Build an OpenGL ES environment

In order to draw graphics with OpenGL ES in your Android application, you must create a view container for them. One of the ways to do this is to implement both a [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) and a [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer). A [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) is a view container for graphics drawn with OpenGL and [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer) controls what is drawn within that view.

This lesson explains how to complete a minimal implementation of [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) and [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer) in a simple application activity.

Declare OpenGL ES use in the manifest

In order for your application to use the OpenGL ES 2.0 API, you must add the following declaration to your manifest:

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

If your application uses texture compression, you must also declare which compression formats your app supports, so that it is only installed on compatible devices.

<supports-gl-texture android:name="GL\_OES\_compressed\_ETC1\_RGB8\_texture" />  
<supports-gl-texture android:name="GL\_OES\_compressed\_paletted\_texture" />

Create an activity for OpenGL ES graphics

Android applications that use OpenGL ES have activities just like any other application that has a user interface. The main difference from other applications is what you put in the layout for your activity. While in many applications you might use [TextView](https://developer.android.com/reference/android/widget/TextView), [Button](https://developer.android.com/reference/android/widget/Button) and [ListView](https://developer.android.com/reference/android/widget/ListView), in an app that uses OpenGL ES, you can also add a [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView).

public class OpenGLES20Activity extends Activity {  
  
    private GLSurfaceView gLView;  
  
    @Override  
    public 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);  
    }  
}

Build a GLSurfaceView object

A [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) is a specialized view where you can draw OpenGL ES graphics. It does not do much by itself. The actual drawing of objects is controlled in the [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer) that you set on this viewThe essential code for a [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) is minimal, so for a quick implementation, it is common to just create an inner class in the activity that uses it:

import android.content.Context;  
import android.opengl.GLSurfaceView;  
  
class MyGLSurfaceView extends GLSurfaceView {  
  
    private final MyGLRenderer renderer;  
  
    public MyGLSurfaceView(Context context){  
        super(context);  
  
        // Create an OpenGL ES 2.0 context  
        setEGLContextClientVersion(2);  
  
        renderer = new MyGLRenderer();  
  
        // Set the Renderer for drawing on the GLSurfaceView  
        setRenderer(renderer);  
    }  
}

One other optional addition to your [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) implementation is to set the render mode to only draw the view when there is a change to your drawing data using the [GLSurfaceView.RENDERMODE\_WHEN\_DIRTY](https://developer.android.com/reference/android/opengl/GLSurfaceView" \l "RENDERMODE_WHEN_DIRTY) setting:

// Render the view only when there is a change in the drawing data  
setRenderMode(GLSurfaceView.RENDERMODE\_WHEN\_DIRTY);

This setting prevents the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) frame from being redrawn until you call [requestRender()](https://developer.android.com/reference/android/opengl/GLSurfaceView" \l "requestRender()), which is more efficient for this sample app. Dopo ce lo fa togliere, non lo mettiamo proprio allora ahhaha

Build a renderer class

The implementation of the [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer) class, or renderer, within an application that uses OpenGL ES is where things start to get interesting. This class controls what gets drawn on the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) with which it is associated. There are three methods in a renderer that are called by the Android system in order to figure out what and how to draw on a [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView):

* [onSurfaceCreated()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer#onSurfaceCreated(javax.microedition.khronos.opengles.GL10,%20javax.microedition.khronos.egl.EGLConfig)) - Called once to set up the view's OpenGL ES environment.
* [onDrawFrame()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer#onDrawFrame(javax.microedition.khronos.opengles.GL10)) - Called for each redraw of the view.
* [onSurfaceChanged()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer#onSurfaceChanged(javax.microedition.khronos.opengles.GL10,%20int,%20int)) - Called if the geometry of the view changes, for example when the device's screen orientation changes.

Here is a very basic implementation of an OpenGL ES renderer, that does nothing more than draw a black background in the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView):

import javax.microedition.khronos.egl.EGLConfig;  
import javax.microedition.khronos.opengles.GL10;  
  
import android.opengl.GLES20;  
import android.opengl.GLSurfaceView;  
  
public class MyGLRenderer implements GLSurfaceView.Renderer {  
  
    public void onSurfaceCreated(GL10 unused, EGLConfig config) {  
        // Set the background frame color  
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  
    }  
  
    public void onDrawFrame(GL10 unused) {  
        // Redraw background color  
        GLES20.glClear(GLES20.GL\_COLOR\_BUFFER\_BIT);  
    }  
  
    public void onSurfaceChanged(GL10 unused, int width, int height) {  
        GLES20.glViewport(0, 0, width, height);  
    }  
}

The code examples above create a simple Android application that displays a black screen using OpenGL. Now you have laid the foundation you need to start drawing graphic elements with OpenGL.

# Define shapes

Being able to define shapes to be drawn in the context of an OpenGL ES view is the first step. Drawing with OpenGL ES can be a little tricky without knowing a few basic things about how OpenGL ES expects you to define graphic objects.

This lesson explains the OpenGL ES coordinate system relative to an Android device screen, the basics of defining a shape, shape faces, as well as defining a triangle and a square.

## Define a triangle

OpenGL ES allows you to define drawn 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 of floating point numbers for the coordinates. For maximum efficiency, you write these coordinates into a [ByteBuffer](https://developer.android.com/reference/java/nio/ByteBuffer), that is passed into the OpenGL ES graphics pipeline for processing.

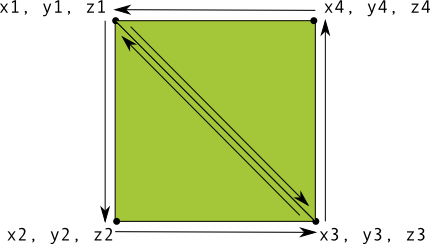
public class Triangle {  
  
    private FloatBuffer vertexBuffer;  
  
    // number of coordinates per vertex in this array  
    static final int COORDS\_PER\_VERTEX = 3;  
    static float triangleCoords[] = {   // in counterclockwise order:  
             0.0f,  0.622008459f, 0.0f, // top  
            -0.5f, -0.311004243f, 0.0f, // bottom left  
             0.5f, -0.311004243f, 0.0f  // bottom right  
    };  
  
    // Set color with red, green, blue and alpha (opacity) values  
    float color[] = { 0.63671875f, 0.76953125f, 0.22265625f, 1.0f };  
  
    public Triangle() {  
        // initialize vertex byte buffer for shape coordinates  
        ByteBuffer bb = ByteBuffer.allocateDirect(  
                // (number of coordinate values \* 4 bytes per float)  
                triangleCoords.length \* 4);  
        // use the device hardware's native byte order  
        bb.order(ByteOrder.nativeOrder());  
  
        // 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);  
    }  
}

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

Note that the coordinates of this shape are defined in a counterclockwise order. The drawing order is important because it defines which side is the front face of the shape, which you typically want to have drawn, and the back face, which you can choose to not draw using the OpenGL ES cull face feature.

## Define a square

Defining triangles is pretty easy in OpenGL, but what if you want to get a just a little more complex? Say, a square? There are a number of ways to do this, but a typical path to drawing such a shape in OpenGL ES is to use two triangles drawn together:



**Figure 1.** Drawing a square using two triangles.

Again, you should define the vertices in a counterclockwise order for both triangles that represent this shape, and put the values in a [ByteBuffer](https://developer.android.com/reference/java/nio/ByteBuffer). In order to avoid defining the two coordinates shared by each triangle twice, use a drawing list to tell the OpenGL ES graphics pipeline how to draw these vertices. Here’s the code for this shape:

public class Square {  
  
    private FloatBuffer vertexBuffer;  
    private ShortBuffer drawListBuffer;  
  
    // number of coordinates per vertex in this array  
    static final int COORDS\_PER\_VERTEX = 3;  
    static float squareCoords[] = {  
            -0.5f,  0.5f, 0.0f,   // top left  
            -0.5f, -0.5f, 0.0f,   // bottom left  
             0.5f, -0.5f, 0.0f,   // bottom right  
             0.5f,  0.5f, 0.0f }; // top right  
  
    private short drawOrder[] = { 0, 1, 2, 0, 2, 3 }; // order to draw vertices  
  
    public Square() {  
        // initialize vertex byte buffer for shape coordinates  
        ByteBuffer bb = ByteBuffer.allocateDirect(  
        // (# of coordinate values \* 4 bytes per float)  
                squareCoords.length \* 4);  
        bb.order(ByteOrder.nativeOrder());  
        vertexBuffer = bb.asFloatBuffer();  
        vertexBuffer.put(squareCoords);  
        vertexBuffer.position(0);  
  
        // initialize byte buffer for the draw list  
        ByteBuffer dlb = ByteBuffer.allocateDirect(  
        // (# of coordinate values \* 2 bytes per short)  
                drawOrder.length \* 2);  
        dlb.order(ByteOrder.nativeOrder());  
        drawListBuffer = dlb.asShortBuffer();  
        drawListBuffer.put(drawOrder);  
        drawListBuffer.position(0);  
    }  
}

This example gives you a peek at what it takes to create more complex shapes with OpenGL. In general, you use collections of triangles to draw objects.

Draw shapes

After you define shapes to be drawn with OpenGL, you probably want to draw them. Drawing shapes with the OpenGL ES 2.0 takes a bit more code than you might imagine, because the API provides a great deal of control over the graphics rendering pipeline.

This lesson explains how to draw the shapes you defined in the previous lesson using the OpenGL ES 2.0 API.

Initialize shapes

Before you do any drawing, you must initialize and load the shapes you plan to draw. Unless the structure (the original coordinates) of the shapes you use in your program change during the course of execution, you should initialize them in the [onSurfaceCreated()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer" \l "onSurfaceCreated(javax.microedition.khronos.opengles.GL10,%20javax.microedition.khronos.egl.EGLConfig)) method of your renderer for memory and processing efficiency.

public class MyGLRenderer implements GLSurfaceView.Renderer {  
  
    ...  
    private Triangle mTriangle;  
    private Square   mSquare;  
  
    public void onSurfaceCreated(GL10 unused, EGLConfig config) {  
        ...  
        // initialize a triangle  
        mTriangle = new Triangle();  
        // initialize a square  
        mSquare = new Square();  
    }  
    ...  
}

Draw a shape

Drawing a defined shape using OpenGL ES 2.0 requires a significant amount of code, because you must provide a lot of details to the graphics rendering pipeline. Specifically, you must define the following:

* *Vertex Shader* - OpenGL ES graphics code for rendering the vertices of a shape.
* *Fragment Shader* - OpenGL ES code for rendering the face of a shape with colors or textures.
* *Program* - An OpenGL ES object that contains the shaders you want to use for drawing one or more shapes.

You need at least one vertex shader to draw a shape and one fragment shader to color that shape. These shaders must be compiled and then added to an OpenGL ES program, which is then used to draw the shape. Here is an example of how to define basic shaders you can use to draw a shape in the Triangle class:

public class Triangle {  
  
    private final String vertexShaderCode =  
        "attribute vec4 vPosition;" +  
        "void main() {" +  
        "  gl\_Position = vPosition;" +  
        "}";  
  
    private final String fragmentShaderCode =  
        "precision mediump float;" +  
        "uniform vec4 vColor;" +  
        "void main() {" +  
        "  gl\_FragColor = vColor;" +  
        "}";  
  
    ...  
}

Shaders contain OpenGL Shading Language (GLSL) code that must be compiled prior to using it in the OpenGL ES environment. To compile this code, create a utility method in your renderer class:

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

In order to draw your shape, you must compile the shader code, add them to a OpenGL ES program object and then link the program. Do this in your drawn object’s constructor, so it is only done once.

public class Triangle() {  
    ...  
  
    private final int mProgram;  
  
    public Triangle() {  
        ...  
  
        int vertexShader = MyGLRenderer.loadShader(GLES20.GL\_VERTEX\_SHADER,  
                                        vertexShaderCode);  
        int fragmentShader = MyGLRenderer.loadShader(GLES20.GL\_FRAGMENT\_SHADER,  
                                        fragmentShaderCode);  
  
        // create empty OpenGL ES Program  
        mProgram = GLES20.glCreateProgram();  
  
        // add the vertex shader to program  
        GLES20.glAttachShader(mProgram, vertexShader);  
  
        // add the fragment shader to program  
        GLES20.glAttachShader(mProgram, fragmentShader);  
  
        // creates OpenGL ES program executables  
        GLES20.glLinkProgram(mProgram);  
    }  
}

At this point, you are ready to add the actual calls that draw your shape. Drawing shapes with OpenGL ES requires that you specify several parameters to tell the rendering pipeline what you want to draw and how to draw it. Since drawing options can vary by shape, it's a good idea to have your shape classes contain their own drawing logic.

Create a draw() method for drawing the shape. This code sets the position and color values to the shape’s vertex shader and fragment shader, and then executes the drawing function.

private int positionHandle;  
private int colorHandle;  
  
private final int vertexCount = triangleCoords.length / COORDS\_PER\_VERTEX;  
private final int vertexStride = COORDS\_PER\_VERTEX \* 4; // 4 bytes per vertex  
  
public void draw() {  
    // Add program to OpenGL ES environment  
    GLES20.glUseProgram(mProgram);  
  
    // get handle to vertex shader's vPosition member  
    positionHandle = GLES20.glGetAttribLocation(mProgram, "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,  
                                 vertexStride, vertexBuffer);  
  
    // get handle to fragment shader's vColor member  
    colorHandle = GLES20.glGetUniformLocation(mProgram, "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);  
}

Once you have all this code in place, drawing this object just requires a call to the draw() method from within your renderer’s [onDrawFrame()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer" \l "onDrawFrame(javax.microedition.khronos.opengles.GL10)) method:

public void onDrawFrame(GL10 unused) {  
    ...  
  
    triangle.draw();  
}

When you run the application, it should look something like this:

There are a few problems with this code 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’s vertices have not been corrected for the proportions of the screen area where the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) is displayed. You can fix that problem using a projection and camera view.

Lastly, the triangle is stationary, which is a bit boring. In the [Add motion](https://developer.android.com/training/graphics/opengl/motion) lesson, you make this shape rotate and make more interesting use of the OpenGL ES graphics pipeline.

Apply projection and camera views

In the OpenGL ES environment, projection and camera views allow you to display drawn objects in a way that more closely resembles how you see physical objects with your eyes. This simulation of physical viewing is done with mathematical transformations of drawn object coordinates:

* *Projection* - This transformation adjusts the coordinates of drawn objects based on the width and height of the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) where they are displayed. Without this calculation, objects drawn by OpenGL ES are skewed by the unequal proportions of the view window. A projection transformation typically only has to be calculated when the proportions of the OpenGL view are established or changed in the [onSurfaceChanged()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer" \l "onSurfaceChanged(javax.microedition.khronos.opengles.GL10,%20int,%20int)) method of your renderer.
* *Camera View* - This transformation adjusts the coordinates of drawn objects based on a virtual camera position. It’s important to note that OpenGL ES does not define an actual camera object, but instead provides utility methods that simulate a camera by transforming the display of drawn objects. A camera view transformation might be calculated only once when you establish your [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView), or might change dynamically based on user actions or your application’s function.

Define a projection

The data for a projection transformation is calculated in the [onSurfaceChanged()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer" \l "onSurfaceChanged(javax.microedition.khronos.opengles.GL10,%20int,%20int)) method of your [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer) class. The following example code takes the height and width of the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) and uses it to populate a projection transformation [Matrix](https://developer.android.com/reference/android/opengl/Matrix) using the [Matrix.frustumM()](https://developer.android.com/reference/android/opengl/Matrix" \l "frustumM(float[],%20int,%20float,%20float,%20float,%20float,%20float,%20float)) method:

// vPMatrix is an abbreviation for "Model View Projection Matrix"  
private final float[] vPMatrix = new float[16];  
private final float[] projectionMatrix = new float[16];  
private final float[] viewMatrix = new float[16];  
  
@Override  
public void onSurfaceChanged(GL10 unused, int width, int height) {  
    GLES20.glViewport(0, 0, width, height);  
  
    float ratio = (float) width / height;  
  
    // this projection matrix is applied to object coordinates  
    // in the onDrawFrame() method  
    Matrix.frustumM(projectionMatrix, 0, -ratio, ratio, -1, 1, 3, 7);  
}

This code populates a projection matrix, mProjectionMatrix which you can then combine with a camera view transformation in the [onDrawFrame()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer" \l "onDrawFrame(javax.microedition.khronos.opengles.GL10)) method.

Define a camera view

Complete the process of transforming your drawn objects by adding a camera view transformation as part of the drawing process in your renderer. In the following example code, the camera view transformation is calculated using the [Matrix.setLookAtM()](https://developer.android.com/reference/android/opengl/Matrix#setLookAtM(float[],%20int,%20float,%20float,%20float,%20float,%20float,%20float,%20float,%20float,%20float)) method and then combined with the previously calculated projection matrix. The combined transformation matrices are then passed to the drawn shape.

@Override  
public void onDrawFrame(GL10 unused) {  
    ...  
    // Set the camera position (View matrix)  
    Matrix.setLookAtM(viewMatrix, 0, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);  
  
    // Calculate the projection and view transformation  
    Matrix.multiplyMM(vPMatrix, 0, projectionMatrix, 0, viewMatrix, 0);  
  
    // Draw shape  
    triangle.draw(vPMatrix);  
}

Apply projection and camera transformations

In order to use the combined projection and camera view transformation matrix shown in the previews sections, first add a matrix variable to the *vertex shader* previously defined in the Triangle class:

public class Triangle {  
  
    private final String vertexShaderCode =  
        // This matrix member variable provides a hook to manipulate  
        // the coordinates of the objects that use this vertex shader  
        "uniform mat4 uMVPMatrix;" +  
        "attribute vec4 vPosition;" +  
        "void main() {" +  
        // the matrix must be included as a modifier of gl\_Position  
        // Note that the uMVPMatrix factor \*must be first\* in order  
        // for the matrix multiplication product to be correct.  
        "  gl\_Position = uMVPMatrix \* vPosition;" +  
        "}";  
  
    // Use to access and set the view transformation  
    private int vPMatrixHandle;  
  
    ...  
}

Next, modify the draw() method of your graphic objects to accept the combined transformation matrix and apply it to the shape:

[KOTLIN](https://developer.android.com/training/graphics/opengl/projection#kotlin)[JAVA](https://developer.android.com/training/graphics/opengl/projection#java)

public void draw(float[] mvpMatrix) { // pass in the calculated transformation matrix  
    ...  
  
    // get handle to shape's transformation matrix  
    vPMatrixHandle = GLES20.glGetUniformLocation(program, "uMVPMatrix");  
  
    // Pass the projection and view transformation to the shader  
    GLES20.glUniformMatrix4fv(vPMatrixHandle, 1, false, mvpMatrix, 0);  
  
    // Draw the triangle  
    GLES20.glDrawArrays(GLES20.GL\_TRIANGLES, 0, vertexCount);  
  
    // Disable vertex array  
    GLES20.glDisableVertexAttribArray(positionHandle);  
}

Now that you have an application that displays your shapes in correct proportions, it's time to add motion to your shapes.

# Add motion

Drawing objects on screen is a pretty basic feature of OpenGL, but you can do this with other Android graphics framework classes, including [Canvas](https://developer.android.com/reference/android/graphics/Canvas) and [Drawable](https://developer.android.com/reference/android/graphics/drawable/Drawable) objects. OpenGL ES provides additional capabilities for moving and transforming drawn objects in three dimensions or in other unique ways to create compelling user experiences.

In this lesson, you take another step forward into using OpenGL ES by learning how to add motion to a shape with rotation.

## Rotate a shape

Rotating a drawing object with OpenGL ES 2.0 is relatively simple. In your renderer, create another transformation matrix (a rotation matrix) and then combine it with your projection and camera view transformation matrices:

private float[] rotationMatrix = new float[16];  
Override  
public void onDrawFrame(GL10 gl) {  
    float[] scratch = new float[16];  
  
    ...  
  
    // Create a rotation transformation for the triangle  
    long time = SystemClock.uptimeMillis() % 4000L;  
    float angle = 0.090f \* ((int) time);  
    Matrix.setRotateM(rotationMatrix, 0, angle, 0, 0, -1.0f);  
  
    // Combine the rotation matrix with the projection and camera view  
    // Note that the vPMatrix factor \*must be first\* in order  
    // for the matrix multiplication product to be correct.  
    Matrix.multiplyMM(scratch, 0, vPMatrix, 0, rotationMatrix, 0);  
  
    // Draw triangle  
    mTriangle.draw(scratch);  
}

If your triangle does not rotate after making these changes, make sure you have commented out the [GLSurfaceView.RENDERMODE\_WHEN\_DIRTY](https://developer.android.com/reference/android/opengl/GLSurfaceView" \l "RENDERMODE_WHEN_DIRTY) setting

## Enable continuous rendering

public MyGLSurfaceView(Context context) extends GLSurfaceView {  
    ...  
    // Render the view only when there is a change in the drawing data.  
    // To allow the triangle to rotate automatically, this line is commented out:  
    //setRenderMode(GLSurfaceView.RENDERMODE\_WHEN\_DIRTY);  
}

# 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 ES graphics? The key to making your OpenGL ES application touch interactive is expanding your implementation of [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) to override the [onTouchEvent()](https://developer.android.com/reference/android/view/View" \l "onTouchEvent(android.view.MotionEvent)) to listen for touch events.

## Setup a touch listener

In order to make your OpenGL ES application respond to touch events, you must implement the [onTouchEvent()](https://developer.android.com/reference/android/view/View" \l "onTouchEvent(android.view.MotionEvent)) method in your [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) class. The example implementation below shows how to listen for [MotionEvent.ACTION\_MOVE](https://developer.android.com/reference/android/view/MotionEvent" \l "ACTION_MOVE) events and translate them to an angle of rotation for a shape.

private final float TOUCH\_SCALE\_FACTOR = 180.0f / 320;  
private float previousX;  
private float previousY;  
  
@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 - 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));  
            requestRender();  
    }  
  
    previousX = x;  
    previousY = y;  
    return true;  
}

Notice that after calculating the rotation angle, this method calls [requestRender()](https://developer.android.com/reference/android/opengl/GLSurfaceView" \l "requestRender()) to tell the renderer that it is time to render the frame. This approach is the most efficient in this example because the frame does not need to be redrawn unless there is a change in the rotation. However, it does not have any impact on efficiency unless you also request that the renderer only redraw when the data changes using the [setRenderMode()](https://developer.android.com/reference/android/opengl/GLSurfaceView" \l "setRenderMode(int)) method, so make sure this line is uncommented in the renderer:

public MyGLSurfaceView(Context context) {  
    ...  
    // Render the view only when there is a change in the drawing data  
    setRenderMode(GLSurfaceView.RENDERMODE\_WHEN\_DIRTY);  
}

## Expose the rotation angle

The example code above requires that you expose the rotation angle through your renderer by adding a public member. Since the renderer code is running on a separate thread from the main user interface thread of your application, you must declare this public variable as volatile. Here is the code to declare the variable and expose the getter and setter pair:

[KOTLIN](https://developer.android.com/training/graphics/opengl/touch#kotlin)[JAVA](https://developer.android.com/training/graphics/opengl/touch#java)

public class MyGLRenderer implements GLSurfaceView.Renderer {  
    ...  
  
    public volatile float mAngle;  
  
    public float getAngle() {  
        return mAngle;  
    }  
  
    public void setAngle(float angle) {  
        mAngle = angle;  
    }  
}

## Apply rotation

To apply the rotation generated by touch input, comment out the code that generates an angle and add a variable that contains the touch input generated angle:

public void onDrawFrame(GL10 gl) {  
    ...  
    float[] scratch = new float[16];  
  
    // Create a rotation for the triangle  
    // long time = SystemClock.uptimeMillis() % 4000L;  
    // float angle = 0.090f \* ((int) time);  
    Matrix.setRotateM(rotationMatrix, 0, mAngle, 0, 0, -1.0f);  
  
    // Combine the rotation matrix with the projection and camera view  
    // Note that the vPMatrix factor \*must be first\* in order  
    // for the matrix multiplication product to be correct.  
    Matrix.multiplyMM(scratch, 0, vPMatrix, 0, rotationMatrix, 0);  
  
    // Draw triangle  
    mTriangle.draw(scratch);  
}

# Reduce overdraw

An app may draw the same pixel more than once within a single frame, an event called overdraw. Overdraw is usually unnecessary, and best eliminated. It manifests itself as a performance problem by wasting GPU time to render pixels that don't contribute to what the user sees on the screen.

## About overdraw

Overdraw refers to the system's drawing a pixel on the screen multiple times in a single frame of rendering.

## Find overdraw problems

The platform offers the following tools to help you determine if overdraw is affecting your app's performance.

### **Debug GPU overdraw tool**

The Debug GPU Overdraw tool uses color-coding to show the number of times your app draws each pixel on the screen. The higher this count, the more likely it is that overdraw affects your app's performance.

### **Profile GPU rendering tool**

The Profile GPU Rendering tool displays, as a scrolling histogram, the time each stage of the rendering pipeline takes to display a single frame. The Process part of each bar, indicated in orange, shows when the system is swapping buffers; this metric provides important clues about overdraw.

On less performant GPUs, available fill-rate (the speed at which the GPU can fill the frame buffer) can be quite low. As the number of pixels required to draw a frame increases, the GPU may take longer to process new commands, and ask the rest of the system to wait until it can catch up. The Process bar shows that this spike happens as the GPU gets overwhelmed trying to draw pixels as fast as possible. Issues other than raw numbers of pixels may also cause this metric to spike. For example, if the Debug GPU Overdraw tool shows heavy overdraw and Process spikes, there's likely an issue with overdraw.

## Fix overdraw

There are several strategies you can pursue to reduce or eliminate overdraw:

* Removing unneeded backgrounds in layouts.
* Flattening the view hierarchy.
* Reducing transparency.

This section provides information about each of these approaches.

### **Remove unneeded backgrounds in layouts**

By default, a layout does not have a background, which means it does not render anything directly by itself. When layouts do have backgrounds, however, they may contribute to overdraw.

Removing unnecessary backgrounds is a quick way of improving rendering performance. An unnecessary background may never be visible because it's completely covered by everything else the app is drawing on top of that view. For example, the system may entirely cover up a parent's background when it draws child views on top of it.

To find out why you're overdrawing, walk through the hierarchy in the [Layout Inspector](https://developer.android.com/studio/debug/layout-inspector) tool. As you do so, look out for any backgrounds you can eliminate because they are not visible to the user. Cases where many containers share a common background color offer another opportunity to eliminate unneeded backgrounds: You can set the window background to the main background color of your app, and leave all of the containers above it with no background values defined.

### **Flatten view hierarchy**

Modern layouts make it easy to stack and layer views to produce beautiful design. However, doing so can degrade performance by resulting in overdraw, especially in scenarios where each stacked view object is opaque, requiring the drawing of both seen and unseen pixels to the screen.

If you encounter this sort of issue, you may be able to improve performance by optimizing your view hierarchy to reduce the number of overlapping UI objects. For more information about how to accomplish this, see [Optimize view hierarchies](https://developer.android.com/topic/performance/optimizing-view-hierarchies).

### **Reduce transparency**

Rendering of transparent pixels on screen, known as alpha rendering, is a key contributor to overdraw. Unlike standard overdraw, in which the system completely hides existing drawn pixels by drawing opaque pixels on top of them, transparent objects require existing pixels to be drawn first, so that the right blending equation can occur. Visual effects like transparent animations, fade-outs, and drop shadows all involve some sort of transparency, and can therefore contribute significantly to overdraw. You can improve overdraw in these situations by reducing the number of transparent objects you render. For example, you can get gray text by drawing black text in a [TextView](https://developer.android.com/reference/android/widget/TextView) with a translucent alpha value set on it. But you can get the same effect with far better performance by simply drawing the text in gray.

# Performance and view hierarchies

The way you manage the hierarchy of your [View](https://developer.android.com/reference/android/view/View) objects can have a substantial impact on your app’s performance. This page describes how to assess whether your view hierarchy is slowing your app down, and offers some strategies for addressing issues that may arise.

## Layout and measure performance

The rendering pipeline includes a layout-and-measure stage, during which the system appropriately positions the relevant items in your view hierarchy. The measure part of this stage determines the sizes and boundaries of [View](https://developer.android.com/reference/android/view/View) objects. The layout part determines where on the screen to position the [View](https://developer.android.com/reference/android/view/View) objects.

Both of these pipeline stages incur some small cost per view or layout that they process. Most of the time, this cost is minimal and doesn’t noticeably affect performance. However, it can be greater when an app adds or removes View objects, such as when a [RecyclerView](https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView) object recycles them or reuses them. The cost can also be higher if a [View](https://developer.android.com/reference/android/view/View) object needs to consider resizing to main its constraints.

Because you cannot move these operations to a worker thread—your app must process them on the main thread—your best bet is to optimize them so that they can take as little time as possible.

### **Manage complexity: layouts matter**

Android [Layouts](https://developer.android.com/guide/topics/ui/declaring-layout) allow you to nest UI objects in the view hierarchy. This nesting can also impose a layout cost. When your app processes an object for layout, the app performs the same process on all children of the layout as well. For a complicated layout, sometimes a cost only arises the first time the system computes the layout. For instance, when your app recycles a complex list item in a [RecyclerView](https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView) object, the system needs to lay out all of the objects.

The most common case in which layout takes an especially long time is when hierarchies of [View](https://developer.android.com/reference/android/view/View) objects are nested within one another. Each nested layout object adds cost to the layout stage. The flatter your hierarchy, the less time that it takes for the layout stage to complete.

If you are using the [RelativeLayout](https://developer.android.com/reference/android/widget/RelativeLayout) class, you may be able to achieve the same effect, at lower cost, by using nested, unweighted [LinearLayout](https://developer.android.com/reference/android/widget/LinearLayout) views instead. Additionally, if your app targets Android 7.0 (API level 24), it is likely that you can use a special layout editor to create a [ConstraintLayout](http://tools.android.com/tech-docs/layout-editor) object instead of [RelativeLayout](https://developer.android.com/reference/android/widget/RelativeLayout). Doing so allows you to avoid many of the issues this section describes. The [ConstraintLayout](http://tools.android.com/tech-docs/layout-editor) class offers similar layout control, but with much-improved performance. This class uses its own constraint-solving system to resolve relationships between views in a very different way from standard layouts.