SenseMe Effects V5.0.0集成文档

目录

1 集成前的准备工作

1.1 资源文件的使用

- 打开Android Studio的工程项目，将SenseMe_Effects SDK目录中的License文件、Model文件和贴纸素材等文件拷贝到工程项目的assets文件夹下。
- 将license文件需重命名为”SenseME.lic“拷贝至assets文件夹目录下。
- 将“face_attribute_x.x.x.model”、“action_x.x.x.model”等模型文件拷贝至assets文件夹目录下。
- 将贴纸素材包分类为“2D”、“3D”、“hand_action”、“deformation”、“segment”，并在assets建立对应子文件夹，将各个贴纸素材分类并拷贝至各子文件夹，并将同名图标一起放入，缺省图标将显示“无”。分类方式可自定义，详请参考Sample中FileUtils文件。
- 在assets文件夹建立“filter”文件夹，将滤镜model拷贝至此文件夹，以备使用。

1.2 Android Studio环境配置

- 首次打开Sample工程时，可能提示找不到NDK，右键工程->Open Module Settings->SDK Location->Android NDK Location，配置NDK路径。
- 将Sample中JNI部分集成到工程时，在工程build.gradle中添加JNI部分的依赖。

```
dependencies {
    ... ...
    //添加STMobileJNI的依赖
    compile project(':STMobileJNI')
    ... ...
}
```

2 SDK的授权

2.1 使用License文件对算法库授权

- SDK根据License文件检查算法库的使用权限，只有通过了授权，SDK的功能才能够被正常使用。
- 将SenseME.lic授权文件拷贝至assets文件夹后，在集成时用户只需要调用STLicenseUtils工具类的checkLicense()函数即可，流程如下：

1. 首先读取license文件内容
2. 获取本地保存的激活码
3. 如果没有则生成一个激活码
4. 如果有, 则直接调用checkActiveCode*检查激活码
5. 如果检查失败，则重新生成一个activeCode
6. 如果生成失败，则返回失败，成功则保存新的activeCode，并返回成功

```
private final static String PREF_ACTIVATE_CODE_FILE = "activate_code_file";
private final static String PREF_ACTIVATE_CODE = "activate_code";
private static final String LICENSE_NAME = "SenseME.lic";
/**
```

```
* 检查activeCode合法性
* @return true, 成功 false,失败
*/
public static boolean checkLicense(Context context) {
    StringBuilder sb = new StringBuilder();
    InputStreamReader isr = null;
    BufferedReader br = null;
    // 读取license文件内容
    try {
        isr = new
InputStreamReader(context.getResources().getAssets().open(LICENSE_NAME));
        br = new BufferedReader(isr);
        String line = null;
        while((line=br.readLine()) != null) {
            sb.append(line).append("\n");
        }
    }
    ... ...

    // license文件为空,则直接返回
    if (sb.toString().length() == 0) {
        LogUtils.e(TAG, "read license data error");
        return false;
    }

    String licenseBuffer = sb.toString();
    SharedPreferences sp =
context.getApplicationContext().getSharedPreferences(
        PREF_ACTIVATE_CODE_FILE, Context.MODE_PRIVATE);

    String activateCode = sp.getString(PREF_ACTIVATE_CODE, null);
    Integer error = new Integer(-1);
    if (activateCode == null||
(STMobileAuthentificationNative.checkActiveCodeFromBuffer(
        context, licenseBuffer, licenseBuffer.length(), activateCode,
activateCode.length()) != 0)) {
        activateCode =
STMobileAuthentificationNative.generateActiveCodeFromBuffer(
            context, licenseBuffer, licenseBuffer.length());

        if (activateCode != null && activateCode.length() >0) {
            SharedPreferences.Editor editor = sp.edit();
            editor.putString(PREF_ACTIVATE_CODE, activateCode);
            editor.commit();
            return true;
        }
        return false;
    }
    return true;
```

```
}
... ...
}
```

2.2 模型文件的使用

- SDK中的模型文件包括“face_attribute_x.x.x.model”、“action_x.x.x.model”和各种滤镜model，也可根据客户需求申请各种功能的子模型。
- “face_attribute_x.x.x.model”在人脸属性接口创建handle时使用。
- 每个滤镜model对应一种滤镜效果，用户可根据需求拷贝对应model。
- “action_x.x.x.model”在HumanAction接口创建Handle时使用。
- 子模型主要是针对HumanAction接口。例如首先使用“action_x.x.x.model”创建句柄，然后使用addSubModel接口添加子模型，详请参考HumanAction部分。
- 在工程中，需控制model文件拷贝至SDCard并将其路径传给SDK。用户可用过工具类FileUtils的copyModelFiles()函数实现，此部分需要在初始化SDK之前完成。详请参考sample工程中的使用。

```
//拷贝各model文件
 public static void copyModelFiles(Context context) {
    copyFileIfNeed(context, FACE_TRACK_MODEL_NAME);
    copyFileIfNeed(context, FACE_ATTRIBUTE_NAME);
    ... ...
}
//拷贝滤镜模型
public static List<String> copyFilterModelFiles(Context context){
... ...
}
```

- 也可不使用拷贝model的方式，通过AssetsManager直接读取model文件，详请参考Sample对model文件的使用。

2.3 素材文件的使用

- 贴纸素材文件需要控制其拷贝至SDCard，然后获得其路径，在切换素材时将路径传给SDK接口。
- 贴纸素材的拷贝可使用工具类FileUtils的copyStickerZipFiless()函数实现：

```
//拷贝贴纸素材文件，返回值为各素材文件的路径
public static ArrayList<String> copyStickerZipFiles(Context context) {
    String files[] = null;
    ArrayList<String> zipfiles = new ArrayList<String>();

    try {
        files = context.getAssets().list("");
    } catch (IOException e) {
        e.printStackTrace();
    }
    ... ...

    return zipfiles;
}
```

- 使用工具类FileUtils的getStickerFiles()函数获取贴纸素材所在路径，供贴纸渲染使用。
- 各子文件夹的拷贝请参考Sample使用，也可根据具体需求自定义拷贝。

3 单输入模式和双输入模式

3.1 单输入模式

- 单输入模式是指相机只通过回调纹理获取数据。由于HumanAction等接口需要输入buffer数据检测，此模式必须通过glReadPixel()的方式获取ARGB格式buffer数据。
- 优点：不需要回调buffer，集成和数据处理简单。
- 缺点：glReadPixel()低端机器耗时多；图像格式为ARGB，而检测人脸需要灰度图，需要转换。

3.2 双输入模式

- 双输入模式是指相机同时回调纹理texture和数据buffer。
- 优点：相比于单输入模式不需要glReadPixel()，图像格式为NV21，直接可以获取灰度图像。
- 缺点：回调的数据buffer预处理复杂，集成较复杂；在使用ARGB格式接口时需转换格式；线程同步问题。
- 注意：在使用双输入模式时要格外注意线程同步问题和mGlSurfaceView.requestRender()的时机。

3.3 单输入模式和双输入模式切换

- 单输入模式和双输入模式Sample中均有实现。单输入模式参考CameraDisplaySingleInput.java，双输入模式参考CameraDisplayDouble。
- Sample中单双输入切换参考Sample中的CameraActivity.java。

```
//双输入使用
private CameraDisplayDoubleInput mCameraDisplay;

//单输入使用
private CameraDisplaySingleInput mCameraDisplay;

......

//单输入使用
mCameraDisplay = new CameraDisplaySingleInput(getApplicationContext(),
mSingleInputChangePreviewSizeListener, glSurfaceView);

//双输入使用
mCameraDisplay = new CameraDisplayDoubleInput(getApplicationContext(),
mDoubleInputChangePreviewSizeListener, glSurfaceView);
```

4 SDK接口的使用

4.1 SDK句柄的初始化

- humanAction接口的初始化
  1.HumanAction接口初始化需要loadModel，是一个耗时操作，建议异步执行。本版本增加了添加子模型的接口，可在初始化时使用必要的model，然后再添加子模型以减少初始化耗时。

```
//HumanAction句柄初始化是耗时操作，建议异步执行
private void initHumanAction() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (mHumanActionHandleLock) {
                //从sd读取model路径，创建handle
                //int result =
mSTHumanActionNative.createInstance(FileUtils.getTrackModelPath(mContext),
mHumanActionCreateConfig);

                //从asset资源文件夹读取model到内存，再使用底层
st_mobile_human_action_create_from_buffer接口创建handle
                int result =
mSTHumanActionNative.createInstanceFromAssetFile(FileUtils.getActionModelNa
me(), mHumanActionCreateConfig, mContext.getAssets());
                LogUtils.i(TAG, "the result for createInstance for
human_action is %d", result);

                if (result == 0) {
                    mIsCreateHumanActionHandleSucceeded = true;
```

```
                    if(mNeedFaceExtraInfo){
                        result =
mSTHumanActionNative.addSubModelFromAssetFile(FileUtils.getEyeBallCenterMod
elName(), mContext.getAssets());
                        LogUtils.i(TAG, "add eyeball center model result
%d", result);
                        result =
mSTHumanActionNative.addSubModelFromAssetFile(FileUtils.getEyeBallContourMo
delName(), mContext.getAssets());
                        LogUtils.i(TAG, "add eyeball contour model result
%d", result);
                        result =
mSTHumanActionNative.addSubModelFromAssetFile(FileUtils.getFaceExtraModelNa
me(), mContext.getAssets());
                        LogUtils.i(TAG, "add face extra model result %d",
result);
                    }

 mSTHumanActionNative.setParam(STHumanActionParamsType.ST_HUMAN_ACTION_PARA
M_BACKGROUND_BLUR_STRENGTH, 0.35f);
                }
            }
        }
    }).start();
}
```

2. 初始化时需要根据使用场景为图片或视频进行配置，也可以根据具体需求进行修改（详见 STMobileHumanActionNative.java）。

```
//创建时默认的配置参数
// 使用多线程，可最大限度的提高速度，并减少卡顿,根据可根据具体需求修改默认配置
// 对视频进行检测推荐使用多线程检
public final static int ST_MOBILE_HUMAN_ACTION_DEFAULT_CONFIG_VIDEO =
    ST_MOBILE_TRACKING_MULTI_THREAD|
    ST_MOBILE_TRACKING_ENABLE_DEBOUNCE|
    ST_MOBILE_TRACKING_ENABLE_FACE_ACTION|
    ST_MOBILE_ENABLE_FACE_DETECT|
    ST_MOBILE_ENABLE_HAND_DETECT|
    ST_MOBILE_ENABLE_SEGMENT_DETECT|
    ST_MOBILE_DETECT_MODE_VIDEO;

//对图片进行检测默认配置
//对图片进行检测只能使用单线程
public final static int ST_MOBILE_HUMAN_ACTION_DEFAULT_CONFIG_IMAGE =
    ST_MOBILE_TRACKING_SINGLE_THREAD|
    ST_MOBILE_ENABLE_FACE_DETECT|
    ST_MOBILE_ENABLE_SEGMENT_DETECT|
    ST_MOBILE_ENABLE_HAND_DETECT|
    ST_MOBILE_DETECT_MODE_IMAGE;
```

- 人脸属性接口句柄的初始化

```
//face attribute句柄初始化
private void initFaceAttribute() {
    int result =
mSTFaceAttributeNative.createInstance(FileUtils.getFaceAttributeModelPath(m
Context));
    LogUtils.i(TAG, "the result for createInstance for faceAttribute is
%d", result);
}
```

- 美颜接口句柄初始化

```
//初始化美颜handle
private void initBeauty() {
    // 初始化beautify,preview的宽高
    int result = mStBeautifyNative.createInstance(mImageHeight,
mImageWidth);
    LogUtils.i(TAG, "the result is for initBeautify " + result);
    ... ...
}
```

- 贴纸接口句柄的初始化

```
//初始化贴纸handle
private void initSticker() {
    iint result = mStStickerNative.createInstance(mContext, null);

    if(mNeedSticker){
        mStStickerNative.changeSticker(mCurrentSticker);
    }

    setHumanActionDetectConfig(mNeedBeautify|mNeedFaceAttribute,
mStStickerNative.getTriggerAction());
    LogUtils.i(TAG, "the result for createInstance for sticker is %d",
result);
}
```

- 滤镜接口句柄的初始化

```
//初始化滤镜句柄
private void initFilter(){
    mSTMobileStreamFilterNative.createInstance();
    LogUtils.i(TAG, "filter create instance result %d", result);

    mSTMobileStreamFilterNative.setStyle(mCurrentFilterStyle);

    mCurrentFilterStrength = mFilterStrength;
    //设置滤镜强度，缺省值为0.5f

 mSTMobileStreamFilterNative.setParam(STFilterParamsType.ST_FILTER_STRENGTH
, mCurrentFilterStrength);
}
```

- 通用物体追踪接口句柄的初始化

```
//初始化通用物体追踪句柄
private void initObjectTrack(){
    int result = mSTMobileObjectTrackNative.createInstance();
}
```

4.2 相机预览数据和纹理的获取

- 首先要开启系统相机权限，在AndroidManifest.xml文件中添加：

```
<uses-permission android:name="android.permission.CAMERA" />
```

- 双输入模式需设置Android Camera同时使用两种回调方式：buffer和texture。单输入模式只需

要设置回调texture。

```
public void startPreview(SurfaceTexture surfaceTexture, PreviewCallback
previewcallback){
    try {
        //设置为回调texture
        mCamera.setPreviewTexture(surfaceTexture);
        if (previewcallback != null) {
        //设置为回调buffer
            mCamera.setPreviewCallback(previewcallback);
        }
        mCamera.startPreview();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- 从Android Camera获取buffer（详请参考Sample的CameraDisplayDoubleInput.java和CameraProxy.java文件）。

```
private Camera.PreviewCallback mPreviewCallback = new
Camera.PreviewCallback() {
    @Override
    public void onPreviewFrame(final byte[] data, Camera camera) {

        if (mCameraChanging || mCameraProxy.getCamera() == null) {
            return ;
        }
        ... ...
        //更新texture
        mGlSurfaceView.requestRender();
    }
};
```

- 从Android Camera获取texture（详请参考Sample的CameraDisplayDoubleInput.java和CameraProxy.java文件）。

```
private void setUpCamera(){
    // 初始化Camera设备预览需要的显示区域(mSurfaceTexture)
   if(mTextureId == OpenGLUtils.NO_TEXTURE){
       mTextureId = OpenGLUtils.getExternalOESTextureID();
       mSurfaceTexture = new SurfaceTexture(mTextureId);
    }
    ... ...
    //mSurfaceTexture添加相机回调（texture和buffer）
    mCameraProxy.startPreview(mSurfaceTexture,mPreviewCallback);
}
```

4.3 相机预览数据和纹理的预处理

- 相机回调数据的预处理（双输入使用，单输入模式不需要）

1. 在竖直使用手机时（home键在下方），Camera回调数据为横向的NV21数据，需将其先旋转为竖向，再做HumanAction检测。

```
//buffer的旋转
STCommon.stImageRotate(data, mRotateData, mImageHeight, mImageWidth,
STCommon.ST_PIX_FMT_NV21, orientation);
}
```

2. 使用rotate后的相机数据做HumanAction检测，获取人脸、手势和前后背景分割信息。若使用前置摄像头，也需要对HumanAction结果做镜像处理。

```
 STHumanAction humanAction = mSTHumanActionNative.humanActionDetect(
              mNv21ImageData, STCommon.ST_PIX_FMT_NV21, mDetectConfig,
getHumanActionOrientation(), mImageWidth, mImageHeight);

            mHumanActionNoMirrow = humanAction;

            //使用前置摄像头，对HumanAction结果做镜像处理
            if(humanAction != null && mCameraID ==
Camera.CameraInfo.CAMERA_FACING_FRONT){
                humanAction  = mSTHumanActionNative.humanActionMirror(
                                mImageWidth, humanAction);
            }
```

3. 说明:HumanAction暂时不支持rotate，只支持mirror，所以需要对buffer进行rotate处理。需要mirror时使用humanActionMirror接口做镜像处理。对HumanAction做镜像处理比对buffer做镜像处理快得多。

- 纹理的预处理（单输入和双输入处理相同）

1. Camera回调纹理同样为横向，且通常是OES格式纹理，需转换为GL_TEXTURE_2D的纹理格式，然后做旋转和镜像等处理，使得预览图像为正向。
2. 为了方便客户集成和使用，Demo提供了纹理预处理类GLRender的preProcess()函数，此做法效率较高。在使用Camera回调的buffer做HumanAction检测时，则无须使用preProcess()函数中的readPixel，需要将preProcess()函数第二个参数传null。

```
//纹理的预处理
int textureId = mGLRender.preProcess(mTextureId, null);
```

3. 如果您在集成时直接在Camera获取的纹理，请参考Demo对GLRender的preProcess()函数的使用；如果您在集成时已经处理过输入纹理，则不需要再调用preProcess()函数。

4.4 帧处理流程

说明：此部分的美颜、贴纸和滤镜接口需要OpenGL环境，需运行在gl线程中

- HumanAction接口使用

```
//检测人脸关键点，脸部动作，手势和前后背景分割
STHumanAction humanAction =
mSTHumanActionNative.humanActionDetect(mRGBABuffer.array(),
        STCommon.ST_PIX_FMT_RGBA8888,mDetectConfig, orientation,
mImageWidth, mImageHeight);
```

1. humanActionDetect的config配置， 默认全部检测。可根据具体需求自定义配置，比如需要前后背景分割添加ST_MOBILE_SEG_BACKGROUND，以“|”相连接，需要检测手势则添加对应配置。

```
//human action默认配置,
//全部检测,不建议使用,耗时、cpu占用率会变高,建议根据需求检测相关动作
public final static long ST_MOBILE_HUMAN_ACTION_DEFAULT_CONFIG_DETECT =
    ST_MOBILE_FACE_DETECT | ST_MOBILE_EYE_BLINK |
    ST_MOBILE_MOUTH_AH | ST_MOBILE_HEAD_YAW |
    ST_MOBILE_HEAD_PITCH | ST_MOBILE_BROW_JUMP |
    ST_MOBILE_HAND_GOOD | ST_MOBILE_HAND_PALM |
    ST_MOBILE_HAND_LOVE | ST_MOBILE_HAND_HOLDUP |
    ST_MOBILE_HAND_CONGRATULATE | ST_MOBILE_HAND_FINGER_HEART |
    ST_MOBILE_DETECT_EYEBALL_CENTER|ST_MOBILE_DETECT_EYEBALL_CONTOUR;

//支持的人脸行为配置
public final static long ST_MOBILE_FACE_DETECT = 0x00000001;     ///<  人脸检测

//人脸动作
public final static long ST_MOBILE_EYE_BLINK = 0x00000002;     ///<  眨眼
public final static long ST_MOBILE_MOUTH_AH = 0x00000004;     ///<  嘴巴大张
public final static long ST_MOBILE_HEAD_YAW = 0x00000008;     ///<  摇头
public final static long ST_MOBILE_HEAD_PITCH = 0x00000010;     ///<  点头
```

```
public final static long ST_MOBILE_BROW_JUMP = 0x00000020;     ///<  眉毛挑动

//手势动作
public final static long ST_MOBILE_HAND_GOOD = 0x00000800;     ///<  大拇哥
2048
public final static long ST_MOBILE_HAND_PALM = 0x00001000;     ///<  手掌
4096
public final static long ST_MOBILE_HAND_LOVE = 0x00004000;     ///<  爱心
16384
public final static long ST_MOBILE_HAND_HOLDUP = 0x00008000;     ///<  托手
32768
public final static long ST_MOBILE_HAND_CONGRATULATE = 0x00020000;     ///<
恭贺（抱拳） 131072
public final static long ST_MOBILE_HAND_FINGER_HEART = 0x00040000;     ///<
单手比爱心 262144
public final static long ST_MOBILE_HAND_TWO_INDEX_FINGER = 0x00080000;///<
平行手指 524288

public final static long ST_MOBILE_HAND_OK = 0x00000200;  ///< OK手势
public final static long ST_MOBILE_HAND_SCISSOR = 0x00000400;  ///< 剪刀手
public final static long ST_MOBILE_HAND_PISTOL = 0x00002000;  ///< 手枪手势
public final static long ST_MOBILE_HAND_FINGER_INDEX = 0x00100000;  ///< 食
指指尖

public final static long ST_MOBILE_SEG_BACKGROUND = 0x00010000;     ///<  前
景背景分割 65536

public final static long ST_MOBILE_DETECT_EXTRA_FACE_POINTS = 0x01000000;
 ///< 人脸240关键点
public final static long ST_MOBILE_DETECT_EYEBALL_CENTER = 0x02000000;
 ///< 眼球中心点
public final static long ST_MOBILE_DETECT_EYEBALL_CONTOUR = 0x04000000;
 ///< 眼球轮廓点
```

- 人脸属性接口的使用。

1. FaceAttribute接口的输入参数依赖于HumanAction参数的输出，也就是说运行人脸属性之前需要先做HumanAction：

```
STFaceAttribute[] arrayFaceAttribute = new
STFaceAttribute[arrayFaces.length];
//计算人脸属性，包括年龄，性别和颜值等信息
result = mSTFaceAttributeNative.detect(mRGBABuffer.array(),
    STCommon.ST_PIX_FMT_RGBA8888, mImageWidth, mImageHeight, arrayFaces,
arrayFaceAttribute);
```

- 美颜接口的使用。

1. 同样的，美颜的大眼瘦脸等功能也依赖于HumanAction：

```
//美颜处理
result = mStBeautifyNative.processTexture(textureId,
    mImageWidth, mImageHeight, arrayFaces, mBeautifyTextureId[0],
arrayOutFaces);
```

2. 美颜参数设置：

```
//设置美颜参数
//红润强度, [0,1.0], 0.0不做红润
mStBeautifyNative.setParam(STBeautyParamsType.ST_BEAUTIFY_REDDEN_STRENGTH,
0.36f);
//磨皮强度, [0,1.0], 0.0不做磨皮
mStBeautifyNative.setParam(STBeautyParamsType.ST_BEAUTIFY_SMOOTH_STRENGTH,
0.74f);
//美白强度, [0,1.0], 0.0不做美白
mStBeautifyNative.setParam(STBeautyParamsType.ST_BEAUTIFY_WHITEN_STRENGTH,
0.30f);
//大眼比例, [0,1.0], 0.0不做大眼效果
mStBeautifyNative.setParam(STBeautyParamsType.ST_BEAUTIFY_ENLARGE_EYE_RATIO
, 0.13f);
//瘦脸比例, [0,1.0], 0.0不做瘦脸效果
mStBeautifyNative.setParam(STBeautyParamsType.ST_BEAUTIFY_SHRINK_FACE_RATIO
, 0.11f);
//小脸比例, [0,1.0], 0.0不做小脸效果
mStBeautifyNative.setParam(STBeautyParamsType.ST_BEAUTIFY_SHRINK_JAW_RATIO,
0.10f);
```

- 贴纸接口的使用。

1. 贴纸功能也需要人脸信息，需先做HumanAction：

```
boolean needOutputBuffer = false; //如果需要输出buffer推流或其他，设置该开关为
true
long stickerStartTime = System.currentTimeMillis();
if (!needOutputBuffer) {
    result = mStStickerNative.processTexture(textureId, humanAction,
            orientation, mImageWidth, mImageHeight,false,
mTextureOutId[0]);
} else {  //如果需要输出buffer用作推流等
    byte[] imageOut = new byte[mImageWidth * mImageHeight * 4];
    result = mStStickerNative.processTextureAndOutputBuffer(textureId,
humanAction, orientation,
            mImageWidth, mImageHeight, false, mTextureOutId[0],
            STCommon.ST_PIX_FMT_RGBA8888, imageOut);
}
```

2. 在切换贴纸时，调用STMobileStickerNative的changeSticker函数，传入贴纸路径(参考setShowSticker函数的使用)。

```
//切换贴纸，参数输入为贴纸素材路径
public void setShowSticker(String sticker) {
    mCurrentSticker = sticker;
    mStStickerNative.changeSticker(mCurrentSticker);
    setHumanActionDetectConfig(mNeedBeautify|mNeedFaceAttribute,
            mStStickerNative.getTriggerAction());
}
```

3. 切换贴纸后，使用STMobileStickerNative的getTriggerAction函数获取当前贴纸支持的手势和前后背景等信息，返回值为long类型。

```
public long getStickerTriggerAction(){
    return mStStickerNative.getTriggerAction();
}
```

4. 根据getTriggerAction函数返回值，重新配置humanActionDetect函数的config参数，使HumanAction检测函数更高效。例：只检测人脸信息和当前贴纸支持的手势等信息时，使用如下配置：

```
mDetectConfig = mSTMobileStickerNative.getTriggerAction()
            |STMobileHumanActionNative.ST_MOBILE_FACE_DETECT;
```

5.贴纸部分增加了声音功能，可设计贴纸自动播放声音或通过TriggerAction，上层通过JNI部分的STSoundPlay.java实现。可参考Sample中STSoundPlay使用，根据具体需求用户可自定义声音播放控制。

- 滤镜接口的使用：

```
//设置滤镜风格，必须在OpenGL线程中调用
mSTMobileStreamFilterNative.setStyle(mCurrentFilterStyle);
//设置滤镜特效强度，范围[0,1],设置为0无滤镜效果，设置为1效果最强
mSTMobileStreamFilterNative.setParam(STFilterParamsType.ST_FILTER_STRENGTH,
            mCurrentFilterStrength);
//滤镜处理函数
int ret = mSTMobileStreamFilterNative.processTexture(
            textureId, mImageWidth, mImageHeight, mFilterTextureOutId[0]);
```

- 通用物体追踪接口的使用：

```
//设置跟踪目标的矩形
if (mNeedSetObjectTarget) {
    mSTMobileObjectTrackNative.setTarget(mRGBABuffer.array(),
            STCommon.ST_PIX_FMT_RGBA8888, mImageWidth, mImageHeight,
tragetRect);
}
... ...

//对连续视频帧中的目标进行实时快速跟踪,返回目标区域rect和置信度（目前只支持一个目标）
float[] socre = new float[1];
STRect outputRect = mSTMobileObjectTrackNative.objectTrack(
            mRGBABuffer.array(), STCommon.ST_PIX_FMT_RGBA8888, mImageWidth,
            mImageHeight, socre);
... ...

//重置通用物体跟踪句柄
mSTMobileObjectTrackNative.reset();
```

4.5 SDK句柄的释放

- 1.GL资源必须在GL线程释放

```
mGlSurfaceView.queueEvent(new Runnable() {
    @Override
    public void run() {
        //释放贴纸句柄
        mStStickerNative.destroyInstance();
        //释放美颜句柄
        mStBeautifyNative.destroyBeautify();
        //释放滤镜句柄
        mSTMobileStreamFilterNative.destroyInstance();

        mRGBABuffer = null;
        deleteTextures();
        if(mSurfaceTexture != null){
            mSurfaceTexture.release();
        }
        mGLRender.destroyFrameBuffers();
    }
});
```

- 2.非GL资源的释放

```
synchronized (mHumanActionHandleLock){
    //释放HumanAction句柄
    mSTHumanActionNative.destroyInstance();
}
//释放人脸属性句柄
mSTFaceAttributeNative.destroyInstance();

//释放通用物体追踪句柄
mSTMobileObjectTrackNative.destroyInstance();
```

5 JNI部分接口说明

5.1 STMobileCommon

其中定义了一些静态常量，如图像格式和错误码等，还有一些静态函数，详见STMobileCommon.java。

```
//进行颜色格式转换，不建议使用关于YUV420P的转换,速度较慢
public static native int stColorConvert(byte[] inputImage, byte[]
outputImage,
                int width, int height, int type);

//进行图片旋转
public static native int stImageRotate(byte[] inputImage, byte[]
outputImage,
                int width, int height, int format, int rotation);

//设置眨眼动作的阈值,置信度为[0,1]，默认阈值为0.5
public native void setEyeblinkThreshold(float threshold);

//设置张嘴动作的阈值,置信度为[0,1]，默认阈值为0.5
public native void setMouthahThreshold(float threshold);

//设置张嘴动作的阈值,置信度为[0,1]，默认阈值为0.5
public native void setHeadyawThreshold(float threshold);

//设置张嘴动作的阈值,置信度为[0,1]，默认阈值为0.5
public native void setHeadpitchThreshold(float threshold);

//设置张嘴动作的阈值,置信度为[0,1]，默认阈值为0.5
public native void setBrowjumpThreshold(float threshold);

//设置人脸106点平滑的阈值. 若不设置，使用默认值. 默认值0.8，//建议取值范围：[0.0,
1.0]. 阈值越大，去抖动效果越好，跟踪延时越大
public native void setSmoothThreshold(float threshold);

//设置人脸三维旋转角度去抖动的阈值. 若不设置，使用默认值. 默认值0.5，建议取值范围:
[0.0, 1.0]. 阈值越大，去抖动效果越好，跟踪延时越大
public native void setHeadposeThreshold(float threshold);
```

5.2 STMobileHumanActionNative

定义了HumanAction检测接口，create和detect时可能用到的config选项。详见STMobileHumanActionNative.java。

```
//对视频进行检测默认配置
public final static int ST_MOBILE_HUMAN_ACTION_DEFAULT_CONFIG_VIDEO;
//对图片进行检测默认配置
public final static int ST_MOBILE_HUMAN_ACTION_DEFAULT_CONFIG_IMAGE;

//创建实例
public native int createInstance(String modelpath, int config);

//从资源文件夹创建实例
public native int createInstanceFromAssetFile(String assetModelpath, int
config, AssetManager assetManager);

//通过子模型创建人体行为检测句柄，st_mobile_human_action_create和
st_mobile_human_action_create_with_sub_models只能调一个
public native int createInstanceWithSubModels(String[] modelPaths, int
modelCount, int config);

//添加子模型．非线程安全，不支持在执行st_mobile_human_action_detect的过程中覆盖正在
使用的子模型
public native int addSubModel(String modelPath);

//从资源文件夹添加子模型．非线程安全，不支持在执行st_mobile_human_action_detect的过
程中添加子模型
public native int addSubModelFromAssetFile(String assetModelpath,
AssetManager assetManager);

//要设置Human Action参数的类型
public native int setParam(int type, float value);

//检测人脸关键点信息、actionInfo和前后背景分割等
public native STHumanAction humanActionDetect(byte[] imgData, int
imageFormat,
            long detect_config,int orientation, int imageWidth, int
imageHeight);

//释放资源
public native void destroyInstance();
```

5.3 STMobileFaceAttributeNative

人脸属性接口，详见STMobileFaceAttributeNative.java。

```
//创建实例
public native int createInstance(String modelpath);
//人脸属性检测
public native int detect(byte[] image, int format, int width,
            int height, STMobile106[] mobile106, STFaceAttribute[]
attributes);
//释放资源
public native void destroyInstance();
```

5.4 STBeautifyNative

美颜处理接口，用户可根据需求选择接口使用，详见STBeautifyNative.java。

```
//创建美颜句柄
public native int createInstance(int width, int height);

//美颜参数设置
public native int setParam(int type, float value);

//对图像buffer做美颜处理，需要在OpengGL环境中调用（运行在OpenGL线程中）
public native int processBufferInGLContext(byte[] pInputImage, int
inFormat,
            int outputWidth, int outputHeight, STMobile106[] arrayFacesIn,
            byte[] pOutImage, int outFormat,
    STMobile106[] arrayFacesOut);

//对图像做美颜处理，此接口针对不在OpenGL环境中(不在opengl线程中)执行函数的用户
public native int processBufferNotInGLContext(byte[] pInputImage, int
inFormat,
            int outputWidth, int outputHeight, STMobile106[] arrayFacesIn,
byte[] pOutImage,
            int outFormat, STMobile106[] arrayFacesOut);

//对OpenGL ES中的纹理进行美颜处理，需要运行在Opengl环境中
public native int processTexture(int textureIn, int outputWidth,
            int outputHeight, STMobile106[] arrayFacesIn,
            int textureOut, STMobile106[] arrayFacesOut);

//对OpenGL ES中的纹理进行美颜处理，需要运行在Opengl环境中.可以输出buffer及人脸信息
public native int processTextureAndOutputTexture(int textureIn, int
outputWidth,
            int outputHeight, STMobile106[] arrayFacesIn, int textureOut,
byte[] outputBuf,
            int format, STMobile106[] arrayFacesOut);

//释放instance，必须在opengl环境中运行
public native void destroyBeautify();
```

5.5 STMobileStickerNative

贴纸处理接口，详见STMobileStickerNative.java。

```
//创建贴纸句柄
public native int createInstance(Context context, String zipPath);

//对OpenGL ES 中的纹理进行贴纸处理，必须在opengl环境中运行，仅支持RGBA图像格式
public native int processTexture(int textureIn, STHumanAction humanAction,
            int rotate, int imageWidth, int imageHeight, boolean
needsMirroring,
            int textureOut);

//对OpenGL ES 中的纹理进行贴纸处理，必须在opengl环境中运行，仅支持RGBA图像格式.支持
buffer输出
public native int processTextureAndOutputBuffer(int textureIn,
STHumanAction humanAction,
            int rotate, int imageWidth, int imageHeight, boolean
needsMirroring,
            int textureOut, int outFmt, byte[] imageOut);

//切换贴纸路径
public native int changeSticker(String path);

//获取当前贴纸的触发动作
public native long getTriggerAction();

//等待素材加载完毕后再渲染，用于希望等待模型加载完毕再渲染的场景，比如单帧或较短视频的3D绘
制等
public native int setWaitingMaterialLoaded(boolean needWait);

//设置贴纸素材图像所占用的最大内存
public native int setMaxMemory(int value);

//通知声音停止函数
public native int setSoundPlayDone(String name);

//销毁实例，必须在opengl环境中运行
public native void destroyInstance();
```

5.6 STSoundPlay

STSoundPlay为声音贴纸的实现设计。

```
//音频播放监听器
public interface PlayControlListener {
    //加载音频素材callback
    void onSoundLoaded(String name, byte[] content);

    //播放音频callback
    void onStartPlay(String name, int loop);

    //停止播放callback
    void onStopPlay(String name);
}

public void setStickHandle(STMobileStickerNative stickHandle);

//设置音频播放完成标志
public void setSoundPlayDone(String name);

//JNI调用，不做混淆
private void onSoundLoaded(String name, byte[] content);

//JNI调用，不做混淆
private void onStartPlay(String name, int loop);

//JNI调用，不做混淆
private void onStopPlay(String name);
......
```

5.7 STMobileStreamFilterNative

gpu滤镜接口，需运行在gl线程，详见STMobileStreamFilterNative.java。

```
//创建实例
public native int createInstance();

//设置滤镜风格，必须在OpenGL线程中调用
public native int setStyle(String styleModelPath);

//设置滤镜参数
public native int setParam(int type, float value);

//对 OpenGL ES 中的纹理进行滤镜处理，必须在OpenGL线程中调用
public native int processTexture(int textureIn,
            int imageWidth, int imageHeight, int textureOut);

//对OpenGL ES 中的纹理进行滤镜处理，并输出buffer，必须在OpenGL线程中调用
public native int processTextureAndOutputBuffer(int textureIn, int
imageWidth,
            int imageHeight,int textureOut, byte[] outImage, int
outFormat);

//对图像buffer做滤镜处理，必须在OpenGL线程中调用
public native int processBuffer(byte[] inputImage, int inFormat, int
imageWidth,
            int imageHeight, byte[] outImage, int outFormat);

//释放滤镜句柄，必须在OpenGL线程中调用
public native void destroyInstance();
```

5.8 STMobileFilterNative

cpu滤镜接口，详见STMobileFilterNative.java。

```
//创建滤镜句柄
public native int createInstance();

//设置滤镜风格
public native int setStyle(String styleModelPath);

//设置滤镜参数
public native int setParam(int type, float value);

//对图像做滤镜处理
public native int process(byte[] inputImage, int inFormat,
            int imageWidth, int imageHeight, byte[] outImage, int
outFormat);

//释放滤镜句柄
public native void destroyInstance();
```

5.9 STMobileObjectTrackNative

通用物体追踪接口，详见STMobileObjectTrackNative.java。

```
//创建通用物体跟踪句柄
public native int createInstance();

//设置跟踪目标的矩形
public native int setTarget(byte[] inputImage, int inFormat,
            int imageWidth, int imageHeight, STRect rect);

//对连续视频帧中的目标进行实时快速跟踪,返回目标区域rect和置信度
public native STRect objectTrack(byte[] inputImage, int inFormat,
            int imageWidth,int imageHeight, float[] resultScore);

//重置通用物体跟踪句柄
public native void reset();

//销毁实例
public native void destroyInstance();
```

6 旋转和方向相关说明

6.1 设备和相机的方向

- 手机的方向可根据传感器获取：

```
//获取设备的方向
private int getCurrentOrientation() {
    int dir = Accelerometer.getDirection();
    int orientation = dir - 1;
    if (orientation < 0) {
        orientation = dir ^ 3;
    }

    return orientation;
}
```

- 相机的方向

1. 前置摄像头：绝大部分手机的前置摄像头的CameraInfo.orientation为270，特殊机型为90。
2. 后置摄像头：绝大部分手机的后置摄像头的CameraInfo.orientation为90，特殊机型为270。

```
private CameraInfo mCameraInfo = new CameraInfo();
//获取相机的方向
public int getOrientation(){
    if(mCameraInfo == null){
        return 0;
    }
    return mCameraInfo.orientation;
}
```

6.2 相机预览数据和纹理的旋转方向

- 相机预览数据的旋转方向

1. 在使用STCommon.stImageRotate接口旋转buffer时使用。

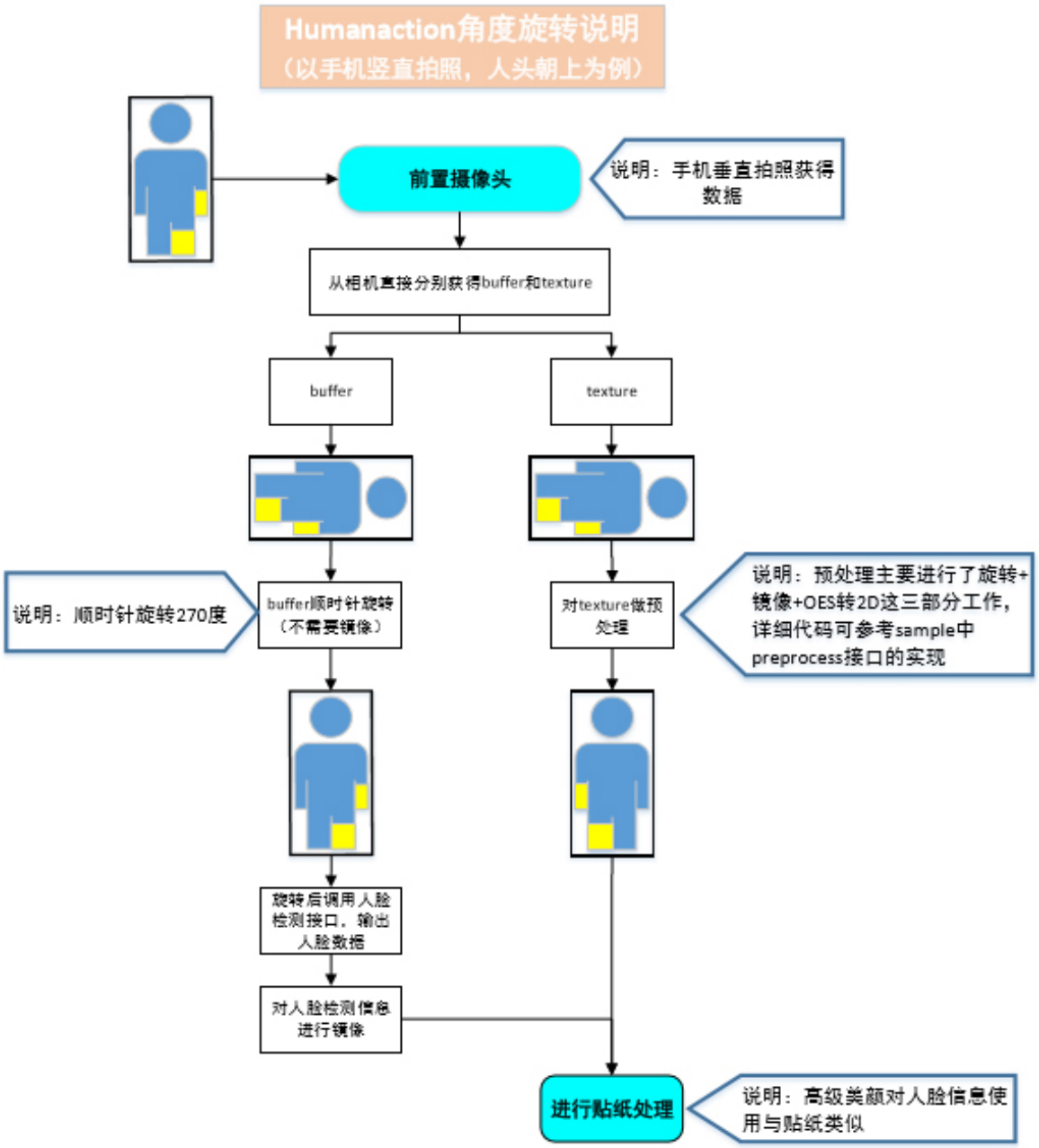
```
private int getRotateOrientation(){
    //相机预览buffer的旋转角度。由于Camera获取的buffer为横向图像，将buffer旋转为竖向（即正向竖屏使用手机时，人脸方向朝上）
    int rotateOrientation = STRotateType.ST_CLOCKWISE_ROTATE_270;

    if(mCameraID == Camera.CameraInfo.CAMERA_FACING_FRONT){
        rotateOrientation  = STRotateType.ST_CLOCKWISE_ROTATE_270;
    }else if(mCameraID == Camera.CameraInfo.CAMERA_FACING_BACK && mCameraProxy.getOrientation() == 90){
        rotateOrientation  = STRotateType.ST_CLOCKWISE_ROTATE_90;
    }else if(mCameraID == Camera.CameraInfo.CAMERA_FACING_BACK && mCameraProxy.getOrientation() == 270){
        rotateOrientation  = STRotateType.ST_CLOCKWISE_ROTATE_270;
    }

    return rotateOrientation;
}
```

- 纹理的旋转

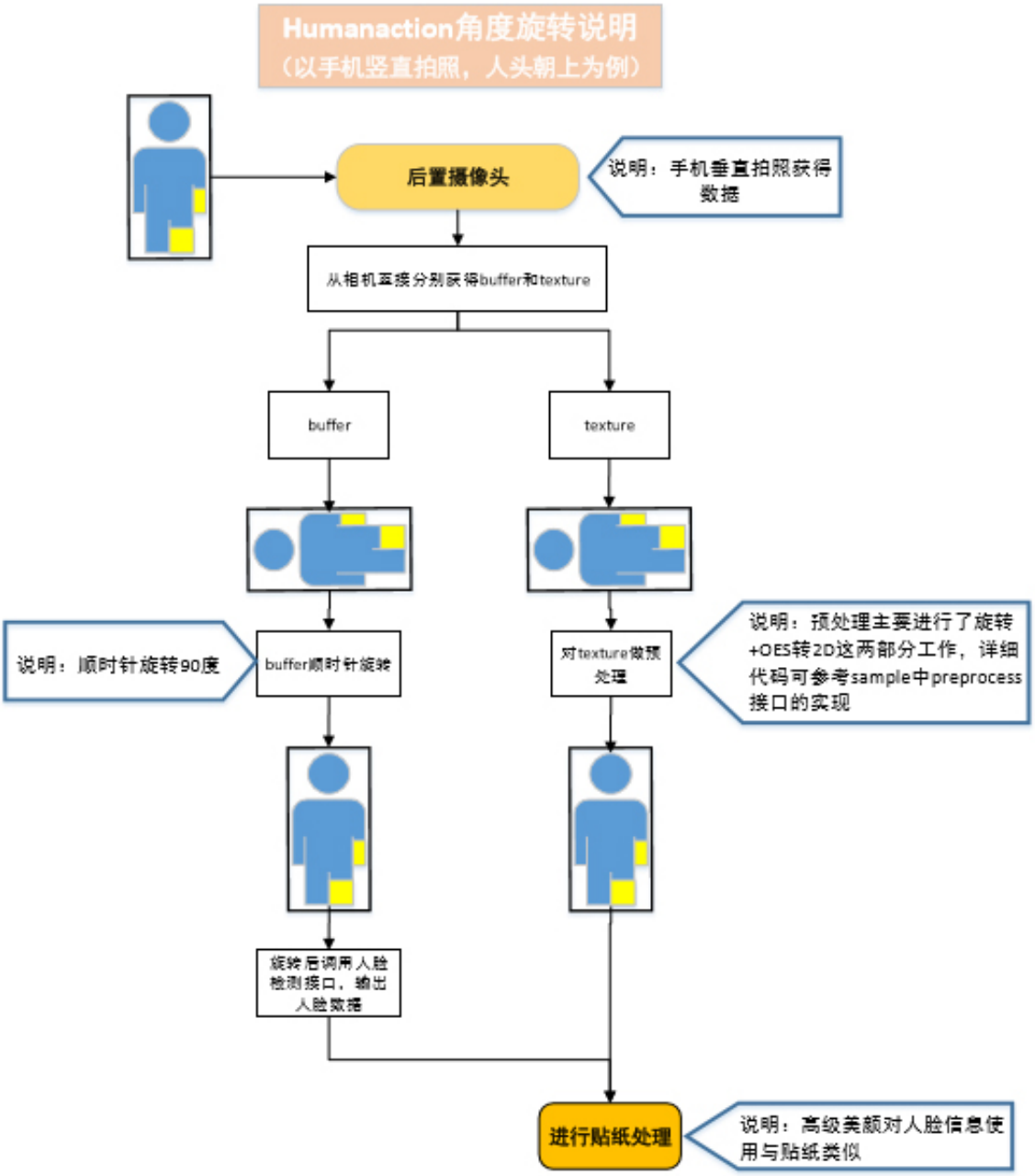
1. 纹理预处理时的旋转方向同buffer相同。当使用前置摄像头时增加了mirror处理。

- 相机预览数据和纹理的处理流程

1. 前置摄像头。如下图所示：



2. 后置摄像头。如下图所示：



6.3 HumanAction检测方向

```
private int getHumanActionOrientation(){
    //用于humanActionDetect接口。根据传感器方向计算出在不同设备朝向时，
      人脸在图片中的朝向。
    int humanOrientation = getCurrentOrientation();

    if(mCameraID == Camera.CameraInfo.CAMERA_FACING_FRONT){
        //前置摄像头nv21 buffer未做mirror，当设备朝向为90或270时，人脸朝向与设备方向
相反，做humanActionDetect时需输入相反方向
        if(humanOrientation == STRotateType.ST_CLOCKWISE_ROTATE_90){
            humanOrientation = STRotateType.ST_CLOCKWISE_ROTATE_270;
        }else if(humanOrientation == STRotateType.ST_CLOCKWISE_ROTATE_270){
            humanOrientation = STRotateType.ST_CLOCKWISE_ROTATE_90;
        }
    }

    return humanOrientation;
}
```