OpenNI2 编程简要说明

目录

OpenNI 2.0 API（应用程序编程接口）提供了访问兼容 OpenNI2 标准的深度传感器的方法。使用该 API 接口使得一个应用程序能够初始化传感器并从设备接收深度流(Depth)，彩色流(RGB)和红外流(IR)，同时 OpenNI2 还提供了一个统一的接口通过深度传感器创建.oni 记录文件。

获取数据视频流（Streams）主要使用下面 4 个类：

1) **openni::OpenNI** – 提供一个静态的 API 入口点。同时提供访问设备的途径，设备相关的事件，版本及错误信息，被用来初始 OpenNI2 环境并建立与 Device 的连接。

2) **openni::Device** – 提供传感器设备连接的接口。 在被创建前要求 OpenNI 先进行初始化。Devices 提供访问 Streams 的途径。

3) **openni::VideoStream** –从一个特定的 Device 中获取数据流，可用来获取 VideoFrameRefs。

4) **openni::VideoFrameRef** – 从特定的 Stream 获取一帧数据。

除了这些主要的类以外，还有许多类和结构体用来保持一些特殊类型的数据，这些类主要服务于封装数据，在其他主要类的章节都有所提及。

配置类：

openni::DeviceInfo - 此类记录了设备的所有配置，包括设备名，URI，USB VID/PID 描述符和供应商。

openni::SensorInfo - 此类存储了传感器的所有配置，OpenNI2 兼容设备一般有 3 个传感器（IR,RGB，DEPTH）。RGB 并非必须。

openni::VideoMode - 此类存储了分辨率，帧率和像素格式。用于视频流的设置和查看设置，由视频帧引用查看这些设置，由传感器信息提供一个视频模式的列表。

openni::CameraSettings – 对彩色摄像头进行曝光，增益，白平衡等调节（需要厂商固件支持）

数据存储类/结构体：

openni::Version – 结构体，存储当前 OpenNI 版本

openni::RGB888Pixel - 结构体，存储彩色像素值。

openni::DepthPixel - 结构体，存储深度像素值。

openni::Grayscale16Pixel - 结构体，存储 IR 像素值。

openni::Array< T > - OpenNI 提供的简单数组类。包含图像数据。

openni::CoordinateConverter – 深度，彩色，世界坐标系之间的相互转换

openni::Recorder - 用来存储 OpenNI 视频流到文件的。

openni::DeviceConnectedListener – 监听设备连接

openni::openDeviceDisconnectedListener - 监听设备断开

openni::DeviceStateChangedListener - 监听设备状态改变

OpenNI 类

OpenNI2 程序入口类 openni::OpenNI，这个类提供了一个 API 的静态入口。可用来访问系统中所有的 OpenNI2 兼容设备并获取数据。例如通过轮询方式访问数据流或者生成设备连接和断开事件等。

设备的基础访问

OpenNI 类提供了 API 的静态入口，使用 OpenNI::initialize()方法。这个方法初始化所有的传感器驱动并且扫描系统中所有可用的传感器设备。所有使用 OpenNI 的应用程序在使用其他 API 之前都应该调用此方法。一旦初始化方法完成，将会开启设备(Device)对象，并使用这些对象来和真实的传感器硬件进行交互。OpenNI::enumerateDevices()方法会返回一个已经连接上系统的所有可用的传感器设备列表。

当应用程序准备退出时，必须调用 OpenNI::shutdown()方法来关闭 OpenNI 环境,否则会导致下次初始化失败。

视频流(Video Streams)的基础访问

视频流有两种访问方式，轮询和监听回调， 其中轮询的方式由 OpenNI 类提供的方法来完成。轮询方式采用 OpenNI:: waitForAnyStream ()方法实现，在设定的等待超时时间内会等待指定流数据，超时则退出。此方法的参数一对应流的列表。当方法调用时，就会锁定直到列表中的流有新的可用数据。然后返回一个状态码(status code)并指向是哪个流有可用数据了。

设备的事件驱动访问

OpenNI 类提供了一个在事件驱动方式中访问设备的框架。OpenNI 定义了 3 个事件：设备连接事件(onDeviceConnected)，设备断开事件(onDeviceDisconnected)，设备状态改变事件(onDeviceStateChanged)。设备连接事件是在一个 OpenNI 新设备连接时产生的，设备断开事件是在一个设备从系统中移除时产生的。设备状态改变事件是在设备的设置被改变时产生的。

可以用下列方法从事件处理列表中增添或者移除监听器类(Listener classes)：

```
Status OpenNI::addDeviceConnectedListener ( DeviceConnectedListener* pListener ) //增加设备连接事件监听器
Status OpenNI::addDeviceDisconnectedListener( DeviceDisconnectedListener* pListener ) //增加设备断开监听器
Status OpenNI::addDeviceStateChangedListener(DeviceStateChangedListener* pListener)//增加设备状态改变监听器
OpenNI::removeDeviceConnectedListener(DeviceConnectedListener* pListener) //移除设备连接事件监听器
OpenNI::removeDeviceDisconnectedListener(DeviceDisconnectedListener* pListener) //移除设备断开事件监听器
OpenNI::removeDeviceStateChangedListener(DeviceStateChangedListener* pListener)//移除设备状态改变监听器
```

通过注册监听器， Listener 会在相应的事件处理函数中操作当前设备，同时 3 个事件都提供了一个指针指向 OpenNI::DeviceInfo 对象。这个对象用来获取被事件提交的设备的细节和标识。此外，设备状态改变事件还提供了一个指针指向 DeviceState 对象，这个对象被用来查看设备新的状态信息。如下描述：

```
OpenNI::DeviceConnectedListener{ virtual void    onDeviceConnected (const DeviceInfo *)=0 }
OpenNI::DeviceDisconnectedListener { virtual void    onDeviceDisconnected (const DeviceInfo *)=0 }
OpenNI::DeviceStateChangedListener { virtual void    onDeviceStateChanged (const DeviceInfo *, DeviceState)=0 }
```

错误信息

在 SDK 中有许多方法都会返回一个类型为“Status”的值。当错误发生，Status 就会包含有一个记录或者显示给用户的代码。OpenNI::getExtendedError()方法会返回更多的关于错误的可读信息。

版本信息

API 的版本信息由 OpenNI::getVersion()方法来获取。这个方法返回应用程序目前使用的 API 的版本信息。

设备类（Device Class）

openni::Device 类提供了一个物理硬件设备的接口，也可以通过 ONI 记录文件提供了一个模拟硬件设备的接口。设备的基本目的是提供流。设备对象被用来连接和配置底层文件或者硬件设备，并从设备中创建流。在设备类能连接到物理硬件设备前，设备必须在物理上正确地连接到主机，并且驱动必须安装完毕，OpenNI2 已经内置市面上常见的 OpenNI 兼容设备的驱动。如果连接的是 ONI 文件，那要求在系统运行应用程序时 ONI 记录文件必须可用，而且应用程序有足够的权限去访问。当然，如前面所述，在连接设备前需确保 openni::OpenNI::initialize()方法被调用，这将会初始化驱动，使 API 知道设备连接了。

基本操作

Device::open()

此方法用来打开具体的物理硬件设备。open()方法有一个参数，设备的 URI（统一资源标识符），方法返回一个状态码指示是否成功。最简单的用法是用常量 openni::ANY_DEVICE 作为设备的 URI，用这个常量会打开系统中所有 OpenNI 兼容设备中的某一个，具体哪一个取决于设备插入顺序。这种方法对系统中只有一个此类设备非常有用。

如果有多个传感器连接到系统，那就需要先调用 OpenNI::enumerateDevices()来获取可用的设备列表。然后找到你要找的设备，通过调用 DeviceInfo::getUri()来获取 URI，用此方法的输出作为 Device::open()的参数，然后就能打开对应的设备了。

如果打开文件，那参数就是 ONI 文件的路径。

Device::close()

close()方法用来关闭硬件设备。按照惯例，所有打开的设备必须要关闭。这会断开设备与系统的连接，这样后面的应用程序连接它们就不会有什么麻烦了。

Device::isValid()

isValid()方法用来确定设备是否正确地和设备对象联系到了

从设备中获取信息

可以获取关于设备的基础信息。信息包括名称，供应商，uri，USB VID/PID（usb 的 id 有两部分，供应商 id 即 VID，产品 id 即 PID）。openni::DeviceInfo 类就提供了相关信息。每个可用信息都有 getter 方法，从给定的设备对象里获取 DeviceInfo 就调用 Device::getDeviceInfo()方法

一个设备一般由多个传感器组成。一个典型的 OpenNI2 兼容设备由 IR（红外）传感器，Color（彩色）传感器和 Depth（深度）传感器组成。Color 传感器不是必须的，有些设备就没有。每一个传感器对应 OpenNI2 中的一个流类型，只有传感器存在才能创建和打开相应的流。

可以从一个设备中得到传感器列表。Device::hasSensor()方法用来查询设备是否有特定的传感器。传感器类型如下：

SENSOR_IR – The IR video sensor 红外视频传感器

SENSOR_COLOR – The RGB-Color video sensor 彩色视频传感器

SENSOR_DEPTH – The depth video sensor 深度视频传感器

如果对应传感器可用，Device::getSensorInfo()方法就可以用来获取其信息。SensorInfo 提供了传感器类型，返回视频模式的数组的 getSupportedVideoModes 方法，单个的视频模式被封装进了 VideoMode 类。

特殊设备功能

对齐（Registration）

OpenNI2 设备的深度流和彩色流分别由红外传感器和彩色传感器产生，但是红外传感器和彩色传感器的中心并不重合，有如人的双眼，是有一定距离的，因此在两个传感器之间获取的数据会产生视差。为了让两个传感器看到的物体一致就需要对其中某一个流数据进行平移，旋转运算，使其与另一个流的视角保持一致。这个变化运算的过程我们就称之为对齐。一般为了保持彩色图不变形，我们会对深度数据进行运算使其向彩色图靠拢，也就是 D2C(depth to color)。

设备对象提供了 isImageRegistrationSupported()方法来测试已连接的设备是否支持对齐功能。如果支持，那 getImageRegistrationMode()能用来查询这个功能的状态，setImageRegistrationMode()就可以设置它。

openni::ImageRegistrationMode 枚举提供了以下值用来 set 或 get：

IMAGE_REGISTRATION_OFF – Hardware registration features aredisabled 硬件对齐功能被禁用

IMAGE_REGISTRATION_DEPTH_TO_IMAGE – The depth image is transformed tohave the same apparent vantage point as the RGB image 硬件对齐开启，对齐类型为 D2C，也就是 深度图像被变换叠加至彩图上

需要注意的是红外和彩色两个传感器的可视范围（FOV）一般并不能完全重叠。这就导致部分深度图不会在显示在结果中。在深度图有毛边的地方的可以看到“影子”或者“空洞”。

帧同步（FramSync)

当深度和彩图流都可用，那可能两个流会出现不同步，会导致帧 ID 不同，一些设备提供了使两个帧同步的功能，为了在确定的最大时间范围内分别从获取到两个帧，通常这个最大值都小于两帧间隔。这个功能就是帧同步。启用或禁用此功能用 setDepthColorSyncEnable()。

通用功能（General Capabilities)

一些设备会定义一些特别的设备功能，OpenNI 2.0 提供了 setProperty()和 getProperty()方法访问这些功能。setProperty 方法用一个属性 Id 和值来设置它。getProperty 方法则返回对应 Id 的属性的值。

文件设备（File Devices）

OpenNI 2.0 提供了记录流数据输出到文件的功能（记录文件是 ONI 文件，通常扩展名为.oni）。可以选择记录设备里的所有的流，或单独录取某个流。

打开文件设备和打开设备差不多，都是调用 Device::open()，只不过文件设备是用文件路径作为 URI。这个功能在运行调试时非常有用。通过录像功能，就使得同一输入能够用于多个**算法**，调试，性能比较。此功能能用于应用的自动化测试，或者是在一个项目中摄像头不足，测试代码就可以用记录文件来替代。最后，录像还可以使得技术支持通过查看用户摄像头的输出文件找出问题实现远程支持。

重放控制类(PlaybackControl class)用于访问文件设备的特殊功能。为了兼容处理文件和设备的代码，OpenNI 提供了 Device::isFile()方法，允许应用在尝试使用重放控制之前确定是文件还是设备。

记录器类（Recorder Class）

记录器类用来记录视频数据到 ONI 文件中。ONI 文件是 OpenNI 记录深度传感器输出的标准记录文件。包含了一到多个流的信息。还包含了设备的设置信息。所以可以用来通过文件实例化设备对象。

设置记录器

1) 调用默认构造函数构造一个记录器对象。这不同于其他类的实例化。
2) 调用 Recorder::creat()方法，参数为记录文件的文件名。创建和写入文件出错时返回一个错误码。
3) 供一个数据流进行记录。使用 Recorder::attach()方法来联系上给定的视频流。如果你记录多个流，那就多次调用来联系每个视频流，也就是逐个添加。

记录

视频流联系上后，调用 Recorder::start()方法开始记录。方法一调用，每帧数据都会被写入 ONI 文件。通过调用 Recorder::stop()方法来结束记录。调用 Recorder::destroy()方法来让文件存盘，释放所有内存。

重放

ONI 可以被许多 OpenNI 程序和公用程序进行重放。程序打开 ONI 文件都是作为文件设备打开的。重放控制封装在重放控制类里（PlaybackControl Class）。

重放控制类(PlaybackControl Class)

在处理记录文件（recorded file）时可能会有一系列操作。这些操作包括在流里查找，确定记录有多长，循环播放，改变重放速度。这个功能封装在 PlaybackControl 类中。

初始化（Initializing）

在使用 PlaybackControl 类之前，必须实例化和从文件初始化一个 Device 类。一旦一个可用的文件设备被创建，你就可以通过调用 Device::getPlaybackControl()获得其中的 PlaybackControl 对象。Device::IsFile()

方法被用来确定一个 Device 是否从一个文件创建的。

查找定位（seek）

提供了两个方法从一个记录中查找定位
PlaybackControl::seek()方法用一个视频流指针（VideoStream pointer）和帧 ID（frameID）作为输入，然后重放到指定的帧。如果一个记录中有多个流，那所有的流都会被设置到同样的位置上（定位的位置是指定流指定帧 ID 的位置）。
PlaybackControl::getNumberOfFrames()方法用来确定这个记录有多长。从根本上确定可用目标来查找很有用。此方法以一个流的指针作为输入，返回指定流所包含的帧的数目。需要注意的是同一记录的不同流可能不同的帧总数。因为帧不会一直都同步。

重放速度（Playback Speed）

此功能在测试一个有很大输入数据集合的算法时很有效，因为可以更快地得到结果。
PlaybackControl::setSpeed()方法使用一个浮点数作为输入。这个输入值作用于记录制作的多种速度。比如记录是一个 30fps 的流，然后输入值为 2.0，那么重放速度为 60fps，如果输入值为 0.5，那重放速度为 15fps。
设置速度为 0.0 会导致流播放速度为极限速度（系统能运行的最大速度）。设置速度为-1 会导致流变成手动读取，即播放会暂停，卡在这一帧，直到应用程序去去读取下一帧。将记录置为手动模式（manual mode），读取将会很紧密地循环，这就和设置速度为 0.0 很像。设置速度 0.0 是因为在用事件驱动模式进行数据读取时很有用。
PlaybackControl::getSpeed()方法会返回最近设置的速度值。

循环播放（Playback Looping）

API 提供了一个循环播放的方法。PlaybackControl::setRepeatEnabled()方法用来开关循环。设置值为 true 则重复读取，读完最后一帧又读第一帧。如果设置值为 false，那么在记录读取完后导致没有数据帧了。
PlaybackControl::getRepeatEnable()可用来获取当前的重复（repeat）值。

视频流类(VideoStream Class)

由设备类创建的视频流类封装了所有的数据流。这就使得你可以对数据流进行开始，停止，和配置。也被用来进行流一级的参数配置。

视频流的基础功能

创建和初始化视频流

调用视频流默认的构造函数会创建一个空的未初始化的视频流对象。在使用前，这个对象必须调用 VideoStream::create()进行初始化。而 create()方法要求一个已经初始化的设备对象。一旦创建，你就可以调用 VideoStream::start()方法来产生数据流。VideoStream::stop()方法则会停止产生数据流。

基于轮询的数据读取

一旦视频流创建完毕，就可以直接通过 VideoStream::readFrame()方法进行读取数据。如果有新的可用数据了，这个方法就会返回一个可以访问由视频流生成的最新的视频帧引用（VideoFrameRef）。如果没有新的帧可用，那就会锁定直到有新的帧可用。需要注意的是，如果非循环地从记录中读取，那么在追后一帧读取完毕后程序将永远卡死在此方法。

基于事件的数据读取

在事件驱动方式下（event driven manner）从视频流中读取数据是可以的。首先，需要创建一个类继承自 VideoStream::Listener 类，此类应该实现方法 onNewFrame()。一旦你创建了这个类，实例化了它，就可以通过 VideoStream::addListener()方法来添加监听器。当有新的帧到达，自定义的监听器类的 onNewFrame()方法就被调用。然后你就可以调用 readFrame()读取了。

获取关于视频流的信息

传感器信息（SensorInfo）和视频模式（VideoMode）
传感器信息和视频模式类可以一直追踪视频流的信息。
视频模式封装了视频流的帧率（frame rate），分辨率（resolution）和像素格式（pixel format）。传感器信息包含了产生视频流的传感器的类型和每个流的视频模式对象列表。通过遍历这个列表，那就能确定传感器生成的流的所有可能的模式。
使用 VideoStream::getSensorInfo 能够得到当前流的传感器信息对象

视野（Field of View）

此功能为确定创建了视频流的传感器的视野范围。使用 getHorizonFieldOfView()和 getVerticalFieldOfView()方法来确定视野。其返回的值是弧度。
有些厂商会设置像素最大最小值（Min and Max PixelValues）在深度流中，通常知道一个像素可能出现的最大值和最小值是很有用的。用 getMinPixelValue()和 getMaxPixleValue()方法就能获取这些信息。

配置视频流

视频模式（Video Mode）

可以设置给定流的帧率（frame rate），分辨率（resolution）和像素格式（pixel type）。设置这些就要用到 setVideoMode()方法。在此之前，你首先需要获取已配置视频流的传感器信息（SensorInfo），然后你才能选择一个可用的视频模式。

裁剪（Cropping）

如果传感器支持裁剪，视频流会提供方法来控制它。使用 VideoStream::isCroppingSupported()方法来确定是否支持。如果支持，使用 setCropping()来使能裁剪和设置裁剪的具体配置。ResetCropping()方法被用来再次

关闭裁剪。getCropping()方法用来获取当前的裁剪设置。

镜像（Mirroring）

镜像，顾名思义，就是使视频流所展现的看起来就像在镜子里一样。启用或禁用镜像，使用VideoStream::setMirroringEnable()方法。设置 true 为启用，设置 false 为禁用。可用 getMirroringEnable()来获取当前设置。

通用属性（General Properties）

在固件层，大多数的传感器设置都存储为地址/值的队（address/value pairs，就是一种键值对）。所以可以通过 setProperty 和 getProperty 方法直接操作。这些方法被 sdk 内部用来实现裁剪，镜像，等等。而它们通常不会被应用程序频繁地使用，因为大多数有用的属性都被更加友好的方法封装了。

视频帧引用类（VideoFrameRef Class）

视频帧引用类封装了从视频流读取的单个帧的所有的相关数据。是视频流用来返回每一个新的帧。它提供了访问包含了帧数据（元数据，工作所需的帧）基础数组。
视频帧引用对象是从 VideoStream::readFrame()方法获取的。
视频帧引用数据可以从红外摄像头，RGB 摄像头或者深度摄像头获取。getSensorType()方法用来确定产生此帧的传感器类型。它会返回传感器类型，一个枚举值。

访问帧数据

VideoFrameRef::getDate()方法返回一个直接指向帧数据的指针。类型为 void，这样每个像素的数据类型才能正确地索引。

元数据（metadata）

每个帧都会提供一系列的元数据来促进数据本身的工作。

数据裁剪（Cropping data）

数据帧引用知道视频流的裁剪设置，因此可以用来确定裁剪框的原点，裁剪框的大小和帧是否启用裁剪功能。实现方法如下：getCropOriginX()，getCropOriginY(),getCroppingEnable().若启用裁剪功能，则裁剪框大小等于帧大小。所以确定这些的方法和确定帧分辨率的方法是一样一样儿的（东北话）。

时间戳（TimeStamp）

每帧数据都有个时间戳。这个值是基于任意 0 值开始的微秒数。是不同于两帧之间时间差。同一设备的所有流用的都是同一 0 值，所以时间戳的差值可以用来比较不同流的帧。OpenNI 2.0 中，时间戳的 0 值是第一帧数据的到达时间。然而这无法保证每次都一样，所以程序代码应该使用时间戳增量。时间戳的值本身不应该用作一种绝对的时间指向。

帧索引（FrameIndex）

除了时间戳，帧还提供了连续的帧索引号。这在确定已知的两帧之间有多少帧很有用。如果流使用了同步方法 Device::setColorDepthSync()，那相应的帧的帧号应该就是一致的。

如果没有同步，那帧号将不一定匹配。这种情况下，用时间戳来确定相关帧的位置更有效。

视频模式（Video Modes）

VideoFrameRef::getVideoMode()用来确定生成当前帧的传感器的视频模式。信息包括像素格式，分辨率，帧率。

数据大小（Data Size）

getDataSize()用来确定图像数组中所有数据的大小。在分配存储帧的缓冲区时或者确定帧数时很有用。需要注意的是这是整个数组的数据大小。用 VideoMode::getPixelFormat()来确定每个数组元素的大小。

图像分辨率（Image Resolution）

getHeight() 和 getWidth() 方法来确定帧的分辨率很容易。这个数据数据也可以通过 VideoFrameRef::getVideoMode().getResolutionX()和 VideoFrameRef::getVideoMode().getResolutionY()来获取，但不适合频繁调用，因为太低效了。

数据有效性（Data Validity）

VideoFrameRef::isValid()方法确定当前视频帧引用是否是有效数据。
在视频帧引用初始化结构体和第一次数据加载之间调用会返回 false。

传感器类型（Sensor Type）

确定产生数据帧的传感器类型用 getSensorType()。方法返回传感器类型，为以下的枚举值：
SENSOR_IR– for an image taken with an IR camera 红外传感器
SENSOR_COLOR – for an image taken with an RGB camera 彩图传感器
SENSOR_DEPTH – for an image taken with a depth sensor 深度传感器

数组跨度（Array Stride）

包含帧的数组跨度可以用 getStrideBytes()来获取。它将返回数组每行数据的大小，单位字节 byte。主要用于索引二维图像数据。