SenseMe Effects 集成文档

目录

1 SDK简介

1.1 SDK功能简介

- SDK主要检测功能包括：人脸106关键点、人脸240（282）点关键点、背景分割、手势检测等
- SDK主要特效功能包括：美颜、微整形、美妆、贴纸和滤镜等

1.2 SDK目录简介

- SDK打包文件主要包括sample、aar文件、STMobileJNI及源码、libst_mobile.so和头文件等
- sample作为特效展示和集成时的参考示例
- aar文件和STMobileJNI，提供SDK核心能力
- 使用c接口文件集成时参考libst_mobile.so和头文件（本文档仅限于使用java接口集成参考）

2 项目中导入SDK

2.1 使用aar文件

- 打开SenseMe_Effects SDK目录，找到其中的STMobileJNI-xxx-release.aar文件
- 拷贝其到项目中的主模块或集成模块的libs目录下，如拷贝到app/libs/目录下，没有libs文件夹可手动创建
- 打开主模块或集成模块的build.gradle文件，在android{…}中加入查询路径：

```
repositories {
        flatDir {
            dirs 'libs'
        }
}
```

- 在主模块或集成模块的build.gradle的 dependencies 下加入 SDK 引用：

```
implementation(name:'STMobileJNI-xxx-release',ext:'aar')
```

- 将模型文件和素材文件拷贝到项目主模块或集成模块的assets文件目录备用
- 将授权文件License.lic拷贝到项目主模块或集成模块的assets文件目录备用

2.2 使用源码依赖

- 打开SenseMe_Effects SDK目录，找到其中的sample/STMobileJNI文件目录
- 拷贝其到项目目录下，打开项目根目录下的setting.gradle文件，添加module：

```
include ':app', ':STMobileJNI'
```

- 在主模块或集成模块的build.gradle的 dependencies 下加入 SDK 引用：

```
//添加STMobileJNI的依赖
implementation project(':STMobileJNI')
```

- 将模型文件和素材文件拷贝到项目主模块或集成模块的assets文件目录备用
- 将授权文件License.lic拷贝到项目主模块或集成模块的assets文件目录备用

2.3 SDK混淆

- 使用aar或源码依赖，需在项目主模块或集成模块的“proguard-rules.pro”文件中，添加SDK的混淆：

```
-keep class com.sensetime.stmobile.* { *;}
-keep class com.sensetime.stmobile.model.* { *;}
```

3 SDK的授权

- 使用License文件授权之后，才能正常使用SDK。可以使用离线授权文件，也可使用在线服务器托管的授权文件

3.1 离线license授权

- 将授权文件License.lic拷贝到项目主模块或集成模块的assets目录或指定文件目录
- 将Sample中的STLicenseUtils类拷贝到项目中
- 打开STLicenseUtils.java文件，修改为离线授权：

```
//是否使用服务器License鉴权
//true：使用服务器下拉授权文件，使用离线接口生成activeCode
//false：使用asset文件夹下的 "SenseME.lic", "SenseME_Online.lic"生成activeCode
private static final boolean USING_SERVER_LICENSE = false;
```

- 根据需要修改license文件路径或名字：

```
//离线generateActiveCode使用
private static final String LOCAL_LICENSE_NAME = "license/SenseME.lic";
```

- 调用静态方法checkLicense函数授权：

```
//鉴权方式有两种，离线和在线
public static boolean checkLicense(final Context context){
    if(USING_SERVER_LICENSE){
        return checkLicenseFromServer(context);
    }else{
        return checkLicenseFromLocal(context);
    }
}
```

3.2 在线license授权

- 将Sample中的STLicenseUtils类拷贝到项目中
- 打开STLicenseUtils.java文件，修改为在线授权：

```
//是否使用服务器License鉴权
//true：使用服务器下拉授权文件，使用离线接口生成activeCode
//false：使用asset文件夹下的 "SenseME.lic", "SenseME_Online.lic"生成activeCode
private static final boolean USING_SERVER_LICENSE = true;
```

- 调用静态方法checkLicense函数授权：

```
//鉴权方式有两种，离线和在线
public static boolean checkLicense(final Context context){
    if(USING_SERVER_LICENSE){
        return checkLicenseFromServer(context);
    }else{
        return checkLicenseFromLocal(context);
    }
}
```

4 集成准备工作

4.1 模型文件的使用

- 模型在算法检测时使用，包括人脸检测模型、背景分割模型等，添加不同模型配合license提供不同的检测能力
- 将模型拷贝到工程assets目录或其中的指定目录，使用模型路径和名字作为索引，sample中参考FileUtils.java，以assets根目录为例：

```
public static final String MODEL_NAME_ACTION =
"M_SenseME_Face_Video_7.0.0.model";
public static final String MODEL_NAME_FACE_ATTRIBUTE =
"M_SenseME_Attribute_1.0.1.model";
public static final String MODEL_NAME_EYEBALL_CENTER =
"M_Eyeball_Center.model";
public static final String MODEL_NAME_EYEBALL_CONTOUR =
"M_SenseME_Iris_2.0.0.model";
public static final String MODEL_NAME_FACE_EXTRA =
"M_SenseME_Face_Extra_Advanced_6.0.11.model";
public static final String MODEL_NAME_HAND = "M_SenseME_Hand_5.4.0.model";
public static final String MODEL_NAME_SEGMENT =
"M_SenseME_Segment_4.10.8.model";
public static final String MODEL_NAME_BODY_FOURTEEN =
"M_SenseME_Body_Fourteen_1.2.0.model";
public static final String MODEL_NAME_AVATAR_CORE =
"M_SenseME_Avatar_Core_2.0.0.model";
public static final String MODEL_NAME_CATFACE_CORE =
"M_SenseME_CatFace_3.0.0.model";
public static final String MODEL_NAME_AVATAR_HELP =
"M_SenseME_Avatar_Help_2.2.0.model";
public static final String MODEL_NAME_TONGUE =
"M_Align_DeepFace_Tongue_1.0.0.model";
public static final String MODEL_NAME_HAIR =
"M_SenseME_Segment_Hair_1.3.4.model";
public static final String MODEL_NAME_LIPS_PARSING =
"M_SenseAR_Segment_MouthOcclusion_FastV1_1.1.2.model";
public static final String HEAD_SEGMENT_MODEL_NAME =
"M_SenseME_Segment_Head_1.0.3.model";
```

- SDK支持从assets文件中加载模型和从sd卡路径加载模型。以从assets加载为例：

```
//创建检测句柄
int result =
mSTHumanActionNative.createInstanceFromAssetFile(FileUtils.getActionModelName()
, mHumanActionCreateConfig, mContext.getAssets());
```

4.2 素材文件的使用

- 素材文件包括贴纸素材、美妆素材和滤镜素材
- 将需要的素材文拷贝到件工程assets目录或其中的指定目录，初始化是将其拷贝到sd卡。特效接口需传入素材的真实路径。例：

```
//切换贴纸
int packageId = mSTMobileEffectNative.changePackage(currentStickerFilePath);

//切换美妆素材
mSTMobileEffectNative.setBeauty(MAKEUP_TYPE, currentMakeupFilePath);
```

5 代码中接入检测

- 检测使用STMobileHumanActionNative中的接口

5.1 检测初始化

- 检测句柄初始化可配置预览模型、图片模式和视频模式

```
STMobileHumanActionNative.ST_MOBILE_HUMAN_ACTION_DEFAULT_CONFIG_VIDEO//预览模式
STMobileHumanActionNative.ST_MOBILE_HUMAN_ACTION_DEFAULT_CONFIG_IMAGE//图片模式
STMobileHumanActionNative.ST_MOBILE_HUMAN_ACTION_DEFAULT_CONFIG_VIDEO_SINGLE_TH
READ//视频模式
```

- 检测接口初始化需要加载所需模型，是一个耗时操作，建议异步执行

```
private void initHumanAction() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (mHumanActionHandleLock) {
                //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.getActionModelName()
, mHumanActionCreateConfig, mContext.getAssets());
                LogUtils.i(TAG, "the result for createInstance for human_action
is %d", result);
            }
        }
    }).start();
}
```

- 可按需要添加子模型，减少初始化耗时，以添加240关键点检测模型为例：

```
//添加240关键点检测
result =
mSTHumanActionNative.addSubModelFromAssetFile(FileUtils.MODEL_NAME_FACE_EXTRA,
mContext.getAssets());
LogUtils.i(TAG, "add face extra model result: %d", result);
```

5.2 配置检测选项

- 检测接口config参数，可手动配置，也可以配合特效接口配置
- 手动配置：

```
//配置人脸检测(106)
mDetectConfig = STMobileHumanActionNative.ST_MOBILE_FACE_DETECT;

//添加人脸240(282)检测
mDetectConfig |= STMobileHumanActionNative.ST_MOBILE_DETECT_EXTRA_FACE_POINTS;
```

- 配合特效接口，更新检测config：

```
/**
 * human action detect的配置选项,根据渲染接口需要配置
 */
public void updateHumanActionDetectConfig() {
    mDetectConfig = mSTMobileEffectNative.getHumanActionDetectConfig();
}
```

- 检测config定义如下，参见STMobileHumanActionNative.java文件

```
//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
public final static long ST_MOBILE_FACE_LIPS_UPWARD = 0x00000040;  ///< 嘴角上扬
public final static long ST_MOBILE_FACE_LIPS_POUTED = 0x00000080;  ///< 嘟嘴

//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_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_HAND_FIST = 0x00200000;  ///< 拳头手势
public final static long ST_MOBILE_HAND_666 = 0x00400000;  ///< 666
public final static long ST_MOBILE_HAND_BLESS = 0x00800000;  ///< 双手合十
public final static long ST_MOBILE_HAND_ILOVEYOU = 0x010000000000L;  ///< 手势ILoveYou

public final static long ST_MOBILE_BODY_KEYPOINTS = 0x08000000;  ///< 检测人体关键点
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

public final static long ST_MOBILE_SEG_MULTI = 0x800000000000L;  ///< 检测多类分割
public final static long ST_MOBILE_DETECT_GAZE = 0x100000000000L; ///< 检测视线方向
```

```
public final static long ST_MOBILE_DETECT_DYNAMIC_GESTURE = 0x200000000000L;
///< 检测动态手势
public final static long ST_MOBILE_DETECT_AVATAR_HELPINFO = 0x800000000000L;
///< 检测avatar辅助信息

public final static long ST_MOBILE_SEG_HAIR = 0x20000000;          ///< 头发分割
public final static long ST_MOBILE_SEG_HEAD = 0x0100000000L;     /// 检测头部分割
public final static long ST_MOBILE_SEG_SKIN = 0x0200000000L;     /// 检测皮肤分割
public final static long ST_MOBILE_DETECT_MOUTH_PARSE = 0x800000000000000L; ///<
检测嘴部遮挡
```

5.3 图像检测

- 检测接口有两种方式，包括直接返回检测结果的接口和不直接返回检测结果的接口
- 直接返回检测结果的接口使用简便，但是jni传输效率较低，性能略差：

```
//检测并返回检测结果
STHumanAction humanAction =
mSTHumanActionNative.humanActionDetect(mNv21ImageData,
STCommon.ST_PIX_FMT_NV21, mDetectConfig, orientation, mImageHeight,
mImageWidth);
```

- 不直接返回检测结果的接口，需要配合特效接口使用，如需使用检测结果需要使用get接口获取，使用较复杂：

```
//检测人连关键点等信息
int ret = mSTHumanActionNative.nativeHumanActionDetectPtr(mNv21ImageData,
STCommon.ST_PIX_FMT_NV21, mDetectConfig, orientation, mImageHeight,
mImageWidth);

//获取检测结果，注getNativeHumanAction不是线程安全的，请在nativeHumanActionDetectPtr
同线程使用或加锁，sample渲染线程使用的是copy之后的检测结果（nativeHumanActionPtrCopy）
STHumanAction humanAction = mSTHumanActionNative.getNativeHumanAction();
```

- 检测参数说明

```
imgData         用于检测的图像数据
imageFormat     用于检测的图像数据的像素格式,比如STCommon.ST_PIX_FMT_NV12。能够独立提取灰
度通道的图像格式处理速度较快,                                比如ST_PIX_FMT_GRAY8,
ST_PIX_FMT_YUV420P, ST_PIX_FMT_NV12, ST_PIX_FMT_NV21
detect_config 检测选项，代表当前需要检测哪些动作，例如
ST_MOBILE_EYE_BLINK|ST_MOBILE_MOUTH_AH表示当前帧只检测眨眼和张嘴
orientation     图像中人脸的方向,例如STRotateType.ST_CLOCKWISE_ROTATE_0
imageWidth      用于检测的图像的宽度(以像素为单位)
imageHeight     用于检测的图像的高度(以像素为单位)
```

5.4 检测句柄销毁

- 销毁检测句柄

```
synchronized (mHumanActionHandleLock) {
    mSTHumanActionNative.destroyInstance();
}
```

6 代码中接入特效

- 特效功能使用STMobileEffectNative实现

6.1 特效句柄的初始化

- 特效句柄初始化

```
int result = mSTMobileEffectNative.createInstance(mContext,
STMobileEffectNative.EFFECT_CONFIG_NONE);
```

- 初始化config参数

```
//创建handle使用
public static final int EFFECT_CONFIG_NONE = 0x0;  //默认模式，输入连续序列帧时使用，比如视频或预览模式
public static final int EFFECT_CONFIG_IMAGE_MODE = 0x2;  //图像模式，输入为静态图像或单帧图像
```

6.2 设置基础美颜、高级美颜和微整形

- 美颜基础美颜、高级美颜和微整形等对应的枚举类型如下，参见STEffectBeautyType.java：

```
// 基础美颜 base
public static final int EFFECT_BEAUTY_BASE_WHITTEN                      = 101;
 // 美白
public static final int EFFECT_BEAUTY_BASE_REDDEN                       = 102;
 // 红润
public static final int EFFECT_BEAUTY_BASE_FACE_SMOOTH                  = 103;
 // 磨皮

// 美形 reshape
public static final int EFFECT_BEAUTY_RESHAPE_SHRINK_FACE               = 201;
 // 瘦脸
public static final int EFFECT_BEAUTY_RESHAPE_ENLARGE_EYE               = 202;
 // 大眼
public static final int EFFECT_BEAUTY_RESHAPE_SHRINK_JAW                = 203;
 // 小脸
```

```
public static final int EFFECT_BEAUTY_RESHAPE_NARROW_FACE               = 204;
 // 窄脸
public static final int EFFECT_BEAUTY_RESHAPE_ROUND_EYE                 = 205;
 // 圆眼

// 微整形 plastic
public static final int EFFECT_BEAUTY_PLASTIC_THINNER_HEAD              = 301;
 // 小头
public static final int EFFECT_BEAUTY_PLASTIC_THIN_FACE                 = 302;
 // 瘦脸型
public static final int EFFECT_BEAUTY_PLASTIC_CHIN_LENGTH               = 303;
 // 下巴
public static final int EFFECT_BEAUTY_PLASTIC_HAIRLINE_HEIGHT           = 304;
 // 额头
public static final int EFFECT_BEAUTY_PLASTIC_APPLE_MUSLE               = 305;
 // 苹果肌
public static final int EFFECT_BEAUTY_PLASTIC_NARROW_NOSE               = 306;
 // 瘦鼻翼
public static final int EFFECT_BEAUTY_PLASTIC_NOSE_LENGTH               = 307;
 // 长鼻
public static final int EFFECT_BEAUTY_PLASTIC_PROFILE_RHINOPLASTY       = 308;
 // 侧脸隆鼻
public static final int EFFECT_BEAUTY_PLASTIC_MOUTH_SIZE                = 309;
 // 嘴型
public static final int EFFECT_BEAUTY_PLASTIC_PHILTRUM_LENGTH           = 310;
 // 缩人中
public static final int EFFECT_BEAUTY_PLASTIC_EYE_DISTANCE              = 311;
 // 眼距
public static final int EFFECT_BEAUTY_PLASTIC_EYE_ANGLE                 = 312;
 // 眼睛角度
public static final int EFFECT_BEAUTY_PLASTIC_OPEN_CANTHUS              = 313;
 // 开眼角
public static final int EFFECT_BEAUTY_PLASTIC_BRIGHT_EYE                = 314;
 // 亮眼
public static final int EFFECT_BEAUTY_PLASTIC_REMOVE_DARK_CIRCLES       = 315;
 // 祛黑眼圈
public static final int EFFECT_BEAUTY_PLASTIC_REMOVE_NASOLABIAL_FOLDS   = 316;
 // 祛法令纹
public static final int EFFECT_BEAUTY_PLASTIC_WHITE_TEETH               = 317;
 // 白牙
public static final int EFFECT_BEAUTY_PLASTIC_SHRINK_CHEEKBONE          = 318;
 // 瘦颧骨
public static final int EFFECT_BEAUTY_PLASTIC_OPEN_EXTERNAL_CANTHUS     = 319;
 // 开外眼角比例

// 调整 tone
public static final int EFFECT_BEAUTY_TONE_CONTRAST                     = 601;
 // 对比度
```

```
public static final int EFFECT_BEAUTY_TONE_SATURATION                    = 602;
 // 饱和度
public static final int EFFECT_BEAUTY_TONE_SHARPEN                       = 603;
 // 锐化
public static final int EFFECT_BEAUTY_TONE_CLEAR                         = 604;
 // 清晰度
```

- 设置基础美颜、高级美颜和微整形强度。以设置美白、瘦脸和瘦鼻翼强度为0.5为例

```
//设置美白
mSTMobileEffectNative.setBeautyStrength(STEffectBeautyType.EFFECT_BEAUTY_BASE_WHITTEN, 0.5f);

//设置瘦脸
mSTMobileEffectNative.setBeautyStrength(STEffectBeautyType.EFFECT_BEAUTY_RESHAPE_SHRINK_FACE, 0.5f);

//设置瘦鼻翼
mSTMobileEffectNative.setBeautyStrength(STEffectBeautyType.EFFECT_BEAUTY_PLASTIC_NARROW_NOSE, 0.5f);
```

- 设置美白和磨皮模式，此接口目前只支持美白和磨皮：

```
//设置美白模式，0为ST_BEAUTIFY_WHITEN_STRENGTH，1为ST_BEAUTIFY_WHITEN2_STRENGTH，2为ST_BEAUTIFY_WHITEN3_STRENGTH
mSTMobileEffectNative.setBeautyMode(STEffectBeautyType.EFFECT_BEAUTY_BASE_WHITTEN, 0);

//设置磨皮模式，默认值2.0，1表示对全图磨皮，2表示精细化磨皮
mSTMobileEffectNative.setBeautyMode(STEffectBeautyType.EFFECT_BEAUTY_BASE_FACE_SMOOTH, 2);
```

6.3 设置美妆

- 美妆类型枚举，参见STEffectBeautyType.java：

```
// 美妆 makeup
public static final int EFFECT_BEAUTY_HAIR_DYE                          = 401;
 // 染发
public static final int EFFECT_BEAUTY_MAKEUP_LIP                        = 402;
 // 口红
public static final int EFFECT_BEAUTY_MAKEUP_CHEEK                      = 403;
 // 腮红
public static final int EFFECT_BEAUTY_MAKEUP_NOSE                       = 404;
 // 修容
public static final int EFFECT_BEAUTY_MAKEUP_EYE_BROW                   = 405;
 // 眉毛
public static final int EFFECT_BEAUTY_MAKEUP_EYE_SHADOW                 = 406;
 // 眼影
public static final int EFFECT_BEAUTY_MAKEUP_EYE_LINE                   = 407;
 // 眼线
public static final int EFFECT_BEAUTY_MAKEUP_EYE_LASH                   = 408;
 // 眼睫毛
public static final int EFFECT_BEAUTY_MAKEUP_EYE_BALL                   = 409;
 // 美瞳
public static final int EFFECT_BEAUTY_MAKEUP_PACKED                     = 410;
 ///< 打包的美妆素材，可能包含一到多个单独的美妆模块，另外，添加时会替换所有现有美妆
```

- 设置美妆，以设置口红素材为例：

```
//设置口红素材
mSTMobileEffectNative.setBeauty(STEffectBeautyType.EFFECT_BEAUTY_MAKEUP_LIP,
typePath);

//从assets文件目录设置口红素材
mSTMobileEffectNative.setBeautyFromAssetsFile(STEffectBeautyType.EFFECT_BEAUTY_
MAKEUP_LIP, assetsTypePath,  mContext.getAssets());
```

- 设置美妆强度, 以设置口红强度为0.5为例：

```
//设置口红强度为0.5
mSTMobileEffectNative.setBeautyStrength(STEffectBeautyType.EFFECT_BEAUTY_MAKEUP
_LIP, 0.5f);
```

6.4 设置滤镜

- 滤镜类型枚举，参见STEffectBeautyType.java：

```
// 滤镜 filter
public static final int EFFECT_BEAUTY_FILTER                            = 501;
 // 滤镜
```

- 设置滤镜：

```
//设置滤镜素材
mSTMobileEffectNative.setBeauty(STEffectBeautyType.EFFECT_BEAUTY_FILTER, path);

//从assets文件目录设置滤镜素材
mSTMobileEffectNative.setBeautyFromAssetsFile(STEffectBeautyType.EFFECT_BEAUTY_
FILTER, assetsTypePath,  mContext.getAssets());
```

- 设置滤镜强度, 以设置滤镜强度为0.5为例：

```
//设置滤镜强度为0.5
mSTMobileEffectNative.setBeautyStrength(STEffectBeautyType.EFFECT_BEAUTY_FILTER
, 0.5f);
```

6.5 设置贴纸

- 切换贴纸

```
//切换贴纸
int packageId = mSTMobileEffectNative.changePackage(mCurrentStickerPath);

//从assets文件目录切换贴纸
int packageId =
mSTMobileEffectNative.changePackageFromAssetsFile(mCurrentStickerPath,
mContext.getAssets());
```

- 添加贴纸

```
//添加贴纸
int packageId = mSTMobileEffectNative.addPackage(mCurrentStickerPath);

//从assets文件目录添加贴纸
int packageId =
mSTMobileEffectNative.addPackageFromAssetsFile(mCurrentStickerPath,
mContext.getAssets());
```

- 贴纸声音自定义播放，贴纸声音默认使用STSoundPlay进行管理，其实现为单实例，与特效句柄绑定，用户可重写PlayControlListener来实现自定义声音播放逻辑。

```
/**
 * 音频播放监听器
 */
public interface PlayControlListener {
```

```
    /**
     * 加载音频素材callback
     *
     * @param name    音频名称
     * @param content 音频内容
     */
    void onSoundLoaded(String name, byte[] content);

    /**
     * 播放音频callback
     *
     * @param name 音频名称
     * @param loop 循环次数，0表示无限循环，直到onStopPlay回调，大于0表示循环次数
     */
    void onStartPlay(String name, int loop);

    /**
     * 停止播放callback
     *
     * @param name 音频名称
     */
    void onStopPlay(String name);

    /**
     * 暂停播放callback
     *
     * @param name 音频名称
     */
    void onSoundPause(String name);

    /**
     * 重新播放callback
     *
     * @param name 音频名称
     */
    void onSoundResume(String name);
}

/**
 * 设置播放控制监听器
 *
 * @param listener listener为null，SDK默认处理，若不为null，用户自行处理
 */
public void setPlayControlListener(PlayControlListener listener) {
    if (listener != null) {
        mPlayControlDefaultListener = listener;
    }
}
```

- 贴纸Trigger信息，贴纸中可包含trigger信息，用于触发贴纸特效，如眨眼、手势等动作，使用getHumanActionDetectConfig接口重新配置检测config

```
public long getStickerTriggerAction() {
    return mSTMobileEffectNative.getHumanActionDetectConfig();
}
```

6.6 特效渲染

- 特效渲染接口配合直接输出检测接口（humanActionDetect）使用，需在opengl线程调用

```
//渲染接口输入参数
STEffectRenderInParam sTEffectRenderInParam = new
STEffectRenderInParam(humanAction, mAnimalFaceInfo[mCameraInputTextureIndex],
renderOrientation, renderOrientation, false, customParam, stEffectTexture,
null);
//渲染接口输出参数
STEffectRenderOutParam stEffectRenderOutParam = new
STEffectRenderOutParam(stEffectTextureOut, null,
mSTHumanAction[mCameraInputTextureIndex]);
long mStartRenderTime = System.currentTimeMillis();
mSTMobileEffectNative.render(sTEffectRenderInParam, stEffectRenderOutParam,
false);
```

- 特效渲染接口配合不直接输出检测接口（nativeHumanActionDetectPtr）使用，需在opengl线程调用

```
//渲染接口输入参数
STEffectRenderInParam sTEffectRenderInParam = new
STEffectRenderInParam(mSTHumanActionNative.getNativeHumanActionPtrCopy(),
mAnimalFaceInfo[mCameraInputTextureIndex], renderOrientation,
renderOrientation, false, customParam, stEffectTexture, null);
//渲染接口输出参数
STEffectRenderOutParam stEffectRenderOutParam = new
STEffectRenderOutParam(stEffectTextureOut, null,
mSTHumanAction[mCameraInputTextureIndex]);
long mStartRenderTime = System.currentTimeMillis();
mSTMobileEffectNative.render(sTEffectRenderInParam, stEffectRenderOutParam,
false);
```

- 接口输入参数(STEffectRenderInParam)说明

```
nativeHumanActionResult 人脸等检测结果，不需要可传0（如基础美颜、滤镜功能）
humanAction 人脸等检测结果，不需要可传null（如基础美颜、滤镜功能）
//注：nativeHumanActionResult和humanAction传入其中一个参数即可，详细说明见
STEffectRenderInParam类

animalFaceInfo 动物检测信息，不需要可传null
rotate 人脸在纹理中的方向
needMirror 传入图像与显示图像是否是镜像关系
customParam 用户自定义参数，不需要可传null
texture 输入纹理，不可传null
image 输入图像数据，不需要可传null
```

- 接口输出参数(STEffectRenderOutParam)说明

```
texture 输出纹理信息，需上层初始化
humanAction 输出脸部变形后的人脸检测结果，不需要可传null
image 输出图像数据，用于推流等场景，不需要可传null
```

6.7 获取当前特效覆盖的美颜参数

- 获取覆盖生效的美颜的信息, 仅在添加、更改和移除贴纸素材之后调用。使用如下：

```
//切换贴纸素材
int packageId = mSTMobileEffectNative.changePackage(mCurrentStickerPath);

//获取覆盖生效的美颜的信息数量
int beautyOverlapCount = mSTMobileEffectNative.getOverlappedBeautyCount();

//取覆盖生效的美颜的信息
if (beautyOverlapCount > 0) {
    STEffectBeautyInfo[] beautyInfos =
mSTMobileEffectNative.getOverlappedBeauty(beautyOverlapCount);
    //TODO：根据获取的刷新UI等
}
```

- getOverlappedBeauty接口输出参数

```
public class STEffectBeautyInfo {
    int type;        // 美颜类型
    int mode;        // 美颜的模式
    float strength;  //美颜强度
    byte[] name = new byte[256];     // 所属的素材包的名字
}
```

6.8 特效句柄销毁

- 特效句柄销毁，需在opengl线程调用

```
mSTMobileEffectNative.destroyInstance();
```

7 帧处理流程

- 以sample中单输入buffer为例，参见CameraDisplaySingleBuffer

7.1 相机回调

- 相机回调设置，获取相机nv21数据

```
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);
}
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();
    }
};
```

7.2 帧检测

- 根据相机获取的图像数据，检测人脸关键点等信息

```
 //检测线程
mDetectThreadPool.submit(new Runnable() {
    @Override
    public void run() {
```

```
        int orientation = getHumanActionOrientation();
        synchronized (mHumanActionLock) {
            long startHumanAction = System.currentTimeMillis();
            int ret =
mSTHumanActionNative.nativeHumanActionDetectPtr(mNv21ImageData,
STCommon.ST_PIX_FMT_NV21, mDetectConfig, orientation, mImageHeight,
mImageWidth);
            LogUtils.i(TAG, "human action cost time: %d",
System.currentTimeMillis() - startHumanAction);

            //nv21数据为横向，相对于预览方向需要旋转处理，前置摄像头还需要镜像

 STHumanAction.nativeHumanActionRotateAndMirror(mSTHumanActionNative,
mSTHumanActionNative.getNativeHumanActionResultPtr(), mImageWidth,
mImageHeight, mCameraID, mCameraProxy.getOrientation(),
Accelerometer.getDirection());
        }
    }
});
```

7.3 帧渲染

- 根据检测到的关键点等信息，以及传感器信息等，进行特效渲染处理，输出渲染后的问题，需要在opengl线程调用

```
//输入纹理
STEffectTexture stEffectTexture = new STEffectTexture(textureId, mImageWidth,
mImageHeight, 0);
//输出纹理，需要在上层初始化
STEffectTexture stEffectTextureOut = new STEffectTexture(mBeautifyTextureId[0],
mImageWidth, mImageHeight, 0);
//输入纹理的人脸方向
int renderOrientation = getCurrentOrientation();

//用户自定义参数设置
int event = mCustomEvent;
STEffectCustomParam customParam;
if(mSensorEvent != null && mSensorEvent.values != null &&
mSensorEvent.values.length > 0){
    customParam = new STEffectCustomParam(new
STQuaternion(mSensorEvent.values), mCameraID ==
Camera.CameraInfo.CAMERA_FACING_FRONT, event);
} else {
    customParam = new STEffectCustomParam(new STQuaternion(0f,0f,0f,1f),
mCameraID == Camera.CameraInfo.CAMERA_FACING_FRONT, event);
}
```

```
//渲染接口输入参数
STEffectRenderInParam sTEffectRenderInParam = new
STEffectRenderInParam(mSTHumanActionNative.getNativeHumanActionPtrCopy(),
mAnimalFaceInfo[mCameraInputTextureIndex], renderOrientation,
renderOrientation, false, customParam, stEffectTexture, null);
//渲染接口输出参数
STEffectRenderOutParam stEffectRenderOutParam = new
STEffectRenderOutParam(stEffectTextureOut, null,
mSTHumanAction[mCameraInputTextureIndex]);
long mStartRenderTime = System.currentTimeMillis();
mSTMobileEffectNative.render(sTEffectRenderInParam, stEffectRenderOutParam,
false);
LogUtils.i(TAG, "render cost time total: %d", System.currentTimeMillis() -
mStartRenderTime);

if(stEffectRenderOutParam != null && stEffectRenderOutParam.getTexture() !=
null){
    textureId = stEffectRenderOutParam.getTexture().getId();
}
```

8 通用物体追踪

- 通用物体追踪使用STMobileObjectTrackNative实现

8.1 创建通用物体追踪句柄

- 创建通用物体追踪句柄

```
protected void initObjectTrack() {
    int result = mSTMobileObjectTrackNative.createInstance();
}
```

8.2 设置通用物体追踪目标区域

- 设置通用物体追踪目标区域

```
STRect inputRect = new STRect(mTargetRect.left, mTargetRect.top,
mTargetRect.right, mTargetRect.bottom);
mSTMobileObjectTrackNative.setTarget(mImageData, STCommon.ST_PIX_FMT_NV21,
mImageHeight, mImageWidth, inputRect);
```

8.3 通用物体追踪

- 通用物体追踪

```
STRect outputRect = mSTMobileObjectTrackNative.objectTrack(mImageData,
STCommon.ST_PIX_FMT_NV21, mImageHeight, mImageWidth, score);
```

- 获取通用物体追踪区域，objectTrack接口输出的Rect即图像中目标区域，每帧均会更新

8.4 通用物体追踪句柄销毁

- 通用物体追踪句柄销毁

```
 mSTMobileObjectTrackNative.destroyInstance();
```

9 人脸属性检测

- 人脸属性检测使用STMobileFaceAttributeNative实现

9.1 创建人脸属性检测句柄

- 创建人脸属性检测句柄，传入人脸属性模型路径

```
//从绝对路径创建
int result =  mSTMobileFaceAttributeNative.createInstance(modelpath);

//从assets路径创建
int result =
 mSTMobileFaceAttributeNative.createInstanceFromAssetFile(modelpath,
mContext.getAssets());
```

9.2 人脸属性检测

- 人脸属性检测

```
STMobile106[] arrayFaces = null;
arrayFaces = humanAction.getMobileFaces();

if (arrayFaces != null && arrayFaces.length != 0) { // face attribute
    STFaceAttribute[] arrayFaceAttribute = new
STFaceAttribute[arrayFaces.length];
    int result = mSTFaceAttributeNative.detect(data, STCommon.ST_PIX_FMT_NV21,
mImageHeight, mImageWidth, arrayFaces, arrayFaceAttribute);
    if (result == 0) {
        if (arrayFaceAttribute[0].getAttributeCount() > 0) {
```

```
            mFaceAttribute =
STFaceAttribute.getFaceAttributeString(arrayFaceAttribute[0]);
        } else {
            mFaceAttribute = "null";
        }
    }
}
```

9.3 人脸属性检测句柄销毁

- 人脸属性检测句柄销毁

```
mSTMobileFaceAttributeNative.destroyInstance();
```

10 动物脸检测

- 目前只支持猫脸检测，依靠STMobileAnimalNative实现

10.1 创建动物脸检测句柄

- 创建动物脸检测句柄，传入动物脸检测模型路径

```
//从绝对路径创建
int result =  STMobileAnimalNative.createInstance(modelpath);

//从assets路径创建
int result =  STMobileAnimalNative.createInstanceFromAssetFile(modelpath,
mContext.getAssets());
```

10.2 动物脸检测

- 动物脸检测

```
STAnimalFace[] animalFaces = mStAnimalNative.animalDetect(imageData, format,
orientation, width, height);
```

10.3 动物脸检测句柄销毁

- 动物脸检测句柄销毁

```
STMobileAnimalNative.destroyInstance();
```

11 相机数据回调模式

- sample中有两种相机输入模式供参考，即单输入buffer和单输入纹理两种模式，sample中默认使

用单输入buffer

11.1 单输入buffer模式

- 单输入buffer模式，即CameraDisplaySingleBuffer单输入模式，使用onPreviewFrame中相机回调的NV21 Buffer生成纹理做渲染，然后用此buffer检测人脸关键点等信息，使用双缓冲的方式检测和渲染在两个线程进行。
- 优点:帧率高，不需要readPixel。
- 缺点:需要使用buffer上传纹理，需要考虑线程同步问题，渲染延后一帧（双缓冲设计）。

11.2 单输入纹理模式

- 单输入纹理模式，即CameraDisplaySingleTexture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()等方式获取buffer数据。建议配合nativeHumanActionDetectPtr使用，优化了jni数据传输，以提升整体效率。
- 优点：不需要回调buffer，cpu占用率低，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。
- 更新：单输入纹理模式更新了流程，优化了readPixel耗时，优化了只检测人脸时图像格式和分辨率。

11.3 不同输入模式切换

- 不同输入模式在Sample中均有实现。
- Sample中单双输入切换参考Sample中的CameraActivity.java。

```
//双输入模式
private BaseCameraDisplay mCameraDisplay;

//默认使用单输入buffer
mCameraDisplay = new CameraDisplaySingleBuffer(getApplicationContext(),
mChangePreviewSizeListener, glSurfaceView);
//使用单输入texture
mCameraDisplay = new CameraDisplaySingleTexture(getApplicationContext(),
mChangePreviewSizeListener, glSurfaceView);
```

12 旋转和方向等相关说明

12.1 相机的方向

- 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。
- 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;
}
```

12.2 检测方向

- 根据设备传感器方向确定HumanActionDetect输入方向：

```
/**
 * 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buffer中的朝向
 * @return 人脸在buffer中的朝向
 */
private int getHumanActionOrientation(){
    boolean frontCamera = (mCameraID == Camera.CameraInfo.CAMERA_FACING_FRONT);

    //获取重力传感器返回的方向
    int orientation = Accelerometer.getDirection();

    //在使用后置摄像头，且传感器方向为0或2时，后置摄像头与前置orentation相反
    if(!frontCamera && orientation == STRotateType.ST_CLOCKWISE_ROTATE_0){
        orientation = STRotateType.ST_CLOCKWISE_ROTATE_180;
    }else if(!frontCamera && orientation ==
STRotateType.ST_CLOCKWISE_ROTATE_180){
        orientation = STRotateType.ST_CLOCKWISE_ROTATE_0;
    }

    // 请注意前置摄像头与后置摄像头旋转定义不同 && 不同手机摄像头旋转定义不同
    if (((mCameraProxy.getOrientation() == 270 && (orientation &
STRotateType.ST_CLOCKWISE_ROTATE_90) == STRotateType.ST_CLOCKWISE_ROTATE_90) ||
            (mCameraProxy.getOrientation() == 90 && (orientation &
STRotateType.ST_CLOCKWISE_ROTATE_90) == STRotateType.ST_CLOCKWISE_ROTATE_0)))
        orientation = (orientation ^ STRotateType.ST_CLOCKWISE_ROTATE_180);
    return orientation;
}
```