【Android 音视频开发打怪升级：OpenGL渲染视频画面篇】二、使用OpenGL渲染视频画面 - 简书

jianshu.com/p/176880b2b3a2

[Android audio and video development and upgrade: OpenGL rendering video pictures] Second, use OpenGL to render video pictures

Table of contents

1. Android audio and video hard decoding articles:

Second, use OpenGL to render video images

- 1. Preliminary understanding of OpenGL ES
- 2. Use OpenGL to render video images
- 3. OpenGL renders multiple videos and realizes picture-in-picture
- 4. In-depth understanding of OpenGL's EGL
- 5, OpenGL FBO data buffer
- 6, Android audio and video hard coding: generate an MP4

Three, Android FFmpeg audio and video decoding articles

- 1, FFmpeg so library compilation
- 2. Android introduces FFmpeg
- 3. Android FFmpeg video decoding and playback
- 4. Android FFmpeg+OpenSL ES audio decoding and playback
- 5. Android FFmpeg + OpenGL ES to play video
- 6, Android FFmpeg simple synthesis of MP4: video unpacking and repackaging
- 7. Android FFmpeg video encoding

In this article you can learn

> The simple application of OpenGL ES on Android has been introduced above. Based on the above basic knowledge, this article will use OpenGL to render video pictures, and explain the knowledge about picture projection to solve the problem of picture stretching and deformation.

1. Rendering the video screen

In the first article [Audio and Video Basics], it was introduced that video is actually composed of pictures. In the above [Introduction to OpenGL ES], it introduced how to render a picture through OpenGL. Guess, the rendering of the video and the rendering of the picture should be similar. Without further ado, let's take a look.

1. Define the video renderer

In the above, the video rendering interface class is defined

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

First, implement the above interface and define a video renderer

```
class VideoDrawer : IDrawer {

    // 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
    )

    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
    // 纹理接收者
    private var mTextureHandler: 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: 激活并绑定纹理单元】
            activateTexture()
            //【步骤4: 绑定图片到纹理单元】
            updateTexture()
            //【步骤5: 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")
            mTextureHandler = GLES20.glGetUniformLocation(mProgram, "uTexture")
            mTexturePosHandler = GLES20.glGetAttribLocation(mProgram,
"aCoordinate")
        }
        //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(GLES11Ext.GL_TEXTURE_EXTERNAL_OES, 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(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR.toFloat())
        GLES20.glTexParameterf(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR.toFloat())
        GLES20.glTexParameteri(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
GLES20.GL_TEXTURE_WRAP_S, GLES20.GL_CLAMP_TO_EDGE)
        GLES20.glTexParameteri(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
GLES20.GL_TEXTURE_WRAP_T, GLES20.GL_CLAMP_TO_EDGE)
    }
```

```
    private fun updateTexture() {
    }

    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, 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;" +
                "attribute vec2 aCoordinate;" +
                "varying vec2 vCoordinate;" +
                "void main() {" +
                "    gl_Position = aPosition;" +
                "    vCoordinate = aCoordinate;" +
                "}"
    }

    private fun getFragmentShader(): String {
        //一定要加换行"\n"，否则会和下一行的precision混在一起，导致编译出错
        return "#extension GL_OES_EGL_image_external : require\n" +
                "precision mediump float;" +
                "varying vec2 vCoordinate;" +
                "uniform samplerExternalOES uTexture;" +
                "void main() {" +
                "  gl_FragColor=texture2D(uTexture, vCoordinate);" +
                "}"
    }

    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
    }
}
```

At first glance, it is exactly the same as the rendered picture. Don't worry, just listen to me one by one. Take a look at the process of this draw:

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

override fun draw() {
    if (mTextureId != -1) {
        //【步骤2: 创建、编译并启动OpenGL着色器】
        createGLPrg()
        //【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】
        updateTexture()
        //【步骤5: 开始渲染绘制】
        doDraw()
    }
}
```

Same place:

1. Vertex coordinate and texture coordinate settings
2. The process of creating a new OpenGL Program and loading the GLSL program. (But only the process is the same, the details are different)
3. The process of draw

Different places:

1. fragment shader

```
//视频片元着色器

private fun getFragmentShader(): String {
    //一定要加换行"\n"，否则会和下一行的precision混在一起，导致编译出错
    return "#extension GL_OES_EGL_image_external : require\n" +
            "precision mediump float;" +
            "varying vec2 vCoordinate;" +
            "uniform samplerExternalOES uTexture;" +
            "void main() {" +
            "  gl_FragColor=texture2D(uTexture, vCoordinate);" +
            "}"
}
```

Compare the fragment shader of the image

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

Now, the first line adds:

```
#extension GL_OES_EGL_image_external : require
```

The rendering of the video screen uses the extended texture of Android

> **Expand the role of texture?**
> We already know that the color space of the video picture is YUV, and to display it on the screen, the picture is RGB, so to render the video picture to the screen, we must convert YUV to RGB. Extending the texture does this transformation.

The texture units in the fourth row are also replaced with extended texture units.

```
uniform samplerExternalOES uTexture;
```

2. activate 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(GLES11Ext.GL_TEXTURE_EXTERNAL_OES, 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(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR.toFloat())
    GLES20.glTexParameterf(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR.toFloat())
    GLES20.glTexParameteri(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
GLES20.GL_TEXTURE_WRAP_S, GLES20.GL_CLAMP_TO_EDGE)
    GLES20.glTexParameteri(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
GLES20.GL_TEXTURE_WRAP_T, GLES20.GL_CLAMP_TO_EDGE)
}
```

Similarly, replace all ordinary texture units with extended texture units **GLES11Ext.GL_TEXTURE_EXTERNAL_OES**

3. update texture unit

```
private fun updateTexture() {

}
```

Similar to rendering a picture, to display the picture, you must first bind the picture (such as the bitmap of the picture) to the texture unit.

But why is it empty now? Because only the above process is used, the video cannot be displayed.

The rendering of the video needs to update the screen through SurfaceTexture. Next, let's see how to generate it.

```
class VideoDrawer: IDrawer {
    //......

    private var mSurfaceTexture: SurfaceTexture? = null


    override fun setTextureID(id: Int) {
        mTextureId = id
        mSurfaceTexture = SurfaceTexture(id)
    }
    //......
}
```

A SurfaceTexture is added to VideoDrawer, and in setTextureID, the SurfaceTexture is initialized with the texture ID.

in updateTexture method

```
private fun updateTexture() {
    mSurfaceTexture?.updateTexImage()
}
```

At this point, you should think that this time it is finally possible, but it is still one step away.

Remember that in the second encapsulation basic decoding framework of hard decoding, it was mentioned that MediaCodec should provide a Surface as a rendering surface. And Surface needs a SurfaceTexture.

Therefore, we need to pass this SurfaceTexture to external use. First add a method in IDrawer

```
interface IDrawer {
    fun draw()
    fun setTextureID(id: Int)
    fun release()
    //新增接口，用于提供SurfaceTexture
    fun getSurfaceTexture(cb: (st: SurfaceTexture)->Unit) {}
}
```

Pass the SurfaceTexture back through a higher-order function parameter. details as follows:

```
class VideoDrawer: IDrawer {

    //......

    private var mSftCb: ((SurfaceTexture) -> Unit)? = null

    override fun setTextureID(id: Int) {
        mTextureId = id
        mSurfaceTexture = SurfaceTexture(id)
        mSftCb?.invoke(mSurfaceTexture!!)
    }

    override fun getSurfaceTexture(cb: (st: SurfaceTexture) -> Unit) {
        mSftCb = cb
    }

    //......
}
```

2. Use OpenGL to play video

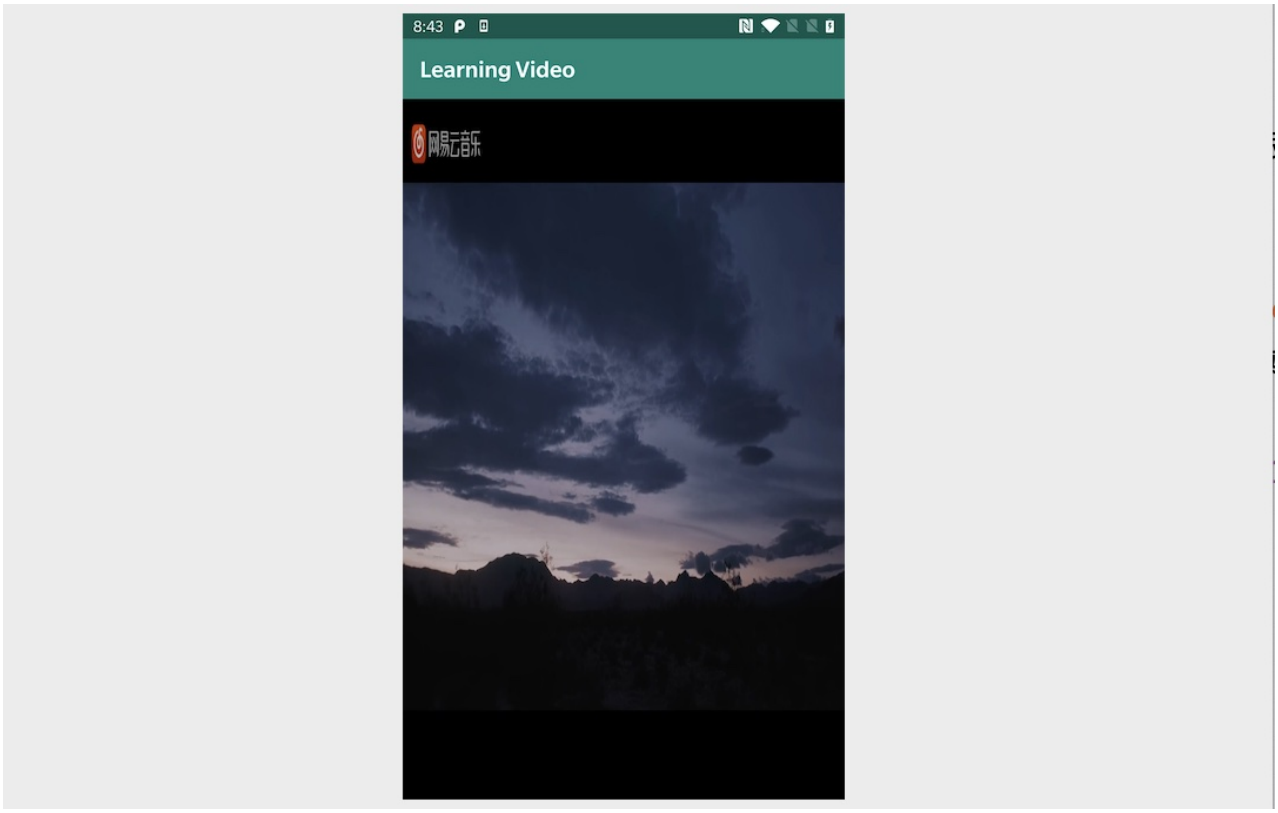
create a new page

```
<?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>
```

```
class OpenGLPlayerActivity: AppCompatActivity() {
    val path = Environment.getExternalStorageDirectory().absolutePath +
"/mvtest_2.mp4"
    lateinit var drawer: IDrawer

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_opengl_player)
        initRender()
    }

    private fun initRender() {
        drawer = VideoDrawer()
        drawer.getSurfaceTexture {
            //使用SurfaceTexture初始化一个Surface，并传递给MediaCodec使用
            initPlayer(Surface(it))
        }
        gl_surface.setEGLContextClientVersion(2)
        gl_surface.setRenderer(SimpleRender(drawer))
    }

    private fun initPlayer(sf: Surface) {
        val threadPool = Executors.newFixedThreadPool(10)

        val videoDecoder = VideoDecoder(path, null, sf)
        threadPool.execute(videoDecoder)

        val audioDecoder = AudioDecoder(path)
        threadPool.execute(audioDecoder)

        videoDecoder.goOn()
        audioDecoder.goOn()
    }
}
```

It's basically the same as playing with SurfaceView, except that OpenGL and Surface are initialized. It's very simple, just look at the above code and won't explain it any more.

> If you use the above code to start playing the video, you will find that the video screen is stretched to the size of the GLSurfaceView window, that is, the full screen is covered.



screen is stretched

Second, the screen ratio correction

projection

The world coordinate of OpenGL is a standardized coordinate system. The xyz coordinate range is (-1 to 1). The default start and end positions correspond to the four corners of the world coordinate plane. At this time, the picture covers the entire screen, so the picture that has not undergone coordinate transformation generally has the problem of deformation.

OpenGL provides two ways to adjust the aspect ratio, namely **perspective projection** and **orthogonal projection** .

What does projection do?

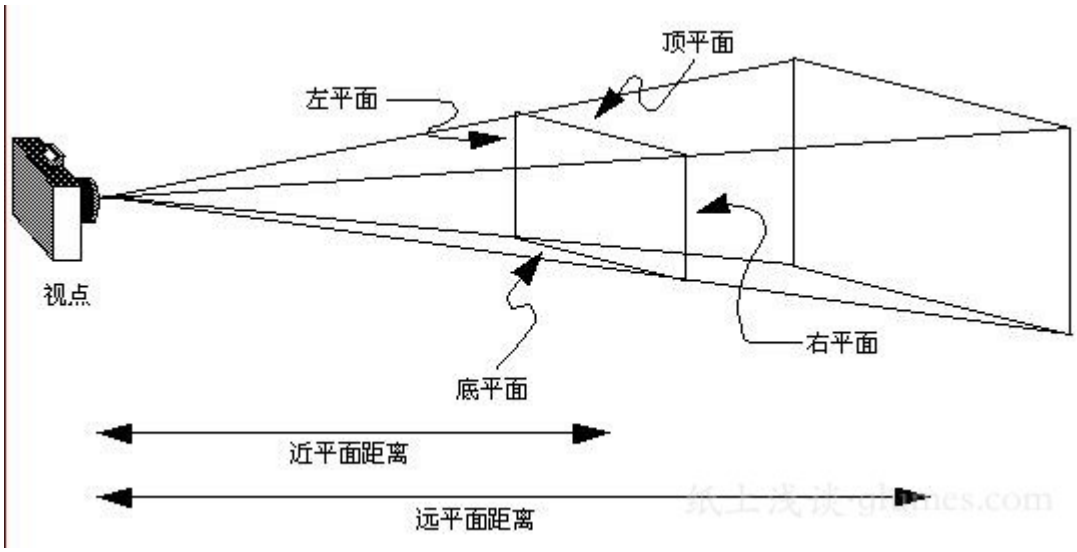
1. The projection specifies the range of the clipping space, that is, the visible space range of the object
2. Project objects in clip space onto the screen

To make it clear that OpenGL projection is not a simple matter, it will involve the definition of various spaces in OpenGL. Here is a brief list:

- **Local space** : the space relative to the object's own province, the origin is in the middle of the object
- **world space** : the coordinate system of the OpenGL world

- **Observation space** : the space of the observer (camera), which is equivalent to the space seen by the eyes of people in the real world. Different observation positions, when looking at the same object, will be different.
- **Clipping space** : the visible space, in which objects can be displayed on the screen
- **Screen space** : the screen coordinate space, which is the screen space of the mobile phone

perspective projection



perspective projection

As you can see from the above figure, the principle of perspective projection is actually the imaging principle of the object viewed by the human eye. Looking forward from the camera, there is a perspective space, similar to the observation angle of the human eye. What people see is projected on the retina, and what the camera sees is the image projected on the **near plane** (the plane closer to the camera).

- Camera position and orientation

First of all, the camera camera is not fixed and can be moved according to your own needs, then you need to set the position and orientation of the camera, which is related to how to observe the object.

> The thing to know is that the camera is still in world coordinate space. Therefore, the set camera position is relative to the origin of the world coordinate.

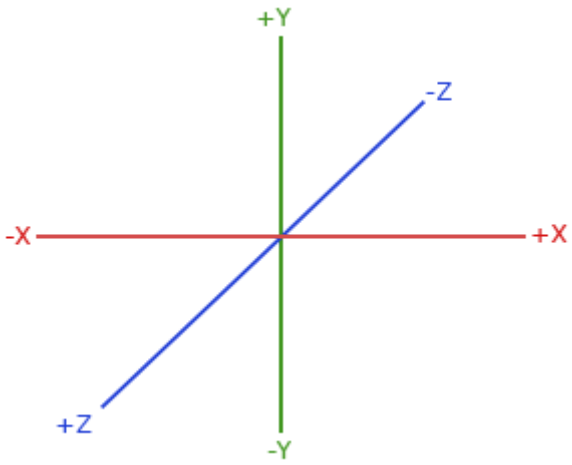
camera position

The OpenGL world coordinate system is a right-handed coordinate system, with the positive X-axis on the right-hand side, the positive Y-axis up, and the positive Z-axis across the screen towards you.

OpenGL world coordinate system

Then the camera coordinate can be (0, 0, 5), which is a point on the Z axis.

camera orientation



After setting the position of the camera, you also need to set the orientation of the camera. The orientation of the camera is determined by the three direction vectors upX, upY, uZ, and the starting point is the coordinate point of the camera. That is to say, the composite vector of these three vectors is the direction directly above the camera.

If the camera is analogous to an adult head, the direction of the composite vector is the direction directly above the head.

For example, you can set the orientation of the camera to (0, 1, 0). At this time, the camera is located at (0, 0, 5), and the upward direction is the Y axis. At this time, the camera just sees the plane composed of XY, which is the picture. front.

If the camera's orientation is set to (0, -1, 0), it is equivalent to the person's head down, which is the picture seen and the picture above is reversed.

near plane and far plane

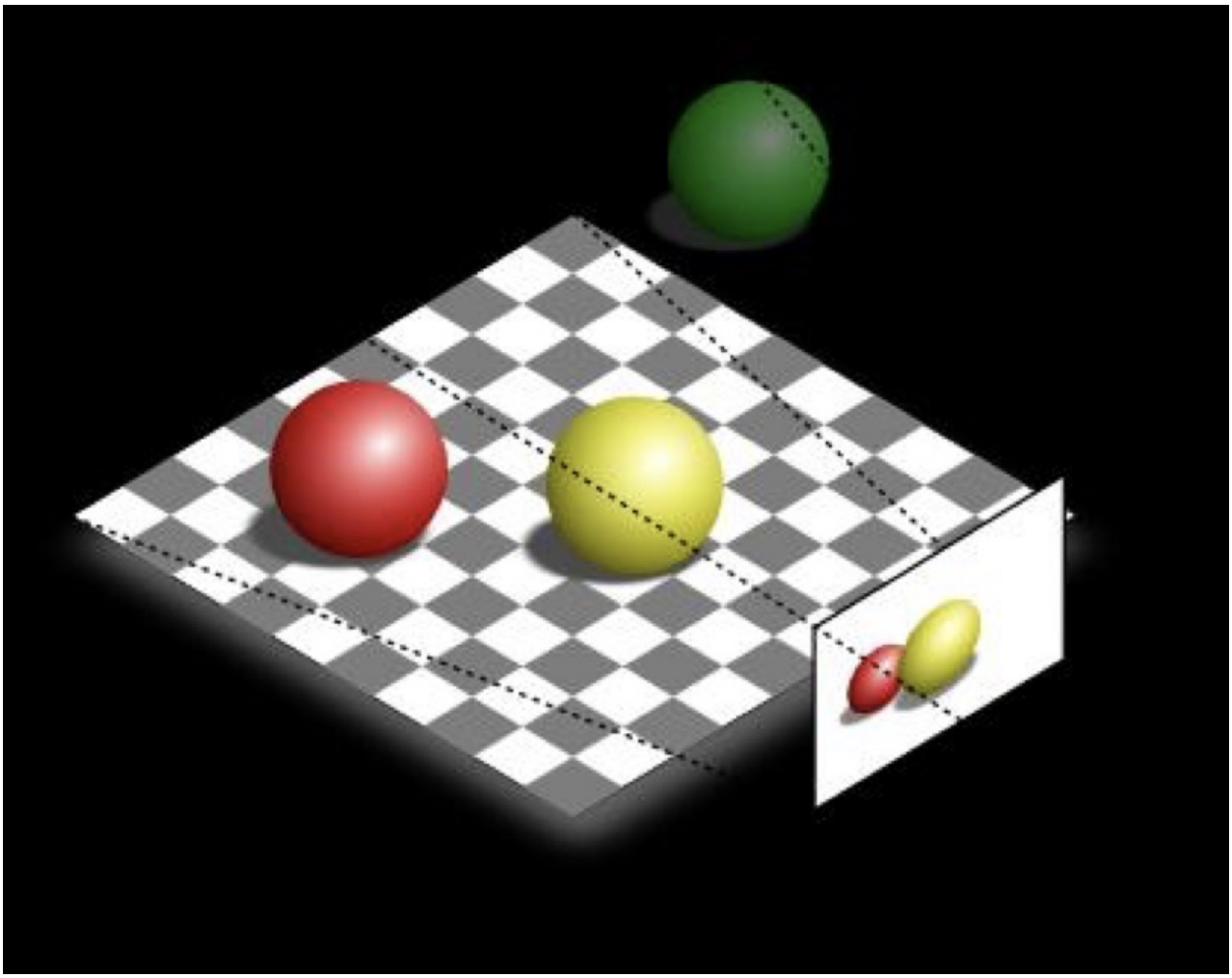
Looking back at the above picture of perspective projection, there are two planes on the right side of the camera, the near plane near the camera and the far plane on the far side.

clipping space

It can be seen that the connection lines of the four sides of the far plane and the near plane finally converge to the position of the camera. The space surrounded by these lines and two planes is the clipping space mentioned above, that is, the visible space.

In this space, the line connecting the surface of the object with the position of the camera and the points left by passing through the near plane form an image, which is **the projection of the object on the near plane, that is, the image seen on the screen of the mobile phone** .

Moreover, the farther away from the camera, the smaller the projection will be, which is exactly the same as the image of the human eye.



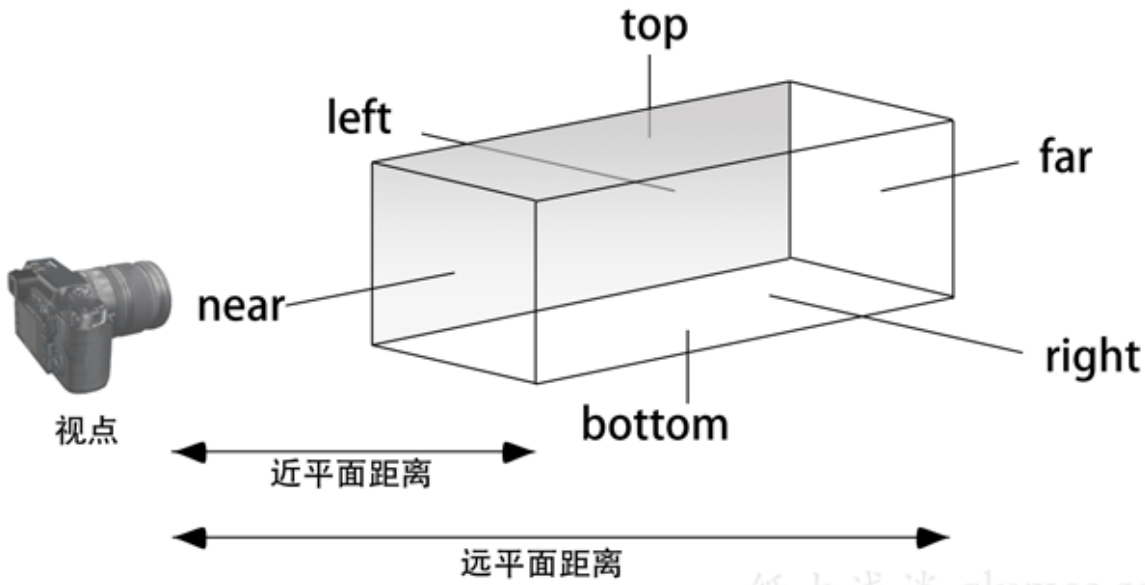
perspective projection imaging

Because of the same imaging principle as the human eye, perspective projection is often used in 3D rendering.

However, this is also more complicated, and I am not very familiar with it. In order to avoid wrong transmission, I will not explain the specific application here. You can read other people's articles to understand, such as [projection matrix and viewport transformation matrix], [projection matrix]

The following introduces the more commonly used projection modes in 2D rendering: orthographic projection.

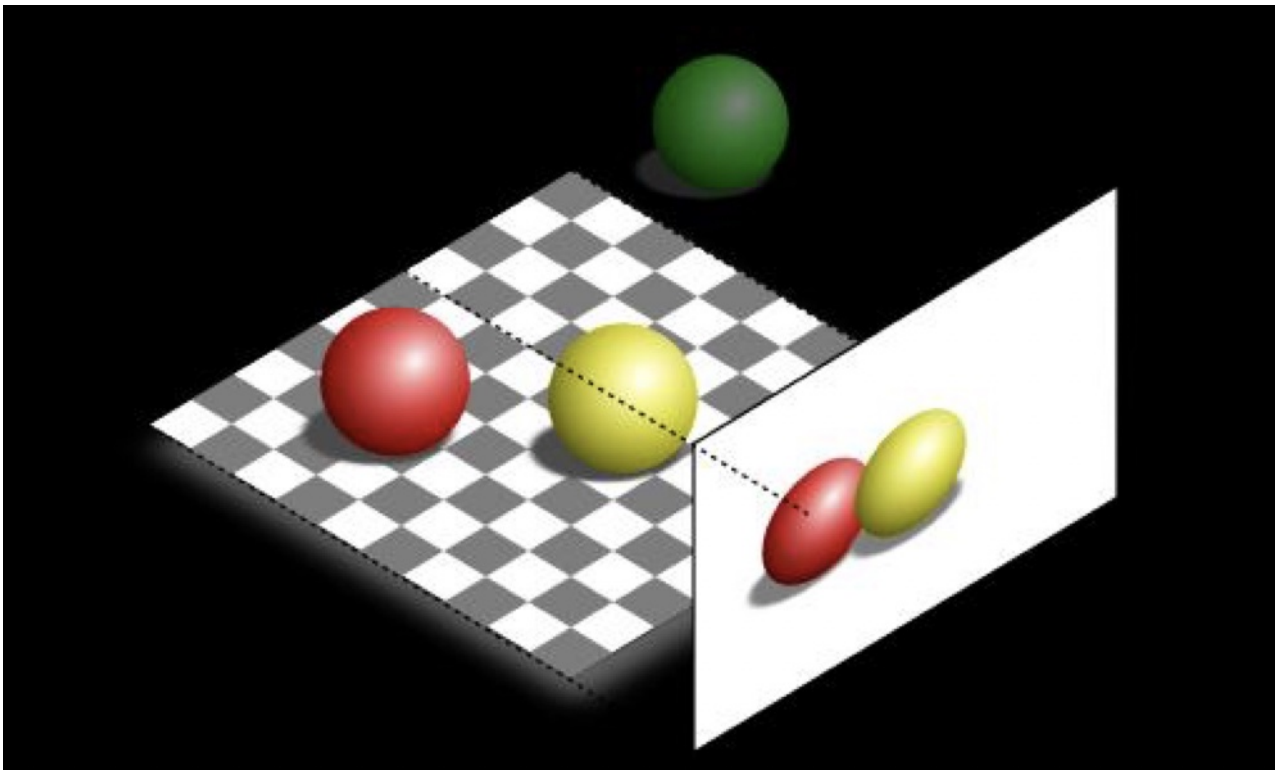
Orthographic projection



Orthographic projection

Like perspective projection, orthographic projection also has a camera, a near plane, and a far plane. The difference is that the camera's line of sight is not focused on a point, but parallel lines. So the visible window between the near plane and the far plane is a cuboid.

That is to say, the vision of the orthogonal projection is no longer like the human eye. All objects in the clipping space, no matter how far or near, as long as they are the same size, the projection on the near plane is the same, and there is no longer a near big, far small Effect.



Orthographic projection imaging

This effect is very suitable for rendering 2D images.

OpenGL provides the Matrix.orthoM function to generate an orthographic projection matrix:

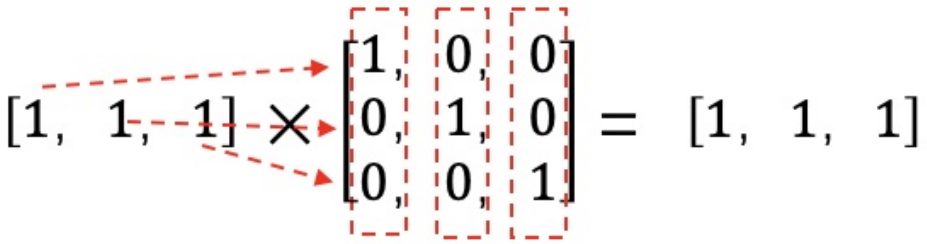
```
/**
 * Computes an orthographic projection matrix.
 *
 * @param m returns the result 正交投影矩阵
 * @param mOffset 偏移量，默认为 0 ,不偏移
 * @param left 左平面距离
 * @param right 右平面距离
 * @param bottom 下平面距离
 * @param top 上平面距离
 * @param near 近平面距离
 * @param far 远平面距离
 */
public static void orthoM(float[] m, int mOffset,
    float left, float right, float bottom, float top,
    float near, float far)
```

In addition to setting the distance between the near plane and the far plane, you also need to set the up, down, left, and right distances of the near plane. These four parameters correspond to the vertical distances between the four sides of the near plane quadrilateral and the origin. **It is also the key to correcting the video picture.**

matrix transformation

In the world of image processing, image transformation is the most used and matrix transformation, which requires a little knowledge of linear algebra.

Let's first look at a simple matrix multiplication:



matrix multiplication

Matrix multiplication is not the same as ordinary numeric multiplication. The rows of the first matrix are multiplied by the columns of the second matrix, and the products of each are added together, resulting in the first row and first column of the result, namely:

```
1x1 + 1x0 + 1x0 = 1
```

Others and so on.

> **Identity Matrix**
> It can be found that no matter what matrix is multiplied by the matrix on the right, the result is the same as the first one. Just like no matter what number is multiplied by 1, the result doesn't change. So this matrix on the right is called the **identity matrix** .

Let's look at another matrix multiplication:

$$[1,\ 1,\ 0] \times \begin{bmatrix} 0.5, & 0, & 0 \\ 0, & 0.5, & 0 \\ 0, & 0, & 1 \end{bmatrix} = [0.5,\ 0.5,\ 0]$$

Matrix scaling

The first two 1s of the right matrix are reduced by half, and the result of the multiplication is exactly the original matrix is reduced by half.

Imagine, if the three numbers of the left matrix are regarded as the xyz of the coordinate point? At this point you should be able to guess how to correct the aspect ratio.

$$[x,\ y,\ 0] \times \begin{bmatrix} 0.5, & 0, & 0 \\ 0, & 0.5, & 0 \\ 0, & 0, & 1 \end{bmatrix} = [0.5x,\ 0.5y,\ 0]$$

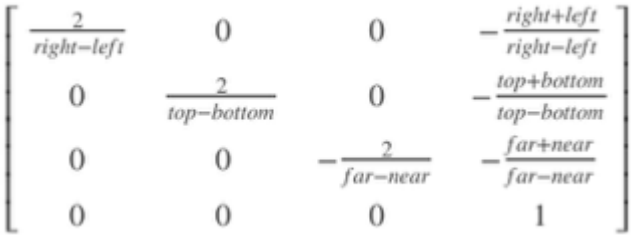
Matrix scaling

Since the video picture is stretched, the most direct method is to reduce the stretched direction of the picture by scaling, and matrix multiplication can just meet the needs of scaling.

Looking back at the method of orthographic projection:

```
public static void orthoM(float[] m, int mOffset,
    float left, float right, float bottom, float top,
    float near, float far)
```

Here is the matrix generated by this method:



Orthographic projection matrix

It can be seen that a scaling matrix is actually generated. Since z and w are both 0, the last two columns can be ignored. Let's focus on the first two: 2/(right - left) and 2/(top - bottom), these two are multiplied by xy respectively and play a role in scaling.

take a chestnut

Suppose, the video has a width and height of 1000x500, and the GLSurfaceView has a width and height of 1080x1920

This is a horizontal video. If the width is used to adapt, 500 is enlarged to 1920, then in order to maintain the proportion, the width must be put to 1000 x (1920/500) = 3840, which exceeds the width of 1080. Therefore, only the height can be scaled to keep the width and height of the final display of the video not exceeding the width and height of the GLSurfaceView.

Correctly scaled horizontal width and height: 1080x540 (500x1080/1000)

How much did you zoom in? Is it 540/500 = 1.08? wrong! ! !

If no zoom processing is performed, the picture is stretched and covered, and the height of the picture should be 1920, so the correct zoom factor should be 1920/540=3.555556 (cannot be divided)

Next, let's see how to set left, right, top, bottom.

It has been known from the above analysis that the width of the video screen is directly stretched to the maximum window, that is, the default is left = -1; right = 1 (tip: Remember that the OpenGL world coordinate origin is in the center of the screen?)

At this time

```
right - left = 2，那么缩放矩阵第一个参数为：
2/（right - left）= 1
```

That is, no scaling.

To reduce the height by 3.555556 times, then

```
2 /（top - bottom）= 2 /（2x3.555556）= 1/3.555556

由于top和bottom互为反数，所以top = - bottom = 3.555556
```

Therefore, the parameters of the orthographic projection are:

```
private var mPrjMatrix = FloatArray(16)

Matrix.orthoM(mPrjMatrix, 0, -1, 1, 3.555556, -3.555556, 1, 2)

//public static void orthoM(float[] m, int mOffset,
//    float left, float right, float bottom, float top,
//    float near, float far)
```

After knowing the calculation principle, let's deduce the zoom factor, how to calculate it through GLSurfaceView and the original aspect ratio of the screen.

Still using the above example, the zoom factor is 1920/540=3.555556, which is equivalent to

```
1920/（500*1080/1000）-->

GL_Height /（Video_Height*(GL_Width/Video_Width)）-->

(GL_Height/GL_Width) * (Video_Width/Video_Height) -->

(Video_Width/Video_Height) / (GL_Width/GL_Height) -->

Video_Ritio/GL_Ritio
```

It can be seen that we don't need to calculate the actual value. We can automatically infer the scaling ratio in the code based on the viewport and the original width and height of the video screen.

Of course, it is necessary to judge the specific situation, there are four situations:

```
1. 视口宽 > 高，并且视频的宽高比 > 视口的宽高比：缩放高度（Video_Ritio/GL_Ritio）

2. 视口宽 > 高，并且视频的宽高比 < 视口的宽高比：缩放宽度（GL_Ritio/Video_Ritio）

3. 视口宽 < 高，并且视频的宽高比 > 视口的宽高比：缩放高度（Video_Ritio/GL_Ritio）

4. 视口宽 < 高，并且视频的宽高比 < 视口的宽高比：缩放宽度（GL_Ritio/Video_Ritio）
```

The above example belongs to the third case.

The rest are no longer deduced, and if you are interested, you can push it yourself to deepen your understanding.

Next, let's see how to implement it in code.

Correct the aspect ratio

IDrawer adds two new interfaces, which are used to set the original width and height of the video, and set the width and height of the OpenGL window.

```
interface IDrawer {

    //设置视频的原始宽高
    fun setVideoSize(videoW: Int, videoH: Int)
    //设置OpenGL窗口宽高
    fun setWorldSize(worldW: Int, worldH: Int)

    fun draw()
    fun setTextureID(id: Int)
    fun getSurfaceTexture(cb: (st: SurfaceTexture)->Unit) {}
    fun release()
}
```

VideoDrawer realizes the correction process as follows:

```
class VideoDrawer: IDrawer {

    //......

    private var mWorldWidth: Int = -1
    private var mWorldHeight: Int = -1
    private var mVideoWidth: Int = -1
    private var mVideoHeight: Int = -1

    //坐标变换矩阵
    private var mMatrix: FloatArray? = null

    //矩阵变换接收者
    private var mVertexMatrixHandler: Int = -1

    override fun setVideoSize(videoW: Int, videoH: Int) {
        mVideoWidth = videoW
        mVideoHeight = videoH
    }

    override fun setWorldSize(worldW: Int, worldH: Int) {
        mWorldWidth = worldW
        mWorldHeight = worldH
    }

    override fun draw() {
        if (mTextureId != -1) {
            //【新增1: 初始化矩阵方法】
            initDefMatrix()
            //【步骤2: 创建、编译并启动OpenGL着色器】
            createGLPrg()
            //【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】
            updateTexture()
            //【步骤5: 开始渲染绘制】
            doDraw()
        }
    }

    private fun initDefMatrix() {
        if (mMatrix != null) return
        if (mVideoWidth != -1 && mVideoHeight != -1 &&
            mWorldWidth != -1 && mWorldHeight != -1) {
            mMatrix = FloatArray(16)
            var prjMatrix = FloatArray(16)
            val originRatio = mVideoWidth / mVideoHeight.toFloat()
            val worldRatio = mWorldWidth / mWorldHeight.toFloat()
            if (mWorldWidth > mWorldHeight) {
                if (originRatio > worldRatio) {
                    val actualRatio = originRatio / worldRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -1f, 1f,
                        -actualRatio, actualRatio,
                        -1f, 3f
```

```
                    )
                } else {// 原始比例小于窗口比例，缩放高度度会导致高度超出，因此，高度以窗口为准，缩放宽度
                    val actualRatio = worldRatio / originRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -actualRatio, actualRatio,
                        -1f, 1f,
                        -1f, 3f
                    )
                }
            } else {
                if (originRatio > worldRatio) {
                    val actualRatio = originRatio / worldRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -1f, 1f,
                        -actualRatio, actualRatio,
                        -1f, 3f
                    )
                } else {// 原始比例小于窗口比例，缩放高度会导致高度超出，因此，高度以窗口为准，缩放宽度
                    val actualRatio = worldRatio / originRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -actualRatio, actualRatio,
                        -1f, 1f,
                        -1f, 3f
                    )
                }
            }
        }
    }

    private fun createGLPrg() {
        if (mProgram == -1) {
            //省略加载shader代码
            //......

            //【新增2: 获取顶点着色器中的矩阵变量】
            mVertexMatrixHandler = GLES20.glGetUniformLocation(mProgram, "uMatrix")

            mVertexPosHandler = GLES20.glGetAttribLocation(mProgram, "aPosition")
            mTextureHandler = GLES20.glGetUniformLocation(mProgram, "uTexture")
            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)
```

```
        // 【新增3: 将变换矩阵传递给顶点着色器】
        GLES20.glUniformMatrix4fv(mVertexMatrixHandler, 1, false, mMatrix, 0)

        //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, 4)
    }

    private fun getVertexShader(): String {
        return "attribute vec4 aPosition;" +
                //【新增4: 矩阵变量】
                "uniform mat4 uMatrix;" +
                "attribute vec2 aCoordinate;" +
                "varying vec2 vCoordinate;" +
                "void main() {" +
                //【新增5: 坐标变换】
                "    gl_Position = aPosition*uMatrix;" +
                "    vCoordinate = aCoordinate;" +
                "}"
    }
    //......
}
```

See the above code for the new content: [Add x:]

As you can see, the vertex shader has changed, a new matrix variable has been added, and the last displayed coordinates are multiplied by this matrix.

```
uniform mat4 uMatrix;
gl_Position = aPosition*uMatrix;
```

In the code, the matrix variables in the shader are also obtained through the OpenGL method, and the scaling matrix is calculated and passed to the vertex shader.

By multiplying the two matrices aPosition and uMatrix, the correct display position of the picture pixel is obtained.

As for the principle of scaling, it has been explained clearly above, and I will not go into details, but only about the settings of the near plane and far plane.

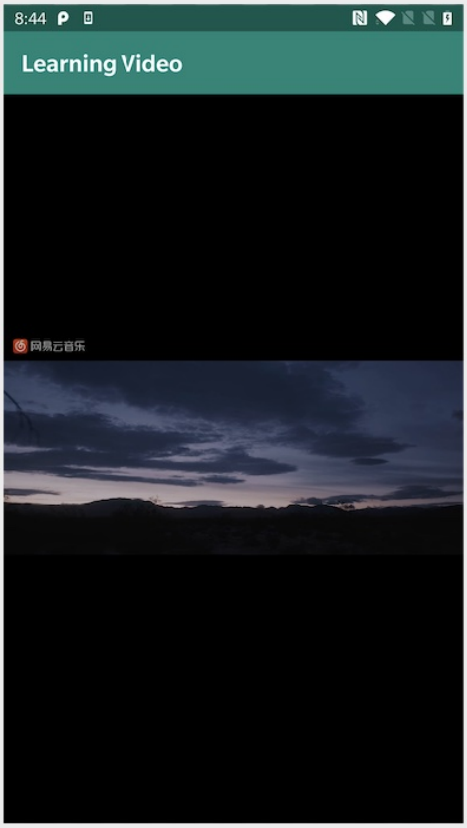
The z coordinate of our vertex coordinate setting is 0, and the default position of the camera is also at 0. In order for the vertex coordinate to be included in the clipping space, near must be <= 0, far must be >= 0, and cannot be equal to both. 0, i.e. near != far .

> Note: near and far are relative to the camera coordinate point, such as near = -1, the actual z coordinate of the near plane is 1, far = 1, and the z coordinate of the far plane is -1. The z-axis is vertical and outward from the phone screen.

See how the external call is made:

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

    //......

    override fun onSurfaceChanged(gl: GL10?, width: Int, height: Int) {
        GLES20.glViewport(0, 0, width, height)
        //设置OpenGL窗口坐标
        mDrawer.setWorldSize(width, height)
    }

    //......

}


class OpenGLPlayerActivity: AppCompatActivity() {

    //......

    private fun initRender() {
        drawer = VideoDrawer()
        //设置视频宽高
        drawer.setVideoSize(1920, 1080)
        drawer.getSurfaceTexture {
            initPlayer(Surface(it))
        }
        gl_surface.setEGLContextClientVersion(2)
        gl_surface.setRenderer(SimpleRender(drawer))
    }
    //......
```

At this point, a beautiful picture can finally be displayed normally.



normal screen.jpg

Change camera position

As mentioned above, OpenGL can set the position and orientation of the camera, but in fact, the above code does not set it, because the camera defaults to the position of the origin. Next, let's take a look at another method of setting the far and near planes.

```
/**
 * Defines a viewing transformation in terms of an eye point, a center of
 * view, and an up vector.
 *
 * @param rm returns the result
 * @param rmOffset index into rm where the result matrix starts
 * @param eyeX eye point X
 * @param eyeY eye point Y
 * @param eyeZ eye point Z
 * @param centerX center of view X
 * @param centerY center of view Y
 * @param centerZ center of view Z
 * @param upX up vector X
 * @param upY up vector Y
 * @param upZ up vector Z
 */
public static void setLookAtM(float[] rm, int rmOffset,
            float eyeX, float eyeY, float eyeZ,
            float centerX, float centerY, float centerZ,
            float upX, float upY, float upZ)
```

(eyeX, eyeY, eyeZ) determines the position of the camera, (upX, upY, upZ) determines the direction of the camera, (centerX, centerY, centerZ) is the origin of the picture, generally (0, 0, 0).

```
//设置相机位置
val viewMatrix = FloatArray(16)
Matrix.setLookAtM(viewMatrix, 0,
                  0f, 0f, 5.0f,
                  0f, 0f, 0f,
                  0f, 1.0f, 0f)
```

The above sets the position of the camera to be 5 from the origin on the z-axis. The upward direction of the camera is the Y axis, facing the xy plane.

This way, if the z-axis of the vertex coordinates is still 0, the near and far planes must be repositioned in order for the picture to be included in clip space.

for example:

```
Matrix.orthoM(mPrjMatrix, 0, -1, 1, 3.555556, -3.555556, 1, 6)
```

How did near = 1, far = 6 come about?

The camera is at z-axis 5. Then in order to include points with z=0, then the near plane can't be > 5 from the camera point, and the far plane can't be < 5 from the camera point. Likewise, near != far.

3. Video Filters

Filters can be seen in many video applications, which can change the style of the video. So how are these filters implemented?

In fact, the principle is very simple, nothing more than changing the color of the picture.

Here's a very simple filter: black and white

Just change the fragment shader:

```
private fun getFragmentShader(): String {
    //一定要加换行"\n"，否则会和下一行的precision混在一起，导致编译出错
    return "#extension GL_OES_EGL_image_external : require\n" +
            "precision mediump float;" +
            "varying vec2 vCoordinate;" +
            "uniform samplerExternalOES uTexture;" +
            "void main() {" +
            "  vec4 color = texture2D(uTexture, vCoordinate);" +
            "  float gray = (color.r + color.g + color.b)/3.0;" +
            "  gl_FragColor = vec4(gray, gray, gray, 1.0);" +
            "}"
}
```

Key code:

```
vec4 color = texture2D(uTexture, vCoordinate);
float gray = (color.r + color.g + color.b)/3.0;
gl_FragColor = vec4(gray, gray, gray, 1.0);
```

Make a simple mean of rgb, and then assign it to rgb and assign it to this mean, you can get a black and white color. Then assign it to the fragment, and a simple black and white filter is done. so easy ~

Of course, many filters are not so simple. You can look at the filters implemented by others, such as [OpenGL ES Introduction: Filters - Zoom, Out of Body, Jitter] and so on.

4. Reference articles

OpenGL Learning Series---Coordinate System

OpenGL Learning Series---Projection Matrix

OpenGL Learning Footprints: Projection Matrix and Viewport Transformation Matrix