Astra SDK v2.0 开发文档

深 圳 奥 比 中 光 科 技 有 限 公 司

1 引言

1.1 安装

1.1.1 系统要求

Windows

- Microsoft® Windows® 10/8.1/8/7 (32 or 64-bit)
- 150 MB 磁盘空间
- USB 2.0 或 USB 3.0 接口
- Microsoft® Visual Studio 2013 Community Edition 版本或更高

Android

- Version 4.4 (Lollipop) 版本或更高
- ARM
- USB 2 OTG 或 USB 3 OTG

Linux

- ubunu14 和 ubuntu16 的 x64 系统
- USB 2.0 或 USB 3.0 接口
- 150 MB 磁盘空间

其他要求

- 设备：Astra
- 设备驱动：windows 下请安装“SensorDriver.exe”驱动文件

1.1.2 安装 SDK

安装 Astra SDK 前，请先到 <http://forum.orbbec.com.cn/>`_下载最新版本 SDK。下载完成后，请将 SDK 解压到您可以记住的位置。

1.2 编译例程

1.2.1 Windows

1) 在已解压的 AstraSDK 路径的/samples/vs2013 或 /samples/vs2015 目录下找到 Astra-samples.sln。
2) 用 Visual Studio 2013 或 Visual Studio 2015 打开对应的 Astra-samples.sln 文件。
3) 在 Visual Studio 中通过菜单中的 **BUILD -> Build Solution** 或按快捷键 **Ctrl+Shift+B** 来编译示例程序。
4) 稍等片刻，编译完成后程序即可开始运行。
5) 如需在 Visual Studio 中启动示例程序，可以右键点击待启动的工程文件，然后选择'Set as StartUp Project'，就可以将该示例程序设置为启动项目。之后通过使用菜单中的 **DEBUG -> Start Debugging** 或点击键盘上的 **F5** 键来进行 debug。

1.2.2 Android

1） 提供了 AstraUnity536Sample*.apk 和 AstraUnity554Sample*.apk，直接采用 adb install 安装 apk 即可。
2） 提供了 AstraSDKUnity536*.unitypackage 和 AstraSDKUnity536*.unitypackage，可以创建一个 unity 工程，导入 unitypackage 文件。

1.2.3 Linux

1) 在使用 SDK 之前，需要先安装相关的工具，通过以下命令：

```
$ apt-get install libavcodec-dev libsfml-dev libavformat-dev libswscale-dev。
```

2) 将我们提供的.tar.gz 压缩文件直接拷贝到 ubuntu14 或 ubuntu16,请不要在拷贝前解压该文件。
3) 在 SDK 的 bin 文件夹下运行相关应用，您可使用如下命令：

```
$ sudo ./ SimpleDepthViewer-SFML
```

1.3 Hello world

在深入细节之前,请简要了解我们的 SDK，其目的是：
1) 正确的初始化和结束 SDK
2) 从摄像头读取数据
3) 检查 ASTRA 的深度相机提供的深度信息

准备：如果您跳过了 SDK 的安装和例程编译环节，请确保至少将 Astra SDK v2.0.7-beta 下载并解压到一个方便访问的目录下。

1）拷贝下面的代码到你的 main.cpp 文件:

```
1.  #include <astra/astra.hpp>
2.
3.  #include <cstdio>
4.  #include <iostream>
5.
6.  int main(int argc, char** argv)
7.  {
8.     std::cout << "hit enter to exit program" << std::endl;
9.     std::cin.get();
10.
11.    return 0;
12. }
```

Line 1 - astra.hpp 是所有基于本 SDK 的应用程序都必须包含的文件。这是 SDK 的核心，是所有基于 DK 的 C++程序都需要的。

Lines 9-10 - 我们将用 std::cin.get()来确保程序关闭窗口之前看到打印信息。

2）初始化和结束 astra,可以通过增加 line3 和 line7 代码实现。

```
1.  int main(int argc, char** argv)
2.  {
3.     astra::initialize();
4.
5.     astra::StreamSet streamSet;
6.
7.     astra::terminate();
8.
9.     std::cout << "hit enter to exit program" << std::endl;
10.    std::cin.get();
11.
12.    return 0;
13. }
```

3）StreamSet 来创建一个 StreamReader (line 6)，然后启动深度流（line 8）。

```
1.   int main(int argc, char** argv)
2.  {
3.     astra::initialize();
4.
5.     astra::StreamSet streamSet;
6.     astra::StreamReader reader = streamSet.create_reader();
7.
8.     reader.stream<astra::DepthStream>().start();
```

```
9.
10.     astra::terminate();
11.
12.     std::cout << "hit enter to exit program" << std::endl;
13.     std::cin.get();
14.
15.     return 0;
16. }
```

4）读取 StreamSet 流

我们的 `StreamSet` 对象来获取一些数据了，现在来试试读取 Astra 提供的其中一个数据流。数据流包含的是摄像头输出的多帧(frames)数据的集合。 Astra 目前支持几种类型的数据流，包括深度，彩色，手点、点云、人体以及背景抠图等。

为了从 Astra 的数据流里拿到一帧的数据，我们需要 `StreamReader` 去访问这些数据（line 6）。在本例程里，我们只读取深度数流。深度流给我们提供的是摄像头每一帧每一个像素看到的场景距离信息。

接下来就可以开始读取多帧数据了，为此我们在 `StreamReader` 的 `get_latest_frame` 函数外面添加一个循环即可。在这个例程里，我们将读取深度流的前面 100 帧并将每一帧的第一个像素值打印到命令行窗口。

```
1.  int main(int argc, char** argv)
2.  {
3.     astra::initialize();
4.
5.     astra::StreamSet streamSet;
6.     astra::StreamReader reader = streamSet.create_reader();
7.
8.     reader.stream<astra::DepthStream>().start();
9.
10.    const int maxFramesToProcess = 100;
11.    int count = 0;
12.
13.    do {
14.       astra::Frame frame = reader.get_latest_frame();
15.       const auto depthFrame = frame.get<astra::DepthFrame>();
16.
17.       const int frameIndex = depthFrame.frame_index();
18.       const short pixelValue = depthFrame.data()[0];
19.
20.       std::cout << std::endl
21.                 << "Depth frameIndex: " << frameIndex
22.                 << " pixelValue: " << pixelValue
23.                 << std::endl
24.                 << std::endl;
```

```
25.
26.        count++;
27.    } while (count < maxFramesToProcess);
28.
29.    std::cout << "Press any key to continue...";
30.    std::cin.get();
31.
32.    astra::terminate();
33.
34.    std::cout << "hit enter to exit program" << std::endl;
35.    std::cin.get();
36.
37.    return 0;
38. }
```

- Line 10 - 保存待处理的最大帧数
- Line 11 - 记录已经处理的帧数
- Line 13-27 - 帧处理循环。
- Line 17 - 从深度帧中得到帧索引的一个拷贝。
- Line 18 - 得到该深度帧数据的第一个像素的值拷贝。
- Line 20-24 - 将上面读取的两个值打印到命令行窗口。
- Line 29-30 - 暂停程序以便查看打印结果。

2 概念

我们已经完成了 Astra SDK 的安装过程，并熟悉了一些基本的操作，现在让我们将视线转移到 SDK 的三个核心概念 - streams，streamsets and readers。了解这些概念对于我们后期的开发和应用有着至关重要的意义。此外，深入理解这三个概念对于后期深入研发以及快速上手未来可能发布的 SDK 新功能都会有极大的帮助。

2. 1 Streams

Streams 是从特定数据源发出的连续帧形成的流。可以简单理解为老电影的拍摄播放模式，由录制的一张张的图像连续输出而形成的视频。每一张图像可看作一帧，Streams 可以理解为一部连续的视频。Streams 与视频的不同点是，stream 并不需要有结束点。

Streams 有很多不同的形式。例如，彩色帧组成彩色视频流，深度帧组成深度视频流。以上两种视频一般都来源于实体相机（Astra 等），然而一些视频流也可以利用 SDK 中的‘plugins’来生成（更多关于‘plugins’的内容，会在后面介绍）。以 SDK 中的手势追踪（hand tracker）来举例，此手部视频流是利用深度流的数据来生成的。

2.2 StreamSets

StreamSets 是一组相互有联系的 streams。接着上面关于老电影的比喻，现代的电影不止有连续的视频，同时也会有音频。视频和音频相结合即为电影，两者缺一不可。这种声音和图像组成的结合体即可理解为一种 streamsets。

streamset 可以包含从实体相机中输出的数据，也可以包含其他从 SDK 插件中生成的数据。比如 Astra 相机就可以看成是由彩色和深度视频流，以及中间件生成的流（比如手势追踪数据）组成的 streamset。

每个 Streamsets 都用一个唯一的 URI 字符串来做标识，这使得开发者在开发过程中可以同时调用多个不同的 streamsets。这个方法对于需要同时调用多个 streamsets 的情况提供了极大的帮助，此方法同样可应用于访问网络服务器产生的上层 stream。

2.3 StreamReaders

StreamReader 是我们查看 stream 里面每一帧的窗口。在读取来自 streamsets 中的任意一帧数据时，我们需要使用 streamset 去创建一个读取工具来访问 stream 里的每一帧。我们再次以老电影为例来解释 Reader 的作用。当一部电影包含了所用需要的音频和视频信息后，电影本身并不具备播放功能。此时需要另一台设备，卷筒式放映机。放映机不仅可以读取电影信息并投射到屏幕上，同时也可以将视频和音频同步，从而达到最佳播放效果。

StreamReader 的作用与卷筒式放映机大致相同。它可以接受来自 streamset（比如电影）的特定流信息同时输出每一帧的信息到目标软件中（即放映机）。每个 reader 可以同时读取许多不同规格类型的 streams。一个 streamset 可以同时在目标程序中拥有多个可以读取其中数据的 reader。如果一个 reader 同时读取了多个不同的 streams，通常在传递到目标前，reader 都会将数据进行同步后再输出。

现在，您应该已经对于 Astra SDK 如何获取和传递从相机中取得的数据有了一些初步的了解。总结来说，每一种从相机中获取的数据均会以帧（frame）的形式存储，连续的帧即为流（stream），streamsets 是由不同又相互联系的流组成，可以使用 StreamReader 来进行数据读取和传递。

那么为什么要使用这种模式呢？因为灵活，可扩展。由于 streams 可以包含任何类型的数据，即使未来出现一些目前 Astra 还不支持的新数据，这样的架构也可以通过插件（plugin）的形式轻松加入新的 streams 类型。更有利的一点是，在加入新的 streams 类型时，这些新 streams 都会遵循现有的模式，所以在有新内容加入时，并不需要重新学习新的工作模式。

3 功能与数据

3.1 功能

经过上述对 SDK 安装，编译例程，SDK 开发中的核心概念了解之后，可以看看我们可以用 SDK 做些什么？ SDK 功能包括：深度图像、彩色图像、红外图像、点云数据、人体

模板、骨架识别、背景抠图、手心跟踪和手势识别。如下图所示：

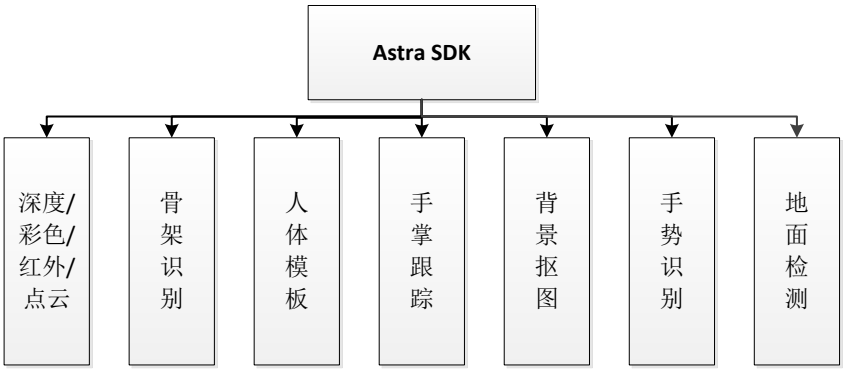

图 1 基本功能

下图显示 SDK 骨架效果图，其中绿色圆点表示骨架的关节点，浅蓝色圆点表示手掌的位置。深蓝色表示 bodymask 人体模板；紫蓝色表示地面检测的结果。

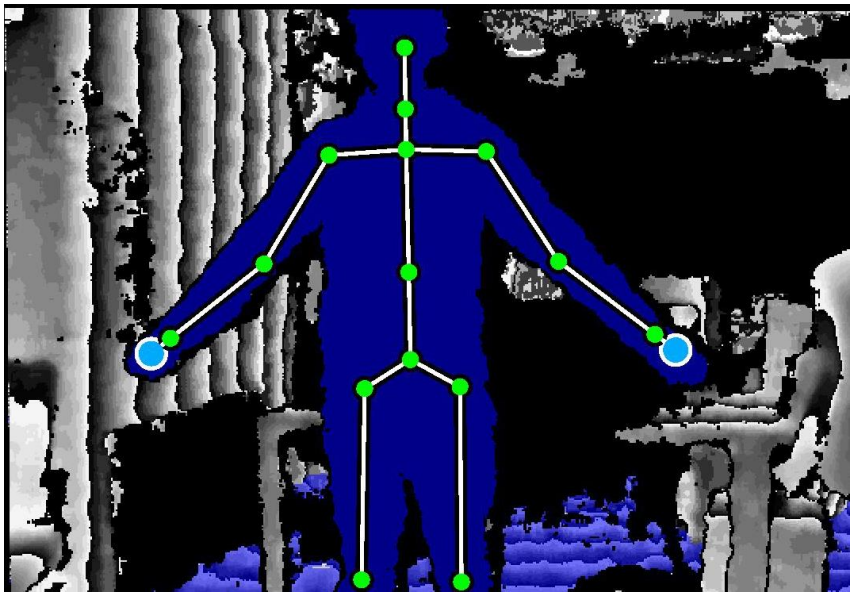

图 2 Astra 骨架效果图

3.1.1 流的类型

1) DepthStream 来自摄像头的深度数据，反映了视场范围内每个像素的深度值，单位是 mm。
2) ColorStream 来自摄像头的 RGB 像素数据，每个 ColorFrame 中的数组 data 包含 0-255 之间的数值，这些数值代表每个像素每个色彩通道的值。
3) PointStream: 从深度数据计算的世界坐标点云(X,Y,Z),每个像素点的数据类型是 astra:Vector3f。
4) HandStream 从深度数据计算出来的手掌的位置。
5) BodyStream 主要通过深度流计算得到的，提供了骨架识别和人体模板的功能。获取 user 的质心的世界坐标系的位置，数据是 astra:Vector3f 类型；user 的 ID，是 unit8 的数据类型。每个 Bodymask 的数据 data 表示每个像素点对应的 user 的 Id 值。

6) MaskedColorStream：对彩色图像中人进行抠图处理，其数据类型是 RGBA。

3.1.2 获取流数据

数据流均可以通过 StreamReaders 中获取，获取方式如下：

深度流：

```
astra::DepthStream depthStream = reader.stream<astra::DepthStream>();
```

彩色流：

```
astra::ColorStream colorStream = reader.stream<astra::ColorStream>();
```

点云流：

```
astra::PointStream pointStream = reader.stream<astra::PointStream>();
```

抠图流：

```
astra:: MaskedColorStream maskedColorStream = reader.stream<astra::MaskedColorStream>();
```

骨架流：

```
astra::BodyStream bodyStream = reader.stream<astra::BodyStream>();
```

手心流：

```
astra:: HandStream handStream = reader.stream<astra::HandStream>>();
```

3.2 基本类型

3.2.1 关节定义

```
enum class JointType : obt_joint_type_t
    {
        Head            = 0,
        ShoulderSpine   = 1,
        LeftShoulder    = 2,
        LeftElbow       = 3,
        LeftHand        = 4,
        RightShoulder   = 5,
        RightElbow      = 6,
        RightHand       = 7,
        MidSpine        = 8,
        BaseSpine       = 9,
        LeftHip         = 10,
        LeftKnee        = 11,
        LeftFoot        = 12,
        RightHip        = 13,
        RightKnee       = 14,
        RightFoot       = 15,
        Unknown         = 255,
```

```
    };
```

3.2.2 Astra 状态类型

```
typedef enum {
    ASTRA_STATUS_SUCCESS = 0,
    ASTRA_STATUS_INVALID_PARAMETER = 1,
    ASTRA_STATUS_DEVICE_ERROR = 2,
    ASTRA_STATUS_TIMEOUT = 3,
    ASTRA_STATUS_INVALID_PARAMETER_TOKEN = 4,
    ASTRA_STATUS_INVALID_OPERATION = 5,
    ASTRA_STATUS_INTERNAL_ERROR = 6,
    ASTRA_STATUS_UNINITIALIZED = 7
} astra_status_t;
```

3.3 类 Class

常用类如下：

1）**DepthStream/ DepthFrame**：获取深度数据，设置深度与彩色对齐，获取设备序列号，设备的 id 等。

2）**ColorStream/ ColorFrame**：获取彩色数据

3）**PointStream/ PointFrame**：获取点云。

4）**MaskedColorStream/ MaskedColorFrame**：获取的是彩色的抠图数据，数据类型是 RGBA.

5）**BodyFrame**：获取 bodymask 数据(数据类型是 uint8_t)和地面数据，以及所有的 bodies 以及每个 body 的质心。.

6）**Body**：骨架 body 的 ID，状态和 joints。

7）**Joint**：关节点类型 type，状态 status，以及在深度图像坐标系（depth position）和世界坐标系下的位置（world position）以及方向（orientation）。

8）**CoordinateMapper**：图像坐标系与世界坐标系之间的转换。

4 例程

SDK 提供了两个方法来获取流数据：轮询和监听。根据您应用的特定需求和其复杂度，两个方法中总有一个更为适用的方法。

4.1 方法

轮询

通过轮询的方法来获取流数据是一种比较直接的方法。在 Hello World 教程里我们采用

的就是这种方法。使用这种方法很简单，你只需要调用 `StreamReader` 的 `get_latest_frame` 函数，然后用模板函数 `get<T>` 来返回得到特定类型的帧。在这里“`T`”必须是一个有效的帧类型。函数 `get_latest_frame` 是阻塞式的，在下一帧数据到来之前，程序将被挂起。如果你想限制阻塞时间，可以给 `get_latest_frame` 函数传递一个 `timeout` 变量。下面通过轮询的方法读取一帧深度帧。

```
1.  astra::initialize();
2.
3.  astra::StreamSet streamSet;
4.  astra::StreamReader reader = streamSet.create_reader();
5.
6.  reader.stream<astra::DepthStream>().start();
7.
8.  astra::Frame frame = reader.get_latest_frame();
9.  const auto depthFrame = frame.get<astra::DepthFrame>();
10.
11. astra::terminate();
```

监听

通过监听的方法来获取数据需要事先做一些设置，这种方法允许开发者将帧的处理委托给一个或多个不同的类。SDK 提供了一个叫 `FrameListener` 的抽象类，该类只实现了一个叫 `FrameListener::on_frame_ready` 的函数。当数据帧到来的时候，该类会立即调用 `FrameListener::on_frame_ready` 并且将帧的引用作为参数传入。`DepthFrameListener` 从 `frame_listener` 继承的 listener 类示例如下：

```
1.  class DepthFrameListener : public astra::FrameListener
2.  {
3.     virtual void on_frame_ready(astra::StreamReader& reader,
4.                                 astra::Frame& frame) override
5.     {
6.        const astra::DepthFrame depthFrame = frame.get<astra::DepthFrame>();
7.
8.        if (depthFrame.is_valid())
9.        {
10.          // do all the things
11.       }
12.    }
13. };
```

定义了 listener 类，将这个 listener 实例化并且用 `StreamReader::add_listener` 函数将其添加到 `StreamReader` 之后才能使用，示例如下：

```
1.  astra::initialize();
2.
```

```
astra::StreamSet streamSet;
astra::StreamReader reader = streamSet.create_reader();

reader.stream<astra::DepthStream>().start();

DepthFrameListener listener;
reader.add_listener(listener);

while(true)
{
    astra_update();
}
```

在使用时，需要循环调用 `astra_update 函数，` 不断更新数据流，直到程序关闭或数据停止更新。添加了 listener，我们需要通过循环调用 `astra_update` 函数来触发事件。通过这种调用，SDK 可以检查是否有新的一帧数据到来，如果有，将调用 `DepthFrameListener::on_frame_ready` 函数，并且传入当前帧的引用。

4.2 示例

现在，我们利用监听方法来从 Astra 读取深度数据。通过后续例程分析，熟悉以下几方面：

- FrameListener 类的目的
- 如何定义 FrameListener
- 用 FrameListener 来处理深度流

准备

下载并解压最新的 Astra SDK，之前做过可跳过。拷贝下面的代码到你的 main.cpp 文件：

```
#include <astra/astra.hpp>
// for std::printf
#include <cstdio>

int main(int argc, char** argv)
{
    astra::initialize();

    astra::StreamSet streamSet;
    astra::StreamReader reader = streamSet.create_reader();

    reader.stream<astra::DepthStream>().start();

    // Your code will go here

```

```
16.     astra::terminate();
17.
18.     return 0;
19. }
```

- Line 7 - 初始化 Astra
- Line 9 - 构造一个 StreamSet
- Line 10 - 创建一个 StreamReader
- Line 12 - 启动一个深度流
- Line 16 - 关闭 Astra

监听流

在 Hello World 教程，我们通过循环调用 `StreamReader` 的 `get_latest_frame` 函数处理了多个数据帧构成的流。这种调用方法非常适合 Hello World 这种简单应用。但是，如果我们想要访问多个数据流，想要处理多个 StreamSet，或者在每个 StreamSet 里有多个 StreamReader ，怎么办？ 在这几种情况下，采用循环的方式会让问题变得非常复杂。
为了让这种问题变得简单， Astra 提供了一种事件机制。该机制定义并创建了 FrameListener。 FrameListener 有个成员函数 on_frame_ready 会在特定类型的一帧新数据到来的时候自动被调用。所以，相比于使用 StreamReader 的 get_latest_frame 函数，监听器（listener）自动并且及时地得到了最新的数据。步骤如下：

1) 定义监听 listener：我们需要定义一个 listener 类来实现接口 FrameListener。这个类使我们能够访问从 Astra 摄像头输出的实际数据。我们将通过 on_frame_ready 函数来得到这些帧。首先将下面的代码拷贝到你的 #include 和 main 函数之间:

```
1.  class DepthFrameListener : public astra::FrameListener
2.  {
3.  public:
4.      DepthFrameListener(int maxFramesToProcess)
5.          : maxFramesToProcess_(maxFramesToProcess)
6.      {}
7.
8.      bool is_finished() const { return isFinished_; }
9.
10. private:
11.     void on_frame_ready(astra::StreamReader& reader,
12.                         astra::Frame& frame) override
13.     {
14.         const astra::DepthFrame depthFrame = frame.get<astra::DepthFrame>();
15.
16.         if (depthFrame.is_valid())
17.         {
18.             print_depth_frame(depthFrame);
19.             ++framesProcessed_;
```

```
20.         }
21.
22.         isFinished_ = framesProcessed_ >= maxFramesToProcess_;
23.     }
24.
25.     void print_depth_frame(const astra::DepthFrame& depthFrame) const
26.     {
27.         const int frameIndex = depthFrame.frame_index();
28.         const short middleValue = get_middle_value(depthFrame);
29.
30.         std::printf("Depth frameIndex: %d value: %d \n", frameIndex, middleVa
    lue);
31.     }
32.
33.     short get_middle_value(const astra::DepthFrame& depthFrame) const
34.     {
35.         const int width = depthFrame.width();
36.         const int height = depthFrame.height();
37.
38.         const size_t middleIndex = ((width * (height / 2.f)) + (width / 2.f))
    ;
39.
40.         const short* frameData = depthFrame.data();
41.         const short middleValue = frameData[middleIndex];
42.
43.         return middleValue;
44.     }
45.
46.     bool isFinished_{false};
47.     int framesProcessed_{0};
48.     int maxFramesToProcess_{0};
49. };
50.
51. int main(int argc, char** argv)
52. {
```

- Line 4- 构造函数，其变量表示程序退出之前总共要处理的帧数。
- Line 8 - is_finished 用于判断是否我们已经处理了足够的帧数，该变量后面会用到。
- Line 14 - 读取深度数据帧
- Line 16 - 检查是否获取到有效的帧。
- Line 20 - 打印深度帧信息到命令行窗口
- Line 38 - 计算深度帧中心像素的索引
- Line 41 - 获取中心像素的值

本例中**唯一必须**的函数是 `on_frame_ready` 函数。

2) 添加监听 listener：定义好了 DepthFrameListener ，需要再 main 函数中增加一个监听，接下来我们在 main 函数中构造一个 listener 并将其添加到上一步创建的 StreamReader 里。

```
1.  int main(int argc, char** argv)
2.  {
3.      astra::initialize();
4.
5.      astra::StreamSet streamSet;
6.      astra::StreamReader reader = streamSet.create_reader();
7.
8.      reader.stream<astra::DepthStream>().start();
9.
10.     int maxFramesToProcess = 100;
11.     DepthFrameListener listener(maxFramesToProcess);
12.
13.     reader.add_listener(listener);
14.
15.     // More of your code will go here
16.
17.     reader.remove_listener(listener);
18.
19.     astra::terminate();
20.
21.     return 0;
22. }
```

- Line 10 - 构造一个会循环 100 次的 `DepthFrameListener`
- Line 13 - 将 listener 添加到 reader
- Line 17 - 将 listener 从 reader 移除

3) 更新监听 listener：我们已经将 Astra 和 `StreamSet` 运行起来了，而且我们还通过 StreamSet 的 `StreamReader` 来监听数据帧。但我们不知道数据帧什么时候会从 Astra 里面输出，所以我们需要在一个循环里反复调用 `astra_update` 去持续更新这些 listener。

```
1.  int main(int argc, char** argv)
2.  {
3.      astra::initialize();
4.
5.      astra::StreamSet streamSet;
6.      astra::StreamReader reader = streamSet.create_reader();
7.
```

```
8.     reader.stream<astra::DepthStream>().start();
9.
10.    const int maxFramesToProcess = 100;
11.    DepthFrameListener listener(maxFramesToProcess);
12.
13.    reader.add_listener(listener);
14.
15.    do {
16.       astra_update();
17.    } while (!listener.is_finished());
18.
19.    reader.remove_listener(listener);
20.
21.    astra::terminate();
22.
23.    return 0;
24. }
```

- Line 15-17- Astra 更新循环。

完成上述代码，您将对 SDK 应用就有了基本的了解。其他数据流的使用和上述例程类似。下面分享 SDK 核心数据流（BodyStream）获取的 main 函数，详情请您参考“SimpleBodyViewer-SFML\main.cpp”文件，另外该例程需要依赖 sfml 库。

```
1. int main(int argc, char** argv)
2. {
3.     astra::initialize();
4.
5.     sf::RenderWindow window(sf::VideoMode(1280, 960), "Simple Body Viewer");
6.
7. #ifdef _WIN32
8.     auto fullscreenStyle = sf::Style::None;
9. #else
10.     auto fullscreenStyle = sf::Style::Fullscreen;
11. #endif
12.
13.     const sf::VideoMode fullScreenMode = sf::VideoMode::getFullscreenModes()[0
    ];
14.     const sf::VideoMode windowedMode(1280, 960);
15.     bool isFullScreen = false;
16.
17.     astra::StreamSet sensor;
18.     astra::StreamReader reader = sensor.create_reader();
19.
```

```
20.     BodyVisualizer listener;
21.
22.     auto depthStream = configure_depth(reader);
23.     depthStream.start();
24.
25.     reader.stream<astra::BodyStream>().start();
26.     reader.add_listener(listener);
27.
28.     while (window.isOpen())
29.     {
30.         astra_update();
31.
32.         sf::Event event;
33.         while (window.pollEvent(event))
34.         {
35.             switch (event.type)
36.             {
37.             case sf::Event::Closed:
38.                 window.close();
39.                 break;
40.             case sf::Event::KeyPressed:
41.             {
42.                 if (event.key.code == sf::Keyboard::C && event.key.control)
43.                 {
44.                     window.close();
45.                 }
46.                 switch (event.key.code)
47.                 {
48.                 case sf::Keyboard::Escape:
49.                     window.close();
50.                     break;
51.                 case sf::Keyboard::F:
52.                     if (isFullScreen)
53.                     {
54.                         window.create(windowedMode, "Simple Body Viewer", sf::
    Style::Default);
55.                     }
56.                     else
57.                     {
58.                         window.create(fullScreenMode, "Simple Body Viewer", fu
    llscreenStyle);
59.                     }
60.                     isFullScreen = !isFullScreen;
61.                     break;
```

```
62.                 case sf::Keyboard::R:
63.                     depthStream.enable_registration(!depthStream.registration_
    enabled());
64.                     break;
65.                 case sf::Keyboard::M:
66.                     depthStream.enable_mirroring(!depthStream.mirroring_enable
    d());
67.                     break;
68.                 default:
69.                     break;
70.                 }
71.                 break;
72.             }
73.             default:
74.                 break;
75.             }
76.         }
77.
78.         // clear the window with black color
79.         window.clear(sf::Color::Black);
80.
81.         listener.draw_to(window);
82.         window.display();
83.     }
```

- Line7 – 定义了一个渲染窗口。
- Line24 – 设备了深度流的模式。
- Line27 – 得到 `BodyStream` 后开启该流。
- Line65 – 使深度图像对齐到彩色图像。
- Line68 – 深度图像镜像。

最后，此开发文档仅为开发者提供的初始文档，不够完善之处，敬请谅解！后续会持续更新，敬请期待！