《基于 Andropid 源生 AR 应用开发》

实验手册 08

版 本 1.0

文档提供：智能设备教研室 丁盟

目录

第8章　Vuforia AR 程序

8.1 实验目的

目的一： 在 Vuforia Android 官方示例程序的基础上开发自己的 AR 应用程序。

8.2 准备工作

准备一： 创建目标数据集

准备一张图片作为图像目标，AR 应用会对本图像目标进行追踪,例如下图。

图 8.2.1

在 Vuforia 开发者网站中一次点击“Develop”、“Target Manager”、“Add Database”，创建一个目标数据库，用来管理目标文件。

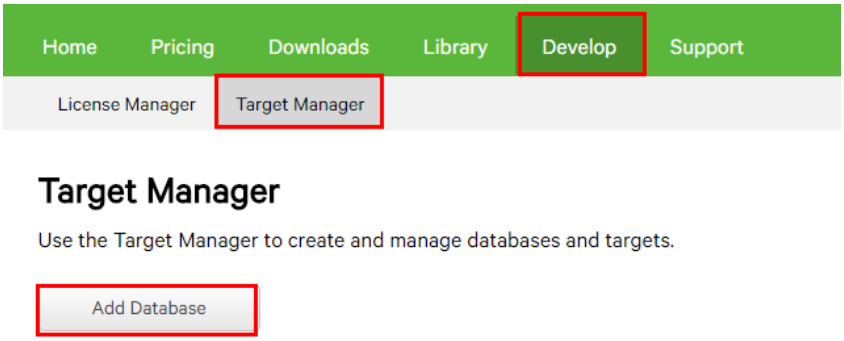

图 8.2.2

进入创建的目标数据库，点击“Add Target”添加目标文件。

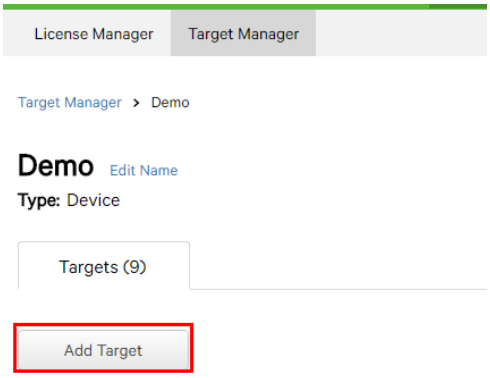

图 8.2.3

在目标添加界面中选择“Single Image”，然后依次输入本地文件路径、图像目标宽度、图像目标名称，最后点击“Add”完成图片目标的添加。

图 8.2.4

添加完目标图片之后，可以看到 Vuforia 对于上传的图像目标识别的的打分，5 星为最高。

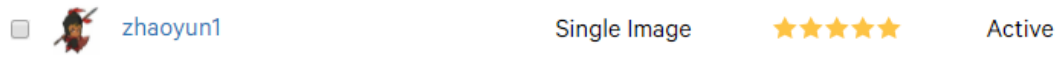

图 8.2.5

此时图像目标已经添加到 Vuforia 的开发者网站的目标数据库中了，接下来需要经过目标数据度对图像目标的转换，得到最终 Vuforia SDK 中目标追踪器能够识别的目标文件，选中需要下载的目标文件，然后点击“Download Database”下载选中的目标文件。

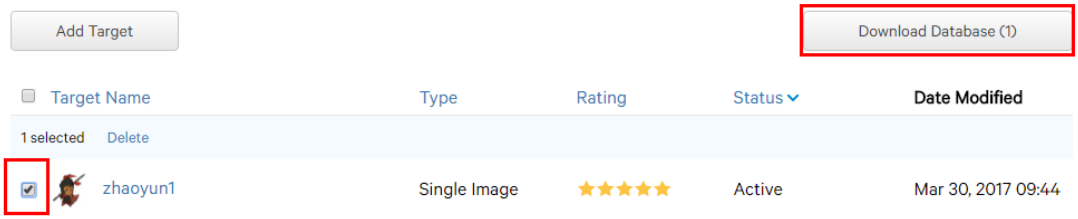

图 8.2.6

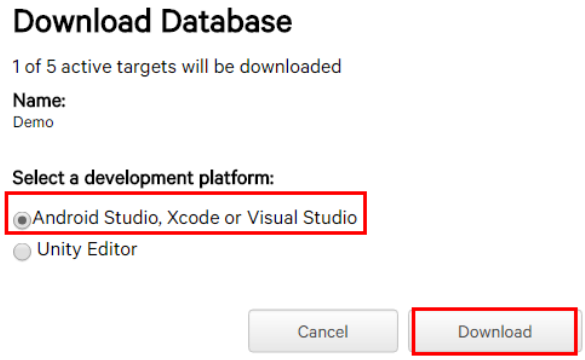

图 8.2.7

下载完成后会得到一个压缩包，将其进行解压后会得到一个.dat 文件与一个.xml 文件，两个文件文件名是相同的，我们可也根据自己的需求对这两个文件进行重命名，默认名是 Vuforia 开发者网站的数据库名称，注意：一定要保证两个文件文件名一致。其中.dat 文件为真实的目标文件，.xml 为目标文件解释文件，Vuforia SDK 通过.xml 文件加载同名.dat 文件。

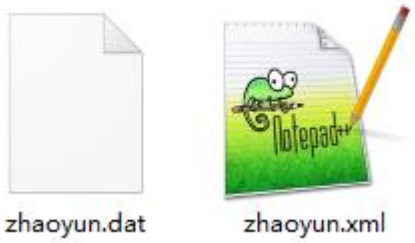

图 8.2.8

准备二： 准备 3D 模型文件

准备.obj 格式的 3D 模型文件，以及其对应的纹理文件。

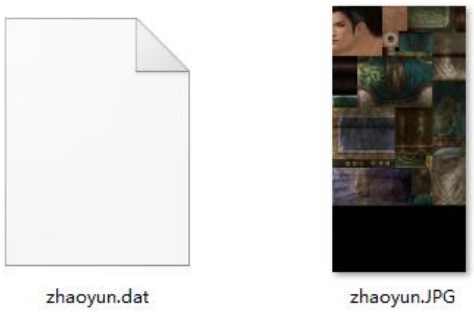

图 8.2.9

准备三： 申请秘钥

通过实验 7 的方法在 Vuforia 开发者网站上申请一个秘钥。

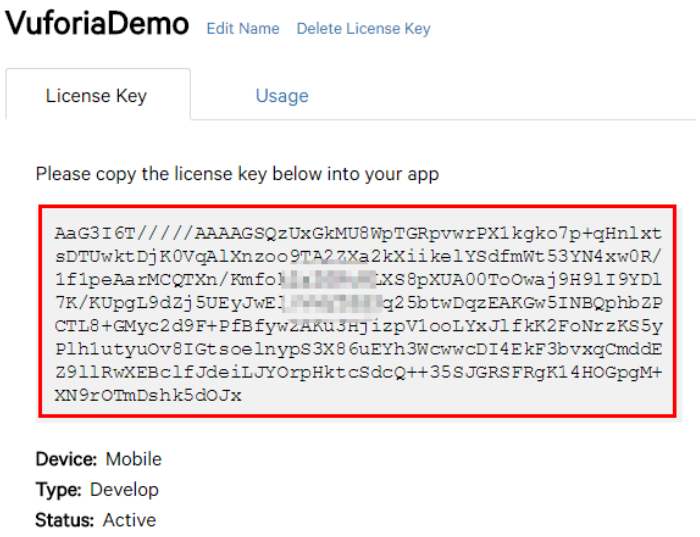

图 8.2.10

8.3 实验步骤

步骤一 使用 Android Studio 创建一个 Android 空项目，并在 AndroidManifest.xml 中添加 Vuforia SDK 所需的权限及 OpenGL ES 版本信息。

```
<!--支持的最小 API SDK 版本-->
<uses-sdk android:minSdkVersion="14" />

<!--OpenGL ES 2.0-->
<uses-feature android:glEsVersion="0x00020000"
android:required="true" />
<!--摄像头-->
<uses-feature android:name="android.hardware.camera" />

<!--使用摄像头-->
<uses-permission android:name="android.permission.CAMERA" />
<!--创建网络连接-->
<uses-permission android:name="android.permission.INTERNET"/>
<!--使用 GSM 网络，3G、4G/WiFi-->
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
<!--读取设备外部存储空间-->
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>
```

步骤二 在 app 的 build.gradle 文件的 android 节点下配置 jniLibs 目录信息，将 libs 目录设置为默认库目录。

```
// jniLibs 目录
sourceSets {
    main {
        jniLibs.srcDirs = ['libs']
    }
}
```

步骤三 接着从之前下载好的 Vuforia for Android SDK 压缩包中将 jar 包与库

文件分别拷贝到 jniLibs 目录下（即 libs 目录下），鼠标右键 Vuforia.jar 文件，选择“add as library”将其添加到项目中，如图所示。

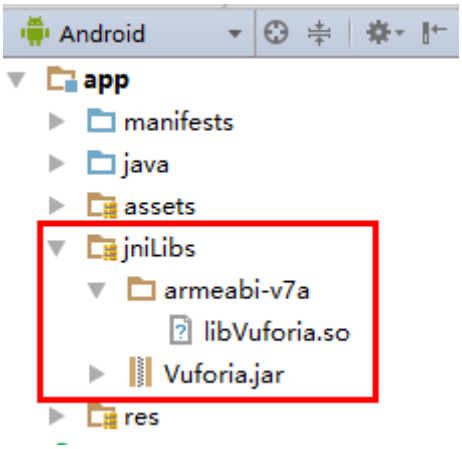

图 8.3.1

步骤四 在项目目录下的“app\src\main\”路径下创建“assets”文件夹，用来存放程序运行时加载的资源文件，将目标数据集文件、模型文件、纹理文件全部拷贝到“assets”文件夹中。

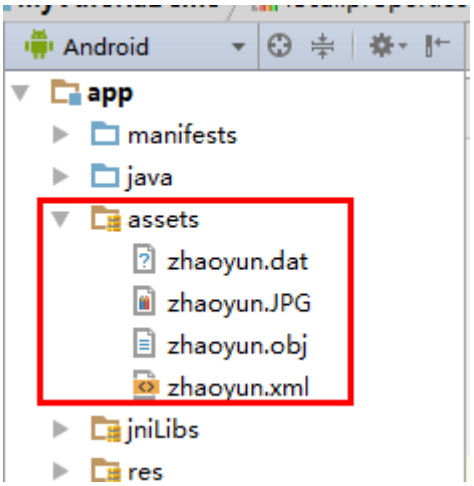

图 8.3.2

步骤五 打开“res\values\strings.xml”文件，添加程序中用到的字符串信息，主要为 Vuforia SDK 错误信息字符串。

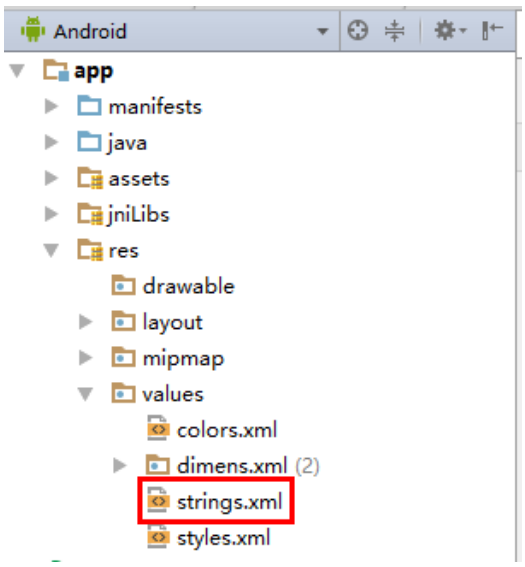

图 8.3.3

```
<string name="INIT_ERROR">Error</string>
<string name="INIT_ERROR_DEVICE_NOT_SUPPORTED">Failed to
initialize Vuforia. This device is not supported</string>
<string name="INIT_ERROR_NO_CAMERA_ACCESS">Failed to initialize
Vuforia. Camera not accessible</string>
<string name="INIT_LICENSE_ERROR_MISSING_KEY">Vuforia App key is
missing. Please get a valid key, by logging into your account at
developer.vuforia.com and creating a new project.</string>
<string name="INIT_LICENSE_ERROR_INVALID_KEY">Invalid Key used.
Please make sure you are using a valid Vuforia App Key.</string>
<string name="INIT_LICENSE_ERROR_NO_NETWORK_TRANSIENT">Unable to
contact server. Please try again later.</string>
<string name="INIT_LICENSE_ERROR_NO_NETWORK_PERMANENT">No
network available. Please make sure you are connected to the
Internet.</string>
<string name="INIT_LICENSE_ERROR_CANCELED_KEY">This app license
key has been canceled and may no longer be used. Please get a new
license key.</string>
<string name="INIT_LICENSE_ERROR_PRODUCT_TYPE_MISMATCH">Vuforia
App key is not valid for this product. Please get a valid key, by
logging into your account at developer.vuforia.com and choosing
the right product type during project creation.</string>
<string name="INIT_LICENSE_ERROR_UNKNOWN_ERROR">Failed to
initialize Vuforia.</string>
<string name="button_OK">OK</string>
```

步骤六 在“res\layout\”中添加一个加载动画页面布局“loading.xml”，文件代码如下。

图 8.3.4

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/camera_overlay_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

        <ProgressBar
            style="@android:style/Widget.ProgressBar"
            android:id="@+id/loading_indicator"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true" />

</RelativeLayout>
```

步骤七 在项目中添加 ModelObject.java 文件，主要完成 3D 模型加载方法封装。

```
package com.hebeiedu.myvuforiademo;

import android.content.res.AssetManager;
import android.util.Log;
```

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.util.ArrayList;

/**
 * 模型数据对象
 */
public class ModelObject {

    private String mFilePath;
    private AssetManager mAssetManager;

    // 最大S纹理坐标
    private static final float MAX_S = 1.0f;
    // 最大T纹理坐标
    private static final float MAX_T = 1.0f;
    // 顶点Buffer
    private FloatBuffer mVertBuff;
    // 纹理坐标Buffer
    private FloatBuffer mTexCoordBuff;
    // 顶点数量
    private int mVerticesNumber = 0;

    /**
     * 构造器
     *
     * @param filePath     模型文件名称
     * @param assetManager 资源管理器
     */
    public ModelObject(String filePath,
                       AssetManager assetManager) {
        mFilePath = filePath;
        mAssetManager = assetManager;

        // 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 = assetManager.open(filePath);
            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(1 - 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);
            }

            // 将数组转换成Buffer
            mVertBuff = ByteBuffer.allocateDirect(vXYZ.length * 4)
                    .order(ByteOrder.nativeOrder())
                    .asFloatBuffer();
            mVertBuff.put(vXYZ);
            mVertBuff.position(0);

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

            mVerticesNumber = vXYZ.length / 3;

        } catch (Exception e) {
            Log.d(this.getClass().getName(),
                    "Failed to load " + filePath + " file." +
e.toString());
        }

    }

    /**
     * 获取顶点坐标Buffer
     * @return  坐标Buffer
     */
    public FloatBuffer getVertices() {
        return mVertBuff;
    }
```

```
    /**
     * 获取纹理映射坐标Buffer
     * @return  坐标Buffer
     */
    public FloatBuffer getTexCoords() {
        return mTexCoordBuff;
    }

    /**
     * 获取顶点数量
     * @return  顶点数量
     */
    public int getNumObjectVertex() {
        return mVerticesNumber;
    }

}
```

步骤八 在项目中添加 LoadingDialogHandler.java 文件，实现加载动画窗口句柄功能。

```
package com.hebeiedu.myvuforiademo;

import android.app.Activity;
import android.os.Handler;
import android.os.Message;
import android.view.View;

import java.lang.ref.WeakReference;

public class LoadingDialogHandler extends Handler {
    private final WeakReference<Activity> mActivity;
    // Constants for Hiding/Showing Loading dialog
    public static final int HIDE_LOADING_DIALOG = 0;
    public static final int SHOW_LOADING_DIALOG = 1;

    public View mLoadingDialogContainer;

    public LoadingDialogHandler(Activity activity)
    {
        mActivity = new WeakReference<Activity>(activity);
    }
```

```
    public void handleMessage(Message msg)
    {
        Activity imageTargets = mActivity.get();
        if (imageTargets == null)
        {
            return;
        }

        if (msg.what == SHOW_LOADING_DIALOG)
        {
            mLoadingDialogContainer.setVisibility(View.VISIBLE);

        } else if (msg.what == HIDE_LOADING_DIALOG)
        {
            mLoadingDialogContainer.setVisibility(View.GONE);
        }
    }

}
```

步骤九 在项目中添加 VideoBackgroundShader.java 文件，主要保存视频背景着色器代码。

```
package com.hebeiedu.myvuforiademo;

/**
 * 视频背景着色器 src 代码
 */
public class VideoBackgroundShader
{
    /**
     * 顶点着色器程序 src 代码
     */
    public static final String VB_VERTEX_SHADER =
        "attribute vec4 vertexPosition;\n" +
        "attribute vec2 vertexTexCoord;\n" +
        "uniform mat4 projectionMatrix;\n" +

        "varying vec2 texCoord;\n" +

        "void main()\n" +
        "{\n" +
        "    gl_Position = projectionMatrix * vertexPosition;\n" +
        "    texCoord = vertexTexCoord;\n" +
        "}\n";
```

```
    /**
     * 片段着色器程序 src 代码
     */
    public static final String VB_FRAGMENT_SHADER =
        "precision mediump float;\n" +
        "varying vec2 texCoord;\n" +
        "uniform sampler2D texSampler2D;\n" +
        "void main ()\n" +
        "{\n" +
        "    gl_FragColor = texture2D(texSampler2D, texCoord);\n"
+
        "}\n";
}
```

步骤十 在项目中添加 Texture.java 文件，完成纹理的加载方法封装。

```
package com.hebeiedu.myvuforiademo;

import android.content.res.AssetManager;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.util.Log;

import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;

/**
 * 纹理类，从 APK 中加载纹理资源
 */
public class Texture {
    // 纹理宽度
    public int mWidth;
    // 纹理高度
    public int mHeight;
    // 通道数量
    public int mChannels;
    // 纹理数据 Buffer
    public ByteBuffer mData;
    // 纹理 ID
    public int[] mTextureID = new int[1];
    // 是否加载成功
    public boolean mSuccess = false;
```

```
    /**
     * 从 APK 中加载资源文件
     * @param fileName  纹理名称
     * @param assets    资源管理器
     * @return  纹理对象
     */
    public static Texture loadTextureFromApk(String fileName,
                                             AssetManager assets) {
        InputStream inputStream = null;

        try {

            inputStream = assets.open(fileName,
AssetManager.ACCESS_BUFFER);

            BufferedInputStream bufferedStream = new
BufferedInputStream(
                    inputStream);
            Bitmap bitMap =
BitmapFactory.decodeStream(bufferedStream);

            int[] data = new int[bitMap.getWidth() *
bitMap.getHeight()];
            bitMap.getPixels(data, 0, bitMap.getWidth(), 0, 0,
                    bitMap.getWidth(), bitMap.getHeight());

            return loadTextureFromIntBuffer(data,
bitMap.getWidth(),
                    bitMap.getHeight());

        } catch (IOException e) {

            Log.e(Thread.currentThread().getStackTrace()[1]
                            .getClassName().getClass().getName(),
                    "Failed to log texture '" + fileName + "' from
APK");
            Log.i(Thread.currentThread().getStackTrace()[1]
                    .getClassName().getClass().getName(),
e.getMessage());
            return null;
        }
    }


    /**
     * 从 Buffer 中获取纹理
     * @param data      Buffer
```

```
     * @param width      纹理宽度
     * @param height     纹理高度
     * @return           Texture 纹理
     */
    public static Texture loadTextureFromIntBuffer(int[] data, int
width,
                                                   int height) {
        int numPixels = width * height;
        byte[] dataBytes = new byte[numPixels * 4];

        for (int p = 0; p < numPixels; ++p) {
            int colour = data[p];
            dataBytes[p * 4] = (byte) (colour >>> 16);      // R
            dataBytes[p * 4 + 1] = (byte) (colour >>> 8);   // G
            dataBytes[p * 4 + 2] = (byte) colour;           // B
            dataBytes[p * 4 + 3] = (byte) (colour >>> 24);  // A
        }

        Texture texture = new Texture();
        texture.mWidth = width;
        texture.mHeight = height;
        texture.mChannels = 4;

        // 使用当前设备的存储方式
        texture.mData =
ByteBuffer.allocateDirect(dataBytes.length).order(
                ByteOrder.nativeOrder());
        int rowSize = texture.mWidth * texture.mChannels;
        for (int r = 0; r < texture.mHeight; r++)
            texture.mData.put(dataBytes, rowSize *
(texture.mHeight - 1 - r),
                    rowSize);
        // 将 Buffer 索引值指向最前
        texture.mData.rewind();

        // 清除相应变量
        dataBytes = null;
        data = null;

        // 记录纹理加载成功
        texture.mSuccess = true;

        return texture;
    }
}
```

步骤十一 在项目中添加 SampleUtils.java 文件，主要功能为着色器矩阵运算工具类。

```
package com.hebeiedu.myvuforiademo;

import android.opengl.GLES20;
import android.util.Log;

/**
 * 着色器及矩阵运算相关工具类
 */
public class SampleUtils
{
    private static final String LOGTAG = "SampleUtils";

    /**
     * 初始化着色程序，着色器初始化主要分为5个步骤进行，详见代码
     * @param shaderType    着色程序类别 GL_VERTEX_SHADER or
GL_FRAGMENT_SHADER
     * @param source        着色程序代码src
     * @return  着色器句柄
     */
    static int initShader(int shaderType, String source)
    {
        // [1] 创建一个新的着色程序，获取着色器句柄
        int shader = GLES20.glCreateShader(shaderType);

        // 非0表示着色器创建成功
        if (shader != 0)
        {
            // [2] 提供着色器源码
            GLES20.glShaderSource(shader, source);
            // [3] 变异着色器程序
            GLES20.glCompileShader(shader);

            // 保存便以结果的数组
            int[] glStatusVar = { GLES20.GL_FALSE };

            // [4] 查询编译结果
            GLES20.glGetShaderiv(shader,
GLES20.GL_COMPILE_STATUS,
                    glStatusVar, 0);

            // 编译失败
            if (glStatusVar[0] == GLES20.GL_FALSE)
            {
```

```
                // [5] 打印Info日志
                Log.i(LOGTAG, "Could NOT compile shader " +
shaderType + " : "
                    + GLES20.glGetShaderInfoLog(shader));
                // [6] 删除着色器
                GLES20.glDeleteShader(shader);
                shader = 0;
            }
        }

        // 返回着色器句柄
        return shader;
    }

    /**
     * 创建着色器程序对象
     * @param vertexShaderSrc       顶点着色器程序src
     * @param fragmentShaderSrc     片段着色器程序src
     * @return  0 失败; !0 成功
     */
    public static int createProgramFromShaderSrc(String
vertexShaderSrc,
        String fragmentShaderSrc)
    {
        // 创建顶点着色器
        int vertShader = initShader(GLES20.GL_VERTEX_SHADER,
vertexShaderSrc);
        // 创建片段着色器
        int fragShader = initShader(GLES20.GL_FRAGMENT_SHADER,
fragmentShaderSrc);

        // 着色器创建失败返回0，是否需要释放创建成功的着色器?
        if (vertShader == 0 || fragShader == 0)
            return 0;

        // 创建一个着色程序对象
        int program = GLES20.glCreateProgram();

        if (program != 0)
        {
            // 将顶点着色器附加到着色程序对象上
            GLES20.glAttachShader(program, vertShader);
            checkGLError("glAttchShader(vert)");

            // 将片段着色器附加到着色程序对象上
            GLES20.glAttachShader(program, fragShader);
            checkGLError("glAttchShader(frag)");
```

```
            // 对着色程序对象进行链接操作，即生成最终的着色程序
            GLES20.glLinkProgram(program);

            int[] glStatusVar = { GLES20.GL_FALSE };
            // 查询链接是否成功
            GLES20.glGetProgramiv(program,
GLES20.GL_LINK_STATUS,
                    glStatusVar, 0);

            // 如果链接失败
            if (glStatusVar[0] == GLES20.GL_FALSE)
            {
                Log.i(LOGTAG, "Could NOT link program : "
                        + GLES20.glGetProgramInfoLog(program));
                // 删除着色程序对象
                GLES20.glDeleteProgram(program);
                program = 0;
            }
        }

        return program;
    }

    /**
     * 检查错误，错误来源为OpenGL ES 2.0
     * @param op    错误相关字符串
     */
    public static void checkGLError(String op)
    {
        for (int error = GLES20.glGetError();
             error != 0;
             error = GLES20.glGetError())
        {
            Log.i(LOGTAG, "After operation " + op + " got glError
0x"
                    + Integer.toHexString(error));
        }
    }

    /**
     * 屏幕坐标到相机坐标，相机图像会进行剪裁以适应不同的横纵比屏幕
     * 图形算法待深入研究
     * @param screenX
     * @param screenY
     * @param screenDX
     * @param screenDY
     * @param screenWidth
     * @param screenHeight
```

```
     * @param cameraWidth
     * @param cameraHeight
     * @param cameraX
     * @param cameraY
     * @param cameraDX
     * @param cameraDY
     * @param displayRotation
     * @param cameraRotation
     */
    public static void screenCoordToCameraCoord(int screenX, int
screenY,
        int screenDX, int screenDY, int screenWidth, int
screenHeight,
        int cameraWidth, int cameraHeight, int[] cameraX, int[]
cameraY,
        int[] cameraDX, int[] cameraDY, int displayRotation, int
cameraRotation)
    {
        float videoWidth, videoHeight;
        videoWidth = (float) cameraWidth;
        videoHeight = (float) cameraHeight;

        // Compute the angle by which the camera image should be
rotated clockwise so that it is
        // shown correctly on the display given its current
orientation.
        // 计算相机图像顺时针旋转的角度，使其在当前方向上显示正确。
        int correctedRotation =
((((displayRotation*90)-cameraRotation)+360)%360)/90;

        switch (correctedRotation)
        {
            case 0:
                break;

            case 1:
                int tmp = screenX;
                screenX = screenHeight - screenY;
                screenY = tmp;

                tmp = screenDX;
                screenDX = screenDY;
                screenDY = tmp;

                tmp = screenWidth;
                screenWidth = screenHeight;
                screenHeight = tmp;

                break;
```

```
            case 2:
                screenX = screenWidth - screenX;
                screenY = screenHeight - screenY;
                break;

            case 3:
                tmp = screenX;
                screenX = screenY;
                screenY = screenWidth - tmp;

                tmp = screenDX;
                screenDX = screenDY;
                screenDY = tmp;

                tmp = screenWidth;
                screenWidth = screenHeight;
                screenHeight = tmp;

                break;
        }

        float videoAspectRatio = videoHeight / videoWidth;
        float screenAspectRatio = (float) screenHeight / (float)
screenWidth;

        float scaledUpX;
        float scaledUpY;
        float scaledUpVideoWidth;
        float scaledUpVideoHeight;

        if (videoAspectRatio < screenAspectRatio)
        {
            // 视频高度将适合屏幕高度
            scaledUpVideoWidth = (float) screenHeight /
videoAspectRatio;
            scaledUpVideoHeight = screenHeight;
            scaledUpX = (float) screenX
                + ((scaledUpVideoWidth - (float) screenWidth) /
2.0f);
            scaledUpY = (float) screenY;
        }
        else
        {
            // 视频宽度将适合屏幕宽度
            scaledUpVideoHeight = (float) screenWidth *
videoAspectRatio;
            scaledUpVideoWidth = screenWidth;
            scaledUpY = (float) screenY
```

```
                + ((scaledUpVideoHeight - (float) screenHeight) /
2.0f);
            scaledUpX = (float) screenX;
        }

        if (cameraX != null && cameraX.length > 0)
        {
            cameraX[0] = (int) ((scaledUpX / (float)
scaledUpVideoWidth) * videoWidth);
        }

        if (cameraY != null && cameraY.length > 0)
        {
            cameraY[0] = (int) ((scaledUpY / (float)
scaledUpVideoHeight) * videoHeight);
        }

        if (cameraDX != null && cameraDX.length > 0)
        {
            cameraDX[0] = (int) (((float) screenDX / (float)
scaledUpVideoWidth) * videoWidth);
        }

        if (cameraDY != null && cameraDY.length > 0)
        {
            cameraDY[0] = (int) (((float) screenDY / (float)
scaledUpVideoHeight) * videoHeight);
        }
    }

    /**
     * 获取正交矩阵，投影矩阵
     * 图形算法待深入研究
     * @param nLeft
     * @param nRight
     * @param nBottom
     * @param nTop
     * @param nNear
     * @param nFar
     * @return
     */
    public static float[] getOrthoMatrix(float nLeft, float
nRight,
        float nBottom, float nTop, float nNear, float nFar)
    {
        float[] nProjMatrix = new float[16];

        int i;
        for (i = 0; i < 16; i++)
```

```
            nProjMatrix[i] = 0.0f;

        nProjMatrix[0] = 2.0f / (nRight - nLeft);
        nProjMatrix[5] = 2.0f / (nTop - nBottom);
        nProjMatrix[10] = 2.0f / (nNear - nFar);
        nProjMatrix[12] = -(nRight + nLeft) / (nRight - nLeft);
        nProjMatrix[13] = -(nTop + nBottom) / (nTop - nBottom);
        nProjMatrix[14] = (nFar + nNear) / (nFar - nNear);
        nProjMatrix[15] = 1.0f;

        return nProjMatrix;
    }
}
```

步骤十二　在项目中添加 SampleAppRendererControl.java 文件，为渲染器控制器接口类。

```
package com.hebeiedu.myvuforiademo;

import com.vuforia.State;

/**
 * 渲染器控制器接口类
 */
public interface SampleAppRendererControl
{

    /**
     * 需要被每一个渲染器实例实现，SampleAppRendering 每个视图的循环中调用
     * @param state
     * @param projectionMatrix
     */
    void renderFrame(State state, float[] projectionMatrix);
}
```

步骤十三　在项目中添加 SampleAppRenderer.java 文件，其中封装了渲染图元的一些操作。

```
package com.hebeiedu.myvuforiademo;
```

```java
import android.app.Activity;
import android.content.res.Configuration;
import android.graphics.Point;
import android.opengl.GLES20;
import android.opengl.Matrix;
import android.util.Log;

import com.vuforia.COORDINATE_SYSTEM_TYPE;
import com.vuforia.CameraDevice;
import com.vuforia.Device;
import com.vuforia.GLTextureUnit;
import com.vuforia.Matrix34F;
import com.vuforia.Mesh;
import com.vuforia.Renderer;
import com.vuforia.RenderingPrimitives;
import com.vuforia.State;
import com.vuforia.Tool;
import com.vuforia.TrackerManager;
import com.vuforia.VIDEO_BACKGROUND_REFLECTION;
import com.vuforia.VIEW;
import com.vuforia.Vec2F;
import com.vuforia.Vec2I;
import com.vuforia.Vec4I;
import com.vuforia.VideoBackgroundConfig;
import com.vuforia.VideoMode;
import com.vuforia.ViewList;

/**
 * 封装了渲染图元的一些操作方法
 */
public class SampleAppRenderer {
    // Vuforia 绘制图元对象
    private RenderingPrimitives mRenderingPrimitives = null;
    // 渲染器控制器对象
    private SampleAppRendererControl mRenderingInterface = null;
    // 当前Acticity
    private Activity mActivity = null;

    // OpenGL ES 渲染器对象
    private Renderer mRenderer = null;
    // 当前视图模式
    private int currentView = VIEW.VIEW_SINGULAR;

    // 近平面
    private float mNearPlane = -1.0f;
    // 远平面
    private float mFarPlane = -1.0f;
```

```
    // Vuforia GL 纹理单元
    private GLTextureUnit videoBackgroundTex = null;

    // AR 模式渲染视频背景
    private int vbShaderProgramID = 0;      // 着色程序对象句柄
    private int vbTexSampler2DHandle = 0;   // 使用哪一个采样器进
行纹理采样变量句柄
    private int vbVertexHandle = 0;         // 顶点位置变量句柄
    private int vbTexCoordHandle = 0;       // 顶点纹理坐标变量句
柄
    private int vbProjectionMatrixHandle = 0;   // 投影矩阵变量句
柄

    // 设备显示大小
    private int mScreenWidth = 0;
    private int mScreenHeight = 0;

    // 是否竖屏
    private boolean mIsPortrait = false;

    // 虚拟摄像头 Y 轴视角度数
    static final float VIRTUAL_FOV_Y_DEGS = 85.0f;
    // π 值
    static final float M_PI = 3.14159f;

    /**
     * 构造器
     * @param renderingInterface    渲染器控制器
     * @param activity              Activity
     * @param deviceMode            设备模式
     * @param stereo                当前查看器是否为活动状态
     * @param nearPlane             近平面
     * @param farPlane              远平面
     */
    public SampleAppRenderer(SampleAppRendererControl
renderingInterface,
                             Activity activity,
                             int deviceMode,
                             boolean stereo,
                             float nearPlane,
                             float farPlane)
    {
        mActivity = activity;
        mRenderingInterface = renderingInterface;

        // Vuforia Renderer 为单例模式，返回渲染器
        mRenderer = Renderer.getInstance();
```

```
        // 检查近平面与远平面参数合法性
        if(farPlane < nearPlane)
        {
            Log.i(this.getClass().getName(), "Far plane should be
greater than near plane");
            // 抛出非法参数异常
            throw new IllegalArgumentException();
        }

        // 设置近平面与远平面
        setNearFarPlanes(nearPlane, farPlane);

        // 检查设备模式合法性
        if(deviceMode != Device.MODE.MODE_AR
                && deviceMode != Device.MODE.MODE_VR)
        {
            Log.i(this.getClass().getName(), "Device mode should
be Device.MODE.MODE_AR or Device.MODE.MODE_VR");
            // 抛出非法参数异常
            throw new IllegalArgumentException();
        }

        // Vuforia Device 为单例模式，返回设备
        Device device = Device.getInstance();

        // 设置设备当前查看器是否为活动状态
        device.setViewerActive(stereo);

        // 设置 She 被为 AR 或 VR 模式
        device.setMode(deviceMode);
    }

    /**
     * 设置近平面与远平面
     * @param near  近平面
     * @param far   远平面
     */
    public void setNearFarPlanes(float near, float far)
    {
        mNearPlane = near;
        mFarPlane = far;
    }

    /**
     * 设置渲染器的背景大小，配置视频模式和设置相机的图像偏移
     */
    public void configureVideoBackground()
```

```
    {
        // Vuforia 手机摄像头对象
        CameraDevice cameraDevice = CameraDevice.getInstance();

        // Vuforia 获取默认视频模式对象
        VideoMode vm =
cameraDevice.getVideoMode(CameraDevice.MODE.MODE_DEFAULT);

        // Vuforia 视频背景配置选项
        VideoBackgroundConfig config = new
VideoBackgroundConfig();

        // 启用视频背景渲染
        config.setEnabled(true);

        // 设置视频显示时的相对位置
        config.setPosition(new Vec2I(0, 0));

        int xSize = 0, ySize = 0;

        // 通过保持横纵比来保持视频的正常渲染。
        // 如果为纵向，则保持 Activity 的高度不变，然后计算获得宽度
        // 反之为横屏，则保持 Activity 的宽度不变
        if (mIsPortrait)    // 纵向
        {
            // 保持高度不变，通过视频模式对象中的宽高比求得宽度
            xSize = (int) (vm.getHeight() * (mScreenHeight /
(float) vm.getWidth()));
            // 高度保持不变
            ySize = mScreenHeight;

            // 如果计算得到的宽度小于实际 Activity 的宽度
            if (xSize < mScreenWidth)
            {
                // 此时以宽度为准保持不变，求高度
                xSize = mScreenWidth;
                ySize = (int) (mScreenWidth * (vm.getWidth() /
(float) vm.getHeight()));
            }
        }
        else                // 横向
        {
            // 同上亦相反
            xSize = mScreenWidth;
            ySize = (int) (vm.getHeight() * (mScreenWidth / (float)
vm.getWidth()));
```

```
            if (ySize < mScreenHeight)
            {
                xSize = (int) (mScreenHeight * (vm.getWidth() /
(float) vm.getHeight()));
                ySize = mScreenHeight;
            }
        }

        // 将计算得到的X、Y 设置到视频背景设置对象中
        config.setSize(new Vec2I(xSize, ySize));

        Log.i(this.getClass().getName(),
                "Configure Video Background : Video (" +
vm.getWidth()
                + " , " + vm.getHeight() + "), Screen (" +
mScreenWidth + " , "
                + mScreenHeight + "), mSize (" + xSize + " , " + ySize
+ ")");

        // 对渲染器背景进行设置

Renderer.getInstance().setVideoBackgroundConfig(config);
    }

    /**
     * 当配置被改变
     * @param isARActive     是否为AR 模式
     */
    public void onConfigurationChanged(boolean isARActive)
    {
        // 更新方向(纵向横向)信息
        updateActivityOrientation();
        // 获取屏幕尺寸
        storeScreenDimensions();

        // 如果是AR 模式则对渲染器背景进行设置
        if(isARActive)
            configureVideoBackground();

        // 获取绘制图元对象
        mRenderingPrimitives =
Device.getInstance().getRenderingPrimitives();
    }


    /**
     * 当Surface 被创建
     */
```

```
    public void onSurfaceCreated()
    {
        // 初始化渲染
        initRendering();
    }

    /**
     * 根据资源判断Activity 方向（纵向or 横向）
     * 主要表现为修改mIsPortrait 取值，true 纵向；false 横向。
     */
    private void updateActivityOrientation()
    {
        // 获取布局对象
        Configuration config =
mActivity.getResources().getConfiguration();
        // 判断方向
        switch (config.orientation)
        {
            case Configuration.ORIENTATION_PORTRAIT:    // 纵向
                mIsPortrait = true;
                break;
            case Configuration.ORIENTATION_LANDSCAPE:   // 横向
                mIsPortrait = false;
                break;
            case Configuration.ORIENTATION_UNDEFINED:   // 未声明
            default:
                break;
        }

        Log.i(this.getClass().getName(), "Activity is in "
                + (mIsPortrait ? "PORTRAIT" : "LANDSCAPE"));
    }

    /**
     * 获取屏幕尺寸
     * 保存在mScreenWidth、mScreenHeight 种
     */
    private void storeScreenDimensions()
    {
        // 获取Activity 显示尺寸
        Point size = new Point();

mActivity.getWindowManager().getDefaultDisplay().getRealSize(s
ize);
        mScreenWidth = size.x;
        mScreenHeight = size.y;
    }
```

```
    /**
     * 初始化渲染
     */
    void initRendering()
    {
        // 根据背景着色器 src 代码创建着色程序对象并获取其句柄
        vbShaderProgramID
                = SampleUtils.createProgramFromShaderSrc(
                VideoBackgroundShader.VB_VERTEX_SHADER,
                VideoBackgroundShader.VB_FRAGMENT_SHADER);

        // 视频背景的渲染模式
        if (vbShaderProgramID > 0)
        {
            // OpenGL 渲染管道切换到着色器模式，并激活指定着色器程序
            GLES20.glUseProgram(vbShaderProgramID);

            // 使用哪一个采样器进行纹理采样
            vbTexSampler2DHandle
                    = GLES20.glGetUniformLocation(
                    vbShaderProgramID,
                    "texSampler2D");

            // 投影矩阵
            vbProjectionMatrixHandle
                    = GLES20.glGetUniformLocation(
                    vbShaderProgramID,
                    "projectionMatrix");

            // 顶点位置
            vbVertexHandle
                    = GLES20.glGetAttribLocation(
                    vbShaderProgramID,
                    "vertexPosition");

            // 顶点纹理坐标
            vbTexCoordHandle
                    = GLES20.glGetAttribLocation(
                    vbShaderProgramID,
                    "vertexTexCoord");

//            // 获取着色器程序中的投影矩阵句柄，出现两遍？
//            vbProjectionMatrixHandle
//                    = GLES20.glGetUniformLocation(
//                    vbShaderProgramID,
//                    "projectionMatrix");
//
//            // 获取着色器程序中的着色器，出现两遍？
```

```
//            vbTexSampler2DHandle
//                    = GLES20.glGetUniformLocation(
//                    vbShaderProgramID,
//                    "texSampler2D");

            // 停止使用的着色器程序
            GLES20.glUseProgram(0);
        }

        // 创建一个 Vuforia GL 纹理单元
        videoBackgroundTex = new GLTextureUnit();
    }

    /**
     * 主要的绘制方法
     * 该方法为渲染设置状态，设置 AR 增强所需的 3D 转换，并调用特定的渲染方法
     */
    public void render()
    {
        // 清空渲染缓冲区（颜色缓冲区、深度缓冲区）
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT |
GLES20.GL_DEPTH_BUFFER_BIT);

        // Vuforia 状态对象
        State state;

        // 获取当前状态
        state =
TrackerManager.getInstance().getStateUpdater().updateState();

        // 开始渲染，标记渲染状态，并返回最新可用状态对象
        mRenderer.begin(state);

        if
(Renderer.getInstance().getVideoBackgroundConfig().getReflecti
on()
                ==
VIDEO_BACKGROUND_REFLECTION.VIDEO_BACKGROUND_REFLECTION_ON)
        {
            GLES20.glFrontFace(GLES20.GL_CW);   // 前置摄像头
        }
        else
        {
            GLES20.glFrontFace(GLES20.GL_CCW);  // 后置摄像头
        }

        // 返回可用于从这些图元绘制的视图集
```

```
        ViewList viewList =
mRenderingPrimitives.getRenderingViews();

        // 通过循环查看视图列表
        for (int v = 0; v < viewList.getNumViews(); v++)
        {
            // 获取视图 ID
            int viewID = viewList.getView(v);

            Vec4I viewport;

            // 获取图的视口
            viewport = mRenderingPrimitives.getViewport(viewID);

            // 设置视口到当前视图
            // 即设置将要绘制的 2D 物体的窗口的 x、y、w、h
            GLES20.glViewport(viewport.getData()[0],
viewport.getData()[1],
                    viewport.getData()[2],
viewport.getData()[3]);

            // 设置剪切区域
            GLES20.glScissor(viewport.getData()[0],
viewport.getData()[1],
                    viewport.getData()[2],
viewport.getData()[3]);

            // 获取当前视图的投影矩阵
            // COORDINATE_SYSTEM_CAMERA  AR
            // COORDINATE_SYSTEM_WORLD   VR
            Matrix34F projMatrix =
mRenderingPrimitives.getProjectionMatrix(viewID,

COORDINATE_SYSTEM_TYPE.COORDINATE_SYSTEM_CAMERA);

            // 创建 GL 矩阵，并设置近平面与远平面
            // convertPerspectiveProjection2GLMatrix() 将投影矩阵
转换成 GL 矩阵
            float rawProjectionMatrixGL[] =
Tool.convertPerspectiveProjection2GLMatrix(
                    projMatrix,
                    mNearPlane,
                    mFarPlane)
                    .getData();

            // 视口矩阵
            float eyeAdjustmentGL[] =
Tool.convert2GLMatrix(mRenderingPrimitives
```

```
                    .getEyeDisplayAdjustmentMatrix(viewID)).getD
ata();

            // 投影矩阵
            float projectionMatrix[] = new float[16];

            // 将调整应用于投影矩阵，合并矩阵
            Matrix.multiplyMM(projectionMatrix, 0,
                    rawProjectionMatrixGL, 0,
                    eyeAdjustmentGL, 0);

            // 当前视图ID
            currentView = viewID;

            // 跳转到 SampleAppRendererControl 接口的实现中执行，进
行帧渲染
            if(currentView != VIEW.VIEW_POSTPROCESS)
                mRenderingInterface.renderFrame(state,
projectionMatrix);
        }

        // 结束渲染，取消渲染状态标记
        mRenderer.end();
    }

    /**
     * 渲染视频背景
     */
    public void renderVideoBackground()
    {
        // 如果当前视图是待后处理的视图则return
        if(currentView == VIEW.VIEW_POSTPROCESS)
            return;

        int vbVideoTextureUnit = 0;

        // 绑定背景视频的纹理，并通过Vuforia 获取纹理ID
        videoBackgroundTex.setTextureUnit(vbVideoTextureUnit);
        // 更新视频背景纹理
        if
(!mRenderer.updateVideoBackgroundTexture(videoBackgroundTex))
        {
            Log.i(this.getClass().getName(),
                    "Unable to update video background texture");
            return;
        }

        // 获取当前视图的投影矩阵
```

```
        float[] vbProjectionMatrix = Tool.convert2GLMatrix(

mRenderingPrimitives.getVideoBackgroundProjectionMatrix(curren
tView,

COORDINATE_SYSTEM_TYPE.COORDINATE_SYSTEM_CAMERA)).getData();

        // Apply the scene scale on video see-through eyewear, to
scale the video background and augmentation
        // so that the display lines up with the real world
        // This should not be applied on optical see-through
devices, as there is no video background,
        // and the calibration ensures that the augmentation matches
the real world
        // 如果查看器处于活动状态，则返回 true，否则为 false。
        if (Device.getInstance().isViewerActive())
        {
            // 获取缩放因子
            float sceneScaleFactor =
(float)getSceneScaleFactor();
            // 根据缩放因子对投影矩阵的 X、Y 轴进行缩放
            Matrix.scaleM(vbProjectionMatrix, 0,
sceneScaleFactor,
                    sceneScaleFactor, 1.0f);
        }

        // 关闭更新深度缓冲区功能
        GLES20.glDisable(GLES20.GL_DEPTH_TEST);
        // 关闭剔除操作效果
        GLES20.glDisable(GLES20.GL_CULL_FACE);
        // 关闭剪裁测试功能
        GLES20.glDisable(GLES20.GL_SCISSOR_TEST);

        // 返回一个简单的网格，适合渲染视频背景纹理。
        Mesh vbMesh =
mRenderingPrimitives.getVideoBackgroundMesh(currentView);

        // 使用选定的着色程序
        GLES20.glUseProgram(vbShaderProgramID);

        // 指定了渲染时索引为 vbVertexHandle(顶点)的属性数组的数据格
式和位置
        /*
================================================================
=
         * index   指定要修改的顶点属性的索引值
         * size    指定每个顶点属性的组件数量。必须为 1、2、3 或者 4。
初始值为 4。
```

```
         *        （如position 是由3 个（x,y,z）组成，而颜色是4 个
（r,g,b,a））
         * type    指定数组中每个组件的数据类型。可用的符号常量有
GL_BYTE, GL_UNSIGNED_BYTE,
         *        GL_SHORT,GL_UNSIGNED_SHORT, GL_FIXED, 和
GL_FLOAT，初始值为GL_FLOAT。
         * normalized  指定当被访问时，固定点数据值是否应该被归一化
（GL_TRUE）或者直接转换
         *             为固定点值（GL_FALSE）。
         * stride  指定连续顶点属性之间的偏移量。如果为0，那么顶点属
性会被理解为:
         *         它们是紧密排列在一起的。初始值为0。
         * pointer 指定第一个组件在数组的第一个顶点属性中的偏移量。
         *         该数组与GL_ARRAY_BUFFER 绑定，储存于缓冲区中。初
始值为0;
         *
================================================================
= */
        GLES20.glVertexAttribPointer(vbVertexHandle, 3,
GLES20.GL_FLOAT,
                false, 0, vbMesh.getPositions().asFloatBuffer());

        // 指定了渲染时索引值为vbTexCoordHandle(纹理)的属性数组的
数据格式和位置
        GLES20.glVertexAttribPointer(vbTexCoordHandle, 2,
GLES20.GL_FLOAT,
                false, 0, vbMesh.getUVs().asFloatBuffer());

        // 根据之前设置的采样器进行纹理采样变量句柄 进行常亮装在
        GLES20.glUniform1i(vbTexSampler2DHandle,
vbVideoTextureUnit);

        // 使用前面指定的两个数组数据
        GLES20.glEnableVertexAttribArray(vbVertexHandle);
        GLES20.glEnableVertexAttribArray(vbTexCoordHandle);

        // 根据之前设置的投影矩阵变量句柄 进行投影矩阵设置
        GLES20.glUniformMatrix4fv(vbProjectionMatrixHandle, 1,
false,
                vbProjectionMatrix, 0);

        // 开始进行渲染
        // param 1: 类型
        // param 2: 数目
        // param 3: 第四个参数的类型
        // param 4: 绘制的时三角形的索引值
        GLES20.glDrawElements(GLES20.GL_TRIANGLES,
```

```
                vbMesh.getNumTriangles() * 3,
                GLES20.GL_UNSIGNED_SHORT,
                vbMesh.getTriangles().asShortBuffer());

        // 最后，禁用前面指定的两个数组数据
        GLES20.glDisableVertexAttribArray(vbVertexHandle);
        GLES20.glDisableVertexAttribArray(vbTexCoordHandle);

        // 检查OpenGL ES 2.0 是否有错，出现错误信息则打印相应log 信息
        SampleUtils.checkGLError("Rendering of the video
background failed");
    }

    /**
     * 获取场景缩放因子
     * @return
     */
    double getSceneScaleFactor()
    {
        // 获取物理摄像机视野的y 维度
        Vec2F fovVector = CameraDevice.getInstance()
                .getCameraCalibration().getFieldOfViewRads();
        float cameraFovYRads = fovVector.getData()[1];

        // 获取虚拟摄像机视野的y 维度
        float virtualFovYRads = VIRTUAL_FOV_Y_DEGS * M_PI / 180;

        // The scene-scale factor represents the proportion of the
viewport that is filled by
        // the video background when projected onto the same plane.
        // In order to calculate this, let 'd' be the distance
between the cameras and the plane.
        // The height of the projected image 'h' on this plane can
then be calculated:
        // d 为镜头到面的距离，面高为h，则1/2 视角（fov/2）的正切为  1/2
* h / d，即h/2d
        //   tan(fov/2) = h/2d
        // 转换得到：
        //   2d = h/tan(fov/2)
        // Since 'd' is the same for both cameras, we can combine
the equations for the two cameras:
        // 物理摄像头与虚拟摄像头需要成像的面到摄像头的距离都为d，可
以得到如下公式
        //   hPhysical/tan(fovPhysical/2) =
hVirtual/tan(fovVirtual/2)
        // 转换得到：
        //   hPhysical/hVirtual =
```

```
tan(fovPhysical/2)/tan(fovVirtual/2)
        // 物理成像面 / 虚拟成像面 就是所需的缩放因子，所以 return 如
下表达式
        return Math.tan(cameraFovYRads / 2) /
Math.tan(virtualFovYRads / 2);
    }
}
```

步骤十四　在项目中添加 SampleApplicationSession.java 文件，为应用程序会话类。

```
package com.hebeiedu.myvuforiademo;

import android.app.Activity;
import android.content.pm.ActivityInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.util.Log;
import android.view.OrientationEventListener;
import android.view.WindowManager;

import com.vuforia.CameraDevice;
import com.vuforia.INIT_FLAGS;
import com.vuforia.State;
import com.vuforia.Vuforia;

/**
 * 应用程序会话类
 */
public class SampleApplicationSession implements
Vuforia.UpdateCallbackInterface {

    // 应用程序会话控制器
    private SampleApplicationControl mSessionControl;

    // 线程锁对象
    private Object mShutdownLock = new Object();

    // 当前 Acitivy 的引用
    private Activity mActivity;

    // Vuforia 所使用的 OpenGL ES 版本
    // 1    Vuforia.GL_20   OpenGL ES 2.0
    // 8    Vuforia.GL_30   OpenGL ES 3.0
```

```
    private int mVuforiaFlags = 0;

    // 初始化 Vuforia 的异步任务，用于进行 vuforia 的初始化
    private InitVuforiaTask mInitVuforiaTask;
    // 加载跟踪器数据的异步任务，用于加载追踪器数据
    private LoadTrackerTask mLoadTrackerTask;

    // 追踪器是否启动
    private boolean mStarted = false;
    // 相机是否启用
    private boolean mCameraRunning = false;

    // CAMERA_DIRECTION 摄像头的设备
    // CAMERA_DIRECTION_DEFAULT 默认摄像头
    // CAMERA_DIRECTION_FRONT 前摄像头
    // CAMERA_DIRECTION_BACK 后摄像头
    private int mCamera =
CameraDevice.CAMERA_DIRECTION.CAMERA_DIRECTION_DEFAULT;


    public SampleApplicationSession(SampleApplicationControl
sessionControl) {
        mSessionControl = sessionControl;
    }

    /**
     * 初始化 Vuforia，以及设置相关参数
     * 共完成 Vuforia SDK 初始化、追踪器初始化、追踪器数据集初始化及加
载任务。
     * 完成后会回调 SampleApplicationControl.onInitARDone()
     * @param activity          当前 Activity
     * @param screenOrientation 屏幕方向
     */
    public void initAR(Activity activity, int screenOrientation)
{
        // 异常对象
        SampleApplicationException vuforiaException = null;

        // 保存当前 Acitvity 的引用
        mActivity = activity;

        /*
          Build.VERSION.SDK_INT 当前系统版本
           Build.VERSION_CODES   编译时 SDK 版本
                ECLAIR_MR1        January 2010:   Android 2.1
                FROYO             June 2010:      Android 2.2
                GINGERBREAD       November 2010:  Android 2.3
```

```
                GINGERBREAD_MR1   February 2011:  Android 2.3.3.
                HONEYCOMB         February 2011:  Android 3.0.
                HONEYCOMB_MR1     May 2011:       Android 3.1.
                HONEYCOMB_MR2     June 2011:      Android 3.2.
                ICE_CREAM_SANDWICH                Android 4.0.

            ActivityInfo.SCREEN_ORIENTATION_SENSOR       由物理感
应器决定显示方向
            ActivityInfo.SCREEN_ORIENTATION_FULL_SENSOR  根据重力
变换朝向，全屏旋转
        */
        // 如果当前显示方向由物理传感器决定，并且编译版本大于2.2，则
将屏幕方向设置为可全屏旋转
        if ((screenOrientation ==
ActivityInfo.SCREEN_ORIENTATION_SENSOR)
                && (Build.VERSION.SDK_INT >
Build.VERSION_CODES.FROYO))
            screenOrientation =
ActivityInfo.SCREEN_ORIENTATION_FULL_SENSOR;

        /**
         * 使用 OrientationChangeListener 在此处捕获方向变化
         * 当180 度旋转时，Android 将不再发送
Activity.onConfigurationChanged()回调
         * 也就是向左横屏到向右横屏时，Vuforia 需要对此操作做出反应
         * 并且SampleApplicationSession 需要更新投影矩阵
         * OrientationEventListener 方向事件监听器
         */
        OrientationEventListener orientationEventListener
                = new OrientationEventListener(mActivity) {
            @Override
            public void onOrientationChanged(int i) {
                int activityRotation =
mActivity.getWindowManager()
                        .getDefaultDisplay().getRotation();
                if (mLastRotation != activityRotation) {
                    mLastRotation = activityRotation;
                }
            }

            int mLastRotation = -1;
        };

        // 如果方向时间监听器能够检测方向则启用
        if (orientationEventListener.canDetectOrientation())
            orientationEventListener.enable();

        // 设置屏幕方向
```

```
        mActivity.setRequestedOrientation(screenOrientation);

        // 如果窗口是可见的话则一直保持光亮和可见
        mActivity.getWindow().setFlags(
                WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON,
                WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

        // Vuforia 所使用的OpenGL ES 版本为2.0
        mVuforiaFlags = INIT_FLAGS.GL_20;

        /**
         * mInitVuforiaTask 用于完成Vuforia 的异步任务对象。
         * 同时可以通过该对象判断本方法initAR()是否执行两遍。
         */
        if (mInitVuforiaTask != null) {
            String logMessage = "Cannot initialize SDK twice";
            vuforiaException = new SampleApplicationException(

SampleApplicationException.VUFORIA_ALREADY_INITIALIZATED,
                    logMessage);
            Log.e(this.getClass().getName(), logMessage);
        }

        /**
         * 通过InitVuforiaTask 异步任务类完成Vuforia AR 功能的初始
化，
         * 其中通过与SampleApplicationControl 类接口的相互调用，
         * 完成Vuforia SDK 初始化、追踪器初始化、追踪器数据集初始化及
加载。
         * 异步任务完成后会调用
SampleApplicationControl.onInitARDone()
         */
        if (vuforiaException == null) {
            try {
                mInitVuforiaTask = new InitVuforiaTask();
                mInitVuforiaTask.execute();
            } catch (Exception e) {
                String logMessage = "Initializing Vuforia SDK
failed";
                vuforiaException = new SampleApplicationException(

SampleApplicationException.INITIALIZATION_FAILURE,
                        logMessage);
                Log.e(this.getClass().getName(), logMessage);
            }
        }

        if (vuforiaException != null)
```

```
            mSessionControl.onInitARDone(vuforiaException);
    }

    /**
     * 当追踪器追踪完成后调用
     *
     * @param state
     */
    @Override
    public void Vuforia_onUpdate(State state) {
        mSessionControl.onVuforiaUpdate(state);
    }

    /**
     * 初始化 Vuforia 的异步任务类
     */
    private class InitVuforiaTask extends AsyncTask<Void, Integer,
Boolean> {
        // 初始化进度值为一个无效值
        private int mProgressValue = -1;

        /**
         * 此方法中进行比较耗时的操作，例如 Vuforia SDK 的初始化工作，
此方法中不能直接操作 UI
         */
        protected Boolean doInBackground(Void... params) {
            // 上同步锁
            synchronized (mShutdownLock) {
                // 对 Vuforia 进行初始化参数设置，Activy、OpenGL ES
版本、密钥
                Vuforia.setInitParameters(mActivity,
mVuforiaFlags,

"AaG3I6T/////AAAAGSQzUxGkMU8WpTGRpvwrPX1kgko7p+qHnlxtsDTUwktDj
K0VqAlXnzoo9TA2ZXa2kXiikelYSdfmWt53YN4xw0R/1f1peAarMCQTXn/Kmfo
k2zZGRtWLXS8pXUA00ToOwaj9H9lI9YDl7K/KUpgL9dZj5UEyJwEl/wwyTSd3q
25btwDqzEAKGw5INBQphbZPCTL8+GMyc2d9F+PfBfyw2AKu3HjizpV1ooLYxJl
fkK2FoNrzKS5yPlh1utyuOv8IGtsoelnypS3X86uEYh3WcwwcDI4EkF3bvxqCm
ddEZ9llRwXEBclfJdeiLJYOrpHktcSdcQ++35SJGRSFRgK14HOGpgM+XN9rOTm
Dshk5dOJx");

                // 等待初始化完成
                do {
                    // Vuforia.init()进行初始化，返回初始化进度
(0~100)，
                    // 当返回值为100 时表示初始化完成，当返回值为-1
时表示初始化出错
                    mProgressValue = Vuforia.init();
```

```
                    // 更新进度信息，onProgressUpdate()方法会进行响
应
                    publishProgress(mProgressValue);

                    // 当没有被取消，并且完成百分比在0-100时继续进行
初始化
                } while (!isCancelled() && mProgressValue >= 0
                        && mProgressValue < 100);

                return (mProgressValue > 0);
            }
        }

        /**
         * 响应doInBackground()中调用publishProgress()操作
         */
        protected void onProgressUpdate(Integer... values) {
            // 处理一些与进度值有关的事情，例如更新进度条显示等
        }

        /**
         * 当doInBackground()后台操作完成时此方法会被调用，
         * 参数为doInBackground()的返回值，此方法在主线程执行，可用
来操作UI。
         */
        protected void onPostExecute(Boolean result) {
            // 异常对象
            SampleApplicationException vuforiaException = null;

            if (result) {
                // 初始化跟踪器
                boolean initTrackersResult;
                initTrackersResult =
mSessionControl.doInitTrackers();

                // 当追踪器创建完成后加载追踪器数据
                if (initTrackersResult) {
                    try {
                        // 创建异步加载跟踪器数据类
                        mLoadTrackerTask = new LoadTrackerTask();
                        mLoadTrackerTask.execute();

                    } catch (Exception e) {
                        String logMessage = "Loading tracking data
set failed";
                        vuforiaException = new
SampleApplicationException(
```

```
SampleApplicationException.LOADING_TRACKERS_FAILURE,
                                logMessage);
                        Log.e(this.getClass().getName(),
logMessage);

mSessionControl.onInitARDone(vuforiaException);
                    }

                } else {
                    vuforiaException = new
SampleApplicationException(

SampleApplicationException.TRACKERS_INITIALIZATION_FAILURE,
                            "Failed to initialize trackers");

mSessionControl.onInitARDone(vuforiaException);
                }
            } else {
                String logMessage;

                // 初始化失败获取错误信息
                logMessage =
getInitializationErrorString(mProgressValue);

                Log.e(this.getClass().getName(),
                        "InitVuforiaTask.onPostExecute: " +
logMessage
                                + " Exiting.");

                // 发送异常并停止初始化
                vuforiaException = new SampleApplicationException(

SampleApplicationException.INITIALIZATION_FAILURE,
                        logMessage);
                mSessionControl.onInitARDone(vuforiaException);
            }
        }
    }

    /**
     * 加载跟踪器数据的异步任务类
     */
    private class LoadTrackerTask extends AsyncTask<Void, Integer,
Boolean> {
        protected Boolean doInBackground(Void... params) {
            // 上同步锁
            synchronized (mShutdownLock) {
                // 加载跟踪器数据集
```

```
                return mSessionControl.doLoadTrackersData();
            }
        }

        protected void onPostExecute(Boolean result) {
            SampleApplicationException vuforiaException = null;

            Log.d(this.getClass().getName(),
"LoadTrackerTask.onPostExecute: execution "
                    + (result ? "successful" : "failed"));

            if (!result) {
                String logMessage = "Failed to load tracker data.";
                // 数据集加载出错
                Log.e(this.getClass().getName(), logMessage);
                vuforiaException = new SampleApplicationException(

SampleApplicationException.LOADING_TRACKERS_FAILURE,
                        logMessage);
            } else {
                // 提示系统进行垃圾回收，但是无法保证系统真的会进行垃
圾回收
                System.gc();

                // 注册回调，当 SDK 完成当前数据集追踪之后回调
Vuforia_onUpdate()

Vuforia.registerCallback(SampleApplicationSession.this);

                mStarted = true;
            }

            // 完成加载跟踪，更新应用程序状态，发送异常检查错误
            mSessionControl.onInitARDone(vuforiaException);
        }
    }

    /**
     * 根据错误代码返回错误信息
     * @param code      错误编号
     * @return          错误文字信息
     */
    private String getInitializationErrorString(int code)
    {
        if (code == Vuforia.INIT_DEVICE_NOT_SUPPORTED)
            return
mActivity.getString(R.string.INIT_ERROR_DEVICE_NOT_SUPPORTED);
        if (code == Vuforia.INIT_NO_CAMERA_ACCESS)
```

```
            return
mActivity.getString(R.string.INIT_ERROR_NO_CAMERA_ACCESS);
        if (code == Vuforia.INIT_LICENSE_ERROR_MISSING_KEY)
            return
mActivity.getString(R.string.INIT_LICENSE_ERROR_MISSING_KEY);
        if (code == Vuforia.INIT_LICENSE_ERROR_INVALID_KEY)
            return
mActivity.getString(R.string.INIT_LICENSE_ERROR_INVALID_KEY);
        if (code ==
Vuforia.INIT_LICENSE_ERROR_NO_NETWORK_TRANSIENT)
            return
mActivity.getString(R.string.INIT_LICENSE_ERROR_NO_NETWORK_TRA
NSIENT);
        if (code ==
Vuforia.INIT_LICENSE_ERROR_NO_NETWORK_PERMANENT)
            return
mActivity.getString(R.string.INIT_LICENSE_ERROR_NO_NETWORK_PER
MANENT);
        if (code == Vuforia.INIT_LICENSE_ERROR_CANCELED_KEY)
            return
mActivity.getString(R.string.INIT_LICENSE_ERROR_CANCELED_KEY);
        if (code ==
Vuforia.INIT_LICENSE_ERROR_PRODUCT_TYPE_MISMATCH)
            return
mActivity.getString(R.string.INIT_LICENSE_ERROR_PRODUCT_TYPE_M
ISMATCH);
        else
        {
            return
mActivity.getString(R.string.INIT_LICENSE_ERROR_UNKNOWN_ERROR)
;
        }
    }

    /**
     * 启动AR，初始化并先运行相机，后运行追踪器
     * @param camera    相机类型
     * @throws SampleApplicationException
     */
    public void startAR(int camera)
            throws SampleApplicationException
    {
        String error;
        if(mCameraRunning)
        {
            error = "Camera already running, unable to open again";
            Log.e(this.getClass().getName(), error);
            throw new SampleApplicationException(
```

```
SampleApplicationException.CAMERA_INITIALIZATION_FAILURE,
                    error);
        }

        // 记录当前相机设备的选择
        mCamera = camera;

        // 初始化相机设备
        if (!CameraDevice.getInstance().init(camera))
        {
            error = "Unable to open camera device: " + camera;
            Log.e(this.getClass().getName(), error);
            throw new SampleApplicationException(

SampleApplicationException.CAMERA_INITIALIZATION_FAILURE,
                    error);
        }

        // 选择视频模式
        if (!CameraDevice.getInstance().selectVideoMode(
                CameraDevice.MODE.MODE_DEFAULT))
        {
            error = "Unable to set video mode";
            Log.e(this.getClass().getName(), error);
            throw new SampleApplicationException(

SampleApplicationException.CAMERA_INITIALIZATION_FAILURE,
                    error);
        }

        // 启动相机
        if (!CameraDevice.getInstance().start())
        {
            error = "Unable to start camera device: " + camera;
            Log.e(this.getClass().getName(), error);
            throw new SampleApplicationException(

SampleApplicationException.CAMERA_INITIALIZATION_FAILURE,
                    error);
        }

        // 启动追踪器
        mSessionControl.doStartTrackers();

        // 记录相机已经启动
        mCameraRunning = true;

        /*
```

```
            FOCUS_MODE_NORMAL  -通过设备提供的默认的对焦模式
            FOCUS_MODE_TRIGGERAUTO  -设置此对焦模式将触发一个自动对焦
操作。
            FOCUS_MODE_CONTINUOUSAUTO  -首发的Android 2.3 和iOS 设备
这一对焦模式下可切换式
              驱动级的连续自动对焦的摄像头。这是AR 的应用程序的最佳对焦
模式，因为它保证了相机
              聚焦在目标上，从而产生最佳的跟踪结果。
            FOCUS_MODE_INFINITY  -设置相机为“无穷大”，通过摄像头驱动程
序的实现提供。（不支持IOS）。
            FOCUS_MODE_MACRO -设置相机“微距”模式，由相机驱动程序的实现
提供。
              这提供了特写（APPX。15 厘米），在AR 调校很少使用的距离急
剧摄像机图像。（不支持IOS）。
             */
            if(!CameraDevice.getInstance()
                    .setFocusMode(CameraDevice.FOCUS_MODE.FOCUS_MODE
_CONTINUOUSAUTO))
            {

if(!CameraDevice.getInstance().setFocusMode(CameraDevice.FOCUS
_MODE.FOCUS_MODE_TRIGGERAUTO))

CameraDevice.getInstance().setFocusMode(CameraDevice.FOCUS_MOD
E.FOCUS_MODE_NORMAL);
            }
        }

        /**
         * 当显示的Surface 尺寸发生变化，例如横纵屏切换
         * @param width      宽度
         * @param height     长度
         */
        public void onSurfaceChanged(int width, int height)
        {
            Vuforia.onSurfaceChanged(width, height);
        }

        /**
         * 当Surface 被创建
         */
        public void onSurfaceCreated()
        {
            Vuforia.onSurfaceCreated();
        }

        /**
```

```
     * 恢复Vuforia，重新启动追踪器和摄像头
     * @throws SampleApplicationException
     */
    public void resumeAR() throws SampleApplicationException
    {
        // Vuforia SDK 恢复
        Vuforia.onResume();

        if (mStarted)
        {
            startAR(mCamera);
        }
    }

    /**
     * 暂停摄像头与Vuforia SDK
     * @throws SampleApplicationException
     */
    public void pauseAR() throws SampleApplicationException
    {
        if (mStarted)
        {
            stopCamera();
        }

        Vuforia.onPause();
    }

    /**
     * 停止摄像头
     */
    public void stopCamera()
    {
        if (mCameraRunning)
        {
            mSessionControl.doStopTrackers();
            mCameraRunning = false;
            CameraDevice.getInstance().stop();
            CameraDevice.getInstance().deinit();
        }
    }

    /**
     * 停止所有所做的初始化工作，并停止Vuforia SDK 工作
     * @throws SampleApplicationException
     */
    public void stopAR() throws SampleApplicationException
    {
        if (mInitVuforiaTask != null
```

```
                && mInitVuforiaTask.getStatus() !=
InitVuforiaTask.Status.FINISHED)
        {
            mInitVuforiaTask.cancel(true);
            mInitVuforiaTask = null;
        }

        if (mLoadTrackerTask != null
                && mLoadTrackerTask.getStatus() !=
LoadTrackerTask.Status.FINISHED)
        {
            mLoadTrackerTask.cancel(true);
            mLoadTrackerTask = null;
        }

        mInitVuforiaTask = null;
        mLoadTrackerTask = null;

        mStarted = false;

        stopCamera();

        synchronized (mShutdownLock)
        {

            boolean unloadTrackersResult;
            boolean deinitTrackersResult;

            // 销毁跟踪器数据
            unloadTrackersResult =
mSessionControl.doUnloadTrackersData();

            // 取消跟踪器的初始化
            deinitTrackersResult =
mSessionControl.doDeinitTrackers();

            // 取消 Vuforia SDK 设置
            Vuforia.deinit();

            if (!unloadTrackersResult)
                throw new SampleApplicationException(

SampleApplicationException.UNLOADING_TRACKERS_FAILURE,
                        "Failed to unload trackers\' data");

            if (!deinitTrackersResult)
                throw new SampleApplicationException(

SampleApplicationException.TRACKERS_DEINITIALIZATION_FAILURE,
```

```
                        "Failed to deinitialize trackers");

            }
        }
    }
```

步骤十五　在项目中添加 SampleApplicationGLView.java 文件，为自定义的 OpenGL ES 的 SurfaceView 类。

```
package com.hebeiedu.myvuforiademo;

import android.content.Context;
import android.graphics.PixelFormat;
import android.opengl.GLSurfaceView;
import android.util.Log;

import javax.microedition.khronos.egl.EGL10;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.egl.EGLContext;
import javax.microedition.khronos.egl.EGLDisplay;

/**
 * 自定义的OpenGL ES 的 SurfaceView
 */
public class SampleApplicationGLView extends GLSurfaceView {

    public SampleApplicationGLView(Context context) {
        super(context);
    }

    /**
     * OpenGL ES SurfaceView 初始化
     * @param translucent   是否半透明
     * @param depth         深度尺寸
     * @param stencil       模型尺寸
     */
    public void init(boolean translucent, int depth, int stencil)
    {
        Log.i(this.getClass().getName(), "Using OpenGL ES 2.0");

        // translucent 半透明的    opaque 不透明的
        Log.i(this.getClass().getName(), "Using "
                + (translucent ? "translucent" : "opaque")
                + " GLView, depth buffer size: " + depth
```

```
                + ", stencil size: " + stencil);

        // 如果需要设置半透明格式，则设置窗口格式为半透明格式
        if (translucent)
        {
            this.getHolder().setFormat(PixelFormat.TRANSLUCENT);
        }

        // 设置上下文工厂使用2.0进行渲染
        setEGLContextFactory(new ContextFactory());

        // 默认GLSurfaceView选择一个最接近16-bit R5G6B5的Surface，
        // 拥有一个16-bit depth的buffer，没有模板（no stencil）。
        // 如果更喜欢一个不同的Surface（例如，你不需要depth buffer）
        // 你能通过setEGLConfigChooser方法覆盖默认的行为。
        setEGLConfigChooser(translucent
                ? new ConfigChooser(8, 8, 8, 8, depth, stencil)
                : new ConfigChooser(5, 6, 5, 0, depth, stencil));
    }

    /**
     * 检查OpengGL 错误
     * @param prompt
     * @param egl
     */
    private static void checkEglError(String prompt, EGL10 egl)
    {
        int error;
        while ((error = egl.eglGetError()) != EGL10.EGL_SUCCESS)
        {

Log.e(Thread.currentThread().getStackTrace()[1].getClassName()
,
                    String.format("%s: EGL error: 0x%x", prompt,
error));
        }
    }

    /**
     * OpenGL 上下文
     * OpenGL的pipeline从程序的角度看就是一个状态机，
     * 有当前的颜色、纹理坐标、变换矩阵、绚染模式等一大堆状态，
     * 这些状态作用于程序提交的顶点 坐标等图元从而形成帧缓冲内的像素。
     * 在OpenGL的编程接口中，Context就代表这个状态机，
     * 程序的主要工作就是向Context提供图元、设置状态，
     * 偶尔也从Context里获取一些信息。
     */
    private static class ContextFactory implements
```

```
            EGLContextFactory
    {
        private static int EGL_CONTEXT_CLIENT_VERSION = 0x3098;

        public EGLContext createContext(EGL10 egl, EGLDisplay
display,
                                        EGLConfig eglConfig)
        {
            EGLContext context;

            Log.i(this.getClass().getName(), "Creating OpenGL ES
2.0 context");
            checkEglError("Before eglCreateContext", egl);

            int[] attrib_list_gl20 = { EGL_CONTEXT_CLIENT_VERSION,
                    2, EGL10.EGL_NONE};

            // display : 系统显示 ID 或句柄，是一个关联系统物理屏幕
的通用数据类型
            // eglConfig : Surface 的 EGL 配置
            // EGL_NO_CONTEXT : 表示不向其它的context 共享资源
            context = egl.eglCreateContext(display, eglConfig,
                    EGL10.EGL_NO_CONTEXT, attrib_list_gl20);

            checkEglError("After eglCreateContext", egl);

            return context;
        }


        public void destroyContext(EGL10 egl, EGLDisplay display,
                                   EGLContext context)
        {
            egl.eglDestroyContext(display, context);
        }
    }

    /**
     * 配置选项
     */
    private static class ConfigChooser implements
            EGLConfigChooser
    {
        // 子类能够访问这些属性成员
        protected int mRedSize;
        protected int mGreenSize;
        protected int mBlueSize;
        protected int mAlphaSize;
```

```
        protected int mDepthSize;
        protected int mStencilSize;

        private int[] mValue = new int[1];

        public ConfigChooser(int r, int g, int b, int a,
                             int depth, int stencil)
        {
            mRedSize = r;
            mGreenSize = g;
            mBlueSize = b;
            mAlphaSize = a;
            mDepthSize = depth;
            mStencilSize = stencil;
        }

        private EGLConfig getMatchingConfig(EGL10 egl, EGLDisplay
display,
                                            int[] configAttribs)
        {
            // 获取匹配的最低限度的 EGL 配置数量
            int[] num_config = new int[1];
            egl.eglChooseConfig(display, configAttribs, null, 0,
num_config);

            int numConfigs = num_config[0];
            if (numConfigs <= 0)
                throw new IllegalArgumentException("No matching
EGL configs");

            // 获取匹配的最低限度的 EGL 配置
            EGLConfig[] configs = new EGLConfig[numConfigs];
            egl.eglChooseConfig(display, configAttribs, configs,
numConfigs,
                    num_config);

            // 返回最佳匹配项
            return chooseConfig(egl, display, configs);
        }

        // 选择配置
        public EGLConfig chooseConfig(EGL10 egl, EGLDisplay
display)
        {
            final int EGL_OPENGL_ES2_BIT = 0x0004;
            final int[] s_configAttribs_gl20 =
{ EGL10.EGL_RED_SIZE, 4,
                    EGL10.EGL_GREEN_SIZE, 4, EGL10.EGL_BLUE_SIZE,
4,
```

```
                    EGL10.EGL_RENDERABLE_TYPE,
EGL_OPENGL_ES2_BIT,
                    EGL10.EGL_NONE };

            return getMatchingConfig(egl, display,
s_configAttribs_gl20);
        }

        // 选择最佳匹配配置
        public EGLConfig chooseConfig(EGL10 egl, EGLDisplay
display,
                                      EGLConfig[] configs)
        {
            for (EGLConfig config : configs)
            {
                int d = findConfigAttrib(egl, display, config,
                        EGL10.EGL_DEPTH_SIZE, 0);
                int s = findConfigAttrib(egl, display, config,
                        EGL10.EGL_STENCIL_SIZE, 0);

                // 应该满足最小的深度值和模板尺寸
                if (d < mDepthSize || s < mStencilSize)
                    continue;

                // 寻找精确匹配的 RGBA
                int r = findConfigAttrib(egl, display, config,
                        EGL10.EGL_RED_SIZE, 0);
                int g = findConfigAttrib(egl, display, config,
                        EGL10.EGL_GREEN_SIZE, 0);
                int b = findConfigAttrib(egl, display, config,
                        EGL10.EGL_BLUE_SIZE, 0);
                int a = findConfigAttrib(egl, display, config,
                        EGL10.EGL_ALPHA_SIZE, 0);

                if (r == mRedSize && g == mGreenSize && b == mBlueSize
                        && a == mAlphaSize)
                    return config;
            }

            return null;
        }

        /**
         * 在指定 EGLConfig 中查找指定 attrib 的值，如果没有此属性，返
回指定的默认值
         * @param egl
         * @param display
         * @param config        指定的 EGLConfig
```

```
         * @param attribute     指定的attrib
         * @param defaultValue  查找失败时返回的默认值
         * @return  查找成功，返回查找值；查找失败，返回参数中指定的
默认值
         */
        private int findConfigAttrib(EGL10 egl,
                                     EGLDisplay display,
                                     EGLConfig config,
                                     int attribute,
                                     int defaultValue)
        {

            if (egl.eglGetConfigAttrib(display, config,
attribute, mValue))
                return mValue[0];

            return defaultValue;
        }
    }
}
```

步骤十六　在项目中添加 SampleApplicationException.java 文件，作为异常处理类。

```
package com.hebeiedu.myvuforiademo;

/**
 * 异常对象类，用于发送在Vuforia 期间的各种错误
 */
public class SampleApplicationException extends Exception
{
    // 初始化失败
    public static final int INITIALIZATION_FAILURE = 0;
    // Vuforia SDK 已经初始化
    public static final int VUFORIA_ALREADY_INITIALIZATED = 1;
    // 追踪器初始化失败
    public static final int TRACKERS_INITIALIZATION_FAILURE = 2;
    // 加载追踪器数据失败
    public static final int LOADING_TRACKERS_FAILURE = 3;
    // 卸载跟踪器数据失败
    public static final int UNLOADING_TRACKERS_FAILURE = 4;
    // 追踪器析构失败
    public static final int TRACKERS_DEINITIALIZATION_FAILURE = 5;
    // 摄像头初始化失败
```

```
    public static final int CAMERA_INITIALIZATION_FAILURE = 6;

    // 初始化错误编号
    private int mCode = -1;
    // 初始化错误信息
    private String mString = "";

    /**
     * 异常对象构造器
     * @param code          错误编号
     * @param description   错误描述信息
     */
    public SampleApplicationException(int code, String
description)
    {
        super(description);
        mCode = code;
        mString = description;
    }

    /**
     * 获取错误编号
     * @return  错误编号
     */
    public int getCode()
    {
        return mCode;
    }

    /**
     * 获取错误描述信息
     * @return  错误描述信息
     */
    public String getString()
    {
        return mString;
    }
}
```

步骤十七　在项目中添加 SampleApplicationControl.java 文件，为应用程序控制器接口类。

```
package com.hebeiedu.myvuforiademo;
```

```java
import com.vuforia.State;

/**
 * 应用控制器接口类
 */
public interface SampleApplicationControl
{
    /**
     * 初始化跟踪器
     * 在SampleApplicationSession 中的InitVuforiaTask 类中
     * 完成Vuforia SDK 初始化工作后调用
     * @return  true    初始化成功
     *          false   初始化失败
     */
    boolean doInitTrackers();

    /**
     * 加载跟踪器数据
     * 在SampleApplicationSession 中的LoadTrackerTask 类中
     * 完成创建追踪器工作后调用
     * @return  true    初始化成功
     *          false   初始化失败
     */
    boolean doLoadTrackersData();

    /**
     * 开始跟踪
     * 在SampleApplicationSession 中的startAR()方法中调用
     * 此时已经初始化完成Vuforia AR 相关设置
     * 完成OpenGL ES 相关设置
     * @return  true    成功
     *          false   失败
     */
    boolean doStartTrackers();

    /**
     * 停止跟踪
     * @return
     */
    boolean doStopTrackers();

    /**
     * 销毁跟踪器数据
     * @return
     */
    boolean doUnloadTrackersData();
```

```
    /**
     * 取消跟踪器的初始化
     * @return
     */
    boolean doDeinitTrackers();

    /**
     * Vuforia AR 初始化完成
     * 在 SampleApplicationSession 中的 LoadTrackerTask 类中
     * 追踪器数据加载完成后调用
     * @param e      异常对象，用于检测初始化是否完成
     */
    void onInitARDone(SampleApplicationException e);

    /**
     * 回调函数，每个周期结束后调用
     * @param state
     */
    void onVuforiaUpdate(State state);

}
```

步骤十八　在项目中添加 MyShader.java 文件。保存着色器代码。

```
package com.hebeiedu.myvuforiademo;

public class MyShader {

    public static final String CUBE_MESH_VERTEX_SHADER = " \n" +
"\n"
            + "attribute vec4 vertexPosition; \n"
            + "attribute vec2 vertexTexCoord; \n" + "\n"
            + "varying vec2 texCoord; \n" + "\n"
            + "uniform mat4 modelViewProjectionMatrix; \n" + "\n"
            + "void main() \n" + "{ \n"
            + "   gl_Position = modelViewProjectionMatrix *
vertexPosition; \n"
            + "   texCoord = vertexTexCoord; \n"
            + "} \n";

    public static final String CUBE_MESH_FRAGMENT_SHADER = " \n"
+ "\n"
            + "precision mediump float; \n" + " \n"
            + "varying vec2 texCoord; \n"
            + "uniform sampler2D texSampler2D; \n" + " \n"
```

```
            + "void main() \n"
            + "{ \n" + "   gl_FragColor = texture2D(texSampler2D,
texCoord); \n"
            + "} \n";

}
```

步骤十九　在项目中添加 MyTargetRenderer.java 文件，为自定义的渲染器类。

```
package com.hebeiedu.myvuforiademo;

import android.opengl.GLES20;
import android.opengl.GLSurfaceView;
import android.opengl.Matrix;
import android.util.Log;

import com.vuforia.Device;
import com.vuforia.Matrix44F;
import com.vuforia.State;
import com.vuforia.Tool;
import com.vuforia.Trackable;
import com.vuforia.TrackableResult;
import com.vuforia.Vuforia;

import java.io.IOException;
import java.util.Vector;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

public class MyTargetRenderer implements GLSurfaceView.Renderer,
        SampleAppRendererControl {

    // 渲染器所属Activity
    private MyTargets mActivity;
    // 当前应用会话
    private SampleApplicationSession vuforiaAppSession;
    // 渲染器常用工具对象
    private SampleAppRenderer mSampleAppRenderer;
    // 纹理动态数组
    private Vector<Texture> mTextures;
    // 渲染器是否激活
    private boolean mIsActive = false;

    // 渲染程序句柄
```

```
    private int shaderProgramID;
    // 着色器中顶点坐标句柄
    private int vertexHandle;
    // 着色器中纹理映射坐标句柄
    private int textureCoordHandle;
    // 着色器中总转换矩阵句柄
    private int mvpMatrixHandle;
    // 着色器中纹理句柄
    private int texSampler2DHandle;

    // 模型是否加载
    private boolean mModelIsLoaded = false;

    // 模型数据对象
    ModelObject mMyObject;

    /**
     * 构造器
     *
     * @param activity 当前Activity
     * @param session  当前应用会话
     */
    public MyTargetRenderer(MyTargets activity,
                            SampleApplicationSession session) {
        mActivity = activity;
        vuforiaAppSession = session;

        // SampleAppRenderer 用来封装渲染图元操作，设置AR/VR 或者立体模式
        // 此处设置视锥体的近平面，远平面
        mSampleAppRenderer = new SampleAppRenderer(this,
mActivity,
                Device.MODE.MODE_AR, false, 0.1f, 5000f);
    }

    /**
     * 设置纹理
     *
     * @param textures 纹理动态数组
     */
    public void setTextures(Vector<Texture> textures) {
        mTextures = textures;
    }

    /**
     * 渲染器是否激活，配置渲染器背景
     *
```

```
     * @param active
     */
    public void setActive(boolean active) {
        mIsActive = active;

        if (mIsActive) {
            // 配置渲染器背景
            mSampleAppRenderer.configureVideoBackground();
        }
    }

    // 渲染器初始化
    private void initRendering() {
        // 设置屏幕背景色 RGBA
        GLES20.glClearColor(0.0f, 0.0f, 0.0f,
Vuforia.requiresAlpha() ? 0.0f
                : 1.0f);
        /**
         * Android 上使用 Opengl 进行滤镜渲染效率较高，
         * 比起单纯的使用 CPU 给用户带来的体验会好很多。
         * 滤镜的对象是图片，图片是以 Bitmap 的形式表示，Opengl 不能直
接处理 Bitmap，
         * 在 Android 上一般是通过 GLSurfaceView 来进行渲染的，
         * 也可以说成 Android 需要借助 GLSurfaceView 来完成对图片的渲
染。
         */
        /**
         * GlSurfaceView 的图片来源依然是 Bitmap，但是 Bitmap 需要以
纹理（Texture）的形式载入到 Opengl 中。
         * 一下为载入纹理的步骤：
         *  1. GLES20.glGenTextures() : 生成纹理资源的句柄
         *     void glGenTextures(GLsizei n, GLuint *textures)
         *    参数说明：
         *     n：用来生成纹理的数量（下面为 1）
         *     textures：存储纹理索引的（下面为一个数组 t.mTextureID）
         *  2. GLES20.glBindTexture(): 绑定句柄
           *  3. GLUtils.texImage2D() : 将 bitmap 传递到已经绑定的纹
理中
         *  4. GLES20.glTexParameteri() : 设置纹理属性，过滤方式，
拉伸方式等
         */
        for (Texture t : mTextures) {
            // 1 生成纹理资源的句柄
            GLES20.glGenTextures(1, t.mTextureID, 0);
            // 2 绑定句柄
            GLES20.glBindTexture(GLES20.GL_TEXTURE_2D,
t.mTextureID[0]);
```

```
            // 3 设置纹理属性，过滤方式，拉伸方式等
            // 这是纹理过滤，MIN,LINEAR 缩小线性过滤，
            // 线性(使用距离当前渲染像素中心最近的4 个纹素加权平均
值.)
            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                    GLES20.GL_TEXTURE_MIN_FILTER,
                    GLES20.GL_NEAREST);
            // 放大线性过滤,后面的 LINEAR 可以更换为NEAREST 接近滤波
            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);

            // 将 bitmap 传递到已经绑定的纹理中
            GLES20.glTexImage2D(GLES20.GL_TEXTURE_2D, 0,
GLES20.GL_RGBA,
                    t.mWidth, t.mHeight, 0, GLES20.GL_RGBA,
                    GLES20.GL_UNSIGNED_BYTE, t.mData);
        }

        // 通过自定义的工具类创建渲染程序
        shaderProgramID =
SampleUtils.createProgramFromShaderSrc(
                MyShader.CUBE_MESH_VERTEX_SHADER,
                MyShader.CUBE_MESH_FRAGMENT_SHADER);

        // 获取着色器中相应变量的句柄
        vertexHandle =
GLES20.glGetAttribLocation(shaderProgramID,
                "vertexPosition");
        textureCoordHandle =
GLES20.glGetAttribLocation(shaderProgramID,
                "vertexTexCoord");
        mvpMatrixHandle =
GLES20.glGetUniformLocation(shaderProgramID,
                "modelViewProjectionMatrix");
        texSampler2DHandle =
GLES20.glGetUniformLocation(shaderProgramID,
                "texSampler2D");

        if (!mModelIsLoaded) {
            mMyObject = new ModelObject("zhaoyun.obj",
```

```
                    mActivity.getResources().getAssets());

            mModelIsLoaded = true;

            // 隐藏加载对话框
            mActivity.loadingDialogHandler
                    .sendEmptyMessage(LoadingDialogHandler.HIDE_
LOADING_DIALOG);
        }

    }

    /**
     * 渲染器创建时调用，只调用一次，完成 OpenGL ES 的相关初始化工作
     * 接口实现：GLSurfaceView.Renderer
     *
     * @param eglConfig
     */
    @Override
    public void onSurfaceCreated(GL10 gl10, EGLConfig eglConfig)
{
        Log.d(this.getClass().getName(),
"GLRenderer.onSurfaceCreated");

        vuforiaAppSession.onSurfaceCreated();

        mSampleAppRenderer.onSurfaceCreated();
    }

    /**
     * 当 Surface 大小发生变化时调用，例如横纵屏切换
     * 接口实现：GLSurfaceView.Renderer
     *
     * @param width  屏幕宽度
     * @param height 屏幕高度
     */
    @Override
    public void onSurfaceChanged(GL10 gl10, int width, int height)
{
        Log.d(this.getClass().getName(),
"GLRenderer.onSurfaceChanged");

        vuforiaAppSession.onSurfaceChanged(width, height);

        mSampleAppRenderer.onConfigurationChanged(mIsActive);

        initRendering();
    }
```

```
    /**
     * OpenGL ES 绘制方法，绘制的入口
     * 接口实现：GLSurfaceView.Renderer
     */
    @Override
    public void onDrawFrame(GL10 gl10) {
        if (!mIsActive)
            return;

        // 调用绘制工具的渲染方法进行绘制
        mSampleAppRenderer.render();
    }

    /**
     * 需要被每一个渲染器实例实现，SampleAppRendering 每个视图的循环中调用
     * 接口实现：SampleAppRendererControl
     *
     * @param state
     * @param projectionMatrix
     */
    @Override
    public void renderFrame(State state, float[] projectionMatrix)
{
        // 渲染视频背景
        mSampleAppRenderer.renderVideoBackground();

        // 这个画图功能函数也在不断的调用
        // 清除颜色缓冲和深度缓冲
        GLES20.glEnable(GLES20.GL_DEPTH_TEST);

        GLES20.glEnable(GLES20.GL_CULL_FACE);
        GLES20.glCullFace(GLES20.GL_BACK);

        // 通过这里检测是否检测到target
        for (int tIdx = 0; tIdx < state.getNumTrackableResults();
tIdx++) {
            // 以下为追踪到的结果，查看一帧中有几个特定的目标，在
Vuforia 中最多可同时跟踪5 个目标
            TrackableResult result =
state.getTrackableResult(tIdx);
            // trackable 为每个可跟踪的内容，具有名字，id 和类型
            Trackable trackable = result.getTrackable();
            // 打印用户数据
            printUserData(trackable);

            Matrix44F modelViewMatrix_Vuforia
```

```
                    =
Tool.convertPose2GLMatrix(result.getPose());

            // 接下来就获取位置矩阵等，为一个4*4 的矩阵，有四个表示坐
标的行向量
            float[] modelViewMatrix =
modelViewMatrix_Vuforia.getData();

            // 模型视图矩阵和投影矩阵的处理
            float[] modelViewProjection = new float[16];

            // 模型平移
            Matrix.translateM(modelViewMatrix, 0, 0.0f, 0.0f,
1.5f);

            // 模型放大
            Matrix.scaleM(modelViewMatrix, 0, 1.0f, 1.0f, 1.0f);

            // 模型旋转
            Matrix.rotateM(modelViewMatrix, 0, 90.0f, 1.0f, 0.0f,
0.0f);

            // 添加Y 轴旋转动画
            if (mTime == 0)
                mTime = System.currentTimeMillis();

            long curTime = System.currentTimeMillis();
            if (curTime - mTime >= 10000)
                mTime = curTime;
            float angle = (curTime - mTime) % 10000 * (360f / 10000);
            Matrix.rotateM(modelViewMatrix, 0, angle, 0.0f, 1.0f,
0.0f);

            // 两矩阵相乘，将结果置于modelViewProjection 数组中
            // 投影矩阵 *  模型矩阵  => 模型投影矩阵
            Matrix.multiplyMM(modelViewProjection, 0,
                    projectionMatrix, 0,
                    modelViewMatrix, 0);

            // vuforiaAppSession.getProjectionMatrix().getData()
相机位置矩阵
            // 激活着色器程序并绑定 vertex/normal/tex 坐标
            GLES20.glUseProgram(shaderProgramID);

            // 渲染
            GLES20.glVertexAttribPointer(vertexHandle, 3,
GLES20.GL_FLOAT,
                    false, 0, mMyObject.getVertices());
```

```
            GLES20.glVertexAttribPointer(textureCoordHandle, 2,
                    GLES20.GL_FLOAT, false, 0,
mMyObject.getTexCoords());

            GLES20.glEnableVertexAttribArray(vertexHandle);

GLES20.glEnableVertexAttribArray(textureCoordHandle);

            // 激活第 0 个纹理并进行绑定，传递给着色器
            // 选择活动纹理单元。函数原型：
                /*
                void glActiveTexture (int texture)
                参数含义：
                texture 指定哪一个纹理单元被置为活动状态。texture 必
须是 GL_TEXTUREi 之一，
                其中 0 <= i < GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS，
初始值为 GL_TEXTURE0。
                */
            GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
            // 确定了后续的纹理状态改变影响哪个纹理，
            // 纹理单元的数量是依据该纹理单元所被支持的具体实现。
            GLES20.glBindTexture(GLES20.GL_TEXTURE_2D,
                    mTextures.get(0).mTextureID[0]);
            GLES20.glUniform1i(texSampler2DHandle, 0);

            // 将 model-view 矩阵传递给着色器
            GLES20.glUniformMatrix4fv(mvpMatrixHandle, 1, false,
                    modelViewProjection, 0);

            // 绘制 3D 模型
//              GLES20.glDrawElements(GLES20.GL_TRIANGLES,
//                      mTeapot.getNumObjectIndex(),
GLES20.GL_UNSIGNED_SHORT,
//                      mTeapot.getIndices());
            GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0,
                    mMyObject.getNumObjectVertex());

            // 禁用前面使用的数组
            GLES20.glDisableVertexAttribArray(vertexHandle);

GLES20.glDisableVertexAttribArray(textureCoordHandle);

            SampleUtils.checkGLError("Render Frame");

        }

        GLES20.glDisable(GLES20.GL_DEPTH_TEST);
```

```
    }

    private long mTime = 0;

    private void printUserData(Trackable trackable) {
        String userData = (String) trackable.getUserData();
        Log.d(this.getClass().getName(), "UserData:Retreived
User Data \"" + userData + "\"");
    }
}
```

步骤二十 在项目中添加 MyTargets.java 文件，其作为程序主 Activity 类，需在“AndroidManifest.xml”文件中设置 MyTargets 类为主窗口类。

```
package com.hebeiedu.myvuforiademo;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.content.pm.ActivityInfo;
import android.graphics.Color;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.ViewGroup;
import android.view.ViewGroup.LayoutParams;
import android.widget.RelativeLayout;

import com.vuforia.CameraDevice;
import com.vuforia.DataSet;
import com.vuforia.ObjectTracker;
import com.vuforia.STORAGE_TYPE;
import com.vuforia.State;
import com.vuforia.Trackable;
import com.vuforia.Tracker;
import com.vuforia.TrackerManager;
import com.vuforia.Vuforia;

import java.util.ArrayList;
import java.util.Vector;


public class MyTargets extends Activity
        implements SampleApplicationControl {
```

```
    private RelativeLayout mUILayout;
    LoadingDialogHandler loadingDialogHandler = new
LoadingDialogHandler(this);

    // 应用程序会话
    SampleApplicationSession vuforiaAppSession;

    // 目标数据集文件名
    private ArrayList<String> mDatasetStrings = new
ArrayList<String>();
    // 目标数据集
    private DataSet mCurrentDataset;
    // 当前追踪器加载的 目标数据集 在 目标数据集文件名 中的索引
    private int mCurrentDatasetSelectionIndex = 0;
    private int mStartDatasetsIndex = 0;
    private int mDatasetsNumber = 0;

    // 目标数据集快速切换
    private boolean mSwitchDatasetAsap = false;

    // 扩展追踪标志位
    private boolean mExtendedTracking = false;

    // 自定义的OpenGL 视图
    private SampleApplicationGLView mGlView;
    // 自定义的渲染器
    private MyTargetRenderer mRenderer;
    // 将要用来渲染的纹理:
    private Vector<Texture> mTextures;

    // 自动对焦
    private boolean mContAutofocus = false;

    // 用来显示SDK 错误的提示框
    private AlertDialog mErrorDialog;

    // 是否是Android 硬件环境
    boolean mIsDroidDevice = false;


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

        // 创建应用程序会话
        vuforiaAppSession = new SampleApplicationSession(this);

        // 加载动画
```

```
        startLoadingAnimation();

        // 目标数据集文件名
        mDatasetStrings.add("zhaoyun.xml");


        vuforiaAppSession
                .initAR(this,
ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);

        // 加载示例纹理
        mTextures = new Vector<Texture>();
        loadTextures();

        // 检测是否是 android 环境
        mIsDroidDevice =
android.os.Build.MODEL.toLowerCase().startsWith(
                "droid");
    }

    /**
     * 加载动画
     */
    private void startLoadingAnimation() {
        mUILayout = (RelativeLayout) View.inflate(this,
R.layout.loading,
                null);

        mUILayout.setVisibility(View.VISIBLE);
        mUILayout.setBackgroundColor(Color.BLACK);

        loadingDialogHandler.mLoadingDialogContainer = mUILayout
                .findViewById(R.id.loading_indicator);

        loadingDialogHandler
                .sendEmptyMessage(LoadingDialogHandler.SHOW_LOAD
ING_DIALOG);

        addContentView(mUILayout, new
LayoutParams(LayoutParams.MATCH_PARENT,
                LayoutParams.MATCH_PARENT));

    }

    /**
     * 是否扩展了追踪活动
     *
     * @return
     */
```

```
    private boolean isExtendedTrackingActive() {
        return mExtendedTracking;
    }

    /**
     * 初始化应用程序AR功能的相关组件，
     * SurfaceView、Renderer、Texture创建
     */
    private void initApplicationAR() {
        // OpenGL ES View 参数
        // 深度尺寸
        int depthSize = 16;
        // 模型尺寸
        int stencilSize = 0;
        // 是否透明
        boolean translucent = Vuforia.requiresAlpha();

        // OpenGL View实例化及初始化
        mGlView = new SampleApplicationGLView(this);
        mGlView.init(translucent, depthSize, stencilSize);

        // 创建渲染器
        mRenderer = new MyTargetRenderer(this,
vuforiaAppSession);

        // 将整个数据传给渲染器
        mRenderer.setTextures(mTextures);
        // OpenGL View设置显然其
        mGlView.setRenderer(mRenderer);
    }

    /**
     * 从APK中加载纹理资源
     */
    private void loadTextures() {

mTextures.add(Texture.loadTextureFromApk("arcanevoidwraith.png
", getAssets()));
     }

     /**
     * 在对话框中显示初始化的错误信息
     *
     * @param message 错误信息
     */
    public void showInitializationErrorMessage(String message) {
        final String errorMessage = message;
```

```
        runOnUiThread(new Runnable() {

            public void run() {
                if (mErrorDialog != null) {
                    mErrorDialog.dismiss();
                }

                // 创建一个对话框用来显示错误信息
                AlertDialog.Builder builder = new
AlertDialog.Builder(
                        MyTargets.this);
                builder.setMessage(errorMessage)
                        .setTitle(getString(R.string.INIT_ERROR))
                        .setCancelable(false)
                        .setIcon(0)
                        .setPositiveButton(getString(R.string.but
ton_OK),
                                new
DialogInterface.OnClickListener() {
                                    public void
onClick(DialogInterface dialog, int id) {
                                        finish();
                                    }
                                });

                mErrorDialog = builder.create();
                mErrorDialog.show();
            }
        });
    }

    /**
     * 初始化跟踪器
     * 接口实现 : SampleApplicationControl
     * 在 SampleApplicationSession 中的 InitVuforiaTask 类中完成
Vuforia SDK 初始化工作后调用
     *
     * @return true    初始化成功
     * false   初始化失败
     */
    @Override
    public boolean doInitTrackers() {
        // 如果跟踪器被正确初始化
        boolean result = true;

        // 获取跟踪器实例
        TrackerManager tManager = TrackerManager.getInstance();
        Tracker tracker;
```

```
        // 初始化跟踪器
        tracker =
tManager.initTracker(ObjectTracker.getClassType());
        if (tracker == null) {
            Log.e(this.getClass().getName(),
                    "Tracker not initialized. Tracker already
initialized or the camera is already started");
            result = false;
        } else {
            Log.i(this.getClass().getName(), "Tracker
successfully initialized");
        }

        return result;
    }

    /**
     * 加载跟踪器数据
     * 接口实现 : SampleApplicationControl
     * 在SampleApplicationSession 中的LoadTrackerTask 类中完成创建
追踪器工作后调用
     *
     * @return true    初始化成功
     * false   初始化失败
     */

    @Override
    public boolean doLoadTrackersData() {
        // 获取跟踪管理器实例
        TrackerManager tManager = TrackerManager.getInstance();
        // 获取之前创建的跟踪器，ObjectTracker 是 Tracker 的子类
        ObjectTracker objectTracker = (ObjectTracker) tManager
                .getTracker(ObjectTracker.getClassType());

        if (objectTracker == null)
            return false;

        // 创建目标数据集
        if (mCurrentDataset == null)
            mCurrentDataset = objectTracker.createDataSet();

        if (mCurrentDataset == null)
            return false;

        /**
         * mDatasetStrings 为字符串链表，中间保存着多个目标数据集文件
名。
```

```
         * 一个追踪器同一时刻只能追踪一个目标数据集，一个目标数据集中可以有多个目标。
         * STORAGE_APPRESOURCE 表示从应用程序的assets 目录读取目标数据集。
         */
        if (!mCurrentDataset.load(
mDatasetStrings.get(mCurrentDatasetSelectionIndex),
                STORAGE_TYPE.STORAGE_APPRESOURCE))
            return false;

        // 追踪器加载并激活数据集
        if (!objectTracker.activateDataSet(mCurrentDataset))
            return false;

        // 获取数据集中目标对象的个数
        int numTrackables = mCurrentDataset.getNumTrackables();

        for (int count = 0; count < numTrackables; count++) {
            // 从数据集中返回一个可被追踪的对象，Trackable 为所有可被追踪对象的基类
            Trackable trackable =
mCurrentDataset.getTrackable(count);
            // 是否扩展了追踪
            if (isExtendedTrackingActive()) {
                // 开始扩展追踪
                trackable.startExtendedTracking();
            }

            // 设置可被追踪对象的名称，每个可被追踪对象都有一个name、ID、type
            String name = "Current Dataset : " +
trackable.getName();
            trackable.setUserData(name);
            Log.d(this.getClass().getName(), "UserData:Set the
following user data "
                    + (String) trackable.getUserData());
        }

        return true;
    }

    /**
     * 开始跟踪
     * 接口实现 : SampleApplicationControl
     * 在SampleApplicationSession 中的startAR()方法中调用
     * 此时已经初始化完成Vuforia AR 相关设置
```

```
     * 完成OpenGL ES 相关设置
     *
     * @return true    成功
     * false   失败
     */
    @Override
    public boolean doStartTrackers() {
        // 用于标示追踪器启动是否正常
        boolean result = true;

        Tracker objectTracker =
TrackerManager.getInstance().getTracker(
                ObjectTracker.getClassType());
        if (objectTracker != null)
            objectTracker.start();

        return result;
    }

    /**
     * 停止跟踪
     * 接口实现 : SampleApplicationControl
     *
     * @return
     */
    @Override
    public boolean doStopTrackers() {
        boolean result = true;

        Tracker objectTracker =
TrackerManager.getInstance().getTracker(
                ObjectTracker.getClassType());
        if (objectTracker != null)
            objectTracker.stop();

        return result;
    }

    /**
     * 销毁跟踪器数据
     * 这部分与doloadTrackersData 功能相反，用于撤销删除Data 数据
     * 接口实现 : SampleApplicationControl 销毁跟踪器数据
     *
     * @return
     */
    @Override
    public boolean doUnloadTrackersData() {
        boolean result = true;
```

```
        TrackerManager tManager = TrackerManager.getInstance();
        ObjectTracker objectTracker = (ObjectTracker) tManager
                .getTracker(ObjectTracker.getClassType());
        if (objectTracker == null)
            return false;

        if (mCurrentDataset != null && mCurrentDataset.isActive())
{
            if
(objectTracker.getActiveDataSet(0).equals(mCurrentDataset)

&& !objectTracker.deactivateDataSet(mCurrentDataset)) {
                result = false;
            } else if
(!objectTracker.destroyDataSet(mCurrentDataset)) {
                result = false;
            }

            mCurrentDataset = null;
        }

        return result;
    }

    /**
     * 取消跟踪器的初始化
     * 接口实现 : SampleApplicationControl
     *
     * @return
     */
    @Override
    public boolean doDeinitTrackers() {
        boolean result = true;

        TrackerManager tManager = TrackerManager.getInstance();
        tManager.deinitTracker(ObjectTracker.getClassType());

        return result;
    }

    /**
     * Vuforia AR 初始化完成
     * 接口实现 : SampleApplicationControl
     * 在 SampleApplicationSession 中的 initAR() 方法中调用,
     * 表示 Vuforia SDK、追踪器、目标数据集均已加载完成,
     * 是否成功加载可通过参数判断。
     *
```

```
     * @param exception 异常对象，用于检测初始化是否完成
     */
    @Override
    public void onInitARDone(SampleApplicationException
exception) {
        if (exception == null) {
            // 初始化 OpenGL ES 相关
            initApplicationAR();

            // 设置渲染器为激活状态
            mRenderer.setActive(true);

            // 在原来的界面上新建一个布局，此方法必须在相机启动与视频
背景配置之前
            addContentView(mGlView,
                    new
ViewGroup.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT,

ViewGroup.LayoutParams.MATCH_PARENT));

            // 将绘制 Layout 放置在视频背景之前
            mUILayout.bringToFront();

            // 设置绘制 Layout 背景为透明
            mUILayout.setBackgroundColor(Color.TRANSPARENT);

            try {
                // CameraDirection 为调用设备的照相机，
                // DEFAULT 默认，FRONT 前摄像头，BACK 后摄像头
                // 启动 AR 追踪器的追踪

vuforiaAppSession.startAR(CameraDevice.CAMERA_DIRECTION.CAMERA
_DIRECTION_DEFAULT);
            } catch (SampleApplicationException e) {
                Log.e(this.getClass().getName(), e.getString());
            }

            // 设置摄像头的聚焦模式为持续自动(最佳的方式)
            boolean result =
CameraDevice.getInstance().setFocusMode(

CameraDevice.FOCUS_MODE.FOCUS_MODE_CONTINUOUSAUTO);

            if (result)
                mContAutofocus = true;
            else
                Log.e(this.getClass().getName(),
                        "Unable to enable continuous autofocus");
```

```
        } else {
            Log.e(this.getClass().getName(),
exception.getString());

showInitializationErrorMessage(exception.getString());
        }
    }

    @Override
    public void onVuforiaUpdate(State state) {
        // 这个函数会不断的调用，表现为更换识别图时对目标数据集的迅速
更换
        // 切换不同的目标数据集
        if (mSwitchDatasetAsap) {
            mSwitchDatasetAsap = false;

            TrackerManager tm = TrackerManager.getInstance();
            ObjectTracker ot = (ObjectTracker)
tm.getTracker(ObjectTracker
                    .getClassType());
            if (ot == null || mCurrentDataset == null
                    || ot.getActiveDataSet(0) == null) {
                Log.d(this.getClass().getName(), "Failed to swap
datasets");
                return;
            }

            doUnloadTrackersData();
            doLoadTrackersData();
        }
    }

    @Override
    protected void onResume()
    {
        Log.d(this.getClass().getName(), "onResume");
        super.onResume();

        // This is needed for some Droid devices to force portrait
        if (mIsDroidDevice)
        {

setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSC
APE);

setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRA
IT);
        }
```

```
        try
        {
            vuforiaAppSession.resumeAR();
        } catch (SampleApplicationException e)
        {
            Log.e(this.getClass().getName(), e.getString());
        }

        // Resume the GL view:
        if (mGlView != null)
        {
            mGlView.setVisibility(View.VISIBLE);
            mGlView.onResume();
        }

    }

    @Override
    protected void onPause()
    {
        Log.d(this.getClass().getName(), "onPause");
        super.onPause();

        if (mGlView != null)
        {
            mGlView.setVisibility(View.INVISIBLE);
            mGlView.onPause();
        }

        try
        {
            vuforiaAppSession.pauseAR();
        } catch (SampleApplicationException e)
        {
            Log.e(this.getClass().getName(), e.getString());
        }
    }

    @Override
    protected void onDestroy()
    {
        Log.d(this.getClass().getName(), "onDestroy");
        super.onDestroy();

        try
        {
            vuforiaAppSession.stopAR();
        } catch (SampleApplicationException e)
```

```
        {
            Log.e(this.getClass().getName(), e.getString());
        }

        // 释放纹理
        mTextures.clear();
        mTextures = null;

        System.gc();
    }
}
```

8.4 实验结论

当前面步骤都完成之后，本次实验项目就全部编写完成，编译运行后效果如下：

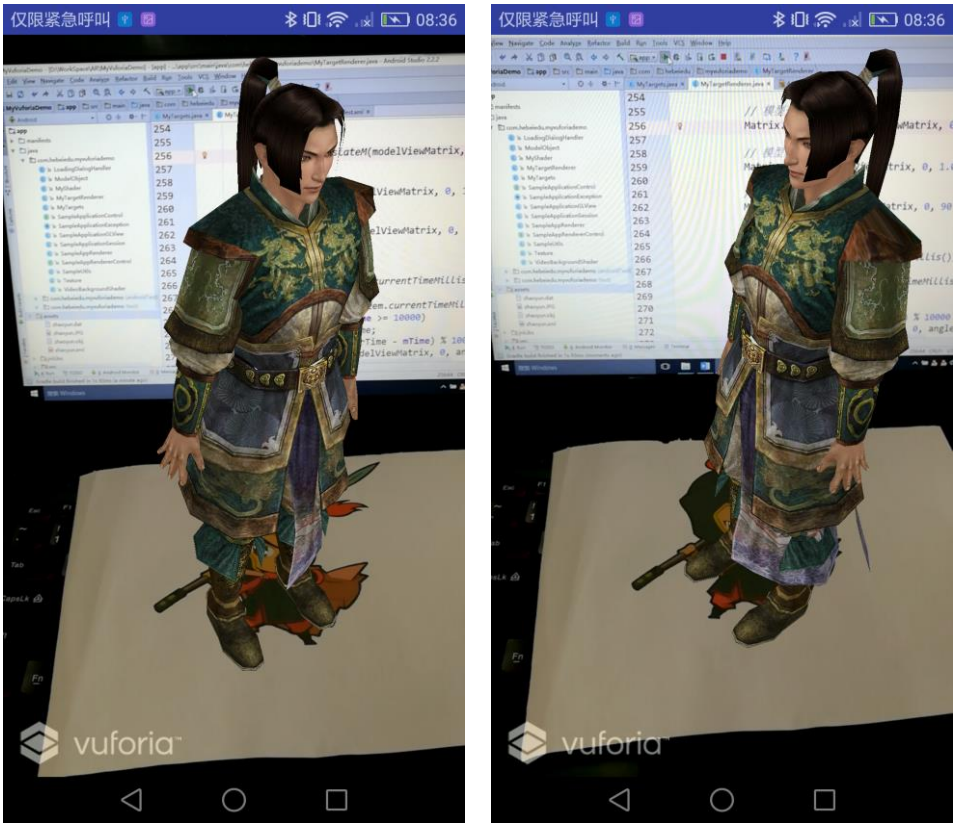

图 8.4.1