Chapter 6 Graphics with OpenGL ES 2.X

6.1 Programmable Pipeline

As we have mentioned in Chapter 4, OpenGL ES is an application programming interface (API) for advanced 3D graphics targeted for embedded devices such as cell phones, digital health kits, and tablets. It consists of well-defined subsets of desktop OpenGL, including profiles for floating-point and fixed-point systems and the EGL specification for portability that binds native windowing systems. (EGL (Embedded Graphics Library) is a native platform graphics interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native window system.) The API is created and maintained by the Khronos Group, which is a member-funded industry consortium founded in January 2000, focusing on the creation of open standard and royalty-free APIs for embedded devices.

OpenGL ES 1.X is based on the traditional fixed pipeline graphics architecture, in which the functionality of each processing stage is fixed. It offers acceleration, image quality and performance. OpenGL ES 2.X supports fully programmable pipeline architecture, a trend in graphics hardware, for creating 3D graphics. In ES 2.X, one can use the OpenGL Shading Language (glsl) to write vertex shaders and fragment shaders. For details of the APIs, one can refer to the Web site,

http://www.khronos.org/opengles/2_X/

Android supports both OpenGL ES 1.X and OpenGL ES 2.X. Figure 6-1 below shows the traditional fixed function pipeline architecture of OpenGL used by OpenGL ES 1.X.

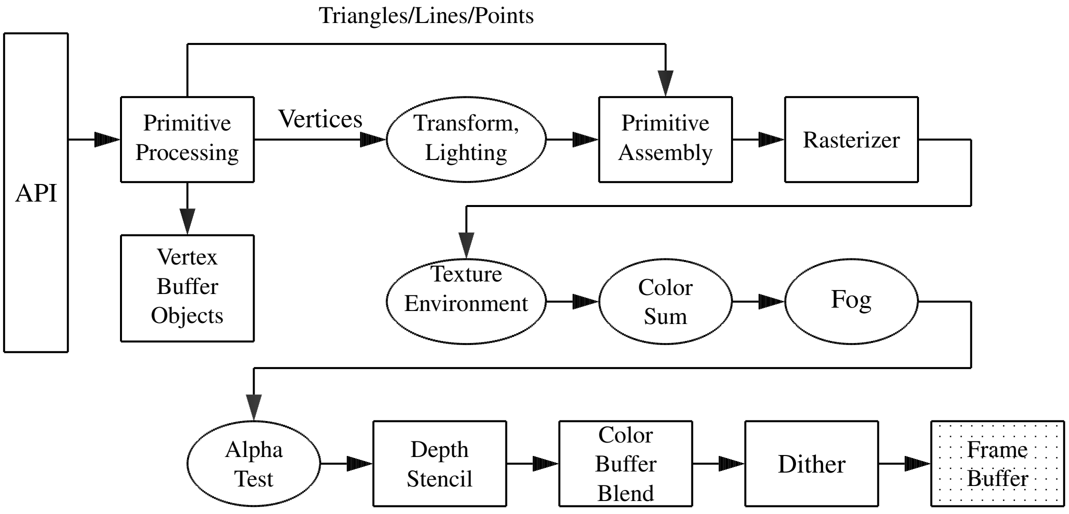


Figure 6-1. Fixed Function Pipeline

OpenGL ES 2.X does not support this fixed pipeline architecture. Its programmable pipeline replaces the fixed function transformation and fragment pipeline of OpenGL 1.X as shown in Figure 6-2 below.

OpenGL ES 2.0 is defined relative to the OpenGL 2.0 specification, emphasizing a programmable 3D graphics pipeline that allows users to create shader and program objects and to write vertex and fragment shaders using OpenGL Shading Language (glsl). It combines a glsl version for programming vertex and fragment shaders that has been adapted for embedded platforms, with a streamlined API from OpenGL ES 1.1 with the fixed functionality replaced by shader programs. This helps minimize the cost and power consumption of advanced programmable graphics subsystems.

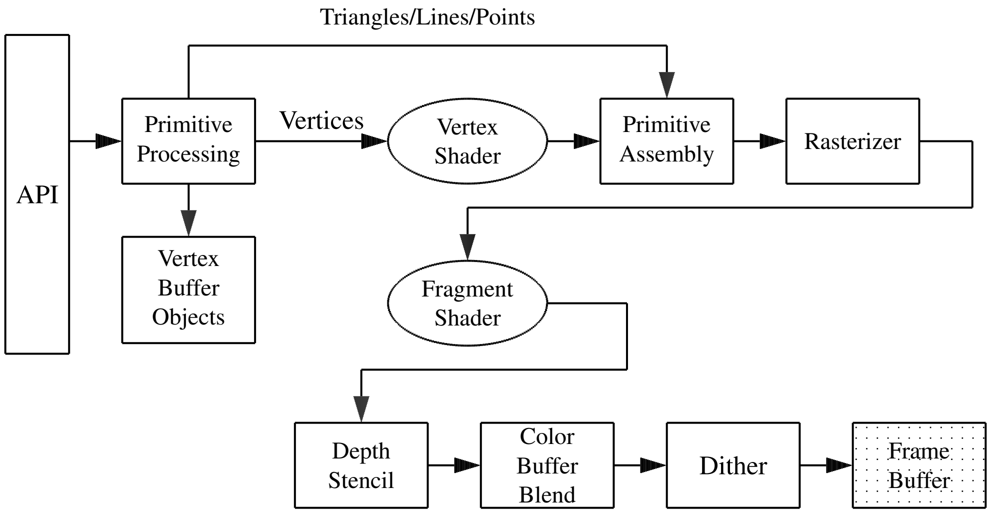


Figure 6-2. ES 2.X Programmable Pipeline

6.2 OpenGL Shading Language (GLSL)

Before we discuss using OpenGL ES 2.X to write graphics applications in Android, we give a brief introduction to OpenGL Shading Language (glsl), a C-like language with some C++ features designed for 3D graphics programming. It is part of OpenGL and thus can be naturally integrated with OpenGL programs with ease. The language is mainly used for processing numerics, but not for strings or characters.

6.2.1 OpenGL Shaders Execution Model

We can consider a driver as a piece of software that manages the access of a hardware. In this sense, we can view OpenGL libraries as drivers because they manage shared access to the underlying graphics hardware and applications communicate with graphics hardware by calling OpenGL functions. An OpenGL shader is embedded in an OpenGL application and may be viewed as an object in the driver to access the hardware. We use the command **glCreateShader**() to allocate within the OpenGL driver the data structures needed to store an OpenGL shader. The source code of a shader is provided by an application by calling **glShaderSource**() and we have to provide the source code as a **null-terminated** string to this function. Figure 6-3 below shows the steps to create a shader program for execution.

As shown in Figure 6-2, there are two kinds of shaders, the vertex shaders and the fragment shaders. A **vertex shader** (program) is a shader running on a **vertex processor**, which is a programmable unit that operates on incoming vertex values. This processor usually performs traditional graphics operations including the following:

1. vertex transformation

2. normal transformation and normalization
3. texture coordinate generation
4. texture coordinate transformation
5. lighting
6. color material application

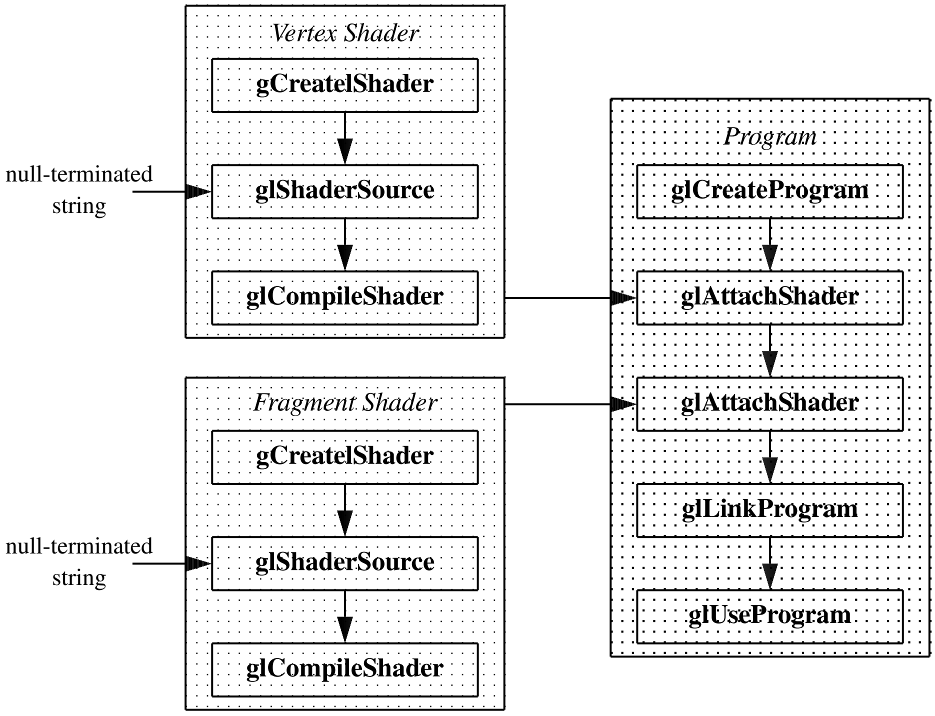


Figure 6-3 Shader Program Development

The following is an example of a simple “pass-through” vertex shader which does not do anything.

```
//A simple pass-through vertex shader
void main()
{
  gl_Position=gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
}
```

A shader program must pass a null-terminated string to the OpenGL function **glShaderSource**(), which is then compiled and linked with the rest of the application by other glsl commands as shown in Figure 6-3.

A **fragment shader** is a shader running on a **fragment processor**, which is a programmable unit that operates on fragment values. A fragment is a pixel plus its attributes such as color and transparency. A fragment shader is executed after the rasterization. Therefore a fragment processor operates on each fragment rather than on each vertex. It usually performs traditional graphics operations including:

1. operations on interpolated values
2. texture access

3. texture application
4. fog effects
5. color sum
6. pixel zoom
7. scaling
8. color table lookup
9. convolution
10. color matrix operations

The following is an example of a simple fragment shader, which sets the color of each fragment for render.

```
//A simple fragment shader
void main()
{
  gl_FragColor = gl_FrontColor;
}
```

6.2.2 OpenGL Shading Language API

Figure 6-3 above shows the development steps of a glsl shader program. The following table lists the OpenGL functions involved in the process.

Table 6-1 OpenGL Commands for Embedding Shaders

glCreateShader()	Creates one or more shader objects.
glShaderSource()	Provides source codes of shaders.
glCompileShader()	Compiles each of the shaders.
glCreateProgram()	Creates a program object.
glAttachShader()	Attach all shader objects to the program.
glLinkProgram()	Link the program object.
glUseProgram()	Install the shaders as part of the OpenGL program.

The following discusses the usual steps to develop an OpenGL shader program.

1. **Creating a Shader Object**

   We first create an empty shader object using the function **glCreateShader**, which has the following prototype:

   Gluint **glCreateShader** (GLenum *shaderType*)
   Creates an empty shader.
   shaderType specifies the type of shader to be created. It can be either GL_VERTEX_SHADER or GL_FRAGMENT_SHADER.
   Return: A non-zero integer handle for future reference.

   ---

2. **Providing Source Code for the Shader**

   We pass the source code to the shader as a null-terminated string using the function **glShaderSource** which has the prototype:

   ---

   void **glShaderSource** (GLuint *shader*, GLsizei *count*, const GLchar ***string*, const GLint *lengthp)

Defines a shader's source code.
shader is the shader object created by glCreateShader().
string is the array of strings specifying the source code of the shader.
count is the number of strings in the array.
lengthp points to an array specifying the lengths of the strings. If NULL, the strings are NULL-terminated.

The source code can be hard-coded as a string in the OpenGL program or it can be saved in a separate file and read into an array as a null-terminated string, which ends with the null character '\0'.

3. **Compiling Shader Object**

We use the function **glCompileShader** to compile the shader source code to object code. This function has the following prototype.

---

void **glCompileShader** (GLuint *shader*)
Compiles the source code strings stored in the shader object *shader*.
The function **glShaderInfoLog** gives the compilation log.

---

4. Linking and Using Shaders

Each shader object is compiled independently. To create a shader program, we need to link all the shader objects to the OpenGL application. These are done within the C/C++ application using the functions **glCreateProgram, glAttachShader, glLinkProgram**, and **glUseProgram**, which have the prototypes listed below. These are done while we are running the C/C++ application. Performing the steps of compiling and linking shader objects are simply making C function calls.

GLuint **glCreateProgram** (void)
Creates an empty program object and returns a non-zero integer handle for future reference.

void **glAttachShader** (GLuint *program*, GLuint *shader*)
Attaches the shader object specified by *shader* to the program object specified by *program*.

void **glLinkProgram** (GLuint *program*)
Links the program objects specified by *program*.

void **glUseProgram** (GLuint *program*)
Installs the program object specified by *program* as part of current rendering state.
If *program* is 0, the programmable processors are disabled, and fixed functionality is used for both vertex and fragment processing.

---

5. **Cleaning Up**

At the end, we need to release all the resources taken up by the shaders. The clean up is done by the commands,

void **glDeleteShader** (GLuint *shader*),
void **glDeleteProgram** (GLuint *program*),
void **glDetachShader** (GLuint *program*, GLuint *shader*).

Listing 6-1 below is a complete example of a shader program; the OpenGL application is the C/C++ program *shaderdemo.cpp*, which does the shader creation, reading shader source code, shader compilation and shader linking. The shader source codes are hard-coded in the program. In compiling *shaderdemo.cpp*, we need to link the GL extension library by "-lGLEW". If we compile "shaderdemo.cpp" to the executable "shaderdemo", we can run the shader program by typing "./shaderdemo" and press 'Enter'.

Program Listing 6-1 Complete Example of a Shader Program

shaderdemo.cpp

```
/*
 shaderdemo.cpp
 Sample program showing how to write GL shader programs.
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <GL/glew.h>
#include <GL/glut.h>

using namespace std;

/*
  Global handles for the currently active program object,
  with its two shader objects
*/
GLuint programObject = 0;
GLuint vShader = 0;
GLuint fShader = 0;
static GLint win = 0;

// String defining vertex shader
char vertexStr[] = "              \
 //a minimal vertex shader  \n   \
 void main(void)                 \
 {                               \
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex; \
 }                               \
";

// String for fragment shader
char fragmentStr[] = "             \
 //a minimal fragment shader \n    \
 void main(void)                   \
 {                                 \
   gl_FragColor = vec4( 1, 0, 0, 1); \
 }                                 \
";
```

```
int readShaderSource( char str[],  GLchar **shader )
{
   // Allocate memory to hold the source of our shaders.
   int shaderSize;

   shaderSize = strlen ( str );

   if ( shaderSize <= 0 ){
       printf("Shader string  empty\n" );
       return 0;
   }
   // Allocate memory for shader
   *shader = (GLchar *) malloc( shaderSize + 1);

   // Read the source code
   strcpy ( *shader, str );

   return 1;
}

int installShaders(const GLchar *vertex, const GLchar *fragment)
{
   GLint  vertCompiled, fragCompiled;  // status values
   GLint  linked;

   // Create a vertex shader object and a fragment shader object
   vShader = glCreateShader(GL_VERTEX_SHADER);
   fShader = glCreateShader(GL_FRAGMENT_SHADER);

   // Load source code strings into shaders, compile and link
   glShaderSource(vShader, 1, &vertex, NULL);
   glShaderSource(fShader, 1, &fragment, NULL);

   glCompileShader(vShader);
   glGetShaderiv(vShader, GL_COMPILE_STATUS, &vertCompiled);
   glCompileShader( fShader );
   glGetShaderiv( fShader, GL_COMPILE_STATUS, &fragCompiled);

   if (!vertCompiled || !fragCompiled)
       return 0;

   // Create a program object and attach the two compiled shaders
   programObject = glCreateProgram();
   glAttachShader( programObject, vShader);
   glAttachShader( programObject, fShader);

   // Link the program object
   glLinkProgram(programObject);
   glGetProgramiv(programObject, GL_LINK_STATUS, &linked);

   if (!linked)
       return 0;
   // Install program object as part of current state
   glUseProgram(programObject);
   return 1;
}

int init(void)
{
  const char *version;
  GLchar *vShaderSource, *fShaderSource;
  int loadstatus = 0;

  version = (const char *) glGetString(GL_VERSION);
  if (version[0] != '2' || version[1] != '.') {
     printf("This program requires OpenGL 2.x, found %s\n", version);
     exit(1);
  }
  readShaderSource( vertexStr, &vShaderSource );
  readShaderSource( fragmentStr, &fShaderSource );
  loadstatus = installShaders(vShaderSource, fShaderSource);

  return loadstatus;
}

static void reshape(int width, int height)
{
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glFrustum(-1.0, 1.0, -1.0, 1.0, 5.0, 25.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0f, 0.0f, -15.0f);
}

void cleanUp(void)
{
  glDeleteShader ( vShader );
  glDeleteShader ( fShader );
  glDeleteProgram ( programObject );
  glutDestroyWindow ( win );
}

static void idle(void)
{
  glutPostRedisplay();
}

static void keyPressed ( unsigned char key, int x, int y )
{
  switch(key) {
  case 27:
     cleanUp();
     exit(0);
     break;
  }
  glutPostRedisplay();
```

```
}

void display(void)
{
  GLfloat vec[4];

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glClearColor( 1.0, 1.0, 1.0, 0.0 ); //get white background color
  glColor3f( 0, 1, 0 ); //green, have no effect if shader is loaded
  glLineWidth ( 3 );
  glutWireSphere(2.0, 12, 6);
  glutSwapBuffers();
  glFlush();
}

int main(int argc, char *argv[])
{
  int success = 0;

  glutInit(&argc, argv);
  glutInitWindowPosition( 0, 0);
  glutInitWindowSize(200, 200);
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
  win = glutCreateWindow(argv[0]);
  glutReshapeFunc(reshape);
  glutKeyboardFunc ( keyPressed );
  glutDisplayFunc(display);
  glutIdleFunc(idle);
  // Initialize the "OpenGL Extension Wrangler" library
  glewInit();
  success = init();
  if ( success )
    glutMainLoop();
  return 0;
}
```

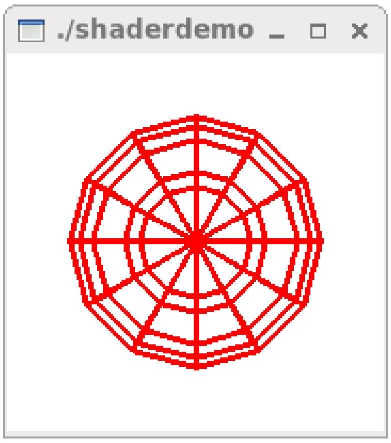


Figure 6-4 Output of Shader Program *shaderdemo.cpp*

6.2.3 Data Types in GLSL

There are four main data types in GLSL: **float, int, bool**, and **sampler**. Vector types are available for the first three types:

vec2, vec3, vec4	2D, 3D and 4D floating point vector
ivec2, ivec3, ivec4	2D, 3D and 4D integer vector
bvec2, bvec3, bvec4	2D, 3D and 4D boolean vectors

For floats there are also matrix types:

mat2, mat3, mat4	$2 \times 2, 3 \times 3, 4 \times 4$ floating point matrix

Samplers are types used for representing textures:

sampler1D, sampler2D, sampler3D	1D, 2D and 3D texture
samplerCube	Cube Map texture
sampler1Dshadow, sampler2Dshadow	1D and 2D depth-component texture

Attributes, Uniforms and Varyings

GLSL shaders have three different input-output data types for passing data between vertex and fragment shaders, and the OpenGL application. The data types are **uniform, attribute** and **varying**. They must be declared as global (visible to the whole shader object). The variables have the following properties:

1. **Uniforms** : These are read-only variables (i.e. A shader object can only read the variables but cannot change them.). Their values do not change during a rendering. Therefore, Uniform variable values are assigned outside the scope of **glBegin/glEnd**. Uniform variables are used for sharing data among an application program, vertex shaders, and fragment shaders.

2. **Attributes**: These are also read-only variables. They are only available in vertex shaders. They are used for variables that change at most once per vertex in a vertex shader. There are two types of attribute variables, user-defined and built-in. The following are examples of user-defined attributes:

   ```
   attribute float x;
   attribute vec3 velocity, acceleration;
   ```

   Built-in variables include OpenGL state variables such as color, position, and normal; the following are some examples:

   ```
   gl_Vertex
   gl_Color
   ```

3. **Varyings**: These are read/write variables, which are used for passing data from a vertex shader to a fragment shader. They are defined on a per-vertex basis but are interpolated over the primitive by the rasterizer. They can be user-defined or built-in.

Built-in Types

The following tables list some more of the GLSL built-in types.

Table D-2 Built-in Attributes (for Vertex Shaders)

gl_Vertex	4D vector representing the vertex position
gl_Normal	3D vector representing the vertex normal
gl_Color	4D vector representing the vertex color
gl_MultiTexCoord$n$	4D vector representing the texture coordinate of texture n

Table D-3 Built-in Uniforms (for Vertex and Fragment Shaders)

gl_ModelViewMatrix	4×4 Matrix representing the model-view matrix
gl_ModelViewProjectionMatrix	4×4 Model-view-projection matrix
gl_NormalMatrix	3×3 Matrix used for normal transformation

Table D-4 Built-in Varyings (for Data Sharing between Shaders)

gl_FrontColor	4D vector representing the primitives front color
gl_BackColor	4D vector representing the primitives back color
gl_TexCoord[n]	4D vector representing the n-th texture coordinate
gl_Position	4D vector representing the final processed vertex position (vertex shader only)
gl_FragColor	4D vector representing the final color written in the frame buffer (fragment shader only)
gl_FragDepth	float representing the depth written in the depth buffer (fragment shader only)

GLSL has many built in functions, including

1. trigonometric functions: **sin, cos, tan**
2. inverse trigonometric functions: **asin, acos, atan**
3. mathematical functions: **pow, log2, sqrt, abs, max, min**
4. geometrical functions: **length, distance, normalize, reflect**

The following is an example of using various data types; it consists of a vertex shader and a fragment shader for defining a modified Phong lighting model.

Program Listing 6-2 Shaders for Modified Phong Lighting

(a) Vertex Shader: phong.vert

```
//phong.vert
varying vec3 N;  //normal direction
varying vec3 L;  //light source direction
varying vec3 E;  //eye position

void main(void)
{
   gl_Position =gl_ModelViewMatrix*gl_Vertex;
   vec4 eyePosition = gl_ModelViewProjectionMatrix*gl_Vertex;
   vec4 eyeLightPosition = gl_LightSource[0].position;

   N = normalize( gl_NormalMatrix*gl_Normal );
   L = eyeLightPosition.xyz - eyePosition.xyz;
   E = -eyePosition.xyz;
}
```

(b) Fragment Shader: phong.frag

```
//phong.frag
varying vec3 N;
varying vec3 L;
varying vec3 E;

void main()
{
  vec3 norm = normalize(N);
  vec3 lightv = normalize(L);
  vec3 viewv = normalize(E);
  vec3 halfv = normalize(lightv + viewv);
  float f;
  if(dot(lightv, norm)>= 0.0) f =1.0;
  else f = 0.0;

  float Kd = max(0.0, dot(lightv, norm));
  float Ks = pow(max(0.0, dot(norm, halfv)), gl_FrontMaterial.shininess);
  vec4 diffuse = Kd * gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse;
  vec4 ambient = gl_FrontMaterial.ambient*gl_LightSource[0].ambient;
  vec4 specular = f*Ks*gl_FrontMaterial.specular*gl_LightSource[0].specular;
  gl_FragColor = ambient + diffuse + specular;
}
```

6.3 Android Graphics with ES 2.0

6.3.1 Drawing a Triangle

The *android.opengl.GLES20* package provides the interface to OpenGL ES 2.0, which supports OpenGL Shader Library APIs. In earlier days, android emulator did not support this feature and one had to use a real android device to test its code in the development process. However, newer emulator versions begin to support ES 2.0.

To use the OpenGL ES 2.0 API, we have to add the following declaration in the manifest file, *AndroidManifest.xml*:

```
  <uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

If our application uses texture compression, we also need to declare which compression formats we support so that devices that do not support theses formats will not run the application:

```
  <supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
  <supports-gl-texture android:name="GL_OES_compressed_paletted_texture" />
```

The following steps walk you through the process of creating an android glsl application, which simply draws a magenta triangle. Suppose we use Eclipse IDE in the development process. We call our project and application *Glsl1*, and the package *opengl.glsl1*. In the example, the vertex and fragment shaders are hard-coded as null-terminated strings in the program:

1. In Eclipse IDE, click **File** > **New** > **Android Application Project** (You may need to click **Poject ..** first if your Eclipse has been setup differently.). The *New Android Application* diaglog shows up.

2. Enter *Glsl1, Glsl1*, and *opengl.glsl1* for the names of application, project, and package respectivley. You may use defaults for other features for selection. Then click **Next** > **Next** > **Next** and **Next**.

3. You may use the default names *MainActivity* and *activity_main* for the names of Activity and Layout. Then click **Finish** to create the project *Glsl1*.

4. In the base directory *Glsl1*, you can find the xml file, *AndroidManifest.xml*, which presents essential information about the app to the Android system. Add the following statement to this file:
   <uses-feature android:glEsVersion="0x00020000" android:required="true" />
   which tells the system that the app requires OpenGL ES 2.0. You may add the statement before the $< application >$ element in the file.

5. Modify the main program *MainActivity.java* in the subdirectory *src/opengl/glsl1* to the following, which along with comments is self-explained:

```
------------------------------------------------------------------------
package opengl.glsl1;

import android.app.Activity;
import android.content.Context;
import android.opengl.GLSurfaceView;
import android.os.Bundle;

public class MainActivity extends Activity {

  private GLSurfaceView glView;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Create a GLSurfaceView instance and set it
    // as the ContentView for this Activity.
    glView = new MyGLSurfaceView(this);
    setContentView(glView);
  }
}

class MyGLSurfaceView extends GLSurfaceView {

  public MyGLSurfaceView(Context context){
    super(context);
    // Create an OpenGL ES 2.0 context
    setEGLContextClientVersion ( 2 );
  public MyGLSurfaceView(Context context){
    super(context);
    // Create an OpenGL ES 2.0 context
    setEGLContextClientVersion ( 2 );
    // Render the view only when there is a change in the drawing data
    // Set the Renderer for drawing on the GLSurfaceView
    setRenderer(new MyRenderer());
  }
}
------------------------------------------------------------------------
```

6. Create a new class by clicking **File** > **New** > **Class**. Enter *MyRenderer* for the name and use defaults for other entries. This creates the class file *src/opengl/glsl1/MyRenderer.java*. Modify this file to the following:

```
----------------------------------------------------------------------
package opengl.glsl1;

import android.opengl.GLES20;
import android.opengl.GLSurfaceView;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

public class MyRenderer implements GLSurfaceView.Renderer {

  private Triangle triangle;
  public void onSurfaceCreated(GL10 unused, EGLConfig config) {
    // Set the background frame color
    GLES20.glClearColor(0.9f, 0.9f, 0.9f, 1.0f);
    // construct a triangle object
    triangle = new Triangle();
  }

  public void onDrawFrame(GL10 unused) {
    // Redraw background color
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    triangle.draw();
  }

  public void onSurfaceChanged(GL10 unused, int width, int height) {
     GLES20.glViewport(0, 0, width, height);
  }
}
----------------------------------------------------------------------
```

This class renders a *Triangle* object, which is defined in the next step.

7. Create another new class by clicking **File** > **New** > **Class**. Enter *Triangle* for the name and use defaults for other entries. This creates the class file *src/opengl/glsl1/Triangle.java*. Modify this file to the following:

```
----------------------------------------------------------------------
package opengl.glsl1;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import android.opengl.GLES20;

public class Triangle {
  // Source code of vertex shader
  private final String vsCode =
    "attribute vec4 vPosition;" +
        "void main() {" +
    "  gl_Position = vPosition;" +
  "}";

   // Source code of fragment shader
   private final String fsCode =
     "precision mediump float;" +
     "uniform vec4 vColor;" +
     "void main() {" +
     "  gl_FragColor = vColor;" +
         "}";

  private int program;
  private int vertexShader;
  private int fragmentShader;
  private FloatBuffer vertexBuffer;
  private int vertexCount = 3;

   // number of coordinates per vertex in this array
   static final int COORDS_PER_VERTEX = 3;
   static float triangleCoords[] = {   // in counterclockwise order:
      0.0f,  0.9f, 0.0f, // top vertex
     -0.5f,  0.1f, 0.0f, // bottom left
      0.5f,  0.1f, 0.0f  // bottom right
    };

    // Set color of displaying object
    // with red, green, blue and alpha (opacity) values
    float color[] = { 0.9f, 0.1f, 0.9f, 1.0f };

    // Create a Triangle object
    Triangle(){
      // create empty OpenGL ES Program,load,attach,and link shaders
      program = GLES20.glCreateProgram();
      vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vsCode);
      fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fsCode);
      // add the vertex shader to program
      GLES20.glAttachShader(program,vertexShader);
      // add the fragment shader to program
      GLES20.glAttachShader(program,fragmentShader);
      GLES20.glLinkProgram(program); //creates ES program executables
      GLES20.glUseProgram( program); //use shader program

      //initialize vertex byte buffer for shape coordinates with
      // paramters (number of coordinate values * 4 bytes per float)
      //use the device hardware's native byte order
      ByteBuffer bb=ByteBuffer.allocateDirect(triangleCoords.length*4);
      bb.order ( ByteOrder.nativeOrder() );

      // create a floating point buffer from the ByteBuffer
      vertexBuffer = bb.asFloatBuffer();
      // create a floating point buffer from the ByteBuffer
      vertexBuffer = bb.asFloatBuffer();
      // add the coordinates to the FloatBuffer
      vertexBuffer.put(triangleCoords);
      // set the buffer to read the first coordinate
      vertexBuffer.position(0);
    } //Triangle Constructor

    public static int loadShader (int type, String shaderCode ) {

      // create a vertex shader type (GLES20.GL_VERTEX_SHADER)
      // or a fragment shader type (GLES20.GL_FRAGMENT_SHADER)
      int shader = GLES20.glCreateShader(type);

      // pass source code to the shader and compile it
      GLES20.glShaderSource(shader, shaderCode);
      GLES20.glCompileShader(shader);

      return shader;
    }

    public void draw() {
      // Add program to OpenGL ES environment
      GLES20.glUseProgram(program);

      //get handle to vertex shader's attribute variable vPosition
      int positionHandle=GLES20.glGetAttribLocation(program,
                                                      "vPosition");

      // Enable a handle to the triangle vertices
      GLES20.glEnableVertexAttribArray(positionHandle);

      // Prepare the triangle coordinate data
      GLES20.glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                             GLES20.GL_FLOAT, false, 0, vertexBuffer);

      // get handle to fragment shader's uniform variable vColor
      int colorHandle=GLES20.glGetUniformLocation(program, "vColor");

      // Set color for drawing the triangle
      GLES20.glUniform4fv(colorHandle, 1, color, 0);

      // Draw the triangle
      GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

      // Disable vertex array
      GLES20.glDisableVertexAttribArray(positionHandle);
    }
 }
------------------------------------------------------------------------
```

This class draws a color triangle using a vertex shader and a fragment shader. The color is defined in the class by the array color[] and its values are passed to the fragment shader via the uniform variable vColor, which is a **vec4** defined in the fragment shader. The coordinates of the vertices of the triangle are defined by the array triangleCoords[] and stored in the byte buffer vertexBuffer. These values are passed to the vertex shader via the attribute variable *vPosition*, which is a **vec4** defined in the vertex shader.

The strings *vsCode* and *fsCode* defines the vertex shader and the fragment shader respectively. If we remove the quotes and the operator + in defining the strings, we can see that the codes of our vertex shader and fragment shader are:

```
// Source code of vertex shader
attribute vec4 vPosition;
void main() {
   gl_Position = vPosition;
}

// Source code of fragment shader
precision mediump float;
uniform vec4 vColor;
void main() {
    gl_FragColor = vColor;
}
```

Note that the keyword **precision** is used to specify the precision of any floating-point or integer-based variable. Keywords **lowp**, **mediump**, and **highp** are used to specifiy low, medium, and high precisions respectively.

8. Run the application by clicking *Run* > **Run** > **Android Application** > **OK**. We should see an output like the one shown in Figure 6-5, displaying a magneta triangle over a grey background. The color of the triangle is passed from the array variable *color* in the graphics application to the fragment shader **uniform** variable *vColor*, which sets the triangle color.

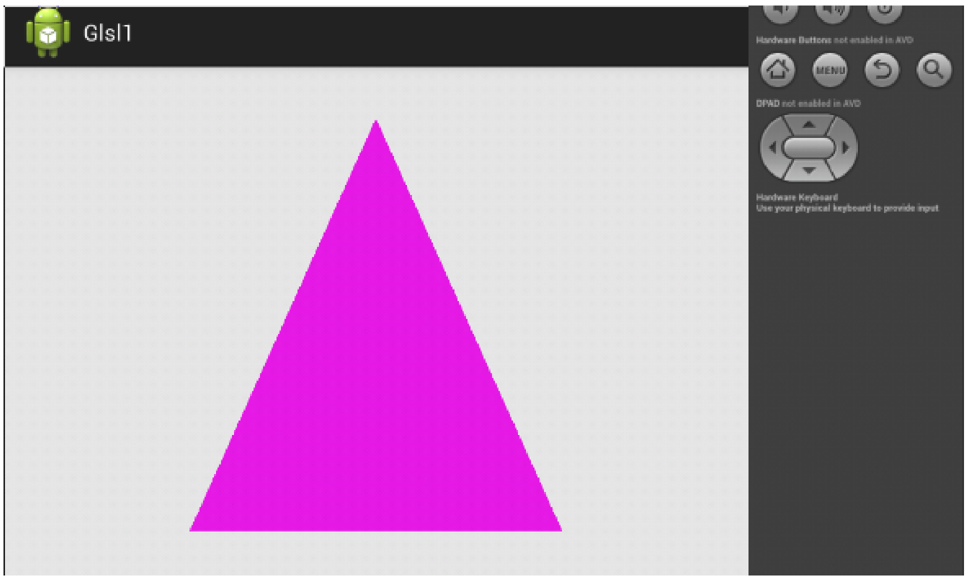
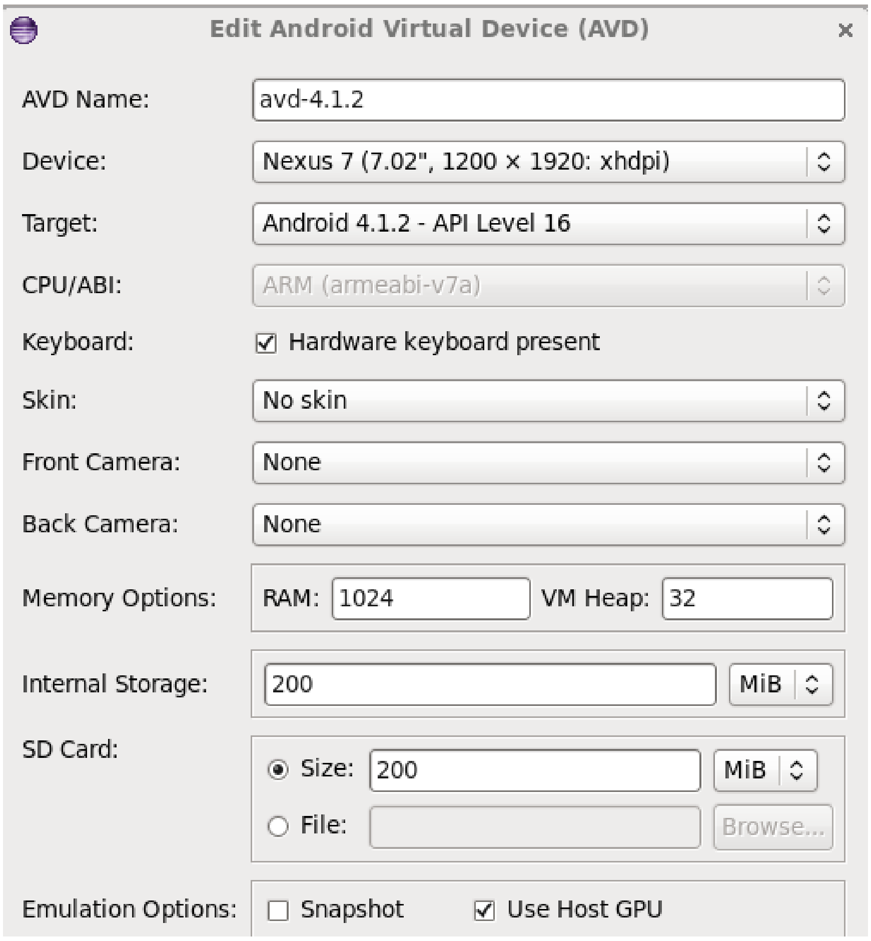

Glsl1
Hardware Buttons not enabled in AVD
MENU
DPAD not enabled in AVD
Hardware Keyboard
Use your physical keyboard to provide input

Figure 6-5 Output of Project *Glsl1*

Not all avd emulator configurations can run OpenGL ES 2.X. Figure 6-6 shows one of the avd configurations that we have used and works, which requires API level 16. It seems that for the emulator to work, a high screen resolution device needs to be chosen.


Edit Android Virtual Device (AVD)

AVD Name:	avd-4.1.2
Device:	Nexus 7 (7.02", 1200 × 1920: xhdpi)
Target:	Android 4.1.2 - API Level 16
CPU/ABI:	ARM (armeabi-v7a)
Keyboard:	☑ Hardware keyboard present
Skin:	No skin
Front Camera:	None
Back Camera:	None
Memory Options:	RAM: 1024 VM Heap: 32
Internal Storage:	200 MiB
SD Card:	● Size: 200 MiB ○ File: Browse...
Emulation Options:	☐ Snapshot ☑ Use Host GPU

Figure 6-6 An avd Emulator Configuration That Runs ES 2.X

6.3.2 Shaders in Files

In the above example, we have hard-coded the shaders in the class *Triangle*. It will be difficult to comprehend the hard-coded code when the shaders become complex. A better way to write the shaders is to handle them separately, saving them in separate files; we then read the codes from the files as null-terminated strings, passing them as input parameters to the function **glShaderSource**(). We discuss this method here, repeating the above example except that we save the shaders in raw data format that we have discussed in Chapter 5.

Suppose we call the project of this example *GlslRaw* and we have created all the classes of the project *Glsl1* above and have added the *uses-features* element to the file *AndroidManifest.xml* to use ES 2.0. The following are the additional steps required to finish this project. Besides reading the shader codes from the raw files, the main difference from before is that we pass the context handle of the main activity to the rendering class so that the class can access the raw files.

1. Click **File** > **New** > **Folder**. Enter *GlslRaw/res* for parent folder and *raw* for folder name. Click **Finish** to create the directory *res/raw*.

2. Click on the folder *raw* in the Package Explorer. Then click **File** > **New** > **File**. Enter *vshader* to create the file *res/raw/vshader*, which will be our vertex shader program. Similarly, create another file, *res/raw/fshader*, which will be our fragment shader program.

3. Write *vshader* with the vertex shader code:

```
// Source code of vertex shader
attribute vec4 vPosition;
void main() {
  gl_Position = vPosition;
}
```

4. Write *fshader* with the fragment shader code:

```
// Source code of fragment shader
precision mediump float;
uniform vec4 vColor;
void main() {
  gl_FragColor = vColor;
}
```

5. The only change to the above file *MainActivity.java* is to pass the context to the renderer by modifying the **setRenderer** statement to:
   setRenderer(**new MyRenderer** (*context*));

6. The file *MyRenderer.java* is almost the same as above. We use a new constructor to save the context of the thread that creates the object and pass this context handle to the *Triangle* object to access the raw files:

```
public class MyRenderer implements GLSurfaceView.Renderer {
  private Triangle triangle;
  private Context context;

  public MyRenderer( Context context0 ) {
     context = context0;
  }

  public void onSurfaceCreated(GL10 unused, EGLConfig config) {
    // Set the background frame color
    GLES20.glClearColor(0.9f, 0.9f, 0.9f, 1.0f);
    // construct a triangle object
    triangle = new Triangle( context );
  }
  .....
}
```

7. The main change to *Triangle.java* is to read the vertex and fragment shaders codes from the files *res/raw/vshader* and *res/raw/fshader* respectively. Other features remain the same as above:

```
public class Triangle
{
 private static String LOG_APP_TAG = "io_tag";
 private Context context;
 private String vsCode = null;
 private String fsCode = null;
 private int program;
 private int vertexShader;
 private int fragmentShader;
```

```
  .....

  // Constructor
  Triangle( Context context0){
   context = context0;
   // get shader codes from res/raw/vshader and res/raw/fshader
   vsCode = getShaderCode( GLES20.GL_VERTEX_SHADER );
   fsCode = getShaderCode( GLES20.GL_FRAGMENT_SHADER );
   program = GLES20.glCreateProgram(); // create empty OpenGL ES Program

   vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vsCode );
   fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fsCode );
   GLES20.glAttachShader ( program, vertexShader );
   GLES20.glAttachShader(program, fragmentShader);
   GLES20.glLinkProgram(program);
   GLES20.glUseProgram( program);
   .....
  }

  // get shader code from file
  protected String getShaderCode( int type ) {
   InputStream inputStream = null;
   String str = null;
   try {
    if ( type == GLES20.GL_VERTEX_SHADER )
     inputStream=context.getResources().openRawResource(R.raw.vshader);
    else
     inputStream=context.getResources().openRawResource(R.raw.fshader);
     byte[] reader = new byte[inputStream.available()];;
     while (inputStream.read(reader) != -1) {}
     str = new String ( reader );
   } catch(IOException e) {
      Log.e(LOG_APP_TAG, e.getMessage());
   }
   return  str;
  }
  .....
 }
```

When we run the application, we will get the same output as the previous example, which is shown in Figure 6-5.

From now on we will always put our shaders in the files *res/raw/vshader* and *res/raw/fshaer*.

6.3.3 Animation

We can easily do animation using glsl by passing a time parameter from the main OpenGL program to the shaders. The time parameter can be used to control the positions, orientations and other attributes of the graphics objects.

We consider a simple example where we display a color triangle expanding, shrinking, flipping and changing color. Suppose we call this project *GlslAnimate*, and have created or modified all the files used in the above project *GlslRaw*, including *MainActivity.java, MyRenderer.java, Triangle.java, vshader, fshader*, and *AndroidManifest.xml* except that now our package name is *opengl.glslanimate*. We need to do some modifications to the files to accomplish animation:

1. The file *MainActivity.java* is the same as before; we do not need to make any modification besides changing the package name.

2. For *MyRenderer*, we need to pass the elapsed time between rendering frames to the vertex shader to animate any desired motion. We use the method **elapsedRealtime**() of the class *SystemClock* discussed in Chapter 4 to obtain the time. This method returns a **long** that represents the time in milliseconds since the bootup of the device. We subtract this time value from the value when the *GLSurfaceView* is created and pass the difference to the *draw* method of the *Triangle* class, which in turn passes it to the vertex shader. The following is the complete code for this class:

```
// MyRenderer.java
package opengl.glslanimate;

import android.content.Context;
import android.os.SystemClock;
import android.opengl.GLES20;
import android.opengl.GLSurfaceView;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

public class MyRenderer implements GLSurfaceView.Renderer {
  private Triangle triangle;
  private Context context;
  private long t0;

  public MyRenderer( Context context0 ) {
    context = context0;
  }

  public void onSurfaceCreated(GL10 unused, EGLConfig config) {
    // Set the background frame color
    GLES20.glClearColor(0.9f, 0.9f, 0.9f, 1.0f);
    // construct a triangle object
    triangle = new Triangle( context );
    t0 = SystemClock.elapsedRealtime();  //initial time
  }

  public void onDrawFrame ( GL10 unused ) {
    // Redraw background color
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    SystemClock.sleep ( 100 );  // delay 0.1 s
    long t = SystemClock.elapsedRealtime() - t0;
    triangle.draw( t );
  }

  public void onSurfaceChanged(GL10 unused,int width,int height) {
    GLES20.glViewport(0, 0, width, height);
  }
}
```

3. For the *Triangle* class, only the **draw** method has been added the variable *deltaTHandle* to pass the elapsed time value to the vertex shader. The following is the new **draw** method: small

```
public class Triangle
{
 .....
 public void draw( long t ) {
  // Add program to OpenGL ES environment
  GLES20.glUseProgram(program);

  // get handle to vertex shader's vPosition member
  int positionHandle = GLES20.glGetAttribLocation(program,
                                              "vPosition");
  // Enable a handle to the triangle vertices
  GLES20.glEnableVertexAttribArray( positionHandle);

  // Prepare the triangle coordinate data
  int vertexStride = 0;
  GLES20.glVertexAttribPointer(positionHandle,COORDS_PER_VERTEX,
           GLES20.GL_FLOAT, false, vertexStride, vertexBuffer);
  // get handle to fragment shader's vColor member
  int colorHandle=GLES20.glGetUniformLocation(program,"vColor");
  // get handle to vertex shader's uniform variable deltaT
  int deltaTHandle=GLES20.glGetUniformLocation(program,"deltaT");
  // Set color for drawing the triangle
  GLES20.glUniform4fv(colorHandle, 1, color, 0);
  // set value for deltaT of vertex shader
  GLES20.glUniform1f( deltaTHandle, (float) t );
  // Draw the triangle
  GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

  // Disable vertex array
  GLES20.glDisableVertexAttribArray(positionHandle);
 }
}
```

4. The vertex shader in *res/raw/vshader* receives the value of the uniform variable *deltaT* from the **draw** method of the *Triangle* class of the application. It makes use of the sin function to obtain a multiplication factor s that changes between -1 and 1. This factor is multiplied to the vertex positions of the triangle and thus the triangle expands and shrinks according to the value of s. When s changes sign (e.g from positive to negative), the triangle flips over. It is defined as a global **varying** variable, so that its value can be passed to the fragment shader for other uses. The following is the complete code of the vertex shader:

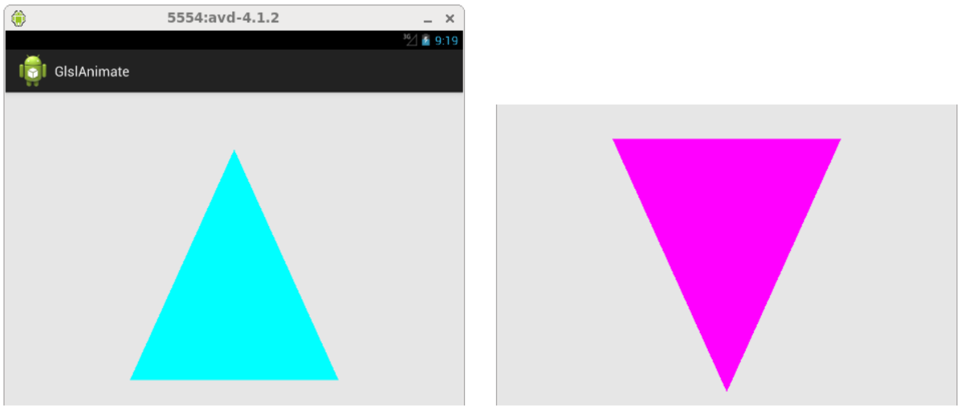
```
// vshader
precision mediump float;
uniform float deltaT;     //value from application program
attribute vec4 vPosition; //value from application program
varying float s;          //value also used in fragment shader
void main(void)
{
  s =  sin ( 0.001 * deltaT );  // scaling factor
  vec4 vPosition1 = vPosition * vec4 ( s, s, s, 1.0 );
  gl_Position = gl_ModelViewProjectionMatrix * vPosition1;
}
```

As s changes sinusoidally, the **vec4** variable *vPosition1* that defines the final vertex positions also changes. This s value is also passed to the fragment shader to define the drawing color.

5. The fragment shader *fshader* takes the value of the varying variable s calculated in the vertex shader. It changes the color of the triangle from cyan to magenta when s changes from positive to negative and vice versa. The complete code is shown below:

```
// fshader
 precision mediump float;
 varying float s;
 void main(void)
 {
   if ( s > 0 )
     gl_FragColor = vec4( 0, 1, 1, 1);
   else
     gl_FragColor = vec4( 1, 0, 1, 1);
 }
```

Figure 6-7 shows two frames of the output.

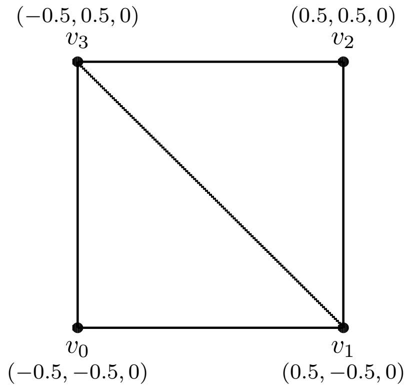


(a) (b)
Figure 6-7 Two Sample Frames of Animated Triangle

6.3.4 Drawing a Square

We have discussed in Chapter 4 that OpenGL ES polygon drawing primitives only support the drawing of triangles. To draw any other kind of polygon, we have to decompose the polygon into triangles. This is true even if we use shaders to draw the polygon. As an example, suppose we want to draw a square like the one shown in Figure 6-8 below.

To draw the square, we first draw the triangle $v_0v_1v_3$ and then draw $v_3v_1v_2$, ordering the vertices in a counter-clockwise direction. We will use the function **glDrawElements**() to perform this task. All we need to do is to supply the vertex coordinates and an array that contains the vertex indices.



Square $v_0v_1v_2v_3 = \triangle v_0v_1v_3 + \triangle v_3v_1v_2$

Figure 6-8. A Square Consisting of Two Triangles

Suppose we name the project that draws the square of Figure 6-8 *GlslSquare*. The files needed are almost the same as those of the project *GlslRaw* except that we replace the *Triangle* class by the *Square* class. The following is the code of the *Square* class but the methods **getShaderCode**() and **loadShader**(), and some data members are not listed as they are identical to that of the *Triangle* class discussed above.

```
public class Square
{
  .....
  private FloatBuffer vertexBuffer;
  private ShortBuffer indexArray;

  // number of coordinates per vertex in this array
  static final int COORDS_PER_VERTEX = 3;

  static float squareCoords[] = {
         -0.5f, -0.5f, 0.0f, // v0 - bottom left
          0.5f, -0.5f, 0.0f, // v1 - bottom right
         -0.5f,  0.5f, 0.0f, // v2 - top left
          0.5f,  0.5f, 0.0f  // v3 - top right
  };

  // draw in the order v0, v1, v2, v2, v1, v3
  private short drawOrder[] = { 0, 1, 2, 2, 1, 3 };
  float color[] = { 0.9f, 0.1f, 0.9f, 1.0f };

  // Constructor
  Square( Context context0){
    context = context0;

    // get shader codes from res/raw/vshader and res/raw/fshader
    vsCode = getShaderCode( GLES20.GL_VERTEX_SHADER );
    fsCode = getShaderCode( GLES20.GL_FRAGMENT_SHADER );
    program = GLES20.glCreateProgram();

    vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vsCode );
    fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fsCode );
```

```
      GLES20.glAttachShader ( program, vertexShader );
      GLES20.glAttachShader(program, fragmentShader);
      GLES20.glLinkProgram(program);
      GLES20.glUseProgram( program);

      // initialize vertex byte buffer for shape coordinates
      ByteBuffer bb = ByteBuffer.allocateDirect(
                          squareCoords.length * 4);
      bb.order(ByteOrder.nativeOrder());
      vertexBuffer = bb.asFloatBuffer();
      vertexBuffer.put(squareCoords);
      vertexBuffer.position(0);
      // initialize byte buffer for the draw list
      ByteBuffer bbOrder = ByteBuffer.allocateDirect(
                             drawOrder.length * 2);
      bbOrder.order(ByteOrder.nativeOrder());
      indexArray = bbOrder.asShortBuffer();
      indexArray.put(drawOrder);
      indexArray.position(0);
    } // Square Constructor

    public void draw () {
      // Add program to OpenGL ES environment
      GLES20.glUseProgram(program);

      int positionHandle = GLES20.glGetAttribLocation(program, "vPosition");
      GLES20.glEnableVertexAttribArray( positionHandle);

      int vertexStride = 0;
      GLES20.glVertexAttribPointer( positionHandle, COORDS_PER_VERTEX,
                      GLES20.GL_FLOAT, false, vertexStride, vertexBuffer);

      int colorHandle = GLES20.glGetUniformLocation(program, "vColor");
      GLES20.glUniform4fv(colorHandle, 1, color, 0);
      // Draw the square
      GLES20.glDrawElements( GLES20.GL_TRIANGLES, drawOrder.length,
                          GLES20.GL_UNSIGNED_SHORT, indexArray );

      // Disable vertex array
      GLES20.glDisableVertexAttribArray(positionHandle);
    }
    .....
  }
```

In the code, *indexArray* contains the vertex indices specified by *drawOrder*, the *length* of which gives the total number of indices and is equal to 6 in the example, representing the two triangles that form the square. The method **glDrawElements**() draws these two triangles, connecting the vertices in the order given by *indexArray*.

Besides replacing the *Triangle* class by the *Square* class in *MyRenderer*, we adjust the display aspect ratio as we want to display a square on the rectangular Android screen. Also we want to display the square at the upper part of the screen. So we set the viewport in the method **onSurfaceChanged**:

```
  public void onSurfaceChanged(GL10 unused, int width, int height) {
    float ratio = (float) width / height;
    GLES20.glViewport(0,  height/3, width,  (int) (height * ratio) );
  }
```

The vertex shader and the fragment shader are the same as those for displaying a triangle:

```
// vshader: Source code of vertex shader
attribute vec4 vPosition;
void main() {
   gl_Position = vPosition;
}

// fshader: Source code of fragment shader
precision mediump float;
uniform vec4 vColor;
void main() {
    gl_FragColor = vColor;
}
```

Figure 6-9 below shows the output of this project.

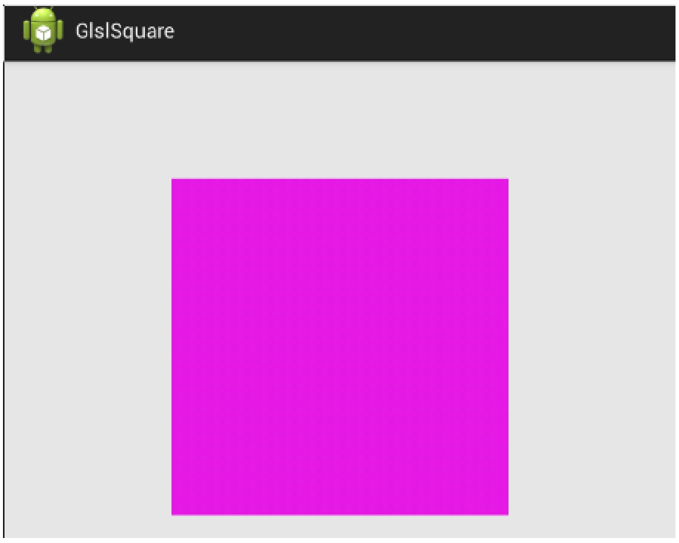


Figure 6-9 Output of Project *GlslSquare*

6.3.5 Drawing a Color Square

In the above example, we draw a square with a single color by passing the color through a **uniform** variable to the fragment shader. Suppose now we want to draw the square with a unique color at each vertex of the square. We cannot pass the colors directly to the fragment shader any more because we need to associate a color with a vertex. To accomplish this, we can pass a color value as a vertex attribute to the vertex shader, which then passes the color value to the color shader through a **varying** variable. The following are the shader codes:

```
// vshader
precision mediump float;
attribute vec4 vPosition;  // value from application program
attribute vec4 sourceColor;
varying vec4 vColor;       // value to be sent to fragment shader
```

```
void main(void)
{
  vColor = sourceColor;
  gl_Position = vPosition;
}

// fshader
varying vec4 vColor;

void main() {
  gl_FragColor = vColor;
}
```

We just need to make minor changes to the class *Square*. All we need to do is to define an array called *colors* to specify the RGBA color at each vertex of the square. We then treat the array *colors* in the way we do to the vertex coordinates array *vertexArray*. Each color is passed to the **varying** variable *sourceColor* of the vertex shader. The following is the modified portion of the code of *Square*:

```
public class Square
{
 .....
 private FloatBuffer vertexBuffer, colorBuffer;
 private ShortBuffer indexArray;

 static float squareCoords[] = {
   -0.5f, -0.5f, 0.0f,  // v0 - bottom left
    0.5f, -0.5f, 0.0f,  // v1 - bottom right
   -0.5f, 0.5f,  0.0f,  // v2 - top left
    0.5f, 0.5f,  0.0f   // v3 - top right
 };

 static float colors[] = {
    1.0f, 0.0f, 0.0f, 1.0f,   // v0 red
    0.0f, 1.0f, 0.0f, 1.0f,   // v1 green
    0.0f, 0.0f, 1.0f, 1.0f,   // v2 blue
    1.0f, 1.0f, 0.0f, 1.0f    // v3 yellow
 };
 private short drawOrder[] = { 0, 1, 2, 2, 1, 3 };

 // Constructor
 Square( Context context0){
   .....
   // initialize vertex byte buffer for shape coordinates
   ByteBuffer bb = ByteBuffer.allocateDirect(squareCoords.length * 4);
   bb.order(ByteOrder.nativeOrder());
   // do the same for colors
   ByteBuffer bbc = ByteBuffer.allocateDirect( colors.length * 4 );
   bbc.order(ByteOrder.nativeOrder());
   // create a floating point buffer from the ByteBuffer
   vertexBuffer = bb.asFloatBuffer();
   // add the coordinates to the FloatBuffer
   vertexBuffer.put(squareCoords);
   // set the buffer to read the first coordinate
```

```
    vertexBuffer.position(0);

    // do the same for colors
    colorBuffer = bbc.asFloatBuffer();
    colorBuffer.put( colors );
    colorBuffer.position(0);

    // initialize byte buffer for the draw list
    ByteBuffer bbOrder = ByteBuffer.allocateDirect(drawOrder.length * 2);
    bbOrder.order(ByteOrder.nativeOrder());
    indexArray = bbOrder.asShortBuffer();
    indexArray.put(drawOrder);
    indexArray.position(0);
  } // Square Constructor

  public void draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(program);

    // get handle to vertex shader's vPosition and sourceColor
    int positionHandle=GLES20.glGetAttribLocation(program,"vPosition");
    int colorHandle=GLES20.glGetAttribLocation(program,"sourceColor");

    GLES20.glEnableVertexAttribArray( positionHandle );
    GLES20.glEnableVertexAttribArray( colorHandle );
    GLES20.glVertexAttribPointer( positionHandle, 3,
         GLES20.GL_FLOAT, false, 0, vertexBuffer);
    GLES20.glVertexAttribPointer ( colorHandle, 4,
          GLES20.GL_FLOAT, false, 0, colorBuffer);
     // Draw the square
     GLES20.glDrawElements( GLES20.GL_TRIANGLES, drawOrder.length,
                      GLES20.GL_UNSIGNED_SHORT, indexArray );

     // Disable vertex arrays
     GLES20.glDisableVertexAttribArray( positionHandle );
     GLES20.glDisableVertexAttribArray( colorHandle );
  }
  .....
}
```

When we run the program, we will see a color square like the one shown in Figure 6-10 below.

6.3.6 Temperature Shaders

As an application to the above example, color square, we use colors to represent temperatures with red meaning hot and blue meaning cold. A warm temperature is a mixture of red and blue. We can imagine that the square is a metallic sheet with each corner connected to a heat or a cooling source. We can express smoothly the surface temperature as a mixture of red and blue. In the example, we assume that the lowest temperature is 0 and the highest is 50.

To accomplish this we pass the temperature value at each square vertex via the **attribute** array variable *vertexTemp* to the vertex shader. The shader normalizes it to a value between 0 and 1 before passing the value to the fragment shader via the **varying** variable *temperature*.

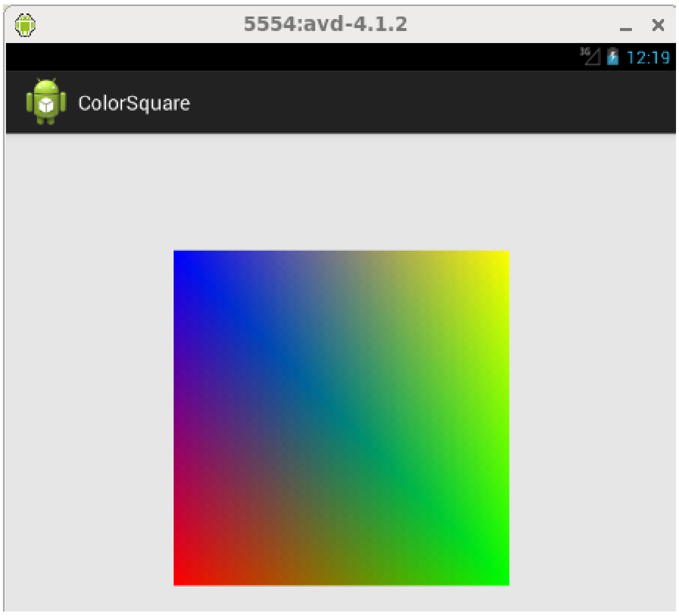


Figure 6-10 Drawing a Color Square

The following is the vertex shader code:

```
// vshader
precision mediump float;
attribute vec4 vPosition;   // value from application program
attribute float vertexTemp; // from application program
varying float temperature;  // value to be sent to fragment shader

void main(void)
{
  // normalize temperature to  [0, 1]
  temperature = ( vertexTemp - 0 ) / 50;
  gl_Position = vPosition;
}
```

The application also passes the blue color that represents the coldest temperature and the red color that represents the hottest temperature to the fragment shader via the **uniform** variables *coldColor* and *hotColor* respectively. Knowing these two colors, the fragment shader gets the normalized temperature value from the vertex shader and calculates a color for it with use of the glsl built-in funciton **mix**:

```
// fshader
uniform vec3 coldColor;
uniform vec3 hotColor;
varying float temperature;  // from vshader, value in [0,1]
void main() {
  vec3 color = mix ( coldColor, hotColor, temperature );
  gl_FragColor = vec4 ( color, 1 );
}
```

The application just needs to define the data for those quantities and gets handles to the **attribute** and **uniform** variables of the shaders to pass the data to them. The following is the modified portion of the *Square* class that does the job:

```
public class Square
{
  .....
  private FloatBuffer vertexBuffer, vertexTempBuffer;
  private ShortBuffer indexArray;

  static float squareCoords[] = {
    -0.5f, -0.5f, 0.0f,  // v0 - bottom left
     0.5f, -0.5f, 0.0f,  // v1 - bottom right
    -0.5f, 0.5f,  0.0f,  // v2 - top left
     0.5f, 0.5f,  0.0f   // v3 - top right
  };
  // Temperature at each vertex
  static float vertexTemp[] = {
     5.0f,             // v0 cold
    12.0f,             // v1 cool
    22.0f,             // v2 warm
    40.0f              // v3 hot (upper right)
  };

   private short drawOrder[] = { 0, 1, 2, 2, 1, 3 };

   // Constructor
  Square( Context context0){
    .....
    // initialize vertex byte buffer for square coordinates
    ByteBuffer bb = ByteBuffer.allocateDirect(squareCoords.length * 4);
    bb.order(ByteOrder.nativeOrder());
    // do the for vertexTemp
    ByteBuffer bbc = ByteBuffer.allocateDirect( vertexTemp.length * 4 );
    bbc.order(ByteOrder.nativeOrder());

    vertexBuffer = bb.asFloatBuffer();
    vertexBuffer.put(squareCoords);
    vertexBuffer.position(0);

    vertexTempBuffer = bbc.asFloatBuffer();
    vertexTempBuffer.put( vertexTemp );
    vertexTempBuffer.position(0);

    // initialize byte buffer for the draw list
    ByteBuffer bbOrder = ByteBuffer.allocateDirect(drawOrder.length * 2);
    bbOrder.order(ByteOrder.nativeOrder());
    indexArray = bbOrder.asShortBuffer();
    indexArray.put(drawOrder);
    indexArray.position(0);

  } // Square Constructor

  public void draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(program);

    // get handle to vertex shader’s vPosition, vertexTemp, ....
    int positionHandle = GLES20.glGetAttribLocation(program,"vPosition");
    int vertexTempHandle=GLES20.glGetAttribLocation(program,"vertexTemp");
    int coldColorHandle =GLES20.glGetUniformLocation(program,"coldColor");
    int  hotColorHandle = GLES20.glGetUniformLocation(program,"hotColor");
    GLES20.glUniform3f (coldColorHandle, 0.0f, 0.0f, 1.0f);// blue = cold
    GLES20.glUniform3f (hotColorHandle, 1.0f, 0.0f, 0.0f); // red = hot

    GLES20.glEnableVertexAttribArray( positionHandle );
    GLES20.glEnableVertexAttribArray( vertexTempHandle );
    GLES20.glVertexAttribPointer( positionHandle, 3,
         GLES20.GL_FLOAT, false, 0, vertexBuffer);
    // pass temperature at each vertex to vertex shader
    GLES20.glVertexAttribPointer ( vertexTempHandle, 1,
          GLES20.GL_FLOAT, false, 0, vertexTempBuffer);
     // Draw the square
     GLES20.glDrawElements( GLES20.GL_TRIANGLES, drawOrder.length,
                GLES20.GL_UNSIGNED_SHORT, indexArray );
     // Disable vertex arrays
     GLES20.glDisableVertexAttribArray( positionHandle );
     GLES20.glDisableVertexAttribArray( vertexTempHandle );
  }
  .....
 }
```

Figure 6-11 below shows the output of this example.

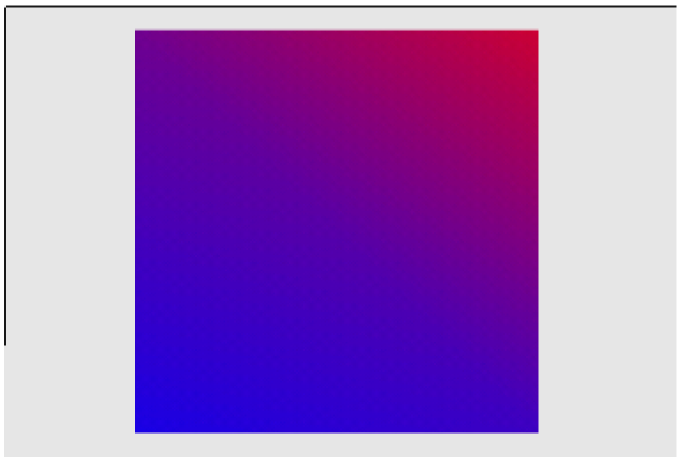


Figure 6-11 Output of Temperature Shaders

6.4 Drawing 3D Objects

6.4.1 Introduction

So far the rendering objects we have considered are two dimensional. If we want to render 3D objects, we need to define a 3D viewing volume, which is referred to as a frustum. Figure 6-12 below shows a typical setup of such a view-frustum, which is bounded by a near plane at $z = -n$, and a far plane at $z = -f$. In the figure, the point O is the origin of the coordinate system, and is the observation point (i.e. location of the camera or eye). The observer looks along the negative $z-$axis. So the near plane is at a distance of n from O and the far plane is f from O. The notations l, r, b, and t denote left, right, bottom and top respectively. So the left boundary of the near plane is at $x = l$, and the right boundary at $x = r$. The bottom and top boundaries of the near plane are at $y = b$ and $y = t$ respectively. Ultimately, all 3D objects inside the frustum are projected onto the near plane, which is viewed by an observer at point O.

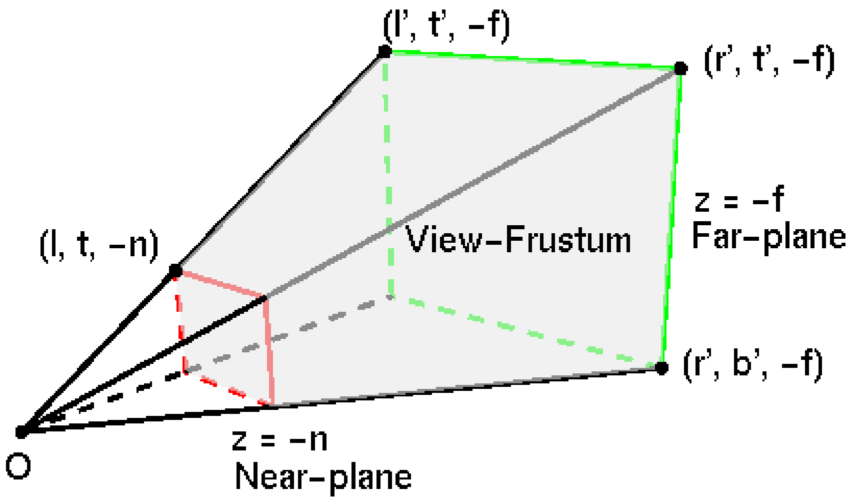


Figure 6-12 Frustum-shaped Viewing Volume

In 3D graphics, both a point and an Euclidean vector can be represented by a 3-tuple (x, y, z). Though they have the same representation and appear to be the same, a point and a vector are different elements. A point represents a location; it does not have any direction or magnitude. On the other hand, a vector indicates a direction but does not specify any location; a vector also has a magnitude (length). For convenience, people use homogeneous coordinates to represent both a vector and a point by introducing the fourth coordinate w:

$$A = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{6.1}$$

In (6.1), A is a vector when $w = 0$, and is a point when $w = 1$. This is consistent with our common intuition about points and vectors. When we add a vector to another vector, we get a new vector as the sum of the w components, which are 0, is 0. When we add a point ($w = 1$) to a vector ($w = 0$), we get a point as the sum of the w components, 0 and 1, is 1. In summary we have,

```
vector + vector   = vector
vector - vector   = vector
vector + point    = point
vector - point    = invalid
point  - vector   = point
point  - point    = vector
point  + point    = invalid
constant x vector = vector
```

We can form a linear combination of points:

$$A = c_1P_1 + c_2P_2 + ... + c_nP_n \tag{6.2}$$

where P_i denotes a point and c_i is a coefficient constant. Whether the result A is a valid point or not depends on the sum of the coefficients

$$S = c_1 + c_2 + ... + c_n \tag{6.3}$$

If $S = 0$, A is a vector. If $S = 1$, A is a valid point and (6.2) is referred to as an affine combination of points. Otherwise A is invalid. Conversely, an Euclidean vector can always be expressed as a linear combination of points.

As a point is represented by a 4×1 matrix as shown in Equation (6.1), any transformation operation on it, such as a rotation, a translation or a scaling, can represented by a 4×4 matrix. For example, if M is a 4×4 matrix, and P and Q are points, the equation $Q = MP$ represents that the point P is transformed to the point Q under the transformation matrix M. We may also write this explicitly as

$$Q = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = MP = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{6.4}$$

Transformations such as translations, rotations, and scalings that change the view of a 3D object are referred to as model-view transformations. These transformation are reversible. For example, we rotate a point about the $z-$axis by 30^o to get to a new location. We can go back to the original position by rotating the new point about the $z-$axis in the opposite direction by 30^o.

At the end, a 3D point is projected on to the near-plane screen, which is two dimensional; this is called projection transformation. In principle, such a transformation is not reversible as going from 3D to 2D will lose information in the process. However, to make a transformation matrix of projection compatible with that of model-view, people have expressed the projection matrix in a way that it is reversible (i.e., its inverse exists). The trick is to store the depth component (the z value) in a separate buffer and handle it separately. The z component recovered by the inverse of the project matrix will be discarded. So in OpenGL, all transformation matrices are 4×4.

In OpenGL ES 2.0, we need to pass the 4×4 matrix M to the vertex shader to perform the 3D transformation. In declaring a variable, we usually use *mv* to refer to **model-view** and *mvp* to refer to **model-view projection**.

6.4.2 Drawing a Tetrahedron

A tetrahedron is composed of four triangular faces, three of which meet at each vertex and thus it has four vertices. It may be the simplest kind of 3D objects.

A tetrahedron can be considered as a pyramid, which is a polyhedron with a flat polygon base and triangular faces connecting the base to a common point. A tetrahedron simply has a triangular base, so a tetrahedron is also known as a **triangular pyramid**.

A **regular tetrahedron** is one in which all four faces are equilateral triangles. The vertices coordinates of a regular tetrahedron with edge length 2 centered at the origin are

$$(1, 0, \frac{-1}{\sqrt{2}}), \ (-1, 0, \frac{-1}{\sqrt{2}}), \ (0, 1, \frac{1}{\sqrt{2}}), \ (0, -1, \frac{1}{\sqrt{2}}) \tag{6.5}$$

We illustrate many basic techniques of drawing 3D objects with OpenGL ES 2.0 through an example of drawing a regular tetrahedron. One main feature is to pass the resulted 4×4 model-view projection matrix to the vertex shader. The matrix transforms a point accordingly and projects it onto a 2D plane.

Suppose we call our project and application *Tetrahedron*. The structure of our project is the same as those discussed above, where the shaders are saved in the files *res/raw/vshader*, and *res/raw/fshader*. Just like before, we have three classes, *MainActivity*, *MyRenderer*, and *Tetrahderon*. The class *MainActivity* is the same as what we have used above .

MyRenderer

The main change to the class *MyRenderer* is that we need to define our viewing environment and caculates the model-view projection transformation matrix, which will be passed to the vertex shader for displaying the vertices of the 3D object properly.

We calculate the data for a projection transformation in the **onSurfaceChanged**() method. This is done using the method **frustumM** of the class *Matrix*, which stores 4×4 matrices in column-major order. The method has prototype,

```
public static void frustumM (float[] m, int offset, float left,
       float right, float bottom, float top, float near, float far)
```

The method defines a viewing frustum as shown in Figure 6-12 above. The second parameter, *offset*, is the offset into float array *m* where the projection matrix data are written. The parameters, *left, right, bottom*, and *top* define the boundaries of the near-plane, *near* and *far* are the distances from the observation point (O in Figure 6-12) to the near-plane and the far-plane respectively. Based on these parameters, OpenGL ES calculates a 4×4 matrix and saves it in the array *m*, the first parameter of the method.

The following is what we use in our *tetrahedron* example,

```
Matrix.frustumM( projectionMatrix, 0, -1, 1, -1, 1, 2, 10);
```

Projecting a 3D object on a plane is like taking a photo of the object. How the 3D object appears on the projection screen depends on the way the camera views the 3D object. We can define the camera view transformation using the **setLookAtM**() method of the class *Matrix*. The method calculates a 4×4 matrix for a specified setup. It has the following prototype:

```
public static void setLookAtM (float[] m, int offset, float eyeX,
             float eyeY, float eyeZ, float centerX, float centerY,
             float centerZ, float upX, float upY, float upZ)
```

The point $(eyeX, eyeY, eyeZ)$ specifies the location of the observation point (eye), and the point (*centerX, centerY, centerZ*) is the view center, the point at which the viewer is looking at. The vector (*upX, upY, upZ*) denotes the up direction of the camera. The method calculates the 4×4 viewing matrix and writes it to the array *m*.

After we have obtained the projection matrix and the viewing matrix, we can multiply them together to form a single matrix, which can be passed to the vertex shader to draw the object. The multiplication can be done by the *Matrix* method **multiplyMM**:

```
public static void multiplyMM (float[] result, int resultOffset,
          float[] lhs, int lhsOffset, float[] rhs, int rhsOffset)
```

The array *lhs* holds the 4×4 left-hand side matrix while *rhs* holds the right-hand side matrix. The product of these two matrices is saved in the array *result*. The offset parameters are offsets into the arrays where data are read or written. The following is the code for the class *MyRenderer* of our example:

```
public class MyRenderer implements GLSurfaceView.Renderer
{
 private final float[] mvpMatrix = new float[16];
 private final float[] projectionMatrix = new float[16];
 private final float[] viewMatrix = new float[16];
 private Tetrahedron tetrahedron;
 private Context context;

 public MyRenderer( Context context0 ) {
    context = context0;
 }
 public void onSurfaceCreated(GL10 unused, EGLConfig config) {
   // Set the background frame color
   GLES20.glClearColor(0.9f, 0.9f, 0.9f, 1.0f);
   // construct a tetrahedron object
   tetrahedron = new Tetrahedron ( context );
 }
 public void onDrawFrame ( GL10 unused ) {
   // Redraw background color
   GLES20.glClear( GLES20.GL_COLOR_BUFFER_BIT );
   // Set the camera position (View matrix)
   Matrix.setLookAtM(viewMatrix,0,0.5f,0,4,0f,0f,0f,0f,1.0f,0.0f);

   // Calculate the product of projection and view transformation
   Matrix.multiplyMM(mvpMatrix,0,projectionMatrix,0,viewMatrix, 0);
   //Draw tetrahedron with resulted model-view projection matrix
   tetrahedron.draw( mvpMatrix );
 }

 public void onSurfaceChanged(GL10 unused, int width, int height) {
    GLES20.glViewport(0,  0, width,  width  );
    Matrix.frustumM( projectionMatrix, 0, -1, 1, -1, 1, 2, 10);
 }
}
```

In the code, the **setLookAtM**() method, has set the viewing point to $(0.5, 0, 4)$ and the view center to $(0, 0, 0)$. The up-vector is $(0, 1, 0)$. This means that the observer is at the x-z plane, looking at the origin mostly along the negative z direction. The y axis is pointing in the up direction. The final model-view projection matrix is held in the array variable *mvpMatrix*, which is passed to the **draw** method of the *Tetrahedron* class, which in turn passes it to the vertex shader.

Tetrahedron

In our project, the *Tetrahedron* class is similar to the *Square* class we discussed in previous sections. However, for simplicity and clarity, we draw the tetrahedron using line strips rather than triangles. The following is a portion of the code of *Tetrahedron* modified from *Square*:

```
public class Tetrahedron
{
  .....
  // vertices coordinates of tetrahedron
  static float tetraCoords[] = {
     1, 0, -0.707f,  -1, 0, -0.707f,
     0, 1,  0.707f,   0, -1, 0.707f
  };
  // Order of indices of drawing the tetrahedron
  private short drawOrder[] = { 0, 1, 2, 0, 3, 1, 2, 3 };
  float color[] = { 0.9f, 0.1f, 0.9f, 1.0f };

  // Constructor
  Tetrahedron( Context context0){
    context = context0;
    // get shader codes from res/raw/vshader and res/raw/fshader
    vsCode = getShaderCode( GLES20.GL_VERTEX_SHADER );
    fsCode = getShaderCode( GLES20.GL_FRAGMENT_SHADER );
    .....
    ByteBuffer bb = ByteBuffer.allocateDirect(tetraCoords.length * 4);
    bb.order(ByteOrder.nativeOrder());
    vertexBuffer = bb.asFloatBuffer();
    vertexBuffer.put(tetraCoords);
    vertexBuffer.position(0);
    // initialize byte buffer for the draw list
    //   with # of coordinate values * 2 bytes per short
    ByteBuffer bbDrawOrder=ByteBuffer.allocateDirect(drawOrder.length*2);
    bbDrawOrder.order(ByteOrder.nativeOrder());
    indexArray = bbDrawOrder.asShortBuffer();
    indexArray.put(drawOrder);
    indexArray.position(0);
  } // Tetrahedron Constructor

  public void draw( float[] mvpMatrix ) {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(program);
    // get handle to shape's transformation matrix i shader
    int mvpMatrixHandle =
                GLES20.glGetUniformLocation( program, "mvpMatrix");
    // Pass model-view projection transformation matrix to the shader
    GLES20.glUniformMatrix4fv(mvpMatrixHandle,1,false,mvpMatrix,0);
    // get handle to vertex shader's vPosition member
    int positionHandle = GLES20.glGetAttribLocation(program,"vPosition");
    // Enable a handle to the triangle vertices
    GLES20.glEnableVertexAttribArray( positionHandle);

    // Prepare the triangles coordinate data
    int vertexStride = 0;
    GLES20.glVertexAttribPointer( positionHandle, COORDS_PER_VERTEX,
            GLES20.GL_FLOAT,false,vertexStride,vertexBuffer);

    // get handle to fragment shader's vColor member
    int colorHandle = GLES20.glGetUniformLocation(program, "vColor");
    // Set color for drawing the triangle
    GLES20.glUniform4fv(colorHandle, 1, color, 0);
    // Draw the tetrahedron using lines
    GLES20.glLineWidth(5);
    GLES20.glDrawElements( GLES20.GL_LINE_STRIP, drawOrder.length,
                          GLES20.GL_UNSIGNED_SHORT, indexArray );
     // Disable vertex array
     GLES20.glDisableVertexAttribArray(positionHandle);
  }
  .....
}
```

Shaders

The fragment shader is the same as before. It simply set the fragment color to the vertex color pass from the application:

```
   uniform vec4 vColor;
```

```
void main() {
    gl_FragColor = vColor;
}
```

The vertex shader is almost as simple except that now it has to multiply the vertex positions with the model-view projection matrix passed from the application:

```
// Source code of vertex shader
uniform mat4 mvpMatrix;
attribute vec4 vPosition;
void main() {
   gl_Position = mvpMatrix * vPosition;
}
```

Figure 6-13 shows the program's output, which is a tetrahedron drawn with line strips.

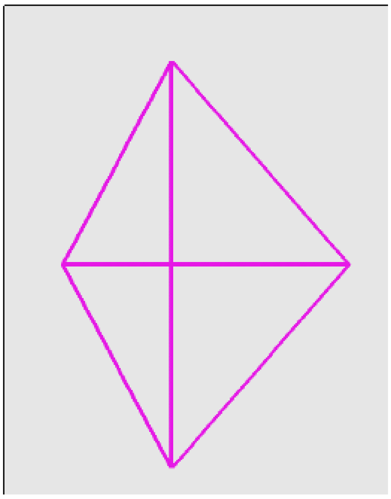


Figure 6-13 Rendering a Tetrahedron

6.4.3 Rotating a Color Tetrahedron

So far the tetrahedron we have drawn is a wireframe, without any actual face. Here, we discuss drawing a solid tetrahedron, each face having a different color. Moreover, we will rotate the tetrahedron by dragging the mouse.

Supose we call this project and application *Tetrahedron1*. We use the same three java files, *MainActivity.java*, *MyRenderer.java* and *Tetrahedron.java* that we have used in the previous sections but with some modifications.

MyRenderer

To rotate a 3D object, one can use the **setRotateM** or **rotateM** method of the class *Matrix*, which rotates the object for a given angle around a specified axis. We use **rotateM** to rotate the tetrahedron, which is performed in the method **onDrawFrame** of the class *MyRenderer*:

```
Matrix.setLookAtM(viewMatrix,0,0.5f,0.5f,5,0f,0f,0f,0f,1f,0.0f);
Matrix.rotateM (rvMatrix, 0, viewMatrix, 0, angle, 1f, 0.2f, 0.2f);
```

Here, the **setLooAtM** method sets the viewing matrix *viewMatrix* as before with the observation point at $(0.5, 0.5, 5)$, looking towards the origin and the up direction is $(0, 1, 0)$, which is along the y-axis. The second and third parameters are offset indices of the result matrix (*rvMatrix*) and the source matrix (*viewMatrix*); both offsets are 0 in the example. The **rotateM** method calculates the matrix that rotates an object around the axis $(1, 0.2, 0.2)$ for an angle specified by the variable *angle*. (Note that the axis in the example has a dominant x component. The rotation is almost like one rotating around the x-axis.) This matrix is multiplied to *viewMatrix* and the result is saved in *rvMatrix*, which is our model-view matrix. This matrix is multiplied to the projection matrix to form the model-view projection matrix *mvpMatrix*, which is eventually used by the vertex shader to render the vertices of the object:

```
Matrix.multiplyMM(mvpMatrix, 0, projectionMatrix, 0, rvMatrix, 0);
tetrahedron.draw( mvpMatrix );
```

To draw a solid tetrahedron, we need to enable the depth-test and cull the back faces, which are facing the interior of the object. In our example, we set front facging to counter-clockwise. The following is the code for the modified *MyRenderer* that does the face culling and rotation:

```
public class MyRenderer implements GLSurfaceView.Renderer
{
 .....
 // view matrix with rotation
 private final float[] rvMatrix = new float[16];
 private float angle = 0;   // angle of rotation
 .....
 public void onSurfaceCreated(GL10 unused, EGLConfig config) {
   // Set the background frame color
   GLES20.glClearColor(0.9f, 0.9f, 0.9f, 1.0f);
   // Cull back faces
   GLES20.glEnable(GLES20.GL_CULL_FACE);
   GLES20.glCullFace(GLES20.GL_BACK);
   // Set front-facing to be counter-clockwise
   GLES20.glFrontFace(GLES20.GL_CCW);

   GLES20.glEnable ( GLES20.GL_DEPTH_TEST );
   //larger z values are nearer to viewpoint
   GLES20.glDepthFunc(GLES20.GL_GREATER);
   // Construct a Tetrahedron object
   tetrahedron = new Tetrahedron ( context );
 }

 public void onDrawFrame ( GL10 unused ) {
   // Redraw background color, clear depth buffer
   GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT|GLES20.GL_DEPTH_BUFFER_BIT);
   // Set the camera position (View matrix)
   Matrix.setLookAtM(viewMatrix,0,0.5f,0.5f,5,0f,0f,0f,0f,1f,0.0f);
   // Multiply view matrix by rotation matrix, result in rvMatrix
   Matrix.rotateM (rvMatrix,0,viewMatrix,0,angle,1f,0.2f,0.2f);
   // Calculate the projection and view transformation
   Matrix.multiplyMM(mvpMatrix,0,projectionMatrix, 0,rvMatrix,0);
   // Draw the object with the transformation matrix
   tetrahedron.draw( mvpMatrix );
 }

 // Get and set angle of rotation
 public float getAngle() {
     return angle;
 }
```

```
  public void setAngle(float angle0) {
      angle = angle0;
  }
}
```

Tetrahedron

The class *Tetrahedron* is similar to that described in the previous section. It is responsible for reading, compiling, loading and running the shaders. Like before, we just need one array to store the coordinates of the four vertices of a tetrahedron; the vertices are shared by the four faces (triangles) of the tetrahedron. We can use a set of draw order to specify one face. For example, the set of indices $(0, 2, 3)$ means to draw the triangle $v_0v_2v_3$; these vertices must be arranged in a counter-clockwise direction when we look at the face from the outside of the object. As there are four faces, we need a total of four draw order lists. Moreover, we need four different colors, one for each face, and we have chosen the colors to be red, green, blue and yellow. We draw the face one at a time, using a different color and draw-order list. Each time we draw a face, the specified color is passed to the fragment shader uniform variable *vColor*. The following is the modified portion of the code of *Tetrahedron*..

```
public class Tetrahedron
{
  .....
  private FloatBuffer vertexBuffer;
  private FloatBuffer colorBuffer[];
  private ShortBuffer indexArray[];
  // number of faces in object
  static final int N_FACES = 4;
  // coordinates of tetrahedron vertices
  static float tetraCoords[] = {
    1, 0, -0.707f,  // vertex v0
   -1, 0, -0.707f,  // v1
    0, 1,  0.707f,  // v2
    0, -1, 0.707f   // v3
  };

  // draw order of  each face
  private short drawOrders[][] = {
     {0, 1, 2},  {0, 2, 3},  {0, 3, 1},  {3, 2, 1}
  };

  // color for each face
  static float colors[][] = {
    {1.0f, 0.0f, 0.0f, 1.0f},   // v0,v1,v2 red
    {0.0f, 1.0f, 0.0f, 1.0f},   // 0, 2, 3 green
    {0.0f, 0.0f, 1.0f, 1.0f},   // 0, 3, 1 blue
    {1.0f, 1.0f, 0.0f, 1.0f}    // 3, 2, 1 yellow
  };

  // Constructor
  Tetrahedron( Context context0){
    .....
    // alloocate memory to store tetrahedron vertices
    ByteBuffer bb = ByteBuffer.allocateDirect(tetraCoords.length * 4);
    bb.order(ByteOrder.nativeOrder());
    vertexBuffer = bb.asFloatBuffer();
    vertexBuffer.put(tetraCoords);
    vertexBuffer.position(0);
    // do the same for colors
    colorBuffer = new FloatBuffer[N_FACES]; // a color for each face
    indexArray = new ShortBuffer[N_FACES];  // N_FACES triangles
    for ( int i = 0; i < N_FACES; i++) {
      ByteBuffer bbc = ByteBuffer.allocateDirect(colors[i].length*4);
      bbc.order(ByteOrder.nativeOrder());
      colorBuffer[i] = bbc.asFloatBuffer();
      colorBuffer[i].put( colors[i] );
      colorBuffer[i].position(0);

      // initialize byte buffer for each face, 2 bytes per short
      ByteBuffer bbDrawOrder =
              ByteBuffer.allocateDirect( drawOrders[i].length * 2);
      bbDrawOrder.order(ByteOrder.nativeOrder());
      indexArray[i] = bbDrawOrder.asShortBuffer();
      indexArray[i].put(drawOrders[i]);
      indexArray[i].position(0);
    }
  } // Tetrahedron Constructor

  public void draw( float[] mvpMatrix ) {
    // get handle to shape's transformation matrix
    int mvpMatrixHandle=GLES20.glGetUniformLocation(program,"mvpMatrix");
    // Pass the projection and view transformation to the shader
    GLES20.glUniformMatrix4fv(mvpMatrixHandle,1,false,mvpMatrix,0);
    // Draw one face at a time
    for ( int i = 0; i < N_FACES; i++){
      // get handles to shaders' vPosition and  vColor member
      int positionHandle=GLES20.glGetAttribLocation(program,"vPosition");
      int colorHandle = GLES20.glGetUniformLocation(program, "vColor");
      // Enable a handle to the triangle vertices
      GLES20.glEnableVertexAttribArray( positionHandle);
      //  GLES20.glEnableVertexAttribArray( colorHandle );

      // Prepare the triangle coordinate and color data
      int vertexStride = 0;
      GLES20.glUniform4fv(colorHandle, 1, colors[i], 0);
      GLES20.glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                  GLES20.GL_FLOAT, false, vertexStride, vertexBuffer);
      GLES20.glDrawElements(GLES20.GL_TRIANGLES,drawOrders[i].length,
                           GLES20.GL_UNSIGNED_SHORT, indexArray[i] );
      GLES20.glDisableVertexAttribArray(positionHandle);
    }
  }
  .....
}
```

MyGLSurfaceView

Like before, this class, *MyGLSurfaceView*, is part of the program file *MainActivity.java*. It is responsible for responding to touch events.

In order to make an OpenGL ES application respond to touch events, we have to implement the **onTouchEvent**() method in the *GLSurfaceView* class. We follow the example presented in the Android developers web site to do our implementation, which listens for **Motion-Event.ACTION_MOVE** events and translates them to an angle of rotation for an object. The following is the code of this class that handles touch events:

```
class MyGLSurfaceView extends GLSurfaceView
{
  private final MyRenderer renderer;

  public MyGLSurfaceView(Context context){
    super(context);
    // Create an OpenGL ES 2.0 context
    setEGLContextClientVersion ( 2 );

    // Set the Renderer for drawing on the GLSurfaceView
    renderer = new MyRenderer ( context );
    setRenderer( renderer );
    // Render the view only when there is a change in the drawing data
    setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
  }

  private final float TOUCH_SCALE_FACTOR = 180.0f / 320;
  private float previousX, previousY;

  @Override
  public boolean onTouchEvent(MotionEvent e) {
    // MotionEvent reports input details from the touch screen
    // and other input controls. Here we are only interested
    // in events where the touch position has changed.
    float x = e.getX();
    float y = e.getY();
    switch (e.getAction()) {
      case MotionEvent.ACTION_MOVE:
        float dx = x - previousX;
        float dy = y - previousY;

        // reverse direction of rotation above the mid-line
        if (y > getHeight() / 2)
          dx = dx * -1 ;
        // reverse direction of rotation to left of the mid-line
        if (x < getWidth() / 2)
          dy = dy * -1 ;

        renderer.setAngle( renderer.getAngle() +
            ((dx + dy ) * TOUCH_SCALE_FACTOR));  // = 180.0f / 320
        requestRender();
    }
    previousX = x;
    previousY = y;
    return true;
  }
}
```

Note that in the code above, after calculating the rotation angle, it calls **requestRender**() to inform the renderer to render the frame. Such an approach is the most efficient in this example because we do not need to redraw a frame unless the rotation angle has been changed. However, this will take effect only if we also request that the renderer redraws only when the data changes. This can be done by setting the render mode to RENDERMODE_WHEN_DIRTY using the **setRenderMode**() method as this class does in its constructor.

Shaders

There is not much change in the shaders. They are just as simple as before:

```
// Source code of vertex shader
uniform mat4 mvpMatrix;
attribute vec4 vPosition;

void main() {
   gl_Position = mvpMatrix * vPosition;
}

// Source code of fragment shader
precision mediump float;
uniform vec4 vColor;
void main() {
    gl_FragColor = vColor;
}
```

When we run the program, we should see a solid color tetrahedron. We can rotate it by dragging the mouse on the screen. Figure 6-14 below shows some sample outputs of this example.

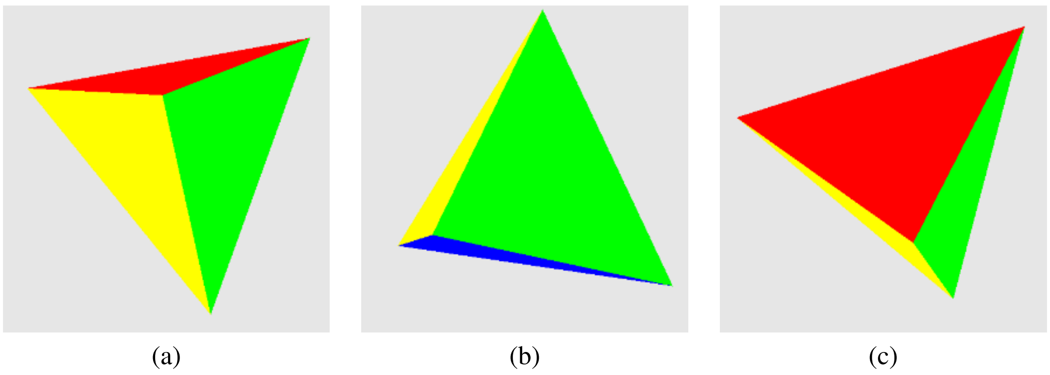


Figure 6-14 Sample Outputs of The Color Tetrahedron Project

6.5 Drawing Spheres

6.5.1 Spherical Coordinates

We can use a mesh of triangles to approximate a spherical surface. We have to define the vertices coordinates of each triangle and every vertex is on the surface of the sphere. In practice, it is easier to calculate the position of a point on a sphere using spherical coordinates, where a point is specified by three numbers: the radial distance r of that point from a fixed origin, its polar angle θ (also called inclination) measured from a fixed zenith direction, and the azimuth angle ϕ of its orthogonal projection on a reference plane that passes through the origin as shown in Figure 6-15.

So in spherical coordinates, a point is defined by (r, θ, ϕ) with some restrictions:

$$\begin{aligned} & r \geq 0 \\ & 0^o \leq \theta \leq 180^o \\ & 0^o \leq \phi < 360^o \end{aligned} \tag{6.6}$$

Cartesian coordinates of a point (x, y, z) can be calculated from the spherical coordinates, (radius r, inclination θ, azimuth ϕ), where $r \in [0, \infty)$, $\theta \in [0, \pi]$, $\phi \in [0, 2\pi)$, by:

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta \end{aligned} \tag{6.7}$$

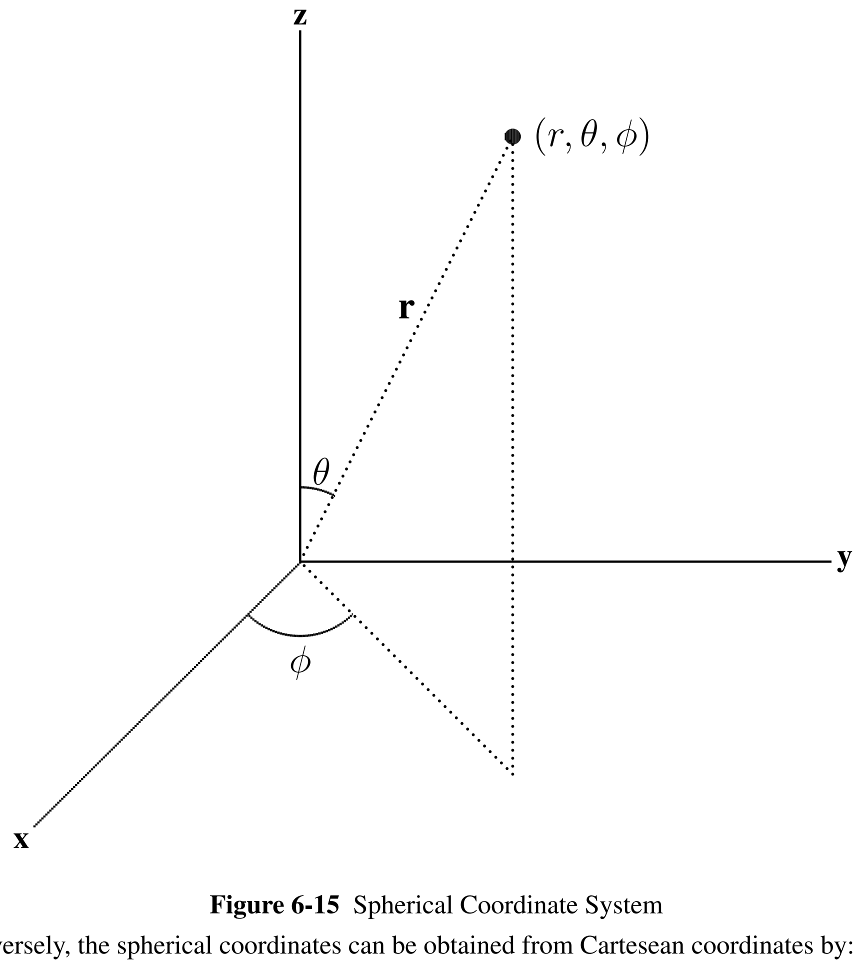


Figure 6-15 Spherical Coordinate System

Conversely, the spherical coordinates can be obtained from Cartesean coordinates by:

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \theta &= \cos^{-1}\left(\frac{z}{r}\right) \\ \phi &= \tan^{-1}\left(\frac{y}{x}\right) \end{aligned} \tag{6.8}$$

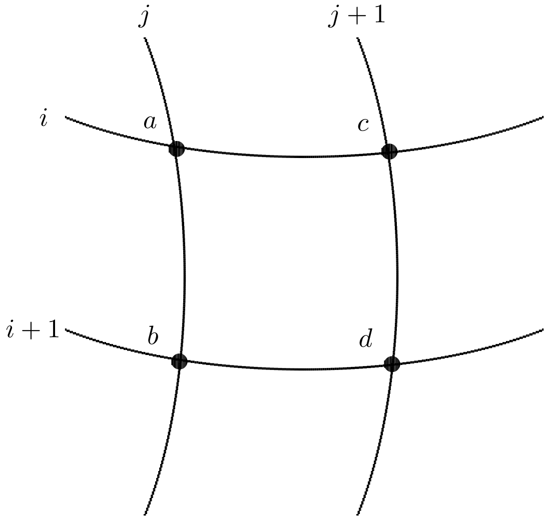
6.5.2 Rendering a Wireframe Sphere

To render a sphere centered at the origin, we can divide the sphere into slices around the z-axis (similar to lines of longitude), and stacks along the z-axis (similar to lines of latitude). We simply draw the slices and stacks independently, which will form a sphere. Each slice or stack is formed by line segments joining points together. Conversely, each point is an intersection of a slice and a stack.

Suppose we want to divide the sphere into m stacks and n slices. Since $0 \leq \theta \leq \pi$, the angle between two stacks is $\pi/(m-1)$. On the other hand, $0 \leq \phi < 2\pi$, the angle between two slices is $2\pi/n$ as the angle 2π is not included. That is,

$$\begin{aligned} \delta\theta &= \frac{\pi}{m-1} \\ \delta\phi &= \frac{2\pi}{n} \end{aligned} \tag{6.9}$$

Figure 6-16 below shows a portion of two slices and two stacks, and their intersection points.



m stacks, n slices

Indices
$a : j \times m + i$
$b : j \times m + i + 1$
$c : (j+1) \times m + i$
$d : (j+1) \times m + i + 1$

Quad $abdc = \triangle abc + \triangle cbd$

Figure 6 - 16. Spherical Surface Formed by Stacks and Slices

Our task is to calculate the intersection points. Suppose we calculate the points along a slice starting from $\phi = 0$, spanning θ from 0 to 2π, and then incrementing ϕ to calculate the next slice. We apply equation (6.7) to calculate the x, y, and z coordinates of each point. For convenience, we define a class called *XYZ* that contains the x, y, z coordinates of a point, and save the points in an *ArrayList* called *vertices*. The following code shows an implementation of such a task, assuming that r is the radius of the sphere:

```
ArrayList<XYZ> vertices = new ArrayList<XYZ>();
final double PI = 3.1415926;
final double TWOPI = 2 * PI;
XYZ p = new XYZ();   // a point
double phi,  theta;

for ( int j = 0; j < n; j++ ) {    // n slices
  phi = j * TWOPI / n;
  for ( int i = 0; i < m; i++ ) { // m stacks
    theta = i * PI / (m-1);       //0 to pi
    p.x = r * (float) (Math.sin ( theta ) * Math.cos ( phi ));
    p.y = r * (float) (Math.sin ( theta ) * Math.sin ( phi ));
    p.z = r * (float) Math.cos ( theta );
    vertices.add ( new XYZ ( p ) );
```

```
   }
 }
```

We can save these coordinates in a **float** array and put them in a byte buffer like what we did in previous examples:

```
  int nVertices = vertices.size();
  float sphereCoords[] = new float[3*nVertices];
  k = 0;
  for ( int i = 0; i < nVertices; i++ ) {
    XYZ v = vertices.get ( i );
    sphereCoords[k++] = v.x;
    sphereCoords[k++] = v.y;
    sphereCoords[k++] = v.z;
  }
  ByteBuffer bb=ByteBuffer.allocateDirect(sphereCoords.length * 4);
  bb.order(ByteOrder.nativeOrder());
  vertexBuffer = bb.asFloatBuffer();
  vertexBuffer.put(sphereCoords);
  vertexBuffer.position(0);
```

Now we have obtained all the intersection points. The remaining task is to define a draw order list that tells us how to connect the points. We use a **short** array, named *drawOrderw* to hold the indices of the vertices in the order we want to connect them. Suppose we first draw the slices. The following code shows how to calculate the indices for the points of the slices:

```
 int k = 0;
 for ( int j = 0; j < n; j++ ) {
  for ( int i = 0; i < m-1; i++ ) {
    drawOrderw[k++] = (short) (j * m + i);
    drawOrderw[k++] = (short)( j* m + i + 1 );
  }
 }
```

The two indices $(j * m + i)$ and $(j * m + i + 1)$ defines two points of a line segment of a slice. Each slice is composed of $m - 1$ line segments. The following code shows the calculations for the stacks:

```
 for ( int i = 1; i < m - 1; i++) {
   for ( int j = 0; j < n; j++){
     drawOrderw[k++] = (short) (j * m + i);
     if ( j == n - 1)  //wrap around: j + 1 --> 0
       drawOrderw[k++] = (short) ( i);
     else
       drawOrderw[k++] = (short) ((j+1)*m + i);
   }
 }
```

Each pair of indices define two end points of a line segment of a stack. When j equals $n - 1$, the next point wraps around so that the last point of the stack joins its first point to form a full circle. So each stack is composed of n segments. Also we do not need to draw the poles, and there are only $m - 2$ stacks. Therefore, the total number of indices in *drawOrderw* is

$$2 \times n \times (m - 1) + 2 \times (m - 2) \times n = 4 \times m \times n - 6 \times n$$

As before, We can put the indices in a byte array to draw slices and stacks, which will form a wireframe sphere:

```
ByteBuffer bbIndices = ByteBuffer.allocateDirect(
                              drawOrderw.length * 2);
bbIndices.order(ByteOrder.nativeOrder());
sphereIndices = bbIndices.asShortBuffer();
sphereIndices.put( drawOrderw );
sphereIndices.position(0);
GLES20.glDrawElements( GLES20.GL_LINES, drawOrderw.length,
                  GLES20.GL_UNSIGNED_SHORT, sphereIndices );
```

Since the first index in *drawOrderw* references the point that is the north pole of the sphere, we can draw the pole using the statement:

```
GLES20.glDrawElements( GLES20.GL_POINTS, 1,
                 GLES20.GL_UNSIGNED_SHORT, sphereIndices );
```

The shaders are similar to those described in previous examples except now we need to specify the point size using **gl_PointSize**:

```
// Source code of vertex shader
uniform mat4 mvpMatrix;
attribute vec4 vPosition;
void main() {
  gl_PointSize = 15;
  gl_Position = mvpMatrix * vPosition;
}
// Source code of fragment shader
precision mediump float;
uniform vec4 vColor;
void main() {
  gl_FragColor = vColor;
}
```

Suppose we set the number of slices to be 24 and the number of stacks to be 16. When we run the program, we will see a wireframe sphere like the one shown in Figure 6-17 below. The point near the bottom of the sphere is its north pole.

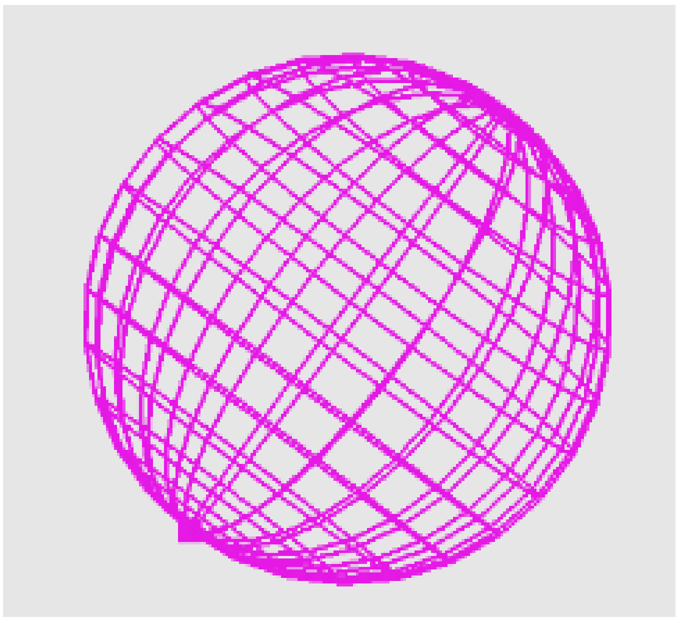


Figure 6-17 Rendering a Wireframe Sphere

6.5.3 Rendering a Color Solid Sphere

Rendering a color solid sphere is similar to rendering a color solid tetrahedron except that we have to calculate the vertices of the triangle mesh. We have already learned how to decompose a sphere into slices and stacks in the previous section. Suppose we have saved all the vertices in *vertexBuffer* as we did in the previous example. The shaders are also the same as those in the previous example, which are very simple.

As shown in Figure 6-16, the intersection points of two latitudes and two longitudes form a quadrilateral, which can be decomposed into two triangles. Since all the vertices coordinates have been calculated, we just need to find out the order of drawing them in the form of triangles.

As shown in the figure, to draw the quad abcd, we first draw the triangle abc and then draw the other triangle cbd, both in a counter-clockwise direction. That means the drawing order of the vertices is a, b, c, c, b, d. The following code shows the implementation of this procedure:

```
// n slices, m stacks
int nTriangles = 2 * n * (m - 1);    //number of triangles
short  drawOrders[][] = new short[nTriangles][3];
int k = 0;
for ( int j = 0; j < n; j++ ) {
  for ( int i = 0; i < m-1; i++ ) {
    short j1 = (short)(j + 1);
    if ( j == n - 1 ) j1 = 0;     //wrap around
    short ia = (short)( j * m + i ) ;
    short ib = (short)( j * m + i + 1);
    short ic = (short) (j1 * m + i );
    short id = (short)( j1 * m + i + 1 );
    drawOrders[k] = new short[3]; //first triangle
    drawOrders[k][0] = ia;
    drawOrders[k][1] = ib;
    drawOrders[k][2] = ic;
    k++;
    drawOrders[k] = new short[3]; //second triangle
    drawOrders[k][0] = ic;
    drawOrders[k][1] = ib;
    drawOrders[k][2] = id;
    k++;
  }
}
```

Suppose we just use the four colors, red, green, blue, and yellow to draw the whole sphere, alternating the colors between adjacent triangles. We can define a **float** array to hold the four colors:

```
static float colors[][] = {
  {1.0f, 0.0f, 0.0f, 1.0f},   // red
  {0.0f, 1.0f, 0.0f, 1.0f},   // green
  {0.0f, 0.0f, 1.0f, 1.0f},   // blue
  {1.0f, 1.0f, 0.0f, 1.0f}    // yellow
};
```

We need a **float** buffer to hold the color and a **short** buffer to hold the vertices of each triangle of the mesh. This can be implemented as follows:

```
// a color for each face
FloatBuffer colorBuffer[] = new FloatBuffer[nTriangles];
ShortBuffer sphereIndices[] = new ShortBuffer[nTriangles];
```

```
for ( int i = 0; i < nTriangles; i++) {
  int j = i % 4;
  ByteBuffer bbc=ByteBuffer.allocateDirect(colors[j].length*4);
  bbc.order(ByteOrder.nativeOrder());
  colorBuffer[i] = bbc.asFloatBuffer();
  colorBuffer[i].put( colors[j] );
  colorBuffer[i].position(0);
  ByteBuffer bbIndices = ByteBuffer.allocateDirect(
                                drawOrders[i].length * 2);
  bbIndices.order(ByteOrder.nativeOrder());
  sphereIndices[i] = bbIndices.asShortBuffer();
  sphereIndices[i].put( drawOrders[i] );
  sphereIndices[i].position(0);
}
```

To draw the sphere, we simply draw all the triangles, each of which is defined by three vertices and three colors:

```
for ( int i = 0; i < nTriangles; i++){
 int positionHandle=GLES20.glGetAttribLocation(program,"vPosition");
 int colorHandle = GLES20.glGetUniformLocation(program, "vColor");
 // Enable a handle to the triangle vertices
 GLES20.glEnableVertexAttribArray( positionHandle);
 int j = i % 4;   // only 4 colors
 GLES20.glUniform4fv(colorHandle, 1, colors[j], 0);
 GLES20.glVertexAttribPointer( positionHandle, COORDS_PER_VERTEX,
              GLES20.GL_FLOAT, false, vertexStride, vertexBuffer);
 GLES20.glDrawElements(GLES20.GL_TRIANGLES, drawOrders[i].length,
                     GLES20.GL_UNSIGNED_SHORT, sphereIndices[i]);
 // Disable vertex array
 GLES20.glDisableVertexAttribArray(positionHandle);
}
```

When we run the program, we will see an output similar to the one shown in Figure 16-17 below.

6.5.4 Lighting a Sphere

Lighting is an important feature in graphics for making a scene appear more realistic and more understandable. It provides crucial visual cues about the curvature and orientation of surfaces, and helps viewers perceive a graphics scene having three-dimensionality. Using the sphere we have constructed in previous sections, we discuss briefly here how to add lighting effect to it.

To create lighting effect that looks realistic, we need to first design a lighting model. In graphics, however, such a lighting model does not need to follow physical laws though the laws can be used as guidelines. The model is usually designed empirically. In our discussion, we more or less follow a simple and popular model called Phong lighting model to create lighting effect. In the model we only consider the effects of a light source shining directly on a surface and then being reflected directly to the viewpoint; second bounces are ignored. Such a model is referred to as a local lighting model, which only considers the light property and direction, the viewer's position, and the object material properties. It considers only the first bounce of the light ray but ignores any secondary reflections, which are light rays that are reflected for more than once by surfaces before reaching the viewpoint. Nor does a basic local model consider shadows created by light.

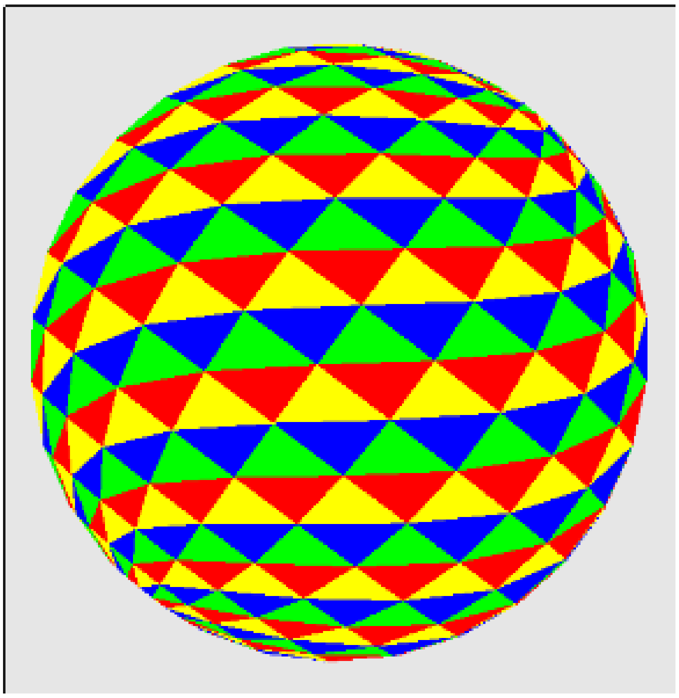


Figure 6-18 Rendering a Color Solid Sphere

In the model, we consider the following features:

1. All light sources are modeled as point light sources.
2. Light is composed of red (R), green (G), and blue (B) colors.
3. Light reflection intensities can be calculated independently using the principle of superposition for each light source and for each of the 3 color components (R, G, B). Therefore, we describe a source through a three-component intensity or illumination vector

$$\mathbf{I} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{6.10}$$

   Each of the components of **I** in (6.10) is the intensity of the independent red, green, and blue components.
4. There are three distinct kinds of light or illumination that contribute to the computation of the final illumination of an object:
   - **Ambient Light**: light that arrives equally from all directions. We use this to model the kind of light that has been scattered so much by its environment that we cannot tell its original source direction. Therefore, ambient light shines uniformly on a surface regardless of its orientation. The position of an ambient light source is meaningless.
   - **Diffuse Light**: light from a point source that will be reflected diffusely. We use this to model the kind of light that is reflected evenly in all directions away from the surface. (Of course, in reality this depends on the surface, not the light itself. As we mentioned earlier, this model is not based on real physics but on graphical experience.)
   - **Specular Light**: light from a point source that will be reflected specularly. We use this to model the kind of light that is reflected in a mirror-like fashion, the way that a light ray reflected from a shinny surface.
5. The model also assigns each surface material properties, which can be one of the four kinds:

- Materials with **ambient reflection properties** reflect ambient light.
- Materials with **diffuse reflection properties** reflect diffuse light.
- Materials with **specular reflection properties** reflect specular light.

In the model, ambient light only interacts with materials that possess ambient property; specular and diffuse light only interact with specular and diffuse materials respectively.

Figure 6-19 below shows the vectors that are needed to calculate the illumination at a point. In the figure, the labels *vPosition*, *lightPosition*, and *eyePosition* denote points at the vertex, the light source, and the viewing position respectively. The labels **L, N, R**, and **V** are vectors derived from these points (recall that the difference between two points is a vector), representing the light vector, the normal, the reflection vector, and the viewing vector respectively. The reflection vector **R** is the direction along which a light from **L** will be reflected if the the surface at the point is mirror-like. Assuming that the center of the sphere is at the origin $O = (0, 0, 0)$, some of them can be expressed as

$$\begin{aligned} \text{light vector } \mathbf{L} &= lightPosition - vPosition \\ \text{normal } \mathbf{N} &= vPosition - O \\ \text{view vector } \mathbf{V} &= eyePosition - vPosition \end{aligned} \tag{6.11}$$

We can normalize a vector by dividing it by its magnitude:

$$\mathbf{l} = \frac{\mathbf{L}}{|\mathbf{L}|}, \quad \mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|}, \quad \mathbf{v} = \frac{\mathbf{V}}{|\mathbf{V}|}, \quad \mathbf{r} = \frac{\mathbf{R}}{|\mathbf{R}|} \tag{6.12}$$

One can easily show that the normalized reflection vector **r** can be calculated from **l** and **n** by the formula,

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l} \tag{6.13}$$

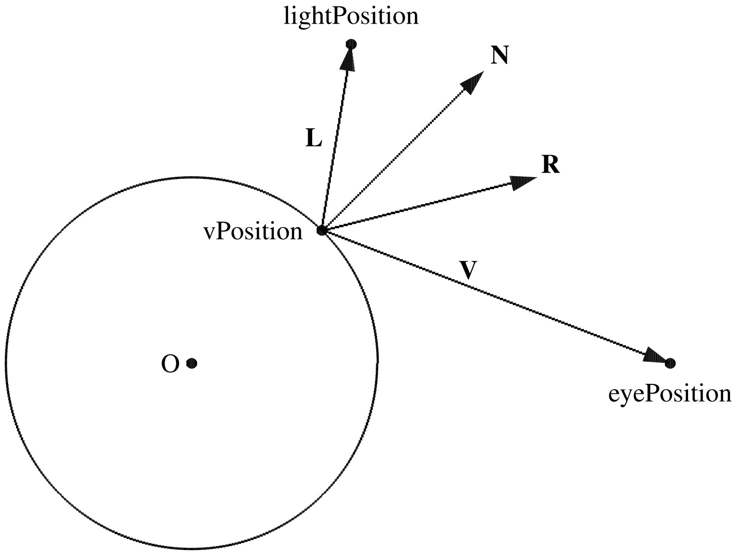


Figure 6-19 Lighting Vectors

Suppose I^{in} denotes the incident illumination from the light source in the direction **l**. The ambient, diffuse, and specular illumination on the point *vPosition* can be calculated according to the following formulas.

The ambient illumination is given by

$$I_a = c_a I_a^{in} \tag{6.14}$$

where I_a^{in} is the incident ambient light intensity and c_a is a constant called the *ambient reflectivity coefficient*.

The diffuse illumination is

$$I_d = c_d I_d^{in} \mathbf{l} \cdot \mathbf{n} \tag{6.15}$$

where $\mathbf{l} \cdot \mathbf{n} = \mathbf{r} \cdot \mathbf{n}$, I_d^{in} is the incident diffuse light intensity and c_d is a constant called the *diffuse reflectivity coefficient*.

The specular illumination can be calculated by

$$I_s \quad = c_s I_s^{in} (\mathbf{r} \cdot \mathbf{v})^f \tag{6.16}$$

where c_s is a constant called the *specular reflectivity coefficient* and the exponent f is a value that can be adjusted empirically on an ad hoc basis to achieve desired lighting effect. The exponent f is ≥ 0, and values in the range 50 to 100 are typically used for shinny surfaces. The larger the exponent factor f, the narrower the beam of specularly reflected light becomes.

The total illumination is the sum of all the above components:

$$\begin{aligned} I &= I_a + I_d + I_s \\ &= c_a I_a^{in} + c_d I_d^{in} (\mathbf{l} \cdot \mathbf{n}) + c_s I_s^{in} (\mathbf{r} \cdot \mathbf{v})^f \end{aligned} \tag{6.17}$$

This model can be easily implemented in the glsl shader language. In our example, where the illuminated object is a sphere, the shader code is further simplified. The positions of the light source, the vertex, and the eye (viewing point) are passed from the the application to the vertex shader as **uniform** variables. The vertex shader calculates the vectors **L**, **N**, and **V** from the positions and pass them to the fragment shader as **varying** variables:

```
   // Source code of vertex shader
   uniform mat4 mvpMatrix;
   attribute vec4 vPosition;
   uniform vec4 eyePosition;
   uniform vec4 lightPosition;
   varying vec3 N; //normal direction
   varying vec3 L; //light source direction
   varying vec3 V; //view vector
   void main() {
      gl_Position = mvpMatrix * vPosition;
      N = vPosition.xyz;   //normal of  a point on sphere
      L = lightPosition.xyz - vPosition.xyz;
      V = eyePosition.xyz - vPosition.xyz;
   }
```

The fragment shader obtains the vectors **L**, **N**, and **V** from the vertex shader, normalizes them, and calculates the reflection vector **r**. It then uses formulas (6.14) to (6.17) to calculate the illumination at the vertex:

```
// Source code of fragment shader
   precision mediump float;
   varying vec3 N;
   varying vec3 L;
   varying vec3 V;
   uniform vec4 lightAmbient;
   uniform vec4 lightDiffuse;
   uniform vec4 lightSpecular;
   //in this example, material color same for ambient, diffuse, specular
   uniform vec4 materialColor;
   uniform float shininess;
```

```
void main() {
  vec3 norm = normalize(N);
  vec3 lightv = normalize(L);
  vec3 viewv = normalize(V);
  // diffuse coefficient
  float Kd = max(0.0, dot(lightv, norm));

  // calculating specular coefficient
  // consider only specular light in same direction as normal
  float cs;
  if(dot(lightv, norm)>= 0.0) cs =1.0;
  else cs = 0.0;
  //reflection vector
  vec3 r = 2.0 *  dot (norm, lightv) * norm  - lightv;
  float Ks = pow(max(0.0, dot(r, viewv)), shininess);
  vec4 ambient = materialColor * lightAmbient;
  vec4 specular = cs * Ks * materialColor *lightSpecular;
  vec4 diffuse = Kd * materialColor *  lightDiffuse;

  gl_FragColor = ambient + diffuse + specular;
}
```

One can modify the code or juggle with it to obtain various lighting effects empirically.

Similar to previous examples, the OpenGL application has to provide the actual values of the **uniform** and **attribute** parameters. The sphere is constructed in the same way that we did in the previous example. However, we do not need to pass in the colors for each triangle as the appearance of the sphere is now determined by its material color and the light colors, and the color at each pixel is calculated by the fragment shader using the lighting model. The following code shows how the application supplies the lighting parameters:

```
public class Sphere
{
 .....
 float eyePos[] = {5f, 5f, 10f, 1f};        //viewing position
 float lightPos[] = {5f, 10f, 5f, 1f};      //light source position
 float lightAmbi[] = {0.1f, 0.1f, 0.1f, 1f};//ambient light
 float lightDiff[] = {1f, 0.8f, 0.6f, 1f};  //diffuse light
 float lightSpec[] = {0.3f, 0.2f, 0.1f,1f} ;//specular light
 //material same for ambient, diffuse, and specular
 float materialColor[] = {1f, 1f, 1f, 1f};
 float shininess = 50f;

 public void draw( float[] mvpMatrix ) {
  // Add program to OpenGL ES environment
  GLES20.glUseProgram(program);
  // get handle to shape's transformation matrix
  int mvpMatrixHandle=GLES20.glGetUniformLocation(program,"mvpMatrix");
  // Pass the projection and view transformation to the shader
  GLES20.glUniformMatrix4fv(mvpMatrixHandle, 1, false, mvpMatrix, 0);
  // Pass lighting parameters
  int eyePosHandle=GLES20.glGetUniformLocation(program, "eyePosition");
  int lightPosHandle=GLES20.glGetUniformLocation(program,"lightPosition");
  int lightAmbiHandle=GLES20.glGetUniformLocation(program,"lightAmbient");
  int lightDiffHandle=GLES20.glGetUniformLocation(program,"lightDiffuse");
  int lightSpecHandle=GLES20.glGetUniformLocation(program,"lightSpecular");
```

```
    int materialColorHandle=GLES20.glGetUniformLocation(program,
                                                          "materialColor");
    int shininessHandle=GLES20.glGetUniformLocation(program, "shininess");
    GLES20.glUniform4fv(eyePosHandle, 1, eyePos, 0);
    GLES20.glUniform4fv(lightPosHandle, 1, lightPos, 0);
    GLES20.glUniform4fv(lightAmbiHandle, 1, lightAmbi, 0);
    GLES20.glUniform4fv(lightDiffHandle, 1, lightDiff, 0);
    GLES20.glUniform4fv(lightSpecHandle, 1, lightSpec, 0);
    GLES20.glUniform1f(shininessHandle, shininess);
    GLES20.glUniform4fv(materialColorHandle, 1, materialColor, 0);
    int vertexStride = 0;
    // Draw the sphere
    GLES20.glLineWidth(3);
    for ( int i = 0; i < nTriangles; i++){
      int positionHandle=GLES20.glGetAttribLocation(program,"vPosition");
      // Enable a handle to the triangle vertices
      GLES20.glEnableVertexAttribArray( positionHandle);
      GLES20.glVertexAttribPointer( positionHandle, COORDS_PER_VERTEX,
                     GLES20.GL_FLOAT, false, vertexStride, vertexBuffer);
      GLES20.glDrawElements(GLES20.GL_TRIANGLES, drawOrders[i].length,
                              GLES20.GL_UNSIGNED_SHORT, sphereIndices[i]);
      // Disable vertex array
      GLES20.glDisableVertexAttribArray(positionHandle);
    }
  }
  .....
}
```

Figure 6-20 below shows an output of this application, where the same sphere of Figure 6-18 has been used.

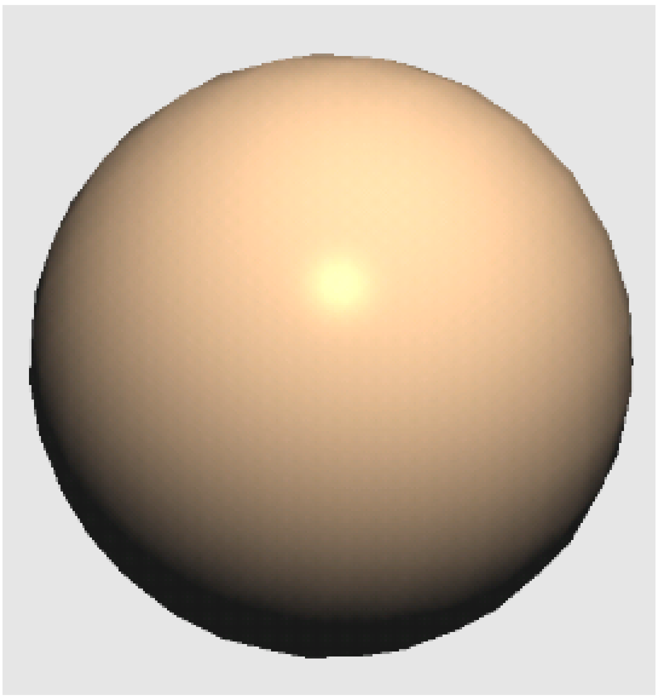


Figure 6-20 Example Rendered Lit Sphere