Tutorials:

OpenGL ES 1.0

In this document

Create an Activity with GLSurfaceView
Draw a Shape on GLSurfaceView
Define a Triangle
Draw the Triangle
Apply Projection and Camera Views
Add Motion
Respond to Touch Events

Related Samples

API Demos - graphics
OpenGL ES 1.0 Sample
TouchRotateActivity

See also

3D with OpenGL
OpenGL ES 2.0

This tutorial shows you how to create a simple Android application that uses the OpenGL ES 1.0 API to perform some basic graphics operations. You'll learn how to:

- Create an activity using `GLSurfaceView` and `GLSurfaceView.Renderer`
- Create and draw a graphic object
- Define a projection to correct for screen geometry
- Define a camera view
- Rotate a graphic object
- Make graphics touch-interactive

The Android framework supports both the OpenGL ES 1.0/1.1 and OpenGL ES 2.0 APIs. You should carefully consider which version of the OpenGL ES API (1.0/1.1 or 2.0) is most appropriate for your needs. For more information, see Choosing an OpenGL API Version. If you would prefer to use OpenGL ES 2.0, see the OpenGL ES 2.0 tutorial.

Before you start, you should understand how to create a basic Android application. If you do not know how to create an app, follow the Hello World Tutorial to familiarize yourself with the process.

Create an Activity with GLSurfaceView

To get started using OpenGL, you must implement both a `GLSurfaceView` and a `GLSurfaceView.Renderer`. The `GLSurfaceView` is the main view type for applications that use OpenGL and the `GLSurfaceView.Renderer` controls what is drawn within that view. (For more information about these classes, see the 3D with OpenGL document.)

To create an activity using `GLSurfaceView`:

1. Start a new Android project that targets Android 1.6 (API Level 4) or higher.
2. Name the project **HelloOpenGLES10** and make sure it includes an activity called `HelloOpenGLES10`.
3. Modify the `HelloOpenGLES10` class as follows:

```
package com.example.android.apis.graphics;

import android.app.Activity;
import android.content.Context;
import android.opengl.GLSurfaceView;
```

```
import android.os.Bundle;

public class HelloOpenGLES10 extends Activity {

    private GLSurfaceView mGLView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Create a GLSurfaceView instance and set it
        // as the ContentView for this Activity.
        mGLView = new HelloOpenGLES10SurfaceView(this);
        setContentView(mGLView);
    }

    @Override
    protected void onPause() {
        super.onPause();
        // The following call pauses the rendering thread.
        // If your OpenGL application is memory intensive,
        // you should consider de-allocating objects that
        // consume significant memory here.
        mGLView.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        // The following call resumes a paused rendering thread.
        // If you de-allocated graphic objects for onPause()
        // this is a good place to re-allocate them.
        mGLView.onResume();
    }
}

class HelloOpenGLES10SurfaceView extends GLSurfaceView {

    public HelloOpenGLES10SurfaceView(Context context){
        super(context);

        // Set the Renderer for drawing on the GLSurfaceView
        setRenderer(new HelloOpenGLES10Renderer());
    }
}
```

> **Note:** You will get a compile error for the `HelloOpenGLES10Renderer` class reference. That's expected; you will fix this error in the next step.

As shown above, this activity uses a single `GLSurfaceView` for its view. Notice that this activity implements crucial lifecycle callbacks for pausing and resuming its work.

The `HelloOpenGLES10SurfaceView` class in this example code above is just a thin wrapper for an instance of `GLSurfaceView` and is not strictly necessary for this example. However, if you want your application to monitor and

respond to touch screen events—and we are guessing you do—you must extend `GLSurfaceView` to add touch event listeners, which you will learn how to do in the Reponding to Touch Events section.

In order to draw graphics in the `GLSurfaceView`, you must define an implementation of `GLSurfaceView.Renderer`. In the next step, you create a renderer class to complete this OpenGL application.

4. Create a new file for the following class `HelloOpenGLES10Renderer`, which implements the `GLSurfaceView.Renderer` interface:

```
package com.example.android.apis.graphics;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView;

public class HelloOpenGLES10Renderer implements GLSurfaceView.Renderer {

    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        // Set the background frame color
        gl.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    }

    public void onDrawFrame(GL10 gl) {
        // Redraw background color
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
    }

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

}
```

This minimal implementation of `GLSurfaceView.Renderer` provides the code structure needed to use OpenGL drawing methods:

- `onSurfaceCreated()` is called once to set up the `GLSurfaceView` environment.
- `onDrawFrame()` is called for each redraw of the `GLSurfaceView`.
- `onSurfaceChanged()` is called if the geometry of the `GLSurfaceView` changes, for example when the device's screen orientation changes.

For more information about these methods, see the 3D with OpenGL document.

The code example above creates a simple Android application that displays a grey screen using OpenGL ES 1.0 calls. While this application does not do anything very interesting, by creating these classes, you have layed the foundation needed to start drawing graphic elements with OpenGL ES 1.0.

If you are familiar with the OpenGL ES APIs, these classes should give you enough information to use the OpenGL ES 1.0 API and create graphics. However, if you need a bit more help getting started with OpenGL, head on to the next sections for a few more hints.

Draw a Shape on GLSurfaceView

Once you have implemented a GLSurfaceView.Renderer, the next step is to draw something with it. This section shows you how to define and draw a triangle.

Define a Triangle

OpenGL allows you to define objects using coordinates in three-dimensional space. So, before you can draw a triangle, you must define its coordinates. In OpenGL, the typical way to do this is to define a vertex array for the coordinates.

By default, OpenGL ES assumes a coordinate system where [0,0,0] (X,Y,Z) specifies the center of the GLSurfaceView frame, [1,1,0] is the top right corner of the frame and [-1,-1,0] is bottom left corner of the frame.

To define a vertex array for a triangle:

1. In your `HelloOpenGLES10Renderer` class, add new member variable to contain the vertices of a triangle shape:

```
    private FloatBuffer triangleVB;
```

2. Create a method, `initShapes()` which populates this member variable:

```
    private void initShapes(){

        float triangleCoords[] = {
            // X, Y, Z
            -0.5f, -0.25f, 0,
             0.5f, -0.25f, 0,
             0.0f,  0.559016994f, 0
        };

        // initialize vertex Buffer for triangle
        ByteBuffer vbb = ByteBuffer.allocateDirect(
                // (# of coordinate values * 4 bytes per float)
                triangleCoords.length * 4);
        vbb.order(ByteOrder.nativeOrder());// use the device hardware's native byte
order
        triangleVB = vbb.asFloatBuffer();  // create a floating point buffer from
the ByteBuffer
        triangleVB.put(triangleCoords);    // add the coordinates to the FloatBuffer
        triangleVB.position(0);            // set the buffer to read the first
coordinate

    }
```

This method defines a two-dimensional triangle with three equal sides.

3. Modify your `onSurfaceCreated()` method to initialize your triangle:

```
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {

        // Set the background frame color
        gl.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

        // initialize the triangle vertex array
        initShapes();
    }
```

> **Caution:** Shapes and other static objects should be initialized once in your `onSurfaceCreated()` method for best performance. Avoid initializing the new objects in `onDrawFrame()`, as this causes the system to re-create the objects for every frame redraw and slows down your application.

You have now defined a triangle shape, but if you run the application, nothing appears. What?! You also have to tell OpenGL to draw the triangle, which you'll do in the next section.

Draw the Triangle

Before you can draw your triangle, you must tell OpenGL that you are using vertex arrays. After that setup step, you can call the drawing APIs to display the triangle.

To draw the triangle:

1. Add the `glEnableClientState()` method to the end of `onSurfaceCreated()` to enable vertex arrays.

```
        // Enable use of vertex arrays
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
```

   At this point, you are ready to draw the triangle object in the OpenGL view.

2. Add the following code to the end of your `onDrawFrame()` method to draw the triangle.

```
        // Draw the triangle
        gl.glColor4f(0.63671875f, 0.76953125f, 0.22265625f, 0.0f);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, triangleVB);
        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
```

3. Run the app! Your application should look something like this:

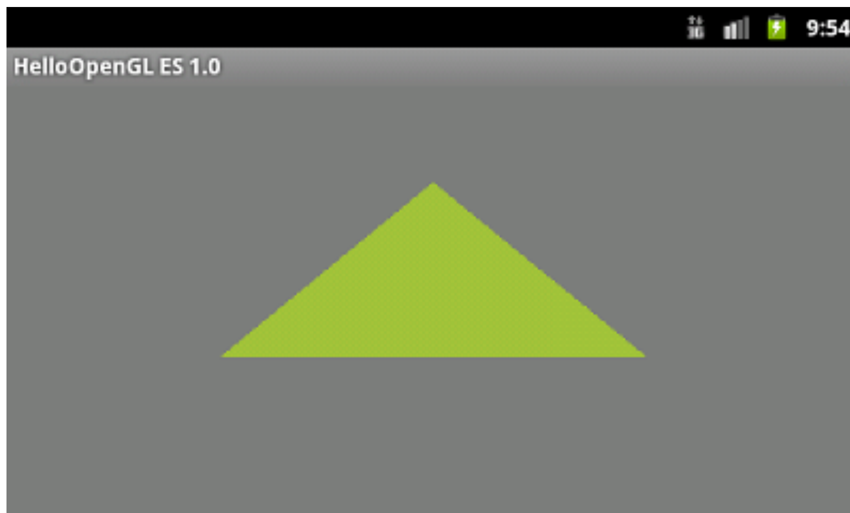

Figure 1. Triangle drawn without a projection or camera view.

There are a few problems with this example. First of all, it is not going to impress your friends. Secondly, the triangle is a bit squashed and changes shape when you change the screen orientation of the device. The reason the shape is skewed is due to the fact that the object is being rendered in a frame which is not perfectly square. You'll fix that problem using a projection and camera view in the next section.

Lastly, because the triangle is stationary, the system is redrawing the object repeatedly in exactly the same place, which is not the most efficient use of the OpenGL graphics pipeline. In the Add Motion section, you'll make this shape rotate and justify this use of processing power.

Apply Projection and Camera View

One of the basic problems in displaying graphics is that Android device displays are typically not square and, by default, OpenGL happily maps a perfectly square, uniform coordinate system onto your typically non-square screen. To solve this problem, you can apply an OpenGL projection mode and camera view (eye point) to transform the coordinates of your graphic objects so they have the correct proportions on any display. For more information about OpenGL coordinate mapping, see Mapping Coordinates for Drawn Objects.

To apply projection and camera view transformations to your triangle:

1. Modify your `onSurfaceChanged()` method to enable `GL10.GL_PROJECTION` mode, calculate the screen ratio and apply the ratio as a transformation of the object coordinates.

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

      // make adjustments for screen ratio
      float ratio = (float) width / height;
      gl.glMatrixMode(GL10.GL_PROJECTION);        // set matrix to projection mode
      gl.glLoadIdentity();                        // reset the matrix to its default
state
      gl.glFrustumf(-ratio, ratio, -1, 1, 3, 7);  // apply the projection matrix
  }
```

2. Next, modify your `onDrawFrame()` method to apply the `GL_MODELVIEW` mode and set a view point using `GLU.gluLookAt()`.

```
    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 its default state

        // When using GL_MODELVIEW, you must set the view point
        GLU.gluLookAt(gl, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

        // Draw the triangle
        ...
    }
```

3. Run the updated application and you should see something like this:

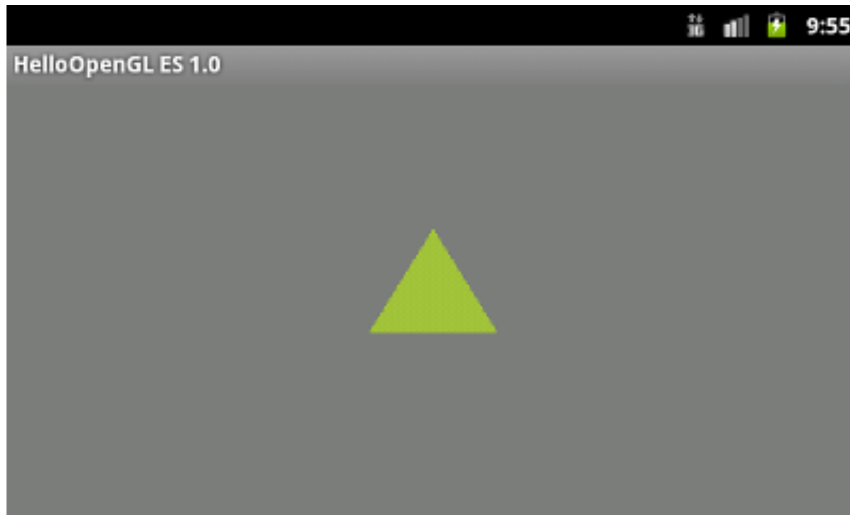

Figure 2. Triangle drawn with a projection and camera view applied.

Now that you have applied this transformation, the triangle has three equal sides, instead of the squashed triangle in the earlier version.

Add Motion

While it may be an interesting exercise to create static graphic objects with OpenGL ES, chances are you want at least *some* of your objects to move. In this section, you'll add motion to your triangle by rotating it.

To add rotation to your triangle:

1. Modify your `onDrawFrame()` method to rotate the triangle object:

```
public void onDrawFrame(GL10 gl) {
    ...
    // When using GL_MODELVIEW, you must set the view point
    GLU.gluLookAt(gl, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    // Create a rotation for the triangle
    long time = SystemClock.uptimeMillis() % 4000L;
    float angle = 0.090f * ((int) time);
    gl.glRotatef(angle, 0.0f, 0.0f, 1.0f);

    // Draw the triangle
    ...
}
```

2. Run the application and your triangle should rotate around its center.

Respond to Touch Events

Making objects move according to a preset program like the rotating triangle is useful for getting some attention, but what if

you want to have users interact with your OpenGL graphics? In this section, you'll learn how listen for touch events to let users interact with objects in your `HelloOpenGLES10SurfaceView`.

The key to making your OpenGL application touch interactive is expanding your implementation of `GLSurfaceView` to override the `onTouchEvent()` to listen for touch events. Before you do that, however, you'll modify the renderer class to expose the rotation angle of the triangle. Afterwards, you'll modify the `HelloOpenGLES10SurfaceView` to process touch events and pass that data to your renderer.

To make your triangle rotate in response to touch events:

1. Modify your `HelloOpenGLES10Renderer` class to include a new, public member so that your `HelloOpenGLES10SurfaceView` class is able to pass new rotation values your renderer:

```
    public float mAngle;
```

2. In your `onDrawFrame()` method, comment out the code that generates an angle and replace the `angle` variable with `mAngle`.

```
        // Create a rotation for the triangle (Boring! Comment this out:)
        // long time = SystemClock.uptimeMillis() % 4000L;
        // float angle = 0.090f * ((int) time);

        // Use the mAngle member as the rotation value
        gl.glRotatef(mAngle, 0.0f, 0.0f, 1.0f);
```

3. In your `HelloOpenGLES10SurfaceView` class, add the following member variables.

```
    private final float TOUCH_SCALE_FACTOR = 180.0f / 320;
    private HelloOpenGLES10Renderer mRenderer;
    private float mPreviousX;
    private float mPreviousY;
```

4. In the constructor method for `HelloOpenGLES10SurfaceView`, set the `mRenderer` member so you have a handle to pass in rotation input and set the render mode to `RENDERMODE_WHEN_DIRTY`.

```
    public HelloOpenGLES10SurfaceView(Context context){
        super(context);
        // set the mRenderer member
        mRenderer = new HelloOpenGLES10Renderer();
        setRenderer(mRenderer);

        // Render the view only when there is a change
        setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    }
```

5. In your `HelloOpenGLES10SurfaceView` class, override the `onTouchEvent()` method to listen for touch events and pass them to your renderer.

```
    @Override
    public boolean onTouchEvent(MotionEvent e) {
        // MotionEvent reports input details from the touch screen
        // and other input controls. In this case, you are only
        // interested in events where the touch position changed.
```

```
        float x = e.getX();
        float y = e.getY();

        switch (e.getAction()) {
            case MotionEvent.ACTION_MOVE:

                float dx = x - mPreviousX;
                float dy = y - mPreviousY;

                // 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 ;
                }

                mRenderer.mAngle += (dx + dy) * TOUCH_SCALE_FACTOR;
                requestRender();
        }

        mPreviousX = x;
        mPreviousY = y;
        return true;
    }
```

> **Note:** Touch events return pixel coordinates which *are not the same* as OpenGL coordinates. Touch coordinate [0,0] is the bottom-left of the screen and the highest value [max_X, max_Y] is the top-right corner of the screen. To match touch events to OpenGL graphic objects, you must translate touch coordinates into OpenGL coordinates.

6. Run the application and drag your finger or cursor around the screen to rotate the triangle.

For another example of OpenGL touch event functionality, see TouchRotateActivity.

← Back to Tutorials ↑ Go to top

Except as noted, this content is licensed under Creative Commons Attribution 2.5. For details and restrictions, see the Content License.
Privacy & Terms - Brand Guidelines - Report Document Issues