By King (http://www.gizzedinc.com)

OpenGL|ES Tutorials (4): Textures & Lighting

Textures

Instead of using colours, it is possible to place a picture onto a side of a cube. In this tutorial we will load a bitmap file (.bmp) and map it onto the sides of our cubes from the last tutorial.

Introduction

This tutorial uses the code from the previous tutorial. We will be adding two functions required to load the bitmaps, and the textures, and some modification to other various parts of the code as well.

Declarations

```
1:      #include <windows.h>
2:      #include <GLES\gl.h>
3:      #include <GLES\egl.h>
4:      #include <KGSDK\framework.h>
5:      #include <KGSDK\frameworkGL.h>
6:
7:      static float rot[3];
8:      static float loc[3];
9:
10:     float xrot = 0.0f;
11:     float yrot = 0.0f;
12:
13:     GLuint texture[1];          //NEW - Storage for our textures
```

Analysis: Only an extra variable has been added here. This is storage for our textures (somewhere to hold them).

Function: LoadBitmap()

```
1:      unsigned char *LoadBitmap(char *filename, BITMAPINFOHEADER *bmpInfo)
2:      {
3:        FILE *file;                              //Actual File
4:        BITMAPFILEHEADER bmpFileHeader;          //Bitmap File Header
5:        unsigned char *bmpImage = NULL;          //Actual Bitmap Image
6:        unsigned char tmpRGB;                    //Temporary storage for when
          converting from BGR to RGB
7:
8:        TCHAR path[256];                         //Current Working Directory
9:        char fullPath[256];                      //Full Path
10:
11:       GetModuleFileName(NULL, path, 256);  //Get Current Working Directory
12:       TCHAR *pos = wcsrchr(path, '\\');    //Find last \\ so that we can
remove the last part
13:       *(pos + 1) = '\0';
```

```
14:         wcstombs(fullPath, path, 256);   //Convert Wide characters to multi-byte
15:         strcat(fullPath, filename);      //Combine path with filename
16:
17:         file = fopen(fullPath,"rb");     //Open File
18:         if (!file)
19:         {
20:           MessageBox(NULL, L"Can't Find Bitmap", L"Error", MB_OK);
21:           return NULL;
22:         }
23:
24:         fread(&bmpFileHeader,sizeof(BITMAPFILEHEADER),1,file); //Read header
file into structure
25:
26:         if (bmpFileHeader.bfType != 0x4D42)  //Check to see if the file is a bitmap
27:         {
28:           MessageBox(NULL, L"Incorrect texture type", L"Error", MB_OK);
29:           fclose(file);
30:           return NULL;
31:         }
32:
33:         fread(bmpInfo,sizeof(BITMAPINFOHEADER),1,file); //Read header info
into structure
34:
35:         fseek(file,bmpFileHeader.bfOffBits,SEEK_SET);   //Move to start of
image data
36:
37:         bmpImage = new unsigned char[bmpInfo->biSizeImage];
38:
39:         if (!bmpImage)
40:         {
41:           MessageBox(NULL, L"Out of Memory", L"Error", MB_OK);
42:           delete[] bmpImage;
43:           fclose(file);
44:           return NULL;
45:         }
46:
47:         fread(bmpImage,1,bmpInfo->biSizeImage,file);    //Load image data
48:
49:         if (!bmpImage)
50:         {
51:           MessageBox(NULL, L"Error reading bitmap", L"Error", MB_OK);
52:           fclose(file);
53:           return NULL;
54:         }
55:         //Convert BGR -> RGB
56:         for (unsigned int i = 0; i < bmpInfo->biSizeImage; i+=3)
57:         {
58:           tmpRGB = bmpImage[i];
59:           bmpImage[i] = bmpImage[i+2];
60:           bmpImage[i+2] = tmpRGB;
61:         }
62:
63:         fclose(file);                        //Close File
64:
65:         return bmpImage;                     //Return Bitmap Image
66:     }
```

Analysis: Before I start analysis this function, you need to now a little about Bitmap files. The first part is the information which can be loaded into a BITMAPFILEHEADER structure. This contains information about the file type, and where the actual image data is stored.

The next part contains information such as the height, and width of the bitmap. This information can be stored into BITMAPINFOHEADER structure.

Finally, after these two headers is the actual image data.

The function above takes two parameters. The first parameter is a pointer to the filename of the bitmap you want to load (Note that this has to be in the same directory as the exec). The second parameter is a pointer to an Information Header. We will need this as we need to know information, such as how wide or tall the image is.

A number of variables are created. The first is a pointer to a FILE Structure, so that we can open the file. The next variable is created to store the Bitmap File Header (BITMAPFILEHEADER). The Actual bitmap image data can be stored as a number of unsigned chars. Variable bmpImage holds the Actual Image data. Because bitmaps store pixel data in BGR format, we need the next variable, tmpRGB, so that we can convert from BGR to RGB. (Blue Green Red → Red Green Blue). Variable path is created to store our current working directory. Variable fullpath, is just variable path, combined with the filename we are passed into this function. This gives the complete path to the bitmap image.

The first thing we do, is retrieve the current working directory, for this we use function: GetModuleFileName. This first parameter is the module, NULL is passed, which means the current module is used. The second parameter specifies where to store it, and the third parameter specifies the maximum number of characters to load.

The problem with using GetModuleFileName is that not only does it return the current working directory, but it also returns the current exec we're running. On line 12 we find the last "\\", and on line 13 we remove it by adding a null-terminated character to just after where we find the "\\" to be, effectively removing anything after that point.
The next function on line 14: wcstombs, simply converts all the wide-characters, from variable path, into multi-byte, and stores it in variable fullpath.

On line 15, we combine the filename we are passed, with the current working directory (fullpath).

On lines 17 -22 we attempt to open the file. If it fails opening, a message box is displayed, stating the error, and then the function immediately stops.

If the file opened successfully, we go onto load the bitmap file header into the structure. A bitmap has a file ID of 0x4D42. The file ID is stored in the bfType variable. We check to see if the file we are loading is actually a bitmap on lines 26 – 31. If it is not, we immediately display an error message, close the file, and quit the function.

Next we load the header info structure on Line 33. On Line 35 we move the file pointer from the start of the file to the start of the image data (SEEK_SET). The bfOffBits variable stores how much bits there actually are in the image data.

On Line 37, memory is allocated for the image data. The size of the actual image data is held in the info header structure. Lines 39 – 45 check whether the memory storage was created successfully, if not the function is exited immediately.

On line 47 we load the actual image data. We check to see whether it was successfully loaded on lines 49 to 54.

As aforementioned, bitmaps store pixels in BGR data. We need to convert this to RGB, which is done on lines 56 to 61. We loop through every the pixels in sets of three's and swap the third value with the first value, hence BGR → RGB.

Finally, we close the file on line 63, and return the bitmap image on line 65.

Function: LoadTextures()

```
1:      bool LoadTextures()
2:      {
3:        BITMAPINFOHEADER info;
4:        unsigned char *bitmap = NULL;     //Pointer to point to image data
5:
6:        bitmap = LoadBitmap("King.bmp", &info);   //Load Bitmap
7:
8:        if (!bitmap)
9:        return false;
10:
11:       glGenTextures(1, texture);    //Generate Texture Identifiers
12:
13:       glBindTexture(GL_TEXTURE_2D, texture[0]);     //Select texture[0]
14:
15:       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info.biWidth, //Set Texture
Properties
16:         info.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
17:         bitmap);
18:
19:       glTexParameterf(GL_TEXTURE_2D,     //Set further texture properties
20:         GL_TEXTURE_MIN_FILTER, GL_LINEAR);
21:       glTexParameterf(GL_TEXTURE_2D,
22:         GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

```
23:
24:         delete[] bitmap;
25:
26:         return true;
27:     }
```

Analysis: Now that we have our LoadBitmap function complete, we can load the textures. The first variable declared on line 3 is for the info header, and the second variable on line 4 is a pointer to the image data. On line 6 we load the bitmap "King.bmp" (Using function LoadBitmap) and store it in the bitmap variable. Immediately after, we check whether the bitmap was loaded successfully, if not, we quit. Note that in OpenGL|ES the textures are required to have dimensions, which are base powers of two (i.e. 2, 4, 8, 16, 32, 64 etc...)

Every texture has a unique identifier that we use to refer to it. To generate a unique identifier for our texture we use the GL command: glGenTextures. This function takes two parameters, the first is how many texture names to generate, and the second is a pointer to an array of unsigned integers, which is where the generated texture names are stored.

Now we have generated the texture identifiers, we select which one we're going to use. For this we use glBindtextures. The first parameter this takes has to be GL_TEXTURE_2D. OpenGL|ES does not support 3D or 1D textures (Although, 1D textures can be achieved by setting the height to 1). The second is the texture identifier.

The next statement, function glTexImage2D, sets some of the textures properties. This function takes a number of parameters.

- Firstly, the first parameter has to be GL_TEXTURE_2D, as aforementioned, OpenGL|ES does not support 1D or 3D textures.
- The second parameter specifies the level of detail. This is only used with mipmaps where different textures are generated depending on the distance of the texture from the viewer.
- The third parameter is the colour components in the texture. Valid flags are: GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE or GL_LUMINANCE_ALPHA. You will generally only use the GL_RGB or GL_RGBA flags. GL_RGBA is only used with .tga files, which contains alpha values.
- The fourth parameter and fifth parameter is the texture width and height, respectively. This can be taken from the info header of the bitmap, as discussed earlier in this chapter.
- The sixth parameter is the border – this must also be 0. OpenGL|ES does not support texture borders either.
- The seventh parameter must be same as the third parameter (GL_RGB)

- The eighth parameter specifies the data type is being used to store the image (GL_UNSIGNED_BYTE)
- The ninth parameter specifies where the image is stored.

glTexParameterf is used to set various other properties. The first parameter has to GL_TEXTURE_2D as mentioned earlier in this tutorial. The next parameter can be GL_TEXTURE_MAG_FILTER or GL_TEXTURE_MIN_FILTER. These specify how your textured is viewed at different magnifications. The last parameter is set to GL_LINEAR. This is the best filter. You don't need to worry much about this entire function for now. Filters will be discussed in the next tutorial.

Finally we clean up, and return true.

Init()

```
1:      void Init()
2:      {
3:          if (!LoadTextures())   //Load Textures, if fails
4:          {
5:              MessageBox(NULL, L"Error loading textures", L"Error", MB_OK);
6:          }else
7:          {
8:              glEnable(GL_TEXTURE_2D);
9:          }
10:
11:         glLoadIdentity();
12:
13:         GLfixed mat[4][4];
14:
15:         float nearz, farz;
16:
17:         farz  = 300.0f;
18:         nearz = 0.01f;
19:
20:         memset(mat, 0, sizeof(mat));
21:         mat[0][0] = (int) (65536.0f * 2.4f);
22:         mat[1][1] = (int) (65536.0f * 3.2f);
23:         mat[2][2] = (int) (65536.0f * (farz / (farz - nearz)));
24:         mat[2][3] = (int) (65536.0f * 1.0f);
25:         mat[3][2] = (int) (65536.0f * ((-farz * nearz) / (farz - nearz)));
26:
27:         glMatrixMode(GL_PROJECTION);
28:         glLoadMatrixx(&mat[0][0]);
29:
30:         rot[0] = 0;
31:         rot[1] = 0;
32:         rot[2] = 0;
33:         loc[0] = 0;
34:         loc[1] = 0;
35:         loc[2] = 3;
36:     }
```

Analysis: The only modification here is the additions of lines 3 to 9. All we do here is Load our Textures on line 3; if it fails we display an error message. If not, we go on to enable 2D Textures using the glEnable command.

Function: DrawCube()

```
1:      void DrawCube()
2:      {
3:        GLfloat cube[] =
4:        {
5:        // FRONT
6:        -0.5f, -0.5f,  0.5f,
7:         0.5f, -0.5f,  0.5f,
8:        -0.5f,  0.5f,  0.5f,
9:         0.5f,  0.5f,  0.5f,
10:       // BACK
11:       -0.5f, -0.5f, -0.5f,
12:       -0.5f,  0.5f, -0.5f,
13:        0.5f, -0.5f, -0.5f,
14:        0.5f,  0.5f, -0.5f,
15:       // LEFT
16:       -0.5f, -0.5f,  0.5f,
17:       -0.5f,  0.5f,  0.5f,
18:       -0.5f, -0.5f, -0.5f,
19:       -0.5f,  0.5f, -0.5f,
20:       // RIGHT
21:        0.5f, -0.5f, -0.5f,
22:        0.5f,  0.5f, -0.5f,
23:        0.5f, -0.5f,  0.5f,
24:        0.5f,  0.5f,  0.5f,
25:       // TOP
26:       -0.5f,  0.5f,  0.5f,
27:        0.5f,  0.5f,  0.5f,
28:        -0.5f,  0.5f, -0.5f,
29:        0.5f,  0.5f, -0.5f,
30:       // BOTTOM
31:       -0.5f, -0.5f,  0.5f,
32:       -0.5f, -0.5f, -0.5f,
33:        0.5f, -0.5f,  0.5f,
34:        0.5f, -0.5f, -0.5f,
35:       };
36:
37:       GLfloat texCoords[] = {
38:       // FRONT
39:       0.0f, 0.0f,
40:       1.0f, 0.0f,
41:       0.0f, 1.0f,
42:       1.0f, 1.0f,
43:       // BACK
44:       1.0f, 0.0f,
45:       1.0f, 1.0f,
46:       0.0f, 0.0f,
47:       0.0f, 1.0f,
48:       // LEFT
49:       1.0f, 0.0f,
50:       1.0f, 1.0f,
51:       0.0f, 0.0f,
52:       0.0f, 1.0f,
53:       // RIGHT
54:       1.0f, 0.0f,
55:       1.0f, 1.0f,
56:       0.0f, 0.0f,
```

```
57:         0.0f, 1.0f,
58:         // TOP
59:         0.0f, 0.0f,
60:         1.0f, 0.0f,
61:         0.0f, 1.0f,
62:         1.0f, 1.0f,
63:         // BOTTOM
64:         1.0f, 0.0f,
65:         1.0f, 1.0f,
66:         0.0f, 0.0f,
67:         0.0f, 1.0f
68:         };
69:
70:        glRotatef(xrot, 1.0f, .0f, 0.0f);
71:        glRotatef(yrot, 0.0f, 1.0f, 0.0f);
72:
73:         glVertexPointer(3, GL_FLOAT, 0, cube);
74:         glTexCoordPointer(2, GL_FLOAT, 0, texCoords);
75:         glEnableClientState(GL_VERTEX_ARRAY);
76:         glEnableClientState(GL_TEXTURE_COORD_ARRAY);
77:
78:         // FRONT AND BACK
79:        glColor4f(1.0f, 0.0f, 0.0f, 1.0f);  //Color: RED
80:        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
81:        glDrawArrays(GL_TRIANGLE_STRIP, 4, 4);
82:
83:        // LEFT AND RIGHT
84:        glColor4f(0.0f, 1.0f, 0.0f, 1.0f); //Color: GREEN
85:        glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);
86:        glDrawArrays(GL_TRIANGLE_STRIP, 12, 4);
87:
88:        // TOP AND BOTTOM
89:        glColor4f(0.0f, 0.0f, 1.0f, 1.0f); //Color: BLUE
90:        glDrawArrays(GL_TRIANGLE_STRIP, 16, 4);
91:        glDrawArrays(GL_TRIANGLE_STRIP, 20, 4);
92:     }
```

Analysis: A few adjustments have been made to this function. Firstly, and most obviously, an entire new array has been added. This are our texture co-ordinates. Like Colour values, these can range from 0.0f to 1.0f, where 0.0f is the minimum and 1.0f is the maximum. Thus, the co-ordinates 0.0f, 0.0f are the bottom-left of the texture and 1.0f, 1.0f is the co-ordinates for the top-right of the texture. Read through this array, and compare it with the cube array, so you understand what is happening. The next modification is the addition of line 74 which is similar to line 73. glTexCoordPointer has the same parameters as glVertexPointer, except that this time we pass in our vertex array. Also, on just like we enable the client-side capability GL_VERTEX_ARRAY on line 75, we also enable the client-side capability GL_TEXTURE_COORD_ARRAY on line 76.

Output

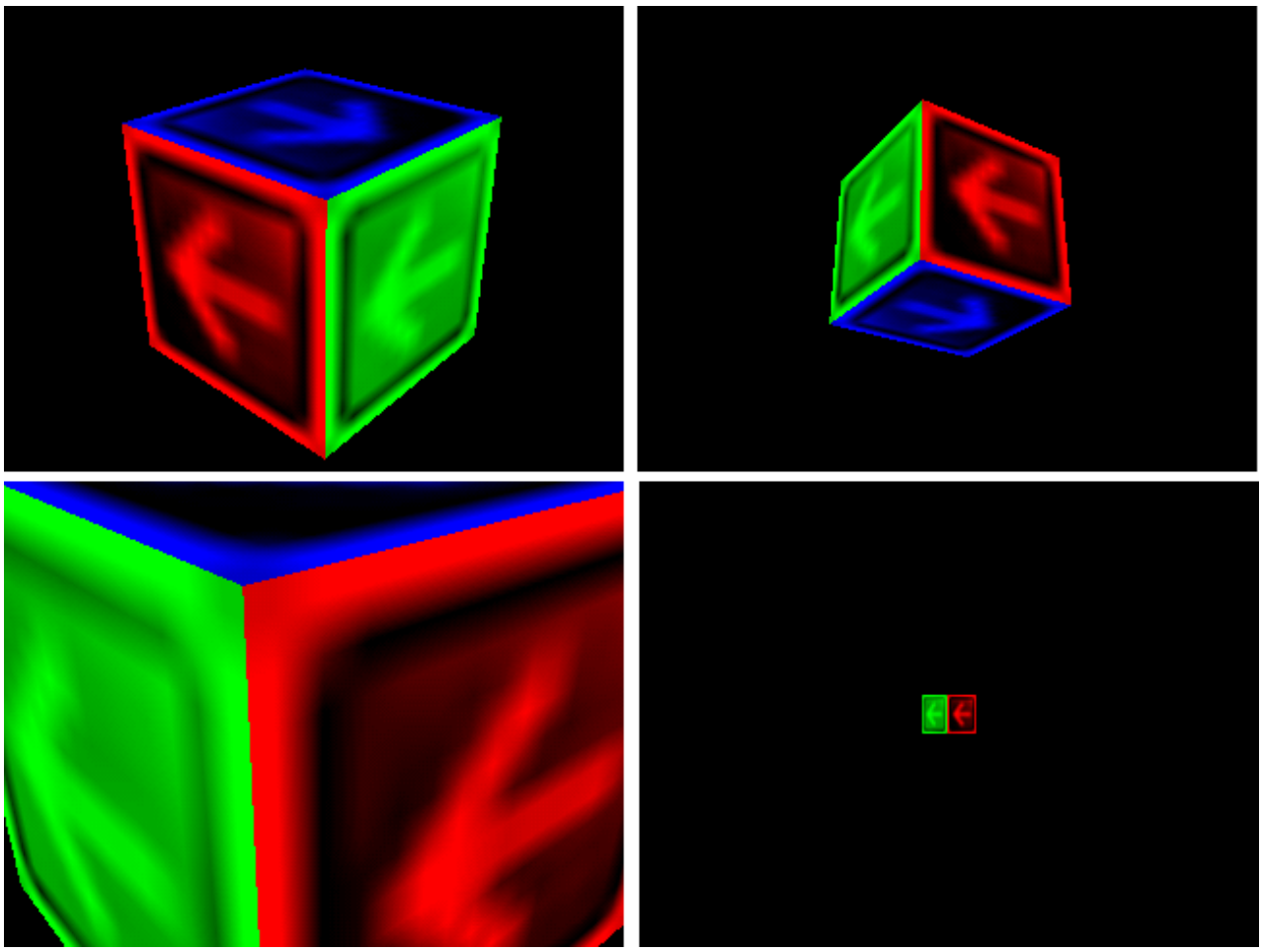

Figure 4.1 Tutorial 4 Output (Part 1)

The texture has been mapped successfully. However, you can see that every face is tinted (using the colour we used to draw that face). Whether, this effect is desirable it can be changed. In the next part of this tutorial you will learn another important part of OpenGL|ES: Lighting.

Lighting

Lighting, as the name refers to, is used to light up objects in your scene. There a few various different types of lighting. The first is ambient lighting. It does not come from any direction, and when it hits a surface, it is reflected in all directions. The next type is Diffuse Lighting. This is exactly the same as ambient lighting, except that it comes from a particular direction. The next type is Specular Lighting. This not only comes from a particular direction, but it is also reflected back in a particular direction. The last type is emissive lighting. It comes from a particular object. It gives off light, but does not reflect any. This is different to the previous three.

There are two important parts to lighting: Materials and Normals. Materials just means that you can set different properties, and how different surfaces will react to the light. Normals are an invisible line perpendicular to any polygon you draw, that you want affected by the light.

Introduction

This tutorial uses the code from the first part of this tutorial. We will be making little modifications this time.

Declarations

```
1:      #include <windows.h>
2:      #include <GLES\gl.h>
3:      #include <GLES\egl.h>
4:      #include <KGSDK\framework.h>
5:      #include <KGSDK\frameworkGL.h>
6:
7:      static float rot[3];
8:      static float loc[3];
9:
10:     float xrot = 0.0f;
11:     float yrot = 0.0f;
12:
13:     GLuint texture[1];
14:
15:     float lightAmbient[] = { 0.2f, 0.3f, 0.6f, 1.0f }; //Ambient Colour
Array
16:     float lightDiffuse[] = { 0.2f, 0.3f, 0.6f, 1.0f }; //Diffuse Colour
Array
17:
18:     float matAmbient[] = { 0.6f, 0.6f, 0.6f, 1.0f };   //Material
Properties for Ambient
19:     float matDiffuse[] = { 0.6f, 0.6f, 0.6f, 1.0f };   //Material
Properties for Diffuse
20 :    bool ColourMaterial = false;     //Colour Material Enabled?
```

Analysis: Four new variables have been added. lightAmbient and lightDiffuse are colour arrays for ambient lighting and diffuse lighting respectively. matAmbient and matDiffuse are the material properties for the lighting Ambient and Diffuse, respectively. The colour arrays are self-explanatory – they are the colour source of the light. The material properties values are a little trickier. What they do is multiply the light source by the values held in the array, to get the resultant colour which comes off the surface. For example, the above material properties will cause the light to lose 40% of the light (Hence the 0.6f). Line 20 declares a variable, of type Boolean, which will keep record of whether or not Colour Material is enabled. You will learn about Colour material later on in this tutorial.

Init

```
1:   void Init()
2:   {
3:        glEnable(GL_LIGHTING);   //Enable Lighting
```

```
4:          glEnable(GL_LIGHT0);      //Enable GL_LIGHT0
5:
6:          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, matAmbient);     //Set
Material Properties
7:          glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, matDiffuse);     //Set
Material Properties
8:
9:          glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient);//Set Light
Properties
10:         glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);//Set Light
Properties
11:
12:         if (!LoadTextures())                    //Load Textures, if fails
13:         {
14:             MessageBox(NULL, L"Error loading textures", L"Error", MB_OK);
15:         }
16:         else
17:         {
18:             glEnable(GL_TEXTURE_2D);
19:         }
20:
21:         glLoadIdentity();
22:
23:         GLfixed mat[4][4];
24:
25:         float nearz, farz;
26:
27:         farz  = 300.0f;
28:         nearz = 0.01f;
29:
30:         memset(mat, 0, sizeof(mat));
31:         mat[0][0] = (int) (65536.0f * 2.4f);
32:         mat[1][1] = (int) (65536.0f * 3.2f);
33:         mat[2][2] = (int) (65536.0f * (farz / (farz - nearz)));
34:         mat[2][3] = (int) (65536.0f * 1.0f);
35:         mat[3][2] = (int) (65536.0f * ((-farz * nearz) / (farz - nearz)));
36:
37:         glMatrixMode(GL_PROJECTION);
38:         glLoadMatrixx(&mat[0][0]);
39:
40:         rot[0] = 0;
41:         rot[1] = 0;
42:         rot[2] = 0;
43:         loc[0] = 0;
44:         loc[1] = 0;
45:         loc[2] = 3;
46:   }
```

Analysis: Lines 3 – 10 have been added to our Init Function. We enable Lighting on Line 3. In OpenGL|ES you can have a maximum of 8 Lights. Line 5 enables GL_LIGHT0, Lights in GL are labelled from GL_LIGHT0 through to GL_LIGHT 7. Next we set the Material properties on line 6 for ambient lighting and on line 7 for diffuse lighting. glMaterialfv takes three parameters: the first is which faces are being updated (In OpenGL|ES must be GL_FRONT_AND_BACK. The second parameter is what lighting we're updating, i.e. GL_AMBIENT and GL_DIFFUSE. The last parameter is pointer to value(s) that the lighting will be set to. glLightfv works in exactly the same way as glMaterialfv, but instead of specifying what material properties we're changing as the first parameter, you specify the Light (GL_LIGHT0) for which properties you are changing.

Render

```
1:      void DrawCube()
2:      {
3:         GLfloat cube[] =
4:         {
5:          // FRONT
6:         -0.5f, -0.5f,  0.5f,
7:          0.5f, -0.5f,  0.5f,
8:         -0.5f,  0.5f,  0.5f,
9:          0.5f,  0.5f,  0.5f,
10:         // BACK
11:        -0.5f, -0.5f, -0.5f,
12:        -0.5f,  0.5f, -0.5f,
13:         0.5f, -0.5f, -0.5f,
14:         0.5f,  0.5f, -0.5f,
15:        // LEFT
16:        -0.5f, -0.5f,  0.5f,
17:        -0.5f,  0.5f,  0.5f,
18:        -0.5f, -0.5f, -0.5f,
19:        -0.5f,  0.5f, -0.5f,
20:         // RIGHT
21:         0.5f, -0.5f, -0.5f,
22:         0.5f,  0.5f, -0.5f,
23:         0.5f, -0.5f,  0.5f,
24:         0.5f,  0.5f,  0.5f,
25:         // TOP
26:        -0.5f,  0.5f,  0.5f,
27:         0.5f,  0.5f,  0.5f,
28:        -0.5f,  0.5f, -0.5f,
29:         0.5f,  0.5f, -0.5f,
30:          // BOTTOM
31:        -0.5f, -0.5f,  0.5f,
32:        -0.5f, -0.5f, -0.5f,
33:         0.5f, -0.5f,  0.5f,
34:         0.5f, -0.5f, -0.5f,
35:          };
36:
37:          GLfloat texCoords[] =
38:          {
39:          // FRONT
40:          0.0f, 0.0f,
41:          1.0f, 0.0f,
42:          0.0f, 1.0f,
43:          1.0f, 1.0f,
44:          // BACK
45:          1.0f, 0.0f,
46:          1.0f, 1.0f,
47:          0.0f, 0.0f,
48:          0.0f, 1.0f,
49:          // LEFT
50:          1.0f, 0.0f,
51:          1.0f, 1.0f,
52:          0.0f, 0.0f,
53:          0.0f, 1.0f,
54:          // RIGHT
55:          1.0f, 0.0f,
56:          1.0f, 1.0f,
57:          0.0f, 0.0f,
58:          0.0f, 1.0f,
59:          // TOP
60:          0.0f, 0.0f,
61:          1.0f, 0.0f,
```

```
62:         0.0f, 1.0f,
63:         1.0f, 1.0f,
64:         // BOTTOM
65:         1.0f, 0.0f,
66:         1.0f, 1.0f,
67:         0.0f, 0.0f,
68:         0.0f, 1.0f
69:         };
70:
71:         glRotatef(xrot, 1.0f, .0f, 0.0f);
72:         glRotatef(yrot, 0.0f, 1.0f, 0.0f);
73:
74:         glVertexPointer(3, GL_FLOAT, 0, cube);
75:         glTexCoordPointer(2, GL_FLOAT, 0, texCoords);
76:         glEnableClientState(GL_VERTEX_ARRAY);
77:         glEnableClientState(GL_TEXTURE_COORD_ARRAY);
78:
79:         // FRONT AND BACK
80:         glColor4f(1.0f, 0.0f, 0.0f, 1.0f);    //Color: RED
81:         glNormal3f(0.0f, 0.0f, 1.0f);
82:         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
83:         glNormal3f(0.0f, 0.0f,-1.0f);
84:         glDrawArrays(GL_TRIANGLE_STRIP, 4, 4);
85:
86:         // LEFT AND RIGHT
87:         glColor4f(0.0f, 1.0f, 0.0f, 1.0f);   //Color: GREEN
88:         glNormal3f(-1.0f, 0.0f, 0.0f);
89:         glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);
90:         glNormal3f(1.0f, 0.0f, 1.0f);
91:         glDrawArrays(GL_TRIANGLE_STRIP, 12, 4);
92:
93:         // TOP AND BOTTOM
94:         glColor4f(0.0f, 0.0f, 1.0f, 1.0f);   //Color: BLUE
95:         glNormal3f(0.0f, 1.0f, 0.0f);
96:         glDrawArrays(GL_TRIANGLE_STRIP, 16, 4);
97:         glNormal3f(0.0f,-1.0f, 1.0f);
98:         glDrawArrays(GL_TRIANGLE_STRIP, 20, 4);
99:
100:        glColor4f(1.0f, 1.0f, 1.0f, 0.0f);    //Color: WHITE
101:    }
```

Analysis: To our render function we have only added Normals. As discussed before normals are an invisible line perpendicular to the surface. You must specify a normal for every surface (each side of the cube). If you know about light in general or have studying it as a topic you will be familiar with normals:

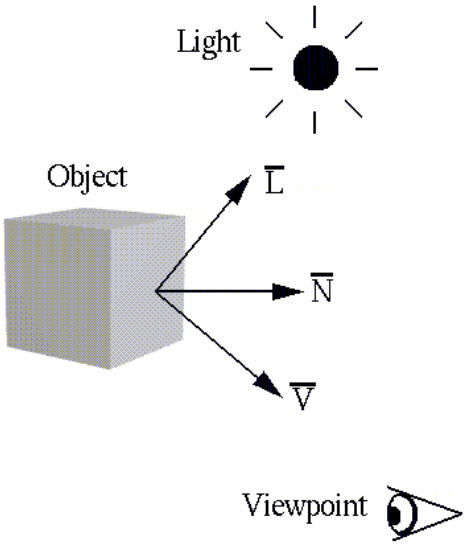

Figure 4.2 Normal

Figure 4.2 shows how a normal works. They work in exactly the same way as they do in real life. The normal is perpendicular (at 90˚) to the surface, as mentioned above. As the light hits the surface, the angle between the light and the normal is equal to the angle between the reflected light and the normal. If you don't understand this then don't worry. It isn't absolutely necessary to know how normals work.

You can add a Normal by using the glNormal3f GL command. Its three parameters are co-ordinates for the x, y and z axis's respectively. For the front surface the following normal co-ordinates would be used: (0, 0, 1). This would be directly perpendicular to the front surface. The back surface would have the co-ordinates (0, 0, -1), and so on. The section of lines 79 – 98 contains a glNormal3f function call preceding every glDrawArrays function call. Note that the Pen Colour is set to White at the end of this function (line 100). This is important, as when we come to Disable Colour Material the Cube will be rendered BLUE, as this was the last colour used. You can see this effect for yourself, by commenting this line out.

Update

```
1:  void Update()
2:  {
3:     GLfixed mat[4][4];
4:
5:     memset(mat, 0, sizeof(mat));
6:     mat[0][0] = (int) (65536.0f *  cos(rot[0]));
7:     mat[0][2] = (int) (65536.0f *  sin(rot[0]));
8:     mat[1][1] = (int) (65536.0f *  cos(rot[1]));
9:     mat[1][2] = (int) (65536.0f *  sin(rot[1]));
10:    mat[2][0] = (int) (65536.0f * -sin(rot[0]) * cos(rot[1]));
11:    mat[2][2] = (int) (65536.0f *  cos(rot[0]) * cos(rot[1]));
12:    mat[3][2] = (int) (65536.0f * loc[2]);
13:    mat[3][3] = 65536;
14:
15:    glMatrixMode(GL_MODELVIEW);
16:    glLoadMatrixx(&mat[0][0]);
```

```
17:
18:    if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_LEFT_SHOULDER ))
19:       loc[2] += 0.3f;
20:    // If the Left Shoulder Button has been pressed: Zoom out
21:    if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_RIGHT_SHOULDER))
22:       loc[2] -= 0.3f;
23:    // If the Right Shoulder Button has been pressed: Zoom In
24:    if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_DPAD_DOWN     ))
25:       xrot -= 1.0f;
26:    // If the Down button (DPAD) has been pressed: Rotate the cube downards
27:    if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_DPAD_UP         ))
28:       xrot += 1.0f;
29:    // If the Up button (DPAD) has been pressed: Rotate the cube upwards
30:    if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_DPAD_LEFT     ))
31:       yrot -= 1.0f;
32:    // If the Left button (DPAD) has been pressed: Rotate the cube towards
the left
33:    if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_DPAD_RIGHT  ))
34:       yrot += 1.0f;
35:    // If the Right button (DPAD) has been pressed: Rotate the cube towards
the Right
36:    if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_FORWARD     ))
37:    {
38:       ColourMaterial =! ColourMaterial;
39:       (ColourMaterial ? glEnable(GL_COLOR_MATERIAL) :
glDisable(GL_COLOR_MATERIAL));
40:    }
41:    // the Forward button has been pressed: Toggle Colour Material
42: }
```

Analysis: Our Update function has changed so that we now look out for the FORWARD button as well. If the forward button has been pressed, we toggle Colour Material Capability.

Output

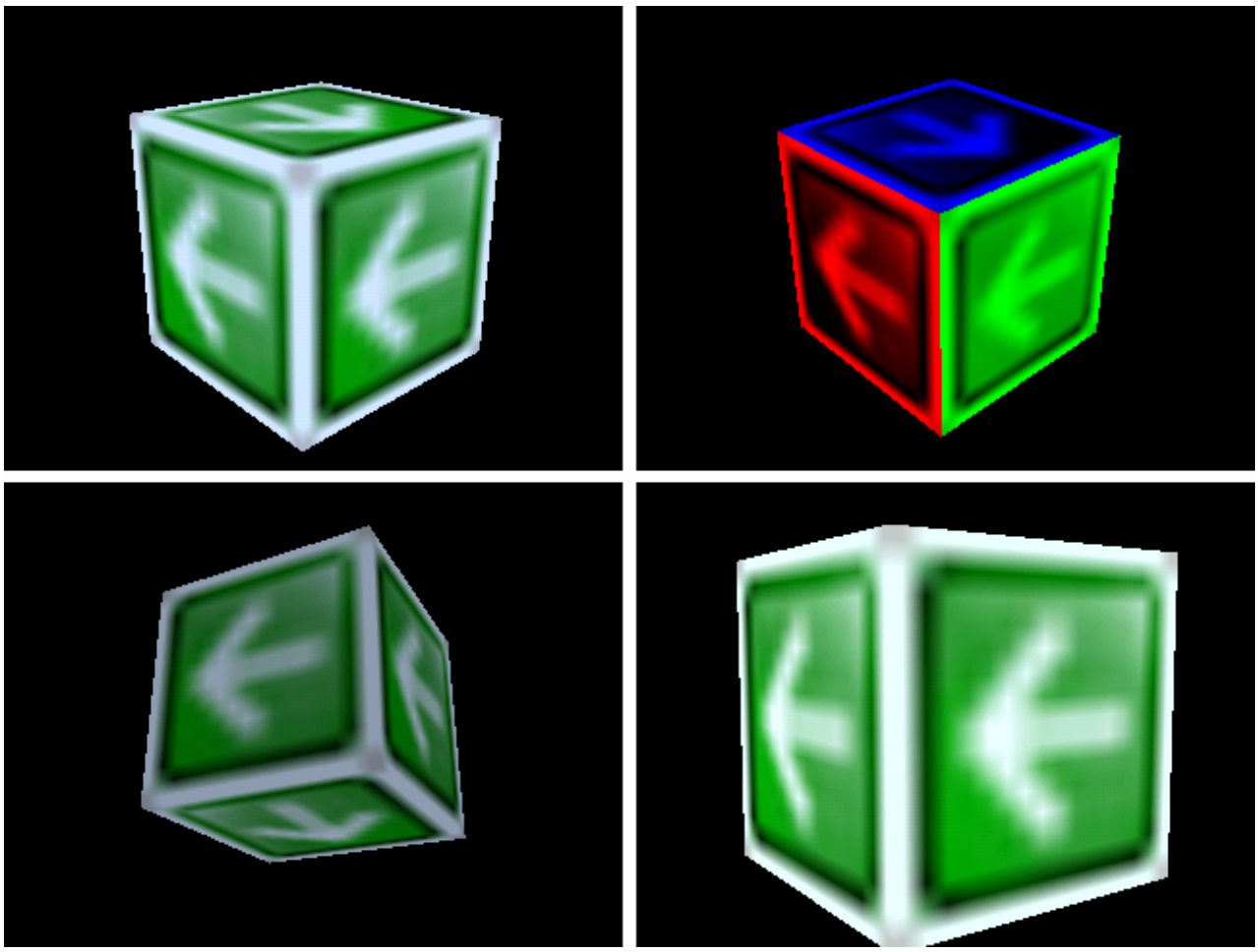

Figure 4.2 Tutorial 4 (Part 2) Output

- End of OpenGL|ES Tutorials: Textures & Lighting -