[Android audio and video development and upgrade: OpenGL rendering video screen] 5. OpenGL FBO data buffer

[Android audio and video development and upgrade: OpenGL rendering video screen articles] 5. OpenGL FBO data buffer

Table of contents

In this article you can learn

> This article will explain how to use FBOs, what effects FBOs can achieve, and how to use multiple texture units in shaders.

Let's first take a look at the out-of-body effect achieved by using FBO:

out of body

1. The difference between FBO and EGL off-screen rendering

In the previous article, I explained how to use EGL, and mentioned that EGL can create a buffer for off-screen rendering. This off-screen rendering method is usually used to simulate the entire rendering window. For example, it can be used for FFmpeg soft coding, which will display The picture in the virtual window is encoded as H264.

At the same time, OpenGL also provides another way of off-screen rendering, namely FBO. FBO can not only achieve off-screen rendering of the entire OpenGL window, but also can be used to process fragmented pictures, that is, small pictures in the window.

The off-screen rendering of EGL will be used in the article about FFmpeg later, so let's leave it alone here.

In video editing, FBO off-screen rendering plays a very important role. Many video filters are used. Let's take a look at how to use FBO.

2. Introduction to FBO

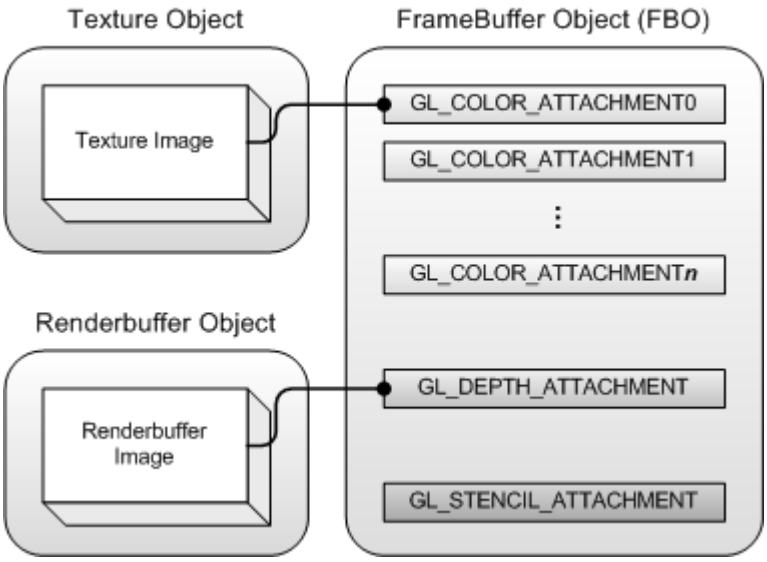
> OpenGL will send data to `FBO` the that is to say, `FBO` it has been serving us silently. So, OpenGL creates a default FBO right from the start.

FBO: Frame Buffer Object, frame buffer object.

From the name, it is often easy to misunderstand that this is a cache space, but in fact, FBO is very important on the last Object. This is a cache object that contains multiple `缓冲索引`, `颜色缓冲（Color buffers）`, `深度缓冲（Depth buffer）`, `模板缓冲（Stencil buffer）`.

The reason why it is said to be a buffer index is because the FBO does not contain these buffer data, but only saves the index address of the buffer data.

FBOs and these buffers are connected by attachment points.



You can see that the FBO contains:

```
1. 多个颜色附着点（GL_COLOR_ATTACHMENT0、GL_COLOR_ATTACHMENT1...）
2. 一个深度附着点（GL_DEPTH_ATTACHMENT）
3. 一个模板附着点（GL_STENCIL_ATTACHMENT）
```

Can be divided into two categories:

Texture Attachment (Color Attachment) : Primarily used to render colors into textures.

Render Buffer Object RBO (Render Buffer Object) : Mainly used for rendering depth information and template information.

> In 2D, only the color attachment is usually used, the other two attachments are usually used in 3D rendering.

As mentioned above, FBO can be used for off-screen rendering. Let's take a look at how to render the picture to a "background" texture through FBO.

> The background here refers to the texture that is not used to display to the window.

3. How to use FBO

1. Create a new texture

```
fun createFBOTexture(width: Int, height: Int): IntArray {
    // 新建纹理ID
    val textures = IntArray(1)
    GLES20.glGenTextures(1, textures, 0)

    // 绑定纹理ID
    GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textures[0])

    // 根据颜色参数，宽高等信息，为上面的纹理ID，生成一个2D纹理
    GLES20.glTexImage2D(GLES20.GL_TEXTURE_2D, 0, GLES20.GL_RGBA, width, height,
        0, GLES20.GL_RGBA, GLES20.GL_UNSIGNED_BYTE, null)

    // 设置纹理边缘参数
    GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MIN_FILTER,
GLES20.GL_NEAREST.toFloat())
    GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_MAG_FILTER,GLES20.GL_LINEAR.toFloat())
    GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_CLAMP_TO_EDGE.toFloat())
    GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_CLAMP_TO_EDGE.toFloat())

    // 解绑纹理ID
    GLES20.glBindTexture(GLES20.GL_TEXTURE_2D,0)
    return textures
}
```

Generating a texture for an FBO is almost the same as a normal texture.

First , generate a texture ID and bind it to OpenGL.

Second , generate a corresponding texture for this texture ID.

> What is used here is `GLES20.glTexImage2D` , when rendering image textures, is used `GLUtils.texImage2D` .

> **Regarding the width and height of the texture created, here is a description:**
> FBO creates a virtual window, so the size can be set according to your own needs, which can be larger than the actual system window. In order to make the video screen ratio normal, you can set the width and height of the OpenGL window and the width and height of the texture to the width and height of the video. Therefore, when OpenGL is rendering, we don't need to correct the scale through matrix transformation, and we can directly stretch it.

Finally , set the texture edge parameters, then unbind.

2. Create a new FrameBuffer

```
fun createFrameBuffer(): Int {
    val fbs = IntArray(1)
    GLES20.glGenFramebuffers(1, fbs, 0)
    return fbs[0]
}
```

Creating a new FrameBuffer is similar to creating a new texture ID, and finally returns the FBO index

3. Bind FBO

```
fun bindFBO(fb: Int, textureId: Int) {
    GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, fb)
    GLES20.glFramebufferTexture2D(GLES20.GL_FRAMEBUFFER,
GLES20.GL_COLOR_ATTACHMENT0,
        GLES20.GL_TEXTURE_2D, textureId, 0)
}
```

First bind the FBO created above, and then `GLES20.GL_COLOR_ATTACHMENT0` bind .

4. Unbind FBO

```
fun unbindFBO() {
    GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, GLES20.GL_NONE)
    GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, 0)
}
```

Unbinding the FBO is relatively simple, in fact, it is to bind the FBO to the default window.

> The one here is `GLES20.GL_NONE` actually `0` the FBO of the default window of the system.

5. Delete FBO

```
fun deleteFBO(frame: IntArray, texture:IntArray) {
    //删除Frame Buffer
    GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, GLES20.GL_NONE)
    GLES20.glDeleteFramebuffers(1, frame, 0)
    //删除纹理
    GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, 0)
    GLES20.glDeleteTextures(1, texture, 0)
}
```

The above is actually the process of using FBO:

1. new texture
2. New FBO
3. Binding attaches the texture to the color attachment point of the FBO
4. **【Rendering】**
5. Unbind FBO
6. delete FBO

Except for step 4, the others are the above encapsulated methods.

So let's take a look at how to render the picture to the texture connected to the FBO.

In order to better understand the entire rendering process, the following is a very classic filter to demonstrate the rendering process.

3. Use FBO to realize the "Out of Body" filter

1. How to achieve out of body

static image out of body

out of body

This effect can be split into 3 effects:

1. Bottom static map
2. upper level zoom
3. upper layer translucent

And then split into 2 combinations:

1. Bottom static map
2. The upper layer keeps zooming in and increases transparency as it zooms in

video out of body

According to the out-of-body effect of the static image, it can be known that **the out-of-body effect of the upper layer is based on the original image** , that is to say, the basic image of the soul will not change.

And every frame of the video is changing.

Therefore, in order to achieve a smoother magnification effect for the "soul" of the upper layer, it is necessary to hold a frame for a period of time so that this frame can complete the complete magnification process.

> Here is a problem: how to save a certain frame of the video?

`FBO` is the key to solving this problem.

2. Package the FBO tool

In order to facilitate the use of FBO-related methods, we encapsulate the above methods in a static tool `OpenGLTools`.

```
object OpenGLTools {
    fun createFBOTexture(width: Int, height: Int): IntArray {
        // 新建纹理ID
        val textures = IntArray(1)
        GLES20.glGenTextures(1, textures, 0)

        // 绑定纹理ID
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textures[0])

        // 根据颜色参数，宽高等信息，为上面的纹理ID，生成一个2D纹理
        GLES20.glTexImage2D(GLES20.GL_TEXTURE_2D, 0, GLES20.GL_RGBA, width,
height,
            0, GLES20.GL_RGBA, GLES20.GL_UNSIGNED_BYTE, null)

        // 设置纹理边缘参数
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MIN_FILTER,
GLES20.GL_NEAREST.toFloat())
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_MAG_FILTER,GLES20.GL_LINEAR.toFloat())
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_CLAMP_TO_EDGE.toFloat())
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_CLAMP_TO_EDGE.toFloat())

        // 解绑纹理ID
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D,0)
        return textures
    }

    fun createFrameBuffer(): Int {
        val fbs = IntArray(1)
        GLES20.glGenFramebuffers(1, fbs, 0)
        return fbs[0]
    }

    fun bindFBO(fb: Int, textureId: Int) {
        GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, fb)
        GLES20.glFramebufferTexture2D(GLES20.GL_FRAMEBUFFER,
GLES20.GL_COLOR_ATTACHMENT0,
            GLES20.GL_TEXTURE_2D, textureId, 0)
    }

    fun unbindFBO() {
        GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, GLES20.GL_NONE)
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, 0)
    }

    fun deleteFBO(frame: IntArray, texture:IntArray) {
        //删除Frame Buffer
        GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, GLES20.GL_NONE)
        GLES20.glDeleteFramebuffers(1, frame, 0)
        //删除纹理
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, 0)
        GLES20.glDeleteTextures(1, texture, 0)
    }
}
```

3. In the video renderer, connect to FBO

New renderer `SoulVideoDrawer`

Here, the previous VideoDrawer is copied directly. If you have read the previous articles, you should be familiar with VideoDrawer. So I won't post the full code here. For details, please check the previous article, or look directly at the source code: VideoDrawer .

In fact, `SoulVideoDrawer` most of the code is the `VideoDrawer` same, here is the complete source code: SoulVideoDrawer .

This time, instead of posting the complete code at one time as before, let's take a step-by-step look at how to use FBO.

```
class SoulVideoDrawer : IDrawer {

    // ......

    // 省略和VideoDrawer一样成员变量

    // ......

//-------------灵魂出窍相关的变量--------------

    /**上下颠倒的顶点矩阵*/
    private val mReserveVertexCoors = floatArrayOf(
        -1f, 1f,
        1f, 1f,
        -1f, -1f,
        1f, -1f
    )

    private val mDefVertexCoors = floatArrayOf(
        -1f, -1f,
        1f, -1f,
        -1f, 1f,
        1f, 1f
    )

    // 顶点坐标
    private var mVertexCoors = mDefVertexCoors

    // 灵魂帧缓冲
    private var mSoulFrameBuffer: Int = -1

    // 灵魂纹理ID
    private var mSoulTextureId: Int = -1

    // 灵魂纹理接收者
    private var mSoulTextureHandler: Int = -1

    // 灵魂缩放进度接收者
    private var mProgressHandler: Int = -1

    // 是否更新FBO纹理
    private var mDrawFbo: Int = 1

    // 更新FBO标记接收者
    private var mDrawFobHandler: Int = -1

    // 一帧灵魂的时间
    private var mModifyTime: Long = -1

    override fun draw() {
        if (mTextureId != -1) {
            initDefMatrix()
            //【步骤1: 创建、编译并启动OpenGL着色器】
            createGLPrg()

            // -------【步骤2:新增FBO部分】-----
```

```
            //【步骤2.1: 更新灵魂纹理】
            updateFBO()
            //【步骤2.2: 激活灵魂纹理单元】
            activateSoulTexture()
            // ---------------------------

            //【步骤3: 激活并绑定纹理单元】
            activateDefTexture()
            //【步骤4: 绑定图片到纹理单元】
            updateTexture()
            //【步骤5: 开始渲染绘制】
            doDraw()
        }
    }

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

Added member variables related to FBO and out-of-body effects.

Focus on the `draw` method , there are 5 steps, but the real increase is actually the second step:

```
步骤2: 新增FBO部分
- 2.1: 更新灵魂纹理【updateFBO】
- 2.2: 激活灵魂纹理单元【activateSoulTexture】
```

Let's look at 2.1 first.

> Update textures attached to FBOs

```
class SoulVideoDrawer : IDrawer {

    // ......

    private fun updateFBO() {
        //【1，创建FBO纹理】
        if (mSoulTextureId == -1) {
            mSoulTextureId = OpenGLTools.createFBOTexture(mVideoWidth,
mVideoHeight)
        }
        //  【2，创建FBO】
        if (mSoulFrameBuffer == -1) {
            mSoulFrameBuffer = OpenGLTools.createFrameBuffer()
        }
        //  【3，渲染到FBO】
        if (System.currentTimeMillis() - mModifyTime > 500) {
            mModifyTime = System.currentTimeMillis()
            // 绑定FBO
            OpenGLTools.bindFBO(mSoulFrameBuffer, mSoulTextureId)
            // 配置FBO窗口
            configFboViewport()

//--------执行正常画面渲染，画面将渲染到FBO上--------------

            // 激活默认的纹理
            activateDefTexture()
            // 更新纹理
            updateTexture()
            // 绘制到FBO
            doDraw()

//--------------------------------------------------

            // 解绑FBO
            OpenGLTools.unbindFBO()
            // 恢复默认绘制窗口
            configDefViewport()
        }
    }

    /**
     * 配置FBO窗口
     */
    private fun configFboViewport() {
        mDrawFbo = 1
        // 将变换矩阵回复为单位矩阵（将画面拉升到整个窗口大小，设置窗口比例和FBO纹理比例一致，
画面刚好可以正常绘制到FBO纹理上）
        Matrix.setIdentityM(mMatrix, 0)
        // 设置颠倒的顶点坐标
        mVertexCoors = mReserveVertexCoors
        //重新初始化顶点坐标
        initPos()
        GLES20.glViewport(0, 0, mVideoWidth, mVideoHeight)
        //设置一个颜色状态
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 0.0f)
        //使能颜色状态的值来清屏
```

```
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT)
    }

    /**
     * 配置默认显示的窗口
     */
    private fun configDefViewport() {
        mDrawFbo = 0
        mMatrix = null
        // 恢复顶点坐标
        mVertexCoors = mDefVertexCoors
        initPos()
        initDefMatrix()
        // 恢复窗口
        GLES20.glViewport(0, 0, mWorldWidth, mWorldHeight)
    }

    private fun activateDefTexture() {
        activateTexture(GLES11Ext.GL_TEXTURE_EXTERNAL_OES, mTextureId, 0,
mTextureHandler)
    }

    private fun activateSoulTexture() {
        activateTexture(GLES11.GL_TEXTURE_2D, mSoulTextureId, 1,
mSoulTextureHandler)
    }

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

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

See the `updateFBO` method , 3 steps:

1. Create textures
2. Create FBO
3. Render the image onto the FBO's texture

The first two steps have been introduced before and will not be repeated here.

Focus on step 3.

Here, a frame of image is kept for 500ms. We use a variable `mModifyTime` to record the time when the current frame is rendered. As long as 500ms pass, the picture will be refreshed once.

Let's take a look at the process of rendering FBO:

```
if (System.currentTimeMillis() - mModifyTime > 500) {
    // 记录时间
    mModifyTime = System.currentTimeMillis()
    // 绑定FBO
    OpenGLTools.bindFBO(mSoulFrameBuffer, mSoulTextureId)
    // 配置FBO窗口
    configFboViewport()
//--------执行正常画面渲染，画面将渲染到FBO上--------------
    // 激活默认的纹理
    activateDefTexture()
    // 更新纹理
    updateTexture()
    // 绘制到FBO
    doDraw()
//-------------------------------------------------
    // 解绑FBO
    OpenGLTools.unbindFBO()
    // 恢复默认绘制窗口
    configDefViewport()
}
```

i. Bind FBO

> `OpenGLTools.bindFBO` After calling , all operations on OpenGL will affect the FBO we created. That is to say, before `OpenGLTools.unbindFBO()` calling Unbind FBO, all the following operations will act on FBO.

ii. Reconfigure the FBO window size

> Set the OpenGL window to the video size, reset the matrix changes (pull the screen to the size of the window), and clear the screen.

> As for why you need to reset the window size, I have already said that when setting the texture size.

> Another point to note is that the texture coordinates `mVertexCoors` are (in fact, they are restored to the default coordinates of OpenGL), so that after rendering to the texture bound to the FBO, the color can be picked normally in the fragment shader.

code show as below:

```
private fun configFboViewport() {
    mDrawFbo = 1
    // 将变换矩阵恢复为单位矩阵
    //（将画面拉升到整个窗口大小，
    // 设置窗口宽高和FBO纹理宽高一致，
    // 画面刚好可以正常绘制到FBO绑定的纹理上）
    Matrix.setIdentityM(mMatrix, 0)
    // 设置颠倒的顶点坐标
    mVertexCoors = mReserveVertexCoors
    //重新初始化顶点坐标
    initPos()
    // 设置窗口大小
    GLES20.glViewport(0, 0, mVideoWidth, mVideoHeight)
    //设置一个颜色状态
    GLES20.glClearColor(0.0f, 0.0f, 0.0f, 0.0f)
    //使能颜色状态的值来清屏
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT)
}
```

iii. Activate and update the original texture of the video

> Note that here is the texture that activates the original rendered video

iv. Rendering

> That is to say, after the FBO is bound, the picture can be rendered to the FBO according to the normal rendering process.

v. Unbind the FBO and restore the window size, texture coordinates, and matrix to the original configuration.

> Switch the rendering back to the original system window, and the screen will be displayed on the system window again.

Through the `mSoulTextureId` above .

4. Realize the effect of out of body

In the front, we rendered a frame to `mSoulTextureId` this texture, and then we will use this texture to enlarge the picture and achieve a transparent gradient to achieve the soul effect.

Back in the draw method, come to step 2.2.

```
override fun draw() {
    if (mTextureId != -1) {
        //【步骤1: 创建、编译并启动OpenGL着色器】
        // -------【步骤2:新增FBO部分】-----
        //【步骤2.1: 更新灵魂纹理】
        //【步骤2.2: 激活灵魂纹理单元】
        activateSoulTexture()
        // ---------------------------

        //【步骤3: 激活并绑定纹理单元】
        activateDefTexture()
        //【步骤4: 绑定图片到纹理单元】
        updateTexture()
        //【步骤5: 开始渲染绘制】
        doDraw()
    }
}
```

See how activation activates the "soul" texture.

> Pass multiple textures to shaders

```
private fun activateSoulTexture() {
    activateTexture(GLES11.GL_TEXTURE_2D, mSoulTextureId, 1, mSoulTextureHandler)
}

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

Slightly different from the previous article, the previous parameters were directly written to death. This time, it has been modified to `activateTexture` pass `纹理类型`, `纹理ID`, `纹理单元索引`, and the corresponding shader `纹理接收器` as parameters.

There are 2 points to note:

1. **About texture type** . `activateSoulTexture` In , it should be noted that the type of the texture is the normal texture type `GLES11.GL_TEXTURE_2D` , not the extended texture `GLES11Ext.GL_TEXTURE_EXTERNAL_OES` , because after the previous rendering, the picture is already a normal texture.
2. **About texture units** . As said in OpenGL basics , OpenGL has multiple texture units built in and can be used at the same time. So here, the texture unit of the normal picture is set to the default, `GLES20.GL_TEXTURE0` and the texture unit of the "soul" is `GLES20.GL_TEXTURE1 = GLES20.GL_TEXTURE0 + 1` .

Next, activate the default normal screen texture `updateTexture()` so that both texture units can be received in the fragment shader.

```
private fun activateDefTexture() {
    activateTexture(GLES11Ext.GL_TEXTURE_EXTERNAL_OES, mTextureId, 0,
mTextureHandler)
}
```

rendering

Finally, start rendering and draw and enter the shader.

Out of Body shader

So much foreshadowing has been done before, in fact, to pass a fixed frame of video to the shader. The real "soul out of body" effect is also in the fragment shader.

The shader code is as follows:

```
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;" +
            "uniform float progress;" +
            "uniform int drawFbo;" +
            "uniform sampler2D uSoulTexture;" +
            "void main() {" +
                // 透明度[0,0.4]
                "float alpha = 0.6 * (1.0 - progress);" +
                // 缩放比例[1.0,1.5]
                "float scale = 1.0 + (1.5 - 1.0) * progress;" +

                // 放大纹理坐标
                "float soulX = 0.5 + (vCoordinate.x - 0.5) / scale;\n" +
                "float soulY = 0.5 + (vCoordinate.y - 0.5) / scale;\n" +
                "vec2 soulTextureCoords = vec2(soulX, soulY);" +
                // 获取对应放大纹理坐标下的像素(颜色值rgba)
                "vec4 soulMask = texture2D(uSoulTexture, soulTextureCoords);" +

                "vec4 color = texture2D(uTexture, vCoordinate);" +

                "if (drawFbo == 0) {" +
                    // 颜色混合 默认颜色混合方程式 = mask * (1.0-alpha) + weakMask *
alpha
                "    gl_FragColor = color * (1.0 - alpha) + soulMask * alpha;" +
                "} else {" +
                "   gl_FragColor = vec4(color.r, color.g, color.b, inAlpha);" +
                "}" +
            "}"
}
```

As you can see, 顶点着色器 the code is the same as normal rendering.

Modifications are there 片元着色器中 .

Briefly analyze:

i. In addition to the parameters required for normal screen rendering, three new parameters have been added:

```
// 动画进度
uniform float progress;
// 是否绘制到FBO
uniform int drawFbo;
// 一帧固定的纹理
uniform sampler2D uSoulTexture;
```

ii. Skip the middle part about the "soul" animation and watch the last one first `if/else`

```
if (drawFbo == 0) {
    // 颜色混合 默认颜色混合方程式 = mask * (1.0-alpha) + weakMask * alpha
    gl_FragColor = color * (1.0 - alpha) + soulMask * alpha;" +
} else {
   gl_FragColor = vec4(color.r, color.g, color.b, inAlpha);
}
```

When the time of a frame exceeds 500ms, a new frame of video will be re-acquired.

Here, `drawFbo` if `1` is , the normal picture will be rendered. At this time, since the FBO has been bound, this frame of picture will be rendered `mSoulTextureID` on .

On the next render, this frame texture will be passed to the fragment shader `uSoulTexture` .

iii. The middle section, on the core of "Out of Body".

```
// 透明度[0,0.4]
float alpha = 0.6 * (1.0 - progress);
// 缩放比例[1.0,1.5]
float scale = 1.0 + (1.5 - 1.0) * progress;

// 放大纹理坐标
float soulX = 0.5 + (vCoordinate.x - 0.5) / scale;
float soulY = 0.5 + (vCoordinate.y - 0.5) / scale;
vec2 soulTextureCoords = vec2(soulX, soulY);

// 获取对应放大纹理坐标下的像素(颜色值rgba)
vec4 soulMask = texture2D(uSoulTexture, soulTextureCoords);
```

First, calculate the transparency. According to the calculation from the outside `progress` , slowly reduce the transparency, the maximum transparency is 0.6.

Then, the scaled coordinates are calculated. `progress` As increases, `scale` the larger. Maximum magnification 1.5 times. Use `scale` to calculate X, Y scaling respectively . As you can see, **`scale` the bigger `soulX/soulY` it is, the smaller it is** . This is because to achieve the effect of magnification, the current point to be rendered should take the color (pixel) corresponding to the smaller coordinate.

Finally, go `soulX` `soulY` to the "Soul" texture to `uSoulTexture` get the color.

iv. Mix the bottom normal picture and the upper "soul" picture, using a common blending algorithm.

```
gl_FragColor = color * (1.0 - alpha) + soulMask * alpha;
```

5. Access the renderer in the page

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

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

    private fun initRender() {
        // 使用“灵魂出窍”渲染器
        drawer = SoulVideoDrawer()
        drawer.setVideoSize(1920, 1080)
        drawer.getSurfaceTexture {
            initPlayer(Surface(it))
        }
        gl_surface.setEGLContextClientVersion(2)
        val render = SimpleRender()
        render.addDrawer(drawer)
        gl_surface.setRenderer(render)
    }

    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()
    }
}
```

The use is exactly the same as the normal use of the OpenGL renderer, the only difference is that it is `VideoDrawer` replaced `SoulVideoDrawer` by .

Finally got the effect at the beginning of the article:

out of body

4. Summary

The above is the whole process of using FBO, and it is very simple to use. Of course, I only focus on the color attachment part, other depth attachment and template attachment are interested in exploring and learning by themselves.

It can be seen that FBO provides us with a good way to realize video processing, many cool effects can be realized, and more interesting effects are waiting for everyone to realize.

Reference article

Frame Buffer Object (FBO) implements Render To Texture/RTT

DEPTH_TEST (depth buffer test)

Stencil_TEST (stencil buffer test)

Getting Started with OpenGL ES: Filters - Zoom, Out of Body, Jitter, etc.