
# 《基于 Andropid 源生 AR 应用开发》

## 实验手册 06

版 本 1.0

文档提供：智能设备教研室 丁盟

# 目录

# 第6章　3D 模型动画

## 6.1 实验目的

**目的一：** 掌握.obj 格式的存储规则以及读取方法。

**目的二：** 掌握触摸旋转模型的方法。

**目的三：** 掌握 3D 动画的实现方法。

**目的四：** 掌握从外部文件读取创建着色器的方法。

## 6.2 准备工作

**准备一：** .obj 文件的存储格式

打开.obj 文件可以看到如下的类似内容。(.obj 文件中"#"号是作为注释符的)。

```
v  7.268521 0.000000 -2.808495
v  43.775921 3.692307 -11.804825
v  40.087490 4.896347 -20.872227
……
vt  0.500000 0.000000 0.000000
vt  0.750000 0.000000 1.000000
vt  0.791667 0.000000 1.000000
……
vn  0.061305 -0.985877 -0.155850
vn  0.176545 -2.839121 -0.448815
vn  0.176545 -2.839120 -0.448815
……
g (null)
s 1
f 1/1/1 3/3/3 2/2/2
f 1/1/1 4/4/4 3/3/3
f 1/1/1 5/5/5 4/4/4
……
```

从上面可以看出一般的 obj 文件中数据的格式是这样的:

| 前缀　　参数 1　　参数 2　　参数 3 … |
| --- |

前缀标识了这一行所存储的信息类型。参数则是具体的数据。OBJ 文件数据类型的前缀一般有：

- v　　表示本行指定一个顶点。 此前缀后跟着 3 个单精度浮点数，分别表示该定点的 X、Y、Z 坐标值;
- vt　　表示本行指定一个纹理坐标。此前缀后跟着两个单精度浮点数。分别表示此纹理坐标的 U、V 值;
- vn　　表示本行指定一个法线向量。此前缀后跟着 3 个单精度浮点数，分别表示该法向量的 X、Y、Z 坐标值;
- f　　表示本行指定一个表面(Face)。一个表面实际上就是一个三角形图元。

其中的参数 v、vt、vn 都是比较好理解的，下面着重说一下 f 类型的数据的几种格式：

1. f 1　2　3

2. f 1/3　2/5　3/4

3. f 1/3/4　2/5/6　3/4/2

第一种类型表示以顶点 1,2,3 作为索引建立三角形;

第二种类型表示以顶点 1,2,3 作为索引建立三角形，并且顶点 1 的纹理坐标为 3，第二个顶点的纹理坐标为 5，第三个顶点的纹理坐标为 4;

第三种类型表示以顶点 1,2,3 作为索引建立三角形，并且顶点 1 的纹理坐标为 3，法理坐标为 4，第二个顶点的纹理坐标为 5，法理坐标为 6，第三个顶点的纹理坐标为 4，法理坐标为 2。

**准备二：** 准备模型文件与纹理贴图文件。


其中 texture.JPG 为纹理贴图文件，三个.obj 文件为三个状态的模型文件。

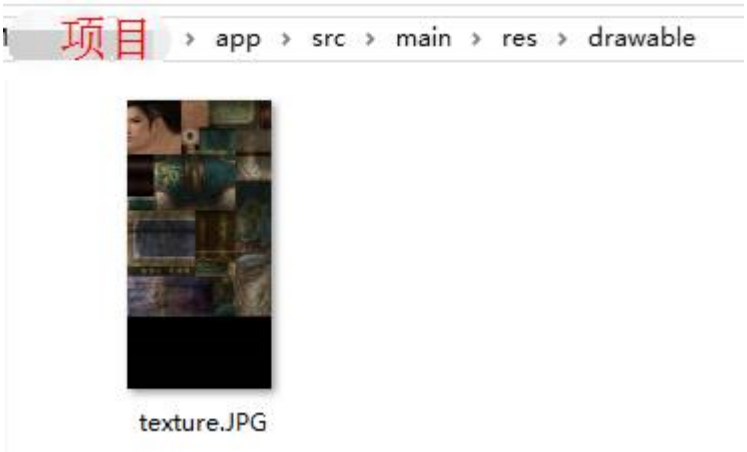
## 6.3 实验步骤

**步骤一** 在 Android Studio 中创建一个新的 Android 项目，在新创建的项目的 AndroidManifest.xml 文件中添加使用到的 OpenGL ES 版本信息，放在 manifest 节点下。

```
<uses-feature android:glEsVersion="0x00020000"
android:required="true" />
```

**步骤二** 在项目的“app\src\main\”路径下创建 assets 资源文件夹，并将之前准备好的模型文件（.obj）放入该文件夹下。


**步骤三** 将准备好的纹理贴图文件放入项目的“app\src\main\res\drawable\”路径下。




**步骤四** 编写 vertex.sh 文件用来保存顶点着色器的源码，编写完成之后放入 assets 资源文件夹中。顶点着色器源码如下：

```
// vertex.sh    顶点着色器源码
```

```
uniform mat4 uMVPMatrix;        // 总变换矩阵
attribute vec3 aPosition1;      // 顶点位置
attribute vec3 aPosition2;      // 顶点位置
attribute vec3 aPosition3;      // 顶点位置
attribute vec2 aTextureCoord;   // 顶点纹理坐标
uniform float uProportion;      // 变化百分比
varying vec2 vTextureCoord;

void main()
{
    vec3 tv;
    if(uProportion <= 1.0)
    {
        tv = mix(aPosition1, aPosition2, uProportion);
    }
    else
    {
       tv = mix(aPosition2, aPosition3, uProportion - 1.0);
    }
    gl_Position = uMVPMatrix * vec4(tv,1);

    // 将接收的纹理映射坐标传递给片元着色器
    vTextureCoord = aTextureCoord;
}
```

**步骤五** 编写 frag.sh 文件用来保存片元着色器的源码，编写完成之后放入 assets 资源文件夹中。片元着色器源码如下：

```
// frag.sh    片元着色器源码
precision mediump float;    // 设置 float 为中等精度
uniform sampler2D uTexture; // 纹理内容数据
varying vec2 vTextureCoord; // 接收从顶点着色器过来的纹理映射坐标

void main()
{
   // 根据纹理映射坐标与纹理数据获取当前片元颜色
   gl_FragColor = texture2D(uTexture, vTextureCoord);
}
```

**步骤六** 在项目中添加 LoadUtil.java 文件，其中定义 LoadUtil 类用来完成.obj 模型文件的数据读取操作。LoadUtil 类的代码如下：

```
/**
 * 加载.obj 模型文件的工具类
 */
public class LoadUtil {

    // 最大S 纹理坐标
    public static final float MAX_S = 1.0f;
    // 最大T 纹理坐标
    public static final float MAX_T = 1.0f;

    /**
     * 从 obj 文件中加载携带顶点信息的物体
     * @param fname
     * @param mv
     * @return
     */
    public static float[][] loadFromFileVertexOnly(String fname,
                                                   MySurfaceView mv)
{
        // 最终返回的模型数据数组
        // modelDataArr[0]  顶点数据坐标
        // modelDataArr[1]  纹理映射坐标
        float[][] modelDataArr = new float[2][];

        // 原始顶点坐标列表--直接从 obj 文件中加载
        ArrayList<Float> alv = new ArrayList<Float>();

        // 顶点组装面索引列表--根据面的信息从文件中加载
        ArrayList<Integer> alFaceIndex = new
ArrayList<Integer>();

        // 结果顶点坐标列表--按面组织好
        ArrayList<Float> alvResult = new ArrayList<Float>();

        // 平均前各个索引对应的点的法向量集合 Map
        // 原始纹理坐标列表
        ArrayList<Float> alt = new ArrayList<Float>();

        // 纹理坐标结果列表
        ArrayList<Float> altResult = new ArrayList<Float>();

        try {
            InputStream in =
mv.getResources().getAssets().open(fname);
            InputStreamReader isr = new InputStreamReader(in);
            BufferedReader br = new BufferedReader(isr);
            String temps = null;
```

```
            // 扫面文件，根据行类型的不同执行不同的处理逻辑
            while ((temps = br.readLine()) != null) {

                // 用空格分割行中的各个组成部分
                String[] tempsa = temps.split("[ ]+");
                if (tempsa[0].trim().equals("v")) {

                    // 此行为顶点坐标
                    // 若为顶点坐标行则提取出此顶点的XYZ坐标添加到原始顶点坐标列表中
                    alv.add(Float.parseFloat(tempsa[1]));
                    alv.add(Float.parseFloat(tempsa[2]));
                    alv.add(Float.parseFloat(tempsa[3]));

                } else if (tempsa[0].trim().equals("vt")) {

                    alt.add(Float.parseFloat(tempsa[1]) * MAX_S);
                    alt.add(Float.parseFloat(tempsa[2]) * MAX_T);

                } else if (tempsa[0].trim().equals("f")) {

                    // 三个顶点索引值的数组
                    int[] index = new int[3];

                    // 计算第0个顶点的索引，并获取此顶点的XYZ三个坐标
                    index[0] =
Integer.parseInt(tempsa[1].split("/")[0]) - 1;
                    float x0 = alv.get(3 * index[0]);
                    float y0 = alv.get(3 * index[0] + 1);
                    float z0 = alv.get(3 * index[0] + 2);
                    alvResult.add(x0);
                    alvResult.add(y0);
                    alvResult.add(z0);

                    // 计算第1个顶点的索引，并获取此顶点的XYZ三个坐标
                    index[1] =
Integer.parseInt(tempsa[2].split("/")[0]) - 1;
                    float x1 = alv.get(3 * index[1]);
                    float y1 = alv.get(3 * index[1] + 1);
                    float z1 = alv.get(3 * index[1] + 2);
                    alvResult.add(x1);
                    alvResult.add(y1);
                    alvResult.add(z1);

                    // 计算第2个顶点的索引，并获取此顶点的XYZ三个坐标
```

```
                    index[2] =
Integer.parseInt(tempsa[3].split("/")[0]) - 1;
                    float x2 = alv.get(3 * index[2]);
                    float y2 = alv.get(3 * index[2] + 1);
                    float z2 = alv.get(3 * index[2] + 2);
                    alvResult.add(x2);
                    alvResult.add(y2);
                    alvResult.add(z2);

                    // 记录此面的顶点索引
                    alFaceIndex.add(index[0]);
                    alFaceIndex.add(index[1]);
                    alFaceIndex.add(index[2]);

                    // 将纹理坐标组织到结果纹理坐标列表中
                    // 第0个顶点的纹理坐标
                    int indexTex =
Integer.parseInt(tempsa[1].split("/")[1]) - 1;
                    altResult.add(alt.get(indexTex * 2));
                    altResult.add(alt.get(indexTex * 2 + 1));

                    // 第1个顶点的纹理坐标
                    indexTex =
Integer.parseInt(tempsa[2].split("/")[1]) - 1;
                    altResult.add(alt.get(indexTex * 2));
                    altResult.add(alt.get(indexTex * 2 + 1));

                    // 第2个顶点的纹理坐标
                    indexTex =
Integer.parseInt(tempsa[3].split("/")[1]) - 1;
                    altResult.add(alt.get(indexTex * 2));
                    altResult.add(alt.get(indexTex * 2 + 1));
                }
            }

            // 生成顶点数组
            int size = alvResult.size();
            float[] vXYZ = new float[size];
            for (int i = 0; i < size; i++) {
                vXYZ[i] = alvResult.get(i);
            }

            // 生成纹理数组
            size = altResult.size();
            float[] tST = new float[size];
            for (int i = 0; i < size; i++) {
                if (i % 2 == 1) {
                    tST[i] = 1 - altResult.get(i);
```

```
                } else
                    tST[i] = altResult.get(i);
            }

            modelDataArr[0] = vXYZ;
            modelDataArr[1] = tST;

        } catch (Exception e) {
            Log.d("load error", fname + " load error");
            e.printStackTrace();
        }
        return modelDataArr;
    }
}
```

**步骤七** 在项目中添加 ShaderUtil.java 文件，其中定义 ShaderUtil 类用来完成着色器以及渲染程序创建的相关操作。ShaderUtil 类的代码如下：

```
/**
 * 加载着色器与渲染程序的工具类
 */
public class ShaderUtil {

    /**
     * 加载指定着色器
     * @param shaderType     着色器类型
     *      - GLES20.GL_VERTEX_SHADER    顶点着色器
     *      - GLES20.GL_FRAGMENT_SHADER 片元着色器
     * @param source         着色器代码字符串
     * @return
     *      - 0     失败
     *      - !0    成功，着色器句柄
     */
    public static int loadShader(int shaderType, String source) {
        // 创建一个新 shader
        int shader = GLES20.glCreateShader(shaderType);

        // 若创建成功则加载 shader
        if (shader != 0) {

            // 加载 shader 的源代码
            GLES20.glShaderSource(shader, source);

            // 编译 shader
            GLES20.glCompileShader(shader);
```

```
            // 存放编译成功 shader 数量的数组
            int[] compiled = new int[1];

            // 获取 Shader 的编译情况
            GLES20.glGetShaderiv(shader,
GLES20.GL_COMPILE_STATUS, compiled, 0);

            // 若编译失败则显示错误日志并删除此 shader
            if (compiled[0] == 0) {
                Log.e("ES20_ERROR", "Could not compile shader " +
shaderType + ":");
                Log.e("ES20_ERROR",
GLES20.glGetShaderInfoLog(shader));
                GLES20.glDeleteShader(shader);
                shader = 0;
            }
        }
        return shader;
    }

    /**
     * 创建渲染程序
     * @param vertexSource      顶点着色器代码字符串
     * @param fragmentSource    片元着色器代码字符串
     * @return
     *      - 0     失败
     *      - !0    成功，渲染程序句柄
     */
    public static int createProgram(String vertexSource, String
fragmentSource) {
        // 加载顶点着色器
        int vertexShader = loadShader(GLES20.GL_VERTEX_SHADER,
vertexSource);
        if (vertexShader == 0) {
            return 0;
        }

        // 加载片元着色器
        int pixelShader = loadShader(GLES20.GL_FRAGMENT_SHADER,
fragmentSource);
        if (pixelShader == 0) {
            return 0;
        }

        // 创建渲染程序
        int program = GLES20.glCreateProgram();
```

```
        // 若程序创建成功则向程序中加入顶点着色器与片元着色器
        if (program != 0) {

            // 向程序中加入顶点着色器
            GLES20.glAttachShader(program, vertexShader);
            checkGlError("glAttachShader");

            // 向程序中加入片元着色器
            GLES20.glAttachShader(program, pixelShader);
            checkGlError("glAttachShader");

            // 链接程序
            GLES20.glLinkProgram(program);

            // 存放链接成功program数量的数组
            int[] linkStatus = new int[1];

            // 获取program的链接情况
            GLES20.glGetProgramiv(program,
GLES20.GL_LINK_STATUS, linkStatus, 0);

            // 若链接失败则报错并删除程序
            if (linkStatus[0] != GLES20.GL_TRUE) {
                Log.e("ES20_ERROR", "Could not link program: ");
                Log.e("ES20_ERROR",
GLES20.glGetProgramInfoLog(program));
                GLES20.glDeleteProgram(program);
                program = 0;
            }
        }
        return program;
    }

    /**
     * 检查每一步操作是否有错误的方法
     * @param op    错误描述字符串
     */
    public static void checkGlError(String op) {
        int error;
        while ((error = GLES20.glGetError()) !=
GLES20.GL_NO_ERROR) {
            Log.e("ES20_ERROR", op + ": glError " + error);
            throw new RuntimeException(op + ": glError " + error);
        }
    }

    /**
     * 从脚本中加载着色器代码字符串
```

```
     * @param fname 脚本名称
     * @param r     资源对象
     * @return      着色器代码字符串
     */
    public static String loadFromAssetsFile(String fname,
Resources r) {
        String result = null;

        try {
            InputStream in = r.getAssets().open(fname);
            ByteArrayOutputStream baos = new
ByteArrayOutputStream();

            int ch = 0;
            while ((ch = in.read()) != -1) {
                baos.write(ch);
            }

            byte[] buff = baos.toByteArray();

            baos.close();
            in.close();

            result = new String(buff, "UTF-8");
            result = result.replaceAll("\\r\\n", "\n");

        } catch (Exception e) {
            e.printStackTrace();
        }

        return result;
    }
}
```

**步骤八** 在项目中添加 MySurfaceView.java 文件，其中定义 MySurfaceView 类表示自定义的 SurfaceView 类，在 MySurfaceView 类中定义内部类 MyRenderer 类用来完成实际的渲染工作。整个 MySurfaceView 类的代码如下：

```
/**
 * SurfaceView 类
 */
class MySurfaceView extends GLSurfaceView {

    // 记录手指滑动时 X 坐标
    private float mPreviousX = 0.0f;
```

```
    // 当前SurfaceView的渲染器对象
    private MyRenderer mRenderer;

    public MySurfaceView(Context context) {
        super(context);

        setEGLContextClientVersion(2);

        mRenderer = new MyRenderer();
        setRenderer(mRenderer);
    }

    /**
     * 渲染器类
     */
    class MyRenderer implements GLSurfaceView.Renderer {

        // 拖拽旋转角度
        public float mAngle;

        // 模型顶点数量
        private int mCount;

        private FloatBuffer mVertexBuffer1;
        private FloatBuffer mVertexBuffer2;
        private FloatBuffer mVertexBuffer3;
        private FloatBuffer mTextureBuffer;

        // 渲染程序句柄
        private int mProgram;

        // 顶点着色器中顶点变量句柄
        private int maPositionHandle1;
        private int maPositionHandle2;
        private int maPositionHandle3;

        // 顶点着色器中纹理坐标变量句柄
        private int maTexturePositionHandle;

        // 顶点着色器模型百分比变量句柄
        private int muProportionHandle;

        // 顶点着色器中坐标转换矩阵句柄
        private int muMVPMatrixHandle;

        // 总坐标转换矩阵
        private float[] mMVPMatrix = new float[16];
        // 模型矩阵
```

```
        private float[] mModelMatrix = new float[16];
        // 视图矩阵
        private float[] mViewMatrix = new float[16];
        // 投影矩阵
        private float[] mProjectionMatrix = new float[16];

        // 控制模型运动的相关比昂两
        private int operator = 1;   // 符号 +1 -1
        private float span = 0.15f; // 模型动作增量
        private float curProportion = 0f;   // 模型动作切换比例

        @Override
        public void onSurfaceCreated(GL10 gl10, EGLConfig
eglConfig) {
            GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

            // 打开背面剪裁
            GLES20.glEnable(GLES20.GL_CULL_FACE);

            // 打开深度检测
            GLES20.glEnable(GLES20.GL_DEPTH_TEST);

            // 初始化图形顶点、颜色数据Buffer
            initShapes();

            // 纹理初始化
            initTexture();

            // 着色程序初始化
            initShader();

            // 设置视图矩阵
            Matrix.setLookAtM(mViewMatrix, 0,
                    0f, 0f, 2f,
                    0f, -1.0f, -5.0f,
                    0f, 1.0f, 0.0f);

            new Thread() {
                @Override
                public void run() {
                    while (true) {
                        curProportion = curProportion + operator *
span;
                        if (curProportion > 2.0f) {
                            curProportion = 2.0f;
                            operator = -operator;
                        } else if (curProportion < 0) {
                            curProportion = 0;
```

```
                        operator = -operator;
                    }
                    try {
                        Thread.sleep(30);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
    }

    @Override
    public void onSurfaceChanged(GL10 gl10, int width, int
height) {
        GLES20.glViewport(0, 0, width, height);

        // 获取投影矩阵
        float ratio = (float) width / height;
        Matrix.frustumM(mProjectionMatrix, 0,
                -ratio, ratio, -1.0f, 1.0f, 1.0f, 10.0f);
    }

    @Override
    public void onDrawFrame(GL10 gl10) {
        // 清除颜色缓冲与深度缓冲
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT |
GLES20.GL_DEPTH_BUFFER_BIT);

        // 使用渲染程序
        GLES20.glUseProgram(mProgram);

        //
==========================================================
        // 设置模型矩阵为单位矩阵，类似于初始化工作
        Matrix.setIdentityM(mModelMatrix, 0);
        // 将模型向远离摄像头的方向移动5.0f，像下方移动4.0f
        Matrix.translateM(mModelMatrix, 0, 0.0f, -4.0f,
-5.0f);

        // 以XY 轴中线进行旋转
        Matrix.rotateM(mModelMatrix, 0, mAngle, 0.0f, 1.0f,
0.0f);

        // 求取MV 矩阵（Model * View）
        Matrix.multiplyMM(mMVPMatrix, 0, mViewMatrix, 0,
mModelMatrix, 0);
        // 求取坐标转换总矩阵 MVP 矩阵(Model * View * Project)
```

```
            Matrix.multiplyMM(mMVPMatrix, 0, mProjectionMatrix,
0, mMVPMatrix, 0);

            // 将矩阵传递给着色器的矩阵变量
            GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1,
false, mMVPMatrix, 0);
            //
==========================================================

            // 将变化百分比传入 shader 程序
            GLES20.glUniform1f(muProportionHandle,
curProportion);

            //
==========================================================
            // 传入顶点数据 Buffer
            GLES20.glVertexAttribPointer(maPositionHandle1, 3,
GLES20.GL_FLOAT,
                    false, 0, mVertexBuffer1);
            GLES20.glEnableVertexAttribArray(maPositionHandle1);

            GLES20.glVertexAttribPointer(maPositionHandle2, 3,
GLES20.GL_FLOAT,
                    false, 0, mVertexBuffer2);
            GLES20.glEnableVertexAttribArray(maPositionHandle2);

            GLES20.glVertexAttribPointer(maPositionHandle3, 3,
GLES20.GL_FLOAT,
                    false, 0, mVertexBuffer3);
            GLES20.glEnableVertexAttribArray(maPositionHandle3);

            // 传入纹理坐标数据 Buffer

GLES20.glVertexAttribPointer(maTexturePositionHandle, 2,
GLES20.GL_FLOAT,
                    false, 0, mTextureBuffer);

GLES20.glEnableVertexAttribArray(maTexturePositionHandle);
            //
==========================================================

            // 绘制图形
            GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, mCount);
        }

        /**
         * 根据资源名称加载模型数据
         *
```

```
         * @param strPath 资源名称
         * @return 模型数据
         */
        private float[][] getPointArr(String strPath) {
            float[][] pointArr;
            pointArr = LoadUtil.loadFromFileVertexOnly(strPath,
MySurfaceView.this);
            return pointArr;
        }

        /**
         * 初始化图形数据
         */
        private void initShapes() {

            float[][] positionArr1 =
getPointArr("youtuiqian.obj");
            float[] positionArr2 = getPointArr("zhanli.obj")[0];
            float[] positionArr3 =
getPointArr("zuotuiqian.obj")[0];

            // 获取顶点Buffer
            mVertexBuffer1 =
ByteBuffer.allocateDirect(positionArr1[0].length * 4)
                    .order(ByteOrder.nativeOrder())
                    .asFloatBuffer();
            mVertexBuffer1.put(positionArr1[0]);
            mVertexBuffer1.position(0);

            mVertexBuffer2 =
ByteBuffer.allocateDirect(positionArr2.length * 4)
                    .order(ByteOrder.nativeOrder())
                    .asFloatBuffer();
            mVertexBuffer2.put(positionArr2);
            mVertexBuffer2.position(0);

            mVertexBuffer3 =
ByteBuffer.allocateDirect(positionArr3.length * 4)
                    .order(ByteOrder.nativeOrder())
                    .asFloatBuffer();
            mVertexBuffer3.put(positionArr3);
            mVertexBuffer3.position(0);

            // 获取纹理坐标Buffer
            mTextureBuffer =
ByteBuffer.allocateDirect(positionArr1[1].length * 4)
                    .order(ByteOrder.nativeOrder())
                    .asFloatBuffer();
            mTextureBuffer.put(positionArr1[1]);
```

```
            mTextureBuffer.position(0);

            mCount = positionArr1[0].length / 3;
        }

        /**
         * 纹理初始化
         */
        private void initTexture() {
            // 保存生成纹理句柄的数组
            int[] textures = new int[1];

            // 1、生成一个纹理，纹理句柄放到数组中
            GLES20.glGenTextures(1, textures, 0);

            // 2、对纹理类型进行绑定
            GLES20.glBindTexture(GLES20.GL_TEXTURE_2D,
textures[0]);

            // 3、设置纹理过滤、环绕方式
            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                    GLES20.GL_TEXTURE_MIN_FILTER,
GLES20.GL_NEAREST);
            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                    GLES20.GL_TEXTURE_MAG_FILTER,
GLES20.GL_LINEAR);

            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                    GLES20.GL_TEXTURE_WRAP_S,
GLES20.GL_CLAMP_TO_EDGE);
            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                    GLES20.GL_TEXTURE_WRAP_T,
GLES20.GL_CLAMP_TO_EDGE);

            // 通过输入流加载图片
            InputStream is = MySurfaceView.this
                    .getResources().openRawResource(R.drawable.t
exture);
            Bitmap bitmapTmp;
            try {
                bitmapTmp = BitmapFactory.decodeStream(is);
            } finally {
                try {
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
```

```
            // 4、实际加载纹理
            GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0,
bitmapTmp, 0);

            // 纹理加载成功后释放图片
            bitmapTmp.recycle();
        }

        /**
         * 初始化渲染程序，并获取相应着色器变量句柄
         */
        private void initShader() {
            // 加载顶点着色器的脚本内容
            String vertexSource =
ShaderUtil.loadFromAssetsFile("vertex.sh",
                    MySurfaceView.this.getResources());

            // 加载片元着色器的脚本内容
            String fragmentSource =
ShaderUtil.loadFromAssetsFile("frag.sh",
                    MySurfaceView.this.getResources());

            // 基于顶点着色器与片元着色器创建程序
            mProgram = ShaderUtil.createProgram(vertexSource,
fragmentSource);

            // 获取程序中顶点位置属性引用
            maPositionHandle1 =
GLES20.glGetAttribLocation(mProgram, "aPosition1");
            maPositionHandle2 =
GLES20.glGetAttribLocation(mProgram, "aPosition2");
            maPositionHandle3 =
GLES20.glGetAttribLocation(mProgram, "aPosition3");

            // 获取程序中总变换矩阵引用
            muMVPMatrixHandle =
GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");

            // 获取程序中顶点纹理坐标属性引用
            maTexturePositionHandle =
GLES20.glGetAttribLocation(mProgram, "aTextureCoord");

            // 变化百分比引用
            muProportionHandle =
GLES20.glGetUniformLocation(mProgram, "uProportion");
        }
    }
```

```
    /**
     * 触摸事件处理
     */
    @Override
    public boolean onTouchEvent(MotionEvent e) {
        float x = e.getX();

        switch (e.getAction()) {
            case MotionEvent.ACTION_MOVE: {
                float dx = x - mPreviousX;

                mRenderer.mAngle += (dx) * (180.0f / 320);
                requestRender();

                mPreviousX = x;
                break;
            }
            case MotionEvent.ACTION_DOWN: {
                mPreviousX = x;
                break;
            }
        }

        return true;
    }
}
```

**步骤九** 在 MainActivity 类中完成 MySurfaceView 的创建以及生命周期的绑定。

```
public class MainActivity extends AppCompatActivity {
    private GLSurfaceView mGLView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mGLView = new MySurfaceView(this);
        setContentView(mGLView);
    }

    @Override
    protected void onPause() {
        super.onPause();
        mGLView.onPause();
    }
```

```
    @Override
    protected void onResume() {
        super.onResume();
        mGLView.onResume();
    }
}
```

## 6.4 实验结论

当编码工作完成后在模拟器或真机中运行项目，窗口中会出现一个运动的3D 人物模型，并且能够通过左右滑动对模型进行旋转。效果如下：


*图 6.4.1*