**Intro –**

openGL ES stands for Open Graphics Language for Embedded Systems

and was

Developed by Khronos Group and released initially 28 July 2003; 16 years ago

**What is it**

OpenGL ES is a subset of the OpenGL, intended for embedded devices such as smartphones, consoles, appliances and vehicles.

[OpenGL](https://en.wikipedia.org/wiki/OpenGL) is a [computer graphics rendering](https://en.wikipedia.org/wiki/Rendering_(computer_graphics)) [application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface) (API) for rendering [2D](https://en.wikipedia.org/wiki/2D_computer_graphics) and [3D computer graphics](https://en.wikipedia.org/wiki/3D_computer_graphics)

The OpenGL ES APIs provided by the Android framework offer a set of tools for displaying high-end, animated graphics and can also benefit from the acceleration of graphics processing units (GPUs) provided on many Android devices.

OpenGL ES is a flavor of the OpenGL specification intended for embedded devices.

OpenGL is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware.

**OpenGL for Embedded Systems** (**OpenGL ES** or **GLES**) is a subset[[2]](https://en.wikipedia.org/wiki/OpenGL_ES#cite_note-2) of the [OpenGL](https://en.wikipedia.org/wiki/OpenGL) [computer graphics rendering](https://en.wikipedia.org/wiki/Rendering_(computer_graphics)) [application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface) (API) for rendering [2D](https://en.wikipedia.org/wiki/2D_computer_graphics) and [3D computer graphics](https://en.wikipedia.org/wiki/3D_computer_graphics) such as those used by [video games](https://en.wikipedia.org/wiki/Video_game), typically [hardware-accelerated](https://en.wikipedia.org/wiki/Hardware_acceleration) using a [graphics processing unit](https://en.wikipedia.org/wiki/Graphics_processing_unit) (GPU). It is designed for [embedded systems](https://en.wikipedia.org/wiki/Embedded_systems) like [smartphones](https://en.wikipedia.org/wiki/Smartphones), [tablet computers](https://en.wikipedia.org/wiki/Tablet_computer), [video game consoles](https://en.wikipedia.org/wiki/Video_game_console) and [PDAs](https://en.wikipedia.org/wiki/Personal_digital_assistant). OpenGL ES is the "most widely deployed 3D graphics API in history".[[3]](https://en.wikipedia.org/wiki/OpenGL_ES#cite_note-3D-BOF-SIGGRAPH_Aug15-3)

OpenGL® ES is a royalty-free, cross-platform API for rendering advanced 2D and 3D graphics on embedded and mobile systems - including consoles, phones, appliances and vehicles. It consists of a well-defined subset of desktop OpenGL suitable for low-power devices, and provides a flexible and powerful interface between software and graphics acceleration hardware.

The Android framework provides plenty of standard tools for creating attractive, functional graphical user interfaces. However, if you want more control of what your application draws on screen, or are venturing into three dimensional graphics, you need to use a different tool. The OpenGL ES APIs provided by the Android framework offers a set of tools for displaying high-end, animated graphics that are limited only by your imagination and can also benefit from the acceleration of graphics processing units (GPUs) provided on many Android devices.

**Differences between openGL and GLES**

OpenGL ES is an “embeddable subset” of OpenGL. It slims down the rather large OpenGL API to the bare essentials, so that it can be implemented on devices with simpler, cheaper hardware, and above all, low enough power requirements to run on batteries.

For example, it is available as standard on smartphones running both Apple’s IOS and Google’s Android operating system.

One significant difference between OpenGL and OpenGL ES is that OpenGL ES removed the need to bracket OpenGL library calls with glBegin and glEnd.

ISERT CODE SNIPPETS

No support for polygons other than triangles.

* ES adds the option to specify coordinates etc as fixed-point values (calls with an x suffix) instead of floating-point (f suffix).
* On some platforms (e.g. Android GLES 2.0) there is no [OpenGL error queue](https://www.khronos.org/opengl/wiki/OpenGL_Error#Catching_errors_.28the_hard_way.29) storage, only last error is stored.

where

Android supports several versions of the OpenGL ES API:

* OpenGL ES 1.0 and 1.1 - This API specification is supported by Android 1.0 and higher.
* OpenGL ES 2.0 - This API specification is supported by Android 2.2 (API level 8) and higher.
* OpenGL ES 3.0 - This API specification is supported by Android 4.3 (API level 18) and higher.
* OpenGL ES 3.1 - This API specification is supported by Android 5.0 (API level 21) and higher.

3.2 a year later with slight improvements ontop of v 3.1

OpenGL ES 3.2 - Additional OpenGL functionality

The latest in the series, OpenGL ES 3.2 added additional functionality based on the [Android Extension Pack](https://www.khronos.org/registry/OpenGL/extensions/ANDROID/ANDROID_extension_pack_es31a.txt) for OpenGL ES 3.1, which brought the mobile API's functionality significantly closer to it's desktop counterpart - OpenGL.

OpenGL ES 3.1 - Bringing Compute to Mobile Graphics

Despite being only a bump in the minor revision of the API, OpenGL ES 3.1 was an enormous milestone for the API, as it added the ability to do general purpose compute in the API, bringing compute to mobile graphics.

OpenGL ES 3.0 - Enhanced Graphics

OpenGL ES 3.0 was another evolutionary step for OpenGL ES, notably including multiple render targets, additional texturing capabilities, uniform buffers, instancing and transform feedback.

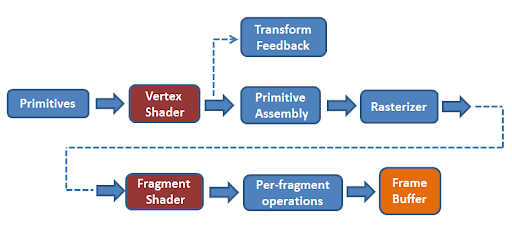
OpenGL ES 2.0 - Programmable Shading

OpenGL ES 2.0 was the first portable mobile graphics API to expose programmable shaders in the then latest generation of graphics hardware. It remains a prevalent API today, and still is the most widely available 3D graphics API and remains a solid choice to target the widest range of devices in the market.

Lower graphics but faster fps

OpenGL ES 1.X - Fixed Function Graphics

OpenGL ES 1.0 and 1.1 were the first portable mobile graphics APIs, defined relative to the OpenGL 1.5 specification, providing fixed function graphics acceleration



How it works

Transforms data into images

Data -> processing->vectorization->primitive shaders-> rasterization->colour shaders->output

Simple description

Description following code snips

Basics

In order to draw graphics with OpenGL ES in your Android application, you must create a view container for them. One of the more straight-forward 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. For more information about these classes, see the [OpenGL ES](https://developer.android.com/guide/topics/graphics/opengl) developer guide.

[GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) is just one way to incorporate OpenGL ES graphics into your application. For a full-screen or near-full screen graphics view, it is a reasonable choice. Developers who want to incorporate OpenGL ES graphics in a small portion of their layouts should take a look at [TextureView](https://developer.android.com/reference/android/view/TextureView). For real, do-it-yourself developers, it is also possible to build up an OpenGL ES view using [SurfaceView](https://developer.android.com/reference/android/view/SurfaceView), but this requires writing quite a bit of additional code.

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.

Android supports OpenGL both through its framework API and the Native Development Kit (NDK). This topic focuses on the Android framework interfaces. For more information about the NDK, see the [Android NDK](https://developer.android.com/tools/sdk/ndk).

There are two foundational classes in the Android framework that let you create and manipulate graphics with the OpenGL ES API: [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) and [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer). If your goal is to use OpenGL in your Android application, understanding how to implement these classes in an activity should be your first objective.

[**GLSurfaceView**](https://developer.android.com/reference/android/opengl/GLSurfaceView)

This class is a [View](https://developer.android.com/reference/android/view/View) where you can draw and manipulate objects using OpenGL API calls and is similar in function to a [SurfaceView](https://developer.android.com/reference/android/view/SurfaceView). You can use this class by creating an instance of [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) and adding your [Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer) to it. However, if you want to capture touch screen events, you should extend the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) class to implement the touch listeners, as shown in OpenGL training lesson, [Responding to touch events](https://developer.android.com/training/graphics/opengl/touch).

[**GLSurfaceView.Renderer**](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer)

This interface defines the methods required for drawing graphics in a [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView). You must provide an implementation of this interface as a separate class and attach it to your [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) instance using [GLSurfaceView.setRenderer()](https://developer.android.com/reference/android/opengl/GLSurfaceView" \l "setRenderer(android.opengl.GLSurfaceView.Renderer)).

The [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer) interface requires that you implement the following methods:

* [onSurfaceCreated()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer#onSurfaceCreated(javax.microedition.khronos.opengles.GL10,%20javax.microedition.khronos.egl.EGLConfig)): The system calls this method once, when creating the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView). Use this method to perform actions that need to happen only once, such as setting OpenGL environment parameters or initializing OpenGL graphic objects.
* [onDrawFrame()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer#onDrawFrame(javax.microedition.khronos.opengles.GL10)): The system calls this method on each redraw of the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView). Use this method as the primary execution point for drawing (and re-drawing) graphic objects.
* [onSurfaceChanged()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer#onSurfaceChanged(javax.microedition.khronos.opengles.GL10,%20int,%20int)): The system calls this method when the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) geometry changes, including changes in size of the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) or orientation of the device screen. For example, the system calls this method when the device changes from portrait to landscape orientation. Use this method to respond to changes in the [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) container.

Where to start?

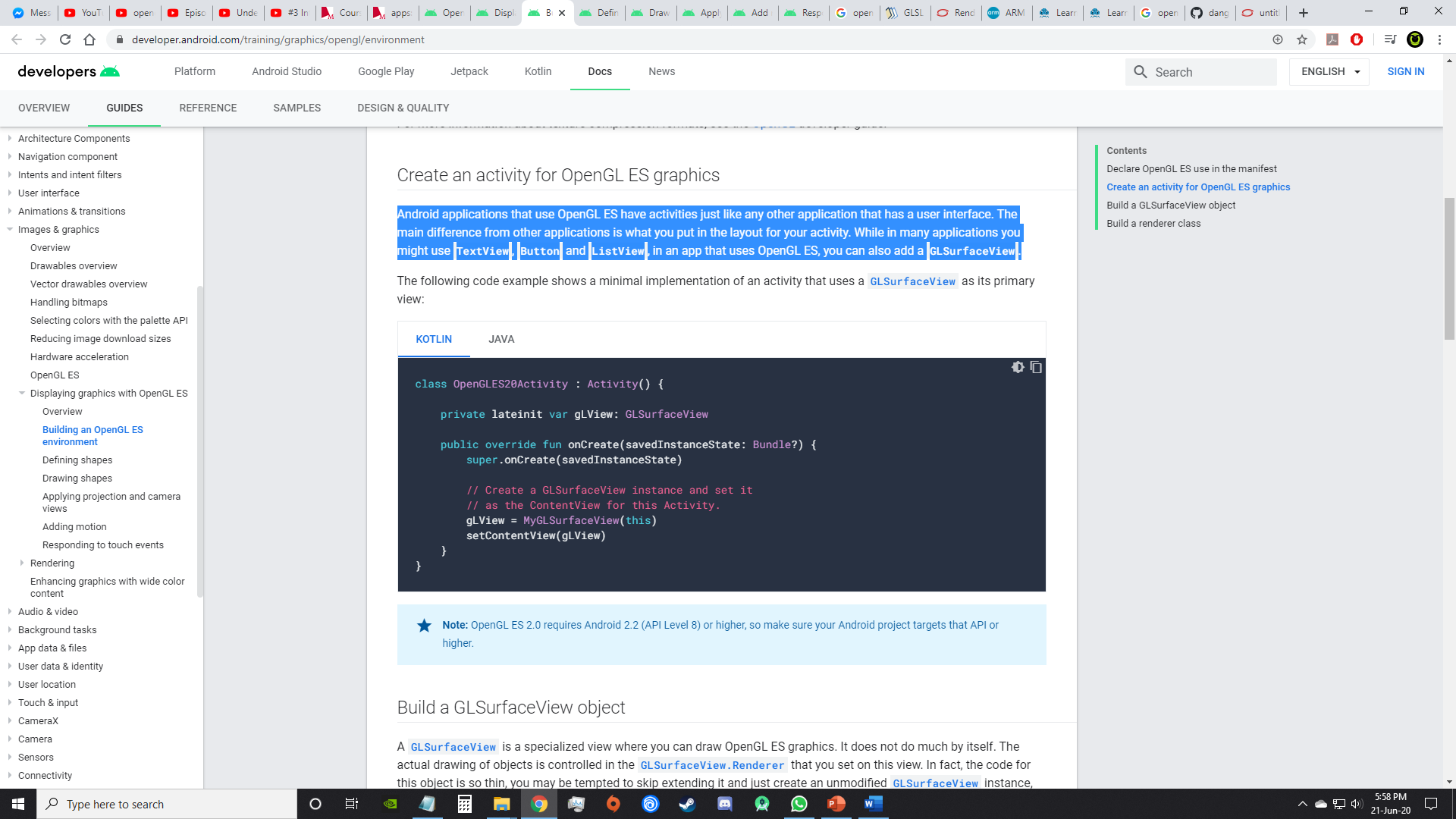
So how do we start coding a 3d object.

Apps that use OpenGL ES have activities just like other applications.

But instead of only using [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), like in normal apps, apps that uses OpenGL ES can also add a [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView).

A [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView) is a specialized view where you can draw OpenGL ES graphics.

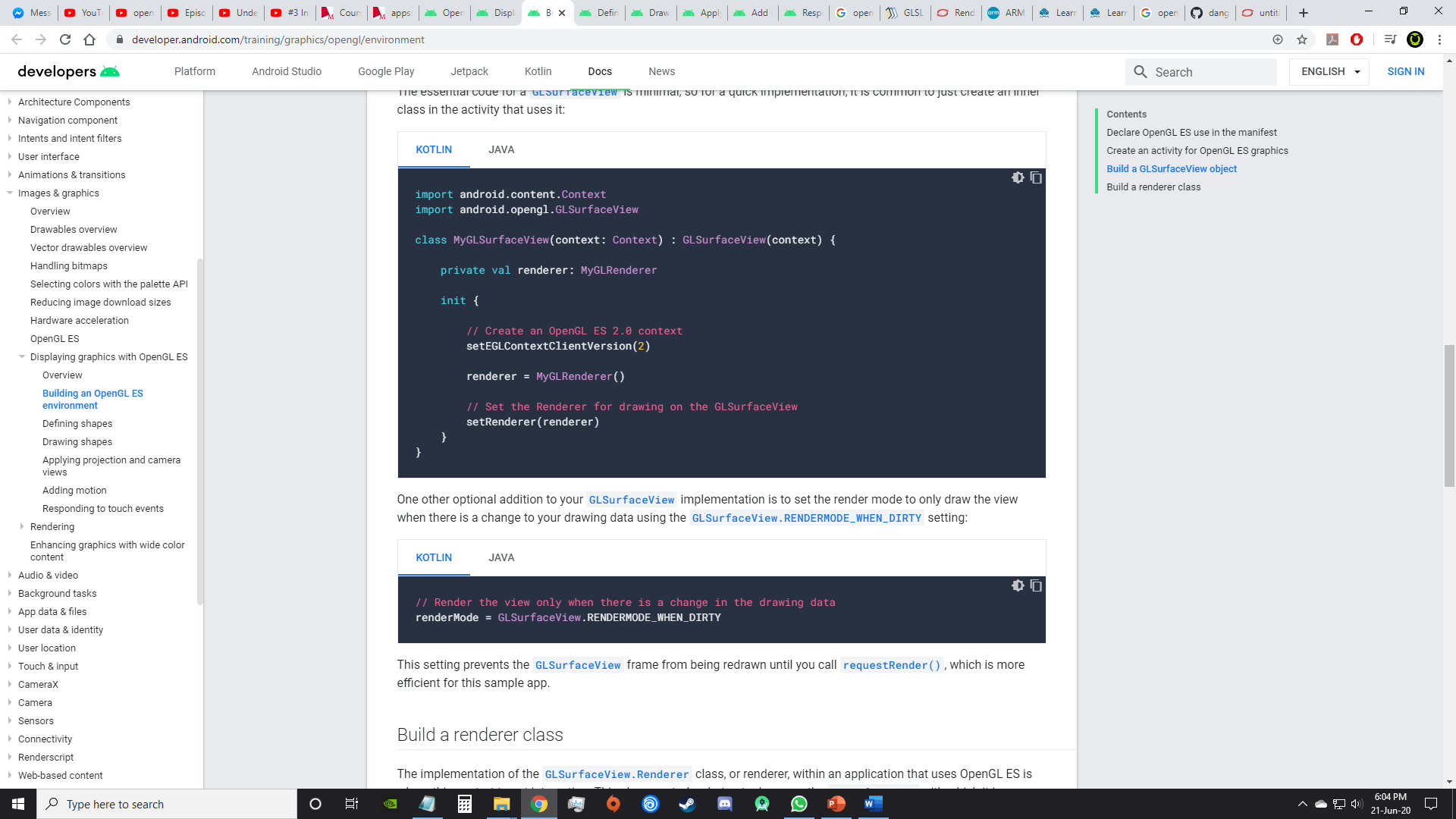
-defines the drawing space/’canvas’



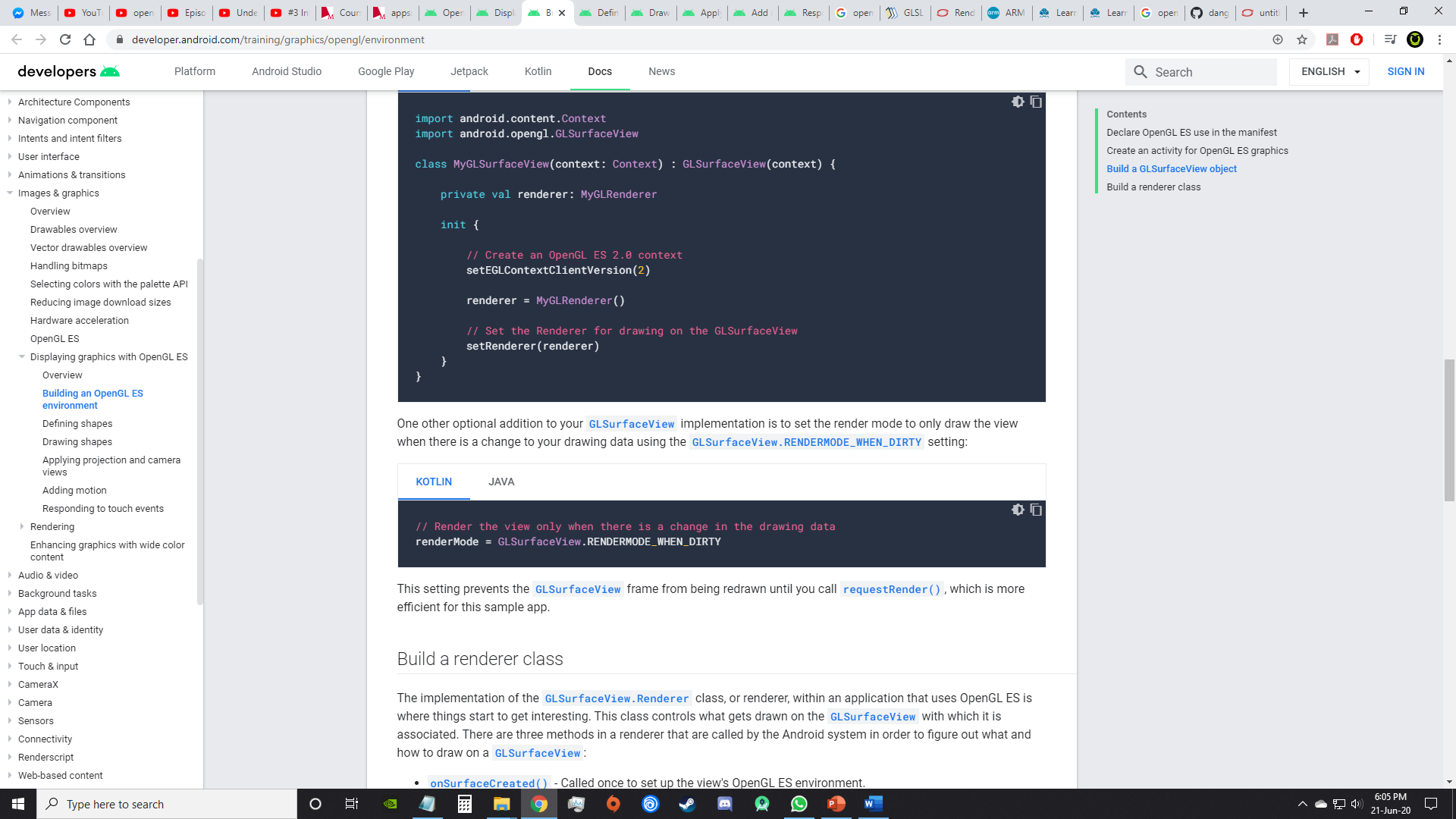
But [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView)  does not do much by itself.

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 view.

The essential code for a [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView)



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:



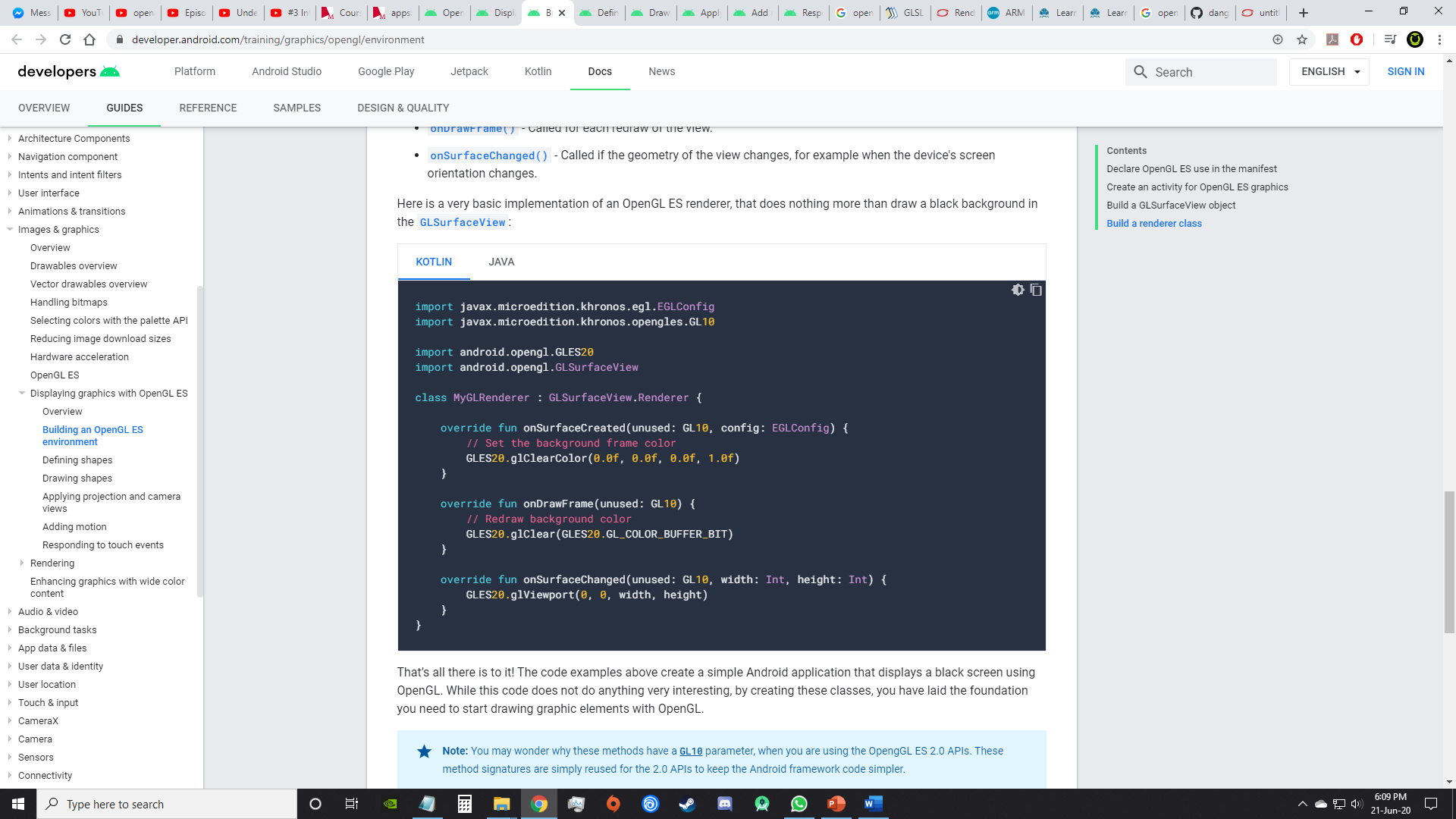
## **Build a renderer class**

The GLSurfaceView.Renderer 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):



# **Define shapes**

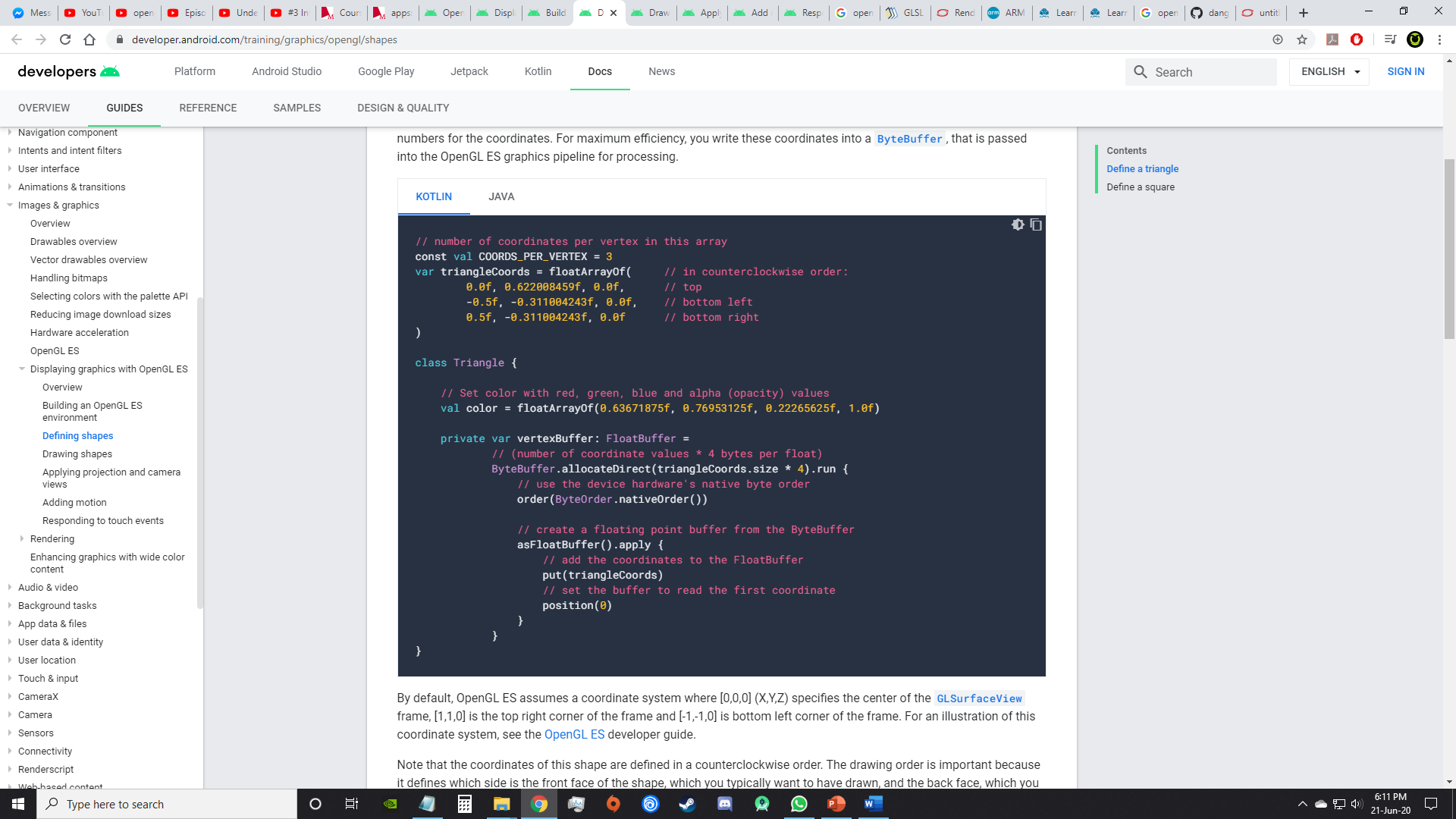
Defining shapes to be drawn is the first step in creating high-end graphics.

What is the most important shape?

## **Define a triangle**

A triangle is defined by the 3D coordinates of its vertices. In OpenGL, the typical way to do this is to define a vertex array of floating point numbers for the coordinates.

These are written into a [ByteBuffer](https://developer.android.com/reference/java/nio/ByteBuffer), that is passed into the graphics pipeline.



## **Shape faces and winding**

How do you know which face is front and which is the back?

The answer has to do with winding, or, the direction in which you define the points of a shape.

Coordinates at
vertices of a triangle

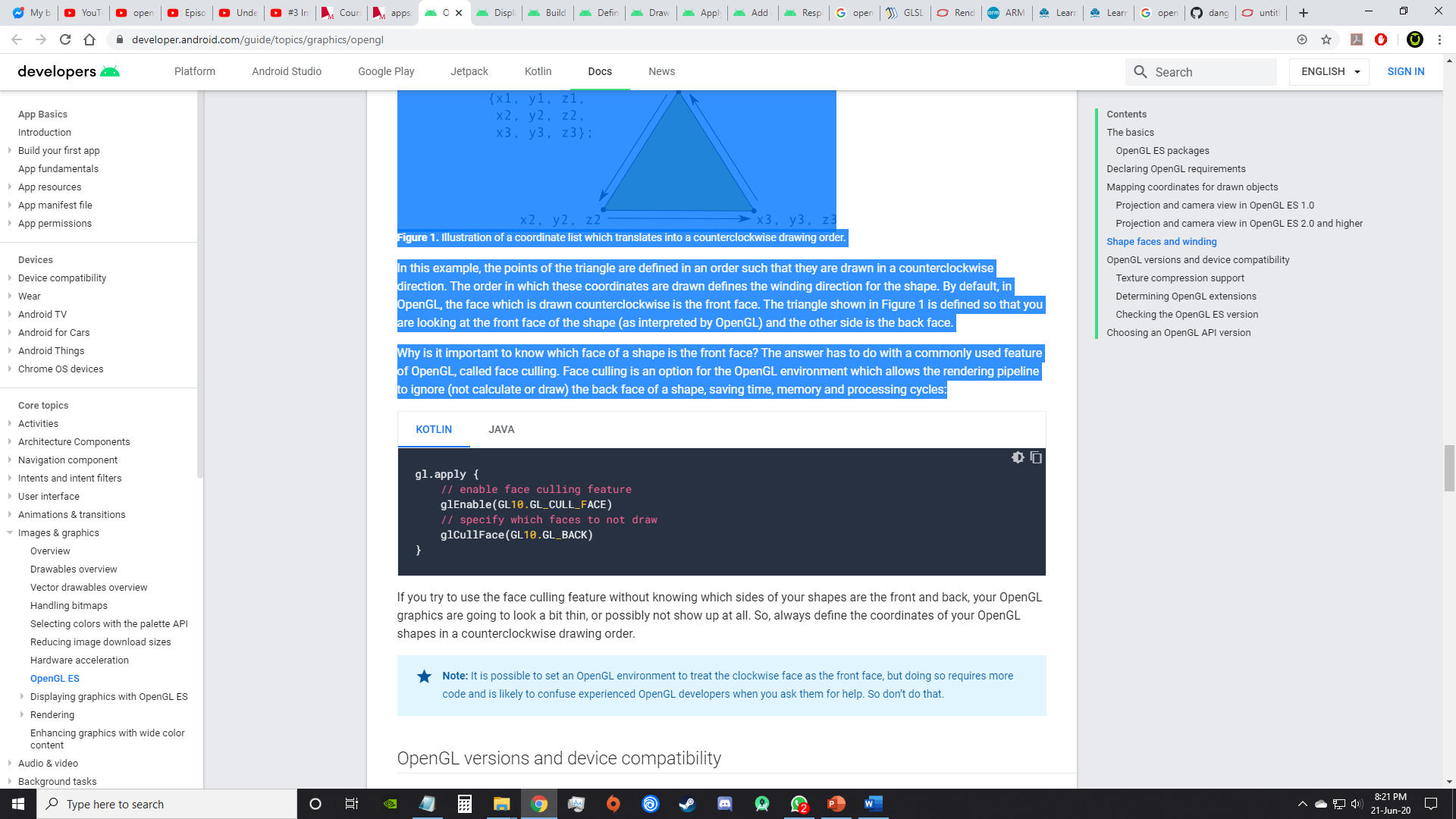
In this example, the points of the triangle are defined in an order such that they are drawn in a counterclockwise direction.

The order in which these coordinates are drawn defines the winding direction for the shape.

By default, in OpenGL, the face which is drawn counterclockwise is the front face

Why is it important to know which face of a shape is the front face?

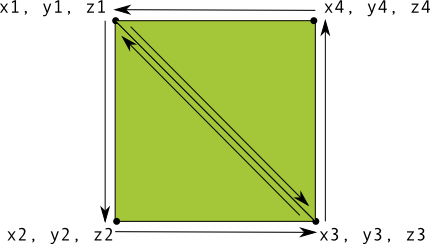
The answer has to do with a commonly used feature of OpenGL, called face culling. Face culling is an option for the OpenGL environment which allows the rendering pipeline to ignore (not calculate or draw) the back face of a shape, saving time, memory and processing cycles:



## **You can also define a square by defining two triangles with overlapping points.**

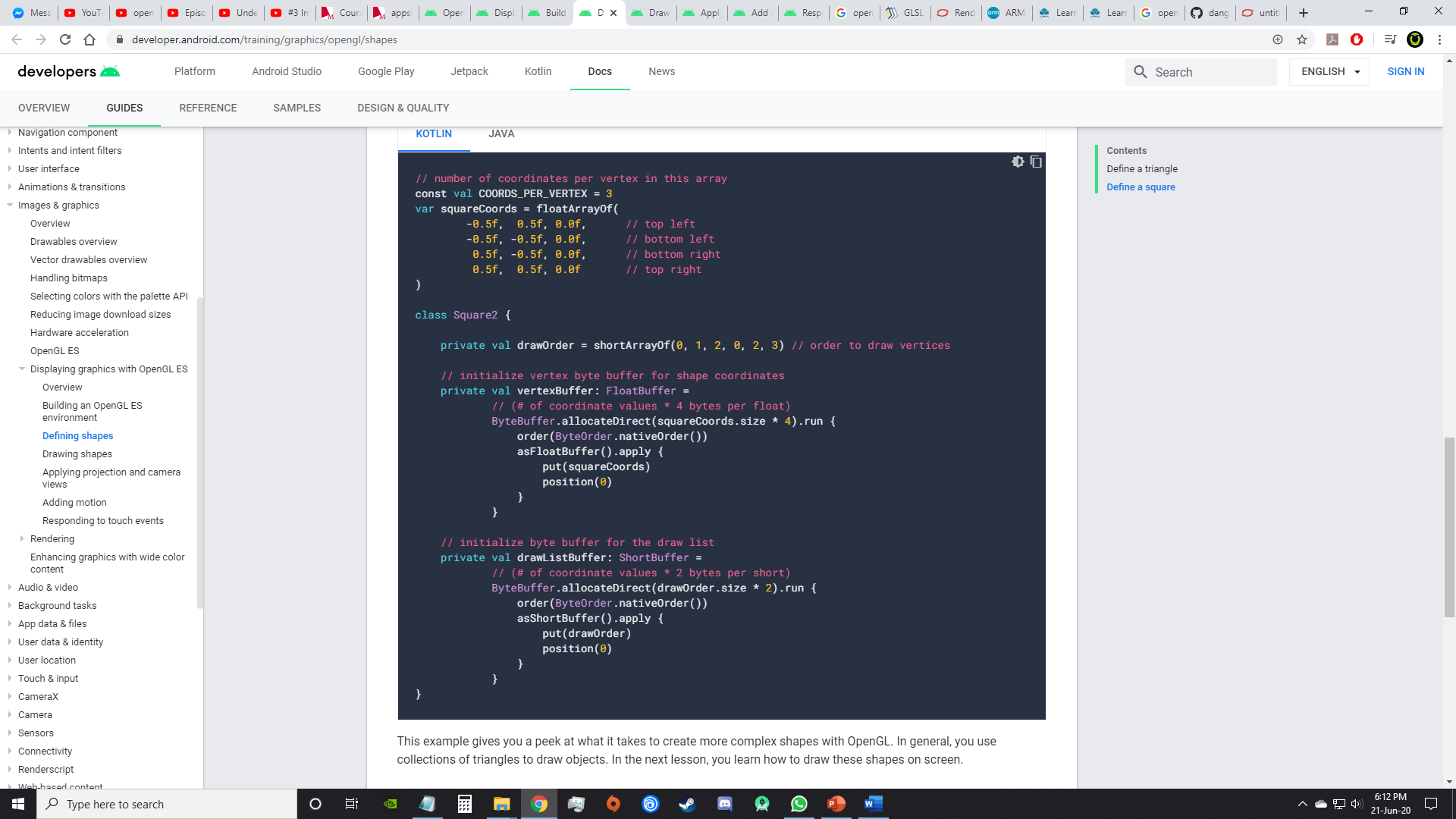
## **Define a square**

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:



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. In the next lesson, you learn how to draw these shapes on screen.

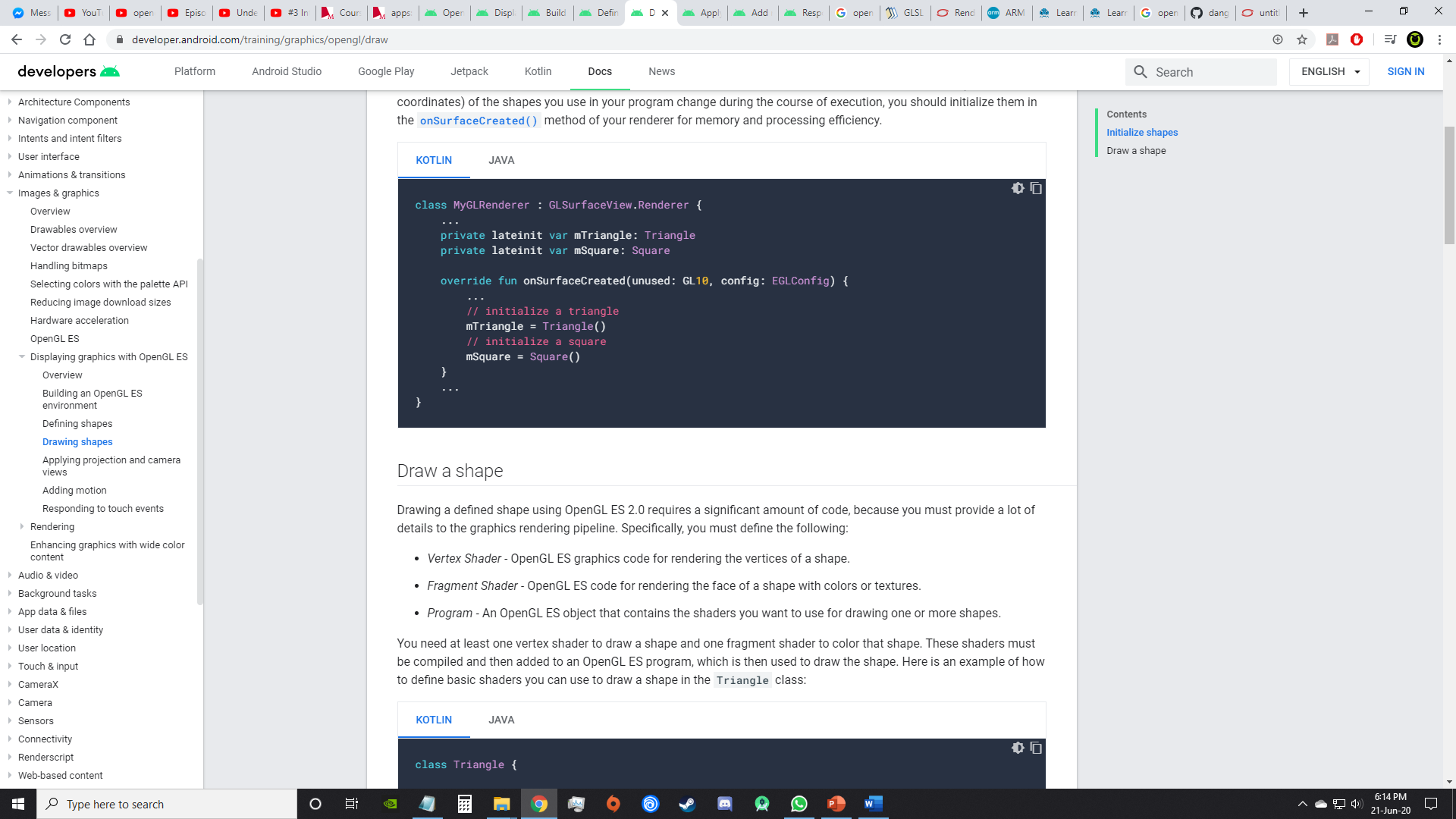
# **Draw shapes**

Once the shapes are defined, its time to draw them!

## **Initialize shapes**

But first you must initialize and load the shapes you plan to draw.

This is typically done in the [onSurfaceCreated()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer#onSurfaceCreated(javax.microedition.khronos.opengles.GL10,%20javax.microedition.khronos.egl.EGLConfig)) method of the renderer unless the structure changes during execution.

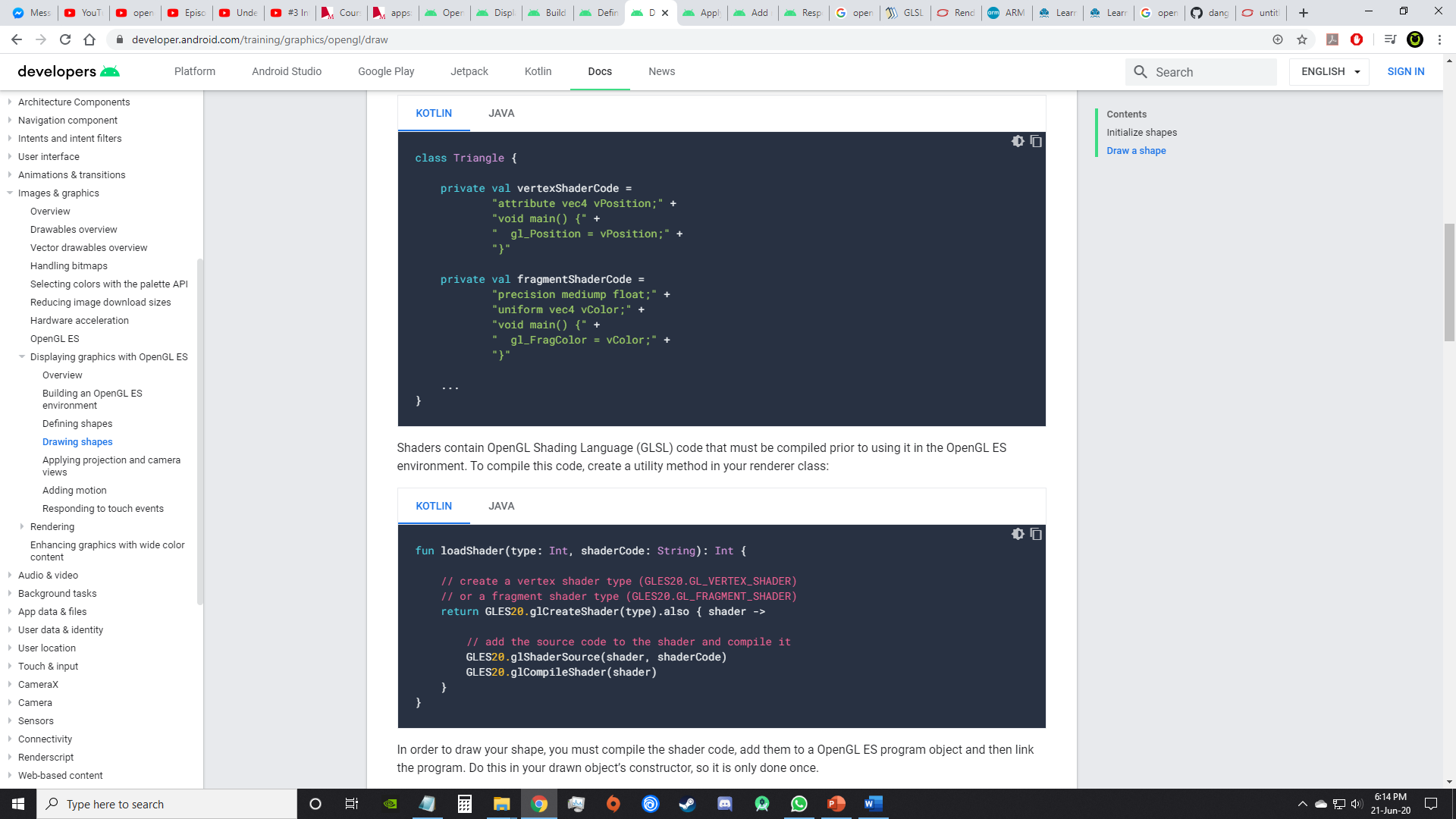


## **Draw a shape**

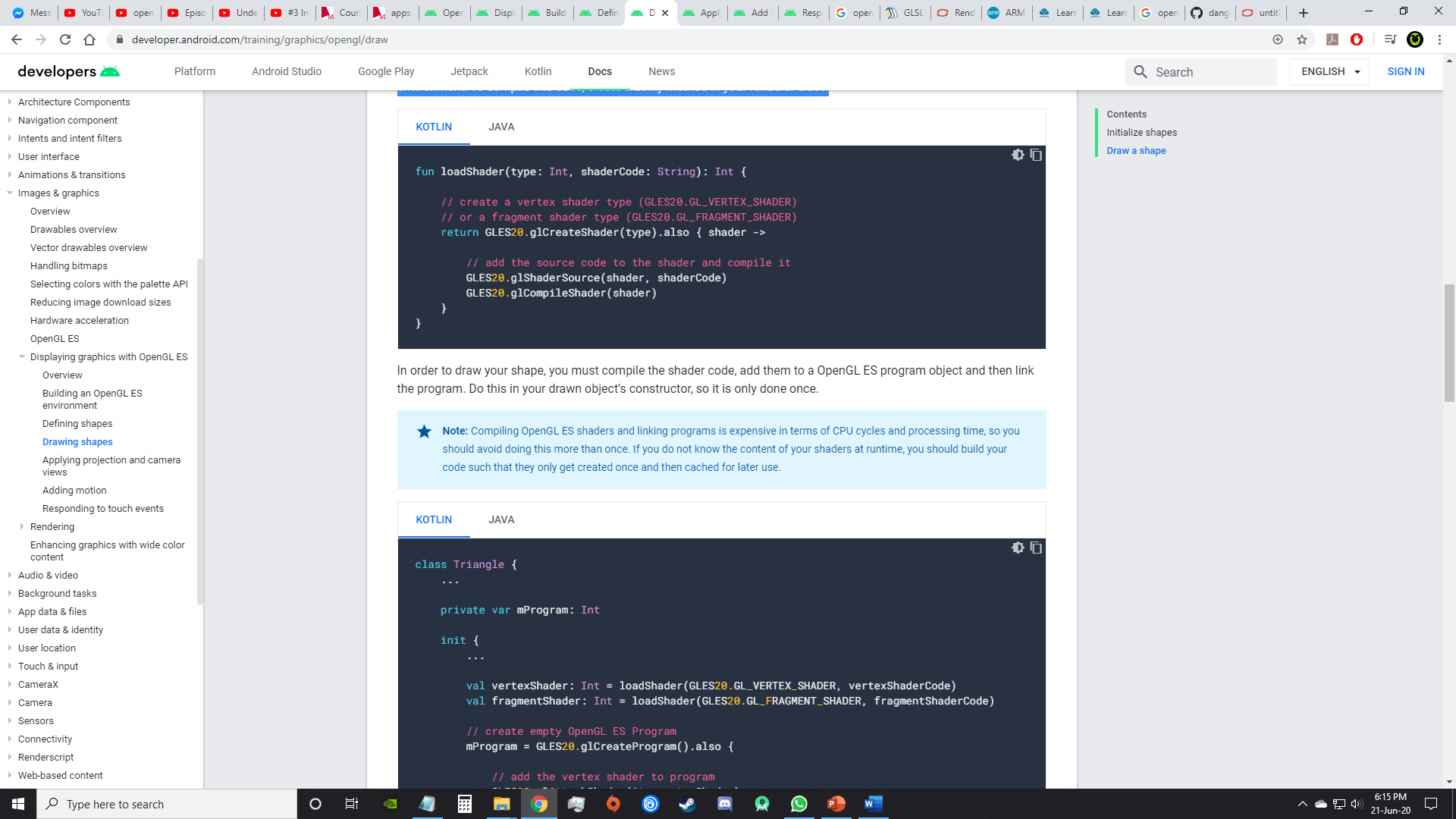
To correctly draw the shape, the following must be defined:

* *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.

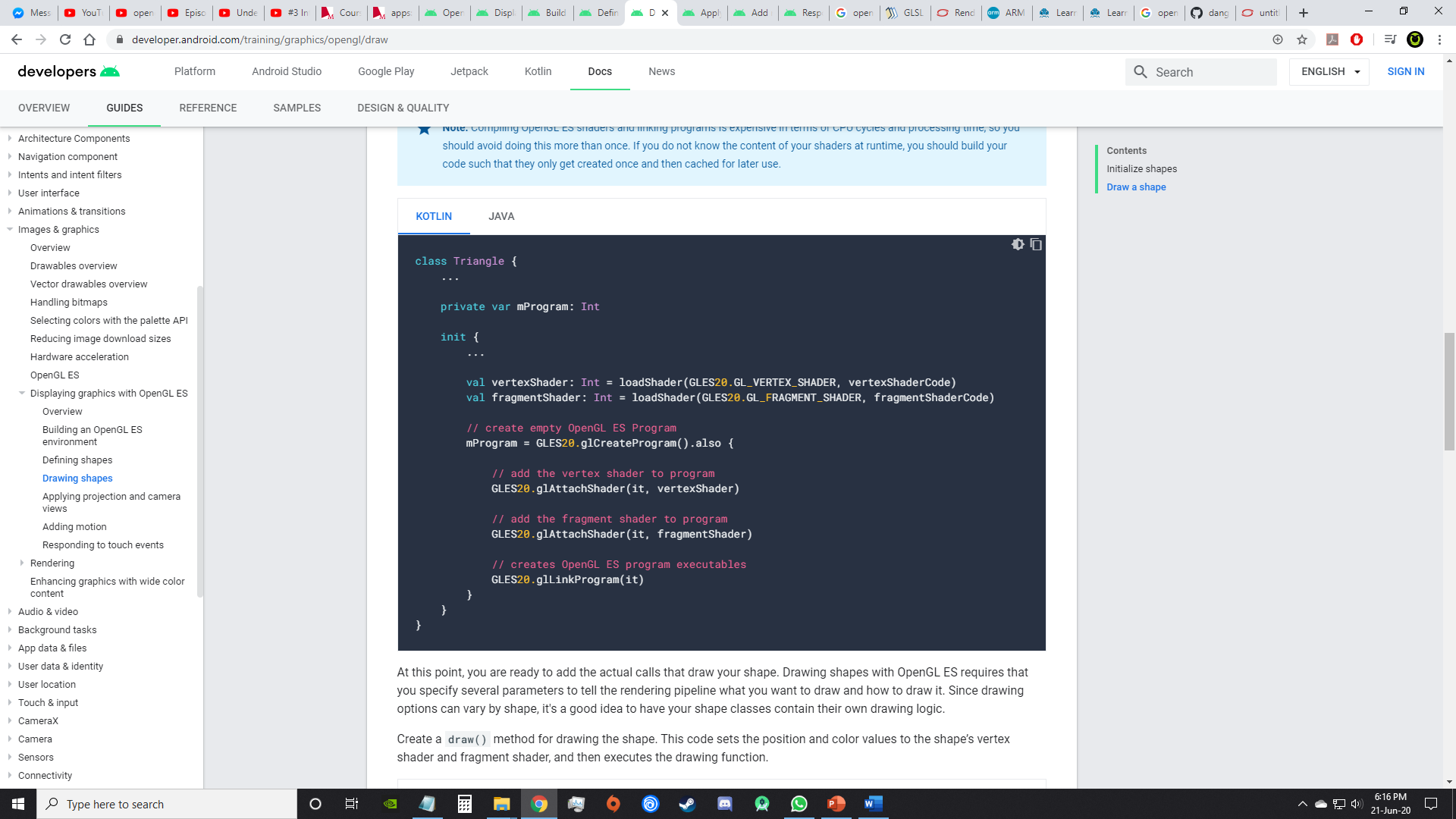
You need at least one vertex shader to draw a shape and one fragment shader to color that shape. These shaders must be compiled and then added to an OpenGL ES program, which is then used to draw the shape. Here is an example of how to define basic shaders you can use to draw a shape in the Triangle class:



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:



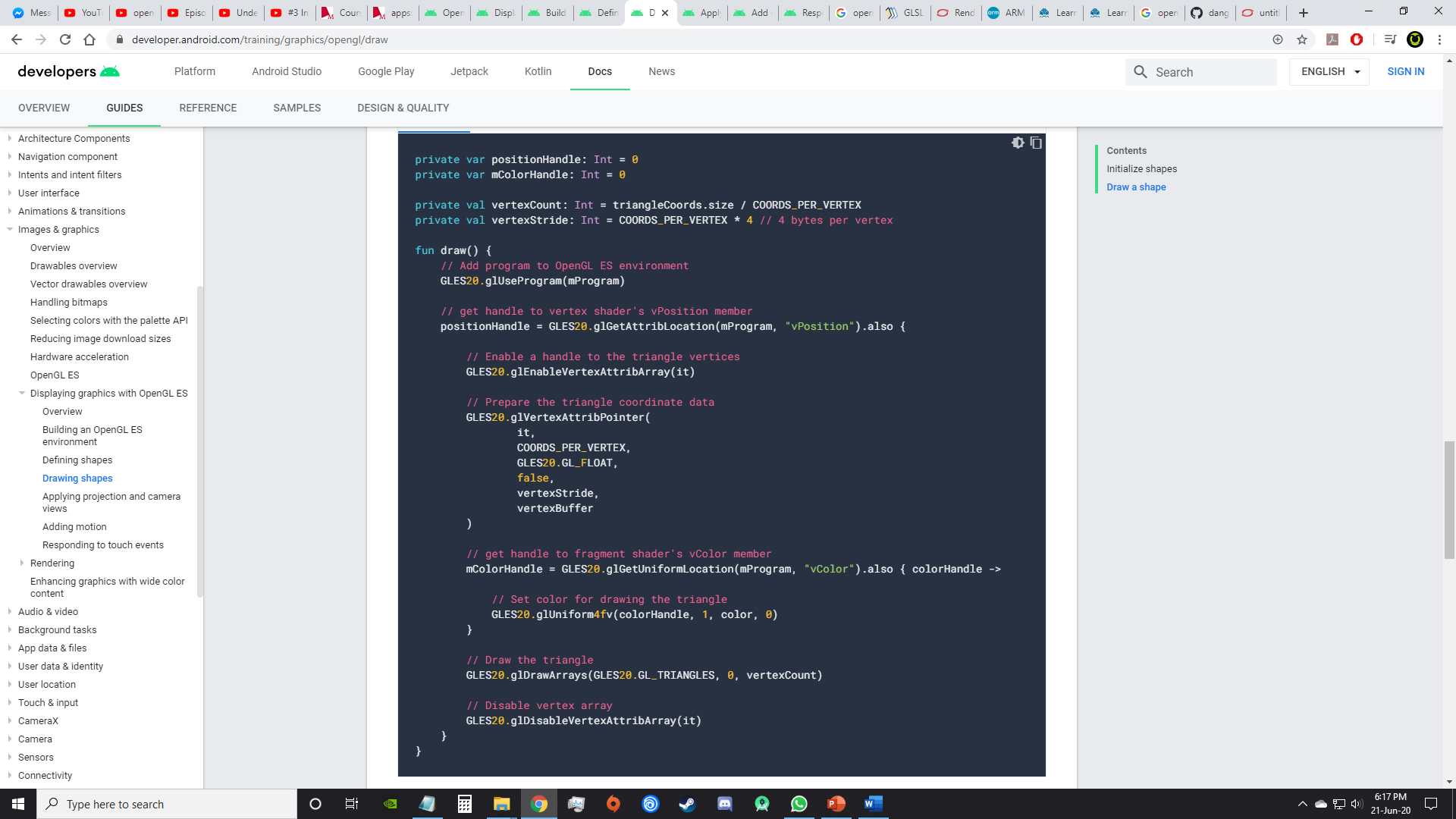
In order to draw your shape, you must compile the shader code, add them to an OpenGL ES program object and then link the program. Do this in your drawn object’s constructor, so it is only done once.



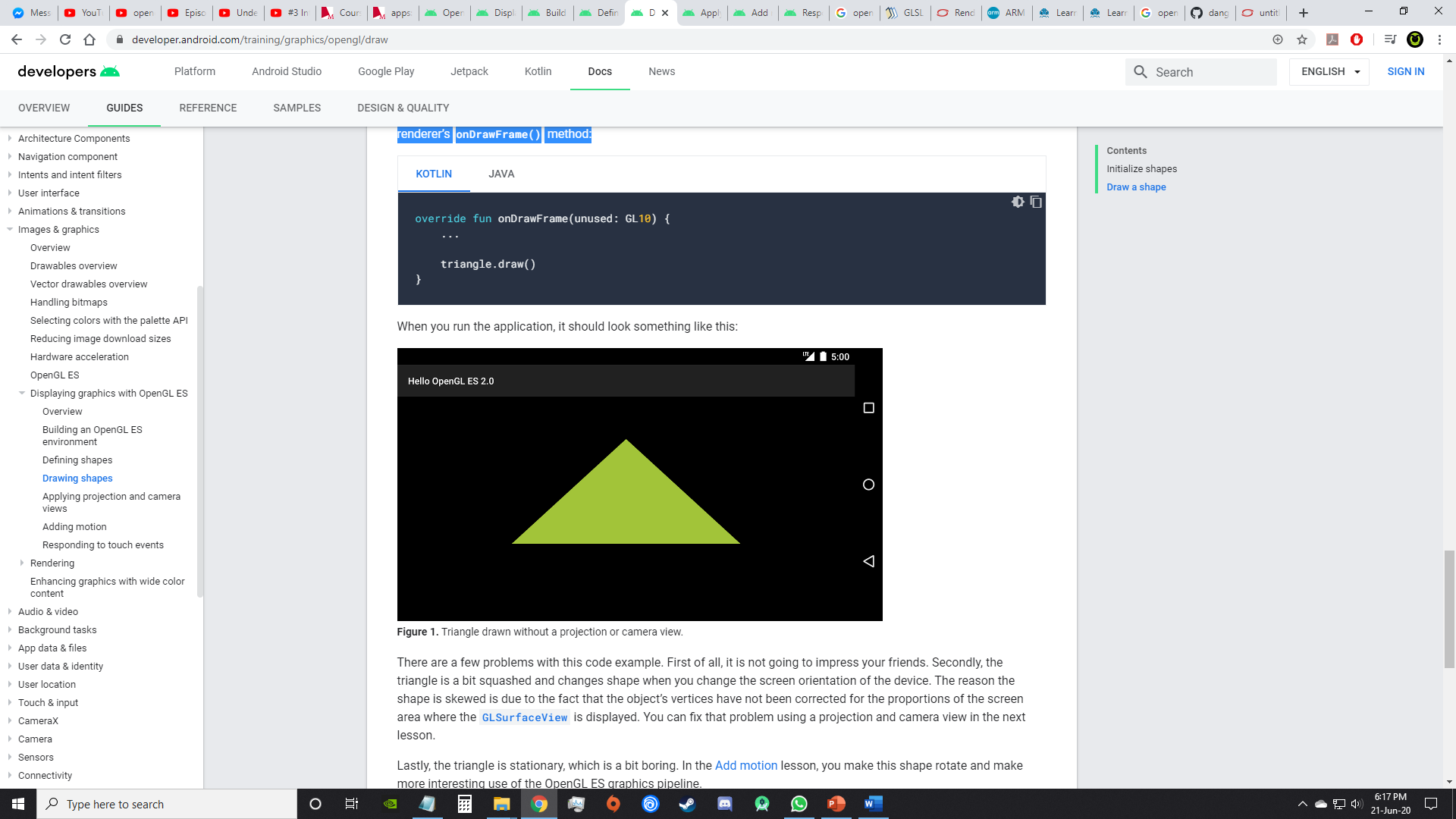
At this point, you are ready to add the actual calls that draw your shape. Drawing shapes with OpenGL ES requires that you specify several parameters to tell the rendering pipeline what you want to draw and how to draw it.

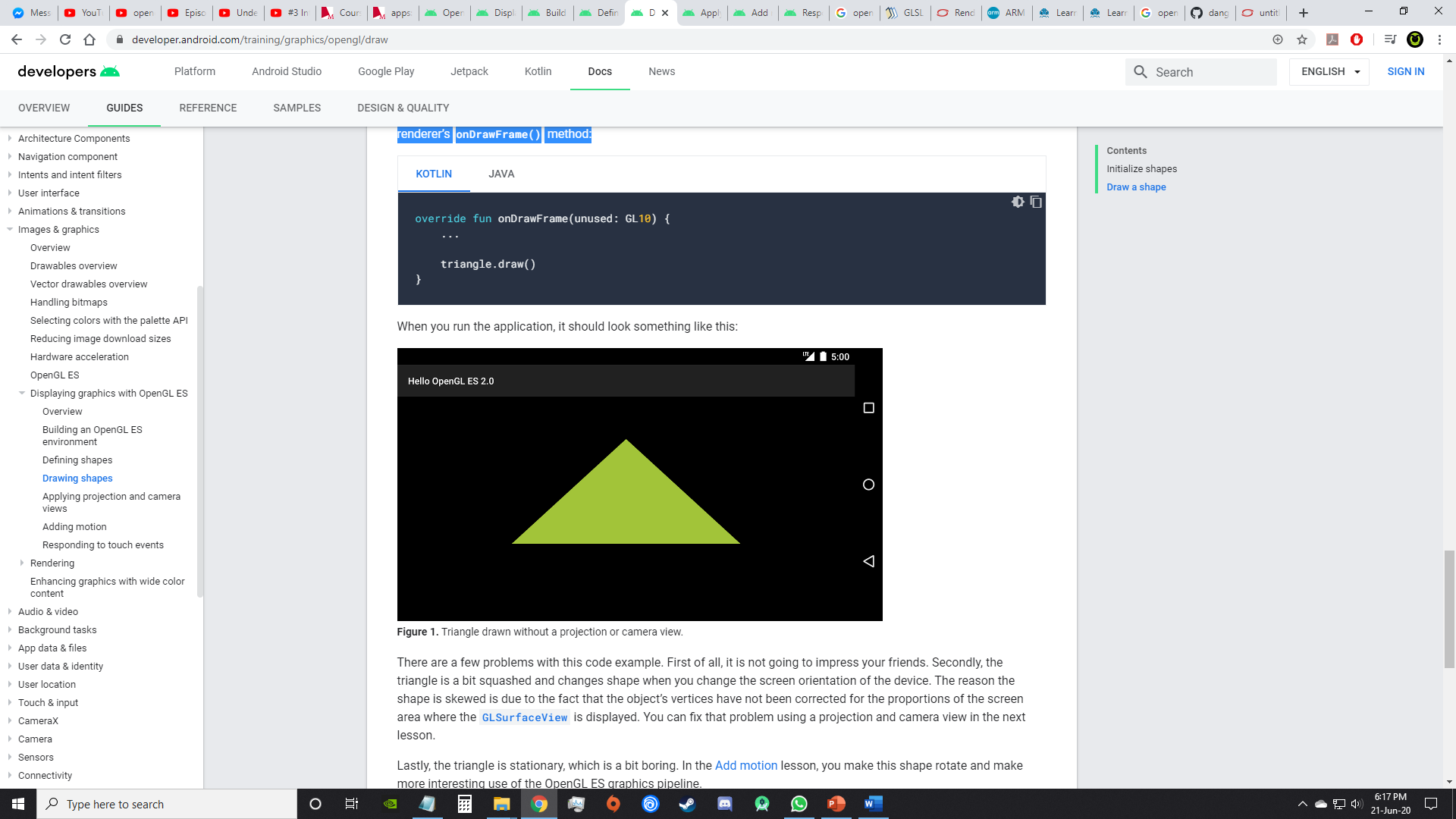
Since drawing options can vary by shape, it's a good idea to have your shape classes contain their own drawing logic.

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.



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:



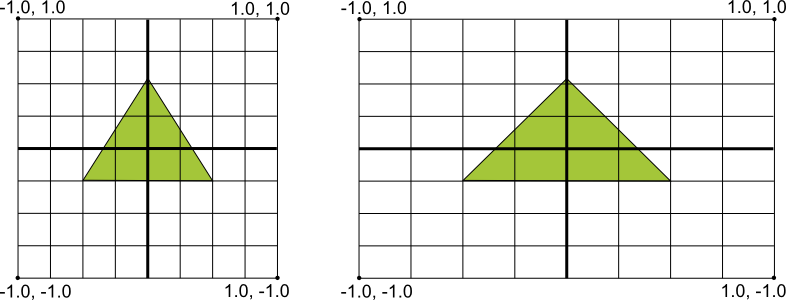


There are a few problems with this code example. First of all, it is not going to impress your friends. Secondly, the triangle is a bit squashed and changes shape when you change the screen orientation of the device. The reason the shape is skewed is due to the fact that the object’s vertices have not 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 in the next lesson.

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.

## **Mapping coordinates for drawn objects**

One of the basic problems in displaying graphics on Android devices is that their screens can vary in size and shape. OpenGL assumes a square, uniform coordinate system and, by default, happily draws those coordinates onto your typically non-square screen as if it is perfectly square.



**Figure 1.** Default OpenGL coordinate system (left) mapped to a typical Android device screen (right).

The illustration above shows the uniform coordinate system assumed for an OpenGL frame on the left, and how these coordinates actually map to a typical device screen in landscape orientation on the right. To solve this problem, you can apply OpenGL projection modes and camera views to transform coordinates so your graphic objects have the correct proportions on any display.

In order to apply projection and camera views, you create a projection matrix and a camera view matrix and apply them to the OpenGL rendering pipeline. The projection matrix recalculates the coordinates of your graphics so that they map correctly to Android device screens. The camera view matrix creates a transformation that renders objects from a specific eye position.

# **Apply projection and camera views**

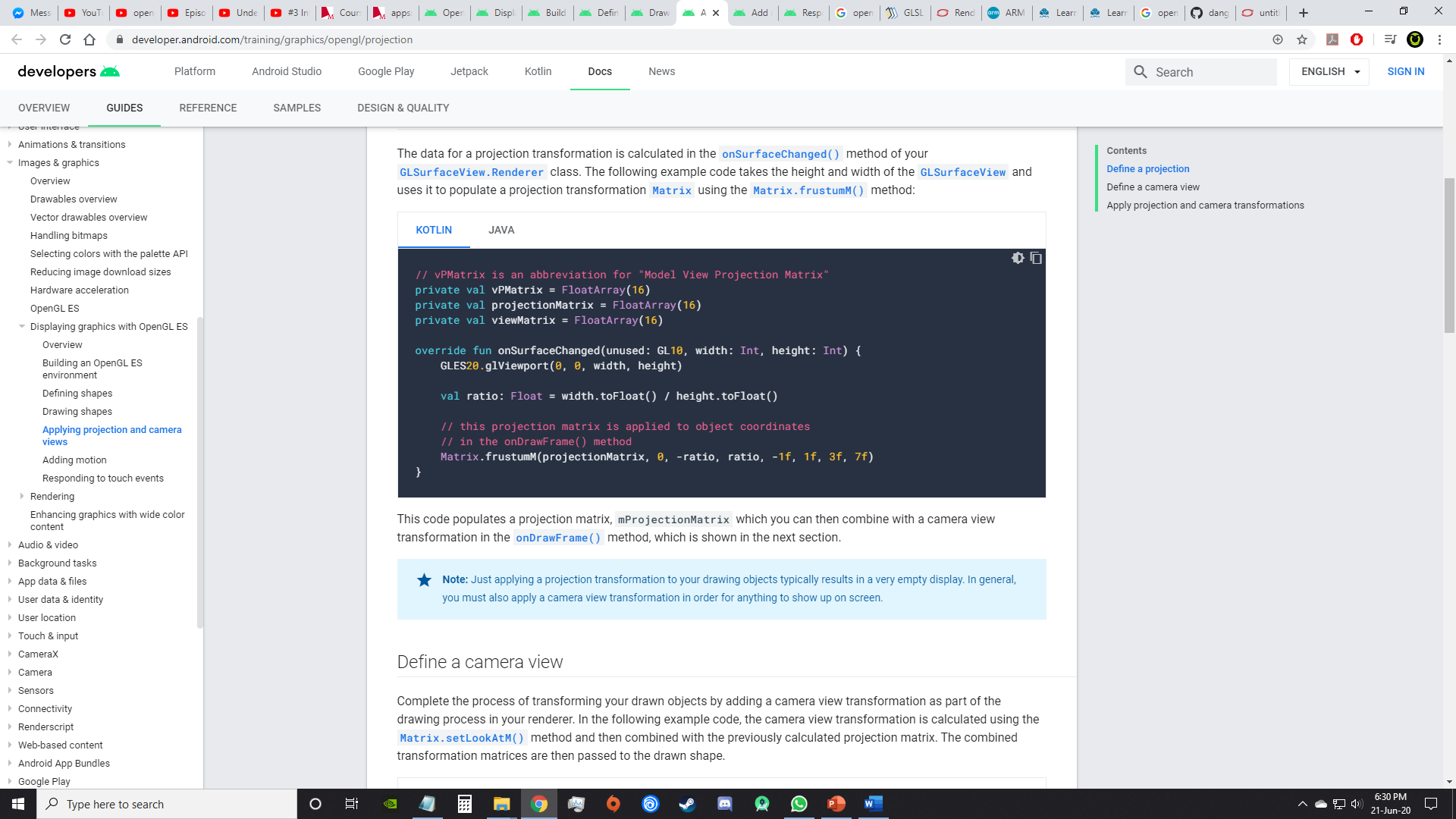
In the OpenGL ES environment, projection and camera views allow you to display drawn objects in a way that more closely resembles how you see physical objects with your eyes. This simulation of physical viewing is done with mathematical transformations of drawn object coordinates:

* *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. For more information about OpenGL ES projections and coordinate mapping, see [Mapping coordinates for drawn objects](https://developer.android.com/guide/topics/graphics/opengl#coordinate-mapping).
* *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.

This lesson describes how to create a projection and camera view and apply it to shapes drawn in your [GLSurfaceView](https://developer.android.com/reference/android/opengl/GLSurfaceView).

## **Define a projection**

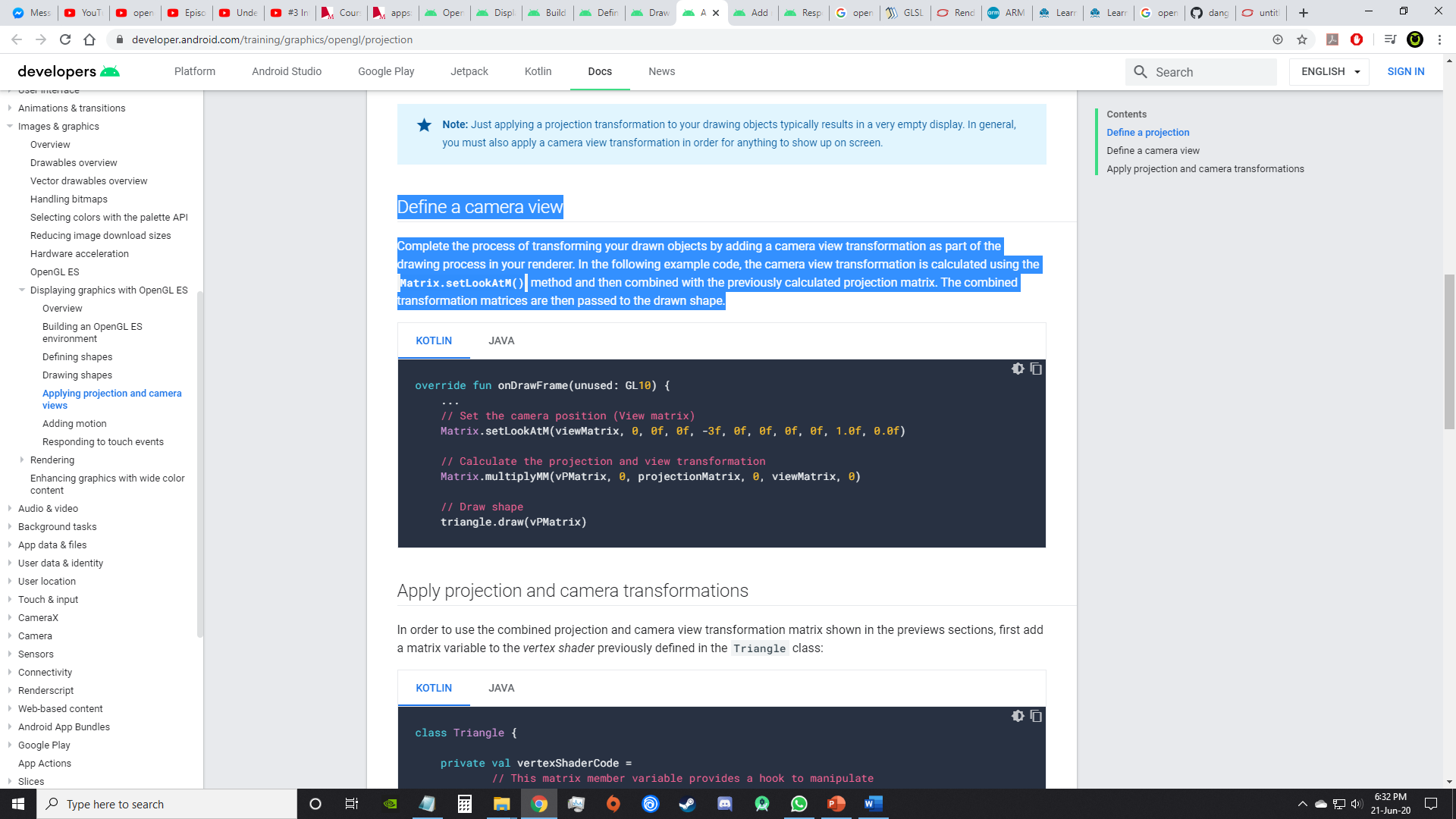
The data for a projection transformation is calculated in the [onSurfaceChanged()](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer" \l "onSurfaceChanged(javax.microedition.khronos.opengles.GL10,%20int,%20int)) method of your [GLSurfaceView.Renderer](https://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer) 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:



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, which is shown in the next section.

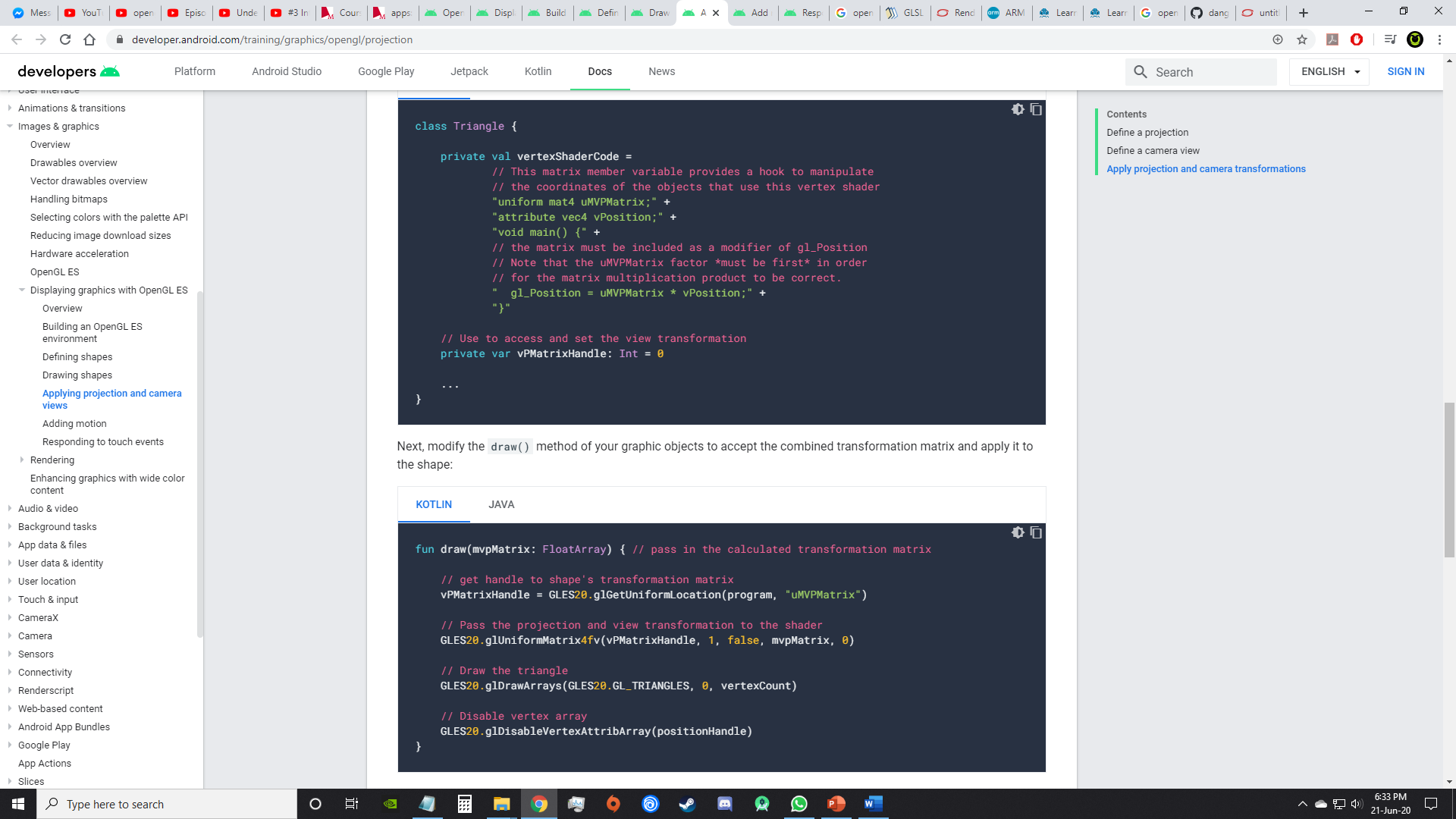
## **Define a camera view**

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" \l "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.

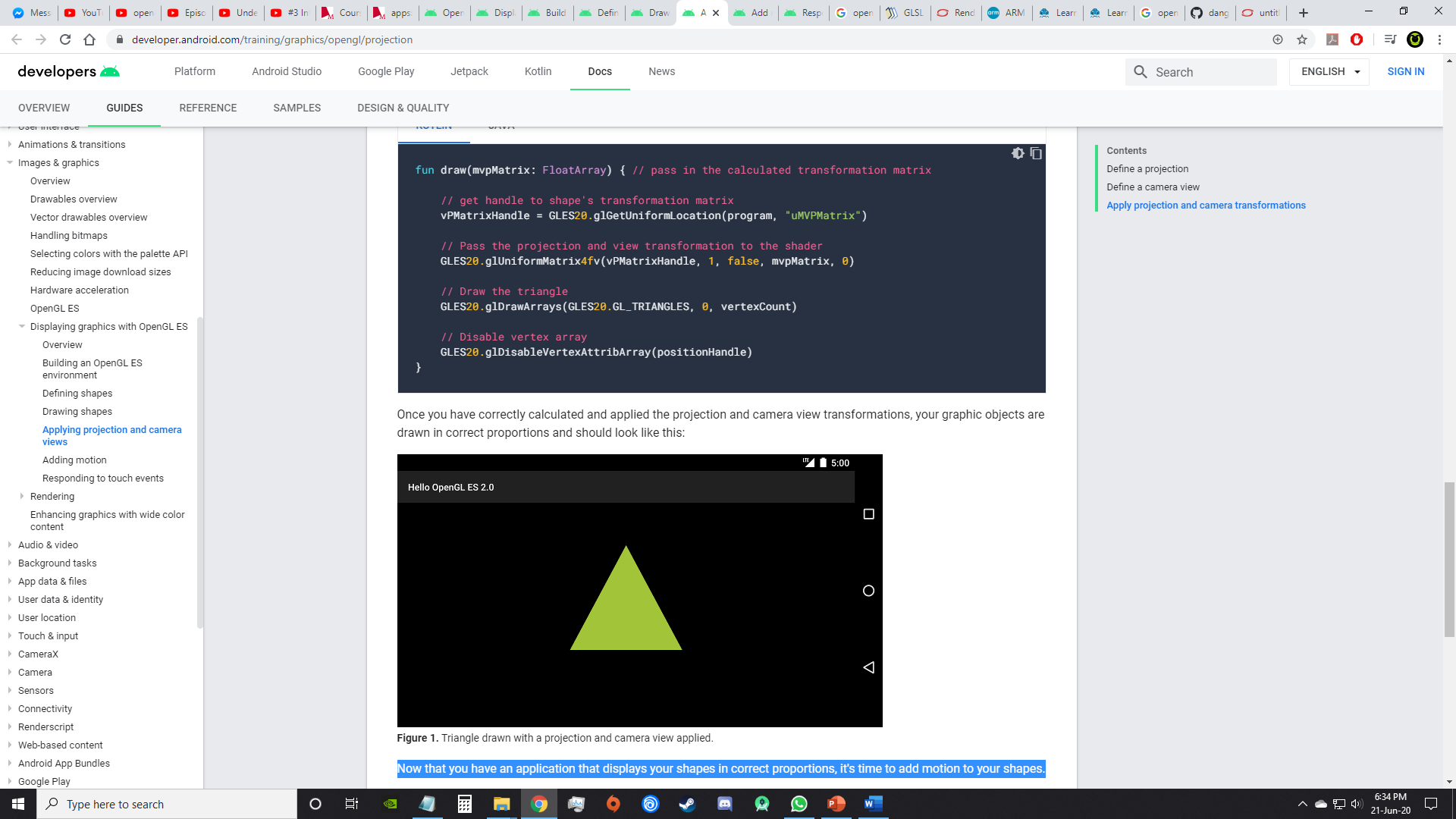


## **Apply projection and camera transformations**

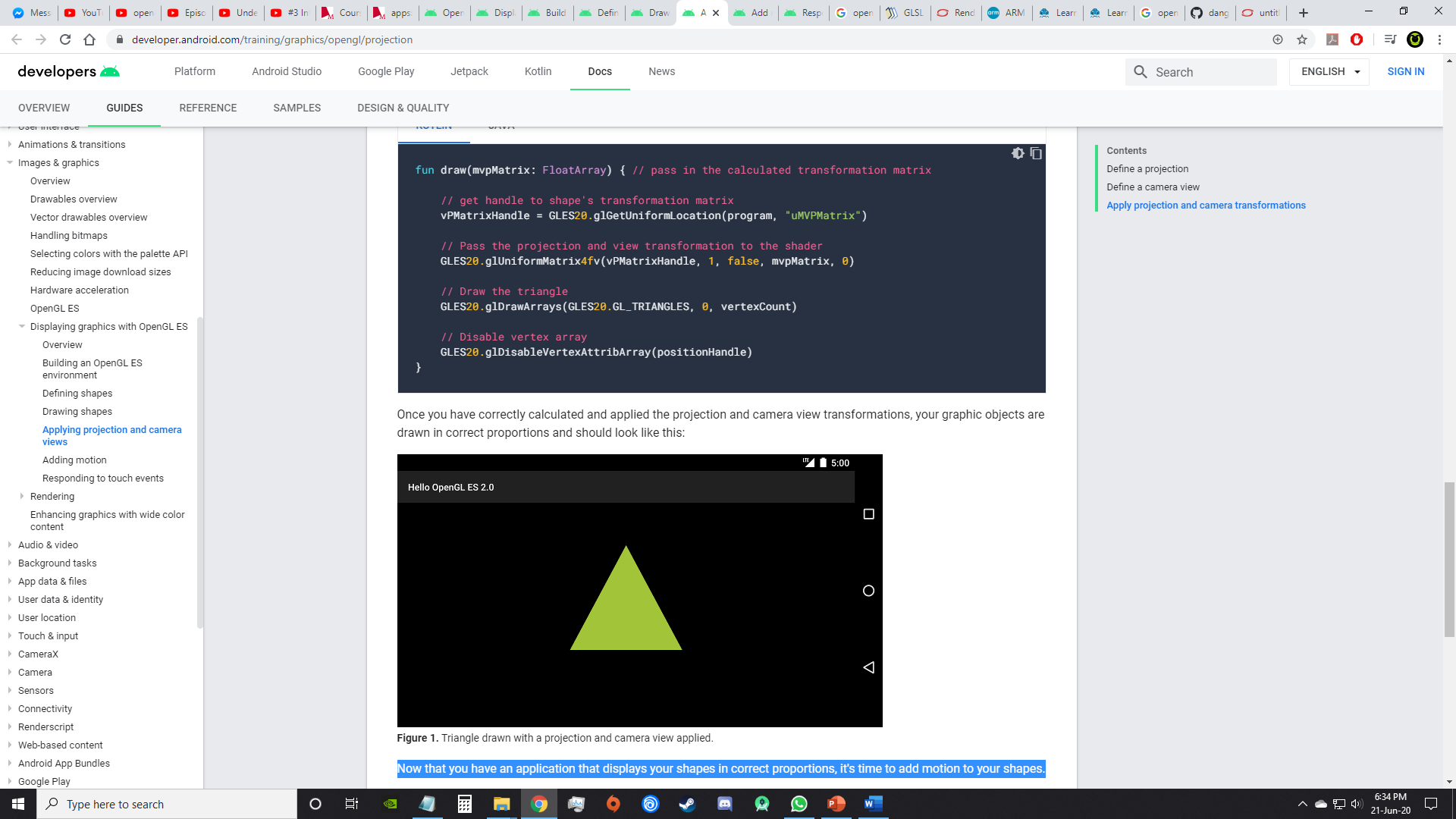
In order to use the combined projection and camera view transformation matrix shown in the previews sections, first add a matrix variable to the *vertex shader* previously defined in the Triangle class:



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



Once you have correctly calculated and applied the projection and camera view transformations, your graphic objects are drawn in correct proportions and should look like this:



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

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.

Code example?

References

<https://developer.android.com/guide/topics/graphics/opengl>

<https://en.wikipedia.org/wiki/OpenGL_ES>

<https://www.khronos.org/opengl/wiki/OpenGL_ES>

<https://www.khronos.org/opengles/>

<https://developer.arm.com/solutions/graphics-and-gaming/apis/opengl-es>

<https://developer.arm.com/docs/dui0555/latest/introduction/the-graphics-pipeline/opengl-es-graphics-pipeline-overview>

<https://www.youtube.com/watch?v=0PTBOX1HHIo>

<https://github.com/danginsburg/opengles3-book/>