[Android audio and video development and upgrade: OpenGL rendering video screen articles] 1. A preliminary understanding of OpenGL ES

jianshu.com/p/2158d4aec142

[Android audio and video development and upgrade: OpenGL rendering video screen] 1. Preliminary understanding of OpenGL ES

Table of contents

In this article you can learn

> This article mainly introduces the basic knowledge related to OpenGL, including coordinate systems, shaders, and basic rendering processes.

an introduction

When it comes to OpenGL, many people must say, I know this thing, it can be used to render 2D pictures and 3D models, and at the same time, they will say that OpenGL is difficult and very advanced, and I don't know how to use it.

1. Why does OpenGL "feel hard"?

- There are many and complex functions, and the rendering process is complex
- GLSL shader language is not easy to understand
- Process-oriented programming thinking is different from object-oriented programming thinking such as Java

2. What is OpenGL ES?

In order to solve the above problems and make OpenGL "not difficult to learn", it needs to be decomposed into some simple steps, and then the simple things are connected in series, and everything will be done.

First, let's take a look at what OpenGL is.

CPU and GPU

On a mobile phone, there are two major components, one is the CPU and the other is the GPU. There are two ways to display the graphical interface on the mobile phone. One is to use the CPU to render, and the other is to use the GPU to render. It can be said that GPU rendering is actually a kind of hardware acceleration.

Why the GPU can greatly improve the rendering speed, because the GPU is best at parallel floating-point operations, which can be used to perform parallel operations on many, many pixels.

OpenGL (Open Graphics Library) is a tool for indirectly operating the GPU. It is a set of defined cross-platform and cross-language graphics APIs. It is the underlying graphics library that can be used for 2D and 3D screen rendering. programming interface.

OpenGL and OpenGL ES

OpenGL ES full name: OpenGL for Embedded Systems, is a subset of OpenGL, designed for small devices such as mobile phone PAD, delete unnecessary methods, data types, functions, reduce volume, and optimize efficiency.

3. OpenGL ES version

The current major versions are 1.0/1.1/2.0/3.0/3.1

- 1.0: This API specification is supported on Android 1.0 and later
- 2.0: Not compatible with OpenGL ES 1.x. Android 2.2 (API 8) and higher versions support this API specification
- 3.0: Backward compatible with OpenGL ES 2.x. This API specification is supported on Android 4.3 (API 18) and higher
- 3.1: Backward compatible with OpenGL ES3.0/2.0. This API specification is supported on Android 5.0 (API 21) and higher

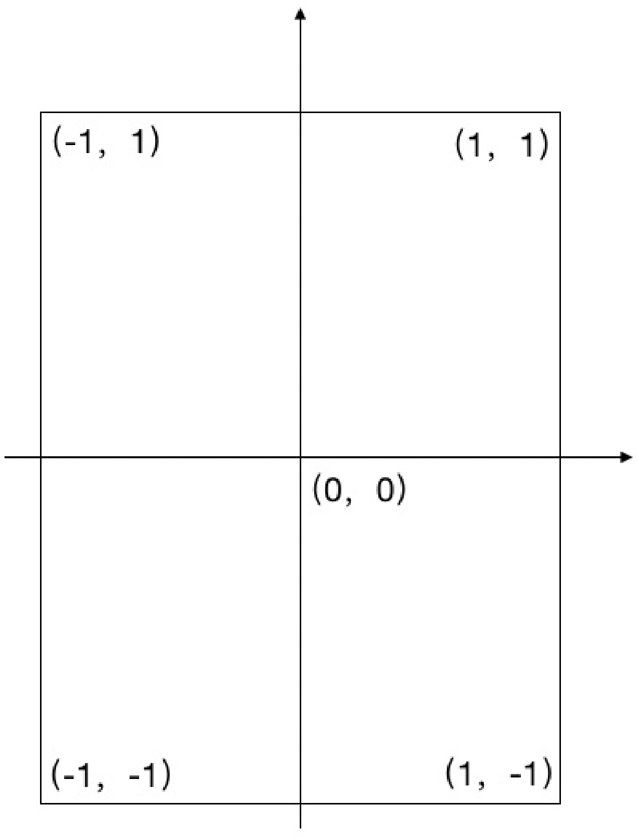
> Version 2.0 is the most widely supported version of Android at present, and the follow-up will mainly focus on this version for introduction and code writing.

2. OpenGL ES coordinate system

In audio and video development, there are mainly two coordinate systems involved: world coordinates and texture coordinates.

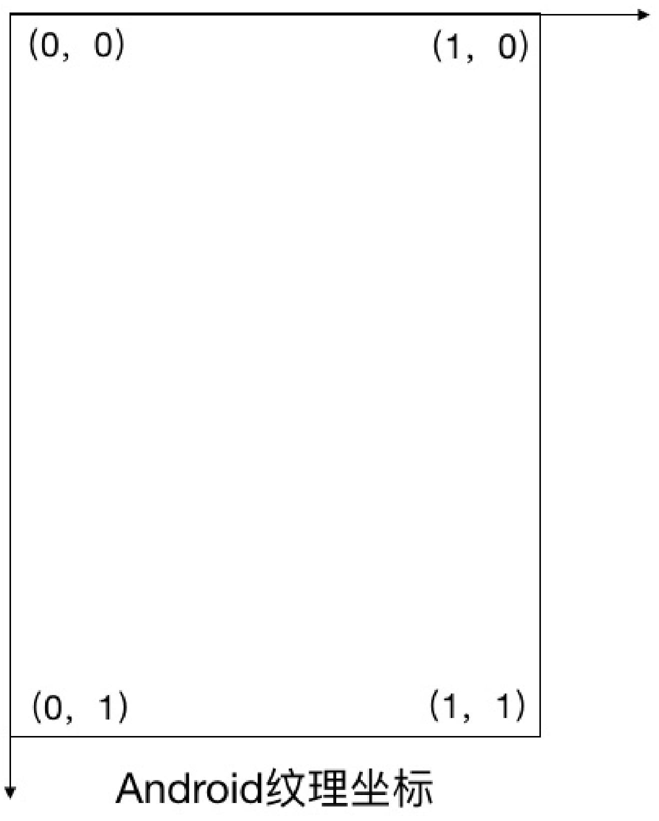
> Since 3D mapping is basically not involved, we only look at the x/y-axis coordinates, ignoring the z-axis coordinates. When it comes to 3D-related knowledge, you can Google it yourself, which is not within the scope of discussion.

First look at the two pictures:



世界坐标

world coordinates



texture coordinates

OpenGL ES world coordinates

You can know from the name that this is the coordinate of OpenGL's own world, a standardized coordinate system, the range is -1 to 1, and the origin is in the middle.

OpenGL ES texture coordinates

Texture coordinates are actually screen coordinates. The origin of the standard texture coordinates is at the bottom left of the screen, while the origin of the Android system coordinate system is at the top left. This is one point to be aware of when using OpenGL for Android.

Texture coordinates range from 0 to 1.

> Note: The xy-axis direction of the coordinate system is very important and determines how to do vertex coordinate and texture coordinate mapping.

So, what is the relationship between these two coordinate systems?

The world coordinate is the coordinate used for display, that is, where the pixel should be displayed is determined by the world coordinate.

Texture coordinates, which indicate where the color of the position specified by the world coordinates should be displayed on the texture. That is, where the color is located is determined by the texture coordinates.

Correct mapping is required between the two in order to display a picture normally.

3. OpenGL Shader Language GLSL

After OpenGL 2.0, a new programmable rendering pipeline has been added, which can control rendering more flexibly. But it also needs to learn another programming language for GPU, the syntax is similar to C language, called GLSL.

Vertex Shader & Fragment Shader

Before introducing GLSL, let's look at two relatively unfamiliar terms: vertex shader and fragment shader.

A shader is a small program that can run on the GPU and is written in the GLSL language. Namely, a vertex shader is a program for manipulating vertices, and a fragment shader is a program for manipulating pixel color properties.

> Simple understanding: In fact, it corresponds to the above two coordinate systems: the vertex shader corresponds to the world coordinates, and the fragment shader corresponds to the texture coordinates.

For each point on the screen, the program fragments in the vertex and fragment shaders are executed once, and they are executed in parallel, and finally rendered to the screen.

GLSL programming

Let's briefly introduce the GLSL language through a simplest vertex shader and fragment shader

```
#顶点着色器

attribute vec4 aPosition;

void main() {
  gl_Position = aPosition;
}
```

```
#片元着色器

void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0)
}
```

First of all, you can see that the GLSL language is a C-like language. The framework of the shader is basically the same as that of the C language. Variables are declared at the top, followed by the main function. In the shader, there are several built-in variables that can be used directly (only the ones commonly used in audio and video development are listed here, and some other 3D development will use them):

Built-in input variables for vertex shaders

gl_Position: vertex coordinates
gl_PointSize: the size of the point, the default value is 1 if there is no assignment

Fragment shader built-in output variables

gl_FragColor: current fragment color

Look back at the shader code above.

1) In the vertex shader, the vertex coordinate xyzw of a vec4 is passed in, and then directly passed to the built-in variable gl_Position, that is, rendering directly according to the vertex coordinate, without performing position transformation.

> Note: The vertex coordinates are passed in the Java code, which will be mentioned later. In addition, w is a homogeneous coordinate, and 2D rendering has no effect.

2) In the fragment shader, directly assign a value to gl_FragColor, which is still a vec4 type of data, which represents the rgba color value, which is red.

You can see that vec4 is a 4-dimensional vector, which can be used to represent coordinates xyzw, rgba, and of course vec3, vec2, etc. You can refer to this article: Shader Language GLSL , which is very detailed, it is recommended to take a look.

In this way, after two simple shaders are connected in series, each vertex (pixel) will display a red dot, and finally the screen will display a red picture.

> The specific GLSL data types and syntax will not be introduced, and the GLSL code involved later will be explained in more depth. For more details, please refer to the author's article [Shader Language GLSL], which is very detailed.

Fourth, Android OpenGL ES rendering process

To be honest, the rendering process of OpenGL is relatively cumbersome, and it is also a place that makes many people daunting. However, if it comes down to it, the entire rendering process is basically fixed. As long as it is packaged according to the fixed process, it is actually not that complicated.

Next, we will enter the actual combat and uncover the mystery of OpenGL layer by layer.

1. Initialization

In Android, OpenGL is usually used with GLSurfaceView. In GLSurfraceView, Google has encapsulated the basic process of rendering.

> It needs to be emphasized here that OpenGL is a state machine based on threads. Operations related to OpenGL, such as creating texture IDs, initialization, rendering, etc., must be completed in the same thread, otherwise an exception will be caused.

Usually developers don't have a deep understanding of this mechanism when they first come into contact with OpenGL, because Google has already done this part of the content for developers in GLSurfaceView. This is a very important aspect of OpenGL, and we will continue to learn more about it in subsequent articles on EGL.

1. New page

```
class SimpleRenderActivity : AppCompatActivity() {
    //自定义的OpenGL渲染器，详情请继续往下看
    lateinit var drawer: IDrawer

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_simpler_render)

        drawer = if (intent.getIntExtra("type", 0) == 0) {
            TriangleDrawer()
        } else {
            BitmapDrawer(BitmapFactory.decodeResource(CONTEXT!!.resources,
R.drawable.cover))
        }
        initRender(drawer)
    }

    private fun initRender(drawer: IDrawer) {
        gl_surface.setEGLContextClientVersion(2)
        gl_surface.setRenderer(SimpleRender(drawer))
    }

    override fun onDestroy() {
        drawer.release()
        super.onDestroy()
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <android.opengl.GLSurfaceView
            android:id="@+id/gl_surface"
            android:layout_width="match_parent"
            android:layout_height="match_parent"/>
</android.support.constraint.ConstraintLayout>
```

The page is very simple. A full-screen GLSurfaceView is placed. When initializing, the version used by OpenGL is set to 2.0, and then the renderer SimpleRender is configured, which inherits from GLSurfaceView.Renderer

> IDrawer will explain in detail when drawing triangles. The interface class is defined only for the convenience of expansion, and the rendering code can also be written directly in SimpleRender.

2. Implement the rendering interface

```
class SimpleRender(private val mDrawer: IDrawer): GLSurfaceView.Renderer {

    override fun onSurfaceCreated(gl: GL10?, config: EGLConfig?) {
        GLES20.glClearColor(0f, 0f, 0f, 0f)
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT)
        mDrawer.setTextureID(OpenGLTools.createTextureIds(1)[0])
    }

    override fun onSurfaceChanged(gl: GL10?, width: Int, height: Int) {
        GLES20.glViewport(0, 0, width, height)
    }

    override fun onDrawFrame(gl: GL10?) {
        mDrawer.draw()
    }
}
```

Note that three callback interfaces are implemented. These three interfaces are the exposed interfaces in the process encapsulated by Google, which are left for developers to implement initialization and rendering, and the callbacks of these three interfaces are all in the same thread. .

> In onSurfaceCreated, two OpenGL ES codes are called to clear the screen, and the clear screen color is black.

```
GLES20.glClearColor(0f, 0f, 0f, 0f)
GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT)
```

At the same time, a texture ID is created and set to the Drawer as follows:

```
fun createTextureIds(count: Int): IntArray {
    val texture = IntArray(count)
    GLES20.glGenTextures(count, texture, 0) //生成纹理
    return texture
}
```

> In onSurfaceChanged, call glViewport to set the width, height and position of the area drawn by OpenGL

> The drawing area mentioned here refers to the drawing area of OpenGL in GLSurfaceView, which is generally fully covered.

```
GLES20.glViewport(0, 0, width, height)
```

> In onDrawFrame, it is the place where the real drawing is realized. The interface will keep calling back to refresh the drawing area. A simple triangle drawing is used here to illustrate the entire drawing process.

2. Render a simple triangle

First define a rendering interface class:

```
interface IDrawer {
    fun draw()
    fun setTextureID(id: Int)
    fun release()
}
```

```
class TriangleDrawer(private val mTextureId: Int = -1): IDrawer {
    //顶点坐标
    private val mVertexCoors = floatArrayOf(
        -1f, -1f,
         1f, -1f,
         0f,  1f
    )

    //纹理坐标
    private val mTextureCoors = floatArrayOf(
        0f,   1f,
        1f,   1f,
        0.5f, 0f
    )

    //纹理ID
    private var mTextureId: Int = -1

    //OpenGL程序ID
    private var mProgram: Int = -1

    // 顶点坐标接收者
    private var mVertexPosHandler: Int = -1
    // 纹理坐标接收者
    private var mTexturePosHandler: Int = -1

    private lateinit var mVertexBuffer: FloatBuffer
    private lateinit var mTextureBuffer: FloatBuffer

    init {
        //【步骤1: 初始化顶点坐标】
        initPos()
    }

    private fun initPos() {
        val bb = ByteBuffer.allocateDirect(mVertexCoors.size * 4)
        bb.order(ByteOrder.nativeOrder())
        //将坐标数据转换为FloatBuffer，用以传入给OpenGL ES程序
        mVertexBuffer = bb.asFloatBuffer()
        mVertexBuffer.put(mVertexCoors)
        mVertexBuffer.position(0)

        val cc = ByteBuffer.allocateDirect(mTextureCoors.size * 4)
        cc.order(ByteOrder.nativeOrder())
        mTextureBuffer = cc.asFloatBuffer()
        mTextureBuffer.put(mTextureCoors)
        mTextureBuffer.position(0)
    }

    override fun setTextureID(id: Int) {
        mTextureId = id
    }

    override fun draw() {
        if (mTextureId != -1) {
            //【步骤2: 创建、编译并启动OpenGL着色器】
            createGLPrg()
            //【步骤3: 开始渲染绘制】
            doDraw()
        }
    }

    private fun createGLPrg() {
        if (mProgram == -1) {
            val vertexShader = loadShader(GLES20.GL_VERTEX_SHADER,
getVertexShader())
            val fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER,
getFragmentShader())

            //创建OpenGL ES程序，注意：需要在OpenGL渲染线程中创建，否则无法渲染
            mProgram = GLES20.glCreateProgram()
            //将顶点着色器加入到程序
            GLES20.glAttachShader(mProgram, vertexShader)
            //将片元着色器加入到程序中
            GLES20.glAttachShader(mProgram, fragmentShader)
            //连接到着色器程序
            GLES20.glLinkProgram(mProgram)

            mVertexPosHandler = GLES20.glGetAttribLocation(mProgram, "aPosition")
            mTexturePosHandler = GLES20.glGetAttribLocation(mProgram,
"aCoordinate")
        }
        //使用OpenGL程序
        GLES20.glUseProgram(mProgram)
    }

    private fun doDraw() {
        //启用顶点的句柄
        GLES20.glEnableVertexAttribArray(mVertexPosHandler)
        GLES20.glEnableVertexAttribArray(mTexturePosHandler)
        //设置着色器参数
        GLES20.glVertexAttribPointer(mVertexPosHandler, 2, GLES20.GL_FLOAT, false,
0, mVertexBuffer)
        GLES20.glVertexAttribPointer(mTexturePosHandler, 2, GLES20.GL_FLOAT,
false, 0, mTextureBuffer)
        //开始绘制
        GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, 4)
    }

    override fun release() {
        GLES20.glDisableVertexAttribArray(mVertexPosHandler)
        GLES20.glDisableVertexAttribArray(mTexturePosHandler)
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, 0)
        GLES20.glDeleteTextures(1, intArrayOf(mTextureId), 0)
        GLES20.glDeleteProgram(mProgram)
    }

    private fun getVertexShader(): String {
        return "attribute vec4 aPosition;" +
                "void main() {" +
                "  gl_Position = aPosition;" +
                "}"
    }

    private fun getFragmentShader(): String {
        return "precision mediump float;" +
                "void main() {" +
                "  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);" +
                "}"
    }

    private fun loadShader(type: Int, shaderCode: String): Int {
        //根据type创建顶点着色器或者片元着色器
        val shader = GLES20.glCreateShader(type)
        //将资源加入到着色器中，并编译
        GLES20.glShaderSource(shader, shaderCode)
        GLES20.glCompileShader(shader)

        return shader
    }
}
```

Even though it just draws a simple triangle, the code still looks complicated. Here it is disassembled into three steps, which is more clear.

1) Initialize vertex coordinates

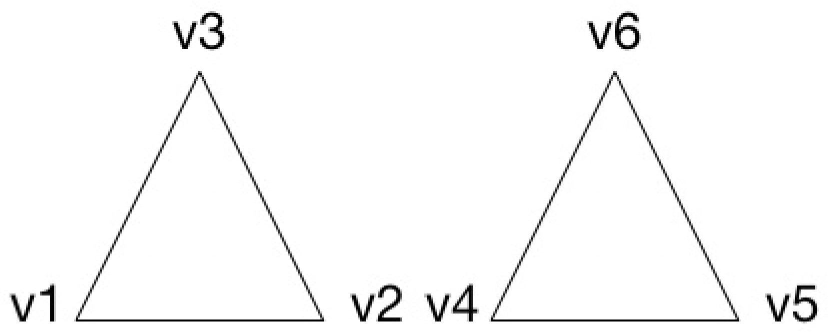
Earlier we talked about the world coordinates and texture coordinates of OpenGL, and these two coordinates need to be determined before drawing.

【important hint】

One thing that hasn't been said is that all OpenGL ES pictures are composed of triangles . For example, a quadrilateral is composed of two triangles. Other more complex graphics can also be divided into large and small triangles.

Therefore, the vertex coordinates are also set according to the connection of the triangles. There are three ways to draw it:

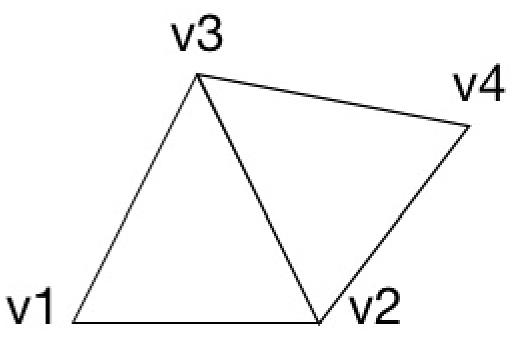
GL_TRIANGLES: constituent triangles of independent vertices



GL_TRIANGLES

GL_TRIANGLES

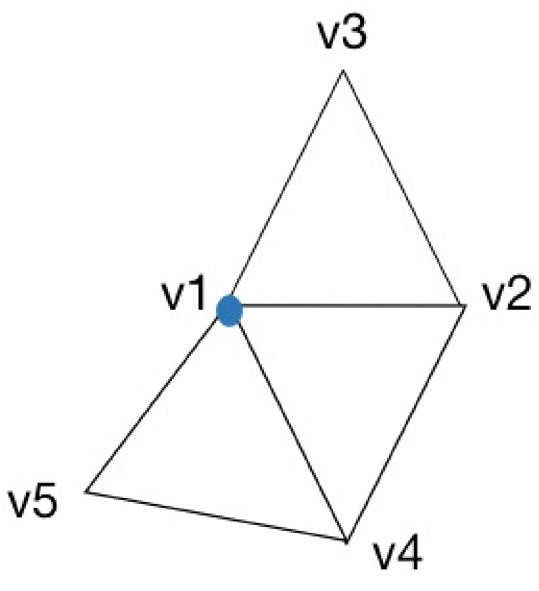
GL_TRIANGLE_STRIP: reuse vertices to form triangles



GL_TRIANGLE_STRIP

GL_TRIANGLE_STRIP

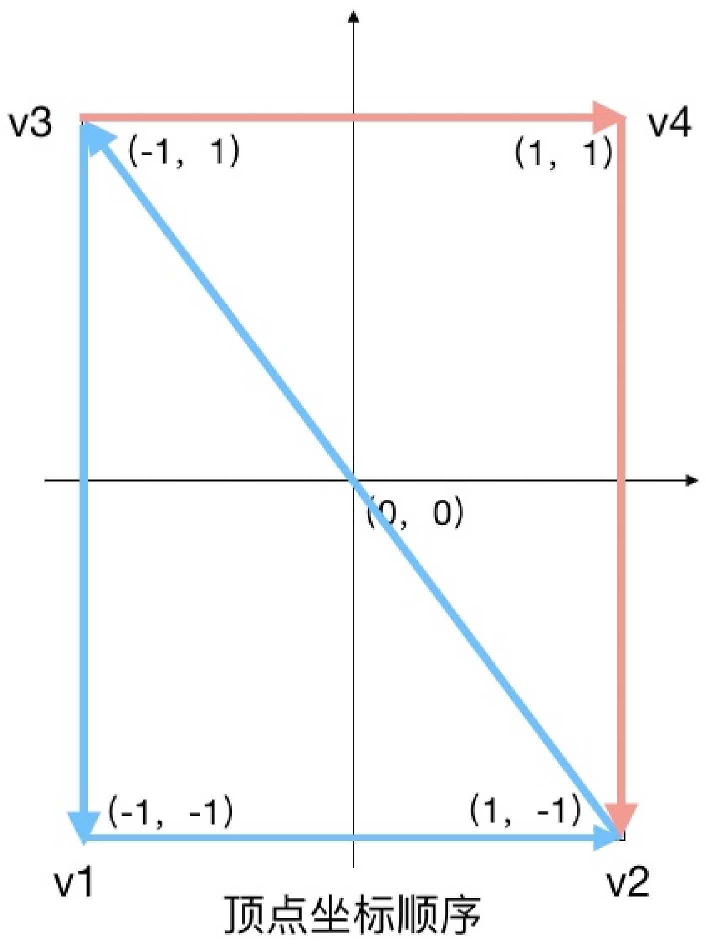
GL_TRIANGLE_FAN: reuse the first vertex to form a triangle



GL_TRIANGLE_FAN

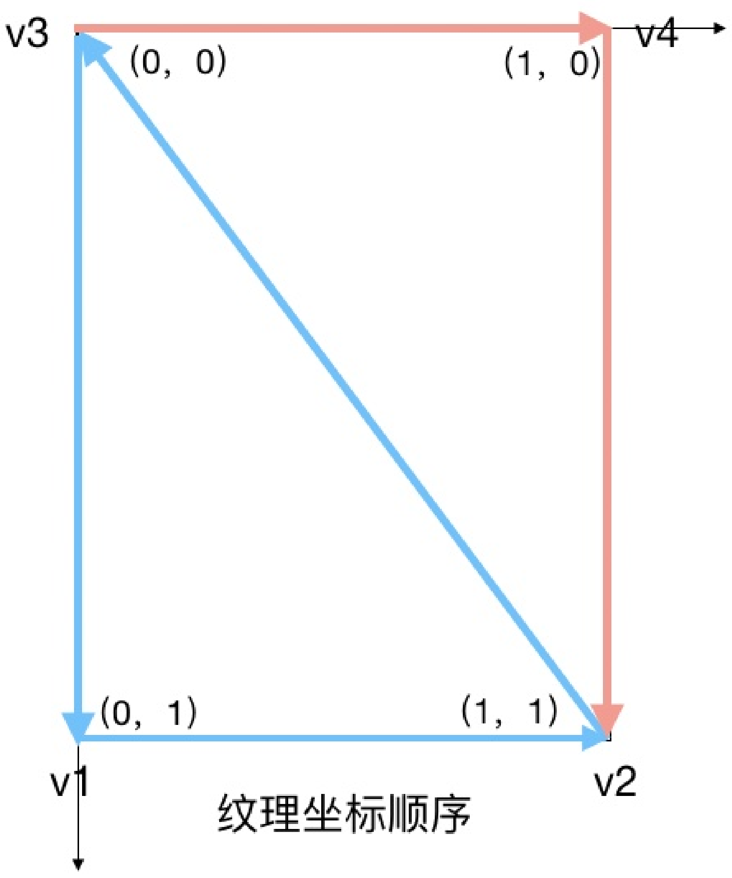
GL_TRIANGLE_FAN

Normally, the GL_TRIANGLE_STRIP drawing mode is generally used. Then the vertex order of a quad looks like this (v1-v2-v3) (v2-v3-v4)



Vertex Coordinate Order

The corresponding texture coordinates should also be in the same order as the vertex coordinates, otherwise there will be anomalies such as inversion and deformation.



texture coordinate order

Since the triangle is drawn, the two coordinates are as follows (only the xy-axis coordinates are set here, the z-axis coordinates are ignored, and each two data constitutes a coordinate point):

```
//顶点坐标
private val mVertexCoors = floatArrayOf(
    -1f, -1f,
     1f, -1f,
     0f,  1f
)
//纹理坐标
private val mTextureCoors = floatArrayOf(
    0f,   1f,
    1f,   1f,
    0.5f, 0f
)
```

In the initPos method, since the bottom layer cannot directly receive the array, convert the array to ByteBuffer

2) Create, compile and start the OpenGL shader

```
 private fun createGLPrg() {
    if (mProgram == -1) {
        val vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, getVertexShader())
        val fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER,
getFragmentShader())

        //创建OpenGL ES程序，注意：需要在OpenGL渲染线程中创建，否则无法渲染
        mProgram = GLES20.glCreateProgram()
        //将顶点着色器加入到程序
        GLES20.glAttachShader(mProgram, vertexShader)
        //将片元着色器加入到程序中
        GLES20.glAttachShader(mProgram, fragmentShader)
        //连接到着色器程序
        GLES20.glLinkProgram(mProgram)

        mVertexPosHandler = GLES20.glGetAttribLocation(mProgram, "aPosition")
        mTexturePosHandler = GLES20.glGetAttribLocation(mProgram, "aCoordinate")
    }
    //使用OpenGL程序
    GLES20.glUseProgram(mProgram)
}

private fun getVertexShader(): String {
    return "attribute vec4 aPosition;" +
            "void main() {" +
            "  gl_Position = aPosition;" +
            "}"
}

private fun getFragmentShader(): String {
    return "precision mediump float;" +
            "void main() {" +
            "  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);" +
            "}"
}

private fun loadShader(type: Int, shaderCode: String): Int {
    //根据type创建顶点着色器或者片元着色器
    val shader = GLES20.glCreateShader(type)
    //将资源加入到着色器中，并编译
    GLES20.glShaderSource(shader, shaderCode)
    GLES20.glCompileShader(shader)

    return shader
}
```

As mentioned above, GLSL is a programming language for GPU, and a shader is a small program. In order to be able to run this small program, it needs to be compiled and bound before it can be used.

The shader in this example is the simplest shader mentioned above.

> As you can see, the shader is actually a string

Enter loadShader, and obtain vertex shader and fragment shader according to different types through GLES20.glCreateShader.

Then call the following method to compile the shader

```
GLES20.glShaderSource(shader, shaderCode)
GLES20.glCompileShader(shader)
```

After compiling the shader, just bind, connect, and enable the program.

Remember above that coordinates in shaders are passed to GLSL by Java?

Careful you may have found these two lines of code

```
mVertexPosHandler = GLES20.glGetAttribLocation(mProgram, "aPosition")
mTexturePosHandler = GLES20.glGetAttribLocation(mProgram, "aCoordinate")
```

Yes, this is the channel through which Java and GLSL interact, and GLSL can be set relevant values through properties.

3) Start rendering and drawing

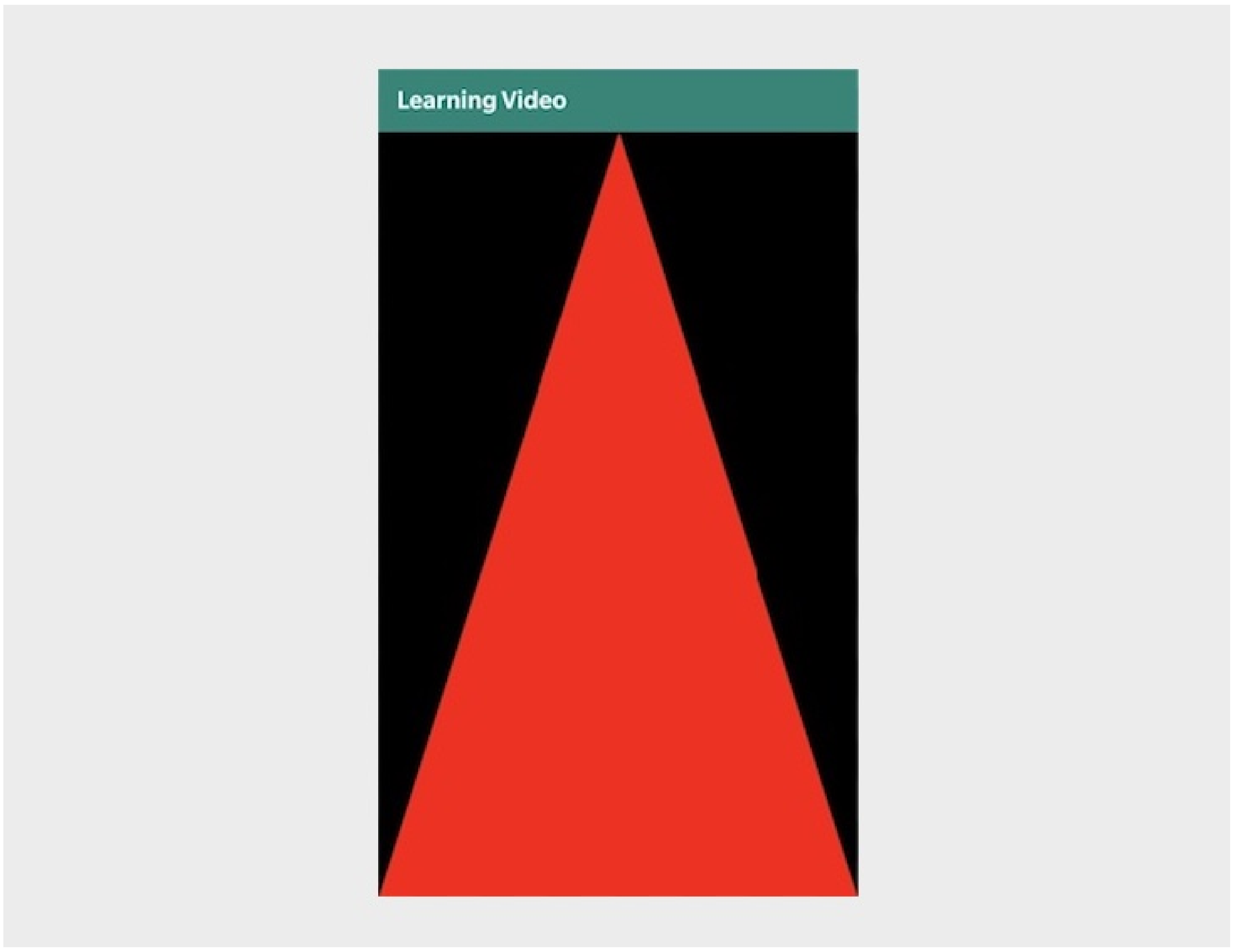
```
private fun doDraw() {
    //启用顶点的句柄
    GLES20.glEnableVertexAttribArray(mVertexPosHandler)
    GLES20.glEnableVertexAttribArray(mTexturePosHandler)
    //设置着色器参数， 第二个参数表示一个顶点包含的数据数量，这里为xy，所以为2
    GLES20.glVertexAttribPointer(mVertexPosHandler, 2, GLES20.GL_FLOAT, false, 0,
mVertexBuffer)
    GLES20.glVertexAttribPointer(mTexturePosHandler, 2, GLES20.GL_FLOAT, false, 0,
mTextureBuffer)
    //开始绘制
    GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, 3)
}
```

First activate the vertex coordinate and texture coordinate properties of the shader, then pass the initialized coordinates to the shader, and finally start the drawing:

```
GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, 3)
```

> There are two ways to draw: glDrawArrays and glDrawElements. The difference between the two is that glDrawArrays directly uses the defined vertex order to draw; while glDrawElements needs to define another index array to confirm the combination and drawing order of vertices.

Through the above steps, you can see a red triangle on the screen.



triangle

Some people may have doubts: when drawing a triangle, only the color value of the pixel is directly set, and the texture is not used. What is the use of the texture?

Next, use the texture to display an image to see how the texture is used.

> It is recommended to look at the process of drawing triangles first. Drawing pictures is based on the above process. Repeated codes will not be posted.

3, texture map, display a picture

Only the part of the code that is different from drawing triangles is posted below. For details, please see the source code .

```
class BitmapDrawer(private val mTextureId: Int, private val mBitmap: Bitmap):
IDrawer {
    //-------【注1：坐标变更了，由四个点组成一个四边形】-------
    // 顶点坐标
    private val mVertexCoors = floatArrayOf(
        -1f, -1f,
        1f, -1f,
        -1f, 1f,
        1f, 1f
    )

    // 纹理坐标
    private val mTextureCoors = floatArrayOf(
        0f, 1f,
        1f, 1f,
        0f, 0f,
        1f, 0f
    )

    //-------【注2：新增纹理接收者】-------
    // 纹理接收者
    private var mTextureHandler: Int = -1

    fun draw() {
        if (mTextureId != -1) {
            //【步骤2: 创建、编译并启动OpenGL着色器】
            createGLPrg()
            //-------【注4：新增两个步骤】-------
            //【步骤3: 激活并绑定纹理单元】
            activateTexture()
            //【步骤4: 绑定图片到纹理单元】
            bindBitmapToTexture()
            //----------------------------------
            //【步骤5: 开始渲染绘制】
            doDraw()
        }
    }

    private fun createGLPrg() {
        if (mProgram == -1) {
            //省略与绘制三角形一致的部分
            //......

            mVertexPosHandler = GLES20.glGetAttribLocation(mProgram, "aPosition")
            mTexturePosHandler = GLES20.glGetAttribLocation(mProgram,
"aCoordinate")
            //【注3：新增获取纹理接收者】
            mTextureHandler = GLES20.glGetUniformLocation(mProgram, "uTexture")
        }
        //使用OpenGL程序
        GLES20.glUseProgram(mProgram)
    }

    private fun activateTexture() {
        //激活指定纹理单元
        GLES20.glActiveTexture(GLES20.GL_TEXTURE0)
        //绑定纹理ID到纹理单元
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, mTextureId)
        //将激活的纹理单元传递到着色器里面
        GLES20.glUniform1i(mTextureHandler, 0)
        //配置边缘过渡参数
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MIN_FILTER,
GLES20.GL_LINEAR.toFloat())
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MAG_FILTER,
GLES20.GL_LINEAR.toFloat())
        GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_S,
GLES20.GL_CLAMP_TO_EDGE)
        GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_T,
GLES20.GL_CLAMP_TO_EDGE)
    }

    private fun bindBitmapToTexture() {
        if (!mBitmap.isRecycled) {
            //绑定图片到被激活的纹理单元
            GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, mBitmap, 0)
        }
    }

    private fun doDraw() {
        //省略与绘制三角形一致的部分
        //......

        //【注5：绘制顶点加1，变为4】
        //开始绘制：最后一个参数，将顶点数量改为4
        GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, 4)
    }

    private fun getVertexShader(): String {
        return "attribute vec4 aPosition;" +
                "attribute vec2 aCoordinate;" +
                "varying vec2 vCoordinate;" +
                "void main() {" +
                "  gl_Position = aPosition;" +
                "  vCoordinate = aCoordinate;" +
                "}"
    }

    private fun getFragmentShader(): String {
        return "precision mediump float;" +
                "uniform sampler2D uTexture;" +
                "varying vec2 vCoordinate;" +
                "void main() {" +
                "  vec4 color = texture2D(uTexture, vCoordinate);" +
                "  gl_FragColor = color;" +
                "}"
    }

    //省略和绘制三角形内容一致的部分
    //......
}
```

Inconsistencies have been identified in the code (see [Note: x] in the code). Take a look at them one by one:

1) Vertex coordinates

The vertex coordinates and texture coordinates are changed from 3 to 4, forming a rectangle, and the combination method is also GL_TRIANGLE_STRIP.

2) Shaders

First, let's talk about qualifiers in GLSL

- Attritude: Generally used for different amounts of each vertex. Such as vertex color, coordinates, etc.
- uniform: Generally used for quantities that are the same for all vertices in a 3D object. Such as light source position, unified transformation matrix, etc.
- varying: Indicates variable, generally used for the amount passed from the vertex shader to the fragment shader.
  const: constant.

Each line of code is parsed as follows:

```
private fun getVertexShader(): String {
    return  //顶点坐标
            "attribute vec2 aPosition;" +
            //纹理坐标
            "attribute vec2 aCoordinate;" +
            //用于传递纹理坐标给片元着色器，命名和片元着色器中的一致
            "varying vec2 vCoordinate;" +
            "void main() {" +
            "  gl_Position = aPosition;" +
            "  vCoordinate = aCoordinate;" +
            "}"
}

private fun getFragmentShader(): String {
    return  //配置float精度，使用了float数据一定要配置：lowp(低)/mediump(中)/highp(高)
            "precision mediump float;" +
            //从Java传递进入来的纹理单元
            "uniform sampler2D uTexture;" +
            //从顶点着色器传递进来的纹理坐标
            "varying vec2 vCoordinate;" +
            "void main() {" +
            //根据纹理坐标，从纹理单元中取色
            "  vec4 color = texture2D(uTexture, vCoordinate);" +
            "  gl_FragColor = color;" +
            "}"
}
```

Two new steps have been added to the drawing process:

3) Activate and bind the texture unit

```
private fun activateTexture() {
    //激活指定纹理单元
    GLES20.glActiveTexture(GLES20.GL_TEXTURE0)
    //绑定纹理ID到纹理单元
    GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, mTextureId)
    //将激活的纹理单元传递到着色器里面
    GLES20.glUniform1i(mTextureHandler, 0)
    //配置纹理过滤模式
    GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MIN_FILTER,
GLES20.GL_LINEAR.toFloat())
    GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MAG_FILTER,
GLES20.GL_LINEAR.toFloat())
    //配置纹理环绕方式
    GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_S,
GLES20.GL_CLAMP_TO_EDGE)
    GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_T,
GLES20.GL_CLAMP_TO_EDGE)
}
```

Since displaying a picture requires a texture unit to transmit the content of the entire picture, a texture unit needs to be activated first.

> **Why is it a texture unit?**
> Because there are many texture units built in OpenGL ES, and they are continuous, such as GLES20.GL_TEXTURE0, GLES20.GL_TEXTURE1, GLES20.GL_TEXTURE3... You can choose one of them, generally the first GLES20 is selected by default. GL_TEXTURE0, and OpenGL activates the first texture unit by default.
> Also, texture unit GLES20.GL_TEXTURE1 = GLES20.GL_TEXTURE0 + 1, and so on.

After activating the specified texture unit, you need to bind it with the texture ID, and when passing it to the shader: GLES20.glUniform1i(mTextureHandler, 0), the second parameter index needs to be consistent with the texture unit index.

> At this point, it can be found that the names of OpenGL methods are relatively regular. For example, GLES20.glUniform1i corresponds to the uniform qualifier variable in GLSL; ES20.glGetAttribLocation corresponds to the attribute qualifier variable in GLSL, etc.

The last four lines of code are used to configure the texture filtering mode and texture wrapping mode (the introduction of these two modes is quoted from [LearnOpenGL-CN])

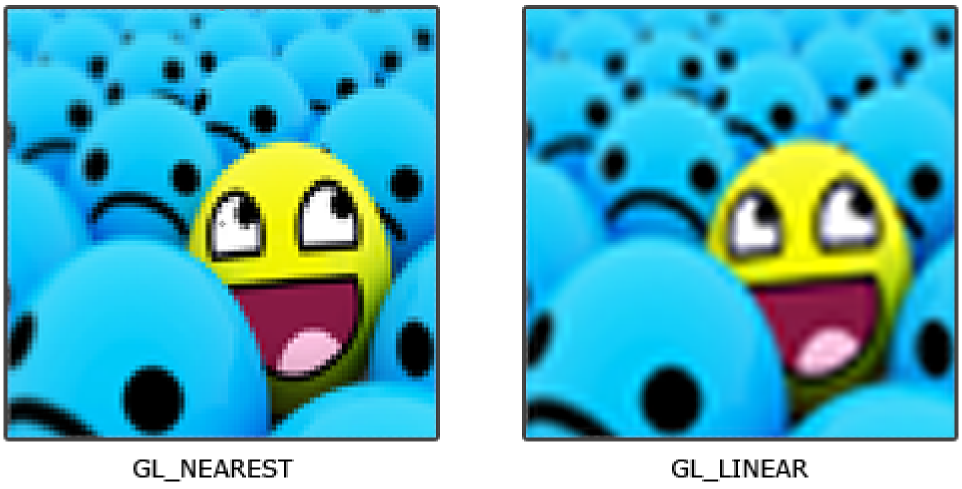
texture filter mode

> Texture coordinates are not resolution dependent, it can be any floating point value, so OpenGL needs to know how to map texels to texture coordinates.

Generally use these two modes: GL_NEAREST (proximity filtering), GL_LINEAR (linear filtering)

When set to GL_NEAREST, OpenGL will choose the pixel whose center point is closest to the texture coordinates.

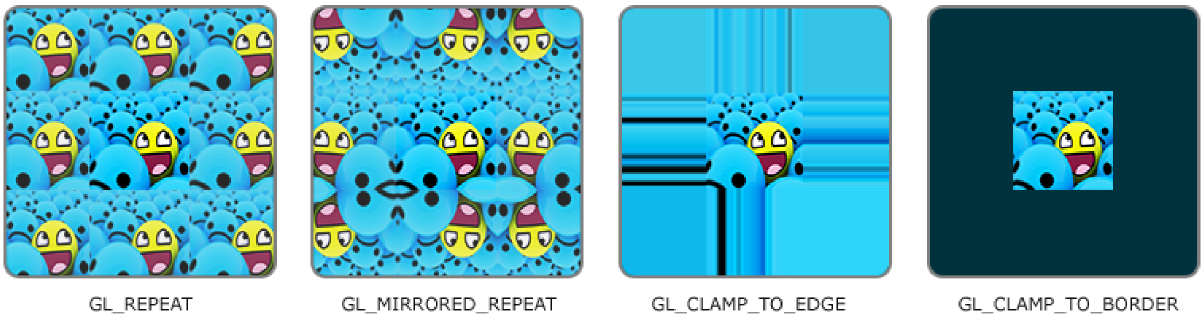
When set to GL_LINEAR, it calculates an interpolation based on the texels near the texture coordinates, approximating the color between these texels.



SourceLearnOpenGL-CN

Texture wrapping

wraparound	**describe**
GL_REPEAT	Default behavior for textures. Repeat texture image.
GL_MIRRORED_REPEAT	Same as GL_REPEAT, but each repeat image is placed in a mirror image.
GL_CLAMP_TO_EDGE	The texture coordinates will be constrained to be between 0 and 1, and the excess will repeat the edges of the texture coordinates, producing an effect of stretched edges.
GL_CLAMP_TO_BORDER	Exceeded coordinates are user-specified edge colors.



SourceLearnOpenGL-CN

4) Bind the image to the texture unit

After activating the texture unit, call the texImage2D method to bind the bmp to the specified texture unit.

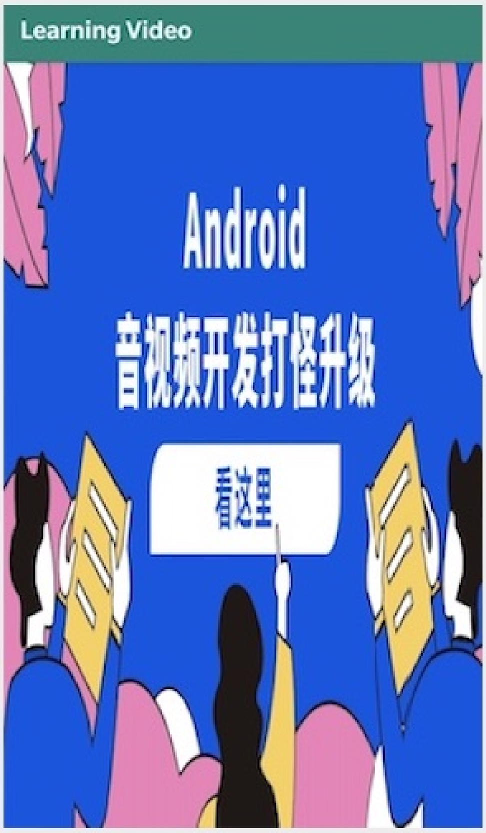
```
GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, mBitmap, 0)
```

5) draw

When drawing, the last parameter of the last sentence is changed from 3 vertices of a triangle to 4 vertices of a rectangle. If you still fill in 3, you will find that half of the picture will be displayed, that is, the triangle (divided diagonally).

```
GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, 4)
```

At this point, a picture is displayed through the texture map.



texture map

> Of course, you will find that this picture is deformed and covers the entire GLSurfaceView window. This involves the problem of vertex coordinate transformation, which will be explained in detail in the next article.

V. Summary

After the simple drawing of triangles and texture maps above, the 2D drawing process of OpenGL ES in Android can be summarized:

1. Configure OpenGL ES version through GLSurfaceView, specify Render
2. Implement GLSurfaceView.Renderer, overwrite the exposed methods, configure the OpenGL display window, and clear the screen
3. Create texture ID

4. Configure vertex coordinates and texture coordinates
5. **Initialize the coordinate transformation matrix**
6. Initialize OpenGL programs, compile and link vertex shaders and fragment shaders, and obtain variable attributes in GLSL
7. Activate texture unit, bind texture ID, configure texture filtering mode and wrapping mode
8. Binding textures (such as binding a bitmap to a texture)
9. start drawing

The above is basically a general process, of course, rendering pictures and rendering videos are slightly different, and point 5 will be mentioned in the next article.