《基于 Andropid 源生 AR 应用开发》

实验手册 05

版 本 1.0

文档提供：智能设备教研室 丁盟

目录

第5章　纹理贴图

5.1 实验目的

目的一： 掌握 OpengGl ES 中纹理贴图的方法。

目的二： 理解纹理与顶点坐标的映射规则。

目的三： 掌握纹理加载方法。

5.2 准备工作

准备一： 在 Android Studio 中根据实验手册 4 创建一个旋转的正方体应用，本次实验是在实验 4 的基础上进行的。

准备二： 根据之前实验手册 4 的顶点顺序，编写纹理坐标数组。

5.3 实验步骤

步骤一 本次实验中需要通过 Context 加载资源图片作为纹理，因此需将 Renderer 放到 SurfaceView 类中，使其作为一个内部类出现，达到使用 Context 加载资源图片的目的。

```
class MySurfaceView extends GLSurfaceView {
    public MySurfaceView(Context context) {
        ......
    }

    // Renderer 作为SurfaceView 的内部类
    class MyRenderer implements GLSurfaceView.Renderer {
        ......
    }
}
```

步骤二 在 Renderer 类中将原来颜色相关成员变量去掉，换成纹理相关的成员变量。

```
// 正方体坐标顶点 Buffer
private FloatBuffer mCubePositions;
// 正方体纹理映射坐标 Buffer
private FloatBuffer mCubeTexturePosition;

// 渲染程序句柄
private int mProgram;

// 顶点着色器中顶点变量句柄
private int mPositionHandle;
// 顶点着色器中纹理坐标变量句柄
private int mTexturePositionHandle;

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

// 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];
```

步骤三 在 Renderer 类中修改着色器代码，使其能够使用纹理坐标和纹理数据。

```
// 顶点着色器代码
private String mVertexShaderCode =
          "uniform mat4 u_MVPMatrix;      \n"
        + "attribute vec4 a_Position;     \n"
        + "attribute vec2 a_TexturePosition;        \n"
        + "varying vec2 v_TexturePosition;          \n"
```

```
        + "void main()                    \n"
        + "{                              \n"
        + "   v_TexturePosition = a_TexturePosition; \n"
        + "   gl_Position = u_MVPMatrix * a_Position;\n"
        + "}                              \n";

// 片元着色器
private String mFragmentShaderCode =
          "precision mediump float;       \n"
        + "varying vec2 v_TexturePosition;        \n"
        + "uniform sampler2D u_Texture;"
        + "void main()                    \n"
        + "{                              \n"
        + "   gl_FragColor = texture2D(u_Texture,
v_TexturePosition);     \n"
        + "}                              \n";
```

步骤四 在 Renderer 中的 initShapes()函数中去掉颜色数组与其 Buffer 转换代码，替换成纹理坐标数组以及纹理坐标转换代码。

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

    // 正方体顶点数据数组
    float cubePosition[] = {
            // 正面 230201
            -1.0f, 1.0f, 1.0f,
            -1.0f, -1.0f, 1.0f,
            1.0f, -1.0f, 1.0f,
            -1.0f, 1.0f, 1.0f,
            1.0f, -1.0f, 1.0f,
            1.0f, 1.0f, 1.0f,
            // 右面 107176
            1.0f, 1.0f, 1.0f,
            1.0f, -1.0f, 1.0f,
            1.0f, -1.0f, -1.0f,
            1.0f, 1.0f, 1.0f,
```

```
            1.0f, -1.0f, -1.0f,
            1.0f, 1.0f, -1.0f,
            // 左面 543532
            -1.0f, 1.0f, -1.0f,
            -1.0f, -1.0f, -1.0f,
            -1.0f, -1.0f, 1.0f,
            -1.0f, 1.0f, -1.0f,
            -1.0f, -1.0f, 1.0f,
            -1.0f, 1.0f, 1.0f,
            // 顶面 521516
            -1.0f, 1.0f, -1.0f,
            -1.0f, 1.0f, 1.0f,
            1.0f, 1.0f, 1.0f,
            -1.0f, 1.0f, -1.0f,
            1.0f, 1.0f, 1.0f,
            1.0f, 1.0f, -1.0f,
            // 底面 347370
            -1.0f, -1.0f, 1.0f,
            -1.0f, -1.0f, -1.0f,
            1.0f, -1.0f, -1.0f,
            -1.0f, -1.0f, 1.0f,
            1.0f, -1.0f, -1.0f,
            1.0f, -1.0f, 1.0f,
            // 背面 674645
            1.0f, 1.0f, -1.0f,
            1.0f, -1.0f, -1.0f,
            -1.0f, -1.0f, -1.0f,
            1.0f, 1.0f, -1.0f,
            -1.0f, -1.0f, -1.0f,
            -1.0f, 1.0f, -1.0f
    };

    // 正方体纹理坐标数据数组
    float[] texturePosition = {
            // 正面 230201
            0.0f, 0.0f,
            0.0f, 1.0f,
            1.0f, 1.0f,
            0.0f, 0.0f,
            1.0f, 1.0f,
            1.0f, 0.0f,
```

```
            // 右面 107176
            0.0f, 0.0f,
            0.0f, 1.0f,
            1.0f, 1.0f,
            0.0f, 0.0f,
            1.0f, 1.0f,
            1.0f, 0.0f,
            // 左面 543532
            0.0f, 0.0f,
            0.0f, 1.0f,
            1.0f, 1.0f,
            0.0f, 0.0f,
            1.0f, 1.0f,
            1.0f, 0.0f,
            // 顶面 521516
            0.0f, 0.0f,
            0.0f, 1.0f,
            1.0f, 1.0f,
            0.0f, 0.0f,
            1.0f, 1.0f,
            1.0f, 0.0f,
            // 底面 347370
            0.0f, 0.0f,
            0.0f, 1.0f,
            1.0f, 1.0f,
            0.0f, 0.0f,
            1.0f, 1.0f,
            1.0f, 0.0f,
            // 背面 674645
            0.0f, 0.0f,
            0.0f, 1.0f,
            1.0f, 1.0f,
            0.0f, 0.0f,
            1.0f, 1.0f,
            1.0f, 0.0f
    };

    // 获取顶点 Buffer
    mCubePositions =
ByteBuffer.allocateDirect(cubePosition.length*4)
            .order(ByteOrder.nativeOrder())
```

```
            .asFloatBuffer();
    mCubePositions.put(cubePosition);
    mCubePositions.position(0);

    // 获取纹理坐标Buffer
    mCubeTexturePosition =
ByteBuffer.allocateDirect(texturePosition.length*4)
            .order(ByteOrder.nativeOrder())
            .asFloatBuffer();
    mCubeTexturePosition.put(texturePosition);
    mCubeTexturePosition.position(0);
}
```

步骤五 在 Surface 中添加成员变量 mTextureId 用来保存纹理句柄。

```
// 系统分配的纹理句柄
int mTextureId;
```

步骤六 在 Surface 中添加 initTexture()方法，完成纹理的初始化工作。

```
// 纹理初始化
public 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);
    // 获取生成的纹理句柄
    mTextureId = textures[0];

    // 2、对纹理类型进行绑定
    GLES20.glBindTexture(GLES20.GL_TEXTURE_2D,
                         mTextureId);
```

```
    // 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 = this.getResources()
        .openRawResource(R.drawable.test);
    Bitmap bitmapTmp;
    try {
        bitmapTmp = BitmapFactory.decodeStream(is);
    } finally {
        try {
            is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // 4、实际加载纹理
    // 方法一
    // GLUtils.texImage2D (GLES20.GL_TEXTURE_2D, 0,
    //                     bitmapTmp, 0 );

    // 方法二
    ByteBuffer buf = ByteBuffer
        .allocate(bitmapTmp.getByteCount());
    buf.order(ByteOrder.nativeOrder());
    bitmapTmp.copyPixelsToBuffer(buf);
    buf.position(0);
```

```
    GLES20.glTexImage2D(GLES20.GL_TEXTURE_2D, 0,
            GLES20.GL_RGBA, 256, 256, 0,
            GLES20.GL_RGBA,
            GLES20.GL_UNSIGNED_BYTE, buf);

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

步骤七 Renderer 中的 onSurfaceCreated()方法中，去掉颜色的相关操作，添加纹理的操作。

```
@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();

    // 加载顶点着色器
    int vertexShader
        = loadShader(GLES20.GL_VERTEX_SHADER,
                     mVertexShaderCode);
    // 加载片元着色器
    int fragmentShader
        = loadShader(GLES20.GL_FRAGMENT_SHADER,
                     mFragmentShaderCode);
```

```
    // 创建渲染程序
    mProgram = GLES20.glCreateProgram();
    // 将着色器添加到渲染程序中
    GLES20.glAttachShader(mProgram, vertexShader);
    GLES20.glAttachShader(mProgram, fragmentShader);
    // 链接渲染程序
    GLES20.glLinkProgram(mProgram);

    // 分别获得着色器相应变量的句柄
    mPositionHandle = GLES20
        .glGetAttribLocation(mProgram, "a_Position");
    mTexturePositionHandle = GLES20
        .glGetAttribLocation(mProgram,
                             "a_TexturePosition");
    mMVPMatrixHandle = GLES20
        .glGetUniformLocation(mProgram, "u_MVPMatrix");

    // 设置视图矩阵
    Matrix.setLookAtM(mViewMatrix, 0,
            0f, 0f, -0.5f,
            0f, 0f, -5.0f,
            0f, 1.0f, 0.0f);
}
```

步骤八 Renderer 中的 onDrawFrame()方法中，将纹理坐标 Buffer 传递给着色器，完成图形的绘制。

```
@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);

        // 传入正方体顶点数据 Buffer
        GLES20.glVertexAttribPointer(mPositionHandle,
```

```
                3, GLES20.GL_FLOAT,
                false, 0, mCubePositions);

GLES20.glEnableVertexAttribArray(mPositionHandle);

        // 传入正方体纹理坐标数据 Buffer

GLES20.glVertexAttribPointer(mTexturePositionHandle,
                2, GLES20.GL_FLOAT,
                false, 0, mCubeTexturePosition);

GLES20.glEnableVertexAttribArray(mTexturePositionHandle);

        // 求取正方体旋转的角度，根据时间确定，10 秒旋转 360 度
        long time = System.currentTimeMillis() % 10000L;
        float angleInDegrees = (360.0f / 10000.0f) * ((int) time);

        // 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
        Matrix.translateM(mModelMatrix, 0, 0.0f, 0.0f, -5.0f);
        // 以 XY 轴中线进行旋转
        Matrix.rotateM(mModelMatrix, 0, angleInDegrees, 1.0f, 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(mMVPMatrixHandle, 1, false, mMVPMatrix, 0);
```

```
        // 绘制图形
        GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0,
36);
    }

}
```

5.4 实验结论

当编码工作完成后在模拟器或真机中运行项目，效果如下：

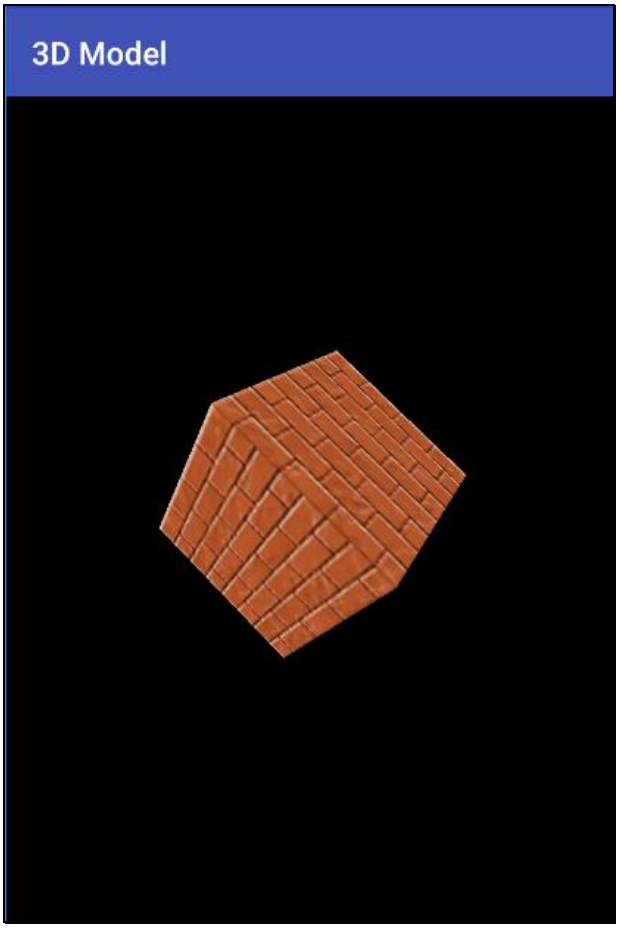

图 5.4.1