4.4 Rendering a 3D Color Cube

In these few sections, we extend the example of the previous section where we rotate a 2D triangle when we drag the mouse. Firstly we discuss rendering a color cube which can be rotated by dragging the mouse. Secondly we discuss putting textures on the cube. In the process, we will discuss other related but stand-alone graphics topics.

4.4.1 Color Cube

A cube consists of 8 vertices and 6 faces, each of which is a square. OpenGL ES 1.X does not support primitives of GL_QUAD or GL_POLYGON. It only renders triangles. So to render any polygon other than a triangle, we have to first decompose it into triangles.

Any polygon has two faces: a front face and a back face. Whether a face is front or back depends on our winding convention, the way we order the vertices of the polygon. In general we want to specify the winding in a way that the back faces of an object are those facing the interior of the object and front faces are those facing the exterior. The winding of a front face could be clockwise or counterclockwise and can be specified by the command `gl.glFrontFace()`. For example,

```gl
gl.glFrontFace ( GL10.GL_CCW );
```

specifies that a face is a front face if the vertices of the triangle is ordered in the counter-clockwise (CCW) direction, and it is a back face if the vertices are ordered in the clockwise (CW) direction. In our discussion, we always consider a face with counter-clockwise winding to be a front face.

In practice, if an object is opaque, we do not want to display any of its back faces as they are facing the interior of the object. We can suppress rendering back faces using the commands,

```gl
gl glEnable( GL10.GL_CULL_FACE );
gl.glCullFace ( GL10.GL_BACK );
```

Now consider a cube whose center is at the origin with length 2. The coordinates of the vertices of the face (square) at \( z = 1 \) are given by

\[
 v_4 = (-1, -1, 1), \quad v_5 = (1, -1, 1), \quad v_6 = (1, 1, 1), \quad v_7 = (-1, 1, 1)
\]

The front face of the square is specified by \( v_4v_5v_6v_7 \) which are CCW and its back face is specified by \( v_4v_7v_6v_5 \) which are CW when we observe the face from a point at \( z > 1 \). The square can be decomposed into two triangles as shown in Figure 4-3. When we make the decomposition, we must be careful that the windings of the triangles are consistent with that of the face considered.

![Figure 4-3. Decomposing a square into 2 triangles](image-url)
In Figure 4-3, the vertices of a polygon are always specified in counterclockwise order. Suppose we call this application and project *cube*. We follow the steps discussed in Section 4.2.1 to create an activity with *GLSurfaceView*. The file *MainActivity.java* of the previous is slightly modified to the code shown in Listing 4-1 below.

**Program Listing 4-1  MainActivity.java for Rendering Cube**

```java
package opengl.cube;

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

public class MainActivity extends Activity {
    private GLSurfaceView mGLView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mGLView = new CubeSurfaceView(this);
        setContentView(mGLView);
    }

    @Override
    protected void onPause() {
        super.onPause();
        // The following call pauses the rendering thread.
        mGLView.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        // The following call resumes a paused rendering thread.
        mGLView.onResume();
    }

    class CubeSurfaceView extends GLSurfaceView {
        private final float TOUCH_SCALE_FACTOR = 180.0f / 320;
        private CubeRenderer renderer;
        private float previousX;
        private float previousY;

        public CubeSurfaceView(Context context){
            super(context);
            // set the renderer member
            renderer = new CubeRenderer();
            setRenderer(renderer);
            // Render the view only when there is a change
            setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
        }

        @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.angle += (dx + dy) * TOUCH_SCALE_FACTOR;
        requestRender();
        previousX = x;
        previousY = y;
        return true;
    }

As we can see from Listing 4-1, the class CubeSurfaceView extends GLSurfaceView and creates
an object of the CubeRenderer class, which is discussed below, to render a color cube.

Listing 4-2 shows the complete code of the file CubeRenderer.java which contains the code of
the class CubeRenderer that implements GLSurfaceView.Renderer, and the code of the class Cube
that has the attributes of a cube with length 2 and center at the origin of the coordinate system.

The methods onSurfaceChanged and onDrawFrame are basically the equivalent of the GLUT
functions glutReshapeFunc and glutDisplayFunc. The former is called when there is a change
in surface size like the case when the phone switches between landscape and portrait modes. The
latter is called each time the cube is rendered.

The class Cube uses two FloatBuffer objects to store vertex and color data and a ByteBuffer to
store the face indices. You may also refer to the comments of the code which give more detailed
explanations of the processing of the data.

Program Listing 4-2  CubeRenderer.java

```java
package opengl.cube;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import android.opengl.GLSurfaceView;
import android.opengl.GLU;
import android.os.SystemClock;
import android.view.MotionEvent;
```
public class CubeRenderer implements GLSurfaceView.Renderer {
    public float angle = 0.0f;  //rotation angle
    private Cube cube = new Cube();

    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        // Set the background frame color to grey, opaque
        gl.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
        gl.glEnable(GL10.GL_CULL_FACE);  //Enable culling faces
        gl.glCullFace(GL10.GL_BACK);  //don’t render back faces
    }

    public void onDrawFrame(GL10 gl) {
        // Redraw background color
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        // Set GL_MODELVIEW transformation mode
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();  // Reset the matrix to identity matrix
        // Move objects away from view point to observe
        gl.glTranslatef(0.0f, 0.0f, -10.0f);
        // Rotate about a diagonal of cube
        gl.glRotatef(angle, 1.0f, 1.0f, 1.0f);
        cube.draw(gl);  // Draw the cube
        gl.glLoadIdentity();  // Reset transformation matrix
    }

    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        gl.glViewport(0, 0, width, height);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();  // Reset projection matrix
        // Setup viewing volume
        GLU.gluPerspective(gl, 45.0f, (float) width / (float) height, 0.1f, 100.0f);
        gl.glViewport(0, 0, width, height);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();  // Reset transformation matrix
    }
}

class Cube {
    private FloatBuffer vertexBuffer;
    private FloatBuffer colorBuffer;
    private ByteBuffer indexBuffer;

    // Coordinates of 8 vertices of 6 cube faces
    private float vertices[] = {
        -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
        1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f,
        -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f,
        1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f
    };

    // Colors of vertices
    private float colors[] = {
        0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f,
        1.0f, 0.5f, 0.0f, 1.0f, 1.0f, 0.5f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f
    };
}
contents

// indices of 12 triangles (6 squares) in GL_CCW
// referencing vertices[] array coordinates

private byte indices[] = {
  5, 4, 0, 1, 5, 0, 6, 5, 1, 2, 6, 1,
  7, 6, 2, 3, 7, 2, 4, 7, 3, 0, 4, 3,
  6, 7, 4, 5, 6, 4, 1, 0, 3, 2, 1, 3
};

public Cube() {
  // initialize vertex Buffer for cube
  // argument=(# of coordinate values * 4 bytes per float)
  ByteBuffer byteBuf = ByteBuffer.allocateDirect(vertices.length * 4);
  byteBuf.order(ByteOrder.nativeOrder());
  // create a floating point buffer from the ByteBuffer
  vertexBuffer = byteBuf.asFloatBuffer();
  // add the vertices coordinates to the FloatBuffer
  vertexBuffer.put(vertices);
  // set the buffer to read the first vertex coordinates
  vertexBuffer.position(0);

  // Do the same to colors array
  byteBuf=ByteBuffer.allocateDirect(colors.length*4);
  byteBuf.order(ByteOrder.nativeOrder());
  colorBuffer = byteBuf.asFloatBuffer();
  colorBuffer.put(colors);
  colorBuffer.position(0);
  // indices are integers
  indexBuffer = ByteBuffer.allocateDirect(indices.length);
  indexBuffer.put(indices);
  indexBuffer.position(0);
}

// Typical drawing routine using vertex array
public void draw(GL10 gl) {
  //Counterclockwise order for front face vertices
  gl.glFrontFace(GL10.GL_CCW);

  //Points to the vertex buffers
  gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
  gl.glColorPointer(4, GL10.GL_FLOAT, 0, colorBuffer);

  //Enable client states
  gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
  gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

  //Draw vertices as triangles
  gl.glDrawElements(GL10.GL_TRIANGLES, 36, GL10.GL_UNSIGNED_BYTE, indexBuffer);

  //Disable client state
  gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
  gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
}

In Listing 4-2 above, the draw method of class Cube is a typical OpenGL drawing routine using vertex array:

1. `gl.glVertexPointer` tells the renderer where to read the vertices coordinates and of what data type they are. The first parameter is the number of coordinates of a vertex and it is 3 here for the \((x, y, z)\) 3D coordinates. The second parameter tells that data type of each coordinate value is float. The third parameter, referred to as stride, is the offset between neighboring vertices in the array. A value of 0 indicates that array is tightly packed, not containing other data such as color values other than the vertex coordinates. The last parameter points to a buffer, `vertexBuffer` in our example, where the vertex coordinates are held.

2. `gl.glColorPointer` tells the renderer where to read the color data of the vertices. It works similarly to `gl.glVertexPointer`. The first parameter is 4 because a color tuple \((r, g, b, a)\) consists of four values representing red, green, blue and alpha (transparency).

3. `gl.glEnableClientState` enables OpenGL to use a vertex array for rendering.

4. `gl.glDrawArrays` tells OpenGL to draw the primitive. In our code, the first parameter, `GL10.GL_TRIANGLES` tells the renderer to draw the vertices held in the vertex buffer as triangles. The second parameter specifies the number (count) of vertices. In our example, we have 6 faces. Each face has 2 triangles and each triangle has 3 vertices. Therefore, the total number of vertices is

\[
\text{count} = 6 \times 2 \times 3 = 36
\]

The third parameter specifies the type of values in the array specified by the fourth parameter which is a pointer pointing to the location where the indices of vertex data are stored.

5. We may think of `glEnableClientState` and `glDisableClientState` as begin ... end statements in a program.

When we compile and run the app, we will see a color cube which may appear 2D as the viewing point is right in front of it. We will see its 3D shape when we drag the mouse which rotates it. Figure 4-4 below shows an output of the app.

![Color Cube](image)
4.4.2 Rendering a Square Only

If we want to render a square only, we can define a class `Square` similar to the class `Cube` presented in Listing 4-2, except that we use the primitive `GL_TRIANGLES` and method `gl.glDrawArrays` rather than `gl.glDrawElements` to render the two triangles. A triangle strip is a series of connected triangles, two in our case. In this method, we only have to define four vertices for a square. Using the square shown in Figure 4-3 as an example, OpenGL first draws the triangle using vertices in the order of \( v_4v_5v_7 \), then it takes the last vertex \( v_7 \) from the previous triangle and uses the last side \( v_7v_5 \) of it as the basis for the new triangle which will be drawn in the order \( v_7v_5v_6 \). Listing 4-3 shows the code of class `Square`.

**Program Listing 4-3  Class Square**

```java
class Square {
    private FloatBuffer vertexBuffer; //buffer holding the vertices
    private float vertices[] = { //Figure 4-3
        -1.0f, -1.0f, 1.0f, // v4 - bottom left
        1.0f, -1.0f, 1.0f, // v5 - bottom right
        -1.0f, 1.0f, 1.0f, // v6 - top left
        1.0f, 1.0f, 1.0f // v7 - top right  
    };

    public Square() {
        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(vertices.length * 4);
        byteBuffer.order(ByteOrder.nativeOrder());
        vertexBuffer = byteBuffer.asFloatBuffer();
        vertexBuffer.put(vertices);
        vertexBuffer.position(0);
    }

    public void draw(GL10 gl) {
        gl.glFrontFace(GL10.GL_CCW);
        gl.glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, vertices.length / 3);
        gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    }
}
```

The only other changes to render the square using the class `CubeRenderer` is to add the data member declaration statement

```
private Square square = new Square();
```

to the class and to replace `cube.draw()` by `square.draw()`. Upon running the code, we should see a white square over a grey background. Again we can drag the mouse to rotate the square.