[Android audio and video development and upgrade: OpenGL rendering video screen] 3. OpenGL rendering multiple videos, realizing picture-in-picture

[Android audio and video development and upgrade: OpenGL rendering video screen] 3. OpenGL renders multiple videos and realizes picture-in-picture

Table of contents

In this article you can learn

> Rendering multiple video images is the basis for audio and video editing. This article will introduce how to render multiple video images into OpenGL, and how to mix, scale, and move the images.

write in front

It has been two weeks since the last update. Since there are a lot of things during this time, please forgive me, friends who pay attention to this series of articles. I will step up the code when I have time. Thank you for your attention and supervision.

Let's take a look at how to render multiple video screens in OpenGL.

1. Rendering multiple screens

In the last article , I explained in detail how to render video images through OpenGL, and perform scale correction on video images. Based on the tools packaged in the previous series of articles, it is very easy to render multiple video images in OpenGL.

The OpenGL Render above is very simple as follows:

```
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)
        mDrawer.setWorldSize(width, height)
    }

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

Only one Drawer is supported. Let's transform it here and change the Drawer to a list to support multiple drawers.

```
class SimpleRender: GLSurfaceView.Renderer {

    private val drawers = mutableListOf<IDrawer>()

    override fun onSurfaceCreated(gl: GL10?, config: EGLConfig?) {
        GLES20.glClearColor(0f, 0f, 0f, 0f)
        val textureIds = OpenGLTools.createTextureIds(drawers.size)
        for ((idx, drawer) in drawers.withIndex()) {
            drawer.setTextureID(textureIds[idx])
        }
    }

    override fun onSurfaceChanged(gl: GL10?, width: Int, height: Int) {
        GLES20.glViewport(0, 0, width, height)
        for (drawer in drawers) {
            drawer.setWorldSize(width, height)
        }
    }

    override fun onDrawFrame(gl: GL10?) {
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT or GLES20.GL_DEPTH_BUFFER_BIT)
        drawers.forEach {
            it.draw()
        }
    }

    fun addDrawer(drawer: IDrawer) {
        drawers.add(drawer)
    }
}
```

Also very simple,

1. Add an addDrawer method to add multiple drawers.

2. Set a texture ID for each drawer in onSurfaceCreated.
3. Set the display area width and height for each drawer in onSurfaceChanged.
4. In onDrawFrame, traverse all drawers and start drawing.

Next, create a new page and generate multiple decoders and renderers.

```
<?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"
xmlns:app="http://schemas.android.com/apk/res-auto">
    <android.opengl.GLSurfaceView
            android:id="@+id/gl_surface"
            android:layout_width="match_parent"
            android:layout_height="match_parent"/>
</android.support.constraint.ConstraintLayout>
```

```
class MultiOpenGLPlayerActivity: AppCompatActivity() {
    private val path = Environment.getExternalStorageDirectory().absolutePath +
"/mvtest.mp4"
    private val path2 = Environment.getExternalStorageDirectory().absolutePath +
"/mvtest_2.mp4"

    private val render = SimpleRender()

    private val threadPool = Executors.newFixedThreadPool(10)

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

    private fun initFirstVideo() {
        val drawer = VideoDrawer()
        drawer.setVideoSize(1920, 1080)
        drawer.getSurfaceTexture {
            initPlayer(path, Surface(it), true)
        }
        render.addDrawer(drawer)
    }

    private fun initSecondVideo() {
        val drawer = VideoDrawer()
        drawer.setVideoSize(1920, 1080)
        drawer.getSurfaceTexture {
            initPlayer(path2, Surface(it), false)
        }
        render.addDrawer(drawer)
    }

    private fun initPlayer(path: String, sf: Surface, withSound: Boolean) {
        val videoDecoder = VideoDecoder(path, null, sf)
        threadPool.execute(videoDecoder)
        videoDecoder.goOn()

        if (withSound) {
            val audioDecoder = AudioDecoder(path)
            threadPool.execute(audioDecoder)
            audioDecoder.goOn()
        }
    }

    private fun initRender() {
        gl_surface.setEGLContextClientVersion(2)
        gl_surface.setRenderer(render)
    }
}
```

The code is relatively simple. The rendering of two video images is added through the previously packaged decoding tool and drawing tool.

> Of course, you can add more frames to render in OpenGL.
> **And, you should have found that rendering multiple videos is actually generating multiple texture IDs, using this ID to generate a Surface rendering surface, and finally giving this Surface to the decoder MediaCodec for rendering.**

Since the two videos I use here are both 1920*1080 wide and high, you will find that only one of the two videos is displayed because they overlap.

The two screens are as follows:

first screen

second screen

2. Get a taste of video editing

Now, the two videos are superimposed together, and the bottom video cannot be seen. Then, let's change the alpha value of the above video to make it translucent. Can't we see the video below?

1) To achieve semi-permeability

First of all, in order to unify, add a new interface to IDrawer:

```
interface IDrawer {
    fun setVideoSize(videoW: Int, videoH: Int)
    fun setWorldSize(worldW: Int, worldH: Int)
    fun draw()
    fun setTextureID(id: Int)
    fun getSurfaceTexture(cb: (st: SurfaceTexture)->Unit) {}
    fun release()

    //新增调节alpha接口
    fun setAlpha(alpha: Float)
}
```

In VideoDrawer, save the value.

For the convenience of viewing, the entire VideoDrawer is posted here (if you don't want to see it, you can skip the added part below):

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

    private var mTextureId: Int = -1

    private var mSurfaceTexture: SurfaceTexture? = null

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

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

    //矩阵变换接收者
    private var mVertexMatrixHandler: 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
    // 半透值接收者
    private var mAlphaHandler: Int = -1

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

    private var mMatrix: FloatArray? = null

    private var mAlpha = 1f

    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)
    }

    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,
                        -actualRatio, actualRatio,
                        -1f, 1f,
                        3f, 5f
                    )
                } else {// 原始比例小于窗口比例，缩放宽度会导致宽度度超出，因此，宽度以窗口为准，缩放高度
                    val actualRatio = worldRatio / originRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -1f, 1f,
                        -actualRatio, actualRatio,
                        3f, 5f
                    )
                }
            } else {
                if (originRatio > worldRatio) {
                    val actualRatio = originRatio / worldRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -1f, 1f,
                        -actualRatio, actualRatio,
                        3f, 5f
                    )
                } 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,
                    3f, 5f
                )
            }
        }

        //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
        )
        //计算变换矩阵
        Matrix.multiplyMM(mMatrix, 0, prjMatrix, 0, viewMatrix, 0)
    }
}

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

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

override fun setAlpha(alpha: Float) {
    mAlpha = alpha
}

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

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

override fun draw() {
    if (mTextureId != -1) {
        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 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)

            mVertexMatrixHandler = GLES20.glGetUniformLocation(mProgram,
"uMatrix")
            mVertexPosHandler = GLES20.glGetAttribLocation(mProgram, "aPosition")
            mTextureHandler = GLES20.glGetUniformLocation(mProgram, "uTexture")
            mTexturePosHandler = GLES20.glGetAttribLocation(mProgram,
"aCoordinate")
            mAlphaHandler = GLES20.glGetAttribLocation(mProgram, "alpha")
        }
        //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() {
        mSurfaceTexture?.updateTexImage()
    }

    private fun doDraw() {
        //启用顶点的句柄
        GLES20.glEnableVertexAttribArray(mVertexPosHandler)
        GLES20.glEnableVertexAttribArray(mTexturePosHandler)
        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)
        GLES20.glVertexAttrib1f(mAlphaHandler, mAlpha)
        //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;" +
                "precision mediump float;" +
                "uniform mat4 uMatrix;" +
                "attribute vec2 aCoordinate;" +
                "varying vec2 vCoordinate;" +
                "attribute float alpha;" +
                "varying float inAlpha;" +
                "void main() {" +
                "    gl_Position = uMatrix*aPosition;" +
                "    vCoordinate = aCoordinate;" +
                "    inAlpha = alpha;" +
                "}"
    }

    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;" +
                "varying float inAlpha;" +
                "uniform samplerExternalOES uTexture;" +
                "void main() {" +
                "  vec4 color = texture2D(uTexture, vCoordinate);" +
                "  gl_FragColor = vec4(color.r, color.g, color.b, inAlpha);" +
                "}"
    }

    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
    }
}
```

In fact, there are very few changes compared to the previous renderer:

```
class VideoDrawer : IDrawer {
    // 省略无关代码......

    // 半透值接收者
    private var mAlphaHandler: Int = -1

    // 半透明值
    private var mAlpha = 1f

    override fun setAlpha(alpha: Float) {
        mAlpha = alpha
    }

    private fun createGLPrg() {
        if (mProgram == -1) {

            // 省略无关代码......

            mAlphaHandler = GLES20.glGetAttribLocation(mProgram, "alpha")

            //......
        }
        //使用OpenGL程序
        GLES20.glUseProgram(mProgram)
    }

    private fun doDraw() {

        // 省略无关代码......

        GLES20.glVertexAttrib1f(mAlphaHandler, mAlpha)

        //......
    }

    private fun getVertexShader(): String {
        return "attribute vec4 aPosition;" +
                "precision mediump float;" +
                "uniform mat4 uMatrix;" +
                "attribute vec2 aCoordinate;" +
                "varying vec2 vCoordinate;" +
                "attribute float alpha;" +
                "varying float inAlpha;" +
                "void main() {" +
                "    gl_Position = uMatrix*aPosition;" +
                "    vCoordinate = aCoordinate;" +
                "    inAlpha = alpha;" +
                "}"
    }

    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;" +
```

```
                "varying float inAlpha;" +
                "uniform samplerExternalOES uTexture;" +
                "void main() {" +
                "  vec4 color = texture2D(uTexture, vCoordinate);" +
                "  gl_FragColor = vec4(color.r, color.g, color.b, inAlpha);" +
                "}"
    }
}
```

Focus on the code for the two shaders:

In the vertex shader, an alpha variable is passed in, the value is passed in by the java code, then the vertex shader assigns this value to inAlpha, and finally to the fragment shader.

> Briefly talk about how to pass parameters to the fragment shader.
> To pass the value in Java to the fragment shader, it is not possible to pass the value directly, it needs to be passed indirectly through the vertex shader.

vertex shader input and output

enter

build-in variables, such variables are built-in parameters of opengl, which can be regarded as the drawing context information of opengl

Uniform variables: Generally used for Java programs to pass information such as transformation matrices, materials, lighting parameters, and colors. Such as: uniform mat4 uMatrix;

attribute variable: generally used to pass in some vertex data, such as vertex coordinates, normals, texture coordinates, vertex colors, etc.

output

build-in variable: the built-in variable of glsl, such as: gl_Position.

varying variable: used by vertex shaders to pass data to fragment shaders. **It should be noted that this variable must be declared in the vertex shader and fragment shader in the same way. Such as the above inAlpha.**

Fragment shader input and output

enter

build-in variable: same as vertex shader.

varying variable: used as input for vertex shader data, consistent with the vertex shader declaration

output

build-in variable: the built-in variable of glsl, such as: gl_FragColor.

Once you know how to pass values, the rest is clear at a glance.

1. Get the alpha of the vertex shader, then pass the value in before drawing.
2. In the fragment shader, modify the alpha of the color value taken from the texture. Finally, assign it to gl_FragColor for output.

Next, in MultiOpenGLPlayerAcitivity, change the translucent value of the upper screen

```
class MultiOpenGLPlayerActivity: AppCompatActivity() {

    // 省略无关代码...

    private fun initSecondVideo() {
        val drawer = VideoDrawer()
        // 设置半透值
        drawer.setAlpha(0.5f)
        drawer.setVideoSize(1920, 1080)
        drawer.getSurfaceTexture {
            initPlayer(path2, Surface(it), false)
        }
        render.addDrawer(drawer)
    }

    //...
}
```

When you think you can output a translucent picture perfectly, you will find that the picture is still not transparent. why?

Because the OpenGL blending mode is not turned on, go back to SimpleRender.

1. Turn on blending mode in onSurfaceCreated;
2. Before starting to draw each frame in onDrawFrame, clear the screen, otherwise there will be picture residue.

```
class SimpleRender: GLSurfaceView.Renderer {

    private val drawers = mutableListOf<IDrawer>()

    override fun onSurfaceCreated(gl: GL10?, config: EGLConfig?) {
        GLES20.glClearColor(0f, 0f, 0f, 0f)

        //------开启混合，即半透明---------
        // 开启很混合模式
        GLES20.glEnable(GLES20.GL_BLEND)
        // 配置混合算法
        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA)
        //------------------------------

        val textureIds = OpenGLTools.createTextureIds(drawers.size)
        for ((idx, drawer) in drawers.withIndex()) {
            drawer.setTextureID(textureIds[idx])
        }
    }

    override fun onSurfaceChanged(gl: GL10?, width: Int, height: Int) {
        GLES20.glViewport(0, 0, width, height)
        for (drawer in drawers) {
            drawer.setWorldSize(width, height)
        }
    }

    override fun onDrawFrame(gl: GL10?) {
        // 清屏，否则会有画面残留
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT or GLES20.GL_DEPTH_BUFFER_BIT)
        drawers.forEach {
            it.draw()
        }
    }

    fun addDrawer(drawer: IDrawer) {
        drawers.add(drawer)
    }
}
```

In this way, you can see a translucent video, superimposed on another video.

Translucent screen

> How about it, do you smell the stinky smell of video editing?

This is actually the most basic video editing principle. Basically, all video editing is based on shaders to transform the picture.

Next, let's look at two basic transformations: move and scale.

2) Mobile

Next, let's see how to change the position of the video by touch and drag.

As mentioned in the previous article, the shifting and scaling of pictures or videos are basically done through matrix transformation.

Android provides a method in Matrix for matrix translation:

```
/**
 * Translates matrix m by x, y, and z
in place.
 *
 * @param m matrix
 * @param mOffset index into m where
the matrix starts
 * @param x translation factor x
 * @param y translation factor y
 * @param z translation factor z
 */
public static void translateM(
        float[] m, int mOffset,
        float x, float y, float z) {
    for (int i=0 ; i<4 ; i++) {
        int mi = mOffset + i;
        m[12 + mi] += m[mi] * x + m[4
+ mi] * y + m[8 + mi] * z;
    }
}
```

In fact, it is to change the value of the last row of the 4x4 matrix.

Among them, x, y, z are the distances moved relative to the current position, respectively.

> It should be noted here that the change value of the translation is multiplied by the scaling ratio. You can figure it out with a pen and paper.

If the original matrix is a unit matrix, you can directly use the above translateM method to move the transformation.

However, in order to correct the proportion of the picture, as detailed in the previous article, the video picture is scaled, so the matrix of the current picture is not a unit matrix.

To do this, to pan the screen, you need to scale x, y, and z accordingly (otherwise, the moving distance will be changed by the scaling factor in the original matrix).

Then, there are two ways to make the screen move the normal distance:

1. Revert matrix to identity matrix -> move -> rescale
2. Use Current Matrix -> Scale Move Distance -> Move

Many people use the first, and here the second.

record zoom ratio

In the previous article , it was explained how to calculate the scaling factor:

```
ratio = videoRatio * worldRatio
或
ratio = videoRatio / worldRatio
```

The scaling factor corresponding to width or height, respectively. In VideoDrawer, record the scaling factor of width and height respectively.

```
class VideoDrawer : IDrawer {

    // 省略无关代码......

    private var mWidthRatio = 1f
    private var mHeightRatio = 1f

    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) {
                    mHeightRatio = originRatio / worldRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -mWidthRatio, mWidthRatio,
                        -mHeightRatio, mHeightRatio,
                        3f, 5f
                    )
                } 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
                    mWidthRatio = worldRatio / originRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -mWidthRatio, mWidthRatio,
                        -mHeightRatio, mHeightRatio,
                        3f, 5f
                    )
                }
            } else {
                if (originRatio > worldRatio) {
                    mHeightRatio = originRatio / worldRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -mWidthRatio, mWidthRatio,
                        -mHeightRatio, mHeightRatio,
                        3f, 5f
                    )
                } 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
                    mWidthRatio = worldRatio / originRatio
                    Matrix.orthoM(
                        prjMatrix, 0,
                        -mWidthRatio, mWidthRatio,
                        -mHeightRatio, mHeightRatio,
                        3f, 5f
                    )
                }
            }

            //设置相机位置
```

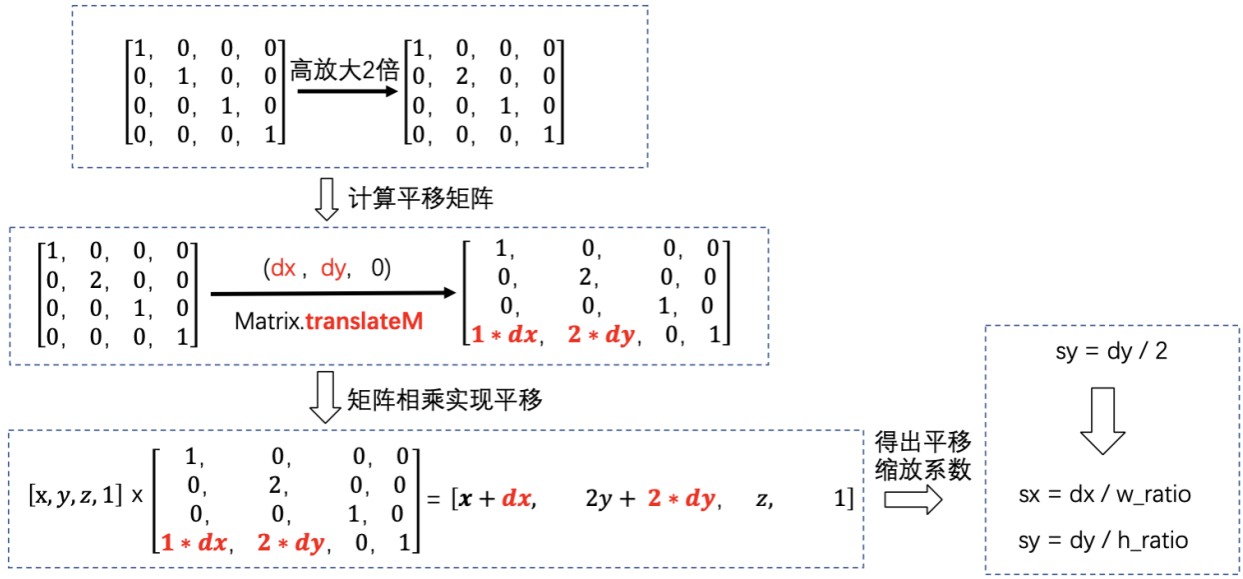
```
            val viewMatrix = FloatArray(16)
            Matrix.setLookAtM(
                viewMatrix, 0,
                0f, 0f, 5.0f,
                0f, 0f, 0f,
                0f, 1.0f, 0f
            )
            //计算变换矩阵
            Matrix.multiplyMM(mMatrix, 0, prjMatrix, 0, viewMatrix, 0)
        }
    }

    // 平移
    fun translate(dx: Float, dy: Float) {
        Matrix.translateM(mMatrix, 0, dx*mWidthRatio*2, -dy*mHeightRatio*2, 0f)
    }

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

In the code, according to the scaling width or height, the corresponding width and height scaling ratios are recorded respectively.

Next, in the translate method, dx and dy are scaled respectively. So how did the scaling come about?

Calculate mobile zoom ratio

First, let's see how the normal matrix translation calculates the scaling.



Ordinary matrix translation scaling factor calculation

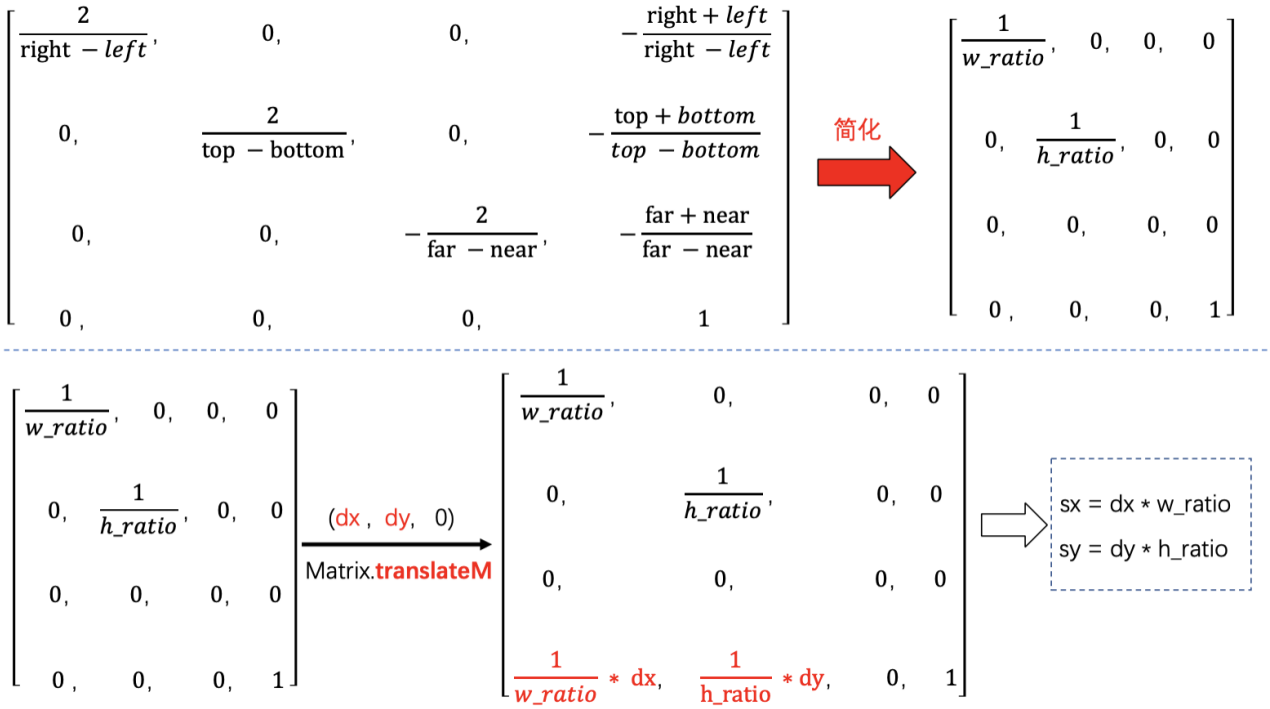
It can be seen that after a unit matrix is enlarged by 2 times in the Y direction, after Matrix.translateM transformation, the actual translation distance is twice the original.

Then in order to restore the moving distance back, this multiple needs to be removed.

end up with:

```
sx = dx / w_ratio
sy = dy / h_ratio
```

Next, let's see how to calculate the moving zoom factor of the OpenGL video screen.



Screen movement zoom factor calculation

The first is that the matrix is an OpenGL orthogonal projection matrix. We already know that left and right, top and bottom are inverses of each other, and are equal to the scaling ratios of the video screen w_ratio, h_ratio (not clear, please see the previous article), So it can be simplified to the matrix on the right.

After conversion by Matrix.translateM, the resulting translations are:

```
x方向：1/w_ratio * dx

y方向：1/h_ratio * dy
```

Therefore, the correct amount of translation can be derived as:

```
sx = dx * w_ratio

sy = dy * h_ratio
```

But why are the translation coefficients in the code all multiplied by 2? which is

```
fun translate(dx: Float, dy: Float) {
    Matrix.translateM(mMatrix, 0, dx*mWidthRatio*2, -dy*mHeightRatio*2, 0f)
}
```

First of all, what do the dx and dy refer to here?

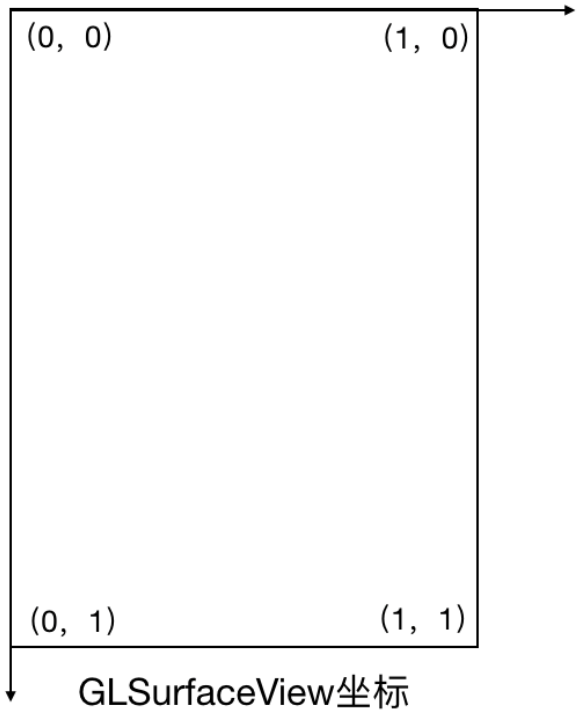
```
dx = (curX - prevX) / GLSurfaceView_Width

dy = (curY - prevY) / GLSurfaceView_Height
```

其中，
curX/curY：为当前手指触摸点的x/y坐标
pervX/prevY：为上一个手指触摸点的x/y坐标

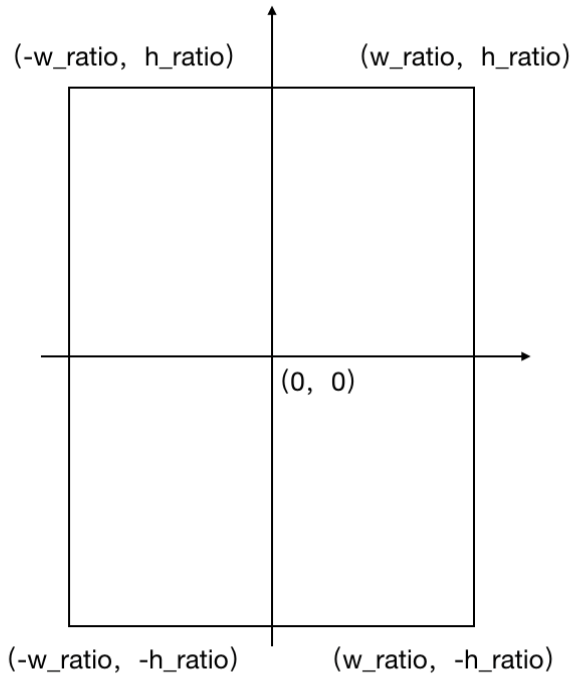
That is, dx and dy are normalized distances, ranging from 0 to 1.

Corresponds to the world coordinates of OpenGL:

```
x方向为 (left, right) -> (-w_ratio, w_ratio)

y方向为 (top, bottom) ->(-h_ratio, h_ratio)
```



GLSurfaceView coordinates



OpenGL世界坐标

OpenGL world coordinates

In fact, the world coordinate width of the entire OpenGL is: 2 times the w_ratio; height is 2 times the h_ratio. Therefore, to convert the actual (0~1) into the distance in the corresponding world coordinates, it needs to be multiplied by 2 to get the correct moving distance.

> Finally, another point to note is that the translation in the y direction is preceded by a negative sign, because the positive direction of the Y axis of the Android screen is down, while the direction of the OpenGL world coordinate Y axis is up, just the opposite.

Get the touch distance and pan the screen

In order to get the touch point of the finger, you need to customize a GLSurfaceView.

```
class DefGLSurfaceView : GLSurfaceView {

    constructor(context: Context): super(context)

    constructor(context: Context, attrs: AttributeSet): super(context, attrs)

    private var mPrePoint = PointF()

    private var mDrawer: VideoDrawer? = null

    override fun onTouchEvent(event: MotionEvent): Boolean {
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                mPrePoint.x = event.x
                mPrePoint.y = event.y
            }
            MotionEvent.ACTION_MOVE -> {
                val dx = (event.x - mPrePoint.x) / width
                val dy = (event.y - mPrePoint.y) / height
                mDrawer?.translate(dx, dy)
                mPrePoint.x = event.x
                mPrePoint.y = event.y
            }
        }
        return true
    }

    fun addDrawer(drawer: VideoDrawer) {
        mDrawer = drawer
    }
}
```

The code is very simple. In order to facilitate the demonstration, only one drawer is added, and it is not used to judge whether the finger touches the actual screen position. As long as there is a touch movement, the screen is panned.

and then put it on the page using

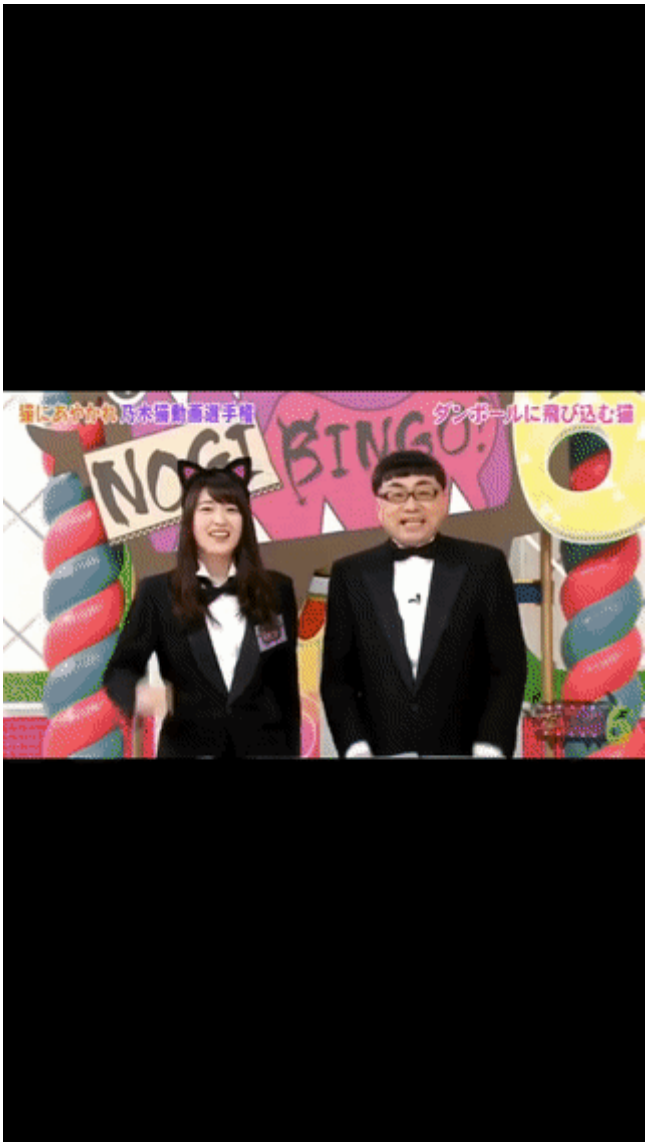
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    <com.cxp.learningvideo.opengl.DefGLSurfaceView
    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

Finally, call addDrawer in Activity and set the drawer of the above picture to DefGLSurfaceView.

```
private fun initSecondVideo() {
    val drawer = VideoDrawer()
    drawer.setVideoSize(1920, 1080)
    drawer.getSurfaceTexture {
        initPlayer(path2, Surface(it), false)
    }
    render.addDrawer(drawer)

    //设置绘制器，用于触摸移动
    gl_surface.addDrawer(drawer)
}
```

In this way, you can move the screen at will.

mobile screen



3) Zoom

It is much simpler than moving and zooming.

Android's Matrix provides a matrix scaling method:

```
/**
 * Scales matrix m in place by sx,
sy, and sz.
 *
 * @param m matrix to scale
 * @param mOffset index into m where
the matrix starts
 * @param x scale factor x
 * @param y scale factor y
 * @param z scale factor z
 */
public static void scaleM(float[] m,
int mOffset,
        float x, float y, float z) {
    for (int i=0 ; i<4 ; i++) {
        int mi = mOffset + i;
        m[     mi] *= x;
        m[ 4 + mi] *= y;
        m[ 8 + mi] *= z;
    }
}
```

This method is also very simple, which is to multiply the scaled position of the matrix corresponding to x, y, and z by the scaling factor.

Add a scaling method scale in VideoDrawer:

```
class VideoDrawer : IDrawer {

    // 省略无关代码.......

    fun scale(sx: Float, sy: Float) {
        Matrix.scaleM(mMatrix, 0, sx, sy, 1f)
        mWidthRatio /= sx
        mHeightRatio /= sy
    }

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

One thing to note here is that when the scaling factor is set, the scaling factor should be accumulated into the scaling factor of the original projection matrix, so that the moving distance can be scaled correctly when panning.

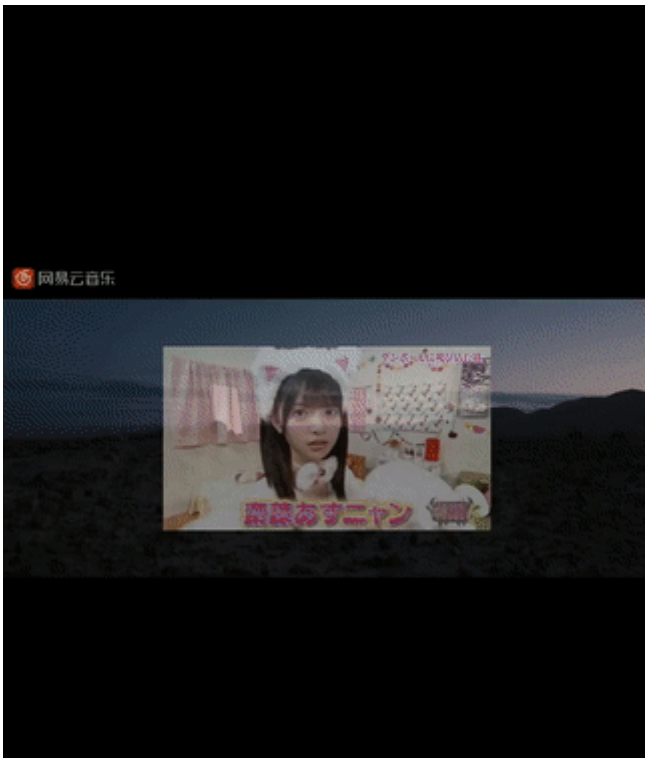
> Note: this is (original scaling factor/factor to be scaled), not "multiply". Because the scaling ratio of the scaling projection matrix is "the bigger the scale, the smaller the scale" (you can look at the matrix of the orthogonal projection again, left, right, top, bottom are the denominators)

Finally, set a zoom factor for the screen, such as 0.5f.

```
private fun initSecondVideo() {
    val drawer = VideoDrawer()
    drawer.setAlpha(0.5f)
    drawer.setVideoSize(1920, 1080)
    drawer.getSurfaceTexture {
        initPlayer(path2, Surface(it), false)
    }
    render.addDrawer(drawer)
    gl_surface.addDrawer(drawer)

    // 设置缩放系数
    Handler().postDelayed({
        drawer.scale(0.5f, 0.5f)
    }, 1000)
}
```

The effect is as follows:

zoom move



3. Aftermath

The above is the most basic knowledge used in audio and video development, but don't underestimate this knowledge. Many cool effects are actually based on these simplest transformations. I hope everyone can gain something.