Const Corner010 As Long = &HFF 'blue
  Const Corner011 As Long = &HFF00FF 'magenta
  Const Corner100 As Long = &HFFFF00 'yellow
  Const Corner101 As Long = &HFFFF 'cyan
  Const Corner110 As Long = &HFF8000 'orange
  Const Corner111 As Long = &HFFFFFF 'white

  '//2. Define the faces
  'top
  CubeVerts(0) = CreateLitVertex(-1, 1, -1, Corner010, 0, 0, 0)
  CubeVerts(1) = CreateLitVertex(1, 1, -1, Corner110, 0, 0, 0)
  CubeVerts(2) = CreateLitVertex(-1, 1, 1, Corner011, 0, 0, 0)

  CubeVerts(3) = CreateLitVertex(1, 1, -1, Corner110, 0, 0, 0)
  CubeVerts(4) = CreateLitVertex(1, 1, 1, Corner111, 0, 0, 0)
  CubeVerts(5) = CreateLitVertex(-1, 1, 1, Corner011, 0, 0, 0)

  'bottom
  CubeVerts(6) = CreateLitVertex(-1, -1, -1, Corner000, 0, 0, 0)
  CubeVerts(7) = CreateLitVertex(1, -1, -1, Corner100, 0, 0, 0)
  CubeVerts(8) = CreateLitVertex(-1, -1, 1, Corner001, 0, 0, 0)

  CubeVerts(9) = CreateLitVertex(1, -1, -1, Corner100, 0, 0, 0)
  CubeVerts(10) = CreateLitVertex(1, -1, 1, Corner101, 0, 0, 0)
  CubeVerts(11) = CreateLitVertex(-1, -1, 1, Corner001, 0, 0, 0)

  'left
  CubeVerts(12) = CreateLitVertex(-1, 1, -1, Corner010, 0, 0, 0)
  CubeVerts(13) = CreateLitVertex(-1, 1, 1, Corner011, 0, 0, 0)
  CubeVerts(14) = CreateLitVertex(-1, -1, -1, Corner000, 0, 0, 0)

  CubeVerts(15) = CreateLitVertex(-1, 1, 1, Corner011, 0, 0, 0)
  CubeVerts(16) = CreateLitVertex(-1, -1, 1, Corner001, 0, 0, 0)
  CubeVerts(17) = CreateLitVertex(-1, -1, -1, Corner000, 0, 0, 0)

  'right
  CubeVerts(18) = CreateLitVertex(1, 1, -1, Corner110, 0, 0, 0)
  CubeVerts(19) = CreateLitVertex(1, 1, 1, Corner111, 0, 0, 0)
  CubeVerts(20) = CreateLitVertex(1, -1, -1, Corner100, 0, 0, 0)

  CubeVerts(21) = CreateLitVertex(1, 1, 1, Corner111, 0, 0, 0)
  CubeVerts(22) = CreateLitVertex(1, -1, 1, Corner101, 0, 0, 0)
  CubeVerts(23) = CreateLitVertex(1, -1, -1, Corner100, 0, 0, 0)

  'front
  CubeVerts(24) = CreateLitVertex(-1, 1, 1, Corner011, 0, 0, 0)
  CubeVerts(25) = CreateLitVertex(1, 1, 1, Corner111, 0, 0, 0)
  CubeVerts(26) = CreateLitVertex(-1, -1, 1, Corner001, 0, 0, 0)

  CubeVerts(27) = CreateLitVertex(1, 1, 1, Corner111, 0, 0, 0)
  CubeVerts(28) = CreateLitVertex(1, -1, 1, Corner101, 0, 0, 0)
  CubeVerts(29) = CreateLitVertex(-1, -1, 1, Corner001, 0, 0, 0)

  'back
  CubeVerts(30) = CreateLitVertex(-1, 1, -1, Corner010, 0, 0, 0)
  CubeVerts(31) = CreateLitVertex(1, 1, -1, Corner110, 0, 0, 0)

```

```
CubeVerts(32) = CreateLitVertex(-1, -1, -1, Corner000, 0, 0, 0)

CubeVerts(33) = CreateLitVertex(1, 1, -1, Corner110, 0, 0, 0)
CubeVerts(34) = CreateLitVertex(1, -1, -1, Corner100, 0, 0, 0)
CubeVerts(35) = CreateLitVertex(-1, -1, -1, Corner000, 0, 0, 0)
End Sub
```

Doesn't that look so pretty! Well, not really...

There's nothing more to this than brute force calculations, I went through (took me about 10 minutes) working out which vertex would have what coordinate, and then what colour it would have. I used a set of constants to set the colours, as it makes it much easier when you want to change the colour of a given corner – saves you having to search through all 36 changing the wrong/old values for the new ones. Now that the array of vertices is completed we can render them in an almost identical way to last time:

```
D3DDevice.BeginScene
  'All rendering calls go between these two lines
  D3DDevice.DrawPrimitiveUP D3DPT_TRIANGLELIST, 12, CubeVerts(0), Len(CubeVerts(0))
D3DDevice.EndScene
```

Hardly complicated really is it? However, if you now run the project to see what it looks like you'll be surprised by the result – a big yellow/magenta/white square covering the entire screen. Great – not like the pretty coloured cube we were aiming for. This is because we "forgot" (or I didn't tell you) about one major thing that needs to be set up: Projection, World and View Matrices.

## Matrices And Transformations

If you think about it logically, we have not yet told Direct3D where we're looking from, or where we're looking at – or what properties our camera has. Which is what we now need to set up. Matrices are used a lot in Direct3D (as in most 3D API's I do believe), they control transformations of world geometry (we'll see that later), they control the configuration of the camera, and the information about where the camera is/where it's looking at. The actual maths behind matrices is quite complicated, and you don't really need to know why they work – or even how they do what they do, you can get along fine in Direct3D by just knowing how to use them. There are 3 types of matrix that we need to set up, they are:

### **The Projection Matrix**

This matrix is usually configured during initialisation and left alone for the rest of the time, you can alter it other times to achieve effects such as zooming, but in general you probably won't need to.

```
D3DXMatrixPerspectiveFovLH(  
    MOut as D3DMatrix,  
    fovy as Single,  
    Aspect as Single,  
    zn as Single,  
    zf as Single )
```

Above is the function prototype for the projection matrix altering function. There are others, but for 99% of Direct3D applications this one will be perfectly acceptable. The parameters should be used like so:

**MOut** : An empty (not a requirement though) D3DMatrix structure that will be filled with the relevant details of a projection matrix

**fovy** : This is the view angle for the camera, the wider the angle this is set at the more "stuff" you'll see on the screen. This is often set at 45, 60 or 90 degrees – but as it's measured in radians (not degrees) you should specify it in terms of PI, so the above list becomes  $\text{Pi}/4$  ,  $\text{Pi}/3$  and  $\text{Pi}/2$  – there are  $2\text{Pi}$  radians in 360 degrees, therefore 1 Pi radians in 180 degrees, so dividing by 2 (for example) would give us the equivalent value (in radians) of 90 degrees.

**Aspect** : This is the aspect ratio for the projection matrix. Leave this at 1 unless you're trying to do weird stuff ! experiment setting it to  $<1$  or  $>1$  value (as in 0.1 or 2, not 20000), you'll notice that things look a little bit strange afterwards. In general, less than 1 stretches the geometry vertically, greater than 1 stretches the geometry horizontally.

**zn** : This is the near clipping plane. Any geometry closer than this to the camera will automatically be not-rendered, or clipped so that only the visible area is rendered. Leave this to something like 0.1 for the best results.

**zf** : this is the far clipping plane, as with the near plane, anything beyond this point (from the camera) will not be rendered, or will be clipped. If you are going to have a lot of things on screen it's best to keep this as close as possible – otherwise you'll end up rendering lots and lots of stuff that might not be visible.

The set up used in the example source code is:

```
D3DXMatrixPerspectiveFovLH matProj, PI / 3, 1, 0.1, 75
```

Finally, we've created the matrix structure (matProj in this case) – but Direct3D doesn't necessarily know it exists – so we must tell it. We use the following line, which must only be called AFTER a successful device creation:

```
D3DDevice.SetTransform D3DTS_PROJECTION, matProj
```

From now until the next time this call is made (with a different matrix) all rendering will take place using this matrices configuration – 60 degrees view and a 75 meter draw depth.

## The View Matrix

This is basically for configuring the camera. When we set this up we can tell Direct3D where the camera is, and where it's looking at. Direct3D will then use this matrix in conjunction with the projection matrix to decide what is going to be rendered – and how it's projected onto the screen.

```
D3DXMatrixLookAtLH (  
    MOut as D3DMATRIX,  
    VEye as D3DVECTOR,  
    VAt as D3DVECTOR,  
    VUp as D3DVECTOR )
```

Not too complicated really. The MOut parameter will be the structure that's filled with the relevant information. Veye is the position of the camera – where the player is looking from. Vat is the position of the target – the center of the screen (in 2D) will be this point in 3D Space. Vup is a proper vector (rather than a position) that tells D3D which way is up, using this you could make it render upside down...

The line from the sample code looks like:

```
D3DXMatrixLookAtLH matView, MakeVector(0, 5, 2), MakeVector(0, 0, 0), MakeVector(0, 1, 0)
```

Above we're using a simple inline function that returns a D3DVECTOR based on the parameters, it's quite simple and you can see it in the sample code. The actual code here tells D3D that the camera is place a little bit up and forward of the origin, and looking at the origin, and up is in the positive Y axis. As with the projection matrix you need to tell D3D that you want to use this camera set up for the duration until you change it again:

```
D3DDevice.SetTransform D3DTS_VIEW, matView
```

## The World Matrix

This is a more interesting one – it is quite likely that you’ll set up and reuse this matrix 10 or more times every frame. It also matters how you set it up – whilst there are several helper functions to set a world matrix up a lot of it is still down to you.

First off then, what is a world matrix? The world matrix is applied to all geometry that is rendered, you can have translations (moving it around), scaling (bigger/smaller) and rotations. If you have a 2x scaling matrix applied then all geometry that’s rendered after that point will be twice as big as the actual vertex coordinates.

Secondly, why do we want to use them? The world matrix is an extremely fast and efficient way of manipulating your vertices – rather than you having to rotate and translate (for example) the raw vertex structures you can set up a world matrix and it will do it all for you. You also have the added advantage of being able to manipulate a single set of vertices to be many objects. For example, the cube we’ve made – you could render it in one position with one texture, then alter the world matrix and render it again with a different texture in a different position – and whilst there is only one physical set of vertices the user will see two essentially different cubes on the screen at the same time.

So they seem to be the perfect answer to what we need (they are, that’s why!) –but before we do a quick run through of how to use them I’m going to confuse you (unless you’re a good mathematician). To combine transformation matrices (The world matrix is a combination of 1 or more of these) we multiply them together, and in the normal world of maths  $A \times B = B \times A$ , remember? Well, in matrix land  $A \times B$  does not equal  $B \times A$ . In which case it matters what order we set up our world matrix. Great – things suddenly got complicated.

There are 3 main types of transformation that you will be applying to the world matrix – Rotation, Scaling and Translation. All of them can be applied around one or more of the axis’ – X, Y or Z. Whilst there’s no reason why you can’t change the order, it will tend to be Rotation then Scaling and lastly translations. The reason for translating last is that the other two are calculated about the origin – all vertices are rotated around the point  $[0,0,0]$  and all vertices are scaled from the point  $[0,0,0]$ . Therefore if you translate something to  $[10,0,0]$  first, then rotate it around the Y axis you’ll get the object doing a loop (which might be what you want), but if you rotate it then translate it then you’ll get the object at  $[10,0,0]$  and spinning on the spot.

Sometimes these things are hard to visualise – but experiment with them and see what results you get from different orders. A further analogy could be the car wheel. Imagine a car wheel that, whilst driving normally, rotates around the X axis, if the car takes damage and the wheel is bent out of place we may choose to rotate it a constant 15 degrees around the Z axis (so it appears bent inwards); if we rotate it around the Z axis first then the X axis we’ll get the wheel oscillating in and out at the top and bottom as the wheel goes around – the part that’s bent outwards will rotate around with the wheel. If we rotate around the X axis first then the Z axis the wheel will revolve normally, except appear to be leaning in one direction – depending on which way the Z axis was rotated either the top of the wheel will stick out and the bottom will stick in – or vice versa. Think about it...

Whilst I've already touched upon this, it is often important to make all geometry relevant to the origin. Geometry is rotated around it, scaled about it and translated relative to it – if your geometry is centred around the point [10,0,0] and you translate it by 10 units along the X axis the centre will become [20,0,0]... which is why the cube that we've already generated has all its coordinates + or – 1 unit of the origin.

Before we move onto the actual sample code for these transformations we'll quickly discuss the structure of setting up the world matrix. Matrices are held in a structure called D3DMATRIX, we will have one master matrix and one temporary matrix. The temporary matrix will be reset each time and have a new transformation applied, then it will be combined (through matrix multiplication) with the master matrix. This looks like this:

- 1) Define a Temporary and master matrix
- 2) Set both to the identity matrix
- 3) Rotate the temporary matrix around the X axis
- 4) Multiply the master and temporary matrix
- 5) Set the temporary matrix to the identity matrix
- 6) Rotate the temporary matrix around the Y axis
- 7) Multiply the master and temporary matrix
- 8) Set the temporary matrix to the identity matrix
- 9) Rotate the temporary matrix around the Z axis
- 10) Multiply the master and temporary matrix
- 11) Set the temporary matrix to the identity matrix
- 12) Scale the temporary matrix by any amount
- 13) Multiply the master and temporary matrix
- 14) Set the temporary matrix to the identity matrix
- 15) Translate the temporary matrix
- 16) Multiply the master and temporary matrix
- 17) Commit the master matrix to the device so that it is applied to all subsequent rendering.

Not too complicated really – you'll pick it up really quickly. There are two things in the above that I haven't yet covered. The first is the identity matrix, when you create a new matrix type it will have 4x4 singles set out with a value of 0. If you made Direct3D use this matrix nothing would appear – the scale is set to 0, so at most all you'd see is a coloured dot at the origin. The identity matrix is one where the scaling values are set to 1.0, and any geometry rendered using it will not be altered at all – it appears in 3D space as it was laid out in the raw vertex data. It is important to reset the temporary matrix each time – otherwise you can get it accumulating multiple transformations in a strange way and you'll get some truly strange results! The second thing is the multiplication part. As initially mentioned,  $[A][B]$  does not equal  $[B][A]$ , so this line has to be in a specific order – you can get slightly different effects depending on which way around it's done, but I strongly suggest you choose a preferred order and stick with it – I have always multiplied the world matrix by the temporary matrix – it's worked fine for me and should do for

you. The same goes with what order you rotate it – as mentioned in the wheel analogy, XYZ is very different from ZXY (or any other combination), Unless I need it to be done in another way I stick with XYZ – you can use that or change it – it’s up to you...

Okay... So I do believe we’re now ready to look at some proper code (at last!), the sample project only does rotation – you can see it as a small challenge to rewrite the sample so that the cube is translated and scaled (perhaps based on input from the user)...

There are 7 functions to perform the translations – there are lots more, but I’ll let you explore them later on (they tend to be more specialised). These are:

```
D3DXMatrixIdentity(MOut As D3DMATRIX)
D3DXMatrixMultiply(MOut As D3DMATRIX, M1 As D3DMATRIX, M2 As D3DMATRIX)
D3DXMatrixRotationX(MOut As D3DMATRIX, angle As Single)
D3DXMatrixRotationY(MOut As D3DMATRIX, angle As Single)
D3DXMatrixRotationZ(MOut As D3DMATRIX, angle As Single)
D3DXMatrixScaling(MOut As D3DMATRIX, x As Single, y As Single, z As Single)
D3DXMatrixTranslation(MOut As D3DMATRIX, x As Single, y As Single, z As Single)
```

Not too complicated really, you’ll see how to use them in a second, they all follow the same format. MOut is the resulting matrix, and angle is always in radians (not degrees).

```
D3DXMatrixIdentity matWorld

D3DXMatrixIdentity matTemp
D3DXMatrixRotationX matTemp, Angle * RAD
D3DXMatrixMultiply matWorld, matWorld, matTemp

D3DXMatrixIdentity matTemp
D3DXMatrixRotationY matTemp, Angle * RAD
D3DXMatrixMultiply matWorld, matWorld, matTemp

D3DXMatrixIdentity matTemp
D3DXMatrixRotationZ matTemp, Angle * RAD
D3DXMatrixMultiply matWorld, matWorld, matTemp

Angle = Angle + 1

D3DDevice.SetTransform D3DTS_WORLD, matWorld
```

The above excerpt is the code used in the sample program to spin the cube around on all axis by 1 degree every frame. This code is updated every frame – so any particularly complicated formula’s that you want to use may well slow things down a bit – although using the D3DX\* functions are fast enough not to worry about.

If you now run the sample project (or your code if copying from the page) you should be greeted with a fairly large, colourful spinning cube... If you have turbo-charged hardware acceleration the cube may be spinning extremely fast (1 degree every frame, 360+ fps will mean it does a complete spin in 1 second!), if this is the case change the angle increment value down to 0.5 or 0.25...

## Vertex Buffers and Index Buffers.

Well, that was one rather large 7 A4 page section done with, things should fly by a little quicker now that the basic theory has been covered. As with most things, once you've jumped over the initial couple of hurdles (setting up D3D, basic 3D geometry) it's all easy – well, sort of ☺

This section is dealing with enhancing your capabilities for rendering geometry, making things easier and faster. Which I'm sure you'll appreciate...

The first stop will be vertex buffers – what are these? I hear you asking... A vertex buffer is an area of memory where you can store your vertex data, this has the advantage of being easier to render – both for you and for the driver (it can perform various optimisations, and access the data faster) and allowing you to archive certain pieces of geometry, a generic cube for example... Whilst not entirely true it can also stop most accidental overwriting/corrupting errors – once the buffer is created it's not something you can accidentally change without realising it – whereas an array of vertex structures as a standard variable could easily be overwritten by a rogue function or library (however unlikely you think it, it will happen sometime).

Creating a vertex buffer is extremely simple once you have the basics from the last section under your belt. At the simplest level it involves copying the data you generated to a custom buffer...

The first part is allocating some space for our buffer:

```
'In Declarations section
Dim vbCube As Direct3DVertexBuffer8

'at the end of InitialiseGeometry() sub
Set vbCube = D3DDevice.CreateVertexBuffer(Len(CubeVerts(0)) * 36, 0, FVF_LVERTEX, D3DPOOL_MANAGED)
If vbCube Is Nothing Then Debug.Print "ERROR: Could not create vertex buffer": Exit Sub
```

Not too complicated really, first we create the buffer with the function built into the Direct3DDevice8 class, then we check to make sure it was created successfully – if it wasn't it usually generates an error on that line. The parameters for the CreateVertexBuffer() function look like:

```
CreateVertexBuffer( _
    LengthInBytes As Long, _
    Usage As Long, _
    FVF As Long, _
    Pool As CONST_D3DPOOL) As Direct3DVertexBuffer8
```

**LengthInBytes** : This is the amount of memory that will be set aside for your buffer, it must be at least as big as the data you want to store in it – any less and you'll get an error when copying the data in, any more and you'll be wasting space. This value can be computed using the Len() function to find the size of one vertex structure, then multiplying it by the number of structures that your using.

**Usage** : This can be left as 0 for most uses, but more specialised cases require that you let Direct3D know that your going to be doing different things with this vertex

buffer. For example, specifying D3DUSAGE\_DYNAMIC will tell Direct3D to let the driver know that we're probably going to be changing the contents around quite a lot – it will then choose the most optimal place in memory to put it. If you don't specify this, the buffer will be considered static – and the driver will place it in the most optimal memory location for that use – which will mean that any lock-write-unlock operations are slower (it assumed that you weren't likely to be doing any).

**FVF** : This lets Direct3D know what sort of vertices you will be storing here, should you ask D3D to perform any operations on this buffer later on it will base them on this flag. Set this to be the same as the one you use for rendering.

**Pool** : This allows you to tell D3D where you would like the vertex buffer to be stored, note that it's where you would LIKE it to be – The driver may well override this setting – should it be impractical (lack of space/lack for support for example). The 3 options are: D3DPOOL\_DEFAULT – it's left mostly up to the driver to choose where it goes – usually in video memory, but other places if the usages flags suggest otherwise. D3DPOOL\_MANAGED – D3D will store a master copy in system memory and will copy the buffer to video memory as and when its required. D3DPOOL\_SYSTEMMEM – the buffer will be stored in system memory, which isn't usually accessible by the 3D device, and it will tend to be quite slow.

Now that we've created our vertex buffer we want to put some stuff in it! For this example we'll copy our cube vertices into the buffer and then render it from there. For this we use a little helper function to copy the data:

```
D3DVertexBuffer8SetData vbCube, 0, Len(CubeVerts(0)) * 36, 0, CubeVerts(0)
```

That call will fill the specified vertex buffer with the specified amount of information from the array provided. A quick run through of the parameters and what they mean then:

```
D3DVertexBuffer8SetData( _  
    VBuffer As Direct3DVertexBuffer8, _  
    Offset As Long, _  
    Size As Long, _  
    Flags As Long, _  
    Data As Any) As Long
```

**Vbuffer** : obviously this is the vertex buffer object that you want the data put into.

**Offset** : The offset in bytes from the beginning of the buffer that we start placing new data – useful if your only wanting to change vertices 10-19 (for example).

**Size** : The amount of data that's going to be copied in, also the amount that D3D expects to find in the data array. This is the same calculation as used in creating the buffer. Try to make sure beforehand that this doesn't exceed the data source size, or the remaining/total space in the vertex buffer

**Flags** : A set of flags defining how you want the data to be replaced/copied – look up CONST\_D3DLOCKFLAGS in the object browser for more details.

**Data** : This is the source array, it can be of any data type – whilst you could trick D3D into storing any type of data it's supposed to be vertex data only (make sure you don't accidentally include any state variables/non-vertex variables).

Okay, you now have all the information required to create a vertex buffer – but currently you can't actually do anything with it. For this sample all we'll do is render it – there are a few other things you can do, but I'll let you explore those yourself...

```
'replace:  
D3DDevice.DrawPrimitiveUP D3DPT_TRIANGLELIST, 12, CubeVerts(0), Len(CubeVerts(0))  
  
'With:  
D3DDevice.SetStreamSource 0, vbCube, Len(CubeVerts(0))  
D3DDevice.DrawPrimitive D3DPT_TRIANGLELIST, 0, 12
```

As you can see, rendering gets considerably easier when using vertex buffers – tell it where the vertex buffer is, and then what part you want rendering. As a side note, the stream number (The first parameter in SetStreamSource) should be left as 0 in most cases – things such as vertex shaders manipulate these further.

Now that you have vertex buffers under your belt we can take a look at index buffers, these require vertex buffers (not technically true, but I'll explain later), which is why they came last. In fact, this is the last section in this article – we are definitely homeward bound...

If you look at our cube geometry you'll notice that it's extremely inefficient – we use 36 vertices to describe an object that (in real life) only has 8 vertices. Wouldn't it be extremely useful if we could only place 1 vertex in each corner, rather than have 3-6 vertices sharing the same coordinate, whilst not always the case, in our example they have the same colour as well – several vertices that are all identical.

This is where indices and index buffers come into play – they allow us to use vertices more than once (to put it simply). We can create 8 vertices, and then, using an index list say that triangle 1 uses vertices 1,2 and 3 and triangle 2 uses vertices 2,4 and 3 – reusing vertices. This has the advantage of being relatively fast, and also saves on the amount of data involved – the index list will always be smaller than the vertex list, and therefore the amount of data that's processed and copied also decreases.

Unfortunately they aren't quite the miracle invention I'm making them out to be – two topics that we haven't yet discussed: Lighting and textures can become significantly worse/harder when using indices – it's not always the case, but in cases such as the cube example it would be very annoying. The exact reasoning will be discussed in the relevant sections later on...

To set up and use indices/index buffers we first need to create a vertex buffer with the key vertices in it – the following code does this for the cube sample:

'In the declarations section:

```
Dim vbCubeIdx As Direct3DVertexBuffer8
```

```
Dim ibCube As Direct3DIndexBuffer8
```

```
Dim vList(0 To 7) As LITVERTEX 'the 8 vertices required
```

```
Dim iList(0 To 35) As Integer 'the 36 indices required (note that the number is the same as the vertex count in the previous version).
```

'in the create geometry section:

```
vList(0) = CreateLitVertex(-1, -1, -1, &HFFFFFF, 0, 0, 0)
```

```
vList(1) = CreateLitVertex(-1, -1, 1, &HFF0000, 0, 0, 0)
```

```
vList(2) = CreateLitVertex(-1, 1, -1, &HFF00, 0, 0, 0)
```

```
vList(3) = CreateLitVertex(-1, 1, 1, &HFF, 0, 0, 0)
```

```
vList(4) = CreateLitVertex(1, -1, -1, &HFF00FF, 0, 0, 0)
```

```
vList(5) = CreateLitVertex(1, -1, 1, &HFFF00, 0, 0, 0)
```

```
vList(6) = CreateLitVertex(1, 1, -1, &HFFF, 0, 0, 0)
```

```
vList(7) = CreateLitVertex(1, 1, 1, &HFF8000, 0, 0, 0)
```

```
Set vbCubeIdx = D3DDevice.CreateVertexBuffer(Len(vList(0)) * 8, 0, FVF_LVERTEX, D3DPOOL_MANAGED)
```

```
If vbCubeIdx Is Nothing Then Debug.Print "ERROR: Could not create vbCubeIdx": Exit Sub
```

```
D3DVertexBuffer8SetData vbCubeIdx, 0, Len(vList(0)) * 8, 0, vList(0)
```

I'm not going to cover this part again – reread the previous section if you're still unclear as to setting up a vertex buffer.

The next part is the new part, we must set up an index list. All this requires is that we fill an array of 16 bit integers (you can use 32 bit integers if you need to) and then copy them to the new array. We'll be rendering our cube using a triangle list again – so we're going to need to describe 12 triangles again, because of this we can pretty much copy exactly what we did for the original triangle. This is what it's going to look like:

'top

```
iList(0) = 2: iList(1) = 6: iList(2) = 3
```

```
iList(3) = 6: iList(4) = 7: iList(5) = 3
```

'bottom

```
iList(6) = 0: iList(7) = 4: iList(8) = 1
```

```
iList(9) = 4: iList(10) = 5: iList(11) = 1
```

'left

```
iList(12) = 2: iList(13) = 3: iList(14) = 0
```

```
iList(15) = 3: iList(16) = 1: iList(17) = 0
```

'right

```
iList(18) = 6: iList(19) = 7: iList(20) = 4
```

```
iList(21) = 7: iList(22) = 5: iList(23) = 4
```

'front

```
iList(24) = 3: iList(25) = 7: iList(26) = 1
```

```
iList(27) = 7: iList(28) = 5: iList(29) = 1
```

'back

```
iList(30) = 2: iList(31) = 6: iList(32) = 0
```

```
iList(33) = 6: iList(34) = 4: iList(35) = 0
```

I've grouped them together into their triangles – each line is a triangle, and there are two triangles to each face – so two lines under each comment/heading. Hopefully it's all fairly explanatory; if we take the first triangle as an example, it uses the vertices 2,6 and 3 to be rendered, these are in the standard clockwise order.

The next part we need to cover is creating an index buffer, and then copying the data into it. Luckily this is almost identical to the creation of a vertex buffer – many of the parameters are the same. To copy our index list into an index buffer we use this piece of code:

```
Set ibCube = D3DDevice.CreateIndexBuffer(Len(iList(0)) * 36, 0, D3DFMT_INDEX16, D3DPOOL_MANAGED)
If ibCube Is Nothing Then Debug.Print "ERROR: Could not create the index buffer": Exit Sub

D3DIndexBuffer8SetData ibCube, 0, Len(iList(0)) * 36, 0, iList(0)
```

Straight away you can see the similarities; the only major different when creating the index buffer is that we need to specify what format the indices are going to be in. There are only two valid options here (despite it listing 30-40 formats): D3DFMT\_INDEX16 and D3DFMT\_INDEX32, these refer to the actual data type that we use to store our indices, a 16 bit integer (The integer data type) can only hold up to 32,767 indices (the maximum + value for an integer), and a 32 bit integer (the long data type) can only hold up to 2,147,483,647 (the maximum + value for a long). Obviously using 32 bit indices take up double the amount of space (4 bytes per value, rather than 2 bytes per value in 16 bit), so unless you are using an extremely large number of indices you should try and stick to using 16 bit indices.

Now that we have our index data stored in a buffer we need to be able to do something with it, for this example all we're going to do is render it – there is very little else you can do with them anyway. The following code segment will render our cube using our vertex buffer/index buffer combo:

```
D3DDevice.SetStreamSource 0, vbCubeIdx, Len(vList(0))
D3DDevice.SetIndices ibCube, 0
D3DDevice.DrawIndexedPrimitive D3DPT_TRIANGLELIST, 0, 36, 0, 12
```

All it involves is setting the current index buffer and vertex buffer – make sure they match up though, then using DrawIndexedPrimitive() to render from them. The parameters are fairly straight forward for the rendering call, you can use the third and fourth to set a range that you want rendering (12-24 for example), you can use the second parameter to make an offset in the vertex buffer (all values in the index list get this number added to them, if it were 10, and iList(3) was 12 the actual vertex used would be 22). The last value states how many primitive we're rendering – in this case the whole cube, 12 triangles.

There are two final things to be covered about the usage of indices and buffers; the first one is about rendering without using the buffers. It is perfectly possible to render our indexed cube without storing the index/vertex data in a buffer, but it is often a great deal slower. The call for doing this is quite a lengthy one – you need to specify the details of both the vertex and index buffers:

```
D3DDevice.DrawIndexedPrimitiveUP D3DPT_TRIANGLELIST, 0, 8, 12, iList(0), D3DFMT_INDEX16, vList(0), Len(vList(0))
```

A very quick summary of what goes where:

**PrimitiveType** : What type of primitives we're rendering

**MinVertexIndex** : The Starting vertex to render from

**NumVertexIndices** : The number of vertices that we're going to render.

**PrimitiveCount** : The number of triangles that we are rendering

**IndexDataArray** : The first element in the array storing indices

**IndexDataFormat** : What format the index data is in, 16 or 32 bit

**VertexStreamZeroDataArray** : The first element of the vertex data array

**VertexStreamZeroStride** : The size of one vertex element

If you have the SDK help file you may realise that the 2<sup>nd</sup> and 3<sup>rd</sup> parameters have different names – This is a typing error in the help file, the parameters I've listed above come straight from the data-tip that appears when you type the line into VB.

### Rendering Summary

Over the course of this article we have covered a total of 4 different ways of rendering our cube:

```
'##RENDERING METHOD 1##  
D3DDevice.DrawPrimitiveUP D3DPT_TRIANGLELIST, 12, CubeVerts(0), Len(CubeVerts(0))  
  
'##RENDERING METHOD 2##  
D3DDevice.SetStreamSource 0, vbCube, Len(CubeVerts(0))  
D3DDevice.DrawPrimitive D3DPT_TRIANGLELIST, 0, 12  
  
'##RENDERING METHOD 3##  
D3DDevice.SetStreamSource 0, vbCubeIdx, Len(vList(0))  
D3DDevice.SetIndices ibCube, 0  
D3DDevice.DrawIndexedPrimitive D3DPT_TRIANGLELIST, 0, 36, 0, 12  
  
'##RENDERING METHOD 4##  
D3DDevice.DrawIndexedPrimitiveUP D3DPT_TRIANGLELIST, 0, 8, 12, iList(0), D3DFMT_INDEX16, vList(0), Len(vList(0))
```

With these 4 methods, and the knowledge of how to set them up you will be perfectly capable of generating most forms of geometry in the best possible way. Experiment with different complexity models to see what advantages of space/speed you can find...

Finally, absolutely finally, I want to explain how to get data from the buffers after you have put it there. Whilst it is quite likely you'll have the original vertex structures around to manipulate there will be times when you need to access the vertex or index data and change it. It is through this method that you can do key frame animation (I have a tutorial on my site about this – see the link in the summary). Also when using indexed rendering you need only change the vertex in the vertex buffer for it to be instantly reflected in the final rendering – all indices that point to that vertex will use the updated copy.

```

D3DIndexBuffer8GetData( _
    IBuffer As Direct3DIndexBuffer8, _
    Offset As Long, _
    Size As Long, _
    Flags As Long, _
    Data As Any) As Long

D3DVertexBuffer8GetData( _
    VBuffer As Direct3DVertexBuffer8, _
    Offset As Long, _
    Size As Long, _
    Flags As Long, _
    Data As Any) As Long

```

As you can see both functions are almost identical – the only difference being their name and the first parameter. A quick run through of the common parameters then:

**Offset** : offset in bytes indicating where the data should be read from

**Size** : The size of the destination buffer in bytes

**Flags** : Not relevant in 99% of cases, examine CONST\_D3DLOCKFLAGS if you think you need something special

**Data** : This is the first element in an array that is going to store the data – it must be in the correct format, and must be the correct size.

Should there be any problem with knowing how big the vertex/index buffer is – or how many vertices/indices are stored inside then you can use the .GetDesc member of either the vertex buffer or index buffer. This will retrieve either a D3DVERTEXBUFFER\_DESC or a D3DINDEXBUFFER\_DESC structure, you can then use the data provided to work out the size, in the case of index buffers the following code will tell you how many indices there are in the buffer:

```

'D3DFMT_INDEX16 = 101
'D3DFMT_INDEX32 = 102
Dim ibDesc As D3DINDEXBUFFER_DESC
Dim IndexCount As Long 'how many indices are in the buffer

ibCube.GetDesc ibDesc
If ibDesc.Format = 101 Then
    '16 bit indices
    IndexCount = ibDesc.Size / 2
ElseIf ibDesc.Format = 102 Then
    '32 bit indices
    IndexCount = ibDesc.Size / 4
Else
    'no idea whats stored here!
End If

Debug.Print IndexCount, " indices in the buffer."

```

And this code will tell you how many vertices there are in a vertex buffer:

```
Dim vbDesc As D3DVERTEXBUFFER_DESC
Dim DummyVertex As LITVERTEX
Dim VertexCount As Long
Dim TotalDivider As Long 'the size of the vertex structure

vbCube.GetDesc vbDesc

If vbDesc.FVF = FVF_LVERTEX Then
    'The type stored is the LVertex type
    VertexCount = vbDesc.Size / Len(DummyVertex)
Else
    'it's some other type of vertex, lets find out:
    If Not (vbDesc.FVF And D3DFVF_XYZ) = 0 Then
        Debug.Print "D3DFVF_XYZ"
        TotalDivider = TotalDivider + 12
    End If

    If Not (vbDesc.FVF And D3DFVF_XYZRHW) = 0 Then
        Debug.Print "D3DFVF_XYZRHW"
        TotalDivider = TotalDivider + 16
    End If

    If Not (vbDesc.FVF And D3DFVF_NORMAL) = 0 Then
        Debug.Print "D3DFVF_NORMAL"
        TotalDivider = TotalDivider + 12
    End If

    If Not (vbDesc.FVF And D3DFVF_DIFFUSE) = 0 Then
        Debug.Print "D3DFVF_DIFFUSE"
        TotalDivider = TotalDivider + 4
    End If

    If Not (vbDesc.FVF And D3DFVF_SPECULAR) = 0 Then
        Debug.Print "D3DFVF_SPECULAR"
        TotalDivider = TotalDivider + 4
    End If

    If Not (vbDesc.FVF And D3DFVF_TEX1) = 0 Then
        Debug.Print "D3DFVF_TEX1"
        TotalDivider = TotalDivider + 8
    End If

    If Not (vbDesc.FVF And D3DFVF_TEX2) = 0 Then
        Debug.Print "D3DFVF_TEX2"
        TotalDivider = TotalDivider + 8
    End If

    VertexCount = vbDesc.Size / TotalDivider
End If

Debug.Print VertexCount, " Vertices in the buffer"
```

Quite lengthy you'll see, but that's just if the format is unknown, in which case we undo the "or-ing" to generate an FVF by "And-ing" it with the relevant components – the ones listed above will detect most common types, but if you start using more dynamic flags then you'll need to add them to this list.

## **SUMMARY**

Well, I've finally completed this article! All 20 A4 pages of it and nearly 10,000 words of it, it even took me almost dead on 7 hours to write the whole thing... and I haven't made a penny/dollar from it ☺ Maybe I should write a book about this stuff...

Once you have this massive amount of information safely behind you then you should be perfectly capable of getting started with your own game, the next article will tidy up all the loose ends and introduce you to loading models and direct3D lighting, by which point you'll be a fairly competent DirectXGraphics developer.

Please feel free to send any comments, criticisms and questions to me: [Jack.Hoxley@DirectX4VB.com](mailto:Jack.Hoxley@DirectX4VB.com) , I always like hearing from people who've read my work...

Also, you can visit my main page – where a lot of this information can be found in different formats/examples, along with plenty of other articles (over 100 in total), go along to [www.DirectX4VB.com](http://www.DirectX4VB.com) and have a look around...

Till next time...

# DirectXGraphics For Visual Basic Part 3

Welcome back for part 3 in my 3 part series on DirectXGraphics... First off, I'm sorry for the rather long delay between parts 2 and 3 (8 months or so!), I've just been phenomenally busy lately. Hopefully to make up for this gap I'll cover everything you need to know to give you a strong foundation in Direct3D8 programming. Direct3D9 / DirectX9 is about to enter it's first beta-testing stage, so it may seem that DX8/D3D8 is getting a little old (it's a 1 year old API now), but you would be foolish to think like that. From what I've seen here-and-there about D3D9 it seems to be not much more than an extension of D3D8, whereas v7 to v8 was a big jump, v8 to v9 is more of a revision. Also, D3D9 wont be much use for a long time yet as there will be very little hardware to support its new features, and very few end-users owning this hardware. Direct3D8's revolutionary pixel/vertex shader technology only exists on 2 or 3 cards (GeForce 3 / 4 and the ATI Radeon's) and isn't really being that extensively used yet, so if we haven't even caught up with that properly, why do we need Direct3D9...?

I'll stop moaning now, and get on with what this article is actually supposed to be about. Part 2 was quite a complicated and fast paced article, and don't expect any let-off just yet - I'm going to be keeping up the pace for this article too. This is the outline:

- 1) Using textures in Direct3D
- 2) Loading 3D Models from files
- 3) Using the Direct3D lighting engine

Doesn't look like much does it? Haha, more fool you if that's what your thinking. These 3 topics alone deserve an article (or two) each... less talk, more learning!

## Using Textures in Direct3D

So far we've seen some basic 3D geometry – a spinning cube, you should be aware that the colour of the vertices depicts what colours you actually see when it's finally rendered. Yet you should also be aware that you can't really do more than create pretty-coloured gradients with it. Say we want to turn our 3D Box into something more interesting – say a wooden crate for example.

I don't think I need to tell you that it's almost impossible to create a decent wooden box appearance using just vertices and their colours. So we're going to use a bitmap to display the colours. As you should be aware, 3D geometry is made up of triangles, and a simple fact of a triangle defined in 3 dimensions is that it is planar – a 2D surface that doesn't have curves or anything like that. Thus it is perfectly suited for projecting a 2D bitmap image onto. This is the basis of texturing – we use a 2D bitmap applied to the 3D triangles in order to make the overall model appear to look more detailed / look like something.

The first step to using textures is to load the texture from the hard-drive / CD-ROM into texture memory. This causes one slight complication already – texture memory is a finite resource, yet art-work tends to happily consume an infinite amount of space! Therefore we can only fit a potentially small amount of art work into memory at any one time. This amount is indicated by the amount of memory the graphics card has "onboard". 32mb is common these days, with 16mb being a past favourite and 64mb being standard on all the new high-tech boards. It is quite easy to work out how much space you are using – based on the internal texture format and the dimensions of the texture itself, also dependent on any additional space required for mip-mapping.

I discussed the `CONST_D3DFORMAT` enumeration in the previous article – what the letters mean, what the numbers are for... if you can't remember that then go read the previous article. As you are aware, a bitmap is made up of a 2D grid of pixels, we need to use the `CONST_D3DFORMAT` enumeration to tell Direct3D how to store the colour for each of those pixels – 32 bit, 16 bit... As you should be aware, 32 bit = 4 bytes, 16 bits = 2 bytes. If we store a standard 256x256 bitmap with 32 bits per pixel we'll need 256kb of texture memory, however, if we store it at 16 bits per pixel we'll only need 128kb of texture memory. This may seem fairly trivial for only one texture – and it is; but if you have 200 textures it's the difference between 50mb and 25mb – suddenly it means a lot more! 25mb will fit into most recent video cards, 50mb will only fit into the (current) top of the range 3D cards. The bottom line being that you must be clever with your choice of texture format. As a general note, you will tend to find that your game runs much faster if the display mode format and the texture formats are the same – as it saves any last minute format conversions from being done (which is just a tiny bit more work to be done).

This following little piece of code will allow you to check what texture formats can be used by the currently installed 3D board. The last parameter (`D3DFMT_X8R8G8B8` in this case) indicates the texture format you want to test.

```

If D3D.CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, _
    D3DWindow.BackBufferFormat, 0, D3DRTYPE_TEXTURE, _
    D3DFMT_X8R8G8B8) = D3D_OK Then

    Debug.Print "32 Bit textures with no alpha are supported"
End If

```

The other rule for textures is their size. Whilst it's not so important with new 3D cards, it is very important if you want to be backwards compatible. It's also generally much faster to stick to using the old style texture size conventions.

1. Stick to using 2<sup>n</sup> texture dimensions, that is 2,4,8,16,32,64,128,256 and so on... anything above 256x256 is getting a little risky – the very popular Voodoo3 chipset doesn't support textures above 256x256 in size, which instantly causes a problem with compatibility. 256x256 is also the optimal size for a texture and tends to give the best all round performance.
2. Textures don't have to be square. This may well cause some problems with very, very old graphics, but we can't be compatible with everyone now...
3. Where possible, group small textures onto one larger texture, this is known as texture-paging sometimes. For example 64 32x32 tile pictures will be okay as 64 different textures, but it'll run much faster to store them all as one 256x256 texture.

This following piece of code retrieves the maximum texture sizes available to the device:

```

D3D.GetDeviceCaps D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, DevCaps
Debug.Print "MAX TEX SIZE: ", DevCaps.MaxTextureWidth & "x" & DevCaps.MaxTextureHeight

```

Enough talking now, lets load a texture into memory. Textures are stored in a *Direct3DTexture8* object, and can be loaded using one of two main functions (provided by the D3DX8 library):

```

CreateTextureFromFile( _
    Device As Direct3DDevice8, _
    SrcFile As String) As Direct3DTexture8

CreateTextureFromFileEx( _
    Device As Direct3DDevice8, _
    SrcFile As String, _
    Width As Long, _
    Height As Long, _
    MipLevels As Long, _
    Usage As Long, _
    Format As CONST_D3DFORMAT, _
    Pool As CONST_D3DPOOL, _
    Filter As Long, _
    MipFilter As Long, _
    ColorKey As Long, _
    SrcInfo As Any, _
    Palette As Any) As Direct3DTexture8

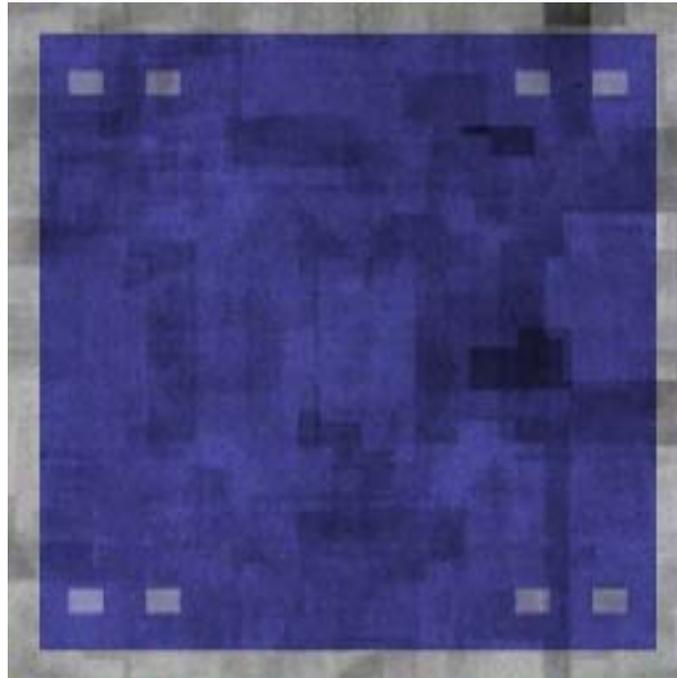
```

As you can see, the first function is much simpler than the second. This is deliberate – sometimes you really don't need that much control over the texture creation process. However, I strongly suggest that you get used to using the second function from square 1. Many of the parameters are fairly simple and don't change much between different uses. The following code is the fairly general implementation:

```
Set CubeTex = D3DX.CreateTextureFromFileEx(D3DDevice, _
    App.Path & "\cube_tex.jpg", _
    256, 256, 1, 0, _
    DispMode.Format, D3DPOOL_MANAGED, _
    D3DX_FILTER_LINEAR, D3DX_FILTER_LINEAR, _
    0, ByVal 0, ByVal 0)
```

Looks a little complicated doesn't it. Well, the first parameter associates the texture with our device (simple enough), the second parameter points to the file where the data is stored (BMP, TGA, JPG allowed). The third and fourth textures indicate the size of the texture in memory, if the file is of different dimensions then D3DX will resize it for you. The fifth and sixth parameters indicate the mip-map levels and the usage – leave these both to 0 in most cases, although in this case I've set the MipLevels parameter to be 1 – I only want one iteration in the mipmap sequence... if I let it do more (setting it to 0 indicates a full chain) then it'll start chewing up my memory! The seventh parameter indicates the format of the texture, as I said earlier, keeping it the same as the device format is best – so that's what I've done. The eighth parameter indicates the memory pool – managed (copied to video memory when needed/moved back to system memory when not needed), default (lets the driver decide where it should go) and system memory (stores it in system memory, which isn't accessible to the 3D Device, yet can be used for some other functions). The ninth parameter and tenth parameters indicate how D3DX filters the input data to fit the memory data – should the two sizes be different; D3DX\_FILTER\_LINEAR will do fine here unless you're resizing the image by more than 2x or 3x the original. The eleventh parameter is the colorkey and isn't being used here – but will be explained later. The last two parameters aren't really that interesting and have been known to cause errors on some systems – so leave them as ByVal 0 unless you really need to.

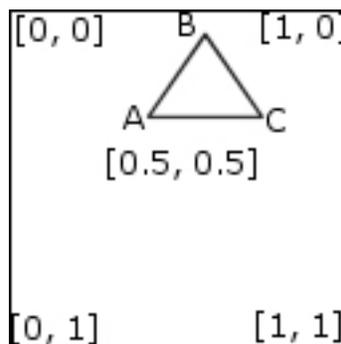
The previous code will now have loaded the following image into texture memory:



Cube\_Tex.Jpg

it's not an amazingly interesting texture, but it'll look alright on our box. The next thing that I need to discuss is texture coordinates.

Texture coordinates are a fun topic, well actually, they're not – because you either get it or you don't; if you don't then you're screwed! I'm only going to go over it briefly here – hopefully you will follow, otherwise, ask some people in the forums on this site, or go in search of some other more in-depth texturing tutorials. The following diagram is required for reference:



The above diagram can be imagined as the Cube\_Tex.jpg shown above. You should be familiar with coordinates in a normal 2D image – X and Y, measured in pixels. We now replace this coordinate system with a scalar system – all pixels are referred to on a 0.0 to 1.0 scale; this is unaffected by the actual pixel dimensions – 256x256 or 128x256, it doesn't matter – they both still use the 0.0 to 1.0 scale. This makes things surprisingly easier actually. Both for us, and for the 3D accelerator. It means

that we can interchangeably use different sized textures (a low-res version and high-res version) with the same piece of code, and expect to get an almost identical result. It also makes it much easier to algorithmically generate texture coordinates (a bit more advanced).

In the above diagram the four corners are labelled with their respective coordinates. I've also drawn a simple triangle on the diagram marked with three vertices, A B and C. At a guess, I'm thinking that A will have coordinates of [0.4,0.3], B will be [0.6,0.1] and C will be [0.75,0.3] – it's only a rough guess, and you could calculate it exactly if you wanted... but I didn't! If this new coordinate system really confuses you still you can use a simple conversion formula:  $(1/\text{Width}) * X$ ,  $(1/\text{Height}) * Y$ , where width, height, x and y are all pixel measurements. On a final note, texture coordinates are usually denoted using U,V and W rather than X,Y and Z – however  $U=X$ ,  $V=Y$ ,  $W=Z$ . It is advised to stick to convention so that other people understand what you're doing.

Now that we've covered loading textures and their coordinate system we can actually try rendering something with it! I'm going to use the cube from the second part of this series – the one with no indices and no vertex/index buffers. There is a good reason for this – I want each vertex to have it's own, different, texture coordinate. This gets difficult when using indices, as in the case of the cube, 3 sides share each vertex, between 3 and 6 triangles as well; therefore I'd need to express up to 6 texture coordinates as a single coordinate – not easy, or in this case, just not possible. Therefore I'm going to have to use the lots-of-vertices cube.

The first step is to assign texture coordinates to each of the vertices, I've only copied out the code for the first face, because it's identical for the other 5, and only takes up lots of space:

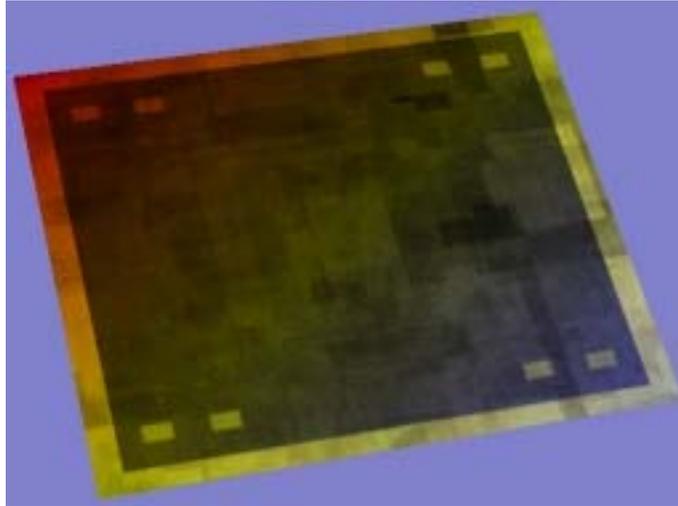
```
CubeVerts(0) = CreateLitVertex(-1, 1, -1, Corner010, 0, 0, 0)
CubeVerts(1) = CreateLitVertex(1, 1, -1, Corner110, 0, 1, 0)
CubeVerts(2) = CreateLitVertex(-1, 1, 1, Corner011, 0, 0, 1)

CubeVerts(3) = CreateLitVertex(1, 1, -1, Corner110, 0, 1, 0)
CubeVerts(4) = CreateLitVertex(1, 1, 1, Corner111, 0, 1, 1)
CubeVerts(5) = CreateLitVertex(-1, 1, 1, Corner011, 0, 0, 1)
```

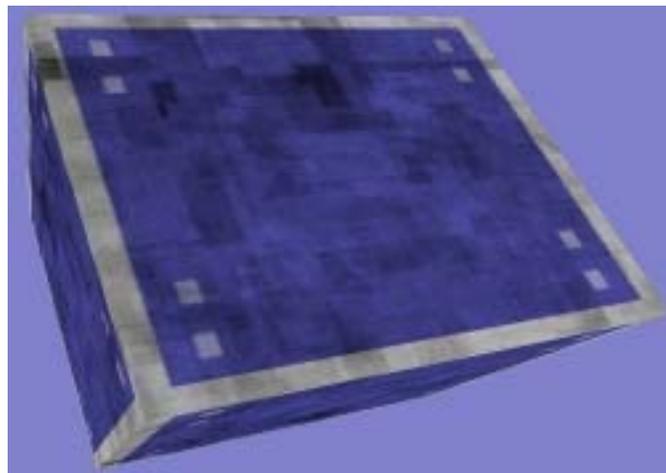
*The parameters in bold are the two texture coordinates.*

The second step is to actually render the cube with the texture applied – which is actually very very easy.

The final result looks like this:



Hmm, so what's gone wrong here then? It's red and yellow? Not much like the picture of the texture above... well, actually, nothing has gone wrong – you've just seen the effects of Direct3D lighting. As you may remember, the original cube geometry had 8 different colours for the corners, well it's these colours that are blending with the texel data to form the final rendered image. This can be used to create brilliant effects – as we shall see later on in the lighting section. If we replace the vertex colours with white then the original texture wont be affected at all – and you'll get an image like this next one:



Which probably looks much more like what you expected.

Okay, so that's texturing covered. Well, as much as you need for a foundation. There is literally tonnes and tonnes more to learn about texturing – but leave it alone till you have this part sorted out in your head. The main areas of advanced texturing come under these headings:

**Alpha channel effects** – opacity/transparency effects in textures.

**Altering pixel data** - generating procedural textures, or applying per-pixel effects.

**Compression** – by default the D3DFMT\_DXT\* formats.

**The Texture cascade** – where you can apply up to 8 textures to each triangle – capable of creating some stunning effects (bump mapping, specular lighting to name two).

**Pixel shaders** – very, very advanced texture effects – brand new to Direct3D8, and quite likely to become a big part of Direct3D 9 and 10...

To get you started on tutorials for some of those features I have the following links:  
<http://www.ancientcode.f2s.net> - has a good article on binary manipulation, and it's uses in altering pixel data.

[http://www.DirectX4VB.com/Tutorials/DirectX8/GR\\_Lesson12.asp](http://www.DirectX4VB.com/Tutorials/DirectX8/GR_Lesson12.asp) - a tutorial on the texture cascade and the effects it presents.

[http://www.DirectX4VB.com/Tutorials/DirectX8/GR\\_Lesson14.asp](http://www.DirectX4VB.com/Tutorials/DirectX8/GR_Lesson14.asp) - a tutorial on accessing and manipulating texture memory.

[http://www.DirectX4VB.com/Tutorials/DirectX8/GR\\_Dot3Bump.asp](http://www.DirectX4VB.com/Tutorials/DirectX8/GR_Dot3Bump.asp) - a tutorial on using the texture cascade to do Dot-3 bump mapping.

## **Loading 3D Models into Direct3D**

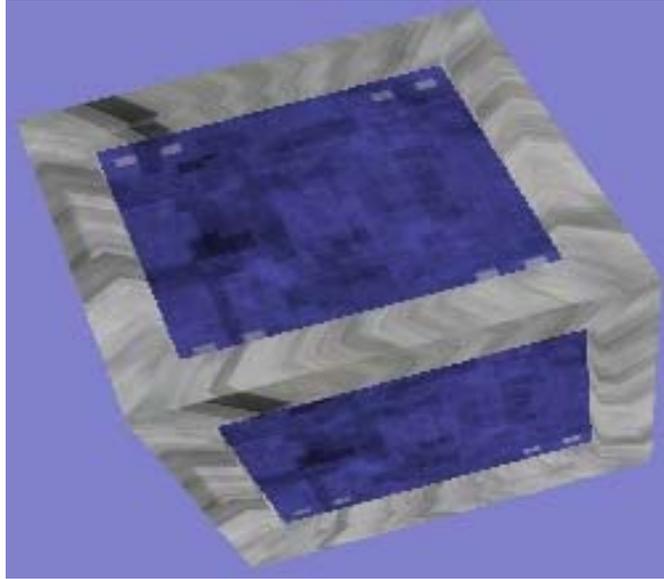
As we've seen so far, you have to manually (or algorithmically) create all the geometry you use. This is absolutely fine for things like cubes, spheres, triangles etc... but how about trying to create a model of a person – in particular an animated character? I know you people aren't stupid – and you wouldn't try to manually type in all the vertices... ☺ The other important factor is the data-driven architecture – A very powerful and popular game programming system. If all your geometry is stored in files, then you need only alter those files for it to be globally changed across the whole game – rather than sifting through all your code, making the changes then recompiling...

So what exactly makes up a model? In general it's just a collection of vertices, indices, textures and materials that when rendered appear as a complete model. They are often referred to as objects or meshes. In general, it's a self contained instance that will, to a certain degree, manage itself.

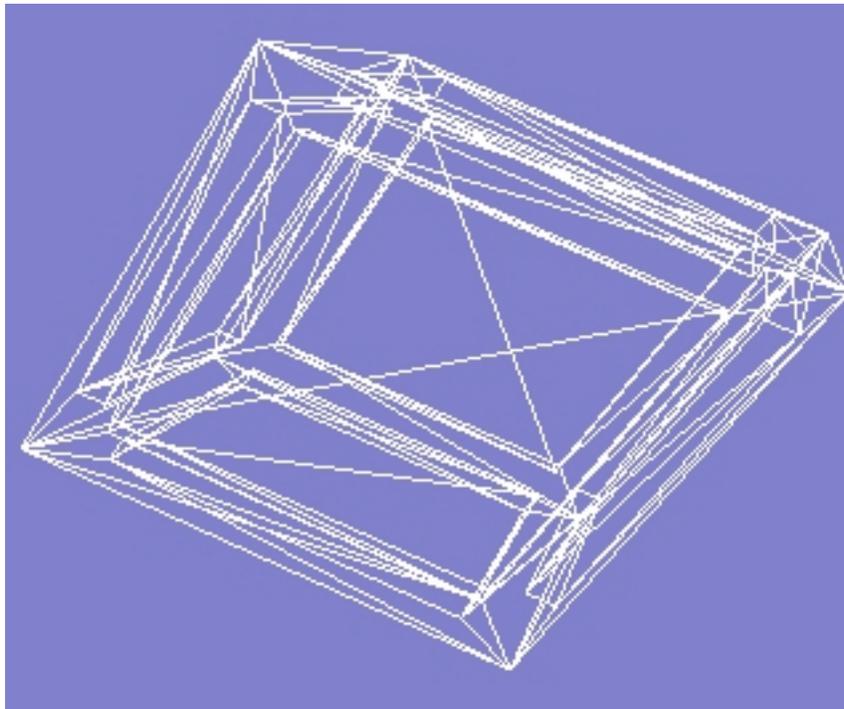
This is where 3D modelling packages such as maya, milkshape 3d, 3DS Max, Truespace, lightwave etc... come into play. One of more of these programs will become your best friend when it comes to creating 3D geometry, purely because they're designed to do it, and are exceptionally good at doing it. Learning a 3D modelling package properly can be as hard, or harder than the actual programming – I strongly recommend getting and reading a good book on your 3D modeller (unless you're already competent of course).

Our task for this section is to take a model created in one of these programs and load it into our Direct3D program for real-time viewing. This is actually a surprisingly easy task. I happen to use 3D Studio Max to do my modelling, and it's native format is the .3DS format – which is convenient as Microsoft built in a converter for the .3DS format to .X format. The .X format is Microsoft's native DirectX file format – and is therefore built into the D3DX API quite nicely. This is all great IF you're using the *correct* tools, if you aren't then you'll need to write your own data parsing function to convert the data in the file into D3D acceptable triangle/vertex data.

To demonstrate using models in direct3d I made a more complicated version of the original cube model. Using 3D Studio Max this was a fairly trivial process of extruding backwards certain parts of each face – such that I was left with a raised border around each face. I then used Max's powerful texturing tools to make the raised borders use the grey part of the texture, and the inner panel to use the purple/blue part of the texture. The final results looked like this:



The 3D effect of the borders isn't too apparent from this static shot; but when you see it rotating in real-time you can notice them quite easily. This next shot shows the geometry in wireframe mode:



This final shot looks quite complicated, because you can see all 6 faces of the cube at once – and as each one has quite a bit of geometry it does look like a mess of lines! However, you can still clearly see that this version is considerably more complicated than the original hard-coded cube model.

Now, onto the code. Luckily for you, the code is really quite simple for both loading, and rendering of models. If you have to write your own loading function then it could

get a little more complicated – depending upon how you code it. We need 4 global declarations before we can get started loading models:

```
Dim nMaterials As Long
Dim MeshMaterials() As D3DMATERIAL8
Dim MeshTextures() As Direct3DTexture8
Dim CubeMesh As D3DXMesh
```

All models are sub-divided into sections – take a car model for example, you may have a section for 4 wheels, another for the windows, and another for the main body. These sections are usually separated by different materials or textures. This explains the first declaration, *nMaterials*, which keeps track of how many sections we have, and will also be used later to redefine the next two arrays of materials and textures. The final declaration, *CubeMesh*, is a class type built into the D3DX runtime library; it will handle the storage/rendering of the model geometry; in essence it just manages a couple of vertex buffers and index buffers. This next little bit of code will load a model from the hard drive / CD-ROM...

```
Set CubeMesh = D3DX.LoadMeshFromX(App.Path & "\cube_3d.x", D3DXMESH_MANAGED, _
    D3DDevice, Nothing, mtrlBuffer, nMaterials)

If CubeMesh Is Nothing Then GoTo BailOut: '//Dont continue if the above call did not work

ReDim MeshMaterials(nMaterials) As D3DMATERIAL8
ReDim MeshTextures(nMaterials) As Direct3DTexture8

For i = 0 To nMaterials - 1

    '//Get D3DX to copy the data that we loaded from the file into our structure
    D3DX.BufferGetMaterial mtrlBuffer, i, MeshMaterials(i)

    '//Fill in the missing gaps - the Ambient properties
    MeshMaterials(i).Ambient = MeshMaterials(i).diffuse

    '//get the name of the texture used for this part of the mesh
    TextureFile = D3DX.BufferGetTextureName(mtrlBuffer, i)

    '//Now create the texture
    If TextureFile <> "" Then 'Dont try to create a texture from an empty string
        Set MeshTextures(i) = D3DX.CreateTextureFromFileEx(D3DDevice, App.Path & "\" & TextureFile, _
            256, 256, D3DX_DEFAULT, 0, _
            D3DFMT_UNKNOWN, D3DPOOL_MANAGED, _
            D3DX_FILTER_LINEAR, D3DX_FILTER_LINEAR, _
            0, ByVal 0, ByVal 0)
    End If

Next I

Debug.Print "Number of Faces in mesh: " & CubeMesh.GetNumFaces
Debug.Print "Number of Vertices in mesh: " & CubeMesh.GetNumVertices
Debug.Print "Number of segments in mesh: " & nMaterials
```

Not hugely complicated really. The last bit isn't really necessary – it just provides some interesting statistics for you. As far as vertex and face count goes, it isn't wise to trust your 3D-renderer when it comes to vertex/face counts, whilst those programs are 100% correct for the geometry in their program, the various

converters, and this loading function sometimes messes things up and adds more vertices.

Also note, that the LoadMeshFromX( ) function gets extremely slow when dealing with medium-large geometry files. Because all the processing is done away from your application you cant easily output a status bar showing the progress of loading it. This is one reason why people often write their own object formats – Ones I have written in the past have loaded a 2000 vertex model in <250ms, whereas with D3DX it's taken 3 seconds or more... This is also partly due to the .X file format specification including lots and lots of other rubbish that you may not actually be interested in – frame hierarchies, animation information; in general it's a very flexible format, but you can get a considerably smaller file, and considerably faster loading times should you design a custom format that is specific to exactly what you want.

The final part for dealing with models is to render them. Luckily for us, this is also very, very simple. These following lines should be placed within a BeginScene()...EndScene() block:

```
For i = 0 To nMaterials - 1
    D3DDevice.SetTexture 0, MeshTextures(i)
    D3DDevice.SetMaterial MeshMaterials(i)
    CubeMesh.DrawSubset i
Next I
```

Basically, all we're doing is looping through all the sections with different materials, committing those materials and textures to the device, then rendering the relevant geometry.

Several of the more popular file formats – mdl, md2, md3, 3ds etc... are covered on [www.wotsit.org](http://www.wotsit.org) - a great site for all file specifications!

## Using Direct3D Lighting

Okay, onto our final section for this article. Lighting in general is a very important topic to understand, and unfortunately, it is quite complicated as well.

It is often a good idea to look at cinema for lighting – cinema has been around for about a century now, and has progressed into a fine art form, and one of the many things that makes or breaks a scene in a film is the lighting, yet the key aspect is that you don't necessarily notice it. Ambient lighting is a very subtle effect that will often set the atmosphere for a film – how many horror films have the scary scenes in broad daylight/with the lights on (well, I know there are a few!). Shadows and the type of lighting (strobe, direct, soft, bright, dark) are also huge factors. In my opinion, computer gaming is only just starting to catch up with true artistic lighting, In the last year or so many of the level-architecture articles on websites have specifically brought lighting up as a major topic – whereas before it was just "put the light where it looks best"... Games such as Max-Payne are the first to put the best lighting algorithms (ray tracing/radiosity) to great use, and I expect many future games to follow this pattern.

Don't get your hopes up straight away though – the Direct3D lighting engine, whilst complicated, is still very, very simple. The first point to notice is that it won't generate shadows, secondly, it doesn't handle reflection or refraction, thirdly, it's only an approximation – accurate only at each vertex. The more complicated solutions require the use of light maps, and other pre-calculated methods (which are too complicated to go into here). I read somewhere that many of the Max-Payne maps required several hours of pre-processing just for the lighting algorithm, so you can appreciate why it's not done in real-time ☺

For now, we'll be happy with the Direct3D lighting engine, once you have mastered this then you can begin to consider other models.

You have actually already seen the effects of the lighting engine in all 3 parts of this series. Whenever we specified a colour for a vertex, and got the gradient of colours across a triangle we were actually seeing D3D lighting at work. To save on processing times, Direct3D will linearly interpolate across a triangle the colours from its 3 vertices – it assumes that the light won't change considerably between them. This, to a certain degree, won't matter for small triangles, but for larger triangles this causes a big problem – if none of the vertices fall within the lights range then it won't be lit, even if a large area of the triangle is actually within the lights range.

In order to proceed with lighting you must use a little maths, the proof behind these equations isn't really too important, all you need to know is how to use the equations to get the results you want. As I've already stated, Direct3D performs its calculations on a per-vertex basis, thus we must include some extra information with every vertex – a normal vector. This vector indicates what direction the vertex is facing, which may seem a little strange – but it makes perfect sense really: A triangle facing away from the light should get no light, whereas a triangle facing the light directly should get lots of light, and how do we tell if the triangle is facing the light or not? Use the normal...

Typically the normal will represent the direction the triangle is facing, however, it doesn't have to! Whilst it often looks a little strange, you can do strange things to the normal and get some very odd effects – not very good for realistic scenes, but fine for more humorous scenes. You also have to take much more care over indices when using D3D lighting – if two (or more) triangles share the same vertex, what direction is it facing? One way of doing this is to generate a normal for each triangle, then average them out to give a final direction for the shared vertex. This method usually works a treat, but there are times when it generates results that look wrong for all the triangles concerned...

If we have a triangle defined by the three vertices v0,v1,v2 then the normal is going to be found using the following function:

```
Private Function GetNormal(v0 As D3DVECTOR, v1 As D3DVECTOR, v2 As D3DVECTOR) As D3DVECTOR
  '//0. Any Variables
  Dim v01 As D3DVECTOR, v02 As D3DVECTOR, vNorm As D3DVECTOR

  '//1. Get the vectors 0->1 and 0->2
  D3DXVec3Subtract v01, v1, v0
  D3DXVec3Subtract v02, v2, v0

  '//2. Get the cross product
  D3DXVec3Cross vNorm, v01, v02

  '//3. Normalize this vector
  D3DXVec3Normalize vNorm, vNorm

  '//4. Return the value:
  GetNormal = vNorm
End Function
```

That's fairly harmless really – the D3DX helper library handles all the complicated maths for us – the cross product, subtraction and normalizing. However, if you want to avoid using the D3DX functions then the function will look like this instead:

```

Private Function GetNormal2(v0 As D3DVECTOR, v1 As D3DVECTOR, v2 As D3DVECTOR) As D3DVECTOR
  '//0. Any Variables
  Dim L As Double
  Dim v01 As D3DVECTOR, v02 As D3DVECTOR, vNorm As D3DVECTOR

  '//1. Get the vectors 0->1 and 0->2
  v01.X = v1.X - v0.X
  v01.Y = v1.Y - v0.Y
  v01.Z = v1.Z - v0.Z

  v02.X = v2.X - v0.X
  v02.Y = v2.Y - v0.Y
  v02.Z = v2.Z - v0.Z

  '//2. Get the cross product
  vNorm.X = (v01.Y * v02.Z) - (v01.Z * v02.Y)
  vNorm.Y = (v01.Z * v02.X) - (v01.X * v02.Z)
  vNorm.Z = (v01.X * v02.Y) - (v01.Y * v02.X)

  '//3. Normalize this vector
  L = Sqr((vNorm.X * vNorm.X) + (vNorm.Y * vNorm.Y) + (vNorm.Z * vNorm.Z))
  vNorm.X = vNorm.X / L
  vNorm.Y = vNorm.Y / L
  vNorm.Z = vNorm.Z / L

  '//4. Return the value:
  GetNormal = vNorm
End Function

```

The above is for reference, should you write an editor that isn't linked to the DX8 runtime library, or should you want to try and optimise parts...

One important factor that I haven't mentioned yet, is that the vertices, v0,v1,v2 need to be in a clockwise order – you should be aware of this, due to the implications of culling by the D3D renderer, but for the maths above, if the triangle vertices were in an anti-clockwise order then the resulting normal would point in the opposite direction – which, in most cases would mean that your vertices don't get lit...

Now that I've covered generating normals, we need to know what to do with them. The following excerpt is the vertex FVF declaration, and the vertex type:

```

Const FVF_VERTEX = (D3DFVF_XYZ Or _
                    D3DFVF_NORMAL Or _
                    D3DFVF_TEX1)

Private Type VERTEX
  P As D3DVECTOR
  N As D3DVECTOR
  T As D3DVECTOR2
End Type

```

The 'P' member is the vertex's position, the 'N' member is the vertex normal and the 'T' member is the texture coordinate.

To demonstrate D3D lighting I'm going to use the two methods already demonstrated in this article – texturing and model loading... simply because it allows me to show you the effects easily.

There are 4 types of light provided for you by Direct3D – point, spot, direction and ambient lights. The first 3 require that you set up a special D3DLIGHT8 object that describes the light, the fourth requires that you set a render state. The following list covers the 4 types of light, in order of processing speed:

### **Ambient Lights**

Ambient lights have no source, no direction and no range – they affect every vertex rendered. The basic result is that no vertices are rendered darker than the currently specified ambient colour – setting it to a dark grey will result in everything being visible a small amount. We set the ambient light value using the following code:

```
D3DDevice.SetRenderState D3DRS_AMBIENT, D3DCOLOR_XRGB(100, 100, 100)
```

### **Directional Lights**

Directional lights are good for general shading of a scene, such that everything is evenly lit up, but you also get a dark side to every object. They can be used very effectively as a sun object.

Directional lights have direction and colour only, no range, no position and no attenuation (see Point lights for more details). A completed Directional light structure looks like this:

```
Dim lghtDirectional As D3DLIGHT8

lghtDirectional.Type = D3DLIGHT_DIRECTIONAL
lghtDirectional.Direction = MakeVector(0, -1, 0)
lghtDirectional.position = MakeVector(1, 1, 1) 'shouldn't be left as 0
lghtDirectional.Range = 1 'shouldn't be left as 0
lghtDirectional.diffuse = CreateD3DCOLOR_XRGB(1, 0, 1) 'green light
```

### **Point Lights**

Point lights have a position and a range, but no direction – they emit light in all directions. A simple real world analogy would be a light-bulb. To set up a point-light you need to fill out a D3DLIGHT8 structure:

```
Dim lghtPoint As D3DLIGHT8

lghtPoint.Type = D3DLIGHT_POINT
lghtPoint.diffuse = CreateD3DCOLOR_XRGB(1, 1, 0)
lghtPoint.position = MakeVector(0, 0, -10)
lghtPoint.Range = 25#
lghtPoint.Attenuation0 = 0#
lghtPoint.Attenuation1 = 1#
lghtPoint.Attenuation2 = 0#
```

The attenuation values are quite important to understand. As we all know, light from a given source decreases the further from the light source that we are – the light attenuates. The three values that D3D allow us to use control how the light attenuates – constant, linear and quadratic (0 through 2 respectively). You should never let all three = 0 at the same time, otherwise you'll get internal divide-by-0 errors. Experiment with different values to see what happens, a value of 1 in Attenuation0 will remove any attenuation, whilst negative values in the other two will cause the light to get brighter the further away from the light source it gets ☺

For those mathematicians amongst us, the general attenuation formula is:

$$A = 1 / (\text{Attenuation0} + D * \text{Attenuation1} + D^2 * \text{Attenuation2})$$

Where D is the distance from the light source to the current vertex. As you can see, the denominator is a standard quadratic equation in the form of  $aX^2 + bX + c$ .

The other point to note is that when specifying colours they are on a 0.0 to 1.0 scale, rather than the standard 0-255 scale. This is because you can specify negative values, and values >1, allowing extra bright lights, and "dark" lights that remove colour rather than add it.

### **Spot Lights**

Finally, we get onto spot lights. These are the slowest type of lighting available, but in some cases look by far the best (A tunnel with several spot lights shining down from the ceiling for example). Hopefully you can visualise in your head a spot-light, and how they interact with the world – a cone of light projected from one point in one direction, brightest in the middle, getting darker towards the edge... It is the fact that it is based on a cone that it requires more calculation time – we need to work out IF it's in the cone, and how close to the "centre" of the cone (for brightness). A completed D3DLIGHT8 object for a spot-light will look like this:

```
Dim lghtSpot As D3DLIGHT8
lghtSpot.Type = D3DLIGHT_SPOT
lghtSpot.Range = 50#
lghtSpot.diffuse = CreateD3DColorVal(1, 0, 0, 1)
lghtSpot.Direction = MakeVector(0, 0, 1)
lghtSpot.position = MakeVector(0, 0, -10)
lghtSpot.Theta = 0.25 * PI
lghtSpot.Phi = 0.5 * PI
lghtSpot.Attenuation0 = 0.1
lghtSpot.Attenuation1 = 1#
lghtSpot.Attenuation2 = 0#
```

As you can see, a spotlight has range, direction, position and colour. It also has two new values – phi and theta. These two values indicate the angle (in radians) of the spot-lights cone. Theta is the inner cone, Phi is the outer cone. Phi must be a positive value between 0 and  $\pi$  (180°) and Theta must also be a positive value between 0 and Phi. If you think about this, it makes perfect sense... An outer angle greater than 180° makes little sense really (it would start shining behind itself), and an inner angle greater than the outer angle doesn't make much sense either.

Remember that these values MUST be set in radians – if you use degrees, all sorts of funky things will start happening! If you really cant get your head around radians then you can multiply a value in degrees by  $(\pi/180)$  where  $\pi$  is the mathematical constant 3.14159...(can be calculated by typing "4\*atn(1)" in the immediate window in VB).

Now that we've learnt how to configure a D3DLIGHT8 object for all 3 main types of light, we're going to need to let the device know about them. You can register as many light objects as you want with D3D, BUT you can only have a certain number *enabled* (on) at any one time. You can detect how many lights may be turned on at any one time by using the *D3DCAPS8.MaxActiveLights* value. If you enable more lights than are supported the call tends to fail, or if it does succeed then it wont actually process the light when doing the calculations... This value tends to be 8 – 16 on the GeForce cards, for most older cards it'll return -1, which indicates that an unlimited number of lights can be "on" at any one time; However, the more lights you enable the slower your program will run.

```
D3DDevice.SetLight 0, lghtPoint
D3DDevice.SetLight 1, lghtDirectional
D3DDevice.SetLight 2, lghtSpot

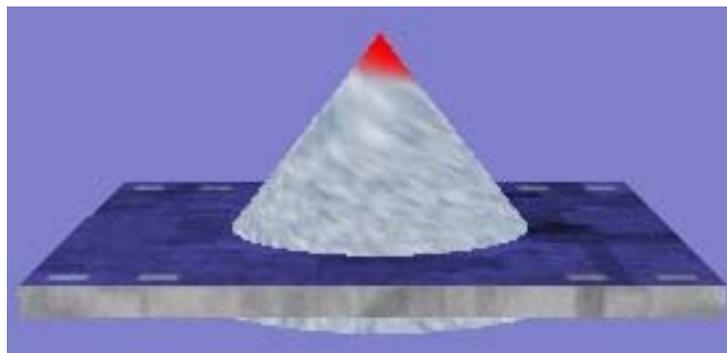
D3DDevice.LightEnable 0, 1
D3DDevice.LightEnable 1, 1
D3DDevice.LightEnable 2, 1

D3DDevice.SetRenderState D3DRS_LIGHTING, 1
```

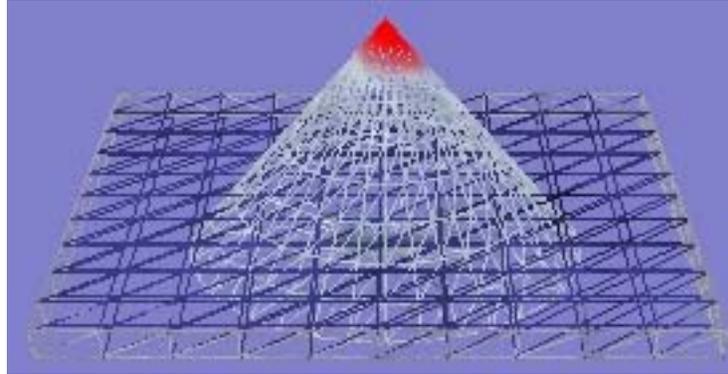
Not too complicated really, the last function, *LightEnable*, takes the index and a simple 1/0 value for on/off respectively. Any geometry processed after the "D3DDevice.LightEnable x, 1" line will be lit by that light (given that the geometry is within the influence of that light). It is perfectly acceptable to turn a light on for only one model – such that only it gets lit by that light.

To show off the lighting we'll need a new model to play with. The cube mesh I made/showed earlier isn't complex enough to show off the new lighting code; instead I'm going to use a much higher vertex-density mesh. This will mean that it runs considerably slower on most machines, but this is only a demo...

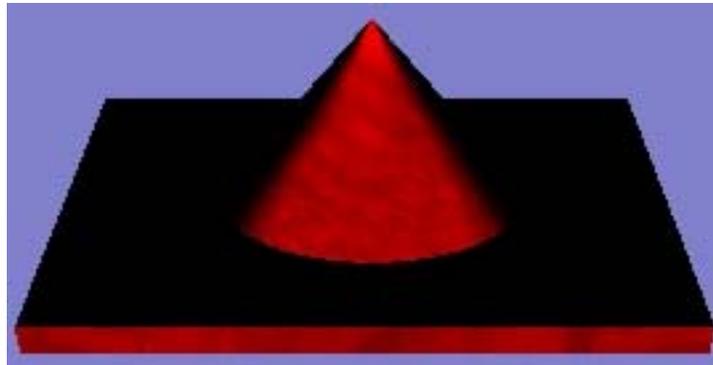
First off: A solid, unlit version of the geometry



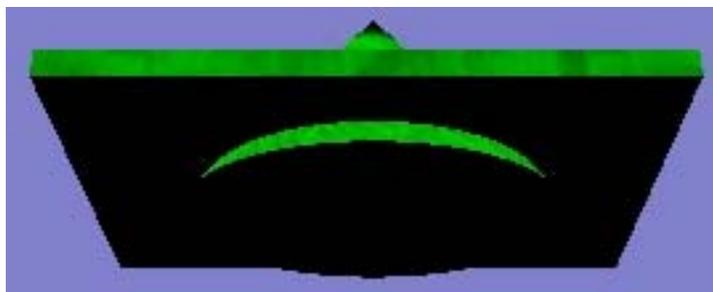
Second: A wireframe version, to show you the complexity of the geometry (3384 vertices)



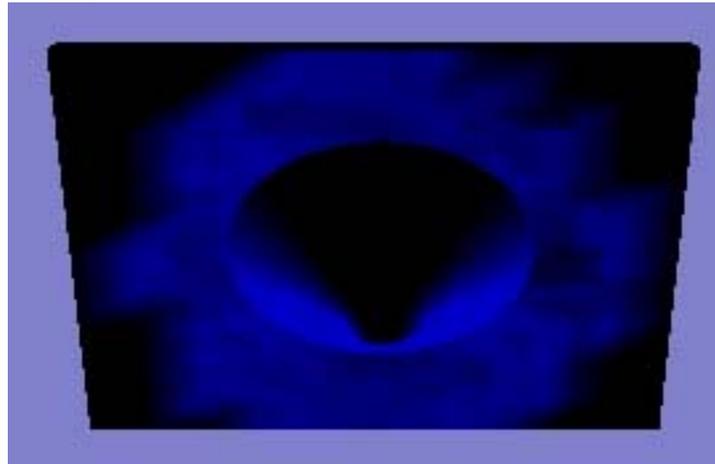
Third: The (red) Point light which is located directly below the camera in this shot, notice the distribution of the lighting.



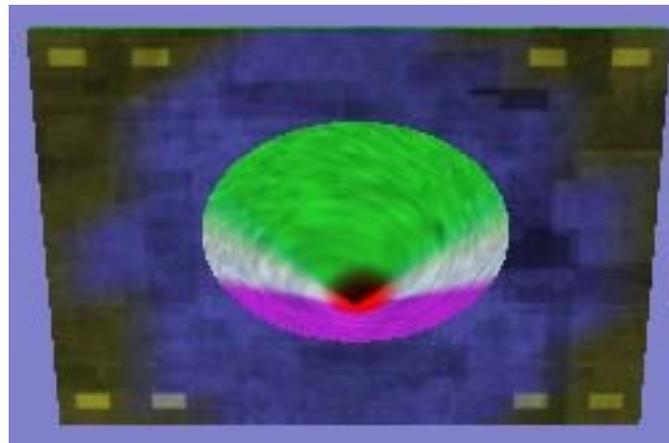
Fourth: The (Green) Directional Light. Notice that the lighting is evenly distributed, and that the entire "bottom" is unlit. Also notice, that if shadows were cast, the bottom of the cone would not be highlighted in green.



Fifth: The (Blue) Spotlight, notice a very distinct spot – indicating the presence of a cone.



Lastly: A nasty mess of all the colours – where the red and blue lights colour the same section we get magenta, in other parts we have a yellow colour.



One thing that is quite clearly visible on the last two is that the lighting on the very tip of the cone isn't lit in the same way that the rest of the cone/model is. This is done deliberately here to show off how textures can affect the final colour. Take the colour red (as on the nose), it can be represented as  $RGB(255,0,0)$ . If we then use a lighting colour of  $RGB(0,1,0)$  we'll get various shades of green, and only green, interpolated across the triangle. This is important – if there was no texture applied, there would ONLY be green pixels. To get the final pixel colour we MULTIPLY the interpolated lighting colour with the texel colour:  $RGB(R_t R_l, G_t G_l, B_t B_l)$  where  $X_t$  = texture and  $X_l$  = lighting. If we go back to the original example of a red texel colour,  $RGB(255,0,0)$ , and a green light,  $RGB(0,1,0)$  that multiplied colour works out as  $RGB(255*0, 0*0, 0*1) = RGB(0,0,0) = \text{Black!}$  Which is what you can see in the above screenshots. The bottom line is this: If the texture doesn't contain any (or very little) of the channel that the light uses, then the resulting pixel will be black. This is easily solved by using ambient lighting; but can also be a useful tool for lighting effects.

## **CONCLUSION**

Okay, so this 3 part series is now complete. I really, really hope that you liked it – either way, drop me an email at [Jack.Hoxley@DirectX4VB.com](mailto:Jack.Hoxley@DirectX4VB.com) , constructive comments are always welcome. From the emails I've received recently this series has been very successful... ☺

As for DirectX programming – you should now have enough knowledge to write a very simple game / engine. Don't be a fool and try a "simple Quake clone", it's not going to happen. However, a nice 3D pong/breakout clone, or a simple maze/puzzle game would make for a good learning project. There is absolutely tonnes and tonnes of stuff left to learn! I have been working with DirectX for 2-3 years now (version 5,6,7,8) and I don't think I've ever learnt *everything* in every release of the API – close, but not quite!

As a final note, this may be the last in this series, but I do have a website that will continue to be updated – where you can find more in depth tutorials on the content covered in this series, more advanced tutorials, and generally newer content. Check it out here: <http://www.DirectX4VB.com> ...

Many thanks for getting this far with my series.  
Jack Hoxley

## **Web Links**

<http://www.microsoft.com/DirectX/>

- the main end-user site

<http://msdn.microsoft.com/directx/>

- the developer's site, where you can download the full DirectX SDK's. This is going to be of most use to you.

<http://developer.nvidia.com/>

- nVidia (3D card giant) has their own developer website here. Definitely worth a look around, even if you don't have an nVidia based card.

<http://www.ati.com/developer/>

- ATI (another 3D card giant) also have a developer-related section to their site. Another good place to look.

<http://www.mwgames.com/voodoovb/>

- One of the best VB gaming websites around, particularly good for it's active forums (you'll often find me hanging around here)

<http://www.rookscape.com/vbgaming/>

- Lucky's VB Gaming Site, another one of the huge VB gaming websites, also with a very active community forum

<http://www.DirectX4VB.com>

- My site!! How could I leave this one out of the list??

## **Recommended Reading**

This following list is just for your reference if you're interested in buying some "real" books on the subject. They all come from my book reviews section – [www.DirectX4VB.com/reviews.asp](http://www.DirectX4VB.com/reviews.asp) ; I strongly suggest you have a look at this page before buying the book...

### **Game Architecture And Design – Rollings And Morris**

- Coriolis Technology Press, 1-57610-425-7

- A great book on the theory behind game design and development

### **Microsoft Visual Basic Game Programming With DirectX – Harbour**

- Premier Press Inc, 1-931841-25-X

- One of the only good books on DirectX in the VB language (most are in C++)

### **Special Effects Game Programming With DirectX – McCuskey**

- Premier Press Inc, 1-931841-06-3

- A more advanced text that you could use to go the next-step from this article.

## **DISCLAIMER AND COPYRIGHT**

The tutorial text and source code are provided as-is, I have done everything reasonable to ensure that details are accurate and the source code works as stated, but given the nature of the topic I cannot guarantee it will work on every computer. I am often happy to help solve any problems you may have (email me!), but bare in mind that I do live a busy and hectic life and may not have a huge amount of spare time.

**©2002 Jack Hoxley – All Rights Reserved**

I am happy for you to redistribute this *pdf* document on any website, ftp location, CD as long as it remains unmodified. The same holds for the source code included in the archive. I would also appreciate it if you let me know when/where you redistribute this file.