OpenSL ES学习总结

简介：

OpenSL ES (Open Sound Library for Embedded Systems)是无授权费、跨平台、针对嵌入式系统精心优化的硬件音频加速API。简单来说OpenSL ES是一个嵌入式跨平台免费的音频处理库。

工作相关：

浏览器内部对音频的处理使用的就是OpenSL ES，所以对此库的相关API及流程进行了学习，并总结如下：

1.创建并设置SL引擎

```
SLObjectltf engineObject = NULL; //
SLEngineltf engineEngine = NULL; //

slCreateEngine(&engineObject, 0, NULL, 0, NULL, NULL);

(*engineObject)->Realize(engineObject, SL_BOOLEAN_FALSE); //
(*engineObject)->GetInterface(engineObject, SL_IID_ENGINE, &engineEngine);
```

对应的chromium实现代码：

```
LOG_ON_FAILURE_AND_RETURN( slCreateEngine(engine_object_.Receive(), 1, option, 0, NULL, NULL), false);
// Realize the SL engine object in synchronous mode.
LOG_ON_FAILURE_AND_RETURN(engine_object_->Realize(engine_object_.Get(), SL_BOOLEAN_FALSE), false);
// Get the SL engine interface which is implicit.
SLEngineItf engine;
LOG_ON_FAILURE_AND_RETURN(engine_object_->GetInterface( engine_object_.Get(), SL_IID_ENGINE, &engine), false);
```

2.创建输出设备

```
SLEngineltf engineEngine = NULL;
SLObjectltf outputMixObject = NULL;

(*engineEngine)->CreateOutputMix(engineEngine, &outputMixObject, 0, 0, 0);
//
(*outputMixObject)->Realize(outputMixObject, SL_BOOLEAN_FALSE); //

//
SLDataLocator_OutputMix outputMix = {
    SL_DATALOCATOR_OUTPUT_MIX,
    outputMixObject
};

SLDataSink audioSnk = {&outputMix, NULL}
```

对应的chromium实现代码：

```
// Create ouput mixer object to be used by the player.
LOG_ON_FAILURE_AND_RETURN((*engine)->CreateOutputMix(engine, output_mixer_.Receive(), 0, NULL, NULL), false);
 // Realizing the output mix object in synchronous mode.
 LOG_ON_FAILURE_AND_RETURN(output_mixer_->Realize(output_mixer_.Get(), SL_BOOLEAN_FALSE), false);
```

3.初始化播放器

```
const SLInterfaceID ids[1] = {SL_IID_BUFFERQUEUE};
const SLboolean req[1] = {SL_BOOLEAN_TRUE};

//
(*engineEngine)->CreateAudioPlayer(
    engineEngine,
    &pcmPlayerObject,
    &slDataSource,
    &audioSnk,
    1,
    ids,
    req);

//
(*pcmPlayerObject)->Realize(pcmPlayerObject, SL_BOOLEAN_FALSE);

//  Player
(*pcmPlayerObject)->GetInterface(
    pcmPlayerObject,
    SL_IID_PLAY,
    &pcmPlayerPlay);
```

对应的chromium实现代码：

```
LOG_ON_FAILURE_AND_RETURN((*engine)->CreateAudioPlayer(engine, player_object_.Receive(), &audio_source,
&audio_sink, base::size(interface_id), interface_id, interface_required), false);
 // Create AudioPlayer and specify SL_IID_ANDROIDCONFIGURATION.
  SLAndroidConfigurationItf player_config;   LOG_ON_FAILURE_AND_RETURN(player_object_->GetInterface
(player_object_.Get(), SL_IID_ANDROIDCONFIGURATION, &player_config),false);
// Set configuration using the stream type provided at construction.
  LOG_ON_FAILURE_AND_RETURN((*player_config) ->SetConfiguration(player_config, SL_ANDROID_KEY_STREAM_TYPE,
&stream_type_, sizeof(SLint32)), false);
// Set configuration using the stream type provided at construction.
  if (performance_mode_ > SL_ANDROID_PERFORMANCE_NONE) {
    LOG_ON_FAILURE_AND_RETURN((*player_config)->SetConfiguration(player_config,
SL_ANDROID_KEY_PERFORMANCE_MODE, &performance_mode_, sizeof(SLuint32)),false);   }
// Realize the player object in synchronous mode.
  LOG_ON_FAILURE_AND_RETURN(player_object_->Realize(player_object_.Get(), SL_BOOLEAN_FALSE), false);
 // Get an implicit player interface.
  LOG_ON_FAILURE_AND_RETURN(player_object_->GetInterface(player_object_.Get(), SL_IID_PLAY, &player_),
false);
```

4. 播放和缓存队列

```
//
(*pcmPlayerObject)->GetInterface(
    pcmPlayerObject,
    SL_IID_BUFFERQUEUE,
    &pcmBufferQueue);

//
(*pcmBufferQueue)->RegisterCallback(
    pcmBufferQueue,
    pcmBufferCallBack,
    NULL);
//

(*pcmPlayerPlay)->SetPlayState(pcmPlayerPlay, SL_PLAYSTATE_PLAYING);

// ""={\0} (0)
(*pcmBufferQueue)->Enqueue(pcmBufferQueue, "", 1);
```

对应的chromium实现代码：

```
// Get the simple buffer queue interface.
LOG_ON_FAILURE_AND_RETURN(player_object_->GetInterface(player_object_.Get(), SL_IID_BUFFERQUEUE,
&simple_buffer_queue_),false);
// Register the input callback for the simple buffer queue.
// This callback will be called when the soundcard needs data.
LOG_ON_FAILURE_AND_RETURN( (*simple_buffer_queue_)->RegisterCallback(simple_buffer_queue_,
SimpleBufferQueueCallback, this), false);
// Verify that we are in a playing state.
SLuint32 state;   SLresult err = (*player_)->GetPlayState(player_, &state);
// Calculate the position relative to the number of frames written.
uint32_t position_in_ms = 0;
SLresult err = (*player_)->GetPosition(player_, &position_in_ms);
// Enqueue the buffer for playback.
err = (*simple_buffer_queue_)->Enqueue(simple_buffer_queue_, audio_data_[active_buffer_index_],
num_filled_bytes);
```