OpenNI e Suas Aplicações

Lorena A. Rezende, Dalton M. Tavares

Departamento de Ciência da Computação – Universidade Federal de Goiás (UFG)
Avenida Dr. Lamartine Pinto de Avelar, 1120, Setor Universitário, Catalão–GO – Brasil

{lorennarezende}@live.com, {dalton.tavares}@catalao.ufg.br

Abstract. *This paper aims to show the results of the combination between the framework OpenNI and the Kinect sensor, used to assist the development of applications based on natural interaction. The use of controllers like keyboards, mouse and remote controls to manage an application is no longer interesting and therefore, are getting obsolete. This opens the way to a more natural interaction between human and machines.*

Resumo. *Este artigo tem como objetivo apresentar o resultado da combinação do framework OpenNI com o sensor kinect, usado para auxiliar no desenvolvimento de aplicações baseadas em interação natural. O uso de controladores como teclados, mouses e controles remotos para manipular uma aplicação já não é mais tão interessante e por isso, estão se tornando obsoletos. Isso abre o espaço a uma interação mais natural entre humano e máquinas.*

1. Introdução

O OpenNI[1] (*Open Natural Interface)* é uma organização sem fins lucrativos que disponibiliza um SDK (*Software Development Kit*) de código aberto (*open source*), o que o torna um *framework* muito útil no auxílio ao desenvolvimento de interfaces (entre dispositivos, aplicações e bibliotecas de *middleware*) para aplicações baseadas em interação natural *(natural interface* ou NI*),* principalmente por meio da audição e visão.

O OpenNI suporta aplicações em 3 dimensões. Um exemplo da sua utilização é no auxílio à engenheiros de turbina, para otimizar seus projetos de forma interativa, permitindo que os usuários possam modificar modelos, re-executar simulações e inspecionar os resultados. Nesse contexto, a plataforma OpenNI e o *middleware* NITE são responsáveis por processarem a profundidade de imagens. A aplicação reconhece as movimentações da mão para modificar lâminas de turbina, em parâmetros como "direita/esquerda" para girar, "cima/baixo" para rotação/elevação, "frente/trás" para aumentar ou diminuir *zoom* e, quando a mão é aberta, a aplicação reconhece que é para parar o processo de interpretação [Rogge et al. 2012].

O Kinect é um sensor desenvolvido em 2010 pela Microsoft® para o console de jogos Xbox 360, o qual permite ao usuário interagir de diversas maneiras (ex. por meio de gestos, movimentos ou voz) com um jogo multimídia. Por possuir sensores de profundidade e também sensores que permitem executar o rastreamento de movimentos em tempo real, esse dispositivo vem ganhando destaque junto à comunidade científica com aplicações nos campos de robótica, interface humano computador e computação

[1]Maiores informações em: http://www.openni.org/about Acesso em: 23/01/2013.

gráfica. Um exemplo da utilização do Kinect é o desenvolvimento de uma ferramenta chamada "*Hand-Potter*" [Moore 2012], muito útil no contexto da criação de objetos virtuais. A *Hand-Potter* utiliza o Kinect e algoritmos avançados que permitem a uma pessoa criar objetos virtuais em 3 dimensões, usando apenas suas mãos. Os algoritmos reconhecem as mãos e identificam sua interatividade. Dessa maneira, ele projeta virtualmente, exatamente a forma que foi "desenhada".

O Kinect foi escolhido como dispositivo de interface com o intuito de explorar os recursos oferecidos pelo OpenNI, devido a capacidade inata desse dispositivo quanto ao reconhecimento de noção de profundidade, gestos, imagens, áudio, além de dar a robôs/aplicações a visão em 3 dimensões. No decorrer desse artigo serão apresentados os requisitos para a utilização do *framework* OpenNI e uma demonstração de sua utilização usando um sensor Kinect.

Esse artigo está organizado da seguinte forma: a seção 2 irá discutir o *framework* OpenNI, a seção 3 apresenta o sensor Kinect, a seção 4 mostra um breve estudo de caso e a seção 5 são apresentadas a conclusão e propostas de trabalhos futuros.

2. O *Framework* OpenNI

O OpenNI SDK é um kit de desenvolvimento de software *open source,* suportado em múltiplas plataformas. Além deste, existe também o OpenNI *framework*[2], utilizado como auxílio ao desenvolvimento de aplicações baseadas em interação natural.

O OpenNI possui uma variedade de aplicações pré-compiladas, que podem ser executadas logo após a instalação. Seu objetivo é padronizar a compatibilidade e interoperabilidade da interação natural, dispositivos, aplicações e *middlewares*. É compatível com suas versões anteriores, de modo que, uma aplicação feita em uma versão anterior pode ser executada usando a versão mais nova do OpenNI, sem precisar ser recompilada.

Também permite que aplicações sejam escritas sem que os desenvolvedores se preocupem com o fornecedor da tecnologia que habilita uma determinada característica (ex. visão 3D). Define os nós de produção, podendo ser do tipo *Device* (sensor), *Depth* (gera mapas profundidade), *Image* (gera mapas de imagem-cor), *IR* (gera mapas de imagem infra-vermelho), *Audio* (gera um fluxo de áudio), *Gestures* (gera retornos quando gestos específicos são identificados), *SceneAnalyser* (analisa o cenário, separando primeiro plano/fundo), *Hand* (gera retorno quando pontos de mão são identificados) e *User* (gera a representação de um usuário no espaço 3D). O *framework* é responsável por traduzir dados brutos em dados "de alto nível", transformando-os em informações que sejam mais fáceis de serem compreendidas por uma biblioteca de *middleware*. No contexto deste, ele será usado para auxiliar a execução de uma aplicação já pré-compilada.

O *framework* é baseado em eventos, realizando *callbacks* sempre que um evento ocorre. Permite que dados da execução de uma aplicação sejam gravados para serem reproduzidos posteriormente, funcionando como uma simulação. No contexto deste, se destina na interface dispositivo-*middleware*/aplicação suportando dados 3D.

[2]Para maiores informações consulte a documentação OpenNI disponível em: http://www.openni.org/openni-sdk/openni-sdk-history-2 Acesso em: 23/01/2013.

3. Kinect

O Kinect é um sensor de 3 dimensões (3D) desenvolvido em 2010 com a ajuda de Andrew Black (IEEE) e sua equipe da Microsoft Research Cambridge, bem como a empresa de semicondutores PrimeSense (tecnologia que possibilita aos dispositivos observar um cenário em 3 dimensões) [Ackerman 2011]. O Kinect possui sensores RGB, infra-vermelho, CMOS e dois microfones embutidos. Foi projetado para o console XBOX 360, com o objetivo de que o controlador seja o próprio jogador (a). Dessa forma, o jogador (a) é responsável por controlar a execução de um game por meio de movimentos corporais. O Kinect consegue abstrair as informações do ambiente usando o laser do sensor infra-vermelho para determinar a distância para cada pixel. Essa informação é mapeada sobre uma imagem a partir de uma câmera RGB padrão. Cada pixel tem uma cor e uma distância, podendo então, ser utilizado para mapear posições do corpo, gestos, movimentos ou ainda gerar mapas 3D [Wang et al. 2012].

A Microsoft lançou o Kinect para o console XBox 360, porém, por ser robusto e de baixo custo, vem sendo utilizado em pesquisas nos campos de robótica e interface humano-computador. No contexto deste, o uso do Kinect se destina a reconhecer um usuário em cena e rastrear seus movimentos.

4. Estudo de Caso

Para utilizar os recursos oferecidos pelo *framework* OpenNI, primeiramente, é necessário ter instalado os seguintes componentes[3]:

- GCC 4.x;
- Python 2.6+/3.x;
- LibUSB 1.0.x;
- FreeGLUT3;
- JDK 6.0;
- OpenNI 1.5.x.x.

Os componentes supracitados são requisitos para se usar os recursos do *framework* OpenNI. Além disso, é necessário instalar o *middleware* NITE e o Sensor PrimeSense de modo a habilitar a operação do Kinect. O NITE é uma biblioteca de *middleware* disponibilizada pela PrimeSense que contém algoritmos para rastreamento de movimentos. Utilizando um sensor, como o Kinect, torna-se possível rastrear os movimentos de um ponto da mão (*hand-point*) ou ainda, movimentos do corpo (*full-body)*. O sensor PrimeSense é uma abstração necessária para a aquisição de dados de um dispositivo e calcular a profundidade.

Quando uma aplicação é executada, os dados são gerados por um dispositivo físico que possui os sensores necessários para sua execução. Como a aplicação criada (seção 4.1 e 4.2) é baseada em 3D, a ferramenta utilizada foi o Kinect. Em nível de hardware, ele possui os sensores necessários: RGB, infra-vermelho (IR) e CMOS.

[3] Disponível em: http://www.openni.org/Downloads/OpenNIModules.aspx. Acesso em: 10/01/2013.

Existe uma camada (SoC – *System on Chip*)[4] que opera sobre o hardware descrito. Esta é responsável por calcular a profundidade, onde um sensor infra-vermelho ilumina a cena e um sensor CMOS decifra a luz recebida pelo infra-vermelho e produz uma imagem de profundidade VGA.

O *framework* OpenNI permite que o *host* (máquina) opere os sensores e acesse todos os dados brutos gerados por ele. O *framework* então, os traduz para informação de alto nível e os enviam para a biblioteca de *middleware* NITE, a qual possui os algoritmos de rastreamento necessários para a execução da aplicação. Uma representação da arquitetura descrita é apresentada na Figura 1.

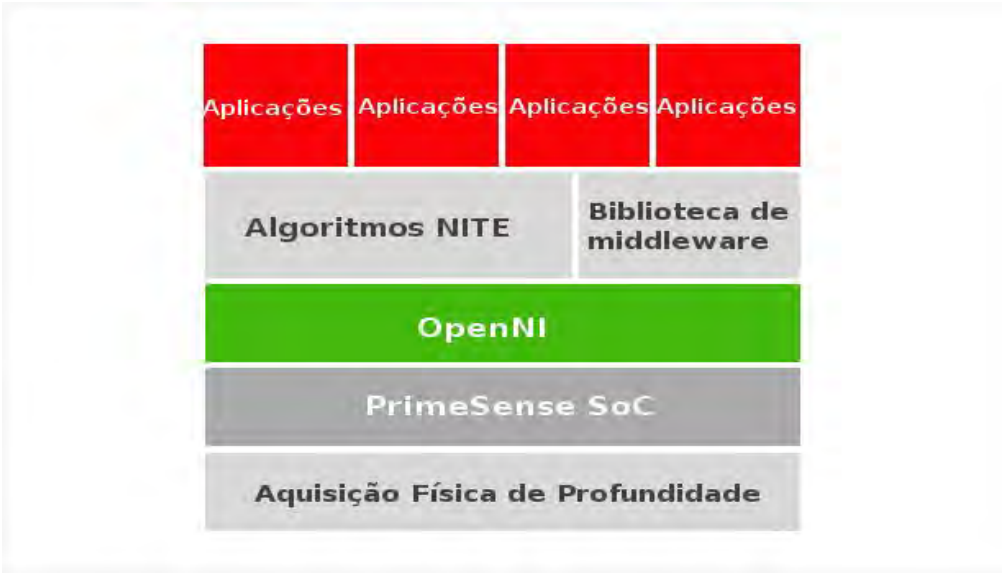

Figura 1. Modelo de representação de uma aplicação com interface natural baseada em dados 3D.[5]

4.1. Aplicação Desenvolvida

A aplicação criada é responsável por identificar um usuário em cena e rastrear seus movimentos. Para sua execução é utilizado o *framework* OpenNI, a biblioteca de *middleware* NITE e como dispositivo físico, o sensor Kinect. A aplicação possui três porções principais representadas pelos seguintes arquivos[6]: uma bilioteca "`opengles`" que permite a reprodução gráfica dos resultados obtidos, o "`SceneDrawer`" que processa os eventos (desenha) e o programa principal "`main`", onde são realizadas as chamadas de funções e *callbacks*.

No programa principal (`main`), são declaradas variáveis globais dos tipos `Context`, `ScriptNode`, `DepthGenerator`, `UserGenerator` e `Player`. O `Context` é um objeto usado para criar os gráficos de produção. Estes são representados por um conjunto de nós que foram produzidos, bem como outros nós os quais são dependentes. Por exemplo, para obter um nó do tipo `UserGenerator` é necessário que exista um nó

[4]Mais informações em: http://www.primesense.com/solutions/technology Acesso em: 28/01/2013.
[5]Imagem original disponível em: http://www.primesense.com/solutions/nite-middleware Acesso: 29/01/2013.
[6]Para maiores informações consulte a documentação OpenNI Compliant disponível em: http://www.openni.org/openni-sdk/openni-sdk-history-2 Acesso em: 23/01/2013.

`DepthGenerator`. No caso de haver mais de um sensor com a mesma funcionalidade, a aplicação pode escolher o gráfico de produção preferido (estes dispositivos podem gerar dados diferentes) levando em consideração um fornecedor específico, por exemplo. O `ScriptNode` é um objeto carregado a partir de uma *string* ou arquivo de script XML, usado para a execução do gráfico de produção. O `DephGenerator` gera um mapa de profundidade, onde cada pixel no mapa tem um valor que representa a distância do mesmo para o sensor. O `UserGenerator` é um nó gerador de dados referentes à informações dos usuários em um cenário, onde cada usuário é identificado como único. Por fim, o nó `Player`, armazena a gravação dos dados de uma sessão que gera dados.

O OpenNI usa contagem por referência para que os objetos sejam compartilhados entre vários componentes da aplicação. Quando um tipo específico de nó é criado, sua referência é setada com o valor 1 e o identificador é retornado ao criador. A contagem de referência é incrementada (+1) no caso de haver novos nós que possuem o mesmo tipo de um nó já existente. Quando os nós não são mais necessários, vão sendo removidas/decrementadas (-1) as referências (por meio da função `void CleanupExit()`) até que ela tenha valor 0 e então, o nó é destruído e a memória antes alocada por ele, é liberada. Como o OpenNI suporta gravações de uma sessão geradora de nós, ao executar a aplicação, se não é passado nenhum parâmetro (o nome de um arquivo de gravação), o gráfico de produção será criado de acordo com um arquivo XML padrão do OpenNI. Caso contrário, ele iniciará a partir do arquivo de gravação especificado.

No caso apresentado, não foi passado nenhum parâmetro. Dessa forma, a execução do programa inicia normalmente, esperando por um evento específico (i.e. o surgimento de um usuário em cena). Se um usuário aparece em cena, ele tem sua posição identificada, para que em seguida o sensor possa ser calibrado e o usuário seja rastreado. Para cada novo usuário que aparece em cena, ele possui um ID (identificador) para que a aplicação possa fazer referência a cada um individualmente. Para que seus movimentos sejam rastreados, é preciso primeiro detectar sua posição. Este procedimento é feito por meio de uma chamada de *callback*, a qual recebe como parâmetros a capacidade de detectar a pose, o ID daquele usuário, e uma *string* que representa a sua posição. Dessa forma, no caso da detecção de posição ser bem sucedida, é registrada a posição e o processo de detecção de posição é finalizado e é realizada uma requisição à calibração.

Em seguida, é feita a calibração, recebendo como parâmetros a capacidade do esqueleto e o ID do usuário. É na calibração que são medidos e registrados o comprimento dos membros do usuário em cena. Se a calibração completa for executada com sucesso, é feita uma chamada para rastreamento passando como argumento o ID. Caso contrário, o processo de calibração é iniciado novamente e a reinicialização pode ser a partir da detecção de pose do usuário ou da calibração. Se o gerador de calibração exige a detecção de pose, é necessário fazer chamada à função que detecta a pose, para executá-la novamente (passando como argumento o ID do usuário). De outra forma, após realizar a calibração novamente, é passado como argumentos o ID do usuário e o valor “TRUE”, este último indica que a calibração vai ser executada novamente para aquele identificador de usuário. Se um usuário sai de cena e as informações referentes ao usuário/cenário que estavam salvas são perdidas, o usuário é declarado como um

usuário perdido e, este tem suas informações excluídas, bem como sua referência. A implementação leva em consideração algum atraso, para que se tenha a certeza de que o usuário realmente saiu do cenário.

Existe uma função (`void SaveCalibration()`) que percorre uma lista de nós existentes, verificando se existem nós calibrados. Em caso afirmativo, eles têm seu ID armazenado em um arquivo. A função (`void LoadCalibration()`) carrega os dados que foram armazenados anteriormente. Esta é muito útil considerando que os desenvolvedores podem armazenar a calibração e, testar posteriormente o aplicativo sem a necessidade de detectar a pose novamente e efetuar a calibração. No caso aqui apresentado, não foi carregado nenhum arquivo (não foi passado como parâmetro).

A função `void glutDisplay()`, é responsável por projetar na tela a representação gráfica dos dados. Por meio do `DepthGenerator` as dimensões `xRes` e `yRes` são acessadas, retornando as dimensões X e Y do *buffer* de profundidade, as quais são necessárias para obter o valor de cada pixel no mapa (de profundidade). Também são incluídos métodos que desenham o mapa de profundidade, usuários e esqueletos. O gráfico é atualizado apenas quando um nó gerador de usuários tem novos dados.

Por fim, neste arquivo, temos a função principal, onde são definidos o tamanho (em relação à altura/largura) e nome da janela onde serão projetados os gráficos de produção ("User Tracker Viewer"), ocorre a inicialização dos manipuladores de eventos os quais se verifica se a característica de esqueleto (representação gráfica virtual em 2D) de um usuário em cena é suportada fazendo com que o esqueleto se mova de acordo com a movimentação do usuário e suas respectivas dependências (ex. para fazer a calibração, é necessário que exista o suporte da capacidade de calibrar e detectar posição).

No arquivo "`SceneDrawer`" estão dispostos os processadores de evento. Ele é o responsável pela verificação da características (ex. caso seja preciso detectar uma posição do usuário antes de fazer sua calibração, ele registra a detecção de posição, caso contrário, ele não suporta detecção de posição). Esse arquivo é usado também para "desenhar" os membros dos usuários, onde a função `DrawJoint()`, recebe como parâmetros o ID do usuário, as juntas 1 e 2 (por padrão) e traça uma linha reta entre as juntas das articulações especificadas. Verifica também se há o rastreamento dos usuários, por meio de conversões: recebe as coordenadas de posição (real) e as converte em coordenadas projetivas a serem representados em um plano 2D. Além disso, gera um histograma para a criação de uma gradiente de profundidade cena-a-cena, a partir dos parâmetros: longe(escuro)/perto(claro). De acordo com os dados de profundidade, estabelece a textura de cada *pixel* de acordo com o usuário (centro de massa de um usuário). A execução da aplicação é ilustrada na Figura 2.

4.2. Procedimento experimental

Na Figura 2, existem 4 usuários em cena e cada um deles passou pelo processo descrito na seção 4.1 (detecção de pose, calibração e rastreamento). No quadro que é disposto na Figura 3, mostra-se a execução da aplicação. Note que o usuário 1 entrou em cena, teve sua posição detectada e foi calibrado (impresso no terminal). Enquanto o usuário 1 era calibrado, o usuário 2 entrou em cena, teve sua posição detectada e inicializou sua calibração, em paralelo, o usuário 1 tem sua calibração finalizada e seu processo de

rastreamento é inicializado. Quando o usuário 3 entra em cena, ele tem sua posição detectada, é calibrado e rastreado e, logo em seguida, o usuário 2 também tem sua calibração finalizada e então seu processo de rastreamento é inicializado.

Em seguida, entra em cena o usuário 4, este entra e sai da cena duas vezes sua posição é identificada, mas ocorre um erro na tentativa de calibração desse usuário, porém, quando por uma terceira vez, ele entra em cena, tem sua posição detectada e começa o processo de calibração novamente. Enquanto isso, o usuário 5 também entra em cena (e o usuário 3 sai), tem seu posicionamento detectado, mas logo depois, sai de cena. O usuário 4 tem sua calibração executada com sucesso e inicia seu processo de rastreamento. Ocorre falha na tentativa de calibrar o usuário 5 (porque ele saiu de cena).

O usuário 5 entra em cena novamente, tem seu posicionamento detectado, é calibrado com sucesso e inicia-se o processo de rastreamento para o mesmo. Por fim, o usuário 3 (que já saiu de cena há algum tempo) é declarado como perdido. Na janela gráfica "User Tracker Viewer", na Figura 2, são representados os esqueletos correspondentes aos usuários supracitados, exceto aqueles que saíram de cena, como no caso do usuário 3.

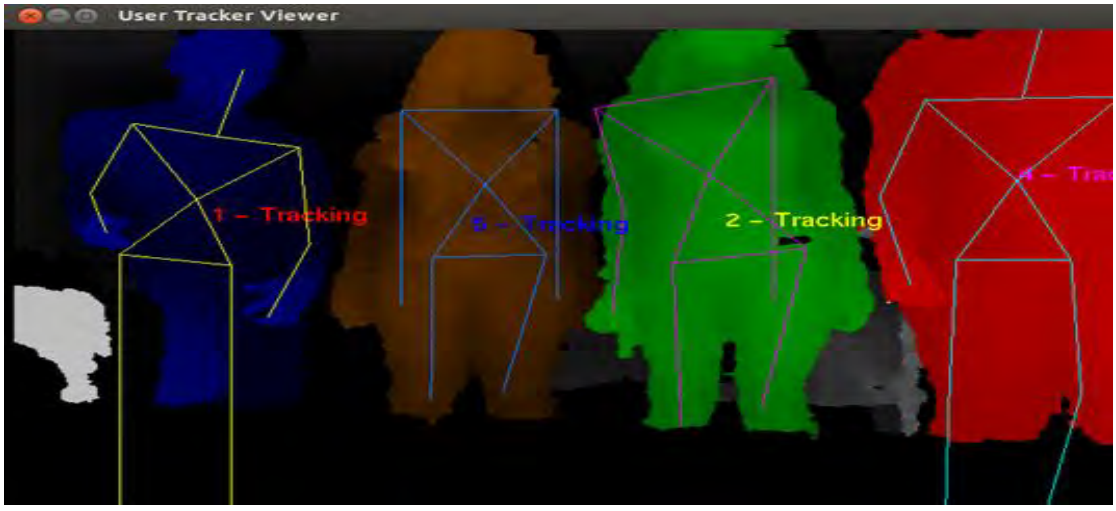

Figura 2. Representação gráfica do resultado da execução da aplicação.

```
.0/Samples/Bin/x86-Release$ sudo ./Sample-NiUserTracker
1355314366 New User 1
1355314366 Calibration started for user 1
1355314366 New User 2
1355314366 Calibration started for user 2
1355314366 Calibration complete, start tracking user 1
1355314366 New User 3
1355314366 Calibration started for user 3
1355314368 Calibration complete, start tracking user 3
1355314368 Calibration complete, start tracking user 2
1355314378 New User 4
1355314378 Calibration started for user 4
1355314383 Calibration failed for user 4
1355314383 Calibration started for user 4
1355314388 Calibration failed for user 4
1355314388 Calibration started for user 4
1355314389 New User 5
1355314389 Calibration started for user 5
1355314392 Calibration complete, start tracking user 4
1355314394 Calibration failed for user 5
1355314394 Calibration started for user 5
1355314395 Calibration complete, start tracking user 5
1355314397 Lost user 3
```

Figura 3. Relação do estado da aplicação

5. Conclusão

O presente artigo discutiu acerca do processo de execução de uma aplicação baseada em interação natural. Para tanto, foram apresentados os seus componentes, em termos de bibliotecas, algoritmos e ferramentas físicas, levando em consideração características de rastreamento de movimentos. Nesse contexto, o OpenNI *framework* foi escolhido por oferecer suporte a uma ampla variedade de tipos de dados, o que, em combinação com o sensor Kinect gera informações relacionadas à profundidade, imagem, áudio e movimentos. Com a combinação dessas duas ferramentas, é possível desenvolver uma variedade de aplicações baseadas em interação natural, não só para a área de robótica ou computação gráfica, mas também no contexto de tecnologia social, como por exemplo, no desenvolvimento de aplicações que auxiliem pacientes em fase de reabilitação.

Vale observar que a organização OpenNI, a partir da versão 2.0 não fornecerá suporte às plataformas Mac e Linux, considerando o suporte ao sensor Kinect, uma vez que este é patenteado pela Microsoft. Em contrapartida, a PrimeSense começou a fabricar dispositivos com as configurações similares às do Kinect.

Referências

Ackerman, E. (2011) “Top 10 Robotic Kinect Hacks”, In: IEEE Spectrum, http://spectrum.ieee.org/automaton/robotics/diy/top-10-robotic-kinect-hacks, Março.

Moore, S. K. (2012) “New Kinect Design Trick: Hand Potter”, In: IEEE Spectrum, http://spectrum.ieee.org/tech-talk/computing/software/new-kinect-design-trick-handy-potter, Agosto.

Rogge, S., Amtsfeld, P., Hentschel, C., Bestle, D. e Maeyer, M. (2012) “Using Gestures To Interactively Modify Turbine Blades In A Virtual Environment”, In: Emergind Signal Processing Applications (ESPA), 2012 IEEE International Conference.

Wang, Y., Yang, C., Wu, X., Xu, S. e Li, H. (2012) “Kinect Based Dynamic Hand Gesture Recognition Algorithm Research”, In: 4th International Conference on Intelligent Human-Machine Systems and Cybernetics.