
OpenGL ES Examples

Version 2.3

December 16, 2004

S60
platform

Legal Notice

Copyright © 2005 Nokia Corporation. All rights reserved.

S60, Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1.	Document Control	4
1.1	References	4
1.2	Abbreviations	4
2.	Introduction	5
3.	Compiling the Examples.....	7
3.1	Compiling Each Example Separately.....	7
4.	Structure of the Examples	8
4.1	NewL	8
4.2	AppInit	8
4.3	AppExit.....	8
4.4	AppCycle.....	9
5.	Examples.....	10
5.1	SimpleCube.....	10
5.2	SimpleCubePixmap.....	12
5.3	SimpleCubePbuffer	12
5.4	SimpleLight	13
5.5	Texture	15
5.6	Multi Texturing.....	17
5.7	Hierarchy	20
5.8	Projective Shadow	21
5.9	Picking and Selection.....	23
5.10	Environmental Mapping	24
5.11	Texture Manager	25
5.12	Camera.....	27
5.13	LensFlare	28
5.14	Billboard	29
5.15	Snow and Rain.....	30

1. Document Control

1.1 References

[1]	<i>S60 OpenGL ES 3D SW Programmer Guide Document</i>	
[2]	<i>GIMP – the Gnu Image Manipulation interface</i>	http://www.gimp.org/

1.2 Abbreviations

EGL	OpenGL ES native platform graphics interface
OpenGL ES	OpenGL for Embedded Systems, a low-level 3D graphics API

2. Introduction

This document describes the example applications included in OpenGL ES. The examples illustrate the basic features of OpenGL ES and explain the usage of the required OpenGL ES commands. The following examples are included:

- SimpleCube:
 - o No lighting or material.
 - o Colorarray, vertexarray.
 - o Demonstration of different drawing and shading modes.
- SimpleCubePixmap:
 - o Based on SimpleCube.
 - o Demonstration of Pixmap surface.
- SimpleCubePbuffer:
 - o Based on SimpleCube.
 - o Demonstration of Pbuffer surface.
- SimpleLight:
 - o Initialization of materials and lighting.
 - o Two different objects with different materials.
 - o Two different lights; a lamp and a spot.
- Texture:
 - o Texture reading from *jpeg* files using Image Converter Library.
 - o Definition of textures, texture coordinates, and texture matrix.
 - o Texture wrapping.
 - o Combine with lighting.
 - o Blending.
- MultiTex:
 - o Demonstration of OpenGL ES multitexturing.
 - o Use of multiple surfaces and rendering contexts
- Hierarchy:
 - o “Robot-like” object hierarchy with multiple joints.
 - o Demonstration of hierarchical modeling using `glPushMatrix()` and `glPopMatrix()`.
- Projective Shadow:
 - o Demonstration of creating planar shadows using proper projection matrix.

- Picking and Selection:
 - o Demonstration of object and/or object polygon selection.
 - o Implemented using alpha color value.
- Environmental Mapping:
 - o Demonstration of one way of implementing environmental mapping.
 - o Implemented using average vertex normals as texture coordinate inputs.
- Texture Manager:
 - o Shows how to implement a texture manager, which could be used to handle all texture loading, deleting, color keying and blitting operations.
- Camera:
 - o Shows how to implement a viewing system for OpenGL ES applications.
- LensFlare:
 - o Shows how to create a lens flare using alpha blending.
- Billboarding:
 - o Demonstrates the billboarding technique, which is an acceleration method where complex geometry is replaced with a 2D sprite.
- Snow/Rain:
 - o Demonstration of particle engines.
 - o Particles are used for many effects such as snow, rain, smoke, sparks, and explosions.

3. Compiling the Examples

This chapter describes how to compile the examples. It is assumed that either an official Symbian SDK or a similar environment that has OpenGL ES libraries is installed. The directory structure of the examples is the following:

- OpenGLEx
 - o SimpleCube
 - *aif*
 - *data*
 - *group*
 - *inc*
 - *sis*
 - *src*
 - o SimpleCubePixmap
 - o SimpleCubePbuffer
 - o SimpleLight
 - o Texture
 - o MultiTex
 - o Hierarchy
 - o Shadows
 - o PickSelect
 - o EnvMapping
 - o Billboard
 - o LensFlare
 - o Rain
 - o Utils
 - o pictures

The following chapter describes how to compile each example.

3.1 Compiling Each Example Separately

The example's group directory contains a `bld.inf` file that can be used to build the example, for instance, as follows:

```
cd OpenGLEx\SimpleCube
bldmake.bat bldfiles
abld.bat build
```

4. Structure of the Examples

All the examples are written for Symbian OS and S60 devices. It is assumed that the reader has basic knowledge of the Symbian program structure, although it is not mandatory in order to comprehend the examples. The actual OpenGL ES part of the examples is separated as much as possible from the Symbian program structure. For example, in `SimpleCube` most of the OpenGL ES code are in files `SimpleCube.h` and `SimpleCube.cpp`. The OpenGL ES parts of these files are clearly marked and documented.

There are methods in the OpenGL ES part of each of the examples that are the same, and describe the different stages of using OpenGL ES. These are:

- `NewL`
 - o Default C++ constructor
 - o `ConstructL`
- `AppInit`
- `AppExit`
- `AppCycle`

4.1 `NewL`

Instantiation of the class that contains the OpenGL ES part of the examples uses the two-phase Symbian construction method. Default `C++ constructor` and `ConstructL` methods are defined as protected members in “<example name>.h” files. Initialization of member variables can be done inside the `C++ constructor`, while all the memory allocation, which has the potential of leaving, should be left inside the `ConstructL` method. `NewL` method is called from the application’s container class.

4.2 `AppInit`

`AppInit` contains OpenGL ES initialization code, for example, the enabling of various tests such as depth test and backface culling, initialization of viewport and projection, and enabling the vertex and color arrays. This method should not contain any code that can leave. As was said earlier, a proper place for code that may leave is in the `ConstructL` method. `AppInit` is called from the application’s container class right after the call to `NewL`.

4.3 `AppExit`

If `AppInit` contains any resource allocations, `AppExit` releases them. This is the case, for example, in the Texture example (see Section 5.5), where `glGenTextures` function is used to generate texture objects. The application container’s destructor calls `AppExit`.

4.4 AppCycle

This is the main rendering method of the examples. All the object drawing and animation is done here. `AppCycle` is called once for every frame by `DrawCallback` method in "*<example name>Container.cpp*" file.

5. Examples

5.1 SimpleCube

This example demonstrates the use of vertex arrays, color arrays and shading modes in OpenGL ES by rotating a colored cube on the screen. The user can change the setup by accessing the menu by pressing the left software button. Figure 5.1 shows screenshots of the example running on the emulator.

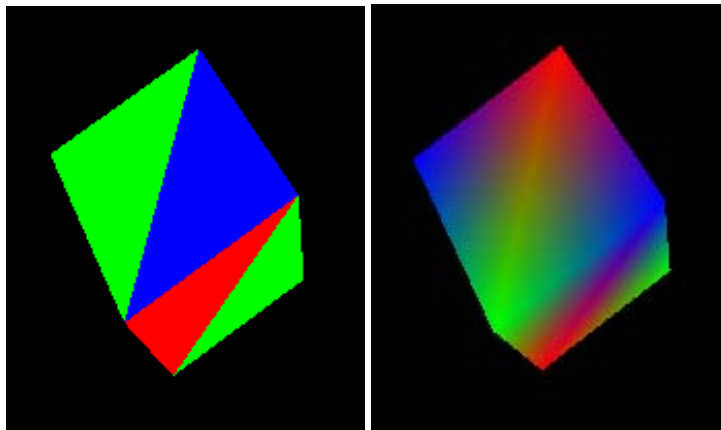


Figure 5.1 Screenshots of SimpleCube running on the emulator

The vertices of the cube are defined as follows:

```
static const GLbyte vertices[8 * 3] =
{
    -1, 1, 1,
    ...
};
```

The indices, that is, the lookup values for drawing the polygons are defined as follows:

```
static const GLubyte triangles[12 * 3] =
{
    // front
    1, 0, 3,
    1, 3, 2,
    ...
};
```

If a predefined vertex color set is wanted, as in this example, a separate color for each vertex has to be defined, for example, as follows:

```
static const GLubyte colors[8 * 4] =
{
    0, 255, 0, 255,    // (Red, Green, Blue, Alpha)
    ...
};
```

After an instance of an example class is created, the `AppInit()` method is called, which finalizes the initialization of OpenGL ES. Firstly, the background color is set to black and back face culling is enabled:

```
glClearColor( 0.f, 0.f, 0.f, 1.f ); // black
glEnable( GL_CULL_FACE );
```

Secondly, the viewport and projection matrix are initialized:

```
glViewport( 0, 0, iScreenWidth, iScreenHeight );
glMatrixMode( GL_PROJECTION );
glFrustumf( -1.f, 1.f, -1.f, 1.f, 3.f, 1000.f ); // ( left, right,
bottom, top, near, far )
glMatrixMode( GL_MODELVIEW );
```

After that, the vertex and color arrays and pointers are enabled and set:

```
glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer( 3, GL_BYTE, 0, vertices );
glEnableClientState( GL_COLOR_ARRAY );
glColorPointer( 4, GL_UNSIGNED_BYTE, 0, colors );
```

All the object geometry is given as vertices. If floating point coordinates are used, the computation slows down considerably (so do not use them!). For more performance increase tips, see reference document [1].

Colors are defined separately for each vertex. The color of a polygon, in this case with flat-shaded polygons, is the color of the last vertex used to form that particular polygon.



Note: No material or lighting is defined in this example. Another possibility is to use a single color per triangle strip, by using `glColor4[f/x]` and disabling `GL_COLOR_ARRAY`.

Shading model is the last thing set in the `AppInit()` method:

```
glShadeModel( GL_FLAT );
```

For every frame the `AppCycle()` method is called. The current frame number is given as an argument to the method. First the color and depth buffers are cleared:

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

Next, the cube object is animated and drawn:

```
glLoadIdentity(); // Change the modelview matrix to identity
glTranslatef( 0, 0, -cameraDistance << 16 ); // By default the camera
looks along negative z-axis
glRotatex( frame << 16, 1 << 16, 0, 0 ); // 1 << 16 == 1.0 as fixed
point
glRotatex( frame << 15, 0, 1 << 16, 0 );
glRotatex( frame << 14, 0, 0, 1 << 16 );
```

There are alternatives to both `glTranslatef(...)` and `glRotatex (...)` in OpenGL ES. These are `glTranslatef(...)` and `glRotatef(...)`, which take floating-point values instead of fixed-point values. Before drawing the object, it has to be scaled to fit the screen:

```
glScalef( aSizeX, aSizeY, aSizeZ );
glDrawElements( GL_TRIANGLES, 12 * 3, GL_UNSIGNED_BYTE, triangles );
```

The first argument to `glDrawElements(...)` specifies what kind of primitives to render. Alternatives to `GL_TRIANGLES` are `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, and `GL_POINTS`. The second argument specifies the number of elements to be rendered. The third argument specifies the type of the values in indices, and it must be either `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT`. The last argument specifies a pointer to the location where the indices are stored. An alternative to `glDrawElements(...)` is to use `glDrawArrays(...)`, which takes the following three arguments: the first is the same as in `glDrawElements(...)`, while the second specifies the starting index in the enabled arrays, and the third specifies the number of indices to be rendered.

5.2 SimpleCubePixmap

This example is based on SimpleCube example but instead of onscreen window rendering a Pixmap surface is used. To create the Pixmap surface, a config that supports Pixmap must be chosen. Otherwise the creation process is the same as with window surfaces.

```
const EGLint attrib_list[] = { EGL_BUFFER_SIZE, BufferSize,
EGL_SURFACE_TYPE, EGL_PIXMAP_BIT, EGL_NONE };
```

When creating the bitmap the size must be specified. This size is automatically used as the OpenGL ES framebuffer size when creating the Pixmap surface. Additionally, the bitmap's color depth has to match the EGL config and the config has to support Pixmap rendering.

```
iPixmap = new CwsBitmap( iEikonEnv->WsSession() );
iPixmap->Create( Window().Size(), iEikonEnv->ScreenDevice() -
>DisplayMode() );

/* Create a pixmap surface */
iEglSurface = eglCreatePixmapSurface( iEglDisplay, config, iPixmap,
NULL );
```

Because pixmaps are single-buffered offscreen surfaces they are not directly visible on the screen. Bitmaps do not have any location attribute so the location on the screen have to be specified. To display bitmap the next lines are needed in Draw() method:

```
/* Wait for the OpenGL ES rendering to complete */
eglWaitGL();

/* Display the pixmap on the screen */
CWindowGC& gc = SystemGc();
gc.BitBlt( TPoint( 0, 0 ), iBmap );
```

All rendering to a Pixmap is guaranteed to be complete after `eglWaitGL` call.

5.3 SimpleCubePbuffer

This example is based on SimpleCube example but instead of onscreen window rendering the Pbuffer surface is used. To create a Pbuffer surface, a config that supports Pbuffers must be chosen.

```
const EGLint attrib_list[] = { EGL_BUFFER_SIZE, BufferSize,
EGL_SURFACE_TYPE, EGL_PBUFFER_BIT, EGL_NONE };
```

There is also a need to create a bitmap where content of the Pbuffer surface is copied for displaying. The surface and the bitmap should be same size.

```
iPixmap = new CwsBitmap( iEikonEnv->WsSession() );
iPixmap->Create( Window().Size(), iEikonEnv->ScreenDevice() -
>DisplayMode() );

/* Create a pbuffer surface */
iEglSurface = eglCreatePbufferSurface( iEglDisplay, config, attribs
);
```

To display the content of the Pbuffer surface the content is first copied to bitmap in the Draw() method and then rendered to the screen similarly as in previous example.

```
eglCopyBuffers(iEglDisplay, iEglSurface, iPixmap);
```

5.4 SimpleLight

This example demonstrates the use of materials and lighting in OpenGL ES. The example consists of two different lights, a lamp and a spotlight. The example also consists of two different objects, a duck and a head. All the object geometries are defined in files `duck.h` and `head.h`. Both objects have different materials. The user can change the setup by accessing the menu by pressing the left software button. Figure 5.2 shows screenshots of the example running on the emulator.

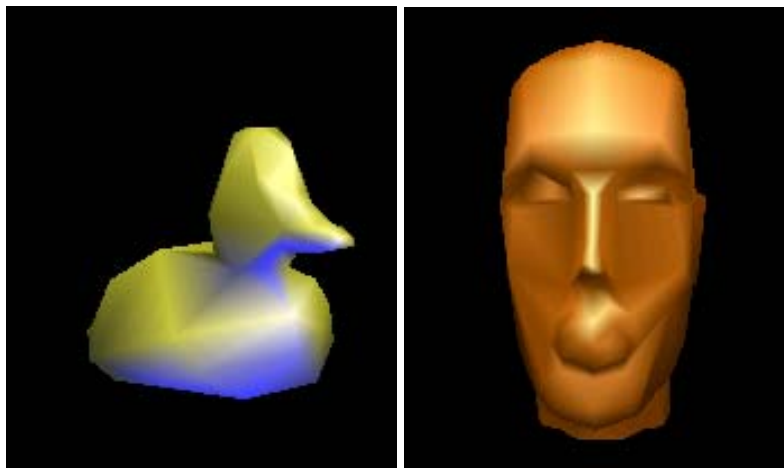


Figure 5.2: Screenshots of SimpleLight running on the emulator

Lighting is enabled in OpenGL ES with the following line:

```
glEnable( GL_LIGHTING );
```

In S60, OpenGL ES can have up to eight different light sources at a time, from `GL_LIGHT0` to `GL_LIGHT7`. These are each separately enabled as follows:

```
glEnable( GL_LIGHT0 );    // GL_LIGHT1, GL_LIGHT2, etc.
```

Each light can have the following properties defined: diffuse, ambient, specular, and position. In addition to these, a spotlight can have its direction, exponent, cut-off values, and three different attenuation values defined. In the example, the following lines set the parameters of the lamp:

```
glLightfv( GL_LIGHT0, GL_DIFFUSE,   lightDiffuseLamp );
glLightfv( GL_LIGHT0, GL_AMBIENT,   lightAmbientLamp );
glLightfv( GL_LIGHT0, GL_SPECULAR,   lightDiffuseLamp );
glLightfv( GL_LIGHT0, GL_POSITION,   lightPositionLamp );
```

In addition to these, the example's spotlight arguments are defined as follows:

```
glLightf( GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5 );
glLightf( GL_LIGHT1, GL_LINEAR_ATTENUATION,    0.5 );
glLightf( GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.5 );

glLightf( GL_LIGHT1, GL_SPOT_CUTOFF, 17.0 );
glLightfv( GL_LIGHT1, GL_SPOT_DIRECTION, lightDirectionSpot );
glLightf( GL_LIGHT1, GL_SPOT_EXPONENT, 2.0 );
```

The total attenuation factor is formed with the following formula:

$$\text{AttenuationFactor} = \frac{1}{k_c + k_l d + k_q d^2}$$

Here d is the distance between the light's position and the vertex, k_c is `GL_CONSTANT_ATTENUATION` value, k_l is the `GL_LINEAR_ATTENUATION` value, and k_q is the `GL_QUADRATIC_ATTENUATION` value. `GL_SPOT_EXPONENT` value controls how concentrated the light is, `GL_SPOT_DIRECTION` specifies the spot direction, and `GL_SPOT_CUTOFF` specifies the cone size of the spotlight, see Figure 5.3.

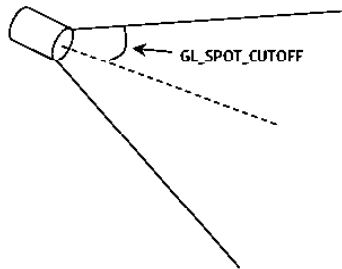


Figure 5.3: Definition of a spot cutoff value

A global scene ambient light is defined by:

```
glLightModelxiv( GL_LIGHT_MODEL_AMBIENT, globalAmbient );
```

Note: Default value of global ambient light is (0.2, 0.2, 0.2, 1.0).

In order for the lights to actually work, the object material needs to be defined. For that, the calculation of object normals is needed. Since the normals need to be unit-length for the calculations, and to make sure they are, we can enable automatic normalization for the normals:

```
glEnable( GL_NORMALIZE );
```

Enabling the autonormalize slows down the renderer handling a bit. This can be avoided by using fixed-point values and by normalizing the normals beforehand.

Enabling of the normal array is done as any other array:

```
glEnableClientState( GL_NORMAL_ARRAY );
```

Set the pointer to the normal array:

```
glNormalPointer( GL_BYTE, 0, objNormaldataDuck );
```

Setting of the object material is much like setting up the lighting. The object's ambient, diffuse, specular colors, the shininess, and the color of any emitted light can be set. Setting of the duck's material is done with the following lines:

```
glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, objDiffuseDuck );
glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, objAmbientDuck );
glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, objSpecularDuck );
glMaterialfv( GL_FRONT_AND_BACK, GL_EMISSION, objEmissionDuck );
glMaterialx ( GL_FRONT_AND_BACK, GL_SHININESS, 10 << 16 );
```

The first argument specifies which face(s) are being updated. The only accepted value is `GL_FRONT_AND_BACK`. The second argument specifies which material parameter(s) are being updated. Possible values are `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION`, and `GL_SHININESS`. The third argument is a pointer to the input data.

5.5 Texture

This example demonstrates texturing in OpenGL ES. Definition of textures, texture coordinates, and the use of texture matrix are shown. Textures are loaded from *jpeg* files using Image Converter Library. This example also demonstrates texture wrapping and combines lighting and blending with textures. Again, the user can change the setup by accessing the menu by pressing the left software button. Figure 5.4 shows a screenshot of the example running on the emulator.



Figure 5.4: Screenshot of Texture running on the emulator

Textures used in this example are made with Gimp (see reference document [2]), except for the `nasa_hubble.jpg` texture, which was downloaded from <http://opposite.stsci.edu/pubinfo/PR/2002/11/index.html>.

In this example, a single cube that has its every side textured is created. In order to do this, duplication of the cube's vertex coordinates is needed. Every vertex appears three times in the vertex list, with three different associated texture coordinates. See the definition of vertices from source code file `Texture.cpp`. Since there are two different textures, the cube has to be constructed from two different objects, each of which is assigned to a certain texture. Two different index lists are defined for this purpose: `triangles` and `triangles2`. `triangles` contains indices for five sides of the cube, where the `nasa_hubble` texture is assigned, and `triangles2` contains indices for the sixth side, where the `ogles.jpg` texture is assigned.

Texture coordinates are defined in the same way as the color values, that is, per vertex. Possible data types for texture coordinates are `GL_BYTE`, `GL_SHORT`, `GL_FIXED`, and `GL_FLOAT`. In this example, the defined coordinate type is `GL_BYTE`, and therefore the possible values are between `[-128, 127]`. Both of the used texture sizes are 256x256 pixels.

Texturing is enabled in OpenGL ES with the following line:

```
glEnable( GL_TEXTURE_2D );
```

Next, an appropriate texture matrix has to be initialized. In OpenGL ES, all the texture coordinates must be between values `[0, 1]`, unless texture wrapping is sought after. To accomplish this, the texture coordinate values are translated to be within the ranges of `[0, 255]`, and then scaled by 255. Remember that the input texture coordinate values were between `[-128, 127]`. The following lines accomplish this:

```
glMatrixMode( GL_TEXTURE );
glLoadIdentity();
glScalef( 1.f / 255.f, 1.f / 255.f, 1.f );
glTranslatef( 128.f, 128.f, 0.f );
glMatrixMode( GL_MODELVIEW );
```

Next, the texture coordinate array is enabled and the pointer is set:

```
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glTexCoordPointer( 2, GL_BYTE, 0, nokTexCoords );
```

Textures are initialized using the following command:

```
glGenTextures( 2, iNokTexObjects );
```

The first argument is the number of textures to be initialized and the second specifies an array where the texture names are stored.

	Note: This function also reserves memory for the textures, which needs to be released in <code>AppExit()</code> method: <pre>glDeleteTextures(2, iNokTexObjects);</pre>
---	---

Texture binding and setting the texture environment are accomplished with the following lines:

```
glBindTexture( GL_TEXTURE_2D, iNokTexObjects[0] );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TEX_WIDTH, TEX_HEIGHT,
              0, GL_RGB, GL_UNSIGNED_BYTE, texPtr );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );
```

The arguments for the `glBindTexture(...)` and `glTexImage2D(...)` are analogous to those in OpenGL.

	Note: After textures have been loaded with <code>glTexImage2D(...)</code>, the original bitmaps can be deleted because the textures have been copied to OpenGL ES texture manager, that is, the textures are stored inside OpenGL ES. See the code in <code>TextureAppUi.CPP</code> for more information. When using lighting with texture mapping, the texture environment has to be changed from <code>GL_REPLACE</code> to <code>GL_MODULATE</code> as in the following line: <pre>glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);</pre>
---	--

The shading model should be set to `GL_SMOOTH` when using lighting.

In this example, texture wrapping is implemented simply by changing the scale in texture matrix. Instead of scaling the texture coordinates by `1.f/255.f` we scale them by `1.f/128.f`. After that the texture coordinates are between `[0, 2]`, which causes the texture to wrap one time. See code in `Texture.cpp`.

Use of blending can create nice-looking effects. In this example, blending is used to create a translucency effect as can be seen in Figure 5.5. In the examples, pressing down the joystick button enables blending.



Figure 5.5: Translucency effect created by using blending function

The following lines set blending function and enable blending:

```
glEnable( GL_BLEND );  
glBlendFunc( GL_ONE, GL_ONE );
```

Depth testing and back face culling must be disabled when enabling blending. Even then, the rendering result depends on the rendering order.

5.6 Multi Texturing

This example demonstrates multiple dynamic lights without enabling OpenGL ES lighting. The effect has been created using multitexturing. Figure 5.6 shows a screenshots of the example running on the emulator.

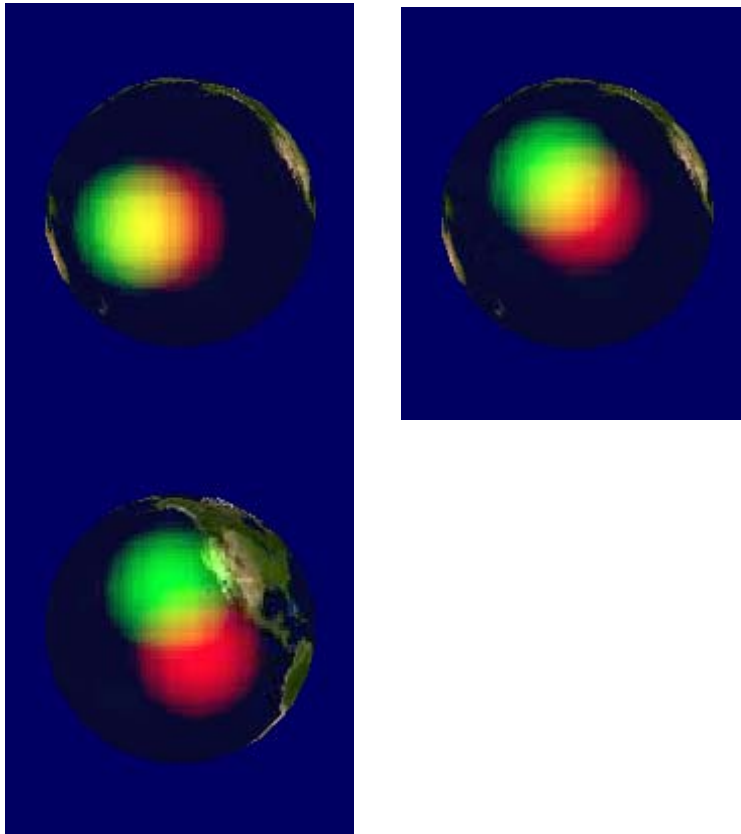


Figure 5.6: Screenshots of MultiTex running on the emulator

The example uses two different textures for sphere object. The world texture is loaded from *jpeg* file using Image Converter Library. The light texture, used as environmental map, is first rendered to pbuffer surface and then copied to created texture. The final textured object is rendered to window surface. The pointer to pbuffer context have to be passed to the window context during creation because surfaces have their own contexts but they need to share the same texture.

```
// Create a window context
// Pbuffer context needs to be given to eglCreateContext because
// Pbuffer and
// window context share the same texture.
iEglContext = eglCreateContext( iEglDisplay, aConfig,
iEglPbufferContext, NULL );
```

The window context needs two texture units, one for world texture and another for envmap texture, which are used at the same time. Both of these texture units need to be initialized differently. The texture unit to which subsequent texture state calls will affect is chosen below:

```
// TEXTURE0 is used for World texture
glActiveTexture( GL_TEXTURE0 );
glClientActiveTexture( GL_TEXTURE0 );

// Enable texturing
glEnable( GL_TEXTURE_2D );

// Enable vertex and texture coordinates arrays.
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_TEXTURE_COORD_ARRAY );

// Set array pointers.
glVertexPointer( 3, GL_SHORT, 0, objVertex );
glTexCoordPointer( 2, GL_SHORT, 0, objTexCoords );
```

```
// Bind the world.jpg texture and set the texture environment.
glBindTexture( GL_TEXTURE_2D, EObjectTex );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, aWidth, aHeight,
              0, GL_RGB, GL_UNSIGNED_BYTE, tex );
glTexEnvx( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );

// Texture to TEXTURE1 is taken from Pbuffer context
glActiveTexture( GL_TEXTURE1 );
glClientActiveTexture( GL_TEXTURE1 );

// Initialize second texture unit here...
```

After that the second texture unit is initialized correspondingly. Now both texture units are initialized and ready to use. TEXTURE0 is activated by default and when it is wanted to use the second texture unit it needs to be activated by using `glActiveTexture()` and `glClientActiveTexture()` functions.

The light texture is created by rendering two colored, moving circles to pbuffer (see Figure 5.7) and copying the pbuffer context to texture using `glCopyTexImage2D()`. In order to let the created two lights overlap while animating, we have to set the texture environment to `GL_ADD` when rendering the lights.

```
glTexEnvx( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_ADD );
```

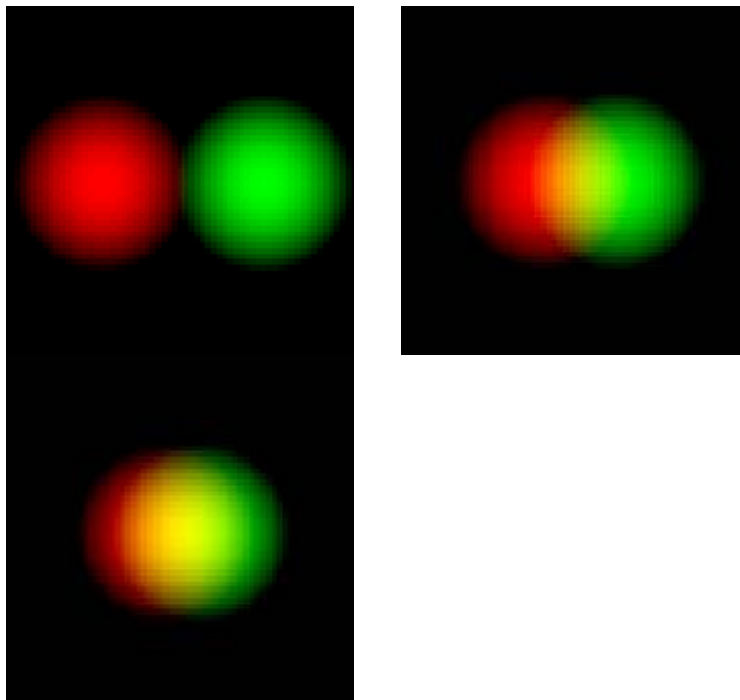


Figure 5.7: Screenshots of lights in Pbuffer running on the emulator.

In `AppInit()` function the `envmap` texture is bound both to pbuffer texture and the second texture unit of the window context. Now both textures in different texture units can be used at the same time to sphere object just activating also the second texture unit.

```
// Set the TEXTURE1 unit as active (envmap texture)
glActiveTexture( GL_TEXTURE1 );
glClientActiveTexture( GL_TEXTURE1 );
```

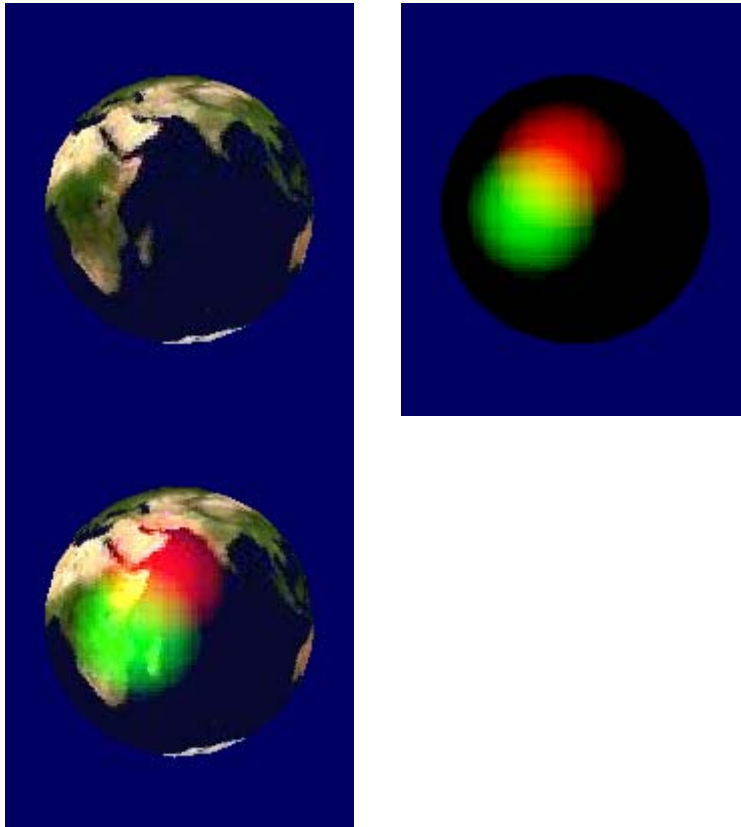


Figure 5.8: Screenshots to show the impact of texture environment mode running on the emulator.

In Figure 5.8, the leftmost screenshot shows the situation where only world texture used, the middle screenshot shows just the envmap texture used, and finally, the rightmost screenshot shows both textures combined with `GL_REPLACE` mode for the first texture (world) and `GL_ADD` for the envmap texture.

5.7 Hierarchy

This example demonstrates hierarchical programming using OpenGL ES `glPushMatrix(...)` and `glPopMatrix(...)`. The user can rotate either the elbow or the fingers of a robotic arm by turning the joystick left or right. Pressing the joystick button down changes the point of rotation. Figure 5.9 shows a screenshot of the example running on the emulator.

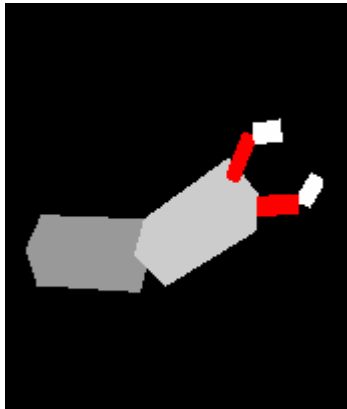


Figure 5.9: Screenshot of Hierarchy running on the emulator

The model and the hierarchy used in this example are shown in Figure 5.10.

There is also a helper function, which draws a 3D coordinate system. Try to call it from different points in the hierarchy to visualize the current location of the origin. All the different elements are drawn using the same origin-centered cube, the side length of which is two. Every time we draw a cube, the origin is in the middle of that cube, and in order to rotate the cube, for example, around a particular side, we first have to move the origin to that side of the cube.

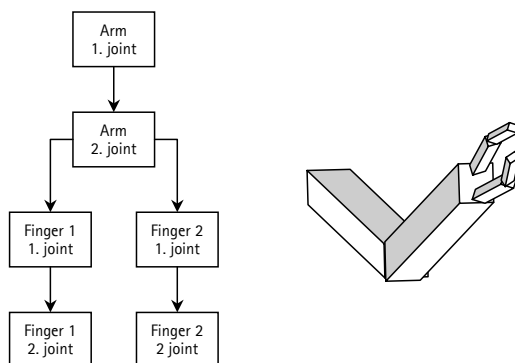


Figure 5.10: Hierarchical model used in this example

5.8 Projective Shadow

This example demonstrates the creation of projective planar shadows. The example consists of a single shadow object, a single light source, and a planar ground plane, where the shadow is projected. The user can change the position of the light source by turning the joystick left or right. Figure 5.11 shows a screenshot of the example running on the emulator.

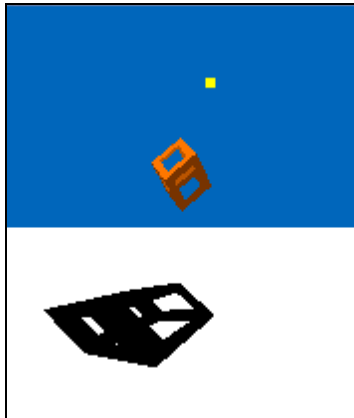


Figure 5.11: Screenshot of Projective Shadows running on the emulator

The order of rendering the scene is important in order to make the shadows appear correctly. The following order is used in this example:

1. Draw the ground objects, that is, the ground plane.
2. For each light source and for each shadow object:
 - a. Set the correct shadow matrix.
 - b. Set the current color to black.
 - c. Draw all the objects that cast shadows with black color.
3. Draw all the shadow objects without the shadow matrix and with the real object colors/textures.

Another important thing is the correct position of the shadow matrix. It has to be placed in the current modelview matrix after the camera transformations and before the model transformations. This is illustrated in Figure 5.12.

The order of matrix operations/pixel

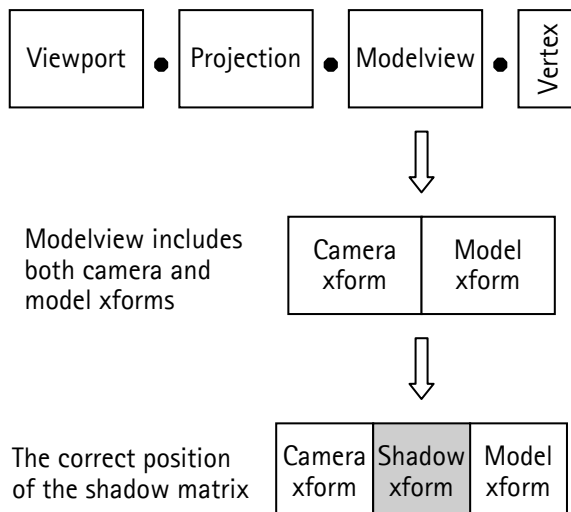


Figure 5.12: Correct positioning of the shadow matrix

The reader is referred to the source code in *Shadows.cpp* for more detailed comments.

5.9 Picking and Selection

This example demonstrates one possible way of implementing object and polygon selection. The example consists of two parts: the first is a single rotating cube, the polygons of which can be chosen. The second part consists of two rotating cubes that can be separately chosen. The user can choose the mode from the menu. Figure 5.13 shows a screenshot of the example running on the emulator.

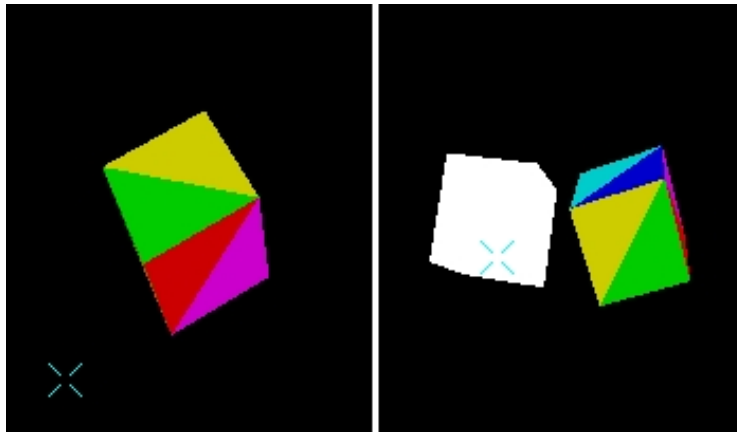


Figure 5.13: Screenshots of Picking and Selection running on the emulator

In this example, the user can freely move a selection cursor or a cross on the screen and select an object or a surface by pressing the joystick button down. See the source code in `PickSelect.cpp` for more information about the choosing of the alpha values for the polygon/object. Remember that the color of the polygon (and the alpha value) is determined by the color of the last pixel of the triangle triplet.

The picking and selection are implemented using the object's alpha color values. Although this restricts the usage of, for example, blending, the implementation is quite simple and it gives a hint of how to implement more sophisticated selection methods. When the user presses the joystick button, the cursor's current location pixel's alpha value is read and then checked to determine whether the cursor was on any polygon/object. If this is the case, the polygon/object is drawn in white color, otherwise nothing happens.

The reading of the selected pixel's alpha value is accomplished with the following command:

```
glReadPixels( iCursorPosX, iCursorPosY, 1, 1, GL_RGBA,
GL_UNSIGNED_BYTE, &aPixel );
```

The first two arguments define the lower left corner of the rectangle where the pixels are read. The next two arguments define the width and height of that window. The fifth and sixth arguments are the format and type of the pixel data, and the last argument is an array pointer where the pixels are stored.

Depth buffering is essential in this implementation because otherwise the alpha value that is read might be wrong. And this is also the reason why blending cannot be used with this implementation.

Another way of implementing selection, and a more general way, too, is to render a 1x1 scissor, that is, 1 pixel in the middle of the cursor to get an alpha value. Be sure to turn any blending and so on off when rendering the scissor. The rendered pixel's alpha value can now be read, and based on that value, it can be decided which polygon/object was rendered to the point where the cursor was. After that, the whole scene can be rendered using, for example, blending.

The selection cursor is implemented by defining an appropriate viewport and setting the correct projection (`glOrthof(...)`) and modelview matrices. The viewport should be set to square to avoid the distortion of the cursor geometry. This requires additional checking to ensure that the cursor stays inside the screen all the time. The reader is referred to the `DrawCursor(...)` and `UpdateCursorPosition(...)` functions in `PickSelect.cpp` for more detailed information and additional comments.

5.10 Environmental Mapping

This example demonstrates one possible way of implementing environmental mapping. The example consists of the same two models as used in SimpleLighting example (see Section 5.4), which are environmental mapped. The user can rotate both objects with the joystick. Figure 5.14 shows screenshots of the example running on the emulator and the texture used in environmental mapping.



Figure 5.14: Screenshots of EnvMapping running on the emulator and the texture used in environmental mapping

Environmental mapping in this example is implemented by using averaged vertex normals $\vec{n}$ as texture input coordinates. An average vertex normal is calculated by averaging every surface normal that is connected to that particular vertex. Figure 5.15 illustrates the situation.

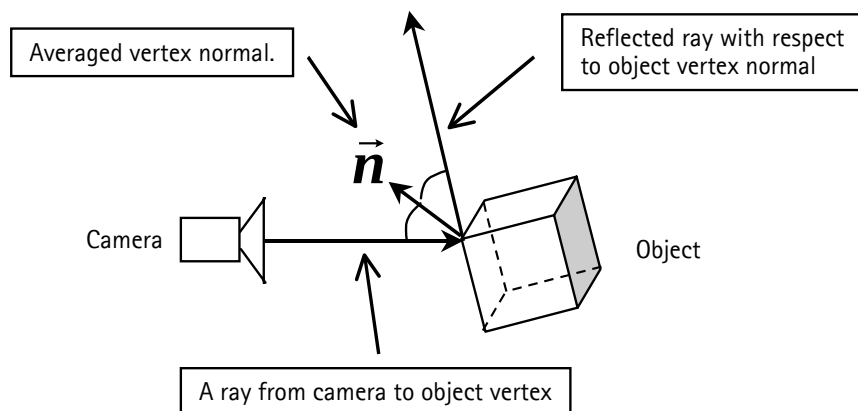


Figure 5.15: An illustration of the definition of averaged vertex normal

The texture coordinates are actually normals, 4D coordinates, where the last value is 1. These coordinate values are given to texture matrix, which rotates them in the same way as the model view matrix does. Output from the texture matrix is also a 4D vector, where only the first two components are used as (u, v) texture coordinates. Since the first three coordinates form a unit vector, also the norm of (u, v) is at most 1, leading to the fact that only a circular area of the texture is used as an environmental map. This is depicted in Figure 5.16. Normally the input texture coordinates would be the coordinates of the reflected ray with respect to the object vertex normal. This means that the method implemented in this example gives only an approximation of the reflectance. An advantage is that the averaged normals can be calculated apriori. See the `CalculateTextureCoordinates(...)` method in file `EnvMapping.cpp` for additional comments.

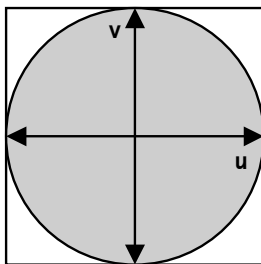


Figure 5.16: The area used of the texture in environmental mapping (grayed area)

The restrictions of implementing the environmental mapping in this way are that we can use only one environmental texture at a time, and only the rotations of the camera and the object are allowed, that is, neither the camera nor object may move around. Both the rotations of the camera and the object are also multiplied within the current texture matrix. See methods `DrawDuck(...)` and `DrawHead(...)` for additional information and comments on setting the texture matrix.

5.11 Texture Manager

This example shows how to implement a texture manager, which could be used to handle all texture loading, deleting, color keying, and blitting operations.

Bitmap loading/converting operations with the Symbian Image Converter Library (class `CImageDecoder`) are asynchronous. Thus, the task of loading textures from bitmaps becomes complex, especially when it comes to loading several textures, since it requires some additional code to handle the communication between the application and the server. The idea behind this texture manager is to encapsulate that communication code within a new class (`CTextureManager`).

The source code for class `CTextureManager` can be found in `Utils\utils.cpp`.

Any image format supported by the Symbian Image Converter Library can be loaded using this manager (the only restriction is that their width and height must be power of 2).

Here is the procedure for using the texture manager:

1. Implement the `MTextureLoadingListener` interface (`Utils\utils.h`).
2. Pass a reference to that interface to `CTextureManager::NewL(...)`.
3. Call `CTextureManager::RequestToLoad(...)` for each texture you want to load.

4. Call `CTextureManager::DoLoadL(...)` to start loading the textures.

After the last step, the texture manager starts loading and converting the textures, and uses methods `MTextureLoadingListener::OnStartLoadingTexturesL(...)` and `MTextureLoadingListener::OnEndLoadingTexturesL(...)` to notify the `MTextureLoadingListener` client when the operations start and stop.

Refer to the Billboard (Section 5.14), Snow and Rain (Section 5.15) and LensFlare (Section 5.13) examples for additional information.

In addition to loading textures, the texture manager could also be used for color keying, and texture blitting operations.

Color keying is a technique used to add transparent areas in textures. It is achieved by specifying a range of colors that should not be drawn. Like OpenGL, OpenGL ES does not support color keys directly, but this effect can be achieved by adding an alpha channel and enabling alpha testing.

Color keys are specified in the form `[MinColor, MaxColor]` where `MinColor` and `MaxColor` are both RGB color values. Any pixel with a color within the range `[MinColor, MaxColor]` does not pass the alpha test, and, consequently, is rendered.

For example, by using the color key range:

```
MinColor[] = { 50, 0, 50 }; //Red=50, Green=0, Blue=50
MaxColor[] = { 200, 0, 200 }; //Red=200, Green=0, Blue=200
```



The texture is rendered as since the purple pixels (Red = 128, Green = 0, Blue = 128) are within the color key interval.

The code below is from `CTextureManager::ImageOperationCompleteL(...)`, and is responsible for creating the alpha channel used for transparency:

```
/*
 * If a color key interval was specified, then we create an alpha
 * channel.
 */
TUint8 * TextureDataWithAlphaChannel = new (ELeave)
TUint8[TextureSize * 4];
TInt TextureSize3 = TextureSize * 3;

// walk through the original image and copy it over
// to the destination image
for (TInt i = 0, j = 0; i < TextureSize3; i += 3, j += 4)
{
    /*
     * If a pixel color is within the color key interval, we
     * assign it
     * an alpha value of 0, Which makes it transparent.
     * Otherwise the alpha value of the pixel is set to 255, full
     * alpha.
     */
    if (TextureData[i] >= iCurrentRequest->iMinColorKey[0] &&
        TextureData[i + 1] >= iCurrentRequest->iMinColorKey[1] &&
        TextureData[i + 2] >= iCurrentRequest->iMinColorKey[2] &&
        TextureData[i] <= iCurrentRequest->iMaxColorKey[0] &&
```

```

TextureData[i + 1] <= iCurrentRequest->iMaxColorKey[1] &&
TextureData[i + 2] <= iCurrentRequest->iMaxColorKey[2] )
TextureDataWithAlphaChannel[j + 3] = 0;
else
TextureDataWithAlphaChannel[j + 3] = 255;

TextureDataWithAlphaChannel[j] = TextureData[i];
TextureDataWithAlphaChannel[j + 1] = TextureData[i + 1];
TextureDataWithAlphaChannel[j + 2] = TextureData[i + 2];
}

```

Alpha testing must be enabled for color keying. Alpha testing is enabled as follows:

```

glEnable(GL_ALPHA_TEST);
glAlphaFunc(GL_GREATER, 0.0f);

```

The Billboard, and Snow and Rain examples (see Sections 5.14 and 5.15) use the color keying techniques.

Another feature of the texture manager is the ability to perform 2D blitting.

2D blitting is done through `CTextureManager::Blit(...)` by rendering a textured polygon on the near depth-clipping plane. The method supports alpha blending and color keying during blit operations.

The LensFlare example (see Section 5.13) uses `CTextureManager::Blit(...)` to render the lens flare effect.

5.12 Camera

This example shows how to implement a viewing system for an OpenGL ES 3D application.

One of the reasons to implement a viewing system for OpenGL ES 3D applications is the absence of the `gluLookAt` function in the OpenGL ES specification.

Another reason is that to make 3D game engines as flexible as possible, programmers need viewing systems that can give the viewer full freedom of movement.

The viewing system is based on two classes: `TCamera` and `T3Dmodel`, both defined in `Utils/util.H` and implemented in `Utils/Utils.CPP`.

- Class `TCamera` uses a UVN vector system. The system works by defining the camera's world space position and "look-at point" within the scene to derive 3 vectors, U, V and N that describe the orientation of the camera space coordinate axes. These vectors are used to produce the view matrix. The main method for this class is `TCamera::LookAt(...)`, which works as the `gluLookAt` function.
- Class `T3DModel` is an abstraction of a 3D model, represented by a world space position and a Yaw, Pitch, and Roll orientation. The class provides two important methods:
 - o `T3DModel::MakeWorldViewMatrix(...)` – builds a matrix to transform coordinates from world space into camera space.
 - o `T3DModel::MakeBillboardWorldViewMatrix(...)` – builds a billboard matrix that is a matrix to transform coordinates from world space into camera space, and at the same time rotate objects so that they always face the camera. This method is used in the Billboard, Rain/Snow examples to draw trees.

See Billboard (Section 5.14), Snow and Rain (Section 5.15), and LensFlare (Section 5.13) examples for additional information.

5.13 LensFlare

This example shows how to create a lens flare effect. Lens flare is the result of bright light refraction into the camera lens. The class `CLensFlareEffect` used in this example is defined in `Utils/Utils.h` and implemented in `Utils/Utils.cpp`.

To produce a lens flare effect, this example uses a lens flare vector, which is a 2D vector that goes from the screen's center to the light's position. This vector is used to position the different parts of the lens flare. This approach is illustrated in Figure 5.17.

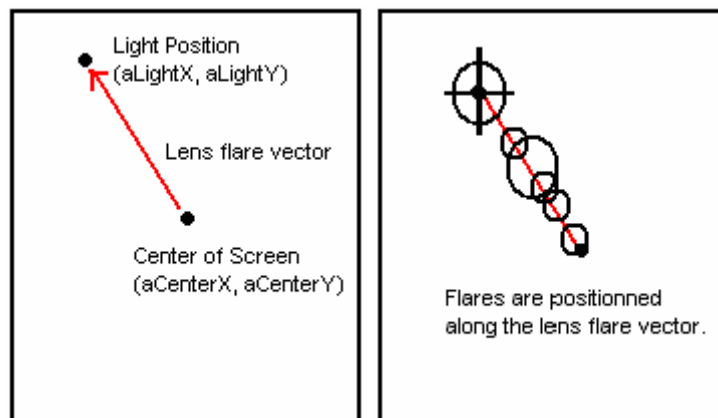


Figure 5.17: An illustration of the method used to produce the lens flare effect

File `Utils\utils.h` contains a type named `TFlareConfig` used to describe the property of a single flare within the lens flare. The type is defined as follows:

```
struct TFlareConfig
{
    TUint8 iIndex;
    GLfloat iLengthScale;
    GLfloat iImageScale;
};
```

Attribute `iIndex` is the index (zero based) of the flare's texture, `iLengthScale` is a scalar value that gives the position of the flare, and `iImageScale` is used to scale the flare's texture. Figure 5.18 shows images of LensFlare example running on the emulator.

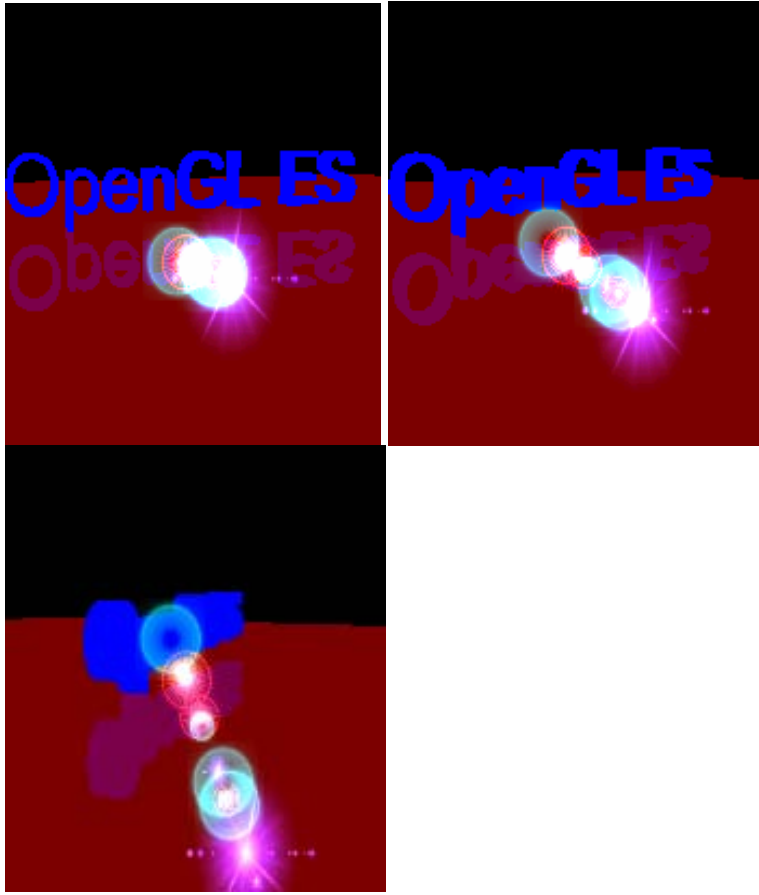


Figure 5.18: Screenshots of LensFlare running on the emulator

5.14 Billboard

This example demonstrates the billboarding technique. Billboarding is a way of making 2D sprites look 3D. It is achieved by applying a 2D texture bitmap to a rectangular primitive. The primitive is then rotated so that it always faces the camera. This technique is often used in games to render trees, bushes, clouds, fire, smoke, explosions, and so on. Figure 5.19 illustrates the idea of billboarding.

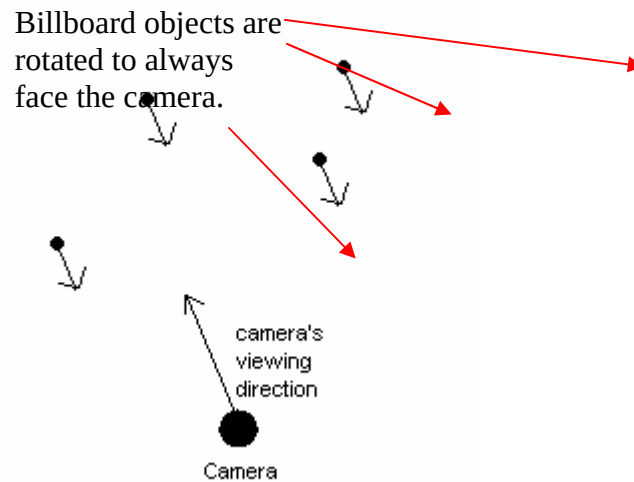


Figure 5.19: An illustration of the billboarding technique

By using billboarding, 3D applications can sometimes gain speed, as the billboard is much faster to draw than a detailed model of a tree, for example. Figure 5.20 shows a screenshot of Billboard example, where the trees are billboards, running on the emulator.

This example uses the `T3DModel::MakeBillboardWorldViewMatrix(...)` mentioned in Section 5.12 in order to achieve the effect. As in this example, billboarding is sometimes combined with color keying to render objects that have non-rectangular images.



Figure 5.20: Screenshot of Billboard running on the emulator

5.15 Snow and Rain

These examples demonstrate how to implement a particle engine.

Particle systems are used in games to produce many effects such as snow, rain, smoke, sparks, explosions, and more. In fact, what looks like snowfall, raindrops, smoke or sparks is just a set of partially transparent textured rectangles moving through the scene. The trick is to make them behave like snow drops or smoke. For example, in the real world, raindrops fall since they are affected by gravity, smoke rises but is also affected by wind and dissolves after some time.

The Rain example uses line primitives to render raindrops, and the Snow example uses rectangle primitives for snowflakes. But in both examples the particles are translucent (alpha blending is enabled), they always face the camera, their positions are governed by some equations and they are updated in each frame.

Figure 5.21 and Figure 5.22 show screenshots of Snow and Rain examples running on the emulator.



Figure 5.21: Screenshot of Snow running on the emulator



Figure 5.22: Screenshot of Rain running on the emulator