Blitz3D

Unofficial Programming Guide

by

Paul Gerfen

Contents of this document written by:

Paul Gerfen

Visit the author's website for updated content, source code, and media:

http://www.gamecoding.co.uk

Table of Contents

Welcome to Blitz3D

"The Ultimate 3D Creation Language"

Hi Guys, Paul here from Gamecoding. Recently I was asked to come up with a selection of tutorials for complete beginners. Seeing as I'm mostly a 2D games programmer - I too could be considered as a total beginner.

B3D is an incredibly powerful language, but believe me - by the time you have worked your way through these mini tutorials, you will have the basic 3D knowledge to realize your dreams and start to create your own projects.

Updates to these tutorials as well as extra notes, can be found on my website www.gamecoding.co.uk or at the official Blitz Basic 3D website, www.blitzbasic.com

Background

Let's begin by running through what we have and more importantly what we shall need:

No doubt you've already had a run through the demos that accompany Blitz3D, and are very keen to get stuck in and produce the next big "Quakey-Game". Unfortunately and this stage all we are going to do is start with the very basic's and work our way through all the main functions of B3D before we even think of moving on to advanced topics.

Hopefully you have a little experience of Blitz Basic programming (or for that matter any Basic language would help), Let's take a look at 3D before we delve in…

As your probably use to with other basic languages you have a set of co-ordinates, X (Left & Right) and Y (Up and Down). With 3D we have an extra coordinate to consider "Z", This gives us depth into the screen. For example if we were standing at a river throwing a rock as far as we can into it, what would be happening is:

As the stone leaves our hand it firstly moves away from us, and then gains height. What is happening here is that the Z coordinate is increasing as the stone moves away from us, while the Y coordinate gains height. Of course the stone will eventually fall to the ground (decrease in Y).

It’s important to try and visualize these 3 axes, as we shall see a 3d world is based entirely on them.

Let's start by running through some of the major parts of Blitz3D before we start to get our hands dirty on some actual code.

Entities
The basic building blocks of all Blitz3D programs.

Cameras
Well, we all need to see what's going on !

Planes
Nope, not the flying kind with wings and engines...

Meshes
Perhaps you'd like to make your own objects from scratch.

Terrains
Rolling Hills and Landscapes, in less time it takes to put the kettle on.

MultiTexturing
Why have a plain object when you can give it a coat of many colours.

Read through some of the info? I hope so... Let’s dig deep and start to produce our first B3D program.

My aim is to take you step by step through the very basic's. You can run the source code that goes with each chapter by either CUT & PASTING from the tutorials themselves, or by loading the accompanying files from the B3D CD. (Recommended!)

Introduction to Entities

Everything going on in a Blitz3D program takes place within a 3D world, which is populated by 'entities'.

Entities come in many flavors, and each kind of entity performs a certain, specialized task, but all entities have a few things in common, the most important of which are:

> All entities have a position, a rotation and a size, and can be moved, rotated and scaled at will.
>
> All entities have an optional 'parent' entity. If an entity has a parent, then it becomes 'attached' to the parent and will move, rotate and scale whenever the parent does.

Now, lets have a quick look at the kind of entities that can be used in the 3D world:

Cameras

Camera entities allows you to 'see' what's going on in the world. If you don't create at least one camera, you wont be able to see anything! Cameras also control where on the screen 3D rendering occurs, and provide atmospheric fog effects.

Lights

Light entities provide illumination for the 3D world. Without any lights, you will still see what's happening in the world thanks to 'ambient light'. Ambient light is light that is 'everywhere at once', and illuminates everything equally. However, since ambient light has no position or direction, you wont get nice shading effects, and everything will look very flat.

Meshes

Mesh entities provide a way to add actual, physical items to the world. Meshes are made up of triangles, and each triangle can be colored or textured in a variety of ways. The most common way of adding a mesh to the world is to load it from a file produced by a 3D modeling program, but you can also create your own meshes.

Pivots

Pivot entities don't actually do a lot! Their main purpose in life is to provide a parent for other entities. This can be useful for many reasons. For example, by attaching a bunch of meshes to a pivot, you can move all the meshes at once simply by moving the pivot!

Sprites

Sprite entities are flat, 2D entities most commonly used for particle effects, like explosions, smoke and fire. Sprites automatically face the camera, regardless of the angle you view them from, so are often textured with a 'spherical' style texture.

Planes

Plane entities are infinitely large, flat surfaces that can be used for a variety of effects such as ground and water. Since planes are entities, they can also be moved and rotated, which means you can also use them for sky and cloud effects.

Terrains

Terrain entities are used to create very large landscapes. They are constructed of a grid of triangles, and are rendered using a technique that draws an approximation of the terrain using a limited number of triangles. This is necessary for very large terrains that may contains millions of triangles - which would not only be far too slow for real-time rendering, but would also take up way too much memory!

These entities form the backbone of any B3D program, it really is worth spending the time getting to know the various instructions associated with them. As we shall see - by using different parameters and the special additional commands for controlling or altering their behavior, we can adapt an entity to our liking. Even if it means defining a special FX that Blitz, with its present command set cannot do !.

Introduction to Camera Entities

Cameras allow you to see what's going on in the 3D world. Without at least one camera, nothing that may be happening in the world will be rendered to the screen.

Cameras also control a number of other effects, including rendering range, zoom and fog.

Creating Cameras

To create a camera, simply use the **CreateCamera** command:

```
;create a new camera!
camera=CreateCamera()
```

Camera Viewport

A camera needs to know where on screen its view of the world is to be rendered. This area of the screen is known as the camera viewport. By default, a new camera's viewport is set to the entire screen.

To change the camera viewport use the **CameraViewport** command:

```
;Draw to top left corner of the screen
CameraViewport
camera,0,0,GraphicsWidth()/2,GraphicsHeight()/2
```

Camera Range

Camera range determines how far into the distance a camera can 'see'. You specify 2 values for camera range - a near value and a far value. All rendering is clipped to this range.

Its a good idea to try and keep the near value as high as possible, while keeping the far value as low as possible. This helps to minimize z-buffer problems like 'jaggies'. By default, the near value for a new camera is 1, and the far value is 1000.

To change the camera range, use the **CameraRange** command:

```
;Sets the near range to 5 and the far range to 5000
```

```
CameraRange camera,5,5000
```

Camera Zoom

Changing a cameras zoom allows you to 'zoom-in' or 'zoom-out' of the action in the world. The default zoom value for a new camera is 1. Zoom values greater than 1 will cause a 'zoom-in' effect, while zoom values less than 1 will cause a 'zoom-out'. Zoom values should never be set at 0 or less!

The **CameraZoom** command controls camera zoom:

```
;Zoom-in a bit!
CameraZoom camera,1.5
```

Fog Effects

Blitz3D allows you to setup fog effects independently for each camera. This can be useful for limiting what a certain camera can see. For example, a racing game might use a camera for its rearview mirror. However, you might want to limit what is visible in the rearview mirror in order to keep the game running fast.

To setup a simple fog effect for a camera, you might use something like this:

```
;Sets the color of fog for this camera
CameraFogColor camera,0,128,255

;Sets the near/far range of fog for this camera
CameraFogRange camera,1,1000

;Sets the fog mode for this camera.
CameraFogMode camera,1
```

Often, the fog range and camera range will be the same. However, by increasing the near value of fog, you can control how 'thick' fog appears to be. Smaller fog near values will result in a thicker fog effect, since the fog will start nearer to the camera.

Unfortunately, fog is not well supported on all graphics cards. While all cards can perform some kind of fog effect, the quality varies a lot. For this reason, its a good idea to make fog optional in your Blitz3D games.

Introduction to Planes

Planes are a simple way to add large, flat surfaces to the world. Planes are useful for such things as ground, sea, sky, clouds and so on. Just about every 3d game will use planes in some way or another.

Creating Planes

How do you create a plane ? - Easy, just use the **CreatePlane** instruction:

```
;create a simple plane.
plane=CreatePlane()
```

Manipulating planes

By default, a new Plane is positioned at 0,0,0 and faces upwards. However, you may want the plane to face a different direction, or to be moved up or down. To do this, you just need to use the standard entity commands.

For example, you might want to use a plane for a layer-of-clouds type effect. In this case, just rotate and position the plane as necessary:

```
;create clouds plane.
plane=CreatePlane()

;rotate plane so it faces downwards.
RotateEntity clouds,0,0,180

;and position it up in the sky!
PositionEntity clouds,0,100,0
```

So now we have a plane that is going to represent the sky, how can we make it look realistic ?

The easy answer would be to Texture it !, as you shall see later on in these tutorials - Blitz3D has ALL the answers.

If you take a careful look at most of today's top titles you will notice that most of the sky scenes contain more than one layer. Usually this consists of 2 - sometimes even 3 planes. The very top plane contains static objects such as the Sun, while the plane in front has a cloud layer. This can

be achieved by using the BRUSHFX instruction (to make the texture slightly see-through) -in conjunction with an animated texture, that is slowly wrap-scrolling. (And YES, B3D can do this effect with ease!)

Introduction to Meshes

Meshes are the entities you will most frequently be working with - unless you're writing something weird (please do...)!

Meshes are made up of vertices and triangles, and can be either loaded from files (perhaps the result of a 3d modeler program), or built 'by hand'. (assembled from within a B3d program in realtime)

Loading Meshes

There are 2 commands provided for loading meshes from files - 'LoadMesh' and 'LoadAnimMesh', both of which load a '.X' or '.3DS' file and return an entity. So what's the difference ?

Models stored in files are usually made up of several 'parts'. In the case of a character model, these parts may represent arms, legs etc.

In addition, files also contain animation information with the 'LoadAnimMesh' command. The entity returned will actually be the parent entity of the whole bunch of child 'parts'.

Therefore, 'LoadAnimMesh' really loads in several meshes !

If you don't need to animate a model, you can use 'LoadMesh' instead. This will 'collapse' all the parts in a mesh and return a single, combined mesh. The collapsed mesh will look the same as a mesh loaded with LoadAnimMesh, only you wont be able to animate it, and it wont have any child meshes.

Why bother with LoadMesh at all ? SPEED! - It's faster for B3D to deal with a single mesh than with multiple meshes, so if you're not planning to animate or doing anything tricky with a mesh, use LoadMesh.

Creating Meshes

Before looking closely at creating meshes, you'll need to know about using 'Brushes' in Blitz3D. A brush is a collection of properties used when rendering triangles. These properties are:

- **Colour** - The colour a triangle is rendered in. (see BrushColor)
- **Texture** - From 0 to 8 texture maps that are used to render a triangle. (see BrushTexture)
- **Alpha** - How transparent a triangle is. Alpha can range from 0 (completely transparent) to 1 (completely opaque). (see BrushAlpha)

- **Shininess** - How 'shiny' a triangle is. This value can range from 0 (not shiny) to 1 (really shiny!). (see BrushShininess)

- **Blend** - The blend mode used to render a triangle. Blend mode describes how a triangle is combined with what's already on the screen. (see BrushBlend)

- **FX** - Optional special effects for rendering. (see BrushFX)

To create a brush, you use the CreateBrush command:

```
;create a brush.
brush=CreateBrush()
```

Once we have a brush, we can set its properties:

```
;a red brush.
BrushColor brush,255,255,0
a shiny red brush.
BrushShininess brush,1
```

So what has all this got to do with meshes ?

Well, when I said meshes are made up of vertices and triangles, I sort of lied !. Meshes are actually made of 'surfaces', and surfaces are made up of vertices and triangles !.

When a surface is created, you provide it with a brush that controls how all the triangles in the surface are rendered.

So, a quick overview:

- A mesh is made up of any number of surfaces.
- Each surface in a mesh contains any number of vertices and triangles.
- All triangles in a surface are rendered using a common brush.

Lets dive on in and create a simple mesh:

```
brush=CreateBrush() ; create a brush
BrushColor brush,255,0,0 ; a red brush
mesh=CreateMesh() ; create a mesh
surf=CreateSurface( mesh,brush) ; create a (red) surface
```

```
AddVertex surf,-1,1,0 ; Now, we add 4 vertices...
AddVertex surf,1,1,0
AddVertex surf,1-,1,0
AddVertex surf,-1,-1,0
AddTriangle surf,0,1,2 ; and 2 triangles...
AddTriangle surf,0,2,3
UpdateNormals mesh
```

This code will create a simple red square mesh entity.

So what's with the weird 'UpdateNormals' command at the end ?. Well, in order for a mesh to be correctly lit, its 'vertex normals' must be calculated. Without going into the gory details, the UpdateNormals command will do this for you. If you are aware of how normals work, you can actually set your own vertex normals using the VertexNormal command. If not, just remember to stick an 'UpdateNormals' command at the end of any mesh modifications you do, or else your meshes will not be correctly lit.

Note that you can create any number of surfaces you want. So, the same mesh can contain many differently coloured/textured/whatever triangles !.

Why bother with surfaces at all ?, Why not just create a bunch of separate meshes, each with its own rendering properties and skip all this surface nonsense ?. Well, again it comes down to SPEED !. It's faster for Blitz3D to handle multiple surfaces - which can NOT be moved or rotated as entities can - than it would be to handle multiple meshes.

Modifying Meshes

Once you've created a mesh, there are various commands available for modifying the mesh in realtime. This can be used to create a range of impressive special effects such as waving flags, rippling water and so on.

These commands are:

- **VertexCoords** - Changes the coordinates of a vertex.
- **VertexColor** - Changes the colour of a vertex.
- **VertexTexCoords** - Changes the texture mapping coordinates of a vertex.
- **VertexNormal** - Changes the normal of a vertex.

For more info on how to use these commands, take a look at the Vertex tutorial further on down the line.

Introduction to Terrains

Terrain entities are used to draw very large landscapes made up of many, many triangles - even up to a million or more!

To actually draw this many triangles would be a very slow process, so terrains use a technique known as '**dynamic level of detail**' to speed things up.

This basically means that instead of drawing all of the triangles in a terrain, a smaller number of triangles is used to render an approximation of the terrain.

Terrains have a few limitations:

- The vertices of a terrain are laid out in a square grid pattern, with the same number of vertices along each edge of the grid.
- Only the height of a vertex may be modified.
- The size of a terrain must be a 'power of 2' value, eg: 2,4,8,16,32...

Creating Terrains

You can create terrains either by loading in a heightmap image, or by creating an 'empty' terrain and setting the height of each vertex yourself.

A heightmap image is simply a grayscale image, where black pixels represent 'low' vertices, and white pixels represent 'high' vertices. A heightmap image must be of a valid terrain size - ie: square, and a power of 2 wide/high.

When a terrain is created, its width and depth are the same as the size of the terrain, and its height is 1. For example, if you create a terrain using:

terrain=CreateTerrain(32)

The terrain will extend from 0 to 32 along the x and z axis', and from 0 to 1 along the y axis.

However, you are free to position and scale the terrain as you see fit using entity manipulation commands such as ScaleEntity, PositionEntity. For example:

```
;terrain is 32 x 1 x 32
terrain=CreateTerrain( 32 )

;terrain is now 320 x 100 x 320
```

```
ScaleEntity terrain,10,100,10

;and centred (on x/z axis') at 0,0,0
PositionEntity terrain,-160,0,-160
```

Terrain Detail Level

You can directly control how many triangles are used to approximate a terrain using the **TerrainDetail** command.

Using less triangles will naturally result in better speed, but beware that below a certain threshold your terrain will start to behave very strangely!

For example, if you ask Blitz3D to render a detailed 1,000,000 triangle terrain using only 100 triangles, chances are it just wont be able to do a decent job.

This usually leads to a phenomenon known as 'pop-in', where Blitz3D just isn't able to make its mind up about how to draw the terrain, and you end up with vertices abruptly changing position, or 'popping', when the camera view changes.

So what number of triangles should be used? Well, this is definitely a 'trial and error' kind of thing and depends on a number of factors:

- The variation in the terrain - ie: how 'bumpy' it is.
- The scale of the terrain - ie: how much of it is visible to the camera.
- The range of the camera.

You can reduce the number of triangles needed to represent a terrain using the following tricks:

- 'Blur' or smooth out the terrain. For example, if you're using a heightmap image, just apply a 'blur' filter or equivalent to the heightmap.
- Increase the scale of the terrain so less of it is visible to the camera.
- Decrease the camera range, again meaning less of the terrain is visible to the camera.

If all of this sounds a bit wishy-washy, well, it is!

Generating nice looking terrain with minimum pop-in is as much an artform as it is a science.

Blitz3D also provides a technique called '**vertex morphing**' to help reduce pop-in. Vertex morphing involves Blitz3D automatically smoothing out the terrain a little to reduce pop-in. The **TerrainDetail** command is also used to enable vertex morphing.

Modifying Terrain

The **ModifyTerrain** command is used to change the height of a terrain vertex, for example:

```
;create a terrain of size 32.
terrain=CreateTerrain( 32 )
;push the centre vertex up half-way.
ModifyTerrain terrain,16,16,.5
```

The parameters used with **ModifyTerrain** must be given in 'terrain coordinates'. This means that the x and z values should always be from 0 to the size of the terrain, and the height value should always be from 0 to 1. In other words, any scale, position or rotation applied to the terrain will be ignored.

Texturing Terrains

By default the terrain you create will be just a plain colour, but by using the **EntityTexture** command it is possible to apply a texture brush to the whole landscape. Unfortunately at this time, it is not possible to texture different parts of the terrain with different textures. But fear not !, remember a brush can have 8 multitextures applied to it. By carefully building up these textures - it is possible to have different texturemaps applied onto the same landscape.

Introduction to Texturing

Textures are probably the coolest thing to happen to 3D graphics since polygons!

Textures are special images which are drawn on top of polygons in order to give them added detail.

Textures can be either created 'from scratch' or loaded from an image file.

Just like ordinary Blitz2D images, textures are made up of pixels. These pixels can be accessed using x,y coordinates just like images - only, in a slightly different way.

Instead of x,y values ranging from 0 up to the size of the texture, x,y values for textures range from 0 to 1. This means the texture pixel at location .5,.5 is always in the center of the texture. This is possibly a little confusing at first, but it actually turns out to be very convenient as you don't have to keep track of how big your textures are!

Texture Flags

When a texture is created, a 'texture flags' value is provided to indicate what type of texture you're after.

Here are the legal texture flags values:

1:	Texture has color information.
2:	Texture has alpha information.
4:	Texture is a masked texture. This means that black pixels are transparent.
8:	Texture is mipmapped.
16:	Texture is horizontally clamped.
32:	Texture is vertically clamped.
64:	Texture is a spherical environment map.

Texture flags can be added together to combine the effect of separate flags. For example, a texture flags value of '3' indicates you want a texture with both color PLUS alpha information.

If neither the color, alpha or mask flags are used, then the texture is created in a special format that allows you to draw to it using standard Blitz2D or even Blitz3D commands! A texture created this way will have color information, but may or may not have alpha information (this depends on what graphics card you're using!), so its really safest to assume it has no alpha information.

If the masked flag is used, the color and alpha flags are ignored.

The clamped flags allow you to control whether the texture 'wraps' or not. By default, textures will 'wrap' or 'repeat' if drawn on large triangles. However, you can prevent this by using the clamped flags.

Finally, the spherical environment map flag provides a dead easy way to do cool 'reflective' type effects. Try it!

Creating Textures

You can either create a texture and draw to it by hand, or load a texture from an image file.

Loading a texture from a file is easy:

```
texture=LoadTexture( "mytexture.jpg",3 )
```

Note the texture flags value of '3'. This indicates we want the texture to contain both color and alpha information.

However, the JPG file format does not support alpha information! In this case, Blitz3D will create its own alpha values based on the color values in the texture.

However, both the PNG and TGA file formats do support alpha information, and Blitz3D will use this information if present.

To create a texture by hand, use the **CreateTexture** command.

```
;create a 256x256 texture
texture=CreateTexture( 256,256,0 )
;retrieve its width
width=TextureWidth( texture )
;retrieve its height
height=TextureHeight( texture )
```

Why are we retrieving the texture size when we've already told Blitz3D how big we want the texture?

Well, this is because all 3D graphics cards have limits on the size of texture they can handle. Typically, texture width and height should be a power of 2 value (eg: 1,2,4,8,16,32,64,128,256,512,1024...) and in some cases - 3DFX cards being the main culprit here - must be no larger than 256 by 256.

There are also rumours out there that some cards can only handle square textures - although here at Blitz Research we are yet to run into any such beast!

A texture size where width and height are both power of 2 values and <= 256 is pretty much guaranteed to be supported by your card - but it never hurts to check!

In the event that you specify a texture size not supported by your graphics card, Blitz3D will pick the nearest useful size instead.

Also note that we are using a texture flags value of 0. This is because we probably want to draw something to the texture, and to be able to use graphics commands on a texture you should not specify the color, alpha or masked texture flags.

OK, now we've created our texture we should probably draw something on it:

```
;set drawing buffer to the texture
SetBuffer TextureBuffer( texture )
;set cls color to red
ClsColor 255,0,0
;clear the texture to red
Cls
;set drawing color to blue
Color 0,0,255
;draw a blue oval.
Oval 0,0,width,height
;back to the back buffer...
SetBuffer BackBuffer()
```

Finally, once you've finished with a texture, use **FreeTexture** to release it.

Animating Textures

ScaleTexture, **RotateTexture** and **PositionTexture** can all be used to animate textures. For example, to cause a texture to scroll, you might use something like...

```
texture_x#=texture_x#+.1 ;add .1 to texture x position.
PositionTexture texture,texture_x#,0 ;position the texture.
```

...in your main loop. You can dynamically scale and rotate textures in a similar way.

Multitexturing

Blitz3D allows you to draw up to 8 textures to a single triangle, a technique known as '**multitexturing**'.

A common use for multitexturing is '**light mapping**', a technique where a texture containing lighting information is combined with a texture containing color information to provide lighting effects.

When you use the **EntityTexture** or **BrushTexture** commands, an optional 'index' parameter allows you to control which of the 8 textures you are setting.

By default, multiple textures are combined using multiplication which achieves a lighting type effect. However, the **TextureBlend** command allows you to perform other operations such as alpha-blend and add on multiple textures.

The Tutorials

You will need the source code files and object files to run these snippets. These can be found on the Blitz3D CD (or in the zip accompanying the Blitz Trial), I would advise opening up the source code projects and switching between Blitz and this tutorial. (As this will hopefully explain the code and various instructions in more detail)

Setting Up
Everything you will need to know about writing a B3D program.

Movement & Rotation
Time to display a shape on the screen and start to give it some movement.

Camera Movement
There's more to B3D than just a static camera.

Object Animation
Time to make our object come alive.

Texturing
Let's make the object more lifelike, by giving it a coat of paint.

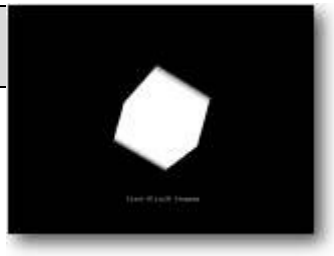
Lighting
B3D supports more than one type of lighting, here we take a brief look at the 3 main types available to us.

Collision Detection
So what does happen when two worlds collide ?

Vertexes
What happens when you smash your object up into separate pieces, and then tear each part up ?

Setting Up

Aim: To setup a simple B3D program
Skill level: Beginner
Files Needed: settingup.bb



Blitz3D sits on top of the regular Blitz, instead of learning a totally new set of commands we can use the same programming techniques as we would with a any Blitz 2D program. (which is very handy if you have used Blitz before)

Let's take a look at our first program, before we go any further.

```
Graphics3D 800,600

SetBuffer BackBuffer()

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

cube=CreateCube()
PositionEntity cube,0,0,5

While Not KeyHit(1)

TurnEntity cube,.1,.2,.3

UpdateWorld
RenderWorld

Text 320,500,"First Blitz3D Program"

Flip

Wend
End
```

Run the program above if you haven't already, hopefully you'll be confronted with a spinning 3D Cube and a short message. Not bad for only 15 lines of code !!!

So what's happening here ? - lets go through it…

Graphics3D 800,600

Perhaps the most important line of the whole program, this instruction is responsible for initializing the 3d graphics card. As you've probably guessed I'm setting up the screen with the res of 800x600. If you have any problems running the programs then change the values of this command. (as not all graphics cards will be able to use the same res)

SetBuffer BackBuffer()

This command is the same as the original Blitz buffer instruction as we use in all our Blitz programs.

camera=CreateCamera()
CameraViewport camera,0,0,800,600

These two camera instructions firstly create a standard camera, then setup the variables to respond to the graphics mode we are in.

Why do we need a camera ?, well the standard camera is what we can see. We could have multiple cameras setup in our program and switch between them. For example we could have one setup to follow a character in our program and one that remains static pointing at something. (for instance as a close circuit TV camera)

light=CreateLight()

We shed some light on our little scene by defining a standard light, by default B3D will already give us a light (so there's no real need for this instruction) - its just good practise to set things up properly.

cube=CreateCube()
PositionEntity cube,0,0,5

To make our little cube, instead of using a 3d modeller program - I've cheated and used the B3D autocreate command to give us a cube.

NOTE: Blitz3D contains a whole host of built in ready-to-use shapes such as Cube, Sphere, Cylinder and Cone. (See doc's for details)

```
PositionEntity cube,0,0,5
```

Now we have cube we need to position it, by using the POSITIONENTITY command. As you can see we positioned the CUBE, but we could also use the command to move anything - including the camera. The coordinates are in the format of X,Y and Z. Remember using negative coordinates will move the shape in the opposite direction, for example: PositionEntity cube,3,-2,7 is really moving the cube 3 units to the right, 2 units up and 7 units into the screen. (away from us)

I expect you will use this command mostly in the beginning sections of your programs to move everything in position.

```
While Not KeyHit(1)
```

The start of our main programming loop, It's good manners to use a loop like this that will quit out when the ESC key is pressed. How many times have you run a program only to find that the only way of quitting out is to restart the computer ?

```
TurnEntity cube,0.1,0.2,0.3
```

There's going to be at least 2 commands that your program will rely heavily on, and this is one of them. This little beauty of an instruction turns the cube by the amount giving in the X,Y and Z amounts. In this case X=.1 of a unit, Y=.2 and Z=.3 - as before you use negative commands to spin in the other direction. (and of course you can use the instruction to spin anything, rather than just a shape - such as the camera)

One important thing to remember is that although it rotates the object, the axis rotate with it. For example you designed a rocket pointing up and then rotated it until it was pointing downwards, if you then move the shape in the Y plane (up & down) down would become up !!. It will make sense once you've played around with the examples...

```
UpdateWorld
RenderWorld
```

The UPDATEWORLD command is responsible for updating the coordinates of the entities in our world that may have moved from their last position. (such as moving, rotating and scaling objects) - and controlling any collisions that may have been setup. Without using this instruction, your world would be a very static place. Lastly we use RENDERWORLD to display the scene into our double buffer ready for us to flip into sight with the FLIP instruction.

```
Text 320,500,"First Blitz3D Program"
Flip
Wend
End
```

This last section of code firstly prints a message to the screen, (which must be done every update in the loop as the buffer is cleared every time the loop repeats). This is a good time to do any other drawing to the screen such as using commands from the original 2d version of blitz. (such as scores or special effects). Now we flip the buffer onto the screen using the old FLIP instruction. The WEND command marks the end of our repeating loop, lastly we come to the END, which as we already know quits the program. (so we must have pressed the ESC key to reach this part in our program)

That's it, we have successfully reached the end of our first program

Movement & Rotation

Aim: Move an object around 3d space
Skill level: Beginner
Files Needed: movement.bb / rocket.3ds

Lets start to get things moving, in this tutorial we will be loading an object which I made up in 3D Studio. Granted its not wonderful and looks more like a candle than a state-of-the-art rocket, but it will do us.

Let's take a look through the full source, feel free to run it and have a play !.

NOTE: You will need to have the object ROCKET.3DS in the same directory as the sourcecode for Blitz to find it

You can move the rocket forwards with the **UP** cursor, and rotate it with the **LEFT** and **RIGHT** cursor keys.

```
Graphics3D 800,600

SetBuffer BackBuffer()

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

rocket=LoadMesh( "rocket.3ds" )
PositionEntity rocket,0,0,7

While Not KeyHit(1)

If KeyDown(200) Then
MoveEntity rocket,0,0.05,0
EndIf

If KeyDown(203) Then
TurnEntity rocket,0,0,1.0
EndIf

If KeyDown(205) Then
TurnEntity rocket,0,0,-1.0
EndIf
```

```
UpdateWorld
RenderWorld

Text 320,500,"Movement & Rotation"

Flip

Wend
End
```

I won't explain every line of code, just the new commands which we have yet to use. First up is:

```
rocket=LoadMesh("rocket.3ds")
```

As you can probably guess, this line of code will load a 3d model called ROCKET.3DS into a handle variable called 'rocket'.

What's a handle variable ?, its exactly the same type of label we use in normal blitz for initializing sprites. (It holds the address in memory of where the data is stored), in this case the variable rocket points to the location of where the model is stored.

```
If KeyDown(200) Then
MoveEntity rocket,0,0.05,0
EndIf
```

These 3 lines tell us that if you are pressing the UP cursor move the object 'ROCKET' in the Y axis 0.05 units upwards.

Let's take a more indepth look at this, remember my rocket is pointing up - so to start with I want my rocket to travel upwards - hence why I am increasing the Y coordinate. (remember in 3D a positive Y means up - not down like in normal coordinate systems, such as plotting sprites in standard Blitz)

If my rocket was a car, that was designed to be facing towards the right - then I would want my car to travel to the right of the screen, (it goes forwards) - so I would be increasing the X coordinate. It's very important to get this right at the beginning of your program, or your shape could start moving in the wrong direction to what you want it to.

```
If KeyDown(203) Then
TurnEntity rocket,0,0,1.0
EndIf
If KeyDown(205) Then
TurnEntity rocket,0,0,-1.0
EndIf
```

It should be quite obvious to you that these lines control the rotation of our rocket. If you press the LEFT cursor - turn the rocket 1.0 unit clockwise, whereas pressing the RIGHT cursor results in turning -1.0 units anti clockwise.

So how comes the shape still moves forward even though we are no longer pointing up ? .. well.. you've got to thank Mr Sibly for giving us the very easy TurnEntity command !. You see as we turn the object, the axis move with it. If we rotate the shape 90 degrees, the Y axis is now pointing where the X axis should be. Don't worry why or how.. just be thankful you don't have to work out any nasty calculations to do it.

Just incase you where wondering, you can rotate the shape without the axis moving by using the RotateEntity command. This would be used mainly for placement of objects in your world - for example, if I designed the rocket on its side. I would firstly rotate it till its standing upright with the Rotate command - then I could use the turn command to move it.

And that's about it really.. The only way of getting to know this command, (which will probably be your most widely used command) - is to experiment. Try getting the rocket to rotate in other directions !!

remember here's the format you use:

```
TurnEntity [what object it is],
    X [amount to rotate about the X axis],
    Y [amount to rotate about the Y axis],
    Z [amount to rotate about the Z axis]
```

NOTE: Blitz3D also contains a command for scaling an object, (see the doc's for info on SCALEENTITY) - if for instance we designed it too small we could increase it on-the-fly, or shrink it if its too large. If you choose to use a multitude of different 3d Packages to design your objects with, then you will need to use this command quite a lot. 3D Studio for instance, likes to scale your object much larger than other packages.

Camera Movement

Aim: Move the camera around 3d space
Skill level: Beginner
Files Needed: camera.bb / house.3ds

So far we have just used a static camera, which just sits there - pointing at the screen. But in the real world of PC games - we will want it to follow the main character around, (if a third person view is required - such as Tomb Raider) or perhaps give us a more personal view such as a first person view (used in Quake or Unreal).

Take a good look at Mr Sibly's 'Marko' demo for a great example of an intelligent camera. The code to control your camera can be just as complicated as the main game code... if you want it to be !

Let's have a look at a basic example:

```
Graphics3D 800,600

SetBuffer BackBuffer()

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

house=LoadMesh( "house.3ds" )
RotateEntity house,0,90,0

While Not KeyHit(1)

If KeyDown(200) Then
MoveEntity camera,0,0,1
EndIf

If KeyDown(208) Then
MoveEntity camera,0,0,-1
EndIf

If KeyDown(203) Then
TurnEntity camera,0,1.0,0
EndIf
```

```
If KeyDown(205) Then
TurnEntity camera,0,-1.0,0
EndIf

UpdateWorld
RenderWorld

Text 335,500,"Camera Movement"

Flip

Wend
End
```

Using the cursor keys you can explore the environment, The UP & DOWN cursors moving forward and back in space - the LEFT & RIGHT cursors panning the camera around on the spot.

There's not really much more to it, we have already covered movement and rotation in the previous tutorial. The only difference here is that we are moving forward into the screen. Just as we moved the shape, we use the same MoveEntity and TurnEntity commands to move the camera, That's what makes B3D such a great language to learn !.

Instead of having to remember loads of different commands for objects, cameras and lights - we just use the same instructions.

Perhaps you want to make the view bob up and down, such as the walking motion in Quake or Doom.. easy enough, just create a loop that moves the height of the camera up and down. I'll leave the actual coding part up to you !.

By Default Blitz3D will stop drawing an object if it is too far away from the camera, this is known as the CAMERA RANGE. Anything past its region will not be drawn. This is very handy if we have alot on screen, anything far into the distance will not show - meaning that we gain some extra processing time.

BUT WAIT... Blitz3D can also give us a fog effect !. We've all seen games that use it, (Unreal or example). The further the object is away from the camera the less we can see it. For more info on these functions see the doc's under Camera Commands. Used correctly, the fog commands can give a 3D world something of a sinister lifelike look !

Object Animation

Aim:	To animate an object
Skill level:	Intermediate
Files Needed:	animation.bb
	gargoyle.md2

So far we have just used static objects, its Time to liven up our 3d world and think about animation. Blitz will load in animated meshes from a variety of sources, including X. format (DirectX), MD2 (used by the Quake 2 engine) and lastly 3DS (3D Studio).

How do you draw and animate an object ?, although B3D has a few commands for building shapes such as the cube and sphere instructions - really we need to use a 3d modeller program to create our object.

Most 3d packages will export/import just about every format you are going to ever need, personally I like to work with the Directx format (X). What modeller you choose to use really depends on your preference, Truespace (www.caligari.com), Rhino 3D (www.rhino3d.com) and Canvas 3d (www.amegia.com) - all support X format without any problems.

Recently I've been using the AC3D modeller for my modelling needs, since its simple but powerful. You can download a fully working trial version from official website - (www.comp.lancs.ac.uk/computing/users/andy/ac3d.html), but of course designing an object that has realistic movement really is an art form all in itself. Most of us including me - are going to find it a struggle, luckily places such as 3dcafe (www.3dcafe.com) contain hundreds of ready made free models for us to use.

Time for the source:

```
Graphics3D 800,600

SetBuffer BackBuffer()

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

man=LoadMD2( "gargoyle.md2" )
PositionEntity man,0,-35,600
RotateEntity man,0,180,0

AnimateMD2 man,1,.1,32,46
```

```
While Not KeyHit(1)

If dist<970 MoveEntity man,.5,0,0
If dist=970 AnimateMD2 man,1,.05,0,31

dist=dist+1

UpdateWorld
RenderWorld

Text 320,500,"An Animated MD2 Demo"

Flip

Wend
End
```

So whats going on ?

```
man=LoadMD2( "gargoyle.md2" )
PositionEntity man,0,-35,600
RotateEntity man,0,-90,0
```

Here, I've used the LoadMD2 command to load an MD2 animated object into the pointer handle 'man'. MD2 models can be found just about anywhere on the net, they are the staple diet of Quake 2 - So as you'd expect there's thousands of ready made objects with animation out there just waiting to be included in your epic. (Although be careful to read any copyright messages that may be attached to them).

Unless you plan on using MD2 models all the time, I expect you will be using the LOADMESH command which does exactly the same thing as the LOADMD2 instruction. (but allows loading of X and 3DS animations)

We wont be going into any of the details that involve creating animations from a modeller, but if your interested I'm sure whatever product you use will contain some sort of tutorial for creating animations. (You should be able to find some tutorials for 3d modelling from my Gamecoding website very soon)

The frames of animation (keyframes) are loaded in with the model, all we have to do is tell B3D where to start - and where to end !. Before we look at this in action, you will notice that I've positioned and rotated the object to a new starting point. Originally the character was designed facing Right, so we need to rotate it by 90 degrees - so it faces us. (I did this by using the RotateEntity command)

AnimateMD2 man,1,.1,32,46

This instruction informs B3D that we want the object 'man' to animate in a loop (so the walking frames can restart), at a speed of .1 of a unit, starting at frame 32 - and ending in frame 46.

When we have set the command up, every update (depending on the animation speed you have selected) the animation will run by itself without us having to do any more to it.

If we were using a regular animated object (X or 3DS format) - we would use the ANIMATE instruction, which works exactly like the ANIMATEMD2 command I've used here in this example.

As you can see by looking through the source, I've setup a loop that continues to move the shape forwards until the counter is equal to 970. (using the MOVEENTITY command)

After that the animation is reset to display the standing stance animation.. (frames 0-31). Why not alter the program to include rotation, so that the character can move about freely.

Although I've setup the animation to loop, we can of course just have it run through the animation loop once if we had wanted to it. To do this you just change the MODE flag in the Animate instruction. Here's the instruction in full:

Animate [entity name], [mode 0,1,2 or 3], [speed], [frame to begin at], [frame to end at]

in the case of MD2's you'd use the instruction:

AnimateMD2 [entity name], [mode 0, 1,2 or 3], [speed], [frame to begin at], [frame to end at]

Just incase your interested, here's a rundown of all the mode switches we can use: (Remember we used the loop anim mode)

0: Stop Anim
1: Loop Anim
2: Ping-Pong Anim
3: One-Shot Anim
NOTE: To play an anim backwards use a negative speed value

B3D thankfully makes animation very easy, it really is just a case of setting up whatever we need the anim to do first using the various flags in the ANIMATION instruction before setting it running. However to make the animation look convincing you need to get the timing right. How many times have you looked at a game only to see the main hero character float-walking over the floor rather than connecting with it ?.

But that's half the fun of it !, as you'll see a little effort can really make a big difference.

B3D contains a large array of animation commands that will suit every purpose you will ever need, but tucked away in the depths you'll find a couple of incredible commands that can morph animation frames from one animation to another. For instance, an object of a man running can switch smoothly to an anim of him standing still. If you run this example a few times and watch the point that the object comes to a halt you'll notice the jump. But, by using these commands we could smooth out the change so it would be unnoticeable. Although I wont be using these commands in this tutorial, I may add a new tutorial on the subject very soon.

So what are you waiting for !, go and design the next Quake !!

Texturing

Aim:	How to texture an object
Skill level:	Intermediate
Files Needed:	texture.bb
	blitztexture.bmp

For our objects to look more realistic we really need to texture them, but what is texture mapping?

Think if it as a way of adding extra details to an object. For example... You design an ordinary white cube, doesn't look much does it ?

But add a texturemap to it, and suddenly you have a BORG cube !.

Textures are bitmap pictures that have been designed with a paint program (that can save BMP format files.)

Another example... If you designed a basic model of a creature, using a texturemap you could add all the external features - such as clothes and facial expressions.

Although B3D has alot of commands to manage multiple textures and special effects, for now we shall just be using the very basic EntityTexture command.

Of course we could save ourselves alot of time by designing our objects and applying the textures to them directly from a 3d modelling program, but that would make this tutorial a waste of time !.

So why would we want to apply a texture in realtime ?, well for starters a new texture may be put onto the object at any time. Take a wall, shoot it ! - now for things to look realistic in our 3d world, we would like to have somekind of indication that it was shot. We could do this by loading a texture of a bullet hole directly onto the wall. Every 3d game currently on the market uses texture trickery in some form or another, as your experience grows - so will your imagination !. Why have a flat river, when you could use texturemaps to give the impression of ripples or waves.

B3D has so many lovely features for us to use including the new ANIMATED textures command, we will be using this later on !. (Quake I/II/III used this trick to give us the impression of moving water)

Let's look at this in practice:

```
Graphics3D 800,600

SetBuffer BackBuffer()

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

cube=CreateCube()
PositionEntity cube,0,0,5

texture=LoadTexture("blitztexture.bmp")
EntityTexture cube,texture

While Not KeyHit(1)

TurnEntity cube,0.1,0.2,0.3

UpdateWorld
RenderWorld

Text 340,500,"Texturing Demo"

Flip

Wend
End
```

As you can see, a texture can make all the difference to a very basic object.

The only 2 lines of code we don't already know are:

```
texture=LoadTexture("blitztexture.bmp")
EntityTexture cube,texture
```

Two nice and friendly commands, the first loads a texturemap into memory - with the pointer variable 'texture' pointing to it.

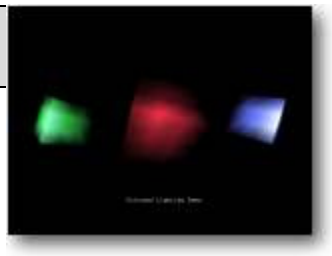
Once we have successfully loaded the texture, we assign it to the object using the **EntityTexture** command. (here we are assigning the variable pointer TEXTURE to the cube we defined earlier)

You could if you wanted load the picture file **BLITZTEXTURE.BMP** into MS Paint and make some changes.. if you wanted to !

And that's all there is to it, but a quick word of advice: If your object isn't textured as it should, always check that the texturemaps are in the right directory. It's all to easy to spend half the day trying to work out a texturing problem, only to find that you moved the program to another directory without dragging the associated texture files with it.

Lighting

Aim:	Lighting a scene
Skill level:	Intermediate
Files Needed:	lights.bb



So we can now move objects (and ourselves) around, texture objects, animate... so what shall we look at now ?

How about lighting, used correctly we can make great use of it to really give our little world ambiance.

By default B3D will already give us a light in our scene, pointing directly into the screen. Think of it as a torch stuck on top of the camera, pointing at whatever we can see.

Sourcecode time:

```
Graphics3D 800,600

SetBuffer BackBuffer()

camera=CreateCamera()
CameraViewport camera,0,0,800,600

AmbientLight 0,0,0

;cube 1
cube=CreateCube()
PositionEntity cube,0,0,5

light=CreateLight(3)
LightColor light,100,20,30
LightConeAngles light,0,45
PositionEntity light,0,0,0.5
LightRange light,8
PointEntity light,cube

;cube 2
cube2=CopyEntity(cube)
PositionEntity cube2,-5,0,8

light2=CreateLight(3)
LightColor light2,40,150,60
```

```
LightConeAngles light2,0,45
PositionEntity light2,-5,0,4.5
LightRange light2,8
PointEntity light2,cube2

;cube 3
cube3=CopyEntity(cube)
PositionEntity cube3,5,0,8

light3=CreateLight(3)
LightColor light3,70,80,190
LightConeAngles light3,0,45
PositionEntity light3,5,0,4.5
LightRange light3,8
PointEntity light3,cube3

While Not KeyHit(1)

TurnEntity cube,0.1,0.2,0.3
TurnEntity cube2,0.3,0.2,0.1
TurnEntity cube3,0.3,0.2,0.1

UpdateWorld
RenderWorld

Text 310,500,"Coloured Lighting Demo"

Flip

Wend
End
```

There's quite a few new commands to look at here.

Remember that torch I spoke about, well the first new command we will learn is strictly for changing the colour of the beam. (so to speak)

AmbientLight 0,0,0

This sets the light to black, so that any additional lights we use will show up with full effect. First thing to remember when using this command is that the colour values range from 0 to 255, and we have 3 separate colour shades to think of RED, GREEN & BLUE. (the same as a TV)

By changing these values we can just about get any colour we want, such as setting them all to 255 will result in the colour white. It's usually easier to work out the values you will need from a paint program such as Paintshop Pro.

light=CreateLight(3)

B3D at present lets us use three different types of lighting. As with the ANMATE instruction these are selected with flags, these are:

> *1: AREALIGHT - Which will light the whole surrounding area around it.*
> *2: POINTLIGHT - A directional light that can light a specific area*
> *3: SPOTLIGHT - Very similar to POINT, but can give off more light on a subject area than using POINTLIGHT.*

For this demo I chose to use SPOTLIGHT.

LightColor light,100,20,30

Same as the AMBIENTLIGHT instruction, this command lets us select the colour of the light. Here I've set it up so that RED=100, GREEN=20 and BLUE=30 - which gives us a nice reddish glow.

LightConeAngles light,0,45

We use this instruction to control the angle of light that comes from our spotlight. In this case I've set it up to 45 degrees, In-other-words.. The light has a radius of 45 degrees, any part of the object that is outside of this will not be lit.

PositionEntity light,0,0,0.5

We move the light into position.

LightRange light,8

This controls the distance of the light, I set it at 8 units.. If I wanted the light to shine further into the distance then I would increase this value.

PointEntity light,cube

This command will point the light at our cube object. No matter where are light is positioned it will point towards the cube. But if you then move the object or light - you will have to repoint it with this command.

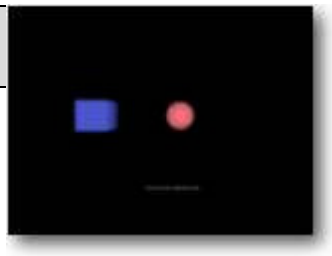
Something else you should think about is whether the light and object can see each other. (the LIGHTRANGE instruction)

I won't go into details with the rest of the program as you should be able to work out what is happening without any additional help.

Where's the fancy lensflare commands, I can hear you cry. Well at present B3D doesn't have any readymade, built-in commands - so its really up to you the programmer - to come up with these effects. And believe me it is possible with a little careful effort. I've seen some of the Beta Testers come up with lighting effects that would equal Unreal !. But then this is a subject for another tutorial...

Collision Detection

Aim:	To detect a collision between 2 objects
Skill level:	Intermediate
Files Needed:	collision.bb



In this example I shall be using 2 ready-made entity objects, a Sphere and a Box.

Hopefully The program will display a message once the sphere comes into contact with the box. Run the program first to fully understand what is happening before delving into the sourcecode below.

```
Graphics3D 800,600

Const CUBE_COL=1
Const SPHERE_COL=2

SetBuffer BackBuffer()

camera=CreateCamera()
CameraViewport camera,0,0,800,600
PositionEntity camera,0,0,-5

light=CreatLight()

cube=CreateCube()
PositionEntity cube,-5,0,5
EntityColor cube,70,80,190
EntityType cube,CUBE_COL

sphere=CreateSphere(12)
PositionEntity sphere,5,0,5
EntityColor sphere,170,80,90
EntityType sphere,SPHERE_COL

Collisions SPHERE_COL,CUBE_COL,3,1

While Not KeyHit(1)

MoveEntity sphere,-0.02,0,0

UpdateWorld
```

```
RenderWorld

If EntityCollided(sphere,CUBE_COL) Then
Text 370,80,"Collided !!!"
EndIf

Text 335,500,"Collision Detection"
Flip

Wend

End
```

Being able to check for collisions is perhaps the most major part of any game. After all, without collision detection what's to stop Mario falling through the floor, or the bullets from Max Payne's gun doing its damage. We need it - whether we want it or not !.

If you look carefully through the above example you will notice quite a few new commands. Let's briefly run through the entire program before we look at these.

Firstly we create 2 objects - a Cube and a Sphere.., then we setup the collision so that B3D will check these objects every time the UPDATEWORLD instruction is called.

Slowly we move the Sphere towards the Cube, until they collide.. after that, we print up the collision message on the screen to signal that we have collided.

So what do these new instructions do:

```
Const CUBE_COL=1
Const SPHERE_COL=2
```

Ok, its not an instruction, but just something I've strung together for this example. When programming always try and make things as easy as possible - It will certainly help when it comes to debugging. (working out any problems)

The Collision instructions we have to setup rely on variables, but rather than just using numbers - I'll use the CONSTANT variables I've setup to represent them.

So in the above two lines, I can use the variable names CUBE_COL every time I want to use a 1 - and SPHERE_COL instead of 2. Although of course I could just use the numbers with the instructions, as you'll see it will help us out.

```
EntityType cube,CUBE_COL
```

After we've setup the basic entity object, we need to setup a collision variable for it. We do this by assigning a number to the entity, As you can see to do this we use the ENTITYTYPE command. Here I've set the entity cube to have a value of 1. (remember the CONST variable is set to 1)

```
EntityType sphere,SPHERE_COL
```

As before, we setup the sphere collision variable to have a value of 2.

AN IMPORTANT NOTE TO REMEMBER !
Every entity **DOES NOT** have to have a separate collision variable number. For example say we created a 3d maze game, that had 10 objects for the sides of the maze. We would want to check if we have collided with a wall, it wouldn't matter which wall.. just a wall. So every wall object(entity) would have the same collision variable.

I would use the code:

```
Const WALL=1
EntityType wall1,WALL
EntityType wall2,WALL... etc
```

Later on when we check for a collision we would just say, is there a collision with the wall ?. Nice and Easy isnt it ?

```
Collisions SPHERE_COL,CUBE_COL,3,1
```

Now the fun begins, this is the main instruction that informs B3D which objects to check for collisions and what action it should take.

The first part of the line "**Collisions SPHERE_COL,CUBE_COL**,3,1", is saying that we want a check to take place between the collision markers 1 and 2. (Remember the Sphere is 1 and the Cube is 2)

If we had more entities with the same collision marker value, then of course these too would be included.

"Collisions SPHERE_COL,CUBE_COL,**3**,1" - The first value (3), represents the type of collision that we want B3D to perform, in this case we are using mode '3' - which is a Sphere-to-Box collision.

B3D has 3 different types of collisions we can perform, these are:

```
1 = Sphere-to-Sphere
2 = Sphere-to-Polygon
3 = Sphere-to-Box
```

Now we come to the last value, "Collisions SPHERE_COL,CUBE_COL,3,**1**". This is the response value, it signals what B3D should do when a collision has taken place. I used the value 1 which is used for a dead stop. (when it collides with something, don't let it move any closer to it)

As before there are 3 mode types we can use:

```
1 = Stop
2 = Slide1 - Full sliding collision
3 = Slide2 - Takes into consideration the angle of slopes
```

Even though in my program I am moving the sphere into the box with the MOVEENTITY command, when it collides (because I've used the STOP mode).. the entity will NOT move through it.

We have one more command to look at, that's the collision check instruction itself "EntityCollided".

```
If EntityCollided(sphere,CUBE_COL) then
Text 370,80,"Collided !!!"
EndIf
```

As you can can probably guess, this instruction (imbedded in an IF statement) - is checking the entity SPHERE for a collision with the collision marker 1 (the cube).

If it has collided then print the message to signal a collision !.

Ok, we have now run through the entire program - but did it make much sense to you ?, To begin with I couldn't grasp it at all !. The best way to understand the various collision instructions is to experiment yourself with the different mode settings. Eventually (if you haven't understood fully by now).. you will realize just how easy and powerful they can be.

Vertexes

Aim:	To edit a meshes vertexes in Real-time
Skill level:	Intermediate
Files Needed:	vertex.bb plane.3ds

If you've already read the section on Meshes beforehand then hopefully you'll know most of what we will be doing already in this tutorial, if not - go away and read it first !.

let's recap:

- Each **MESH** (a 3d object), is made up of SURFACES.
- Each **SURFACE** has a BRUSH.
- Each **BRUSH** can be assigned 8 different texturemaps.
  (which can be overlaid on each other to create new effects)
- Each SURFACE is made up of TRIANGLES.
- Each **TRIANGLE** is made up of 3 VERTEXES.

So, armed with that info - you should know what makes a 3d object tick !. Lets take a flat square as an example, it is made up of 4 vertexes and 2 triangles. What we are planning of doing is to take 2 of those vertexes and change their coordinates.

Infact as mentioned in the Introduction to Meshes, we can even change the colour of the vertexes in realtime too. Run the example - what you can hopefully see is a square object (which is slowly spinning on the Z plane), being pulled out of shape in 2 corners - while every-so-often the colours change.

It's a very easy effect to create, I wont go into great detail about how/why the program works - but here's a quick rundown if your interested:

We setup the variable 'COUNTER', which does exactly that.. to be used as a counter. Every time the program runs through its main loop, it is incremented. Based on what value the counter is equal to, corresponds to what direction we should pull the vertexes. If the counter reaches 1000 then change the colour of each vertex to a random selection, before resetting the counter value.

Let's take a look:

```
Graphics3D 800,600

SetBuffer BackBuffer()
```

```
camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

plane=LoadMesh("plane.3ds")
PositionEntity plane,0,0,25
EntityFX plane,2

surface=GetSurface(plane,CountSurfaces(plane))

VertexColor surface,0,255,0,0
VertexColor surface,1,0,255,0
VertexColor surface,2,0,0,255
VertexColor surface,3,255,0,255

While Not KeyHit(1)

TurnEntity plane,0,0,.3

counter=counter+1

If counter<500 Then
x1#=-.01
y1#=-.01
x2#=+.01
EndIf

If counter>499 Then
x1#=+.01
y1#=+.01
x2#=-.01
EndIf

xx#=VertexX(surface,0)
yy#=VertexY(surface,0)
zz#=VertexZ(surface,0)

VertexCoords surface,0,xx+x1,yy+y1,zz

xx#=VertexX(surface,2)
yy#=VertexY(surface,2)
zz#=VertexZ(surface,2)

VertexCoords surface,2,xx+x2,yy+y1,zz
```

```
If counter=1000 Then
counter=0
VertexColor
surface,0,Rnd#(0,255),Rnd#(0,255),Rnd#(0,255)
VertexColor
surface,1,Rnd#(0,255),Rnd#(0,255),Rnd#(0,255)
VertexColor
surface,2,Rnd#(0,255),Rnd#(0,255),Rnd#(0,255)
VertexColor
surface,3,Rnd#(0,255),Rnd#(0,255),Rnd#(0,255)
EndIf

UpdateWorld
RenderWorld

Text 350,500,"Vertex Control"

Flip

Wend
End
```

So how do we get at the vertexes of the object ?

Well for starters we load the object with the LoadMesh command, the object we are loading is of course called Plane.3ds.

EntityFX plane,2

Now here's a new command we haven't seen before !, this command is really more of mode switch than anything else. But setting values we can access the entity in different ways. the mode value '2' is to able vertex colouring on the whole entity, by default this is turned off.

Here's those mode settings:

1 = Full-Bright
2 = Use Vertex Colours
4 = Flatshading
8 = Disable Fog

There is another command very similar to EntitiyFX called BRUSHFX. This uses the same mode settings, but instead of changing the whole entity will work on a single brush. (remember a mesh has surfaces, with brushes applied to them)

surface=GetSurface(plane,CountSurfaces(plane))

In order to get at the vertexes we must first unlock them, we do this by creating a pointer variable that holds the memory address to the surfaces on the mesh.

Calm down !, we don't have to get our hands dirty calling with lots of nasty math's - instead we just use the GETSURFACE command, which likes us to pass firstly the mesh name - and secondly the amount of surfaces it has. As you can see I've cheated and used the COUNTSURFACES command to do this for me.

```
VertexColor surface,0,255,0,0
VertexColor surface,1,0,255,0
VertexColor surface,2,0,0,255
VertexColor surface,3,255,0,255
```

Before going into the main loop, I've set the colour of each vertex to a different colour. This gives us a nice rainbow effect !. As you can see we pass the pointer variable SURFACE to the VERTEXCOLOR command, as well as the vertex number (0-3, since our object only has 4 points) - followed by the colour values for the Red, Green and Blue shades. (must be in the range of 0 (Dark) through to 255 (Light))

```
xx#=VertexX(surface,0)
yy#=VertexY(surface,0)
zz#=VertexZ(surface,0)
```

Since I want the coordinates of the mesh to change all the time, I cant set it with a value that doesn't change. Every update I've got to get the current coordinates and slightly update them (by adding an offset to the X and Y coords).

I do this by firstly, getting the current X,Y and Z vertex coords - using the various get vertex commands.

VertexX(surface,0) - gives us access to the X coordinate of the object surface, at vertex 0.

Just as, VertexY(surface,99) - would give us access to the Y coodinate of vertex 99 !!!.

```
VertexCoords surface,0,xx+x1,yy+y1,zz
```

As you've probably worked out by now, this is the main instruction for changing the actual Vertex positions. It needs to be called with the Surface pointer value, followed by the new values for the X, Y and Z positions.

And that's all there is to it !!

But why would you want to change the coordinates ?

All games will alter their objects, its just a case of working out how, and where they do it. Imagine you've just written a driving simulation.. wouldn't it be nice when you crash the car to reflect the damage ?. Perhaps crumple that fender.. or crack that window.

Just like a certain other car game currently in the charts, they use exactly the same method. You gotta hand it to B3D - You want it.. it's there, now go and use it wisely !.

For the latest release of

Blitz Basic 3D

visit

www.blitzbasic.com