Capítulo

3

Desenvolvimento de Ambientes Virtuais Interativos usando Java e Kinect

Almerindo Nascimento Rehem Neto, Celso Alberto Saibel Santos, Lucas Aragão de Carvalho e Clebeson Canuto dos Santos

Abstract

This course has a mission to show the new concepts of the framework OpenNI – Open Natural Interaction – version 2.2 - Open Natural Interaction and the development of applications using hardware capture depth, such as the Microsoft Kinect sensors and Carmine 1.08 PrimeSense's so adherent the standards of OpenNI.

Resumo

Este curso tem a missão de mostrar os novos conceitos do framework OpenNI – Open Natural Interaction – versão 2.2 e o desenvolvimento de aplicações que utilizem hardware de captura de profundidade, como os sensores Kinect da Microsoft e o Carmine 1.08 da Primesense, de forma adepta aos padrões do OpenNI.

3.1. Introdução

A evolução das tecnologias para captura de interações dos usuários, muitas vezes de forma ubíqua, criou novos paradigmas para a concepção de interfaces para aplicações interativas. O *Wiimote* [Nintendo 2009] e o *Kinect* [Microsoft 2010, OpenKinect 2011, PrimeSense 2011] são os dois principais ícones comerciais que caracterizam essa nova forma de se pensar em interfaces, facilitando a comunicação entre usuário e computador por meio de interações naturais.

O *Wiimote* (ou *Wii Remote*), lançado em 2006, baseado em acelerômetros, é o dispositivo principal de interação do console *Nintendo Wii*. O *Kinect*, fabricado pela *PrimeSence*, é o dispositivo principal de interação com o console *Xbox* 360, lançado em 2010 pela *Microsoft*, que trouxe como grande inovação uma interface baseada no reconhecimento de gestos. Apesar do foco inicial na área de jogos interativos,

Este material de suporte do minicurso é uma reprodução autorizada pela SBC do Capítulo 3 do Livro "Tutoriais do X Simpósio Brasileiro de Sistemas Colaborativos e XII Simpósio Brasileiro sobre Fatores Humanos em Sistemas Computacionais" **apresentado no SBSC IHC 2013.**

pesquisadores e desenvolvedores têm utilizado estas plataformas como base para a construção de interfaces naturais para aplicações interativas [Shape Quest 2010, Saffer 2009, Shiratori and Hodgins 2008, Nakra et al 2009].

No caso do *Kinect*, em particular, menos de uma semana após seu lançamento oficial, a empresa *Adafruit* lançou o desafio "*Open KinectChallenge*", que oferecia U$3.000 para o primeiro desenvolvedor de um driver de código aberto para o novo dispositivo [Adafruit 2011]. Com o desafio e a recompensa lançados, não demorou para que a primeira implementação *opensource* de drivers para o *Kinect* (a biblioteca *libfreeneck* [OpenKinect 2011]) fosse apresentada, abrindo o caminho para o desenvolvimento de aplicações com novas formas de interação fora da área de jogos.

3.2. Interação Natural

Rehem et al. [Rehem 2011] consideram que a interação natural procura formas de tornar uma Interface o mais natural possível, com objetivo de fazer com que as pessoas manipulem uma máquina sem perceber qual(is) artefato(s) é (são) utilizado(s) e sem necessariamente aprender um novo vocabulário correspondente ao conjunto de instruções desta interface. Uma das principais motivações da área de Interação Humano-Computador (IHC) é melhorar a adaptação dos sistemas computacionais às necessidades do usuário [Hewett et al 1996]. É exatamente neste sentido que caminham as chamadas interfaces de interação natural .

Para Heckel [Heckel 1993], a interface deve procurar reproduzir a linguagem natural do usuário de computador ou da máquina, tentando prover o uso de palavras e expressões conhecidas, respeitando o vocabulário do usuário. Esta tentativa terá seus reflexos na redução do uso da memória e na diminuição do esforço cognitivo do usuário e, consequentemente, influirá numa interação mais simples, agradável e natural.

Valli [Valli 2007] define a interação natural em termos experimentais: as pessoas naturalmente se comunicam através de gestos, expressões, movimentos, além de explorarem o mundo real através da observação e manipulação de objetos físicos. E como uma tendência cada vez mais forte, as pessoas querem interagir com a tecnologia da mesma forma como lidam com o mundo real no cotidiano. A criação de novos paradigmas de interação e padrões alternativos de mídia que explorem as novas capacidades dos sensores presentes nas máquinas, e ainda, respeitem a espontaneidade inerente ao modo como o ser humano descobre e interage com o mundo é o principal desafio da construção de novas interfaces para interação. A tendência de quebra do paradigma tradicional das interfaces é evidenciada através de diversos experimentos usando novas propostas de interação surgindo cotidianamente na Web [Kinecthacks 2010] e do interesse de empresas como a Microsoft, que começa a dar importância na exploração dessas novas formas de interação [Toyama 1998]. Desta forma, os usuários têm sido cada vez mais direcionados a um cenário de computação ubíqua no qual os serviços digitais (ou computacionais) são disponíveis e acessíveis a partir de qualquer lugar, permitindo que as pessoas interajam de forma bastante natural com esses serviços [Cordeiro 2009], [Gregory et al 2000], [Weiser and Brown 1996].

Saffer [Saffer 2009] considera um gesto na interação natural usuário computador como qualquer movimento físico que um sistema digital possa reconhecer e ao qual possa responder. Um som, um inclinar de cabeça ou até mesmo uma piscada de olho podem ser considerados gestos. Um dos problemas enfrentados neste tipo de

reconhecimento é estabelecer o contexto das interações de forma a facilitar o reconhecimento dos gestos e a possibilitar a construção de interfaces naturais para cada tipo de cenário de utilização.

Gallud et al. [Gallud et al 2010] definem a comunicação não verbal como aquela que utiliza pistas e sinais sem estrutura sintática e separa essa comunicação em três tipos:

- Paralinguagem: Refere-se aos elementos que acompanham expressão linguística, consistindo de pistas e sinais que sugerem uma interpretação diferente do que está sendo dito, percebe-se pela intensidade e/ou altura da voz do emissor bem como chorar, rir, controle respiratório, etc.
- Linguagem corporal: Caracteriza-se pela utilização de gestos, porém seguida de características distinguíveis utilizando o tronco, extremidade e cabeça também pela proximidade entre emissor e receptor, expressões de estado como felicidade, honestidade, cumplicidade, etc.
- Linguagem sonora: Acompanha os gestos e caracteriza-se pela interpretação qualquer som que expresse emoção, contextualiza uma cena e, até mesmo, o silêncio pode ter significado em uma mensagem.

A detecção e interpretação de tantos sinais para uma resposta adequada dependem de análise e contextualização extremamente rápidas dos sinais recebidos. Assim, inúmeros fatores podem influenciar o entendimento correta de uma mensagem na comunicação usuário computador.

Se for possível utilizar linguagem corporal e falada como intervenções de usuário, abre-se a possibilidade de se criarem interfaces através das quais o usuário e a máquina podem se comunicar de forma mais natural. Por outro lado, a criação de interfaces baseadas na comunicação por gestos e voz traz consigo um novo tipo de problema: Quais gestos de interação e comandos associados devem ser utilizados para realizar uma determinada tarefa? Como facilitar o desenvolvimento de aplicações que necessitem utilizar gestos a principal interação com o sistema? Alguns trabalhos focam nestes problemas, ainda em aberto, para tentar auxiliar os desenvolvedores na redução de esforço e tempo de trabalho, como é o caso do *Touch The Air Framework* [Rehem, Santos and Carvalho 2013], que utiliza como base o *framework* desenvolvido pela *PrimeSense* [OpenNI 2011].

A proposta deste capítulo é apresentar uma forma padronizada de desenvolver módulos de componentes que somados viabilizariam a construção de ambientes virtuais interativos baseados na comunicação por gestos. Através da abordagem apresentada, os componentes gerados podem ser compartilhados e reutilizados para aplicações de diferentes finalidades, além de permitirem a criação de um ecossistema de bibliotecas e aplicações para auxiliar os desenvolvedores da área em questão.

3.3. Os Novos Dispositivos de Interação Humano-Computador

O *Wiimote* e o *Playstation Move* se parecem muito em termos de funcionamento. Eles são baseados essencialmente em emissão e detecção de luz infravermelha e uma câmera de RGB tradicional. A luz infravermelha é detectada pela câmera RGB e assim, é possível detectar a posição do ponto de referência: no caso do *Wiimote* esse ponto é o próprio controle e no caso do Playstation Move, é a peça com uma extremidade

luminosa que melhora a precisão que é dada pela câmera RGB. Este processo pode ser feito de forma 2D resultando numa posição na tela, exatamente como um ponteiro de mouse, ou de forma 3D, onde para detectar a distância do controle até a tela é utilizada uma triangulação. Além disso, quando há um acelerômetro disponível ao conjunto de sensores, além de posição e distância, as inclinações em qualquer eixo também podem ser detectadas. As inclinações definem os chamados graus de liberdade (*Degrees Of Freedom* – DOF) para os movimentos do controle (ver Figura 3.1).

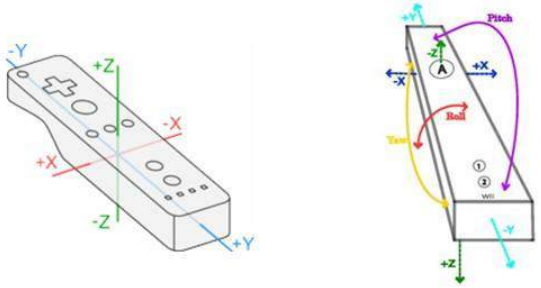

Figura 3.1.Eixos de referência e (b) os seis graus de liberdade do Wiimote. Fontes: [Wiiremote 2010],[Twiky 2010]

De fato, estas técnicas já eram conhecidas desde a década de 80 quando a Nintendo lançou o controle *Power Glove* (Figura 3.2) que funcionava de maneira muito similar aos dispositivos *Wiimote* e *Playstation Move*, com a diferença que, ao invés do infravermelho, utilizavam-se sensores de ultrassom. Estes sensores permitiam medir o tempo entre a reflexão das ondas emitidas por um aparelho posicionado na TV e o material da luva para determinar a posição da luva.

Figura 3.2. Nintendo Power Glove. Fonte: [BlaGames 2008]

A principal inovação trazida pelo *Kinect* é a geração de um mapa de profundidade, que possibilita ao dispositivo entregar informações 3D da cena completa, incluindo o jogador/ator e não somente informações do controle. Para se conseguir este resultado, utiliza-se o conceito de luz estruturada: uma luz com um padrão de projeção bem definido, que permite o cálculo das informações da cena partindo da combinação entre os padrões do que é projetado e da distorção da projeção em função dos obstáculos encontrados pela luz (ver Figura 3.3).

Figura 3.3. Luz estruturada (a) usada no mapeamento da face [18] e (b) Nuvem de pontos de luz emitidos a partir do sensor IR do Kinect.

Para conseguir esses resultados, o Kinect possui um sofisticado algoritmo de processamento paralelo, embarcado no chip (*SoC – System on Chip*) desenvolvido pela *PrimeSense*, necessário para extrair o mapa de profundidade [PrimeSense 2011]. Para ter maior precisão nos dados dos sensores, as imagens são alinhadas pixel a pixel, ou seja, cada pixel de imagem colorida é alinhado a um pixel da imagem de profundidade. Além disso, o chip *SoC* sincroniza (no tempo) todas as informações dos sensores (profundidade, cores e áudio) e as entrega através do protocolo USB 2.0 [Crawford 2010],[PrimeSense 2011].

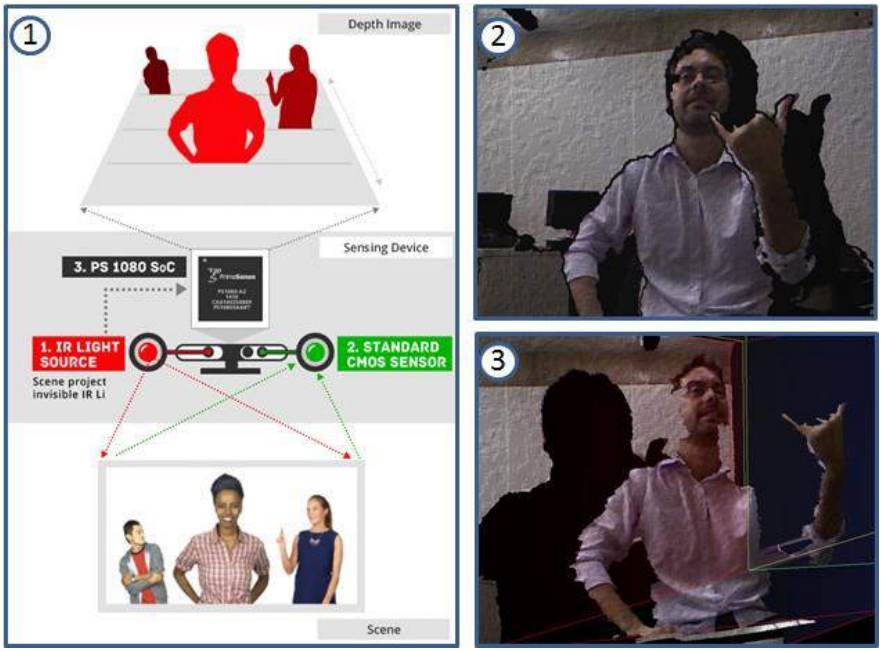

Figura 3.4. (1) Exemplo do diagrama do sensor da PrimeSense [PrimeSense 2011]. (2) e (3) são imagens RGB com seus pixels associados à profundidade.

A Figura 3.4 mostra um diagrama de funcionamento de sensores 3D baseados no chip da *PrimeSense* e exemplificados em três passos. O primeiro e o segundo passo, ilustram os sensores de IR e RGB tendo seus dados capturados e processados pelo chip *SoC*, que faz todo o processamento necessário para definir e entregar o mapa de profundidade ilustrado, como resultado final, o segundo e o terceiro itens da figura.

O *Kinect* dispõe de vários recursos (som, imagem, profundidade, infravermelho, motor de movimentação) com um alto índice de precisão e sincronismo em um único dispositivo. Estes recursos oferecem diversas possibilidades de interação entre os usuários e serviços e aplicações computacionais. Contudo, as formas e tecnologias para acesso a esses recursos podem ser as mais variadas possíveis. Na tentativa de se resolver o impasse, a *PrimeSense* disponibilizou o *framework OpenNI* (*Open Natural Inrterface*) e o módulo NiTE (*Natural Integration Middleware*) [PrimeSense 2011]. As tecnologias, tratadas na sequência do texto, permitem construir interfaces de aplicações sofisticadas, em um nível mais alto de abstração, tendo como base a combinação dos recursos disponíveis em sensores 3D e sem se restringir somente à plataforma *Kinect*.

3.4. O Framework *OpenNI*

O termo *OpenNI* designa uma instituição sem fins lucrativos e também uma marca registrada da *PrimeSense* [PrimeSense 2011]. A *OpenNI* busca promover a

interoperabilidade e compatibilidade entre aplicações, *middleware* e dispositivos de interação natural, atuando na verificação e certificação da conformidade das soluções de diferentes fabricantes e desenvolvedores aos padrões definidos pelo *framework OpenNI*.

O principal propósito da instituição *OpenNI* é especificar uma API padrão para a comunicação, tanto para dispositivos (sensores), quanto para aplicações e *middlewares*. Com isso, busca-se quebrar a dependência entre os sensores e os *middlewares*, além de permitir o desenvolvimento e portabilidade de aplicações para diferentes módulos de *middlewares* com menor esforço adicional (conceito "*write once, deploy everywhere*"). Mais ainda, a API fornece: (i) o acesso direto aos dados brutos dos sensores aos desenvolvedores e (ii) a possibilidade de se construir dispositivos que funcionem em qualquer aplicação compatível com o padrão proposto aos fabricantes.

O *framework OpenNI* habilita aos desenvolvedores acompanharem as cenas do mundo real (cenas em 3D), utilizando tipos de dados processados a partir da captura de um sensor de entrada (por exemplo, a representação do contorno de um corpo humano ou a localização de uma mão numa cena). Graças ao padrão *OpenNI* essas cenas são construídas com um alto nível de desacoplamento, isto é, podem ser construídas de forma independente dos fabricantes de sensores ou de um determinado *middleware*. A Figura 3.5 oferece uma visão abrangente das camadas da arquitetura *OpenNI* e dos conceitos associados ao padrão. Estas camadas são divididas em *Top*, *Middle* e *Botton*. Elas podem ser descritas da seguinte forma:

- ***Top***: representa os *softwares* que implementam aplicações de Interação Natural seguindo o padrão *OpenNI*;
- ***Middle***: representa o próprio *framework OpenNI* e oferece interfaces de comunicação para interagir tanto com os *middlewares*, quanto com os dispositivos (sensores);
- ***Bottom***: ilustra a camada de mais baixo nível, onde se situam os dispositivos de hardware responsáveis por capturar elementos visuais e auditivos da cena.

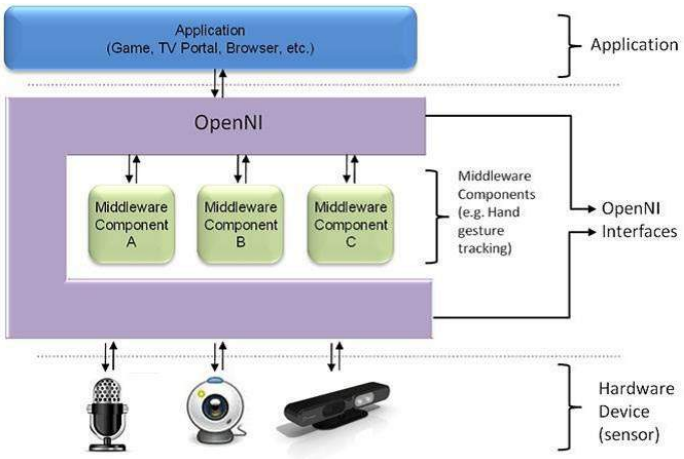

Figura 3.5. Abstração e componentização do *OpenNI*. Fonte: [PrimeSense 2011]

Pelo exposto até aqui, o conceito de *framework* está associado a uma camada de abstração que fornece interfaces para os componentes de softwares (*middlewares*) e para os dispositivos físicos através de uma API. Esta API habilita registros de múltiplos componentes (módulos) no *framework* que são responsáveis por produzir ou processar dados dos sensores físicos de forma flexível. Atualmente, os componentes suportados

pela arquitetura *OpenNI* são agrupados em dois tipos, os módulos de sensores e *middlewares*, descritos como se segue:

Módulos de sensores:

1. **Câmera RGB**: dispositivo responsável por gerar imagens coloridas;
2. **Sensor 3D**: dispositivo capaz de gerar o mapa de profundidade;
3. **IR câmera**: dispositivo de infravermelho;
4. **Dispositivo de áudio**: um ou mais microfones.

Middlewares:

1. **Análise de corpo**: componente de software capaz de processar os dados de entrada dos sensores e retornar informações relacionadas ao corpo humano. Como por exemplo, uma estrutura de dados que descreva as articulações, orientação e o centro de massa de uma pessoa na cena;
2. **Análise de mão**: componente capaz de processar dados dos sensores, reconhecer e retornar uma coordenada de localização do centro de uma mão em cena;
3. **Detecção de gestos**: componente capaz de identificar gestos predefinidos e alertar a aplicação todas as vezes que houver ocorrência desses gestos;
4. **Análise de cena**: componente capaz de analisar os dados da cena e produzir informações como: (i) a separação dos planos foreground e do background; (ii) as coordenadas do chão e (iii) a identificação de atores em cena.

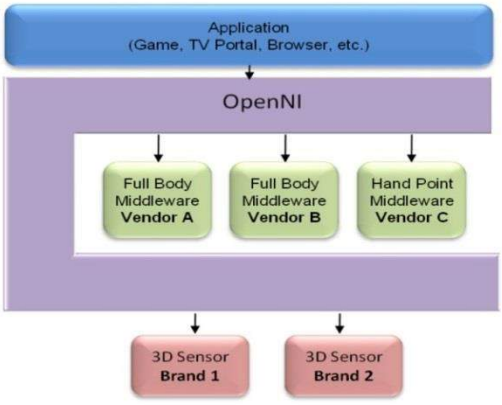

Figura 3.6. Exemplo de uso da arquitetura onde são registrados cinco módulos simultâneos em uma única instância do *OpenNI*. Fonte: [PrimeSense 2011]

A Figura 3.6 ilustra um exemplo de aplicação dos componentes da arquitetura para construir um cenário onde cinco módulos são registrados para trabalhar em uma única instância do *framework OpenNI*. Dois dos módulos registrados são sensores 3D conectados fisicamente ao *host*. Os outros três, são componentes de *middleware*, sendo que os dois primeiros módulos retornam dados relacionados ao corpo humano (*full body*) e o último, dados relacionado à mão (*hand point*).

Atualmente, o *framework* está passando por um redesenho de sua arquitetura. A versão estável anterior do *OpenNI* (versão 1.5), embora tenha impulsionado a comunidade científica e o desenvolvimento de diversas aplicações, ao ganhar popularidade, sofreu diversas críticas pela complexidade do seu uso. A versão 1.5, além de possuir conceitos com um nível de complexidade que dificultam o entendimento do

funcionamento do *framework*, também exigia o uso de muitas linhas de código para a concepção de tarefas simples. Após diversas sugestões de melhorias no *framework* e visando atender às necessidades da comunidade científica e dos desenvolvedores, a *PrimeSense*, principal mantenedora da *OpenNI*, iniciou a concepção de uma nova arquitetura (denominada *OpenNI* 2.x), em tese, mais simples e fácil de se utilizar.

Os próximos tópicos irão detalhar a API *OpenNI* em sua versão 2.2 (a mais atual no momento). É importante ressaltar que, devido ao seu lançamento recente, no momento ainda não é possível encontrar uma documentação detalhada sobre a nova arquitetura *OpenNI* e sua integração com a linguagem Java. Por consequência, a concepção deste texto exigiu estudos e análises cautelosas de todas as classes (e métodos associados) da versão 2.2 do *framework OpenNI*. Como contribuição, o texto traz uma série de comentários e explicações que buscam facilitar o entendimento das classes e dos principais métodos da nova versão do *framework*, além de apresentar exemplos práticos que auxiliam a instalação e demonstração do seu funcionamento.

3.4.1 Entendendo a estrutura da API Java *OpenNI*

Este tópico trata das classes (e de seus respectivos métodos) que fazem parte do *wrapper* Java, disponível na versão 2.2 do *OpenNI*, além de detalhar o funcionamento desses métodos, com o objetivo de possibilitar um melhor entendimento e utilização dos recursos disponibilizados.

Tabela 3.1.Relação das classes do *wrapper Java OpenNI2.2*

CLASSES	CLASSE DE APOIO	DESCRIÇÃO
CoordinateConverter.java	SIM	Usada para converter pontos entre sistemas de coordenadas diferentes.
CropArea.java	SIM	Classe de apoio que serve para encapsular recortes de informações.
Device.java	**NÃO**	**Responsável por toda a comunicação da aplicação com o dispositivo de Hardware.**
DeviceInfo.java	SIM	Classe de apoio que guarda atributos de determinado dispositivo.
ImageRegistrationMode.java	SIM	Classe de apoio que fornece nomes de sequencia para os valores dos códigos dos tipos de registros de Imagem.
NativeMethods.java	SIM	Classe que encapsula toda a API em C/C++ através do JNI para que se possa utilizá-la desde o JAVA.
OpenNI.java	**NÃO**	**Classe principal que faz a configuração do SDK e abri/fecha a comunicação com os Drivers.**
OutArg.java	SIM	Classe de apoio que define tipos de dados para a classe NativeMethods.
PixelFormat.java	SIM	Classe de apoio que define os tipos de pixels a ser utilizados.
PlaybackControl.java	NÃO	**Classe que gerencia reprodução de arquivos .ONI.**
Point2D.java	SIM	Classe de apoio para manipulação de pontos em 2D.
Point3D.java	SIM	Classe de apoio para manipulação de pontos em 3D.
Recorder.java	**NÃO**	**Classe responsável pela gravação em disco das capturas de uma aplicação (grava arquivos .ONI).**
SensorInfo.java	SIM	Classe de apoio que guarda informações de in determinado tipo de sensor. Informações como o tipo do sensor e o modo de vídeo utilizado.
SensorType.java	SIM	classe de apoio que armazena os três tipos de sensores ser utilizados (RGB, Infravermelho e de Profundidade).
Version.java	SIM	Classe de apoio que retorna qual a versão do OpenNI está sendo utilizada.
VideoFrameRef.java	**SIM**	**Classe que armazena os dados e atributos de cada frame que é capturado pelo dispositivo.**
VideoMode.java	**NÃO**	**Classe que configura o modo de vídeo a ser utilizado**
VideoStream.java	**NÃO**	**Classe responsável pela transmissão das capturas de frames feita pelo dispositivo.**

O *wrapper Java* do *OpenNI* 2.2 é formado de 19 classes das quais 7 são, consideradas como principais pelos autores deste texto. Este agrupamento se deve ao fato destas classes serem as principais responsáveis pelo gerenciamento dos dispositivos e pela captura e gravação dos dados das cenas. As demais classes são consideradas como de apoio, uma vez que são voltadas, de alguma forma, ao auxílio das chamadas classes principais. A Tabela 3.1 faz uma breve descrição do objetivo de cada classe Java e destaca em negrito as classes que foram categorizadas como sendo mais relevantes para os desenvolvedores que utilizam o *wrapper Java OpenNI 2.2*.

3.4.2 Detalhando as Classes do Wrapper Java OpenNI 2.2

Nesta seção serão explicadas todas as classes citadas na Tabela 3.1, mostrados os principais métodos associados, além de alguns códigos de exemplo de uso. É importante ressaltar que todas as documentações disponíveis no site oficial da *OpenNI* [OpenNI 2011] estão voltadas para a API original em C++. A documentação aqui apresentada é fruto de um trabalho árduo de engenharia reversa e análise dos códigos disponibilizados no site.

a) A classe Device

A classe Device é a responsável por estabelecer a conexão com um único dispositivo físico. Ela possibilita também simular um dispositivo de hardware, caso ele não esteja conectado ao computador, através da leitura das informações armazenadas em arquivo com extensão ".oni" (seção k)). Este último arquivo deve ser previamente criado e possuir informações das capturas de um dispositivo físico para que se consiga fornecer os dados necessários à simulação do *Kinect*, por exemplo.

Tabela 3.1.Métodos da Classe Device

Métodos	Descrição
open() e open(String uri)	Para abrir conexão com algum dispositivo é utilizado o método, open(). A depender dos parâmetros passados é possível comunicar com um dispositivo físico (hardware) ou simular um através de conexão com arquivo em disco na extensão .oni. Para conectar-se a um dispositivo físico podemos utilizar o construtor default ou passar o caminho completo do dispositivo, que poderia ser a URI da porta USB onde ele está.
close()	É responsável por fechar a conexão com o dispositivo. Esse método deve ser chamado sempre que se haja terminada a aplicação ou quando o dispositivo não vá mais ser utilizado.
getDeviceInfo()	Retorna as informações do dispositivo que se esta utilizando. Esse retorno vem na forma de um objeto do tipo DeviceInfo.
getHandle()	Utilizado pelo JNI para identificar o endereço de memória C/C++ onde se encontra este objeto.
getImageRegistrationMode()	Retorna um objeto do tipo ImageRegistrationMode.
isFile()	Retorna um boolean indicando true se o Device foi criado através de um arquivo ou false se é uma comunicação direta com o hardware.
getPlaycackControl()	Esse método permite controlar a execução da simulação de um dispositivo físico. Caso o dispositivo físico seja realmente um hardware ao invés de simulação através de arquivo .oni, este método irá retornar sempre um objeto nulo.
getSensorInfo(SensorType type)	Retorna o objeto SensorInfo que contem informações associadas ao tipo do sensor passado como parâmetro (SensorType).
hasSensor(SensorType type)	Esse método permite saber se o dispositivo conectado possui um determinado tipo de sensor passado como parâmetro, por exemplo RGB,IR ou DEPHT.
isImageRegistrationModeSupported (ImageRegistrationMode mode)	Indica se o dispositivo suporta ou não o registro de um determinado tipo de imagem o qual está sendo passado como parâmetro.
setDepthColorSyncEnabled (boolean bln)	Permite capturar,de forma síncrona, os sensores DEPHT e RGB.
setImageRegistrationMode (ImageRegistrationMode mode)	Configura o Device para operar com um determinado tipo de imagem ImageRegistrationMode.

Um dispositivo físico tem como objetivo capturar *streams* a todo o momento, sendo que as capturas podem ser feitas por tipos distintos de sensores. Assim, o objetivo da classe *Device* é comunicar-se com esse dispositivo físico e capturar a informação "crua" de cada um de seus sensores. Após capturar essas informações, outra classe poderá armazená-las num arquivo para que a aplicação possa utilizá-las posteriormente.

Para que a classe *Device* possa se comunicar com um dispositivo físico, é necessário que este esteja conectado ao computador e que todos os seus *drivers* de comunicação sejam corretamente instalados. Caso algo não esteja sendo feito de maneira correta, pode ser que a classe *Device* não encontre o dispositivo e então será lançada a mensagem de exceção "*No Device is Connected*".

A Tabela 3.1 ilustra todos os métodos da classe Device, bem como as explicações dos conceitos de cada um deles. A seguir, demonstraremos como criar e utilizar um objeto do tipo Device para conectar com os dispositivos.

O Método *open*() – construindo um *Device*

O método *open*() é permite iniciar uma conexão com algum dispositivo. A depender dos parâmetros passados é possível se comunicar com um dispositivo físico (*hardware*) ou simular um através da conexão com arquivo de extensão ".oni" em disco. Para conectar-se a um dispositivo físico, pode-se utilizar o construtor *default* ou passar, como parâmetro, o caminho completo do dispositivo, que poderia ser a URI da porta USB onde ele está conectado. Os exemplos abaixo ilustram o uso do método:

Conecta-se com o primeiro dispositivo físico encontrado.
Device.open();

Conecta-se com o dispositivo que estiver conectado através da URI passada por parâmetro.
Device.open(String URI);

Caso existam mais de um dispositivo conectado ao computador e o desenvolvedor não saiba qual dos dispositivos é o correto, pode-se utilizar o método *enumeratDevices*(), da classe *OpenNI* (seção f)), para se obter a lista de dispositivos conectados no momento. Essa lista contém objetos do tipo *DeviceInfo* (Tabela 3.1), que são responsáveis por guardar informações dos dispositivos. Cada objeto *DeviceInfo* possui um método chamado *getURI*() que retorna o caminho completo do dispositivo em questão, facilitando assim o trabalho do desenvolvedor.

O trecho de código da Figura 3.7 ilustra a utilização de uma conexão com o primeiro dispositivo físico, utilizando os métodos citados até o momento.

```
17          //Obtendo uma lista com os dispositivos conectados ao sistema
18          List<DeviceInfo> devicesInfo= OpenNI.enumerateDevices();
19          //Verificando se existe dispositivo conecta ao sistema
20          if (devicesInfo.isEmpty()) {
21              JOptionPane.showMessageDialog(null, "No device is connected",
22                                            "Error", JOptionPane.ERROR_MESSAGE);
23              return;
24          }
25          //Referenciando o DeviceInfo da posição "0" da lista "devicesInfo".
26          DeviceInfo deviceInfo=devicesInfo.get(0);
27          //Criando um Device a partir da URI obtida de "deviceInfo"
28          Device device=Device.open(deviceInfo.getUri());
```

Figura 3.7. Trecho de código com a utilização de um objeto do tipo Device

b) A classe PlaybackControl

A facilidade de se reproduzir dados previamente capturados permite que os desenvolvedores possam trabalhar simulando o ambiente desejado, o que é muito interessante para reprodução e depuração durante o desenvolvimento. Isto possibilita ainda reduzir custos, pois não é necessário adquirir um sensor 3D para cada desenvolvedor construindo um ambiente interativo. Essa característica de simulação também permite que se realize algum tipo de tratamento dos dados capturados antes de apresentá-los, ou mesmo, a visualização destas informações em um momento posterior, como no caso de sistemas de câmeras de segurança.

A classe *PlaybackControl* controla a reprodução de um arquivo ".oni" (seção k)) e serve para simular, a partir deste arquivo, um dispositivo físico. Com ela é possível controlar a velocidade de reprodução, posicionar a reprodução em um determinado

frame, saber qual a duração da reprodução, dentre outras funcionalidades. A Tabela 3.2 ilustra detalha todos os métodos da classe *PlaybackControl*.

A sequência do texto demonstrará como é possível criar e utilizar um objeto do tipo *PlaybackControl* para reprodução das *streams* gravadas em um arquivo .oni.

Tabela 3.2. Métodos da Classe PlaybackControl

Método	Descrição
seek(VideoStream stream, int i)	Esse método retorna uma determinada imagem de índice ***i***. Ou seja, como o arquivo utilizado para a simulação do dispositivo físico é formado por um conjunto de imagens capturadas e indexadas de maneira sequencial. Pode-se, com esse método, posicionar a execução em um ponto especifico desse conjunto. Para isso, se faz necessário indicar onde estão as imagens e qual o seu índice através dos parâmetros ***stream*** e ***i***, respectivamente.
getNumberOfFrames(VideoStream stream)	Permite saber quantos *frames* tem a gravação ao total. Assim é possível estimar o tempo que vai levar a execução ou mesmo delimitar o numero máximo do índice que pode ser passado no parâmetro ***i***.
getSpeed()	Esse método retorna um *float* que diz qual a velocidade atual da execução.
setSpeed(float speed)	Permite configurar a velocidade de execução.
getRepeatEnable()	Esse método retorna um boolean que indica se a reprodução irá se repetir ou não.
setRepeatEnable(boolean bool)	Configura o modo de reprodução em *loop* conforme o valor passado como parâmetro.

O Método Device.get*PlaybackControl* () - construindo um *PlaybackControl* :

Não é possível criar um objeto *PlaybackControl* diretamente. Esse tipo de objeto só pode ser construído através da classe *Device*, pois ela é a responsável por fazer uma conexão com os dispositivos físicos ou virtuais. O trecho de código da Figura 3.8 ilustra como criar um objeto *PlaybackControl* de forma correta.

```
31          PlaybackControl pbControl;
32          //Verificando se o Device foi criado a partir de um arquivo
33          if(device.isFile()){
34              pbControl=device.getPlaybackControl();
35          }
36          else{
37              JOptionPane.showMessageDialog(null, "Device is not a file.",
38                                            "Error", JOptionPane.ERROR_MESSAGE);
39              return;
40          }
```

Figura 3.8. Criando um objeto *PlaybackControl* a partir da instância de uma classe *Device*

Na linha 33 do código da Figura 3.8 é feita a verificação se o *Device* está conectado a um dispositivo virtual, isto é, se os dados das *streams* são provenientes de um arquivo .oni. Se esta condição for satisfeita, na linha 34, o objeto *PlaybackControl* será obtido através do método get*PlaybackControl*() descrito na Tabela 3.1.

c) A classe VideoStream

A classe *VideoStream* é responsável por encapsular todos os fluxos de dados criados pela classe *Device*. Permite configurar o modo em que o vídeo será executado, além de diversas outras configurações relacionadas ao vídeo. Esta é uma das classes do *OpenNI* que é indispensável para construir qualquer aplicação, uma vez que, para exibir ou tratar um conjunto de imagens, se faz necessário que todo o fluxo das capturas de informações passe por esse objeto. É a única classe que pode se comunicar com a classe *Device* para obter os *frames* capturados pelo dispositivo. Por ser tão importante, ela é a classe mais extensa, em quantidade de métodos, da API.

O Método *create*() – construindo um *VideoStream*

A classe *VideoStream* é responsável por se conectar em um determinado sensor de um dispositivo. Para isso, se faz necessário passar como parâmetros um dispositivo e o tipo de sensor que será utilizado para fornecer os dados da *stream*.

Para inicializar um objeto do tipo *VideoStream* se faz necessário chamar o método abstrato desta classe, intitulado de *create*(), passando como parâmetros um objeto do tipo *Device* e um *SensorType*.

O trecho de código da Figura 3.9 (linha 34) demonstra como instanciar um objeto do tipo *VideoStream* que está associado a um sensor RGB. Pode-se observar que, somente será criado um *VideoStream*, após verificado se o dispositivo suporta sensores do tipo RGB (linhas 26, 27 e 31).

```
25          //Obtendo e referenciando um SensorType do tipo COLOR
26          SensorType type=SensorType.COLOR;
27          boolean isSupported=device.hasSensor(type);
28          //Criando uma variável VideoStream
29          VideoStream videoStream;
30         //Verificando se o dispositivo suporta sensor do tipo COLOR (RGB)
31          if(isSupported){
32          //Criando um objeto VideoStream, responsável pela devolução
33          //dos dados capturados, e referenciando-o com a variável videoStream.
34          videoStream=VideoStream.create(device, type);
35          }else{
36            JOptionPane.showMessageDialog(null, "Sensor not supported.",
37                                          "Error", JOptionPane.ERROR_MESSAGE);
38              return;
39          }
```

Figura 3.9. Criando um *VideoStream*

A Tabela 3.3 faz um resumo de todos os outros métodos da classe *VideoStream*, com o objetivo de fornecer os conceitos de utilização de cada um deles.

Tabela 3.3.Métodos da Classe VideoStream.

Método	Descrição
create(Device device, SensorType type)	É um método abstrato da classe VideoStream que é responsável por criar um objeto deste tipo, já que essa classe não possua construtor.
addNewFrameListener(VideoStream.NewFrameListener nl)	Quando um frame é capturado o dispositivo gera um evento para avisar ao VideoStream que tem um novo frame a ser buscado. Este método é utilizado para adicionar um listener, chamado de NewframeListener, ao VideoStream.
removeNewFrameListener(VideoStream.NewFrameListener nl)	Remove um NewFrameListener antes adicionado. Desta forma, o VideoStream ficará sem receber eventos desse listener.
destroy()	Serve para limpar toda a memória utilizada pelo VideoStream.
getCameraSettings()	Retorna um objeto do tipo CameraSettings.
isCroppingSupported()	Indica se o VideoStream suporta recortes.
getCropping()	Retorna um objeto do tipo CropArea.
setCropping(CropArea ca)	Passa um objeto do tipo CropArea para o VideoStream.
getHandle()	Retorna o numero do endereço de memória onde está o objeto VideoStream que foi criado pelo código nativo em C/C++.
getHorizontalFieldOfView()	Obtém o campo de visão horizontal dos frames recebidos a partir desse stream.
getVerticalFieldOfView()	Obtém o campo de visão vertical dos frames recebidos a partir desse stream.
getMaxPixelValue()	Fornece o valor máximo para pixels obtidos pelo VideoStream.
getMinPixelValue()	Fornece o valor mínimo para pixels obtidos pelo VideoStream.
getMirroringEnabled()	O VideoStream tem a capacidade de fazer refletir os pixels da imagem através de um eixo vertical, fazendo com que a imagem fique como se estivesse sendo refletida por um espelho. Ou seja, esse método é o responsável por indicar se a reflexão esta ativada ou não.
setMirroringEnabled(boolean bln)	É responsável por ativar ou desativar a reflexão dos pixels de um VideoStream.
getSensorInfo()	Como o VideoStream foi criado a partir do objeto Device, é possível saber as informações de seus sensores através desse método.
getSensorType()	Fornece qual o tipo de sensor utilizado para adquirir as imagens, retornado assim um objeto do tipo SensorType.
getVideoMode()	Retorna um objeto do tipo VideoMode que configura os parâmetros do modo de vídeo utilizados pelo VideoStream.
setVideoMode(VideoMode mode)	Permite a configuração do objeto VideoStream através das informações encapsuladas em um VideoMode.
readFrame()	É utilizado para ter acesso ao novo frame capturado através do objeto de retorno do tipo VideoFremeRef.
start()	Inicia a busca pelas imagens capturadas disponíveis no objeto Device.
stop()	Esse método interrompe a busca de imagens no Device. Ele pode ser chamado quando se deseja pausar a exibição do um vídeo, por exemplo.

d) A classe VideoStream.CameraSettings

CameraSettings é uma classe interna da classe *VideoStream* que fornece possibilidades de configurações de uma câmara RGB. Ela permite ativar ou desativar o balanço automático do branco e também a exposição automática.

A Tabela 3.4 ilustra todos os métodos da classe *CameraSettings*, bem como as explicações dos conceitos associados a cada um deles. Na sequência do texto será explicado o funcionamento do método utilizado para ter acesso ao objeto *CameraSettings* (Figura 3.10).

Tabela 3.4.Métodos da Classe VideoStream.CameraSettings

Método	Descrição
getAutoExposureEnabled()	Verifica se a opção de exposição automática está habilitada.
getAutoWhiteBalanceEnabled()	Verifica se o balanço automático de branco está habilitado.
setAutoExposureEnabled (boolean enabled)	Habilita ou desabilita a exposição automática da câmara.
setAutoWhiteBalanceEnabled (boolean enabled)	Habilita ou desabilita o balanceamento automático de branco da câmara.

Acessando o objeto *CameraSettings*:

Não existe construtor para esta classe. Um objeto desse tipo só poderá ser obtido através da chamada do método *getCameraSettings*() do *VideoStream*.

```
31        SensorType type=SensorType.COLOR;
32        //Criando um Device
33        Device device=Device.open();
34        //Criando um objeto VideoStream, responsável pela devolução
35        //dos dados capturados, e referenciando-o com a variável stream.
36        VideoStream stream=VideoStream.create(device, type);
37        //Criando um CameraSettings
38        VideoStream.CameraSettings cs =stream.getCameraSettings();
39
```

Figura 3.10. Criando um objeto do tipo *CameraSettings*

e) A classe VideoMode

Um objeto da classe *VideoMode* é utilizado para configurar o comportamento da captura dos frames do sensor de vídeo. Os atributos disponíveis para essas configurações incluem: (i) o formato de pixel das capturas; (ii) a resolução da imagem e (iii) a taxa de frames por segundo (fps). Assim, é possível determinar, por exemplo, que cada imagem tenha resolução de 640x480 pixels ou que a taxa de captura de imagens seja 30 fps.

A Tabela 3.5 traz todos os principais métodos da classe *VideoMode* e fornece os conceitos básicos para utilização de cada um deles. Em seguida, um trecho de código ilustrando a criação de um objeto do tipo *VideoMode* é apresentado.

Tabela 3.5. Métodos da Classe VideoMode

Método	Descrição
setResolution(int resX, int resY)	Permite passar a resolução em que as imagens devem ser capturadas.
setPixelFormat(PixelFormat pf)	Configura o formato de *pixel* desejado para as imagens.
setFps(int fps)	Possibilita determinar a quantidade de *frames* que devem ser capturados a cada segundo.

Construindo um *VideoMode*:

Para se criar um objeto do tipo *VideoMode*, se faz necessário configurar pelo menos os atributos relacionados à taxa fps, ao formato do pixel e às resoluções no eixo X e no eixo Y. O trecho de código da Figura 3.11 ilustra como instanciar o objeto da classe *VideoMode* com o construtor padrão (linha 45), como configurar esses parâmetros utilizando os métodos *setters* (linhas 47, 49 e 51) e como criar um outro objeto da mesma classe com a passagem de todos os parâmetros no construtor (linha 54).

```
44          //Criando um modo de video (VideoMOde)  vazio
45          VideoMode mode=new VideoMode();
46          //Passando a resolução dascapturas
47          mode.setResolution(640,480);
48          //Passando a quantidade de FLOPS
49          mode.setFps(30);
50          //Passando o tipo de Pixel
51          mode.setPixelFormat(PixelFormat.RGB888);
52
53          //Criando um VideoMode já com todas as características
54          VideoMode mode1=new VideoMode(640,480,30,PixelFormat.RGB888.toNative());
55
```

Figura 3.11. Exemplos de como criar um *VideoMode*

f) A classe OpenNI

A classe *OpenNI* é responsável pela inicialização da aplicação, permitindo o acesso às bibliotecas ou SDK. É por meio desta classe que é estabelecida a conexão com o dispositivo físico, através da verificação de seus respectivos *drivers* instalados.

Tabela 3.6. Métodos da Classe OpenNI

Método	Descrição
initialize()	Responsável por toda a configuração do ambiente, desde a checagem dos drives até a carga das bibliotecas. Importante ressaltar que deve ser executado antes do método de criação de um *Device*. Do contrário, o dispositivo físico não vai poder ser localizado pelo *Device*.
enumerateDevices()	Retorna uma lista de objetos do tipo *DeviceInfo*, que nada mais é que uma lista com informações de todos os dispositivos físicos conectados ao sistema.
getExtendedError()	Retorna a descrição de um erro. Esse método é comumente utilizado quando se deseja saber qual foi o erro que gerou uma determinada exceção.
waitForAnyStream(List<VideoStream> list, int i)	Aguarda até que um novo *frame*, de qualquer um dos *VideoStreams* pertencentes à lista, seja identificado ou que o seu tempo de espera tenha sido ultrapassado.
shutdown()	Cancela a comunicação da aplicação com o dispositivo. Deve ser chamado no final de cada aplicação para que o dispositivo físico seja liberado.
addDeviceConnectedListener(OpenNI.DeviceConnectedListener dl)	Adiciona um novo *listener* de conexão de dispositivo à aplicação.
addDeviceDisConnectedListener(OpenNI.DeviceDisConnectedListener dl)	Adiciona um novo *listener* de desconexão de dispositivo à aplicação.
addDeviceStateChangedListener(OpenNI.addDeviceStateChangedListener dl)	Adiciona um novo *listener* de verificação de estado de um dispositivo à aplicação.
removeDeviceConnectedListener(OpenNI.DeviceConnectedListener dl)	Remove um determinado *listener* de conexão de dispositivo.
removeDeviceDisConnectedListener(OpenNI.DeviceDisConnectedListener dl)	Remove um determinado *listener* de desconexão de dispositivo.
removeDeviceStateChangedListener(OpenNI.addDeviceStateChangedListener dl)	Remove um *listener* de verificação de estado de um dispositivo.

A classe *OpenNI* possui três *listeners* (*DeviceConnectedListener*, *DeviceDisconectedListener* e *DeviceStateChangedListener*) que escutam o estado do dispositivo físico e disparam eventos sempre que um dispositivo físico seja conectado, desconectado ou seu estado tenha sido alterado sistema.

Essa classe não tem construtor, pois todos os seus métodos são abstratos. Contudo, seu método *initialize*() deverá ser o primeiro a ser chamado em uma aplicação, pois, sem essa chamada, não será possível acessar os dispositivos e seus sensores disponíveis. De toda forma, é importante conhecer todos os métodos disponíveis nessa classe, bem como seus métodos. A Tabela 3.6 ilustra todo o conjunto de métodos disponíveis na classe e uma breve explicação de cada um destes métodos.

g) A classe SensorType

A classe *SensorType* utilizada apenas para armazenar definições de tipos de sensores suportados[1]. Basicamente, ela não possui métodos disponíveis e dentre as definições armazenadas, estão:

- **IR**: para imagens geradas a partir do sensor de infravermelho;
- **COLOR**: para imagens geradas a partir do sensor RGB e;
- **DEPTH**: para imagens geradas a partir do sensor de profundidade.

h) A classe SensorInfo

Um objeto desse tipo armazena informações do sensor que são utilizados pela classe *Device* e *VideoStream*. Dentre as informações presentes nesse objeto, podemos encontrar o tipo de sensor (*SensorType*), além de uma lista de modos de vídeo (*VideoMode*) suportados pelo sensor.

[1] Na versão atual do *OpenNI*, não existe suporte aos sensores de áudio e motor.

Conforme mostrado na Tabela 3.7, essa classe possui apenas dois métodos. Um exemplo de como construir um *SensorInfo*, bem como a utilização de um de seus métodos comumente utilizado, são descritos a seguir.

Tabela 3.7. Métodos da Classe SensorInfo

Métodos	Descrição
getSensorType()	Retorna o tipo de sensor *(IR, RGB ou DEPTH)*.
getSupportedVideoModes()	Retorna uma lista de modos de vídeo suportados pelo sensor.

Construindo um objeto do tipo *SensorInfo*:

Esse objeto não possui construtor. Sua instância só poderá ser obtida através de um objeto do tipo *Device* ou *VideoStream*. Os trechos de código abaixo ilustram capturas das instâncias de *SensorInfo* obtidas através do *Device* (a) e do objeto *SensorInfo* (b).

```
SensorType type = SensorType.DEPTH;
Device device= Device.open();
VideoStream stream=VideoStream.create(device, type);

//Obtendo um SensorInfo
SensorInfo info;
//Através de um Device
info =device.getSensorInfo(type);
```

```
SensorType type = SensorType.DEPTH;
Device device= Device.open();
VideoStream stream=VideoStream.create(device, type);

//Obtendo um SensorInfo
SensorInfo info;
// ou Através de um VideoStream
info=stream.getSensorInfo();
```

Figura 3.12. Obtendo SensorInfo através do *Device* (a) e (b) do *VideoStream*

O Método *getSupportedVideoModes*()

O trecho de código ilustrado a seguir na Figura 3.13 ilustra como utilizar o objeto *SensorInfo* para verificar os modos de operação suportados pelo sensor. O exemplo mostra como a lista de configurações suportadas por um sensor RGB pode ser capturada. Esse tipo de abordagem é a mais recomendada, uma vez que evita que os desenvolvedores emitam parâmetros de configurações inválidos ou não suportados pelo sensor em que está tendo acesso no momento.

```
30          SensorType type=SensorType.COLOR;
31          boolean isSupported=device.hasSensor(type);
32          //Criando uma variável VideoStream
33          VideoStream videoStream;
34         //Verificando se o dispositivo suporta sensor do tipo COLOR (RGB)
35          if(isSupported){
36          //Criando um objeto VideoStream, responsável pela devolução
37          //dos dados capturados, e referenciando-o com a variável stream.
38          videoStream=VideoStream.create(device, type);
39          }else{
40            JOptionPane.showMessageDialog(null, "Sensor not supported.",
41                                          "Error", JOptionPane.ERROR_MESSAGE);
42              return;
43          }
44          //Obtendo uma lista com os modos de vídeo suportados pelo sensor COLOR
45          List<VideoMode> supportedVideoModes=videoStream.getSensorInfo().
46                                              getSupportedVideoModes();
47
```

Figura 3.13. Exemplo de uso do método *getSupportedVideoModes*()

i) A classe DeviceInfo

A classe *DeviceInfo* encapsula todas as informações referentes a um dispositivo. Algumas dessas informações comumente utilizadas pelos desenvolvedores são o nome do dispositivo, nome do seu fabricante e a URI.

A Tabela 3.8 lista todos os métodos disponíveis nessa classe e faz uma breve explicação dos conceitos de cada um desses métodos. Note que os objetos do tipo *DeviceInfo* devem ser obtidos através do método *getDeviceInfo*() do classe *Device* (seção a)).

Tabela 3.8. Métodos da Classe DeviceInfo

Métodos	Descrição
getName()	Retorna uma *String* com o nome do dispositivo.
getUri()	Retorna a rota de conexão do dispositivo no sistema: *URI*.
getUsbProductId()	Retorna o identificador do dispositivo USB (*PID*).
getVendor()	Retorna o nome do fabricante do dispositivo.
getUsbVendorId()	Retorna o identificador do fornecedor/fabricante do dispositivo (*VID*).

j) A classe VideoFrameRef

A classe *VideoFrameRef* encapsula todos os dados relacionados a um determinado *frame* obtido por um *VideoStream*. É a principal classe utilizada pelo *VideoStream* para retornar cada novo *frame* capturado. Um objeto *VideoFrameRef* permite acessar um *buffer* que contém os dados do *frame*, bem como quaisquer outros metadados que se façam necessários para trabalhar com o *frame* atual.

Construindo um *VideoFrameRef*

Dado que a classe *VideoFrameRef* não possui construtor, um objeto desta classe exige um objeto *VideoStream* e uma chamada ao método *readFrame* para sua instanciação. O exemplo da Figura 3.14 ilustra uma criação de um *VideoFrameRef*. Porém, o mais natural seria que o *VideoFrameRef* fosse obtido dentro de um método que captura o evento produzido pelo *NewFrameListener*. Isto porque, o *VideoFrameRef* guarda informações de apenas um *frame* e assim, se for necessário capturar todos os *frames* de uma sequência, deve ser criado um *VideoFrameRef* para cada nova captura.

```
30  SensorType type=SensorType.COLOR;
31  Device device=Device.open();
32  VideoStream stream= VideoStream.create(device, type);
33  //Criando um FrameRef
34  VideoFrameRef ref =stream.readFrame();
35
```

Figura 3.14. Criando um VideoFrameRef

A Tabela 3.9 lista todos os métodos disponíveis na classe *VideoFrameRef*, bem como uma breve descrição dos conceitos de cada um desses métodos.

Tabela 3.9. Métodos da Classe VideoFrameRef

Método	Descrição
getCropOriginX()	Indica a coordenada ***X*** do canto superior esquerdo da janela onde será obtida a informação. Retorna a distância de origem do recorte a partir do lado esquerdo da imagem, em *pixels*.
getCropOriginY()	Indica a coordenada ***Y*** do canto superior esquerdo da janela onde será obtida a informação. Retorna a distância de origem do recorte da parte superior da imagem, em *pixels*.
getCroppingEnabled()	Indica se a propriedade de recorte está ativa no momento em que o frame tenha sido capturado.
getData()	Retorna um buffer com todos os bytes/pixels do frame capturado.
getFrameIndex()	Este método retorna o número de identificação do frame atual, que é atribuído ao frame no momento em que é capturado.
getHeight()	Fornece a altura do *frame* atual, medida em *pixels*. Este método depende da ativação ou não da opção de recorte. Caso a opção de recorte estiver ativada, o método retornará a altura da janela de recorte. Do contrário, irá retornar a resolução ***Y*** do *VideoMode*, utilizado para reproduzir o *frame*.
getSensorType()	Retorna o tipo de sensor utilizado para a captura do *frame*.
getStrideInBytes()	Fornece o comprimento de uma linha de *pixels*, medido em *bytes*.
getTimestamp()	Retorna o tempo da captura do *frame* atual, a partir de um tempo zero arbitrado pelo dispositivo.
getWidth()	Fornece a largura do frame atual, medida em pixels. Este método depende da ativação ou não da opção de recorte. Caso a opção de recorte estiver ativada, o método retornará a largura da janela de recorte. Do contrário, irá retornar a resolução **X** do *VideoMode*, utilizado para reproduzir o *frame*.
getVideoMode()	Retorna uma referência ao objeto *VideoMode* que associado ao *frame* atual.
release()	Libera a referência ao *frame*.

k) A classe Recorder

A classe *Recoder* é a responsável por criar e gravar em disco um arquivo ".oni", que contenha os dados capturados através de um sensor vinculado ao *VideoStream*. Isso permite que, posteriormente, esse arquivo seja utilizado como provedor de informações de um dispositivo virtual. Assim, a reprodução dos dados nele armazenados, simula um dispositivo físico em funcionamento (seção b)).

Todos os métodos desta classe estão na Tabela 3.10. Na sequência, um exemplo que ilustra a criação e inicialização do objeto *Recorder* para executar uma gravação dos *frames* em um arquivo em disco na Figura 3.15.

Tabela 3.10. Métodos da Classe Recorder

Método	Descrição
create (String fileName)	Método estático que cria um arquivo com o nome especificado em *fileName*. É indispensável que no nome do arquivo contenha também uma extensão, que aqui deve ser ***.ONI***, pois caso contrário o arquivo não poderá ser reproduzido.
addStream(videoStream stream, boolean bln)	Adiciona uma ou mais *stream* para que seus *frames* seja guardados no arquivo ***.ONI*** . O parâmetro ***boolean bln*** diz respeito à compressão do arquivo. Aconselha-se que seja falso. Pois caso seja verdadeiro, o arquivo será salvo de maneira comprimida com perda de qualidade de informações. O que as vezes não é recomendável, pois a qualidade da reprodução poderá ser prejudicada.
start()	Esse método inicia a gravação dos dados do *stream* no arquivo.
stop()	Esse método da uma pausa na gravação dos dados.
destroy()	Finaliza a gravação dos dados junto com os objetos criados pela gravadora.

No trecho de código ilustrado na Figura 3.15, para que a classe *Recorder* funcione a contento, é necessário que tenha sido instanciada e inicializada passando como parâmetro o caminho onde será armazenado o arquivo ".oni" (linha 55). Em seguida (linha 59), o *Recorder* é vinculado a um ou mais *streams* e após isso, deve-se solicitar o início da gravação para que sejam armazenados cada um dos *frames* de cada *stream* no arquivo especificado como parâmetro.

```
53          //Criando uma gravadora e especificando o nome e extensão
54          //do arquivo onde serão gravado os dados
55          Recorder recorder= Recorder.create("Exemplo.ONI");
56          //Adicinando uma stream a qual fornecerá os dados a serem gravados
57          //e fornecendo a informação que diz se o arquivo deverá ser gravado de
58          //maneira compactada.
59          recorder.addStream(stream, false);
60          //Iniciando o processo de gravação dos dados
61          recorder.start();
62
```

Figura 3.15. Exemplo de como instanciar uma classe do tipo *Recorder*

l) A classe CordinateConverter

A classe *CordinateConverter* permite converter pontos entre os diferentes sistemas de coordenadas. Neste ponto, cabe uma breve explicação para facilitar o entendimento do leitor no que se refere aos conceitos e à importância desta classe. As aplicações *OpenNI* geralmente usam dois sistemas de coordenadas distintos para representar a profundidade. Estes dois sistemas de coordenadas são comumente relatados como sendo coordenadas de profundidade e coordenadas de representação no mundo real.

O sistema de coordenadas de profundidade trabalha sobre as representações de dados nativos extraídos diretamente dos sensores. Neste sistema, a estrutura é um mapa (matriz bidimensional) da imagem e um valor de profundidade é atribuído a cada pixel da imagem. Este valor de profundidade representa a distância entre o plano da câmara e qualquer objeto. As coordenadas X e Y são simplesmente o local no mapa, onde a origem é o canto superior esquerdo do campo de visão.

Já no sistema de coordenada do mundo real, cada coordenada é especificada pelos eixos X, Y e Z. Como exemplo, do ponto de vista da câmera, um objeto que se move da esquerda para a direita está se movendo ao longo do eixo X. Um objeto que se move para cima e para baixo está se movendo ao longo do eixo Y, e um objeto se afastando da câmera está se movendo ao longo do eixo Z. A Figura 3.16 ilustra a disposição dos eixos desse sistema de coordenadas.

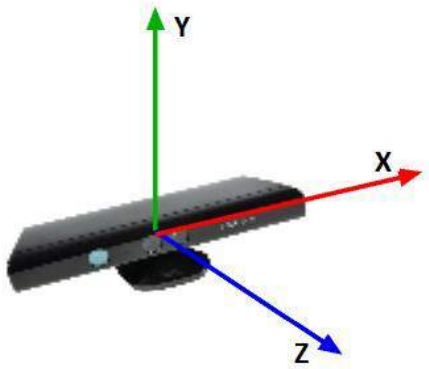

Figura 3.16. Sistema de Coordenadas do Mundo Real

Importante ressaltar que não se deve fazer conversão dos sistemas de coordenadas em tempo real devido ao alto custo de processamento. É aconselhável que o desenvolvedor trabalhe todo o software em cima do sistema de coordenadas nativo (sistema de coordenadas de profundidade) e faça a conversão somente no momento anterior que necessitar produzir a saída dos dados.

Todos os métodos disponíveis para a classe *CordinateConverter*, assim como uma breve descrição de cada um deles, estão descritos na Tabela 3.11.

Tabela 3.11. Métodos da Classe CordinateConverter

Métodos	Descrição
convertDepthToColor(VideoStream depthStream, VideoStream colorStream, int depthX, int depthY, DepthPixel depthZ, int pColorX, int pColorY)	Para um determinado ponto de profundidade, fornece as coordenadas do valor da cor correspondente.
convertDepthToWorld(VideoStream depthStream, int depthX, int depthY, DepthPixel depthZ, float pWorldX, float pWorldY, float pWorldZ)	Converte um único ponto do sistema de coordenadas de profundidade para o sistema de coordenadas de mundo real.
convertDepthToWorld (VideoStream depthStream, float depthX, float depthY, float depthZ, float pWorldX, float pWorldY, float pWorldZ)	Converte um único ponto, a partir de uma representação de ponto flutuante, de um sistema de coordenadas de profundidade em um sistema de coordenadas do mundo real.
onvertWorldToDepth (VideoStream depthStream, float worldX, float worldY, float worldZ, int pDepthX, int pDepthY, DepthPixel pDepthZ)	Converte um único ponto do sistema de coordenadas mundo real para o sistema de coordenadas de profundidade.
convertWorldToDepth (VideoStream depthStream, float worldX, float worldY, float worldZ, float pDepthX, float pDepthY, float pDepthZ)	Converte um único ponto, a partir de uma representação de ponto flutuante, de um sistema de coordenadas de mundo real, em um sistema de coordenadas de profundidade.

m) As demais classes

As demais classes serão apenas listadas e descritas sem maiores detalhes por se tratarem apenas de classes de apoio e por, em sua maioria, possuem apenas métodos *Getters* e *Setters* de fácil entendimento.

- **Classe *NativeMethods***: Responsável por todo o mapeamento da API *OpenNI* 2.x em C/C++ utilizado pelo *wrapper Java* da API 2.x para a comunicação com o código nativo através do JNI.

- **Classe *OutArg***: Serve de apoio à classe *NativeMethods* para definir tipos de variáveis utilizadas pelo código nativo.
- **Classe *CropArea***: Essa classe encapsula informações de recortes de dados.
- **Classe *PixelFormat***: Define os diferentes tipos de pixels utilizados pela API.
- **Classe *Version***: Identifica qual a versão atual do *OpenNI* que a aplicação está utilizando.
- **Classe *ImageRegistrationMode***: Dois ou mais dispositivos podem trabalhar em conjunto para a formação de uma só imagem com o *framework OpenNI*. Os dados dos dispositivos são sobrepostos fazendo com que, por exemplo, uma imagem RGB possa se sobrepor a uma imagem de profundidade, produzindo como resultado final uma única imagem. Alguns dispositivos possuem a capacidade de fazer cálculos matemáticos necessários para essas sobreposições em seu hardware. Esta classe permite saber se o dispositivo utilizado tem ou não essa capacidade para que possa gerenciá-la.
- **Classes *Point3D* e *Point2D***: Encapsulam, respectivamente, pontos de 3 dimensões e pontos de 2 dimensões.

3.5. Programando com o *OpenNI*

As seções seguintes deste capítulo terão foco nos aspectos práticos da programação utilizando o *framework OpenNI*, fornecendo o conhecimento básico e essencial para qualquer desenvolvedor iniciar suas aplicações relacionadas à interação natural dentro desta plataforma. O texto está estruturado de forma a apresentar: (i) a montagem e a configuração de um ambiente de desenvolvimento; (ii) a integração entre o *wrapper Java* do *OpenNI* [OpenNI 2011] e o *NetBeansIDE* [Netbeans 2012]; (iii) a criação de uma aplicação básica com Java e *OpenNI*; (iv) a criação de uma aplicação simulando um dispositivo físico e (iv) de uma aplicação para seguir os movimentos das mãos (*Hand Tracking*).

3.5.1 Montagem e configuração de um ambiente para o desenvolvimento

O ambiente de exemplo para criar aplicações integrando o *OpenNI* e a linguagem Java utiliza um *Kinect* como hardware de obtenção de dados, o *framework OpenNI*2.2 e o *middleware* NITE. O texto explica como obter todos os arquivos utilizados, como instalá-los e configurá-los e como testar todo o ambiente antes que o desenvolvedor inicie a programação. Em seguida, será mostrado exemplo de uma construção de aplicação própria no ambiente montado.

Este trabalho não abordará as instalações do sistema operacional Windows 7 Home basic (64 Bits), jdk 1.7.0_21 (64 Bits), jdk1.7.0_21 (32 Bits) e a IDE NetBeans IDE 7.3, utilizados como base para o ambiente demonstrado.

a) Instalação do Kinect SDK

Kinect SDK é o ambiente de programação criado pela Microsoft [Microsoft 2010], contendo os drivers e bibliotecas necessários para programação com *Kinect*. Para sua instalação e configuração, se faz necessário a execução dos passos que se seguem:

1. *Download* e instalação do KinectSDK através do link:

http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx;

2. Após instalação concluída, conectar um *Kinect* na porta USB 2.0[2] e verificar se os *drivers* do sensor foram instalados com sucesso, acessando "Panel de controle» Sistema e segurança»Sistema» Gerenciador de dispositivo", conforme ilustra a Figura 3.17.

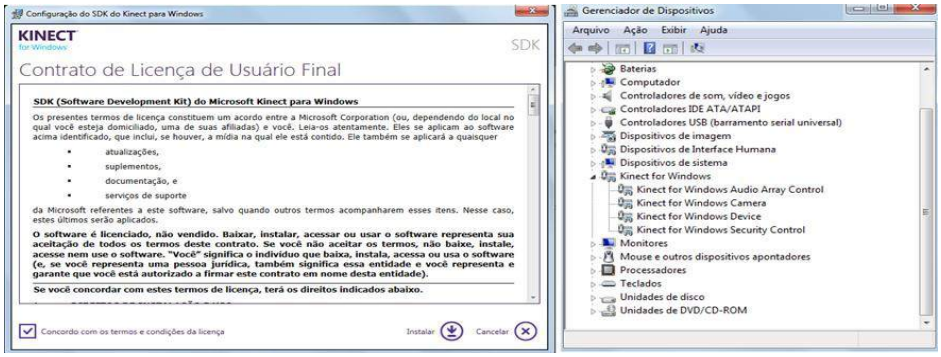

Figura 3.17. a) Instalando o Kinect SDK e b) Verificando se foi instalado corretamente no gerenciador de dispositivos

b) Instalação do OpenNI

O *OpenNI* é um SDK aberto, que é utilizado para desenvolvimento de bibliotecas de *middleware* e aplicações que utilizam sensores 3D. Esta plataforma fornece suporte a diversos sistemas operacionais e a sensores 3D distintos como o *Kinect* [Microsoft Kinect 2010], Carmine ou Xition [PrimeSense 2011]. A instalação e configuração do *OpenNI* será descrita e ilustrada a seguir:

1. É necessário obter o *OpenNI* através do *download* do SDK acessando a URL http://www.OpenNI.org/OpenNI-sdk/
2. Após a conclusão do *download*, executar o arquivo para instalá-lo mantendo suas configurações padrões, conforme ilustra (Figura 3.18).

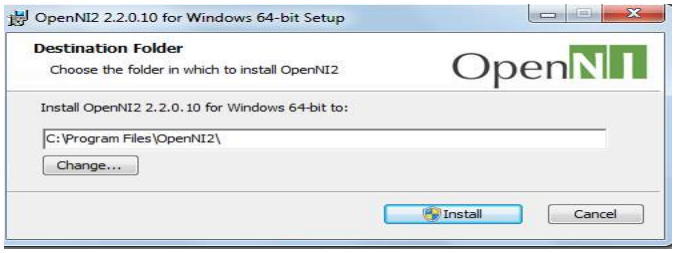

Figura 3.18. Instalação do *OpenNI*

c) Instalação do Middleware NITE

O *NITE* é um *middleware* de Visão Computacional 3D que possibilita criar aplicações que controlem movimentos da mão ou de todo o corpo. O presente trabalho não abordará a construção de aplicações com este *middleware*. Porém, no ambiente de desenvolvimento serão previstas a instalação e configuração do *NITE* para uso futuro. Os passos para instalação deste *middleware* se resumem a: (i) obter o *NITE* mais atual

[2] É recomendável o uso da porta USB 2.0, pois a conexão com uma porta 3.0 pode causar travamentos indesejáveis na obtenção de dados do Sensor.

(*NITE* 2.x) através da URL: http://www.OpenNI.org/files/nite/ e (ii) executar o arquivo para instalá-lo.

d) Testando a Instalação

Para vericar se a instalação está funcionando corretamente, basta acessar a pasta onde estão os exemplos do *OpenNI* ("C:\Program Files\OpenNI2\Samples\Bin", em geral) e executar um dos arquivos disponíveis. Se a execução mostrar uma imagem de algum sensor, significa que tudo foi instalado com sucesso. Na Figura 3.19 é ilustrado a execução, com sucesso, do exemplo *SimpleViewer*.

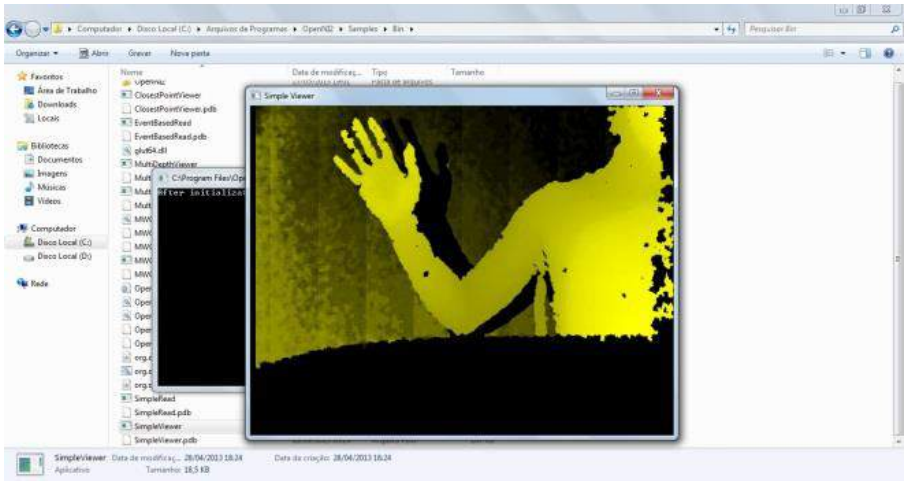

Figura 3.19. Executando o exemplo *SimpleViewer* para testar a instalação

e) Outras Considerações

O diagrama da Figura 3.20 ilustra como deveria ficar o ambiente de desenvolvimento após todas as instalações sugeridas nos tópicos anteriores. Estaria disponível para o desenvolvedor a IDE *NetBeans* 7.3, que poderá fazer compilações para computadores 64/32 bits, através dos dois tipos de JDK disponíveis na IDE. Ainda na IDE, será possível desenvolver códigos utilizando o *NITE*, o *OpenNI* ou ambos.

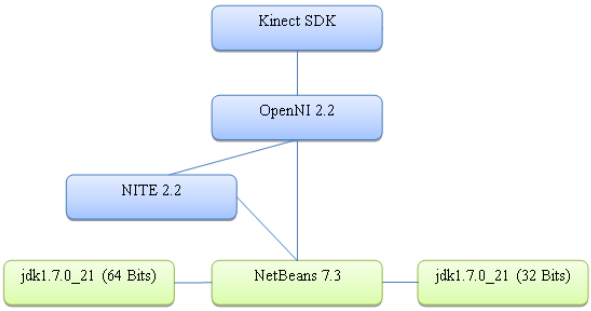

Figura 3.20. Diagrama de Componentes do Ambiente de Desenvolvimento

Importante notar que o *OpenNI*, no Sistema Operacional *Windows*, está diretamente ligado ao SDK da *Microsoft*. Pois, é com este SDK que a *Microsoft* disponibiliza os *drivers* necessários para acessar os sensores do dispositivo físico (Kinect). Logo, é por meio desses *drivers* que o *OpenNI* consegue capturar os dados de todos os sensores.

3.5.2 Integrando o *wrapper* Java do *OpenNI* com o *NetBeansIDE*

Nesta seção será mostrado como localizar o *wrapper* Java após a instalação dos pacotes necessários. Além disso, será mostrada toda a configuração necessária para que o usuário deixe a sua IDE *NetBeans* funcional e isenta de erros.

a) Wrappers Java

A localização dos *wrapper*s do *OpenNI* e do *NITE* ficam na pasta "Redist" de cada instalação. E, por padrão, se encontram nos respectivos endereços:

"C:\Program Files\OpenNI2\Redist\org.OpenNI.jar"

"C:\Program Files\PrimeSense\NiTE2\Redist\com.primesense.nite.jar"

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no OpenNI2.jni in java.library.path
  at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1860)
  at java.lang.Runtime.loadLibrary0(Runtime.java:845)
  at java.lang.System.loadLibrary(System.java:1084)
  at org.openni.NativeMethods.<clinit>(NativeMethods.java:44)
  at org.openni.OpenNI.initialize(OpenNI.java:113)
  at org.openni.Samples.SimpleViewer.SimpleViewerApplication.main(SimpleViewerApplication.java:193)

Exception in thread "main" java.lang.UnsatisfiedLinkError: no NiTE2.jni in java.library.path
  at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1860)
  at java.lang.Runtime.loadLibrary0(Runtime.java:845)
  at java.lang.System.loadLibrary(System.java:1084)
  at com.primesense.nite.NativeMethods.<clinit>(NativeMethods.java:31)
  at com.primesense.nite.NiTE.initialize(NiTE.java:15)
  at com.primesense.nite.Samples.UserViewer.UserViewerApplication.main(UserViewerApplication.java:71)
```

Figura 3.21. Exceção lançada quando não é encontrado o pacote java que possue a implementação JNI (o *wrapper* Java)

Para utilizar o *NetBeansIDE*, é importante colocar o endereço de ambos *wrapper*s dentro do path do *Windows*. Desta maneira, os programas em desenvolvimento escritos com o auxílio desta IDE serão executados com êxito. Caso contrário erros, conforme ilustra a Figura 3.21 , serão bastante comuns no momento da execução da aplicação.

b) Testando o NetBeans - Executando uma aplicação exemplo em Java

Para verificar se o ambiente foi montado e configurado corretamente, basta executar os exemplos disponibilizados pelo *framework*. Para isso, o texto discute os exemplos "SimpleViewer.java" e "UserView.java", que por padrão encontram-se na pasta Redist[3] da intalação *OpenNI* e do NITE , respectivamente.

No caso do exemplo "UserView.java", ou qualquer outro exemplo relacionado ao *middleware* NITE, se faz necessário copiar toda a pasta NITE2, que se encontra na pasta Redist, para o diretório raiz do projeto do *NetBeans*. Do contrário, ocorrerão "*exceptions*" pela falta do arquivo "s.dat", conforme ilustrado na Figura 3.22.

```
Could not find data file .\NiTE2/s.dat
current working directory = C:\Users\Lucas\Documents\NetBeansProjects\UserViewer
Exception in thread "main" java.lang.RuntimeException
  at com.primesense.nite.NativeMethods.checkReturnStatus(NativeMethods.java:40)
  at com.primesense.nite.UserTracker.create(UserTracker.java:90)
  at com.primesense.nite.Samples.UserViewer.UserViewerApplication.main(UserViewerApplication.java:80)
```

Figura 3.22. Exceção lançada ao não encontrar o arquivo de banco de dados utilizado pelo *NITE*

Para se executar os exemplos no *NetBeans*, deve-se criar um projeto, copiar os fontes disponíveis e fazer a importação das bibliotecas (org.OpenNI.jar), do *wrapper*

[3] Diretório padrão onde se encontram os exemplo disponibilizados do OpenNI e do NiTE são, respectivamente: C:\Program Files\OpenNI2\Redist. e C:\Program Files\Primesense\Nite2\Redist

Java do *OpenNI* e, (com.primesense.nite.jar) do *wrapper java* do NITE . Em seguida, deve-se compilar o projeto criado e executar separadamente cada um dos exemplos. O resultado final das execuções das aplicações exemplo, deve ser similar à Figura 3.23.

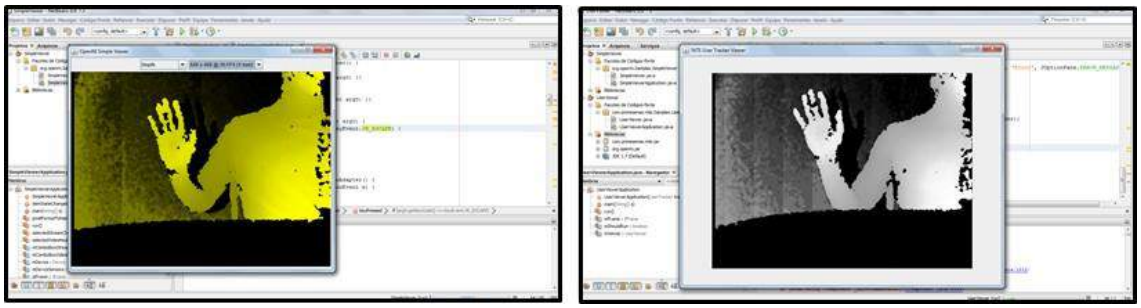

Figura 3.23. a) Resultado da execução do *SampleViewer.java* e b) Resultado da execução do *UserViewer.java*

3.5.3 Criando uma aplicação básica em Java com o *OpenNI*

Concluída a configuração do ambiente conforme os tópicos anteriores, esta parte do trabalho traz exemplos de como se construir uma aplicação básica para obter os dados do sensor RGB do *Kinect*. O exemplo visa colocar em prática os conhecimentos do *framework* e conceitos vistos até aqui.

3.5.4 Preparando o projeto Java, na IDE NetBeans, para fazer o primeiro exemplo com o *OpenNI*

Para que seja iniciado o desenvolvimento de uma aplicação Java com o *OpenNI*, é necessário que o usuário crie um novo projeto Java no *NetBeans*. E, após um novo projeto Java ter sido criado, é de fundamental importância importar as bibliotecas necessárias para programação com o *OpenNI* e *NITE*, localizadas em suas respectivas pastas *Redist* e cujos os nomes são *org.OpenNI.jar* e *com.primesense.nite.jar*[4].

a) Criando a primeira aplicação OpenNI: Iniciando e obtendo conexão com o sensor RGB

Basicamente, para se construir uma aplicação com o *OpenNI*, é necessário conhecer pelo menos quatro classes principais, que estão presentes na maioria das aplicações que utilizam esse *framework*. As principais classes são *OpenNI*, *Device*, *VideoStream* e *VideoMode*, ilustradas no diagrama de classes da Figura 3.24 e comentadas mais adiante. Embora estas classes já tenham sido citadas anteriormente (seção 3.4.2), é necessário compreender a relação entre elas.

A Classe *OpenNI* é responsável por inicializar todo o *framework* (seção f)). É através do método estático *OpenNI.initialize*() que é verificado quais os dispositivos são compatíveis e disponíveis. E, em seguida, carregados os drivers necessários para o acesso aos dispositivos no momento da execução da aplicação. Caso a aplicação não utilize esse método irá apresentar um erro comum no momento de sua execução. Será lançado uma exceção com o nome de *RuntimeException*.

4 Por ser trivial, assume-se que o leitor já possua os conhecimentos necessários para se criar um novo projeto java no NetBeans, bem como adcicionar bibliotecas .jar.

A Classe *Device* é a responsável por fazer uma conexão com o dispositivo físico ou virtual (seção a)). É a partir da sua instância que os dados necessários para o acesso aos sensores DEPTH, RGB e IR são obtidos.

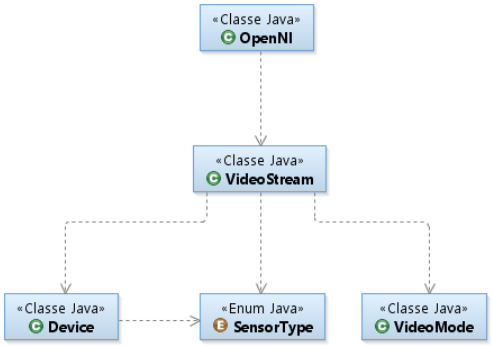

Figura 3.24. Diagrama das principais classes, comum na maioria das aplicações qualquer aplicação *OpenNI*

O trecho de código ilustrado na Figura 3.25 mostra como inicializar a biblioteca e criar um objeto do tipo Device

```
14  public class ExampleNI {
15
16      public static void main(String[] args) {
17          OpenNI.initialize();
18          Device device = Device.open();
19      }
20  }
```

Figura 3.25. Inicializando o *OpenNI* e criando um *Device*

A Classe *VideoStream* é a classe central do *OpenNI*. A partir dela é possível ler os dados provenientes do dispositivo(s) conectado(s) e encapsulá-los em um único fluxo, que além de fornecer os dados de vídeo, também fornece informações como campo de visão, modos compatíveis de vídeo, valores máximos e mínimos de pixels válidos, entre outras (seção c).

```
16  public class ExampleNI {
17
18      private static VideoStream videoStream;
19
20      public static void main(String[] args) {
21          OpenNI.initialize();
22          Device device = Device.open();
23          videoStream = VideoStream.create(device, SensorType.COLOR);
24      }
25  }
```

Figura 3.26. Criando um VideoStream

Para se capturar dados de um sensor, é necessário se criar de um fluxo de dados vinculado a algum tipo de sensor existente no dispositivo físico. O trecho de código que se segue (Figura 3.26) ilustra como criar um objeto do tipo *VideoStream* que encapsula o fluxo de dados proveniente de um sensor RGB.

Na linha 23, pode-se observar que no parâmetro do método *VideoStrem.create* (*device*, *SensorType.COLOR*), existe uma referência ao dispositivo físico (*device*) e a um tipo de sensor (*SensorType.COLOR*) ao qual o *VideoStream* deverá ser associado (ver seção g)). Neste caso, o *SensorType*.COLOR está se referindo ao sensor RGB, mas poderia ter sido associado a outros sensores, como os de profundidade e o de infravermelho.

A Classe *VideoMode* encapsula um grupo de informações do *VideoStream*. Como por exemplo, a resolução da imagem capturada, a taxa de quadros por segundo (fps) e o formato de pixel (seção e)).

O trecho de código da Figura 3.27 configura o sensor RGB no modo de funcionamento padrão (resolução de 640x480 pixels, taxa de 30 fps e formato dos pixels de 24 bits RGB). Na linha 26, através do método *getSupportedVideoModes*().*get*(1), é capturado o modo de funcionamento padrão do sensor. Em seguida, na linha 27, o *VideoMode* desejado é passado como parâmetro para o sensor RGB.

```
18  public class ExampleNI {
19
20      private static VideoStream videoStream;
21
22      public static void main(String[] args) {
23          OpenNI.initialize();
24          Device device = Device.open();
25          videoStream = VideoStream.create(device, SensorType.COLOR);
26          VideoMode mode = videoStream.getSensorInfo().getSupportedVideoModes().get(1);
27          videoStream.setVideoMode(mode);
28          videoStream.start();
29      }
30  }
31
```

Figura 3.27. Criando o VideoMode e iniciando a captura de dados do sensor

Como boa prática de programação, recomenda-se que se obtenha uma lista de configurações suportadas para um determinado sensor (utilizando o método *SensorInfo.getSupportedVideoModes*()). Em seguida, o desenvolvedor deve utilizar a lista obtida para configurar o sensor, reduzindo a possibilidade de que um modo inválido seja escolhido para o mesmo.

A partir deste ponto, o código está pronto para iniciar a captura dos dados do sensor RGB em tempo real e, para isso, basta que o método *videoStream.start*() seja chamado (Linha 28 da Figura 3.27). Porém, para que o usuário visualize os dados capturados em tela, alguns componentes gráficos do Java devem ser utilizados.

b) Visualizando os dados do sensor RGB

Embora não faça parte do *framework*, uma classe de exemplo (*SimpleViewer.java*) é disponibilizada junto com o pacote do *OpenNI*. Ela foi utilizada como modelo para criação do componente visual, denominado de *ComponentViewer*. A Classe *ComponentViewer* estende a classe *java.awt.Component* e implementa a interface *NewframeListener* da classe *VideoStream*. Ela serve basicamente para receber dados de um *VideoStream* e exibi-los na tela.

```
18  public class ExampleNI {
19
20      private static VideoStream videoStream;
21      private static ComponentViewer viewer;
22
23      public static void main(String[] args) {
24          OpenNI.initialize();
25          Device device = Device.open();
26          videoStream = VideoStream.create(device, SensorType.COLOR);
27          VideoMode mode = videoStream.getSensorInfo().getSupportedVideoModes().get(1);
28          videoStream.setVideoMode(mode);
29          viewer = new ComponentViewer();
30          viewer.setStream(videoStream);
31          viewer.setSize(mode.getResolutionX(), mode.getResolutionY());
32      }
33  }
```

Figura 3.28. Criação e configuração do component visual responsável por exibir os dados do sensor em tela

O trecho de código da Figura 3.28, nas linhas 29,30 e 31, ilustra a criação do componente visual, a passagem da referência do sensor utilizado e a configuração do tamanho em tela. Após o componente visual criado e configurado, se faz necessário

adicioná-lo a um *JFrame* e informar as propriedades necessárias para sua correta exibição em tela (Figura 3.29).

```
18  public class ExampleNI {
19
20      private static VideoStream videoStream;
21      private static ComponentViewer viewer;
22      private static JFrame frame;
23
24      public static void main(String[] args) {
25          OpenNI.initialize();
26          Device device = Device.open();
27          videoStream = VideoStream.create(device, SensorType.COLOR);
28          VideoMode mode = videoStream.getSensorInfo().getSupportedVideoModes().get(1);
29          videoStream.setVideoMode(mode);
30          viewer = new ComponentViewer();
31          viewer.setStream(videoStream);
32          viewer.setSize(mode.getResolutionX(), mode.getResolutionY());
33          frame = new JFrame("Example NI");
34          frame.setSize(viewer.getWidth() + 20, viewer.getHeight() + 80);
35          frame.add("Center", viewer);
36          frame.setVisible(true);
37          frame.setDefaultCloseOperation(frame.EXIT_ON_CLOSE);
38      }
39  }
```

Figura 3.29. Iniciando a captura do fluxo de dados do sensor RGB e exibindo o *ComponentViewer* em um *JFrame*

3.5.5 Simulando um *Kinect* fisicamente conectado ao computador

As seções anteriores mostraram como acessar um determinado sensor e iniciar a captura de seu fluxo de dados (seção a)). Nesta parte do texto será demonstrado como capturar o fluxo de dados de um sensor e armazená-lo em disco para futura utilização em uma simulação de dispositivo físico.

a) Gravando o fluxo de dados de um sensor em arquivo

Após a criação e conexão com um sensor físico, demonstradas na seção a), é possível armazenar o fluxo de dados obtido em um arquivo do disco rígido através da classe *Recorder*. Antes de descrever o funcionamento desta classe, cabe ressaltar que ela não faz conexão com um sensor físico. Esta função fica a cargo da classe *VideoStream* descrita anteriormente. A classe *Recorder* se limita a receber uma ou mais *streams* e um caminho de arquivo e gravar o fluxo de dados dessa(s) *stream*(s) no arquivo passado como parâmetro.

Para fazer a gravação em arquivo se faz necessário utilizar um objeto do tipo *Recorder* (já descrita na seção k)). Como parâmetro deve-se passar o caminho completo onde se deseja que o arquivo[5] seja salvo.

```
52          //Criando uma gravadora e especificando o nome e extensão
53          //do arquivo onde serão gravado os dados
54          Recorder recorder= Recorder.create("Exemplo.ONI");
```

Figura 3.30. Criando um Recorder e um arquivo .ONI

```
55          //Adicinando uma stream a qual fornecerá os dados a serem gravados
56          //e fornecendo a informação que diz se o arquivo deverá ser gravado de
57          //maneira compactada ou não.
58          recorder.addStream(videoStream, false);
59          //Iniciando o processo de obtenção dos dados
60          videoStream.start();
61          //Iniciando o processo de gravação dos dados
62          recorder.start();
```

Figura 3.31.Iniciando a classe recorder

Os trechos de códigos das FigurasFigura 3.30 e Figura 3.31 ilustram, respectivamente: (i) como criar e configurar um *Recorder* para que as informações

[5] Os arquivos utilizados pela classe *Recorder* obrigatoriamente deverão ter extensão ".oni".

capturadas sejam salvas no arquivo “./Exemplo.oni” e (ii) como iniciar a gravação do fluxo de dados capturados pelo *videoStream* (Linha 62).

b) Reproduzindo o arquivo com a gravação dos dados

Após o arquivo “.oni” ter sido gravado (seção a)), já é possível utilizá-lo como fonte de dados para simular um dispositivo físico. Nas etapas seguintes serão descritos os passos necessários para fazer essa simulação. Todas as etapas aqui descritas são bem semelhantes ao processo de obtenção de dados da seção a). Por esse motivo, em alguns casos, ao invés de descrevê-las novamente, apenas serão citadas as seções referentes às respectivas explicações.

Primeiro, deve-se criar um dispositivo virtual que segue os mesmos fundamentos do método *Open*(*String URI*) da classe *Device* (seção a)). A diferença é que, ao invés de se indicar no parâmetro URI o caminho de um dispositivo físico, será indicado o caminho e nome do arquivo “.oni” gravado em disco (seção a)). A Figura 3.32 mostra como criar um dispositivo através de um arquivo em disco.

```
12          //Inicializando o OpenNI
13          OpenNI.initialize();
14          //Criando um Device a partir de um arquivo .oni criado por um Recorder
15          Device device=Device.open("Exemplo.ONI");
```

Figura 3.32.Figure 1.32. Criando um Device a partir de um arquivo ".oni"

Depois de configurar o *Device* como dispositivo virtual, apontando para o arquivo “.oni”, deve ser inicializada a captura de um fluxo de dados de um dos sensores disponíveis, através do *VideoStream*, conforme ilustrado na linha 23 do trecho de código da Figura 3.26 (seção a)).

A classe que se utilizará para gerenciar toda a reprodução do arquivo é a *PlaybackControl* descrita na seção b). Para que fique mais claro, a Figura 3.33 mostra uma maneira correta de se obter um *PlayBackControl*, fazendo a verificação se o *Device* foi realmente configurado para ser um dispositivo virtual.

```
14          //Criando um Device a partir de um arquivo .oni criado por um Recorder
15          Device device=Device.open("Exemplo.ONI");
16          //Criando um PlaybackControl
17          PlaybackControl pbControl;
18          if(device.isFile()){
19              pbControl=device.getPlaybackControl();
20          }else{
21           JOptionPane.showMessageDialog(null, "Device isn't  a file.",
22                                         "Error", JOptionPane.ERROR_MESSAGE);
23              return;
24          }
```

Figura 3.33. Criando um PlaybackControl

Com a *PlaybackControl* é possível controlar toda a reprodução do fluxo de dados obtido de qualquer sensor virtual. O trecho de código da Figura 3.34 a seguir apresenta exemplos práticos, além de algumas funcionalidades da classe.

```
40          //Configurando tempo de execução
41          pbControl.setSpeed(1);
42          //Habilitando repetição automátic
43          pbControl.setRepeatEnabled(true);
44          //Obtendo o número total de frames da gravação
45          int numOfFrames=pbControl.getNumberOfFrames(videoStream);
46          //Iniciando obtenção dos dados
47          videoStream.start();
48          //Posicionando a execução no meio da gravação
49          pbControl.seek(videoStream, numOfFrames/2);
```

Figura 3.34. Exemplos das funcionaliddes - seek, repeat,

As funcionalidades destacadas no exemplo da Figura 3.34 permitem:

1. Saber qual a velocidade atual de reprodução, através do método *getSpeed*() ou indicar qual a velocidade desejada, através do método *setSpeed*(*float speed*) (linha 41).
2. Fazer com que a gravação possa se repetir a cada vez que chegar ao fim, através do método *setRepeatEnabled*(*boolean repeat*) (linha 43). Isso é muito útil nos casos em que a gravação não tem dados suficientes para suprir o tempo requerido pela simulação.
3. Identificar qual a quantidade total de frames da gravação para depois calcular a duração[6] da reprodução, através do método *getNumberOfFrames*(*VideoStream stream*) (linha 45).
4. Posicionar a reprodução em partes distintas da gravação, apenas fazendo uso do método *seek*(*int frame*)[7] (linha 49).

A partir deste ponto, é necessário iniciar a obtenção do fluxo, com o método *start*() do objeto *VideoStream* e exibir as imagens como mostrado na seção c) para simular a obtenção de dados de um dispositivo físico.

Para auxiliar o desenvolvedor, a Figura 3.35 mostra o código completo, necessário para se efetuar a simulação com todas as funcionalidades citadas até o momento.

```
11      public static void main(String[] args) {
12         //Inicializando o OpenNI
13         OpenNI.initialize();
14         //Criando um Device a partir de um arquivo .oni criado por um Recorder
15         Device device=Device.open("Exemplo.ONI");
16         //Criando um PlaybackControl
17         PlaybackControl pbControl;
18         if(device.isFile()){
19             pbControl=device.getPlaybackControl();
20         }else{
21          JOptionPane.showMessageDialog(null, "Device isn't  a file.",
22                                        "Error", JOptionPane.ERROR_MESSAGE);
23             return;
24         }
25         //Obtendo e referenciando um SensorType do tipo COLOR
26         SensorType type=SensorType.COLOR;
27         boolean isSupported=device.hasSensor(type);
28         //Criando uma variável VideoStream
29         VideoStream videoStream;
30        //Verificando se o dispositivo suporta sensor do tipo COLOR (RGB)
31         if(isSupported){
32         //Criando um objeto VideoStream, responsável pela devolução
33         //dos dados capturados, e referenciando-o com a variável videoStream.
34         videoStream=VideoStream.create(device, type);
35         }else{
36           JOptionPane.showMessageDialog(null, "Sensor not supported.",
37                                         "Error", JOptionPane.ERROR_MESSAGE);
38             return;
39         }
40         //Configurando tempo de execução
41         pbControl.setSpeed(1);
42         //Habilitando repetição automática
43         pbControl.setRepeatEnabled(true);
44         //Obtendo o número total de frames da gravação
45         int numOfFrames=pbControl.getNumberOfFrames(videoStream);
46         //Iniciando obtenção dos dados
47         videoStream.start();
48         //Posicionando a execução no meio da gravação
49         pbControl.seek(videoStream, numOfFrames/2);
50     }
```

Figura 3.35. Exemplo de código completo para a simulação de um dispositivo físico

[6] Para calcular o tempo total da reprodução apenas se faz necessário obter a quantidade total de frames, dividir pelo produto da velocidade de reprodução e a velocidade de captura (FLOPS).

[7] O *seek* só funciona caso o fluxo esteja em execução, caso contrário não será possível posicionar em um determinado *frame* do *stream*.

3.5.6 Criando aplicações para seguir os movimentos das mãos (Hand Tracking).

Antes de dar continuidade às explicações sobre como construir o exemplo para seguir os movimentos das mãos será feita uma breve explanação do *middleware* NITE, principal responsável pelos recursos utilizados no exemplo que se segue.

O *NiTE* é um *middleware* desenvolvido pela *PrimeSense* que utiliza o *framework OpenNI* para prover funções interativas aos aplicativos que utilizam sensores de profundidade, como por exemplo, o *Kinect* ou o *Carmine* 1.08 [OpenNI 2011],[PrimeSense 2011].

Enquanto o *OpenNI* é responsável pela comunicação e obtenção de dados oriundos dos dispositivos compatíveis e conectados ao sistema, o *NiTE* fica responsável pelo processamento e reconhecimento dos dados obtidos. Com este processamento, é possível: (i) obter quais são os usuários presentes na cena; (ii) identificar os pixels pertencentes a cada usuário e ao *background*; (iii) criar um mapeamento das principais articulações dos usuários e (iv) detectar a posição das mãos no espaço (*hand tracking*).

Dentre as diversas possibilidades oferecidas do *middleware* NITE, os exemplos que se seguem irão se restringir às características necessárias para a construção de uma aplicação com *hand tracking*. Para isso, a seção seguinte fará uma explanação apenas de algumas das características e métodos do *NITE* necessárias para a execução e o entendimento dos exemplos mostrados.

a) Criando um HandTracker utilizando o NITE.

No NITE, as classes disponíveis para a busca e identificação das mãos dos usuários presentes em uma cena são: *HandTracker*, *HandData* e *HandTrackerFrameRef*. A Tabela 3.12 descreve estas classes e sua utilização numa aplicação de *HandTracker*.

Tabela 3.12. Classes utilizadas para o HandTracker

Classes	Descrição
HandTracker	É a principal classe do algoritmo rastreador de mãos. Ela, junto com *UserTracker*, é uma das duas principais classes do *NITE*.
HandData	Esta classe encapsula os dados de uma mão durante um *frame* de detecção de mãos. Ela pode ser usada para descobrir: onde está a mão no espaço; o ID da mão e o status de rastreamento.
HandTrackerFrameRef	Contém todas as saídas, a partir de um único quadro de profundidade, do algoritmo de rastreamento da mão. Ela contém todas as mãos e gestos detectados, no momento atual.
GestureData	Esta classe armazena dados sobre um gesto que está a ser detectado. "Gestos", neste contexto, se referem aos movimentos das mãos detectados diretamente do DepthMap. Objetos desta classe armazenam os dados para uma instância específica de um determinado gesto.

O exemplo *HandTracker* exige a captura dos eventos de detecção de mão para cada *frame* enviado pelo dispositivo. Para isso o método *onNewFrame*(*HandTracker ht*) da interface *NewFrameListener*, explicado na Tabela 3.13 deve ser implementado.

Tabela 3.13. Objeto Listener responsável pela detecção da mão

Interfaces	Descrição
HandTracker.NewFrameListener	Esta é uma interface usada para reagir aos acontecimentos gerados pela classe HandTracker. Para usar essa interface, se faz necessário: • derivar uma classe a partir dela e implementar a função *onNewFrame()*. Esta é a função de retorno que será chamada quando um evento for gerado. • Criar uma nova instância de sua classe derivada. Em seguida, usar o *HandTracker.create()* • *E* adicionar seu escutador criado ao HandTracker através do método addNewFrameListener () Assim, o HandTracker lançará um evento sempre que um novo frame de detecção de mão for encontrado (*onNewFrame*). A função *de* retorno especificada será então chamada.

A linha 22 do trecho de código da Figura 3.36 mostra a inicialização do *middleware* NITE com o método *NITE.initialize*(). A linha 23 cria um rastreador de mãos e a 24, a detecção do gesto "mãos levantadas" é iniciada. Cabe destacar ainda, na linha 24, o método *startGestureDetection*(*GestureType.HAND_RAISE*). Ele é responsável por inicializar o algoritmo de detecção de mãos, para qualquer mão que aparecer realizando o gesto especificado pelo *Enum GestureType*. A Tabela 3.14 descreve os demais tipos de gestos disponíveis no *GestureType*.

```
16  public class ExampleNITE_HandTracker implements HandTracker.NewFrameListener {
17
18      private static HandTracker handTracker;
19      private static HandTrackerFrameRef handFrameRef;
20
21      public static void main(String[] args) {
22          NiTE.initialize();
23          handTracker = HandTracker.create();
24          handTracker.startGestureDetection(GestureType.HAND_RAISE);
25      }
26
27
28      @Override
29      public void onNewFrame(HandTracker ht) {
30          try {
31              handFrameRef = handTracker.readFrame();
32              for (GestureData gesture : handFrameRef.getGestures()) {
33                  if (gesture.isComplete()) {
34                      handTracker.startHandTracking(gesture.getCurrentPosition());
35                  }
36              }
37          } catch (Exception ex) {
38          }
39      }
```

Figura 3.36. Criação de uma aplicação NiTE e indicação do gesto a ser rastreado

Tabela 3.14. Tipos de gestos disponíveis no Enum GestureType

Gesto	Descrição
GestureType.CLICK	Gesto de click com a mão
GestureType.WAVE	Gesto de tchau
GestureType.HAND_RAISE	Mãos levantadas

Após terem sido iniciados o *middleware*, a detecção de gesto e o rastreamento das mãos, falta verificar se o gesto realmente ocorreu por completo e capturar a posição da mão que o fez. Para isso, é necessário que sejam capturadas todas as referências dos *frames* relacionados com a detecção de mão e obtidos pelo algoritmo de *HandTracker*, através do código da linha 31 da Figura 3.36.

Ainda na mesma figura, as linhas 32, 33 e 34, estão associadas, respectivamente, às seguintes ações: (i) busca dos gestos detectados pelo *handFrameRef* ; (ii) verificação se o gesto foi totalmente finalizado, através do método *isComplete*(); e, (iii) o rastreamento da posição da mão que executou tal gesto, caso tenha sido finalizado.

```
42      public void printCoordinates() {
43          try {
44              for (HandData hand : handFrameRef.getHands()) {
45                  if (hand.isTracking()) {
46                      com.primesense.nite.Point2D<Float> pos =
47                              handTracker.convertHandCoordinatesToDepth(hand.getPosition());
48                      System.out.println(pos.getX().intValue() + "  " + pos.getY().intValue());
49                  }
50              }
51          } catch (Exception ex) {
52          }
53      }
```

Figura 3.37. Código para imprimir as coordenadas da mão em um plano 2D

Até este ponto, já é possível fazer o rastreamento das mãos. Porém, para que seja possível acompanhar sua posição no espaço, foi criado um método *printCoordinates*(), que imprime as coordenadas cartesianas 2D da mão, no console, caso esta mão esteja sendo monitorada (Figura 3.37). Para que o método seja chamado a cada 200 ms, a *thread* atual é alterada conforme ilustra o código da Figura 3.38.

```
55          void run() {
56            synchronized (this) {
57                while (true) {
58                    try {
59                        Thread.sleep(200);
60                        printCoordinates();
61                    } catch (InterruptedException e) {
62                        e.printStackTrace();
63                    }
64                }
65            }
66      }
```

Figura 3.38. Criando a thread para chamar o método *printCordinates*() de tempos em tempos

Para finalizar o exemplo, no método *main* da aplicação, é criada uma instância da classe atual (linha 27), passando sua referência ao *Listener HandTracker* do *middleware* NiTE (linha 28). Em seguida, é iniciada a captura dos eventos gerados (linha 29), conforme ilustra o trecho de código da Figura 3.39.

```
23      public static void main(String[] args) {
24          NiTE.initialize();
25          handTracker = HandTracker.create();
26          handTracker.startGestureDetection(GestureType.HAND_RAISE);
27          final ExampleNITE_HandTracker app = new ExampleNITE_HandTracker();
28          handTracker.addNewFrameListener(app);
29          app.run();
30      }
```

Figura 3.39. Adicionando a classe *ExampleNite_HandTracker* ao *Listener* e iniciando a thread

b) Modificando o HandTracker para exibir imagens nas mãos dos usuários em cena

Com o intuito de chamar a atenção do leitor para algumas possibilidades de interação com o ambiente virtual, serão fornecidos conceitos básicos de exibição e interação com um ícone ou uma imagem. Através desses conceitos e com poucas modificações, o desenvolvedor poderá interagir com esses objetos virtuais através de movimentos e gestos com as mãos.

Para ilustrar uma aplicação mais visual e interativa, será aproveitado o exemplo que mostra os dados do sensor RGB da seção b), adicionando as funcionalidades de

rastreamento de mãos dos usuários presentes em cena (seção 3.5.6) e sobreposição dessas mãos com uma determinada imagem.

As partes destacadas no código ilustrado pela Figura 3.40 foram as modificações realizadas sobre o exemplo *ExampleNI* (seção b)). Porém, a classe auxiliar *ComponentViewer* também recebeu modificações com a adição de mais dois métodos, *setHandTracker*(*HandTracker tracker*) e *setImage*(*String image*), explicados adiante.

```
20  public class ExampleNI2 {
21
22      private static VideoStream videoStream;
23      private static ComponentViewer viewer;
24      private static JFrame frame;
25      private static HandTracker handTracker;
26
27      public static void main(String[] args) {
28          OpenNI.initialize();
29          Device device = Device.open();
30          videoStream = VideoStream.create(device, SensorType.COLOR);
31          VideoMode mode = videoStream.getSensorInfo().getSupportedVideoModes().get(1);
32          videoStream.setVideoMode(mode);
33          viewer = new ComponentViewer();
34          viewer.setStream(videoStream);
35          viewer.setSize(mode.getResolutionX(), mode.getResolutionY());
36          frame = new JFrame("Example NI2");
37          frame.setSize(viewer.getWidth() + 20, viewer.getHeight() + 80);
38          frame.add("Center", viewer);
39          frame.setVisible(true);
40          frame.setDefaultCloseOperation(frame.EXIT_ON_CLOSE);
41          videoStream.start();
42
43          handTracker = HandTracker.create();
44          handTracker.startGestureDetection(GestureType.HAND_RAISE);
45          viewer.setHandTracker(handTracker);
46          viewer.setImage("./TTAir.PNG");
47      }
48  }
```

Figura 3.40. Classe executável do exemplo modificado para exibir imagens sobrepondo as mãos dos usuários

O método *setHandTracker*(*HandTracker tracker*) é responsável, basicamente, por passar a referência do rastreador de mãos para o *ComponentViewer* e adicionar o *listener* de *frames* relacionados com detecção de mãos. Assim, o *ComponentViewer* poderá ter o controle do rastreamento das mãos, capturando suas coordenadas no espaço. A Figura 3.41 ilustra o trecho de código com as modificações necessárias para acrescentar essa funcionalidade.

```
17  public class ComponentViewer extends Component
18          implements VideoStream.NewFrameListener, HandTracker.NewFrameListener {
19
20      int[] mImagePixels;
21      VideoStream mVideoStream;
22      VideoFrameRef mLastFrame;
23      BufferedImage mBufferedImage;
24      HandTracker mTracker;
25      HandTrackerFrameRef handFrameRef;
26      String fileName;
27      BufferedImage image;
28
29      public void setHandTracker(HandTracker tracker) {
30          mTracker = tracker;
31          mTracker.addNewFrameListener(this);
32      }
```

Figura 3.41.Trecho de código ilustrando a adição do método *setHandTracker*()

Para dar continuidade ao exemplo, se faz necessário carregar a imagem passada como parâmetro, do disco para a memória. Assim, com a referência da imagem em memória, será possível implementar o método para desenhá-la em tela. A Figura 3.42 exibe o trecho de código necessário para executar tal funcionalidade. Neste trecho, o método *setImage*(*String image*) passa a referência do caminho da imagem no disco e chama o método *loadImage*() que faz a carga da imagem para memória.

Para exibição das imagens, nas posições onde estão sendo rastreadas as mãos dos usuários em cena, se faz necessário modificar o método *paint*(*graphics g*) do componente visual responsável pela exibição em tela. O trecho de código da Figura 3.43 exemplifica uma maneira de desenhar uma determinada imagem sobrepondo a mão detectada e rastreada pelo middleware NITE.

```
84      public void setImage(String image) {
85          this.fileName = image;
86          this.image = loadImage();
87      }
88
89      private BufferedImage loadImage() {
            BufferedImage image = null;
91          try {
92              image = ImageIO.read(new File(fileName));
93          } catch (Exception erro) {
94              System.out.println("Arquivo não encontrado");
95          }
96          return image;
97      }
```

Figura 3.42. Trecho de código que ilustra os métodos necessários para carregar uma imagem em memória (*setImage(String image) e loadImage()*)

```
71          try {
72              for (HandData hand : handFrameRef.getHands()) {
73                  if (hand.isTracking()) {
74                      com.primesense.nite.Point2D<Float> pos =
75                              mTracker.convertHandCoordinatesToDepth(hand.getPosition());
76                      g.drawImage(this.image, framePosX + pos.getX().intValue() - 20,
77                              framePosY + pos.getY().intValue() - 20, this);
78                  }
79              }
80          } catch (Exception ex) {
81          }
```

Figura 3.43. Bloco de código pertencente ao método *paint(Graphics g)* da classe *ComponentViewer*

Por fim, conforme já explicado no exemplo da Figura 3.36 da seção a), se faz necessário a implementação do método *onNewFrame*(*HandTracker tracker*) para fazer o rastreamento da mão que executou um determinado gesto.

```
137     @Override
        public void onNewFrame(HandTracker tracker) {
139         try {
140             handFrameRef = mTracker.readFrame();
141             for (GestureData gesture : handFrameRef.getGestures()) {
142                 if (gesture.isComplete()) {
143                     mTracker.startHandTracking(gesture.getCurrentPosition());
144                 }
145             }
146         } catch (Exception ex) {
147         }
148     }
```

Figura 3.44. Trecho de código que implementa do método da interface *HandTracker.NewFrameListener*

Após todas as modificações até aqui citadas, já é possível visualizar o exemplo em execução conforme ilustra a Figura 3.45. No exemplo, a logomarca do grupo de pesquisa *Touching The Air* é exibida nas mãos dos usuários em cena, após os mesmos executarem com êxito um determinado gesto monitorado pelo *middleware*.

Figura 3.45.Tela de uma aplicação que reconhece um gesto, e inicia o rastreamento da mão exibindo uma imagem na posição da mão rastreada.

3.6. Conclusão

Este capítulo tratou do desenvolvimento de aplicações voltadas para interação natural utilizando o *framework OpenNI*. A intenção não foi esgotar o assunto, mas levantar questões importantes e divulgar algumas das tecnologias mais recentes ligadas à Interação Natural. Foram apresentados alguns conceitos sobre a Interação Natural, dispositivos envolvidos e sua evolução, assim como uma explicação mais detalhada sobre o *Kinect* e toda a motivação para as pesquisas relacionadas a esta plataforma. As seções finais do capítulo foram dedicadas ao *framework OpenNI* e à sua aplicação em um estudo de caso que mostrou o desenvolvimento, passo a passo, de aplicações em Java para ambientes interativos.

Como foi observado no texto, as interfaces voltadas à interação natural trazem consigo vários desafios e oportunidades para os desenvolvedores de software e produtores de dispositivos. Atualmente, nota-se uma tendência de utilização de plataformas similares ao *Kinect*, não apenas na sua função original de controle de jogos, mas também como um novo periférico de entrada para a comunicação usuário computador.

Outro ponto importante é que o *framework OpenNI* está se tornando a principal aposta dos desenvolvedores de aplicações e comunidades científicas de interação natural. Não só por ser um excelente *framework*, mas por ter se tornado uma plataforma padrão (de facto) de desenvolvimento para estas aplicações. O projeto *OpenNI* abordado no capítulo oferece uma padronização de desenvolvimento, uma abstração maior dos elementos de hardware e uma facilidade de integração de outros componentes de softwares para interpretação dessas informações. Isto que permite tanto reuso das implementações feitas, quanto a leitura conjunta de sensores de mesma natureza aumentando a acuidade da informação.

3.7. Agradecimentos

Os autores agradecem à CAPES pelo suporte financeiro ao primeiro autor através da bolsa do Programa de Demanda Social para o Programa Multiinstitucional de Pós-graduação em Ciência da Computação UFBA/UNIFACS/UEFS (28001010061P1), à PROPEX do Instituto Federal de Sergipe e à *PrimeSense Corporation* pelos equipamentos e apoio.

Referencias

Cordeiro Jr. A. A. A. Modelos e Métodos para Interação Homem-Computador Usando Gestos Manuais. Doutorado, LNCC, 2009.

Valli A. Natural interaction white paper, 2007.

Gregory D. Abowd and Elizabeth D. Mynatt. Charting past, present, and future research in ubiquitous computing. ACM Trans. Comput.-Hum. Interact., 7(1):29– 58, Mar 2000.

BlaGames. Games: Manual de instruções power glove, 2008.

Wiiremote Commander. Wiiremote commander, 2010.

Adafruit Corporation. Open kinect challenge, 2011.

Crawford. How microsoft kinect works, 2010.

Gallud, J.A. and Villanueva, P.G. and Tesoriero, R. and Sebastian, G.and Molina S., and Navarrete A. Gesture-based interaction: Concept map and application scenarios. In Advances in Human-Oriented and Personalized Mechanisms, Technologies and Services (CENTRIC), 2010 Third International Conference on, pages 28–33, 2010.

Heckel, P. Software amigável. Técnicas de projeto de software para uma melhor interface com o usuário. Ed. Campus Ltda, Rio de Janeiro, 1993.

Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, and V. Strong. Curricula for Human Computer Interaction. ACM SIGCHI, 1996.

Kinecthacks. Hacks para o kinect, 2010.

Microsoft. Kinect web site, 2010.

Netbeans. Netbeans ide - the smarter and faster way to code. 2012

Nintendo Organization. Wii console controllers, 2009.

OpenKinect Organization. Open kinect roadmap, 2011.

OpenNI Organization. OpenNI corporation web site, 2011.

PrimeSense Organization. Primesense natural interaction, 2011.

Shape Quest. Point cloud data from structured light scanning,2010.

A.N. Rehem Neto, C.A.S. Santos, and M. V. R. Andrade. Interfaces para aplicações de interação natural baseadas na api OpenNI e na plataforma kinect. Tópicos em Banco de Dados, Multimídia e Web / XVII Webmedia / XXVI SBBD. Florianópolis, Brazil: Sociedade Brasileira de Computação, 2011, v. 1, p. 35-60.

A.N. Rehem Neto, C.A.S. Santos, and L. A. Carvalho. Touch the air: Um *framework* orientado a eventos para ambientes interativos. *In* Webmedia, 2013 (*to appear*).

Microsoft Research. Kinect for windows sdk, 2010.

Saffer. Designing Gestual Interfaces. Ot'Reilly, 2009.

Shiratori, T. and Hodgins ,J. K.. Accelerometer-based user interfaces for the control of a physically simulated character. In ACM SIGGRAPH Asia 2008 papers, SIGGRAPH Asia '08, p.123:1–123:9, New York, NY, USA, 2008. ACM.

Shiratori, T and Hodgins, J. K.. Accelerometer-based user interfaces for the control of a physically simulated character. ACM Trans. Graph., 27(5):123:1– 123:9, December 2008.

Nakra T., Ivanov Y., Smaragdis P., and Ault C. The usb virtual maestro: an interactive conducting system. NIME2009, pages 250–255, 2009.

Toyama, K. Look, ma - no hands! - hands-free cursor control with real-time 3D face tracking, 1998.

Twiky, K.. Using Nintendo wiimote with kyma, 2010.

Weiser, M, and Brown, J. S.. Designing calm technology. Powergrid Journal, 1, 1996.