RAFAEL CARVALHO SLOBODIAN

RODRIGO RICHA CAMPAGNOLI

SIMULAÇÃO DE BRAÇO MECÂNICO CONTROLADO PELO *WIIMOTE* E WII MOTION PLUS™

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Bacharel em Ciência da Computação do curso de Ciência da Computação da Universidade Federal Fluminense.

Orientador: ESTEBAN WALTER GONZALEZ CLUA

Coorientador: ANSELMO ANTUNES MONTENEGRO

Niterói

2010

RAFAEL CARVALHO SLOBODIAN

RODRIGO RICHA CAMPAGNOLI

SIMULAÇÃO DE BRAÇO MECÂNICO CONTROLADO PELO *WIIMOTE* E WII MOTION PLUS™

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Bacharel em Ciência da Computação do curso de Ciência da Computação da Universidade Federal Fluminense.

BANCA EXAMINADORA

Prof. Dr. ESTEBAN WALTER GONZALEZ CLUA – Orientador

UNIVERSIDADE FEDERAL FLUMINENSE

Prof. Dr. ANSELMO ANTUNES MONTENEGRO - Coorientador

UNIVERSIDADE FEDERAL FLUMINENSE

Profª. Drª. FLAVIA CRISTINA BERNARDINI

UNIVERSIDADE FEDERAL FLUMINENSE

Profª. Drª. DANIELA GORSKI TREVISAN

UNIVERSIDADE FEDERAL FLUMINENSE

Niterói

2010

AGRADECIMENTOS

Primeiramente à meu pai, Léo, pelo apoio incondicional. Se há alguém responsável por tudo o que sou hoje é você. Obrigado pelo amor, boa educação, criação, sabedoria e, acima de tudo, por sua amizade.

À minha esposa, Juliana, minha alma gêmea, por estar sempre ao meu lado. Muito obrigado por todo apoio, amor e carinho, mesmo nas horas mais difíceis. Eu amo você.

À meus irmãos, Alex e Rony, e minha cunhada, Thais, pelo apoio e amizade.

À meus grandes amigos da turma 204.31. Esta turma, que me acolheu desde o início, se tornou uma família, e graças à amizade e apoio de vocês que foi possível encarar esta jornada. Vocês estavam juntos no dia a dia, na véspera de cada prova. Muito obrigado por tudo, e que nunca percamos contato.

Ao Professor Esteban, pela ideia inicial e oportunidade na criação. Obrigado por manter vivo o sonho, e fazer acreditar que é possível, trabalhar com games.

Ao Professor Anselmo, acadêmico exemplar, por todo apoio e tempo dedicado as nossas dificuldades.

À meu bom amigo Marcos “Sméagol” Vinicius Policarpo Côrtes, cuja ajuda, atenção e apoio foram inestimáveis para a conclusão deste projeto.

À meu bom amigo Breno Mattos de Paula pelo apoio e ajuda no desenvolvimento e elaboração da construção dos tópicos desta monografia.

Ao Fernando Ribeiro, pela ajuda com a criação do modelo tridimensional do braço mecânico.

E, finalmente, aos Professores e amigos Célio Vinicius Neves de Albuquerque e Inhaúma Neves Ferraz, por todo apoio e amizade nesta jornada.

Rafael Carvalho Slobodian

À Deus pela vida e a Jesus Cristo pelo caminho.

Aos meus pais Edgard e Maria da Penha pela minha base educacional e comportamental. Vocês representam tudo o que eu sou em caráter e dignidade.

À meu irmão Eduardo por aumentar e melhorar minhas chances. Obrigado por olhar e orar por mim quando estava só.

À Francisca e minha avó Zilda pelas pessoas que vocês são e pelo papel de mãe que tiveram.

À minha namorada Roberta pela compreensão e carinho. Eu te amo.

À minha família pela ajuda nas dificuldades e amigos pela força e boa energia.

Ao amigo Marcos Vinicius Policarpo Côrtes, cuja ajuda, atenção e apoio foram inestimáveis para a conclusão deste projeto.

À Breno Mattos de Paula pelo apoio e ajuda no desenvolvimento e elaboração da construção dos tópicos desta monografia.

Ao Fernando Ribeiro, pela ajuda com a criação do modelo tridimensional do braço mecânico.

Rodrigo Richa Campagnoli

"Programming today is a race
between software engineers
striving to build bigger and better
idiot-proof programs

And the Universe trying to produce
bigger and better idiots

So far, the Universe is winning"

Rick Cook, The Wizardry Compiled.

RESUMO

Neste trabalho é desenvolvida uma pesquisa para permitir utilizar o Wii™ Remote combinado com a extensão Wii Motion Plus™ para fazer o controle de uma simulação de um braço mecânico. Esta pesquisa consiste da tentativa de criar uma técnica mais intuitiva para controlar este tipo de dispositivo, ao combinar o monitoramento de orientação em tempo real do controle, obtido através da interpretação do sinal fornecido pelo giroscópio do Wii Motion Plus™, com informações de deslocamento obtidas através do pressionar de botões direcionais do Wii™ Remote, e aplicá-las no efetuador de um braço mecânico tridimensional através da técnica de cinemática inversa *Cyclic Coordinate Descent* (CCD). Os resultados obtidos para esta técnica foram bastante satisfatórios, com destaque para o monitoramento preciso em tempo real de três graus de liberdade de rotação. Além disto, como apêndice neste trabalho, é realizado um estudo para interpretar as informações fornecidas pelo acelerômetro do Wii Remote, na tentativa de fazer o monitoramento contínuo do deslocamento do dispositivo. Os resultados obtidos nesse estudo não foram suficientemente satisfatórios, entretanto abrem um precedente na pesquisa da integração de informações do giroscópio com o acelerômetro para criar uma técnica de *tracking* independente de marcadores fiduciais.

Palavras-chave: Wii, Wii Remote, Wiimote, Wii Motion Plus, acelerômetro, giroscópio, robótica, braço mecânico, cinemática inversa, cyclic coordinate descent, interpretação de sinais.

ABSTRACT

In this work we developed a research to enable the use the Wii™ Remote combined with the Wii Motion Plus™ extension to control the simulation of a mechanical arm. This research consists into the attempt of creating a more intuitive technique to control this kind of device, by combining real time monitoring of orientation of the controller, obtained by interpreting the signal gathered from the gyroscope included into the Wii Motion Plus™, with displacement information generated by pressing directional buttons in the Wii™ Remote, and applying them to the end effector of a tridimensional mechanical arm through the Cyclic Coordinate Descent (CCD) inverse kinematics technique. The results obtained were very satisfactory, with highlights to the precise real time monitoring of three degrees of freedom of rotation. Beyond that, attached to this work is the research to interpret and monitor the information gathered from the Wii™ Remote's accelerometer, in the attempt to continuously track the displacement of the device. The results obtained in this last research were not sufficiently satisfactory, although they open a precedent in the research of integrating information of a gyroscope with an accelerometer to create a tracking technique independent of fiducial markers.

Keywords: Wii, Wii Remote, Wiimote, Wii Motion Plus, accelerometer, gyroscope, robotics, mechanical arm, inverse kinematics, cyclic coordinate descent, signal interpreting.

SUMÁRIO

INTRODUÇÃO

Com a industrialização e o advento da robótica, foram introduzidos nos meios de produção equipamentos mecânicos/eletrônicos para substituir a mão de obra humana. Suas vantagens sobre a mão de obra humana são inúmeras: Maior eficiência, menor custo e maior segurança aos trabalhadores. Assim, elas se tornaram essenciais para o desenvolvimento da sociedade[1].

O equipamento robótico de maior uso é sem dúvida o braço mecânico. Por se tratar de um "robô" especializado em uma tarefa e que não possui necessidades de locomoção (exigindo menos componentes), o equipamento se tornou atrativo para uso nas mais diversas áreas.

Normalmente o controle destes equipamentos é realizado por painéis munidos de botões (teclados) e manetes (joystick). Eles são programados para uma tarefa (escolhe-se uma forma de trabalho e ele realiza a operação pré-programada), ou são controlados em tempo-real por um operador. Entretanto, tal controle exige do operador conhecimento de como "operar" esta máquina de forma eficiente usando controles. O tempo de aprendizado pode ser alto devido à falta de familiaridade do operador com sistemas de entrada destes ou pela própria inadequação destes dispositivos de entrada com o braço mecânico que se quer operar.

Por tais motivos é interessante e viável a busca de novos dispositivos de entrada que sejam mais próximos de como o operador faria o trabalho sem o equipamento robótico. Assim, com um dispositivo mais "fácil" de operar, espera-se uma menor curva de aprendizagem e uma diminuição nos erros de operação.

---

[1] Não entraremos em detalhes dos danos que isto causou como a marginalização de mão de obra humana e a mais-valia gerada aos operadores destas máquinas.

Se observarmos os dispositivos de *input* de dispositivos eletrônicos existentes, existe um que tem ganhado muita notoriedade: o Wii™ Remote (ou *Wiimote*, neologismo cunhado da união do nome do console de videogame com a palavra *remote*, usada para designar controles remotos, em Inglês). Trata-se do controle do videogame Wii™, da Nintendo®. Enquanto os anteriores eram bases onde existiam botões que, quando acionados, realizam uma tarefa, o *Wiimote* possui uma forma de bastão, que lembra muito o formato de um controle remoto tradicional, e deve ser segurado por uma das mãos. Quando o usuário move o bastão, o dispositivo detecta o movimento através de um acelerômetro e passa as informações para o console. Assim ele consegue reconhecer o movimento e realizar as tarefas associadas. Se notarmos, tal comando de "mover a mão" é muito mais intuitivo que o comando de pressionar um botão. Isto é, o ser humano está muito mais familiarizado em mover os braços para realizar certas operações (como mover móveis ou arremessar uma bola) do que pressionar um botão.

Além dos motivos relacionados com a aplicação de tal técnica, existe outro fator a ser levado em consideração. Desde o início do desenvolvimento de aplicações com o *Wiimote*, devido à limitações impostas pelas características do acelerômetro[2], houve a predominância de algoritmos de aprendizagem, usando redes neurais ou cadeias de Markov, para detectar o movimento. Isto é, o sistema treinava um algoritmo de aprendizagem para relacionar dado movimento do *Wiimote* a uma ação desejada. Tal técnica é eficiente no disparo de ações discretas. Entretanto, para aplicações onde a precisão dos movimentos e o nível de detalhes são exigidos ou é necessário o rastreamento em tempo real de posição e orientação do controle, tal técnica não é aplicável. Isto é, se precisamos detectar em qual ponto do espaço o Wii Remote está e conseguir processar esta informação para realizar uma ação contínua em nosso ambiente de simulação, tais técnicas não solucionam o problema. Outra abordagem então é necessária. Para suprir essa carência, é acoplado ao controle um acessório para o *Wiimote*, chamado Wii Motion Plus™, que consiste de um sensor de giros chamado giroscópio. Neste trabalho, descrevemos como podemos captar esta informação com nível de precisão razoável com este acessório, para que representemos o movimento de rotação contínuo em nossa simulação.

[2] Limitações estas abordadas no Capítulo 2, Seção 2.1.1: *Wiimote*, tópico Acelerômetro.

Então o presente trabalho, vendo as vantagens de usabilidade do *Wiimote*, procura investigar uma forma de usá-lo para comandar um braço mecânico, permitindo ao operador realizar sua tarefa através da máquina de forma mais intuitiva e com menor curva de aprendizagem. O resultado final consiste de um controlador, cuja orientação é determinada pelo giroscópio e o deslocamento se dá através do pressionar dos botões presentes no *Wiimote*.

O apêndice deste projeto ainda apresenta os resultados da pesquisa com a implementação do monitoramento em tempo real (*tracking*) do *Wiimote* acoplado à um Wii Motion Plus™, combinando os sinais do acelerômetro e do giroscópio. Esta ideia foi inspirada pela implementação, de código fechado, criada pela empresa AiLive®, demonstrada em (AiLive, 2008).

No Capítulo 1, daremos uma breve explicação do que é um braço mecânico e como podemos usar o *Wiimote* para controlá-lo. Já no Capítulo 2 são introduzidas as tecnologias utilizadas no projeto e suas particularidades, além dos componentes de software e APIs usadas como base. No Capítulo 3 daremos detalhes da implementação da técnica usada para captar e interpretar as informações do *Wiimote* e transferi-las para o braço mecânico através de cinemática inversa. No Capítulo 4 será explicado como usar este software criado e quais são suas características. No Capítulo 5, exibiremos os resultados técnicos e analisaremos o quanto ele foi satisfatório. Em seguida, descrevemos as considerações finais sobre o projeto, exibindo prós e contras, quais os ganhos obtidos e os trabalhos futuros, e as referências bibliográficas. Por fim, no Apêndice I é detalhada a arquitetura das aplicações do XNA™ Framework, no Apêndice II é detalhada a pesquisa de integração do acelerômetro com o giroscópio e seus resultados, e no Apêndice III pode ser conferido o código-fonte gerado para esta solução.

1. BRAÇOS MECÂNICOS

Segundo (Russell, et al., 2004), um braço mecânico pode ser visto como uma espécie de robô classificada como **manipulador**. Isto é, um dispositivo robótico que não possui capacidade de movimento e está sempre ancorado em uma plataforma que possibilita seu trabalho. Ele é composto por uma série de articulações e por um dispositivo efetuador que está conectado na extremidade oposta a qual ele é fixado na plataforma de trabalho. O seu uso em fábricas de automóveis é o mais notável, já que todo o processo atualmente depende destes robôs. Também braços mecânicos são encontrados como complemento de robôs mais complexos, como, por exemplo, robôs de exploração.

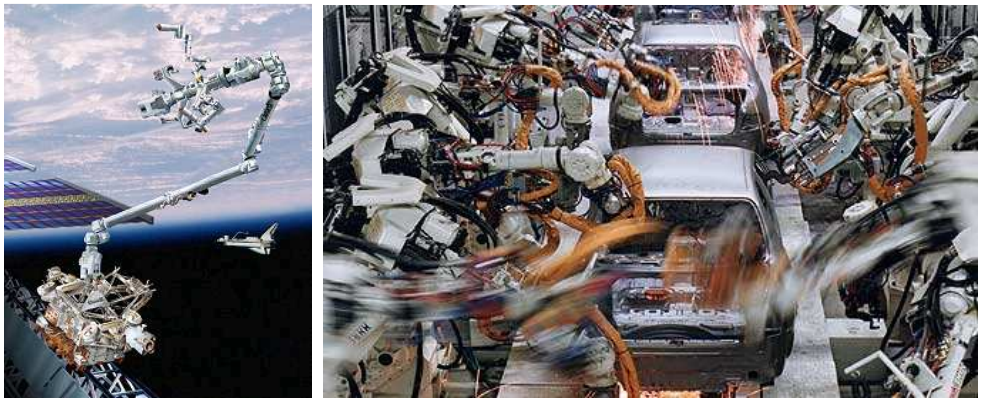
O efetuador é uma pinça ou qualquer outro dispositivo voltado para a tarefa que o braço mecânico se propõe. Este possui certo grau de liberdade, que caracteriza em quais dimensões ele pode rotacionar ou transladar. Além disto, o braço como um todo possui outros graus de liberdade, um para cada articulação que o compõe.

As áreas de atuação de braços mecânicos são inúmeras. A principal deles é a indústria, em especial nas linhas de montagem, e o principal motivo disto é que seu uso é mais barato que a mão de obra humana tradicional. Na agricultura, muitos maquinários são utilizados para extração de produtos agrícolas.

Outro problema que pode ser atacado com estes tipos de robôs são os que colocam o agente da tarefa em ambientes arriscados. Onde o risco de danificação do agente é real, o uso de braços mecânicos ajuda a poupar a vida de seres humanos como, por exemplo, na limpeza de locais radioativos, salvamento em lugares com estrutura danificada, desativação de bombas e minas, ou então permitir acesso a ambientes onde seria impossível o acesso humano, como exploração do mar profundo ou trabalho no espaço, auxiliando astronautas, como demonstrado na **Figura 1**.

Na medicina, o seu uso no auxílio em operações já é difundido. Devido à sua precisão, os traumas em pacientes decorrentes de cirurgias são reduzidos.

(a) (b)



(c)

Figura 1: Robôs e suas aplicações**.** (a) - (NASA, 2001), (b) - (The Guardian, 2009), (c) - (Intuitive Surgical, 2005).

Segundo (OSHA, 2008) e (Wikipedia, 2010), em braços mecânicos autônomos a técnica mais usada para sua programação é a aprendizagem assistida. Isto é, é preciso muitas vezes "ensiná-los" a se comportar no ambiente de trabalho. Assim, um agente externo (humano) entra em contato com a máquina e deve guiá-la para que realize a ação. Isto possui seus riscos inerentes ao ser humano. Então, desenvolver dispositivos de entrada e controle destes equipamentos ainda é algo muito importante e exige pesquisas da comunidade científica. Além disto, formas de controle remoto, onde o usuário esteja em local seguro, longe da máquina, também é desejável.

O *Wiimote* pode proporcionar isto de forma eficiente e com baixo custo, uma vez que é um dispositivo sofisticado e já difundido pela sociedade devido à grande popularidade do videogame Wii™.

1.1. DISPOSITIVOS DE ENTRADA

Podemos considerar que um braço mecânico pode ser controlado através de um software, tornando-o autônomo e sendo um robô, no sentido mais estrito da palavra. Ou

pode ser controlado diretamente por um operador, que utiliza um dispositivo de entrada para realizar os comandos. Esta segunda técnica é o foco do presente trabalho.

O uso de um teclado ou painéis com botões para programação e controle de um braço mecânico é a forma mais tradicional existente. O operador entra com os comandos desejados e o equipamento realiza o trabalho. Esta forma é muito eficiente para braços mecânicos robóticos, no qual ele já possui na memória os movimentos que deve realizar, o que faz com que durante seu trabalho não seja necessária a assistência do operador. Entretanto, quando ele não possui uma programação prévia e exige que o operador lhe dê comandos para seu funcionamento, o teclado se torna uma forma ineficiente. Típicas máquinas que exigem este grau de controle são os robôs de exploração.

Além disto, estes painéis podem ser assessorados com dispositivos que detectam coordenadas no espaço, como o mouse e o joystick. Ambos têm a mesma proposta: o mapeamento para um plano 2D de um movimento realizado (o mouse move-se sobre uma mesa, enquanto o joystick detecta movimento de um bastão, alavanca ou botão direcional).

1.2. MOVIMENTO DAS MÃOS

Segundo (Hand, 1997), uma tarefa simples de manipulação de objetos no espaço, transladando-os e girando-os (manipulação de objetos) em aplicações 3D, só é possível com os dispositivos tradicionais de entrada devido à criação de ferramentas metafóricas, como a ampliação e rotação. Isto é, não é possível manipular o objeto diretamente e sim somente depois de avisar ao sistema “Olhe, eu vou mover/girar/esticar meu objeto agora” e só depois realizar a tarefa.

Em virtude desta limitação dos sistemas de entrada tradicionais, existem estudos em como usar o próprio movimento das mãos para realizar estas tarefas, fazendo então sua execução muito próxima a que seria realizada no mundo real. Uma das formas mais naturais de se realizar isto seria a detecção dos movimentos da mão, o que pode ser feito das mais diversas formas. Em (Sturman, et al., 1994) discute-se um conjunto de protótipos de luvas com sensores que captam o movimento da mão e geram informações ao computador sobre a sua posição no espaço. Para isto, pode-se usar informação óptica (LEDs, ou a silueta da mão), magnética (usa-se um campo magnético e sensores na luva para detectar sua posição), ou sonoras (emissão de ondas sonoras). A AT&T® também fez estudos para o uso

de uma destas luvas no controle de um braço robótico. Entretanto, não foi encontrado nenhum estudo científico sobre o uso de luvas eletrônicas para o controle de braços mecânicos.

Seguindo esta proposta, de que o uso das mãos como dispositivo de entrada em um sistema que exija manipulação de objetos 3D é mais eficiente do que usar simples teclados, mouses ou joysticks, veremos como pode ser usado um novo dispositivo de entrada que visa detectar o movimento das mãos: O Wii™ Remote.

1.3. WII™ REMOTE

O *Wiimote* é um dispositivo de *input* desenvolvido pela Nintendo® para seu console de videogame Wii™ (**Figura 2**). O mais notável é que ele tenta quebrar o padrão vigente de joypad/joystick para dispositivo de entrada. O Wiimote possui forma de bastão e aparência muito similar à um controle remoto comum, para estimular a atratividade do dispositivo entre um público alvo comumente não adepto dos videogames. O usuário deve segurá-lo entre as mãos e realizar movimentos condizentes com o que se espera na simulação realizada pelo console. Desta maneira, amplia-se a sensação de realidade e liberdade no controle do jogo. Para realizar esta tarefa, o *Wiimote* possui dois dispositivos sensoriais principais: uma câmera e um acelerômetro. A câmera detecta luzes infravermelhas posicionadas em frente à televisão através da "barra sensora" do console, permitindo detectar sua posição em relação à ela. Já o acelerômetro detecta a variação de aceleração do dispositivo, o que, em teoria, permite calcular sua velocidade e deslocamento.

Figura 2: Wii™ Remote (Wikipedia, 2006).

Além do acelerômetro, com a adição do acessório Wii Motion Plus™, podemos usar um giroscópio para obter a rotação do dispositivo. Ele também é munido de botões de ação e um botão direcional.

Em virtude destas características do *Wiimote*, é proposto neste trabalho o seu uso para determinar a posição do efetuador de um braço mecânico, e a determinação das articulações por cinemática inversa. Resumidamente, a pesquisa consiste em utilizar os ângulos obtidos pela interpretação do sinal do giroscópio, para estimar a orientação do controle, e utilizar os botões **direcionais**, **+** (mais) **e** – (menos), para controlar o deslocamento do controle. Desta maneira é possível determinar seis graus de liberdade[3] para o efetuador (rotação e translação).

No Apêndice II são exibidos os resultados da pesquisa de realizar o *tracking* do dispositivo combinando o giroscópio com o acelerômetro. O objetivo é utilizar as medições de aceleração obtidas pelo sinal do acelerômetro, para estimar sua velocidade e deslocamento. Para isso é necessário descontar a força normal exercida sobre este dispositivo, então uma estimativa é obtida através da orientação do controle.

[3] *"Graus de liberdade, no contexto da mecânica, é o conjunto de deslocamentos ou rotações independentes que especificam completamente a posição e orientação deslocada ou deformada de um corpo ou sistema. Uma partícula que se move em um espaço tridimensional possui três componentes translacionais de deslocamento como graus de liberdade, enquanto um corpo rígido pode possuir até seis graus de liberdade incluindo as três rotações possíveis. Translação é a habilidade de se mover sem rotacionar, enquanto rotação é o movimento angular executado sobre um eixo."* Definição retirada de (Wikipedia, 2010).

2. TECNOLOGIAS UTILIZADAS

As tecnologias utilizadas por este projeto podem ser divididas entre hardware e software. O hardware utilizado consiste basicamente de um *Wiimote* e um Wii Motion Plus™. Seus componentes, sensores e particularidades são descritas nesse capítulo. Em termos de software, é utilizada a linguagem C# em conjunto com o framework XNA™, para a produção da simulação, e também é utilizada a biblioteca **WiimoteLib** como API de comunicação com o *Wiimote*.

2.1. HARDWARE

2.1.1. *Wiimote*

O Nintendo® Wii™ foi lançado no final de 2006 e trouxe consigo uma interface de controle inercial para jogos de vídeo que é fundamentalmente diferente dos tradicionais (joystick, teclado, mouse). O *Wiimote* surgiu como um dos grandes diferenciais de interação, e consiste do principal objeto de pesquisa deste trabalho.

Segundo (Lee, 2009), o controle, totalmente sem fio, funciona a uma frequência de 100Hz, ou seja, envia informações de seus sensores cem vezes por segundo, possui uma câmera infravermelha, que capta os sinais das luzes infravermelhas da barra emissora, um acelerômetro, responsável pela medição da força aplicada nos movimentos do jogador e botões diversos, como um direcional, outro em forma de gatilho na parte inferior, seis botões na superfície superior e um botão para ligar/desligar o console; e um *plug* de entrada para periféricos.

Ao contrário de controles de outros videogames, ele permite ser utilizado tanto por jogadores destros quanto canhotos, ao contrário de outros controles, os quais exigem que

os jogadores obrigatoriamente usem-no de uma mesma maneira sem levar em conta sua mão dominante.

Nesta seção, descreveremos o que são acelerômetros e, em particular, o acelerômetro do *Wiimote*, assim como todos os componentes deste controle.

Acelerômetro

Acelerômetros são dispositivos que captam variações de aceleração e as transformam em um sinal elétrico analógico. São baseados na Segunda Lei de Newton (uma força aplicada a uma determinada massa gera uma aceleração diretamente proporcional à esta força, $F = m.a$).

No *Wiimote,* o modelo usado é o ADXL330 (**Figura 3)**, o qual tem uma sensibilidade de aceleração de +/- 3g[4] (qualquer medição além destes valores pode ser considerada ruído), possui oito bits de informação por eixo e uma taxa de atualização de 100 Hz (Lee, 2009).

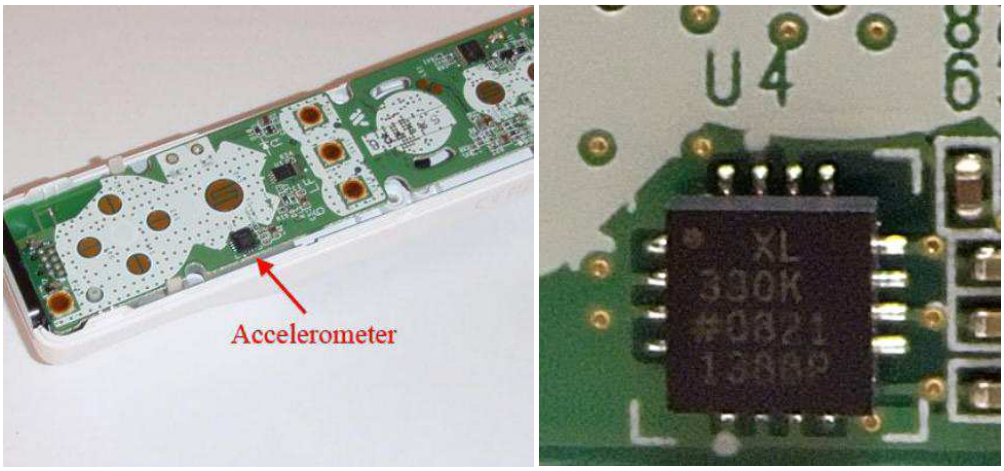


Figura 3: Acelerômetro contido dentro do *Wiimote*.

O acelerômetro funciona num sistema de coordenadas próprio, sendo que o eixo X possui o sentido invertido e o eixo Y e Z são trocados (o eixo Y equivale ao eixo Z e o eixo Z equivale ao eixo Y), como demonstrado na **Figura 4**.

[4] A medição do acelerômetro é feita em termos de força G.

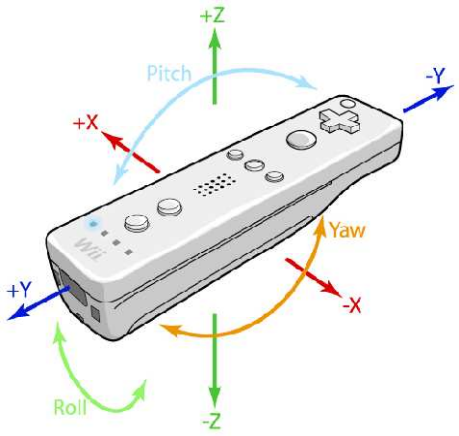


Figura 4: Eixo de coordenadas do acelerômetro do *Wiimote*.

Uma particularidade fundamental do acelerômetro para este projeto é saber com precisão o que este dispositivo mede. Segundo (Redell, 1998), ao realizar uma análise "caixa preta" do acelerômetro, como um dispositivo detector de aceleração, pode-se afirmar que ele nunca mede a aceleração gravitacional. Ao invés disso, ele pode ser considerado um dispositivo que mensura o desvio da queda livre.

A principal motivação desta afirmação é que o acelerômetro, quando em queda livre, apresenta uma medição de zero, sem importar sua orientação. Nessa situação ele está acelerando para "baixo" e ainda assim mensurando zero. Claramente, devido a isso, é válido afirmar que este dispositivo não está mensurando a aceleração gravitacional. Por outra perspectiva, ele está mensurando corretamente que o desvio da queda livre é zero.

Em outra situação, quando colocado sobre uma mesa, ele mede uma aceleração de 1.0G para cima. Como a queda livre consiste de 1.0G de aceleração para baixo, novamente podemos afirmar que o acelerômetro está mensurando corretamente o desvio da queda livre, que é uma aceleração de 1.0G de intensidade para cima devida à força normal exercida pela mesa.

Com isto em mente, se pode afirmar que como o acelerômetro nunca detecta a força gravitacional diretamente, e sim o desvio da queda livre, suas medições podem ser usadas, em determinadas situações, para inferir as propriedades da gravidade local se utilizado algum outro conhecimento que pode-se ter sobre a situação corrente.

Em outras palavras, suponha-se que o acelerômetro está contido dentro de uma caixa, sem referência externa nenhuma. Nesta situação, sem nenhum conhecimento adicional, é impossível determinar a direção ou força da gravidade. Por outro lado, em uma situação conhecida, como quando sobre a mesa, é possível inferir a força e direção da

gravidade sendo exercida sobre o acelerômetro: é exatamente igual e oposta à leitura do acelerômetro.

Com isto em mente, pode-se concluir que apesar do acelerômetro sofrer continuamente os efeitos da força gravitacional, nem sempre é possível inferir o vetor gravitacional que afeta suas medições. Somente com a adição de informações externas, como, por exemplo, a utilização de um sensor extra (por isso a grande utilidade do giroscópio do Wii Motion Plus™), é possível determinar a intensidade e direção do vetor gravitacional exercido sobre o dispositivo.

Devido a esta limitação é que o acelerômetro, por si só, não pode realizar leituras contínuas de movimento. A força gravitacional sempre irá distorcer suas leituras. Desta maneira a única utilização possível deste dispositivo por si só é o reconhecimento de padrões de movimento. Ao reconhecer padrões de leitura do sinal, treinado através de redes bayesianas ou cadeias de Markov, é possível disparar gatilhos que ativam determinadas ações.

Botões

O *Wiimote* possui doze botões, demonstrados na **Figura 5**. O botão **Power**, outros quatro formando um **direcional**, os botões **+** (mais), Home e **–** (menos), o botão **B** na parte traseira do controle (como um gatilho para ser usado com o dedo indicador) e outros três botões, **A**, **1** e **2**, para serem usados com o polegar.

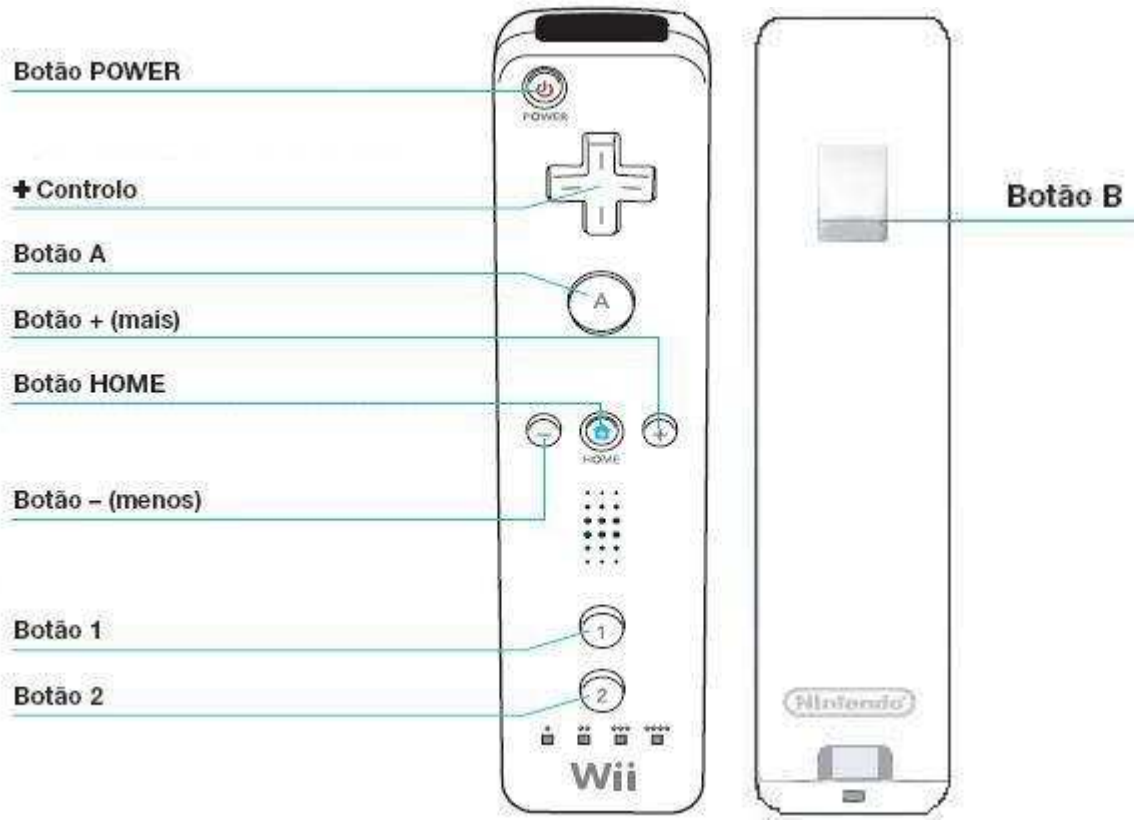


Figura 5: Descrição dos botões do *Wiimote*.

Porta de extensão

A porta de extensão do *Wiimote* permite a ligação de acessórios externos como, por exemplo, o *Nunchuck* da Nintendo® (como mostrado na **Figura 6**) ou o Classic Controller™, além de ser a porta de entrada utilizada pelo Wii Motion Plus™. Ela possui seis pinos (usados para energia e comunicação com os periféricos), dispondo de 3,3V de alimentação e 400kHz de comunicação serial I2C (Lee, 2009). A porta de extensão pode ser vista com detalhes na **Figura 7**.

Figura 6: *Wiimote* com extensão *Nunchuck* conectada. Imagem retirada de http://www.wiibellion.com/images/remote.jpg

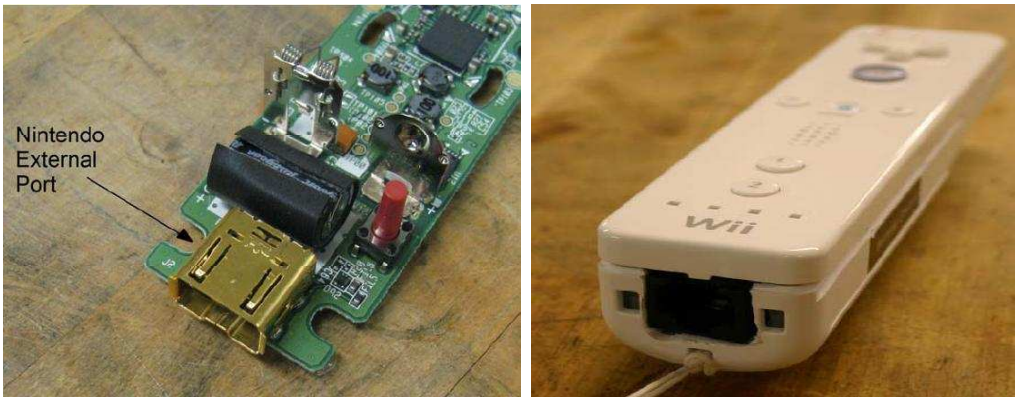


Figura 7: Detalhes internos da porta de extensão do *Wiimote*. Imagens retiradas de http://mobilityrerc.catea.org/factsheets/Wiimote+Assembly+Instructions.pdf

Câmera infravermelha

Uma câmara infravermelha, feita pela PIXART (PIXART, 1998), está localizada na parte frontal do *Wiimote*. Ela possui um chip integrado de rastreamento multiobjetos, que oferece alta resolução e alta velocidade de monitoramento de até quatro pontos infravermelhos (*IR*) simultâneos.

Através de uma barra que emite luz infravermelha a partir de duas fontes principais, demonstrada na **Figura 8**, a câmera infravermelha pode calcular a posição 3D do usuário e seus movimentos, usando triangulação com essas duas fontes.

Figura 8: Barra emissora de infravermelhos do console Wii™. Imagem retirada de http://rosseto.files.wordpress.com/2009/11/sensor_bar.png?w=500&h=71

A câmera (**Figura 9**) possui uma resolução de 128 x 96 monocromática com *built-in* de processamento de imagem. O processador usa 8x análises de subpixel para fornecer uma resolução final de 1.024 x 768 para pontos monitorados. Capaz de monitorar até quatro objetos em movimento, estes dados são os únicos dados disponíveis para o usuário. A distância entre os centros dos clusters LED é de 20 cm (Wiibrew, 2009). Possui quatro bits de tamanho do ponto ou intensidade da luz, taxa de atualização de 100 Hz e 45 graus de campo visual horizontal (Lee, 2009).

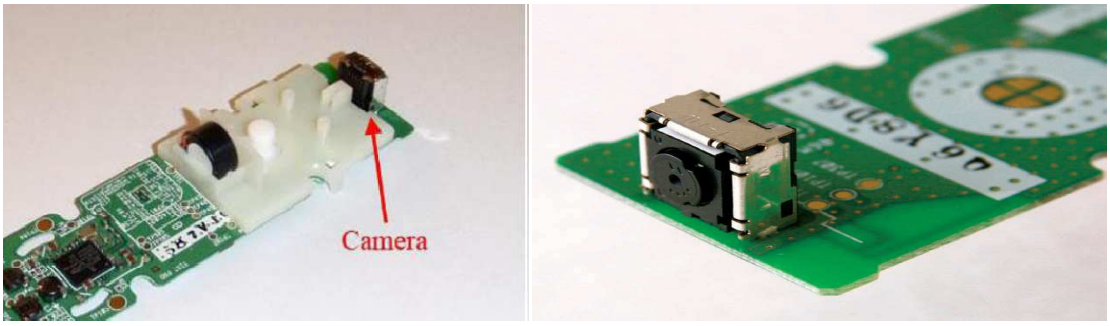


Figura 9: Detalhes internos da câmera infravermelha do *Wiimote*. Imagens retiradas de http://www.engr.colostate.edu/ece-sr-design/AY08/SKIES/Wiimote%20Lab%20revised2.pdf e (Lee, 2009).

Autofalante

O autofalante, demonstrado na **Figura 10**, é utilizado para emitir sons referentes à utilização em um determinado momento do jogo. Possui uma baixa qualidade e é usada para sons de curta duração. É similar, em qualidade, ao som de um telefone.

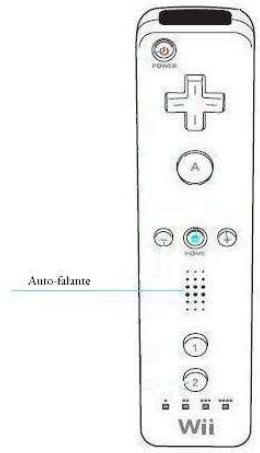


Figura 10: Indicação do autofalante do *Wiimote*.

Force Feedback (*Rumble*)

O force feedback é fornecido através de um *rumble pak*[5], um dispositivo dentro do corpo do *Wiimote*. O *Rumble*, demonstrado na **Figura 11**, é feito através de um motor com um peso desequilibrado que pode ser ativado para fazer o controlador vibrar (Wiibrew, 2009).

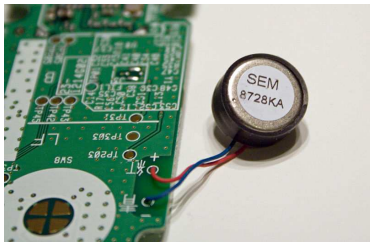


Figura 11: Dispositivo de vibração do *Wiimote*. Imagem retirada de http://wiibrew.org/wiki/File:Wii_Remote_Rumble.jpg

Bluetooth

O *Wiimote* utiliza como protocolo de comunicação o padrão HID (*Human Interface Device*) para comunicação Bluetooth[6]. Suas especificações foram desenvolvidas e licenciadas pelo consórcio Bluetooth SIG (*Special Interest Group*), formado pelas empresas Ericsson®, Intel®, IBM®, Toshiba® e Nokia®, em 1998.

O alcance de dispositivos Bluetooth os caracteriza em três classes distintas: A classe 1 (potência máxima de 100 mW, alcance de até 100 metros), a classe 2 **(**potência máxima de 2,5 mW, alcance de até 10 metros) e a classe 3 (potência máxima de 1 mW, alcance de até 1 metro). O *Wiimote* é um dispositivo Bluetooth de classe 2.

O padrão possui envio de sinais fracos (cerca de um miliwatt) e sua comunicação é feita em uma frequência entre 2,402 GHz e 2,480 GHz. Essa banda de frequência, chamada de ISM (*Industrial, Scientific, Medical*), foi reservada por acordo internacional para o uso de dispositivos industriais, científicos e médicos. No Bluetooth, pode-se utilizar até setenta e

[5]Rumble Pak é um acessório que produz vibrações lançado para Nintendo64 em 1997 pela Nintendo.

[6] Bluetooth é uma especificação industrial para áreas de redes pessoais sem fio (Wireless Personal Area Networks - PANs), que provê uma maneira de conectar e trocar informações entre dispositivos compatíveis com a tecnologia como telefones celulares, notebooks e consoles de videogames digitais, através de uma frequência de rádio de curto alcance globalmente não licenciada e segura.

nove frequências (ou vinte e três, dependendo do país) dentro da faixa ISM, cada uma espaçada da outra por 1 MHz.

Os dispositivos Bluetooth possuem baixa velocidade de transmissão. Até a versão 1.2 era de, no máximo, 1Mbps. Esse valor passou para, no máximo, 3Mbps na versão 2.0 e na versão 3.0 é capaz de atingir até 24Mbps.

2.1.2. Wii Motion Plus™

O Wii Motion Plus™, ilustrado na **Figura 12**, foi anunciado pela Nintendo® em julho de 2008 em uma conferência de imprensa na *E3 Media & Business Summit* e lançado em junho de 2009. O dispositivo consiste basicamente da integração (**Figura 13**) ao controle de um sensor adicional chamado giroscópio capaz de mensurar a rotação realizada pelo dispositivo. Ele fornece medições em uma escala própria, que quando propriamente normalizada, fornece a informação em termos de velocidade angular.

Figura 12: O Wii Motion Plus™, dispositivo de extensão para o *Wiimote*. Imagem retirada de http://www.technotalks.com/wp-content/uploads/2009/04/wii-motion-plus.jpg

De acordo com Junji Takamoto, líder da parte de desenvolvimento de hardware do Motion Plus (Aveline, 2009) o desenvolvimento do Wii Motion Plus™ começou no início de 2008. Foi Genyo Takeda, chefe do Departamento de Desenvolvimento de Produtos da Nintendo®, quem idealizou a integração do *Wiimote* à um sensor giroscópio.

O *Wiimote* original era muito limitado em que poderia detectar. Era possível a utilização como sensor de deslocamento, com três graus de liberdade (deslocamento em X, Y e Z), ou sensor de inclinação (quando parado), com dois graus de liberdade (rotação em torno de X e Z). Desta forma não era capaz de detectar todas as variações angulares e por isso não diferenciava bem movimentos retos dos curvos. Com o Wii Motion Plus™, são adicionados três giroscópios ao *Wiimote*, dispostos ortogonalmente entre si, passando, desta forma, a detectar ângulos não antes detectados, em especial *Yaw* (rotação em torno do eixo Y), como demonstrado na **Figura 14**.

Figura 13: Integração do Wii Motion Plus™ com o *Wiimote*. Imagem retirada de http://assets.gearlive.com/playfeed/blogimages/wii-motion-plus.jpg

O dispositivo utiliza um sensor do tipo IDG-600, que possui dois giroscópios integrados dispostos em dois eixos. O sensoriamento do terceiro eixo é feito através do giroscópio X3500W, produzido pela EPSON TOYOCOM® (Wikipedia, 2008) (rosseto.wordpress.com, 2009).

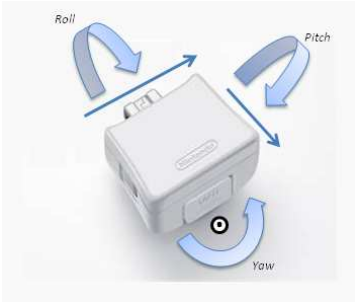


Figura 14: Eixos de rotação detectados pelo Wii Motion Plus™. Imagem retirada de http://rosseto.files.wordpress.com/2009/11/gyro1.png?w=349&h=297

Giroscópio

O giroscópio do Wii Motion Plus™ apresenta algumas peculiaridades. Deseja-se que o dispositivo seja capaz de detectar tanto movimentos precisos quanto movimentos mais rápidos. Desta forma, fazendo-se um movimento largo com o *Wiimote*, ele poderia passar do limite que o sensor consegue detectar, ficando sem ter certeza de que os dados recebidos são confiáveis. Isto obrigou a equipe de desenvolvimento a adotar a estratégia de aumentar a sensibilidade do giroscópio do dispositivo para cinco vezes mais que o nível de um giroscópio normal, permitindo a detecção de até 1600 graus de movimento por segundo (aproximadamente quatro e meia rotações completas). Dessa forma, há segurança de que os movimentos rápidos seriam detectados.

Surge daí uma contradição, pois uma vez que a largura de banda disponível para a transmissão das informações é fixa e limitada, ao melhorar a capacidade de detectar movimentos rápidos, a capacidade de detectar movimentos lentos diminui.

Então, adotou-se a estratégia de fazer o com que o sensor tivesse dois modos: um para movimentos rápidos e outro para movimentos lentos, alternados através de três *flags* de controle (uma para cada eixo). Como os dados do sensor são transmitidos sem fio, a resolução deles é pré-determinada. Então mapear as informações para dois modos através de diferentes taxas de amostragem tornou possível a alta sensibilidade de movimentos mais precisos e ao mesmo tempo permitiu o reconhecimento de movimentos mais dinâmicos.

Outra peculiaridade é que a sensibilidade da habilidade de reconhecimento do giroscópio é afetada por mudanças de temperatura. Normalmente, se algo não está em movimento, os dados enviados deveriam ser zero. Mas no caso deste sensor, mesmo quando está completamente parado, dados de "um" ou "dois" (ruído) continuariam sendo enviados por um tempo. Então mesmo que ninguém esteja tocando nele, ele se comporta como se estivesse em movimento. O termo técnico para esse comportamento é "desvio de temperatura". Não é só a temperatura que pode fazer isso: umidade ou impactos repentinos têm o mesmo efeito. Então se faz necessário tratar esse tipo de ruído.

2.2. SOFTWARE

2.2.1. XNA™

Microsoft® XNA™ (sigla em inglês que significa *XNA's Not Acronymed*) é um framework desenvolvido pela Microsoft® que serve para o desenvolvimento de jogos e simuladores para PC´s com Windows™ e para o console Xbox 360™. Baseado na linguagem de programação C# e implementado sobre o framework .NET™, o XNA™ e seu conjunto de aplicações, chamado XNA Game Studio™, é configurado como uma extensão ao Microsoft® Visual Studio™, incluindo as versões Express Edition. Ele veio para substituir o managed DirectX™ e pode ser adquirido gratuitamente.

A primeira versão lançada foi o XNA™ 1.0 em 2006. Com o passar do tempo a Microsoft® fez modificações consideráveis no XNA™ Framework. Na versão 2.0, lançada em 2008, destaca-se a conexão em rede; e na versão 3.0, lançada mais tarde no mesmo ano, destaca-se um melhor suporte a MP3. Em 2009 foi lançada a versão 3.1, cujas principais

novidades são: suporte a avatares 3D do Xbox 360™ nos jogos, suporte a vídeos e a possibilidade de controlar diretamente a execução de áudio em buffers.

O XNA possui uma rica API para criação tanto de jogos 3D quanto de jogos 2D, além de permitir a manipulação de som e entrada de dados como teclado, mouse e joystick (Xbox 360™). Ele incorpora as funcionalidades do DirectX™ e, desta forma, os desenvolvedores não precisarão se "preocupar" tanto em utilizar funções diretas e DLL's do DirectX™ para ter acesso a certas funcionalidades, como manipulação de texturas e reconhecimento de *input*.

A solução desenvolvida neste projeto está fortemente atrelada à arquitetura de funcionamento das aplicações criadas com o XNA™ Framework. Esta arquitetura é detalhada no Apêndice I, cuja leitura é fortemente recomendada para permitir o correto entendimento do algoritmo criado.

2.2.2. WiimoteLib

Desenvolvida em C# usando o framework .NET™ (em código gerenciado[7]), a biblioteca **WiimoteLib** encapsula classes e objetos necessários para a conexão e comunicação entre o código e o *Wiimote*, e é a API de base para este projeto. Criada por Brian Peek e primeiramente lançada em Março de 2007 (Peek, 2007), ela explora virtualmente todas as funcionalidades do controle e de suas extensões, com pequenas exceções. Na versão 1.8 beta 1, a utilizada neste projeto e última disponível até o momento, possui como principais problemas o suporte apenas experimental para a utilização do alto-falante embutido no *Wiimote* e para a utilização o Wii Motion Plus™. Entretanto, o problema que este último apresenta aparenta ser somente com relação ao processo de normalização do sinal emitido pela extensão, de sua escala própria para velocidade angular, que ainda é um tanto incerta para o desenvolvedor em alguns aspectos. Entretanto a obtenção do sinal é perfeitamente funcional, então é feita a interpretação deste baseando-se no processo de normalização sugerido por (WiiBrew, 2010) e (Peek, 2009 p. Comentários). Este processo de

[7] *"Todo código que é gerado na plataforma .NET é chamado de **código gerenciado**. Gerenciado porque ele é controlado pelo CLR (Common Language Runtime) que fornece os seguintes serviços: integração entre linguagens, segurança, suporte a versões e coleta de objetos que não estão em uso (Garbage Collection)."*
Definição retirada de (Guimarães, 2004).

normalização e o aproveitamento das informações do Wii Motion Plus™ são descritos no Capítulo 3, Seção 3.2.2.

A principal classe da biblioteca é a **Wiimote**. É através de sua instância que se estabelece a comunicação com o controle e suas extensões, manipula-se seu estado e obtêm-se os sinais enviados pelos seus sensores.

Conectando a aplicação com o *Wiimote*

O processo de conexão do *Wiimote* resume-se às seguintes etapas: Primeiramente, antes de executar a rotina a seguir, certificar-se que o controle está conectado de maneira adequada ao computador; em seguida, criar uma instância de **Wiimote** e chamar o método **Connect( )**; após o retorno positivo do método, utilizar **SetReportType( )** com o tipo de relatório desejado, de acordo com as funcionalidades do controle que deseja-se usar; em seguida, no caso deste projeto, inicializar o Wii Motion Plus™ através do método **InitializeMotionPlus( )**; e finalmente, obtêm-se as informações fornecidas pelo controle através da propriedade **WiimoteState**, e pode-se manipular o seu estado atual através de seus métodos internos públicos (tais como **SetLEDs( )**, **SetRumble( )** e etc.). O código fonte contendo as principais declarações desta classe é demonstrado no Apêndice III, seção 1.

Integração com o XNA™

A integração da biblioteca com o XNA™ se deu de maneira descomplicada, não somente por ser implementada na mesma linguagem (C#) e framework (.NET™), mas também pelo seu funcionamento interno. Como o *Wiimote* envia continuamente um sinal contendo informações de seus sensores internos, sendo por diante apenas por “sinal”, a uma frequência aproximada de 100 Hz (Lee, 2009), o funcionamento da biblioteca originalmente requeria a implementação de um **EventHandler** que seria invocado toda vez que houvesse novas informações disponíveis. De maneira análoga, o método **Update( )** do XNA™ se encaixou perfeitamente na posição deste **EventHandler**, pois este é chamado continuamente durante a execução do *Game Loop*. O fato desses métodos não operarem na mesma sincronia não afeta o funcionamento da biblioteca quando utilizada no XNA™, ou seja, o fato do método **Update( )** ser chamado antes ou depois que informações atualizadas do controle estejam disponíveis (frequência esta dependente da taxa de atualização configurada e do desempenho da máquina executando a aplicação), enquanto o

EventHandler seria invocado justamente na ocorrência de tal evento, é indiferente para a simulação implementada em XNA™ devido sua forte dependência da variação do tempo medida entre a execução dos loops. Por exemplo, se em um determinado instante o controle não registrar variação na medição do sinal, significa que a variação temporal entre a chamada do **Update( )** também foi muito pequena, o que não irá interferir no resultado apresentado para o usuário. Levando isto em consideração, uma vez o controle conectado foi necessário apenas resgatar suas informações dentro do método **Update( )**, dispensando assim a implementação do **EventHandler**.

3. IMPLEMENTAÇÃO DO CONTROLE DO BRAÇO MECÂNICO

A proposta de solução para o problema descrito neste trabalho é detalhada neste capítulo. Apresenta-se em detalhes a solução e o software criados como objetivo final deste projeto, contendo a biblioteca e a aplicação criada, enumerando cada uma de suas respectivas classes e seus algoritmos.

3.1. VISÃO GERAL

O processo de desenvolvimento consistiu basicamente de quatro etapas principais:

1) Primeiramente, ser capaz de se comunicar com o *Wiimote* e obter o sinal que ele envia continuamente;
2) Em seguida, interpretar este sinal e produzir uma saída uniforme, consistindo de rotação e translação, para que possa ser facilmente empregada em soluções de animação procedural tridimensional;
3) Encapsular este processo em uma biblioteca, de forma que fosse facilmente integrada a outros projetos;
4) Por fim, desenvolver a simulação do braço mecânico em uma solução multicamadas, utilizando a biblioteca criada.

Uma vez decidida a plataforma de desenvolvimento utilizada no projeto, C# e XNA™, deu-se início à primeira etapa do desenvolvimento: a comunicação com o *Wiimote*. A maior parte dos problemas dessa etapa foi resolvida com a adoção da biblioteca **WiimoteLib**, desenvolvida por Brian Peek (Peek, 2009), como descrito no Capítulo 2, Seção 2.2.2.

Com a API de comunicação com o *Wiimote* definida, foi criada a classe **InputHandler**, baseada na implementação sugerida por (Carter, 2008), que encapsula o acesso às principais entradas de dados utilizadas, o *Wiimote*, o teclado e o mouse, como um componente do XNA™.

Também foi criado um componente simples de câmera e sua derivação para câmera de primeira pessoa, denominados respectivamente **Camera** e **FirstPersonCamera**, para visualização e navegação do ambiente 3D gerado, e um componente de medição de frames por segundo, denominado **FPS**, para avaliação de desempenho da aplicação, ambos também baseados na implementação sugerida por (Carter, 2008).

Resolvida a primeira etapa, iniciou-se a interpretação do sinal do *Wiimote*. Para isso, foi necessária a criação de um projeto de testes no XNA™, o mais simples possível, que utilizava os componentes descritos acima e continha apenas uma espada, que deveria ser animada em correspondência com os movimentos interpretados do *Wiimote*, como mostra a **Figura 15** a seguir. Dentro desta aplicação foram desenvolvidos todos os passos para a interpretação dos sinais, desde a calibração até a geração dos valores de saída.

Figura 15: Aplicação de teste utilizada durante o processo de desenvolvimento da etapa de interpretação do sinal do *Wiimote*. Nas laterais estão as informações de debug geradas.

Na etapa seguinte, o código desenvolvido na aplicação de teste para a interpretação do sinal do *Wiimote* foi encapsulado em uma classe com interface de acesso uniforme, chamada **WiimoteInput**. Esta classe também utiliza a arquitetura de componente do XNA™, sendo possível integrá-la facilmente a qualquer projeto baseado nesse framework. Então, esta e as outras classes criadas anteriormente (**Camera**, **FirstPersonCamera**, **InputHandler** e **FPS**) foram agrupadas em uma biblioteca chamada **WiimoteInputLib**.

Para realizar corretamente o deslocamento da mão (efetuador) do braço mecânico na simulação, se fez necessária a utilização de uma técnica chamada cinemática inversa.

Para isso, foi criada a classe **InverseKinematics** que centraliza esta operação através de métodos estáticos. Esta classe foi então adicionada à biblioteca **WiimoteInputLib**.

Por fim, na última etapa foi criada uma solução denominada **BracoMecanicoWiimote**, que agrupa o código desenvolvido, composto por um projeto de aplicação do XNA™ chamado **Simulacao** e pela biblioteca **WiimoteInputLib**[8]. O projeto **Simulacao** consiste basicamente da simulação do braço mecânico, e é composto por três classes: **Program**, que é o ponto de entrada para o software e dispara o *Game Loop*[9] do XNA™; **BracoMecanicoWiimote**, que é a classe principal da simulação, o *Game Loop* em si (classe derivada de **Game**); e **BracoMecanico**, que é um componente que encapsula a operação de desenhar e manipular o braço mecânico (e utiliza para isso intensivamente a biblioteca desenvolvida).

Pode-se ter uma visão geral do que foi desenvolvido e descrito acima através do diagrama de classes da solução, ilustrado pelo **Diagrama 1** a seguir, que descreve toda a solução **BracoMecanicoWiimote**. Os componentes individuais da solução supracitados e o processo desenvolvido para a interpretação do sinal do *Wiimote* são detalhados nas seções a seguir.

[8] A biblioteca **WiimoteLib** também faz parte da solução, uma vez que é a API de comunicação com o *Wiimote*. Entretanto, vale ressaltar que essa biblioteca não foi criada por este autor, apenas utilizada.

[9] Termo definido no Apêndice I.

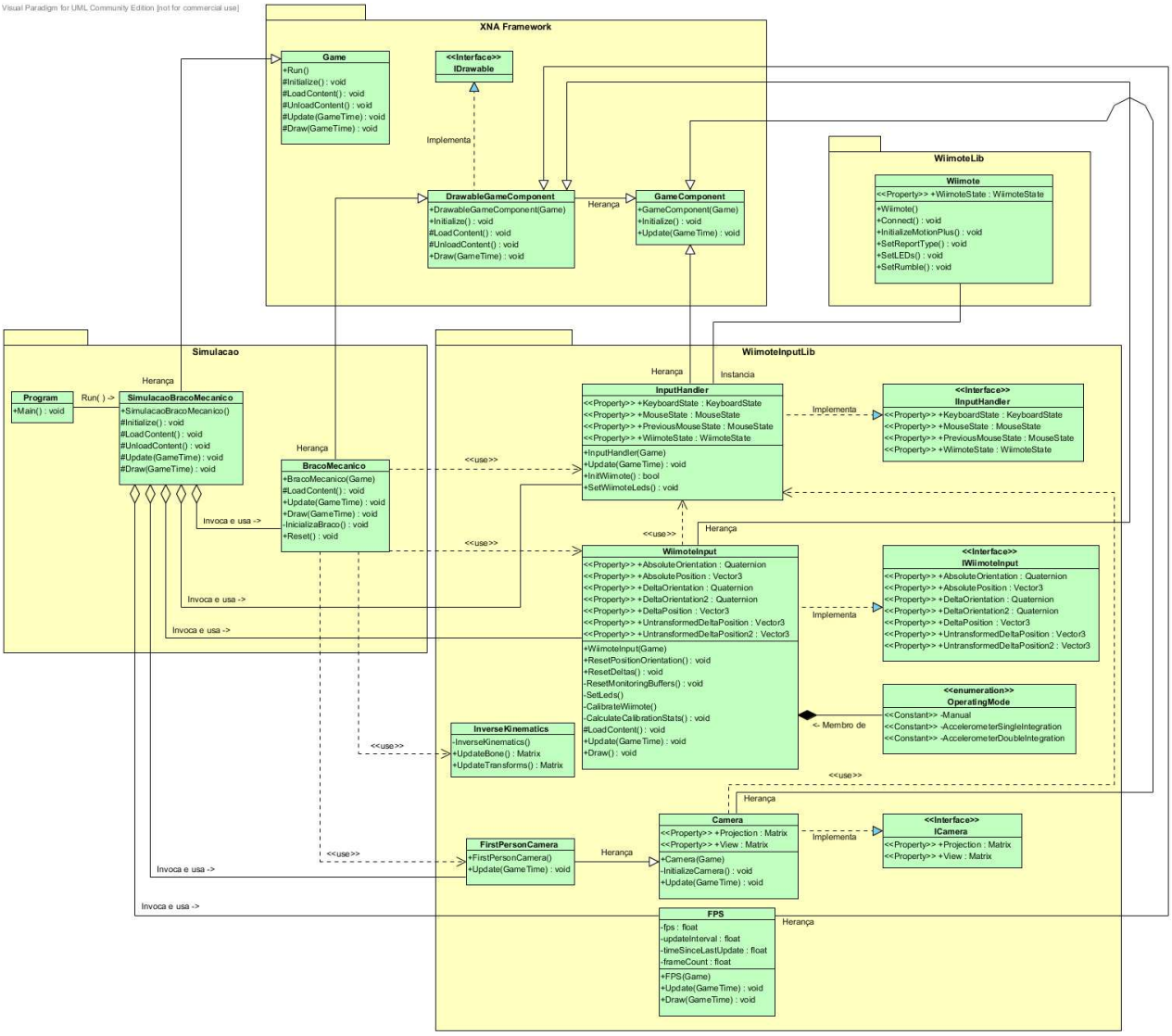


Diagrama 1: Diagrama de classes da solução **BracoMecanicoWiimote**. Os pacotes **Simulacao** e **WiimoteInputLib** foram detalhados o máximo possível, entretanto os atributos foram omitidos por questão de espaço. Os pacotes **WiimoteLib** e **XNA™ Framework**, APIs utilizadas como base, apresentam apenas as informações mais relevantes ao projeto.

3.2. BIBLIOTECA **WIIMOTEINPUTLIB**

A **WiimoteInputLib** é a biblioteca criada para agrupar as principais classes desenvolvidas para este projeto. É composta pelas classes **WiimoteInput**, **InputHandler**, **Camera**, **FirstPersonCamera**, **FPS** e **InverseKinematics**, como citado anteriormente na Seção 3.1, e também pelas interfaces **IWiimoteInput**, **IInputHandler** e **ICamera**.

A **WiimoteInput**, principal classe da biblioteca, permite a utilização do *Wiimote* como dispositivo de entrada ao fornecer informações de rotação interpretadas em tempo real e translação do controle. Ela faz uso intensivo da classe **InputHandler** para se comunicar com o *Wiimote* e obter o sinal gerado pelos seus sensores. O *enumeration* **OperatingMode**,

declarado dentro de **WiimoteInput**, é usado internamente para indicar qual é o modo de operação atual da classe, como detalhado no Apêndice II, Seção 4.

A classe **InputHandler**, outro componente muito importante da biblioteca, agrupa as operações de comunicação e aquisição de informações dos dispositivos de entrada necessários: *Wiimote*, teclado e mouse. Para obter as informações do *Wiimote* em específico, esta cria uma instância da classe **Wiimote** (da biblioteca **WiimoteLib**) e implementa o procedimento de conexão descrito no Capítulo 2, Seção 2.2.2.

A classe **Camera** e sua derivada **FirstPersonCamera**, responsáveis pela visualização e navegação no mundo 3D gerado pelo XNA™, também fazem uso intensivo da classe **InputHandler** para movimentação da câmera no ambiente. Elas fornecem uma saída uniforme composta por duas matrizes: **Projection** e **View**, indispensáveis nos procedimentos de *rendering*.

A classe **FPS** funciona de maneira independente e seu objetivo é unicamente informar em tempo real a taxa de frames por segundo da aplicação que está sendo executada, para efeitos de medição de desempenho e depuração.

A classe **InverseKinematics** encapsula os métodos necessários para o cálculo da cinemática inversa, técnica necessária para a correta movimentação do braço mecânico na simulação.

As interfaces **IWiimoteInput**, **IInputHandler** e **ICamera**, implementadas pelas classes **WiimoteInput**, **InputHandler** e **Camera**, respectivamente, especificam propriedades para uniformizar a comunicação entre as classes de componentes cujo funcionamento depende das supracitadas. Esta é uma particularidade da arquitetura do XNA™, que especifica que a comunicação entre componentes deve ser feita através de serviços[10].

Desenvolvida utilizando a arquitetura de componentes do framework XNA™ e suas estruturas de dados, a biblioteca foi especificada de maneira que pudesse ser integrada facilmente a qualquer projeto. Apesar de sua dependência de plataforma, é possível sua portabilidade para outros ambientes e tipos de aplicação com a realização de alguns ajustes na estrutura de funcionamento das classes, em específico no que se refere à chamada contínua dos métodos **Update( )** e **Draw( )**.

[10] Definição de serviços detalhada e exemplificada no Apêndice I.

O **Diagrama 2** a seguir descreve o funcionamento interno da biblioteca e demonstra suas interdependências.

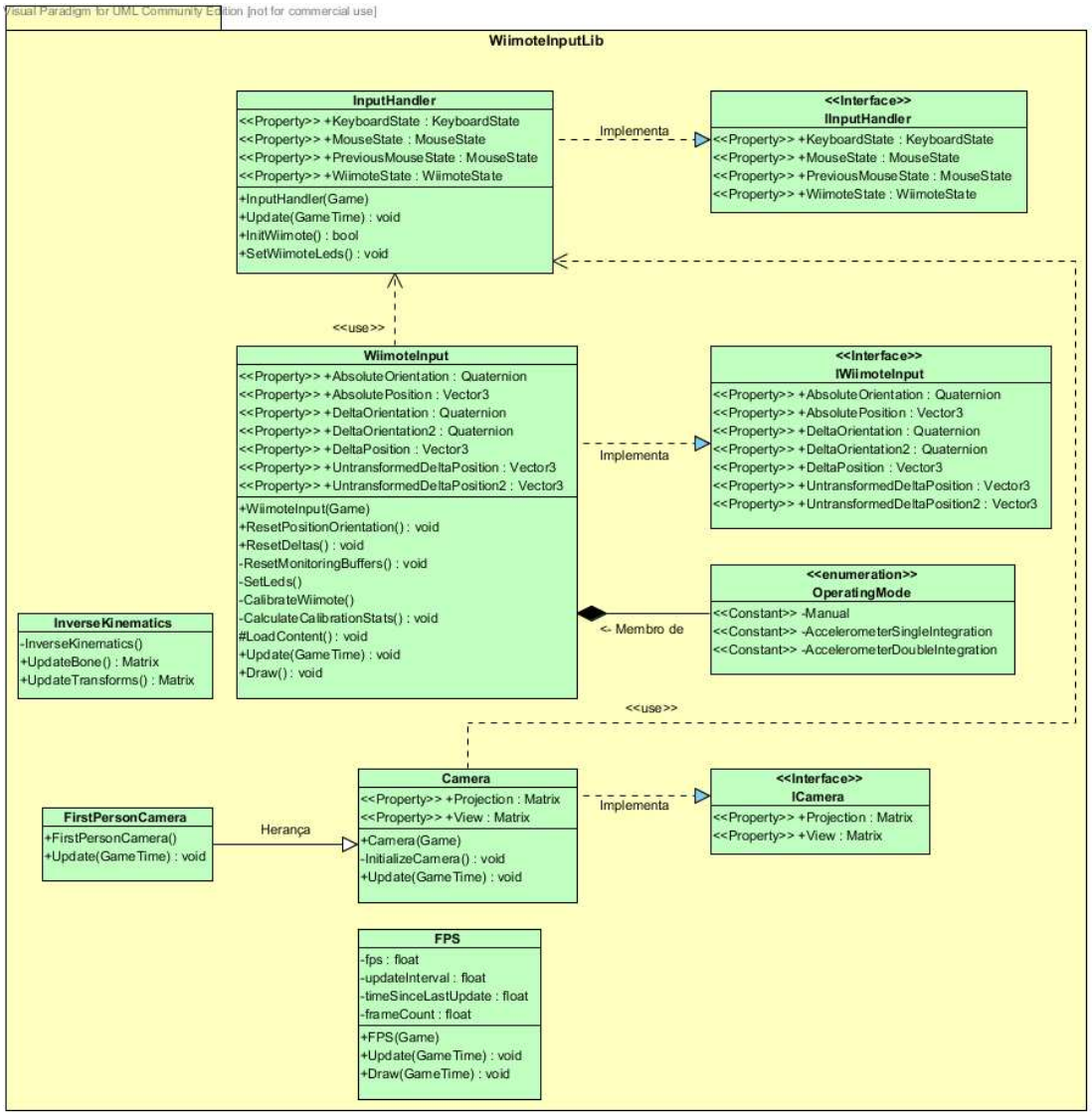


Diagrama 2: Diagrama de classes da biblioteca **WiimoteInputLib**. Este diagrama apresenta de forma mais clara o relacionamento entre seus componentes internos, uma vez que suas relações com componentes externos foram ocultadas.

3.2.1. Classe InputHandler

Pode-se afirmar que a classe **InputHandler** foi o ponto de partida para a execução do projeto. Como descrito na Seção 3.1, a primeira das quatro grandes etapas do desenvolvimento foi resolver a comunicação com o *Wiimote*. Era necessária uma interface única e simples para a aquisição do sinal dos sensores do controle. Com isso em mente, esta foi a primeira classe a ser desenvolvida.

Entretanto, verificou-se que, além de centralizar o acesso às informações do *Wiimote*, esta classe teria o potencial de também agrupar e uniformizar o acesso à outros dispositivos de entrada utilizados no projeto: o teclado e o mouse. Então, baseando-se na implementação sugerida por (Carter, 2008)[11], foi criado o componente **InputHandler** e sua interface de acesso, **IInputHandler**.

É importante ressaltar que sua função é crítica para a operação do sistema, pois esta é responsável por agrupar as operações de comunicação e aquisição de informações de todos os dispositivos de entrada necessários para a operação do projeto.

Seu funcionamento obedece à lógica de sua classe pai, **GameComponent** (descrita no Apêndice I), portanto, após a inicialização de seus atributos no método **Initialize( )**, praticamente toda sua operação ocorre no método **Update( )**. Dentro deste método, é feita a aquisição das informações de estado de Mouse, Teclado e *Wiimote*, e armazenadas em atributos que podem ser acessados por sua interface. No caso específico do Mouse, é armazenado também seu estado anterior para possibilitar, através da obtenção da diferença dos estados, a verificação de direção, sentido e velocidade que o cursor está se movendo. Note que para obter as informações de estado do *Wiimote*, é criada internamente uma instância da classe **Wiimote** que se conecta automaticamente ao controle durante a inicialização do componente. Para inicializar o *Wiimote*, foi criado o método **InitWiimote( )**, que se conecta ao dispositivo, configura seu modo de operação e inicializa o Wii Motion Plus™. Por fim, também foi criado um método chamado **SetWiimoteLEDs( )** servindo de intermediário para a chamada de método similar da classe **Wiimote**, com o intuito de permitir a mudança do estado dos LEDs do controle durante a execução da aplicação.

O código-fonte desta classe pode ser visto no Apêndice III, Seções 2.1 e 2.2.

3.2.2. Classe WiimoteInput

A classe **WiimoteInput** foi criada como resultado da solução da segunda e terceira grandes etapas de desenvolvimento, como descrito na seção 3.1, e concentra o produto de 85% do esforço desenvolvido no projeto. Esta classe é responsável pelo processo de interpretação do *input* fornecido pelo *Wiimote* e Wii Motion Plus™, e fornece informações

[11] Sugestão de implementação obtida do Capítulo 5 deste livro.

de rotação, em tempo real, e translação (variação instantânea e absoluta, de acordo com a posição calibrada) na forma de, respectivamente, *quaternions* e vetores tridimensionais.

Esta pode ser considerada a classe mais importante deste projeto, pois constitui a base para a interação com o usuário desejada no produto final, ou seja, a movimentação da simulação do braço mecânico utilizando o *Wiimote* como controlador.

As etapas necessárias para este processo de interpretação foram definidas especialmente após um entendimento mais preciso do funcionamento do acelerômetro e de suas limitações, detalhado no Capítulo 2, Seção 2.1.1.

Como citado na seção 3.1, o algoritmo para interpretar os sinais do *Wiimote* foi todo desenvolvido em uma aplicação de teste simplificada, que permitia a visualização imediata de seus resultados. Devido às particularidades do domínio do problema abordado, e em parte pelo fator de novidade do projeto (o que de certa forma limita a literatura disponível no assunto), boa parte do processo de desenvolvimento e ajuste fino dos resultados obtidos foi baseado em experimentação, tentativa e erro, seguido de observação. Por isso a aplicação de testes foi tão necessária. Uma vez que os resultados obtidos foram considerados satisfatórios, passou-se à etapa de encapsulamento da solução na classe **WiimoteInput**.

Como a estrutura de funcionamento de uma aplicação no XNA™ (derivada da classe **Game**) é muito similar à de um **GameComponent**[12], o processo de encapsulamento do algoritmo gerado na aplicação de testes foi rápido, pois sua lógica se manteve intocada. Implementada como um **DrawableGameComponent** do XNA™ (*Drawable* unicamente para permitir a exibição de informações de debug), ela faz uso intensivo da classe **InputHandler** (acessada como serviço através de sua interface, **IInputHandler**) para a obtenção do sinal emitido pelos sensores do controle, matéria prima para sua execução. O último detalhe foi definir a interface **IWiimoteInput** de comunicação do componente, que especifica o acesso às informações de rotação através de *quaternions* e às informações de deslocamento através de vetores tridimensionais, então codificada em conjunto com **WiimoteInput**.

Nas seções a seguir é detalhado o algoritmo de interpretação descrito acima.

[12] Como visto no Apêndice I.

Visão geral do processo de interpretação de *input* do *Wiimote*

O processo de interpretação de *input* do *Wiimote* e do Wii Motion Plus™ é um algoritmo que pode ser descrito em quatro etapas distintas e sequenciais:

1) Calibração;
2) Interpretação do giroscópio para estimar a orientação do controle;
3) Interpretação dos botões para obter o deslocamento;
4) Comunicação da classe.

A primeira das etapas do algoritmo, a calibração, normalmente é realizada apenas uma vez e no início da execução. Durante esta etapa obtêm-se amostras dos sinais do *Wiimote* e do Wii Motion Plus™ em condições ideais de funcionamento (parado, sobre a mesa com os botões virados para cima), que são então utilizadas na geração de valores de referência que serão aplicados durante o processo de interpretação. As amostras são obtidas em um intervalo de tempo pré-determinado, e então estatísticas relevantes, descritas no detalhamento deste procedimento abaixo, sobre essas amostras são calculadas para a geração desses valores de referência.

Em contrapartida com a primeira, as três etapas seguintes ocorrem em tempo real, continuamente[13], durante todo o período de duração da simulação, à medida que é feita a leitura dos sinais dos sensores do *Wiimote*.

A segunda etapa, estimativa da orientação do controle, consiste especificamente da interpretação do sinal do giroscópio do Wii Motion Plus™. Uma vez obtido, o sinal é normalizado, ou seja, sua escala é convertida de unidades próprias para velocidade angular, então filtrado, descartando-se valores considerados como ruído, e, finalmente, integrado, para obter a variação angular medida pelo sensor. Esta variação é então adicionada à orientação inicial, obtida durante a etapa de calibração, obtendo-se assim a orientação absoluta do controle.

A terceira etapa consiste de obter o deslocamento através do pressionar de botões direcionais, presentes no *Wiimote*. Desta maneira o deslocamento ocorre com uma taxa de variação (velocidade) constante, pré-configurada em três níveis de intensidade.

A última etapa, a comunicação da biblioteca, consiste em uniformizar a saída fornecida através de estruturas de dados simples, porém que descrevem com precisão a

[13] Para isto, o algoritmo é implementado dentro do método **Update( )** do componente.

movimentação estimada. Para isso, são usados *quaternions* para armazenar a orientação absoluta e sua variação instantânea, e vetores tridimensionais para armazenar a posição absoluta e suas variações instantâneas transformadas e não transformadas de acordo com a orientação absoluta. Com esta interface de saída, é possível modificar com flexibilidade o algoritmo interno, sendo necessário apenas manter esta forma final de armazenamento da solução.

Calibração

A calibração do controle consiste de um algoritmo relativamente simples, entretanto de fundamental importância para o processo de interpretação. Ela é disparada logo no início da execução da aplicação, e possui duração de dois segundos (configurável através da constante **CALIBRATION_TIME**). Durante esses dois segundos, o controle deve permanecer sobre uma superfície estática e nivelada, em posição padrão (com seus botões virados para cima), e sem sofrer perturbações. Caso essas condições não sejam atendidas, serão gerados valores de referência impróprios, o que afetaria fatalmente o processo de interpretação de movimentos. Nessa situação o procedimento seria refazer o processo, posicionando novamente o controle como descrito acima e apertando a tecla **Espaço** do teclado para disparar a nova calibração.

Existem duas variáveis booleanas que são utilizadas como *flags* para o controle da etapa de calibração, **m_bCalibrating** e **m_bCalculateCalibrationStats**. O motivo disso é a arquitetura da aplicação. Em uma aplicação de tempo real, todo o processamento de informações é feito continuamente através de um mesmo método, **Update( )**, que é chamado continuamente durante a execução do programa. A única maneira de preservar, para chamadas futuras, comandos fornecidos em um determinado instante é armazenando-os em variáveis de controle com um escopo externo, nesse caso atributos da classe. Desta maneira, as *flags* indicam qual o estado atual de operação da classe. Nesse caso, por exemplo, quando pressionada a tecla **Espaço** para invocar o procedimento de recalibração, a variável **m_bCalibrating** vai indicar para as chamadas de **Update( )** que as amostras do sinal devem ser colhidas e o processo de interpretação interrompido momentaneamente. Após os dois segundos de execução seu estado é invertido e ela volta a ativar o processo de interpretação, mas não antes da variável **m_bCalculateCalibrationStats** indicar a

necessidade de se recalcular as estatísticas da calibração, para geração dos valores de referência.

Para inicializar o processo de calibração, foi criado um método chamado **CalibrateWiimote( )**. Este método é responsável pela reinicialização das variáveis de estatísticas e de controle necessárias para a calibração do *Wiimote* e do Wii Motion Plus™. Além disso, ele zera os atributos de controle e de resultados da interpretação do sinal[14], através dos métodos **ResetPositionOrientation( )**, **ResetDeltas( )** e **ResetMonitoringBuffers( )**[15]. Este método é chamado na inicialização do código e toda vez que a tecla **Espaço** é pressionada.

Uma vez executado **CalibrateWiimote( )**, estabelece-se o cenário requerido para a execução da calibração durante o método **Update( )**. Durante os dois segundos da execução, controlados pela variável **m_fCalibrationEnd**, amostras dos sinais do acelerômetro[16] e do giroscópio são armazenadas nos vetores **m_cv3CalibrationMotionPlus** e **m_cv3CalibrationAccelerometer**, respectivamente, e seus índices **m_iCalibrationIndexMotionPlus** e **m_iCalibrationIndexAccelerometer** atualizados de acordo. Quando o tempo pré-determinado acaba, a *flag* **m_bCalibrating** é invertida e a amostragem termina. Porém, antes de iniciar o processo de interpretação do sinal, a *flag* **m_bCalculateCalibrationStats** garante a chamada do método **CalculateCalibrationStats( )**, para que os valores de referência sejam gerados.

A etapa de calibração é concluída após a execução deste método, no qual são gerados os valores de referência tão importantes para a correta execução do restante do processo de interpretação. O algoritmo aqui é dividido em duas partes: cálculo das estatísticas do sinal do giroscópio e cálculo das estatísticas do sinal do acelerômetro.

O cálculo das estatísticas do giroscópio consiste em, para cada eixo, computar a média dos sinais e os picos obtidos. Essa informação em si não é tão relevante para o processo de interpretação atual, apenas para a eliminação do ruído, e foi mais utilizada no

---

[14] Assume-se que, como o processo de calibração está sendo feito ou refeito, a calibração anterior não era válida e as informações fornecidas anteriormente não eram precisas (se existentes). Por isso elas são reiniciadas.

[15] Estes três métodos serão descritos mais tarde.

[16] As amostras do sinal do acelerômetro e os valores de referência gerados para este dispositivo são utilizados no Apêndice II, Seção 2.

início do desenvolvimento do algoritmo de estimativa da orientação como informação de debug, que permitiu o maior entendimento das características do sinal. O que é sim, de extrema importância, é saber a orientação inicial do controle (em descanso sobre a mesa). Este valor é utilizado para obter a orientação absoluta do controle, já que o processo de integração da velocidade angular medida pelo giroscópio, fornece apenas a variação instantânea, que deve ser somada a este valor de referência.

Já o cálculo das estatísticas do acelerômetro consiste em computar a média do módulo da força normal sendo exercida sobre o controle, quando este está parado, e seu desvio padrão. Como durante a calibração sabemos que o controle está parado e sem sofrer perturbações, a força normal é a única força sendo exercida sobre o controle, portanto esta média calculada, armazenada em **m_fAccelerometerMean**, é um precioso valor de comparação. Como é detalhado no Apêndice II, com certa margem de erro, é possível indicar com este valor momentos que o controle está em posição estacionária, gatilho que dispara a técnica de autocorreção do vetor de força normal.

Obtenção da orientação do controle usando o Wii Motion Plus™

A estimativa da orientação é feita por um algoritmo que interpreta única e exclusivamente o sinal fornecido pelo giroscópio do Wii Motion Plus™. Ele funciona em três passos: primeiramente faz-se a normalização do sinal, e em seguida faz-se a integração numérica do sinal normalizado para obter a variação angular da orientação para aquele instante, e por fim aplica-se um filtro. O resultado obtido pode então ser usado para obter a orientação absoluta do controle.

O primeiro passo, a normalização do sinal, consiste em transformar as medidas obtidas pelo sensor de uma escala própria para uma escala padrão de velocidade angular. Esta escala própria é baseada em particularidades do hardware da Nintendo®, e por isso é misteriosa até certo ponto. Tal particularidade influenciou inclusive no desenvolvimento da biblioteca **WiimoteLib**, como citado no Capítulo 2, Seção 2.2.2, de tal forma que o criador da biblioteca não incluiu nativamente a normalização do sinal do giroscópio do Wii Motion Plus™ por conta de informações inconclusivas, como pode ser visto em (Peek, 2009). A solução encontrada para este problema foi baseada em informações obtidas em (WiiBrew, 2010), onde uma equipe de desenvolvedores e hackers fez a engenharia reversa do dispositivo e sugeriu um método para conversão dessas medidas para velocidade angular.

De acordo com esta fonte, quando estático, o giroscópio emite um sinal que varia em torno de aproximadamente 8000 unidades (em uma escala própria) para cada eixo, variando de dispositivo para dispositivo, sendo necessária a calibração por alguns segundos para obter o valor correto (feito na etapa anterior). Quanto à variação, afirma-se que o controle funciona em dois modos diferentes de operação, como confirmado também por (Aveline, 2009), um para detectar movimentos mais rápidos e outro para movimentos mais precisos. Quando operando em modo preciso, a fonte afirma que uma variação em torno de 20 unidades corresponde a uma taxa de velocidade angular de 1 grau por segundo, enquanto em modo rápido essa mesma variação corresponde a uma taxa de 5 graus por segundo.

A técnica desenvolvida nesse projeto para fazer esta conversão baseou-se nas informações acima e na observação dos resultados obtidos nos cálculos estatísticos feitos sobre a amostra do sinal do giroscópio do processo de calibração. Nestas observações foi possível verificar que o valor médio que o controle assumia quando em posição estática variava em torno de 7890 unidades. Ainda em posição estática, observou-se que o sinal variava ininterruptamente aproximadamente 100 unidades para mais e para menos, com picos de 130 e vales de 70 dependendo do eixo. A estratégia decidida então, para aumentar a visibilidade das variações, foi imediatamente dividir o sinal fornecido por 100 (valor definido pela constante **MOTIONPLUS_DIVISION_RATE**). Sendo assim, cada unidade que o sinal varia agora corresponde a 5 graus por segundo em modo preciso e 25 graus por segundo em modo rápido[17]. No código foram definidas as variáveis **fPitchMultiplier , fYawMultiplier** e **fRollMultiplier**, as quais fazem correspondência com a taxa de velocidade angular do modo sendo executado, que são multiplicadas pela variação do sinal em cada eixo para aquele determinado instante. Desta maneira elas são inicializadas com o valor 5, e caso as *flags* de modo rápido estejam ativadas, **PitchFast**, **YawFast** e **RollFast** (propriedades de **Wiimote.WiimoteState.MotionPlusState**), respectivamente e independentemente para cada variável, seus valores são modificados para 25.

A afirmação de (WiiBrew, 2010) que o sinal fornecido pelo giroscópio do Wii Motion Plus™ caracteriza uma velocidade angular (após a normalização do sinal), pode ser confirmada com a análise gráfica de uma amostra deste sinal. O gráfico a seguir, baseado em

[17] Como o sinal foi divido por 100 e a velocidade angular era medida em termos de 20, cada unidade variada de **sinal/100** agora corresponde a 5 vezes à taxa de velocidade angular descrita anteriormente.

uma amostra de um movimento realizado pela mão de um usuário caracterizando um giro de 180° em sentido anti-horário ao redor do eixo Z (*roll*), possui curva que descreve claramente o comportamento de um gráfico de velocidade. A deformidade do gráfico na região que seria seu pico se dá devido à mudança de modo de operação do dispositivo, passando de modo preciso para modo rápido. Note que no modo rápido cada unidade de variação significa mais variação angular. Nesse caso corresponde à cinco vezes mais velocidade do que no modo anterior, e quando aplicados os multiplicadores (descritos no parágrafo anterior) este sinal será corrigido.

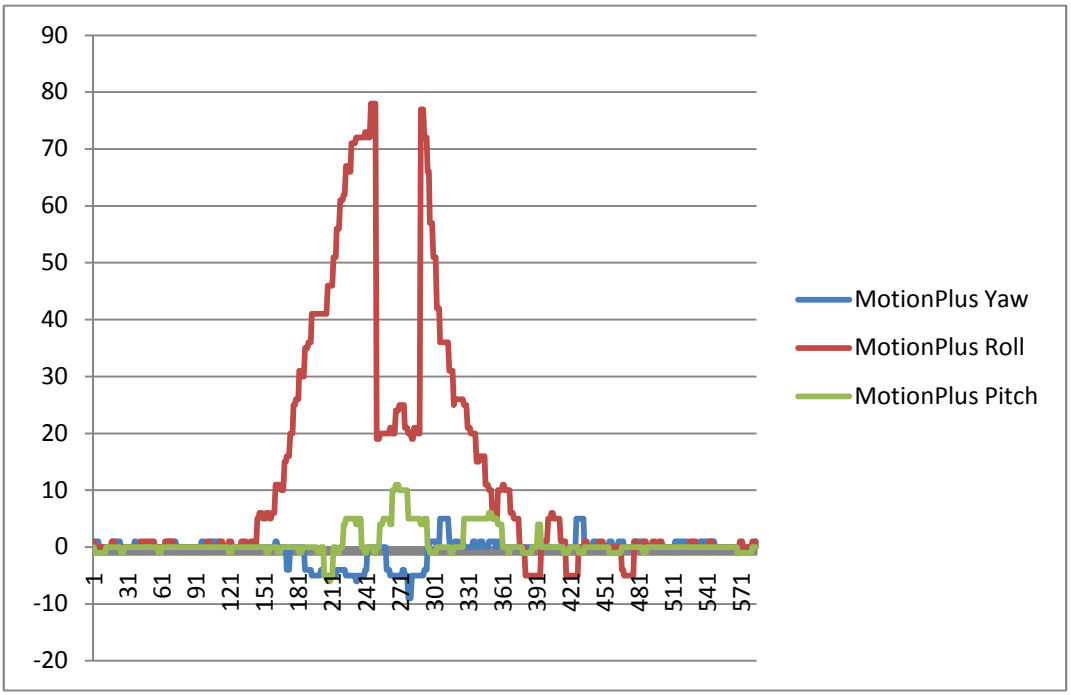


Gráfico 1: Amostra de sinal do giroscópio do Wii Motion Plus™. O movimento realizado durante a amostragem foi uma rotação de 180° em sentido anti-horário ao redor do eixo Z (*roll*).

Dando sequência ao segundo passo do algoritmo, constata-se que para obter a variação angular instantânea do controle, uma vez com a certeza de lidar com um sinal de velocidade angular, é necessário fazer a integração do sinal em relação à variação temporal, tal como confirmado em (Wikipedia, 2010). Como o processo é composto apenas por variáveis definidas (sem incógnitas) em um intervalo definido, a técnica de integração deverá ser numérica. A técnica escolhida foi a regra trapezoidal, devido sua fácil adaptação ao problema e pequeno erro inerente (quando comparado com as outras técnicas disponíveis para aplicação neste problema), como descrito em (Wikipedia, 2009). Sua equação consiste de:

$$(1) \quad \int_a^b f(x)dx \cong (b-a)\frac{f(a)+f(b)}{2}$$

A integração trapezoidal é uma técnica de aproximação da integral definida, que possui um erro inerente, e graficamente pode ser representada como a figura a seguir:

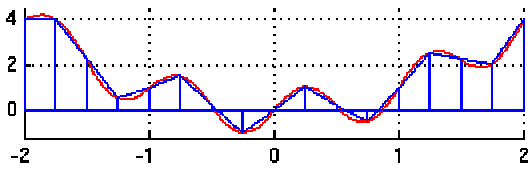


Figura 16: Demonstração gráfica da integração trapezoidal em relação à função definida. Os trapézios (em azul) aproximam-se o máximo possível da curva da função (em vermelho).

Entretanto, como se trata de uma integração em tempo real, de valores informados pelo giroscópio $f(t)$ (pois variam em função do tempo) em relação ao tempo Δt, podemos descrever a equação em termos de t como:

$$(2) \quad \int_{t-1}^{t} f(t)dt \cong \Delta t\frac{f(t-1)+f(t)}{2}$$

Esse procedimento deve ser repetido para cada eixo do controle. Nesse caso, utilizamos as variáveis **m_v3_fA_AngularVelocity** ($\boldsymbol{f(t-1)}$)e **m_v3_fB_AngularVelocity** ($\boldsymbol{f(t)}$), do tipo **Vector3** (ou seja, que armazena as informações dos três eixos), nas quais as medições do sinal são registradas para cada instante discreto de tempo $\boldsymbol{t}$. Finalmente, considerando que, para o método **Update( )**, $\boldsymbol{\Delta t} =$ **GameTime.ElapsedGameTime.TotalSeconds**, que representa a medição do intervalo de tempo entre as chamadas desse método, é possível calcular a integração.

Uma vez calculada a integração, obtêm-se a variação angular para aquela variação temporal $\boldsymbol{\Delta t}$. Entretanto, essa variação angular ainda está em termos da escala própria do Wii Motion Plus™, então multiplicamos o resultado pelos multiplicadores definidos acima para converter a variação em termos de ângulos de Euler, que pode ser utilizado pela aplicação.

Com a obtenção da variação angular, o passo final é filtrar o ruído. Como visto em (Aveline, 2009), este ruído depende unicamente de fatores de hardware e condições de funcionamento, então se pode afirmar que ele independe do modo de operação (que não passa de *flags* de controle interpretadas via software). Portanto, uma vez que em modo

rápido qualquer variação na medição do sinal representa uma variação angular de maior magnitude que em modo preciso, a variação decorrente do ruído (que permanece a mesma em termos absolutos) é amplificada por essa diferença de magnitude.

Sabendo disso, os multiplicadores são levados em conta no cálculo do filtro. A técnica de filtragem adotada foi a adoção de um valor de corte, um limite inferior, independente para cada eixo, que deve ser atendido para o sinal ser considerado uma variação angular válida. Qualquer valor abaixo desse limite inferior é descartado. O valor de corte é calculado com a expressão **ANGULAR_ERROR_THRESHOLD * fXXXMultiplier** (sendo **XXX** o ângulo sendo considerado, *Pitch*, *Yaw* ou *Roll*, uma vez que esse valor de corte é definido para os três eixos). A constante **ANGULAR_ERROR_THRESHOLD** foi definida a partir de observações do comportamento do sinal, e se ajusta bem ao dispositivo utilizado.

Após todos esses passos tem-se variações angulares em termos de ângulos de Euler (*Pitch*, *Yaw* e *Roll*), variações estas relativas ao eixo do próprio controle, que então são convertidas para o *quaternion* **m_quaDeltaRotation**. Sabendo que, ao multiplicar dois *quaternions*, o primeiro será considerado uma variação angular em relação ao eixo absoluto do segundo, multiplica-se o *quaternion* obtido com o *quaternion* **m_quaRotation,** que representa a orientação absoluta do controle (inicializado como Identidade). Desta maneira, atualiza-se a orientação absoluta e conclui-se esta etapa.

O algoritmo utilizado nesta etapa está sujeito a um erro cumulativo decorrente de diversos fatores, como erro da técnica de integração trapezoidal (como pode ser observado na **Figura 16**, e descrito pela equação a seguir), erro da técnica de filtragem (que pode descartar valores relevantes), erro de ruído (não detectado pelo algoritmo) e até mesmo erro na técnica de normalização do sinal (que foi desenvolvida por observação de valores amostrais). Este erro faz com que a estimativa da orientação seja menos precisa, e por ser cumulativo ele aumenta proporcionalmente à duração contínua do processo. Na equação a seguir, o termo $\boldsymbol{x_0}$ representa um ponto dentro do intervalo de integração e o termo $\boldsymbol{h}$ representa o comprimento desse intervalo.

$$(3) \quad -\frac{h^3}{12} f''(x_0)$$

Tendo isso em mente, a solução adotada para minimizar este tipo de erro foi limitar a execução do algoritmo à instantes que o botão **B** do controle estivesse pressionado. Desta maneira, toda vez que o botão não estiver pressionado o erro cumulativo do processo é

zerado, e também permite ao usuário fazer ajustes manuais para corrigir a orientação do controle.

Obtenção do deslocamento através dos botões do *Wiimote*

O projeto até a etapa corrente já permite o monitoramento de três graus de liberdade de movimento, de rotação. A obtenção dos outros três graus de liberdade, de deslocamento, é realizada através da interpretação dos botões do *Wiimote*, utilizando o direcional para controlar o movimento para frente, para trás, para esquerda e para direita (no plano horizontal), e utilizando os botões **+** e **-** para, respectivamente, controlar o movimento para cima e para baixo (no plano vertical). O movimento então se dá em velocidade fixa, com três níveis pré-configurados no software. Por fim, a movimentação obtida com o pressionar destes botões é transformada de acordo com a orientação do controle, pois toda movimentação se baseia em relação um eixo relativo, e nunca absoluto.

Comunicação da classe

A comunicação da classe se dá através da implementação de uma interface chamada **IWiimoteInput**. O objetivo da classe é fornecer a orientação e o deslocamento interpretados do *Wiimote*, portanto o acesso à esses dados é feito através de *quaternions* para a rotação e vetores para o deslocamento.

Para a orientação são utilizados dois *quaternions*: **AbsoluteOrientation** e **DeltaOrientation**, representando, respectivamente, a orientação absoluta do controle e a variação angular instantânea obtida para aquele instante de tempo.

Para o deslocamento são utilizados três vetores tridimensionais: **AbsolutePosition**, **DeltaPosition** e **UntransformedDeltaPosition**, representando, respectivamente, a posição absoluta do controle, o deslocamento instantâneo transformado pela orientação absoluta e o deslocamento instantâneo sem transformar por esta orientação.

A implementação do algoritmo para esta classe, descrito nessa seção, pode ser conferido no Apêndice III, Seções 2.3 e 2.4.

3.2.3. Classe InverseKinematics

Uma vez concluída a etapa de interpretação do *Wiimote*, a última etapa do projeto consiste em aplicar as informações obtidas de rotação e deslocamento na movimentação do

braço mecânico. A abordagem adotada por este projeto foi de aplicar estas informações diretamente à mão do braço mecânico, também chamado, no domínio deste problema, de efetuador. Primeiramente, aplica-se a rotação à junção do efetuador. Em seguida aplica-se o deslocamento obtido, transformado de acordo com a rotação (para dar o efeito de avançar sempre na direção cuja mão está apontando), novamente ao efetuador. Entretanto, enquanto a rotação poderia ser aplicada diretamente à junção da mão, o mesmo não poderia ser feito com o deslocamento, pois isso faria com que a mão se desconectasse do braço mecânico. O deslocamento do efetuador está sujeito à limitação de movimentação de seus membros anteriores.

Em outras palavras, a mão herda os efeitos da movimentação de seus membros anteriores. Por exemplo, na cinemática tradicional, ao rotacionar o braço, o antebraço e a mão são rotacionadas e deslocadas de acordo. Isso ocorre porque esses membros são níveis inferiores na hierarquia de conexões. O mesmo ocorre ao rotacionar o antebraço, a mão é deslocada e rotacionada de acordo. Como pode ser observado, em momento algum pode ser aplicado diretamente o deslocamento, pois isto causaria a disjunção de um ou mais membros da estrutura. Então, para movimentar a mão de acordo com o deslocamento obtido pelo *Wiimote*, é necessário traduzir este deslocamento em rotações para os membros anteriores, de forma que estes façam a mão chegar o mais próximo possível de seu objetivo. Este tipo de técnica se chama cinemática inversa.

A cinemática inversa é um problema clássico da robótica. Segundo (Santos, et al., 2005), o problema pode ser descrito como *"dada a posição-orientação desejada do efetuador, obter o conjunto dos ângulos-de-junta que alcançam esta posição-orientação"*. O citado efetuador, no caso deste projeto, é a mão do braço mecânico. Trata-se de um problema complexo, de solução não linear, com diversas abordagens possíveis (Tolani, et al., 2000). Neste projeto, a solução adotada consiste da utilização do algoritmo *Cyclic Coordinate Descent* (CCD), descrito em (Habib, 2008), devido sua fácil aplicabilidade a este projeto, e seu fácil entendimento. A implementação utilizada foi baseada em (XNA Creators Club, 2009).

Este algoritmo interativo faz uso de uma heurística (descrita abaixo) para determinar, para cada membro, o ângulo a ser percorrido para aproximar o efetuador o máximo possível de seu objetivo. Segundo (XNA Creators Club, 2009 p. HTML Documentation), a implementação adotada para este projeto realiza o cálculo da junta mais próxima do

efetuador para a junta mais distante, até atingir a raiz da estrutura. O algoritmo requer como entrada a posição e orientação de cada junta da estrutura, e a posição final que se deseja para o efetuador. Para cada junta, é realizada uma sequência de passos. Primeiramente, computa-se dois vetores, um vetor com a direção da junta atual ao efetuador, e outro vetor com a direção da junta atual à posição de destino, e os normaliza. Em seguida, computa-se a matriz de rotação necessária para rotacionar o efetuador para a posição de destino. Para isso, primeiramente computa-se um eixo de rotação ao fazer o produto vetorial (produto externo) entre os dois vetores obtidos, e então computa-se o produto escalar (produto interno) entre os dois vetores para obter o ângulo de rotação ao redor deste eixo obtido. Desta maneira, usando o eixo e o ângulo obtido, pode-se determinar a matriz de rotação necessária para aquela junta. Finalmente, aplica-se a matriz de rotação obtida à junta em questão e itera para a próxima junta da estrutura.

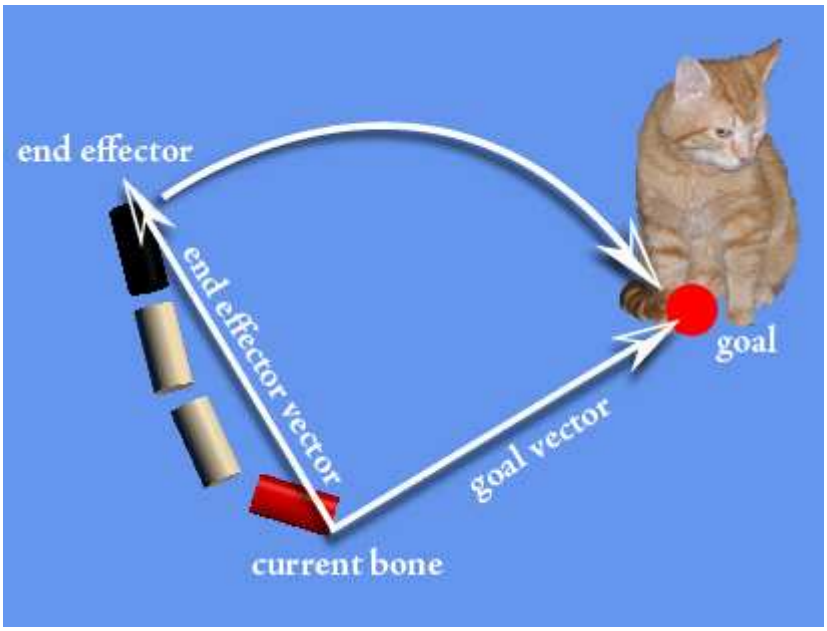


Figura 17: Esquema explicativo de um passo da iteração na implementação utilizada do algoritmo *Cyclic Coordinate Descent* (CCD). Imagem retirada de (XNA Creators Club, 2009 p. HTML Documentation).

No caso da classe **InverseKinematics** em questão, o algoritmo foi implementado em dois métodos estáticos. O método **UpdateBone( )** realiza o cálculo descrito acima, apenas para uma junta. A iteração entre as juntas fica a cargo da classe **BracoMecanico**. O método **UpdateTransforms( )** se encarrega de atualizar todas as matrizes de transformação dos *bones* da estrutura após a movimentação realizada no método anterior. Esta implementação garante que a classe **InverseKinematics** pode ser reutilizada facilmente por qualquer outra

aplicação de animação 3D, uma vez respeitado o formato das estruturas de dados necessárias para seus parâmetros. Vale ressaltar que o algoritmo desenvolvido, em prol da legibilidade, não é otimizado. Também vale notar que não há restrições (*constraints*) de movimentação para as juntas.

O código-fonte com a implementação desta classe pode ser conferido no Apêndice III, Seção 2.5.

3.2.4. Classe Camera

Para plotar na tela qualquer elemento tridimensional em uma aplicação que utiliza o framework XNA™, é necessária a configuração de uma câmera. Esta classe implementa uma câmera estática, com a habilidade de olhar ao redor, como um **GameComponent**, que fornece as matrizes de projeção e de visão necessárias para a aplicação. O acesso à essas matrizes é feito através da interface **ICamera**. Para controlar a visão da câmera via teclado esta classe faz uso do componente **InputHandler**. A implementação adotada neste projeto segue a sugestão de (Carter, 2008).

O código-fonte com a implementação desta classe pode ser conferido no Apêndice III, Seções 2.6 e 2.7.

3.2.5. Classe FirstPersonCamera

A classe **FirstPersonCamera** estendeu o funcionamento da classe **Camera**, permitindo também a movimentação pelo ambiente usando as teclas W, A, S, D, de modo similar ao funcionamento de uma câmera em primeira pessoa, e sua implementação também é baseada em (Carter, 2008), e pode ser conferida no Apêndice III, Seção 2.8.

3.2.6. Classe FPS

A última classe da biblioteca consiste de um componente do tipo **DrawableGameComponent** que atua como contador de frames por segundo. Sua função se resume a mensurar o desempenho do código ao exibir continuamente, na barra de títulos do software, a quantidade de frames *renderizados* por segundo. Sua implementação foi baseada em (Carter, 2008), e seu código-fonte pode ser conferido no Apêndice III, Seção 2.9.

3.3. APLICAÇÃO **SIMULACAO**

A camada **Simulacao** consiste da aplicação de simulação do braço mecânico em si. Ela é responsável por agrupar as classes de base para a execução da simulação com o XNA™ framework, que por sua vez fazem uso intensivo da biblioteca **WiimoteInputLib**, desenvolvida nesse projeto, para operar o braço mecânico. Ela agrupa três classes: **SimulacaoBracoMecanico**, **BracoMecanico** e **Program**, descritas abaixo. Os relacionamentos entre essas classes e os componentes da biblioteca podem ser visto com mais detalhes no **Diagrama 1**.

A classe **SimulacaoBracoMecanico** é a principal classe da aplicação. Derivada da classe **Game**, ela consiste do *Game Loop* do XNA™ em si. Nesta classe são registrados todos os componentes e serviços utilizados pela aplicação, cujos quais o framework se encarrega de chamar corretamente seus métodos.

A classe **BracoMecanico** encapsula toda a operação de controle, movimentação e desenho do braço mecânico pela aplicação. Esta classe faz uso intensivo da classe **WiimoteInput**, para obter as informações do controle e convertê-las para a movimentação do braço mecânico através da classe **InverseKinematics**. Uma vez atualizado o posicionamento do braço mecânico, este é plotado na tela.

Por fim, a classe **Program** consiste simplesmente do ponto de entrada para a aplicação. Ela invoca o início do *Game Loop* ao instanciar e invocar o início da operação da classe **SimulacaoBracoMecanico**.

O funcionamento de cada uma dessas classes é detalhado nas seções a seguir.

3.3.1. Classe SimulacaoBracoMecanico

Classe principal da aplicação, com herança da classe **Game**, **SimulacaoBracoMecanico** é a classe que implementa o funcionamento do *Game Loop* do XNA™ framework. O funcionamento dessa classe, principal componente da simulação como uma aplicação em tempo real, possui a mesma estrutura descrita no Apêndice I.

Em resumo, esta classe é responsável pelo instanciamento de todos os componentes necessários para sua operação, desde os criados na biblioteca **WiimoteInputLib** até o componente **BracoMecanico**. Além de instanciá-los, a classe também é responsável pela chamada dos métodos de funcionamento dos **GameComponent** registrados, como

Initialize( ), **Update( )** e **Draw( )**. O único comando inserido diretamente nesta classe foi, dentro do método **Update( )**, a chamada do método **Reset( )** da classe **BracoMecanico** ao pressionar a tecla **R** do teclado, para reinicializar o posicionamento do braço mecânico.

O código-fonte com a implementação desta classe pode ser conferido no Apêndice III, Seção 3.1.

3.3.2. Classe BracoMecanico

A classe **BracoMecanico** é um componente com herança de **DrawableGameComponent** e é utilizada para encapsular a exibição e manipulação do modelo 3D do braço mecânico.

Funcionamento interno e recursos utilizados

O funcionamento interno desta classe é o mesmo dos outros componentes, como descritos no Apêndice I.

Em seu construtor são resgatados, da classe pai **Game**, os componentes necessários para sua execução: **Camera**, **InputHandler** e **WiimoteInput**.

No método **LoadContent( )** são carregados os dois recursos necessários para a execução da simulação do braço mecânico: o modelo tridimensional do braço mecânico em si, e um modelo tridimensional de uma bola, utilizado como debug indicando a posição exata do efetuador, para a operação de cinemática inversa. Também é feita a inicialização das variáveis de controle do braço mecânico ao chamar o método **InicializaBraco( )**.

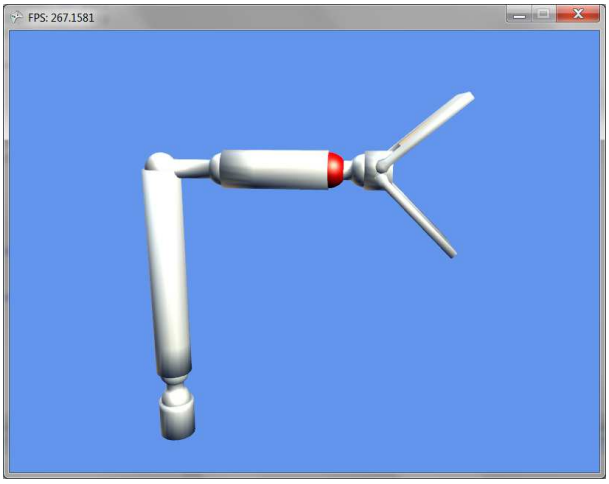


Figura 18: Recursos utilizados no projeto, demonstrados na aplicação em execução. Em branco, o modelo tridimensional de braço mecânico desenvolvido para este projeto. Em vermelho, o modelo

tridimensional da bola, usada como debug do processo de cinemática inversa ao simular o efetuador do modelo.

No método **Update( )** é realizada a leitura do input gerado pela classe **WiimoteInput**, e então atualizado o posicionamento do braço mecânico de acordo com as informações recebidas e processadas pelo processo de cinemática inversa. Para isso, os métodos da classe **InverseKinematics** são chamados para cara *bone* do modelo, usando como parâmetros as variáveis de controle do braço e o posicionamento gerado pela **WiimoteInput**.

Finalmente, no método **Draw( )** é feito o desenho do braço mecânico na tela e da bola vermelha de debug, utilizando suas variáveis de controle, de acordo com a posição de câmera obtida pela classe **FirstPersonCamera**.

Animação do modelo

A animação do modelo tridimensional do braço mecânico consiste basicamente em criar as estruturas de dados necessárias para o uso dos métodos da classe **InverseKinematics**.

Uma vez com o modelo 3D carregado, são necessários quatro vetores de matrizes para descrever seu posicionamento. Cada matriz dessas listas está associada à um determinado *bone* em um certo contexto. O primeiro consiste de um vetor de matrizes que armazena a posição inicial de cada *bone* do modelo, chamada de *bind pose*, denominado **m_cmxBindPose**. O segundo consiste de um vetor de matrizes que armazena a posição atual de cada *bone* relativa à *bind pose*, ou seja, o *offset* atual de cada *bone* relativo à sua posição inicial, denominado **m_cmxBoneTransforms**. O terceiro consiste de um vetor de matrizes que armazena a posição de cada *bone* relativa à posição de seu *bone* pai, denominada **m_cmxLocalTransforms**. Por fim, o último vetor de matrizes armazena a posição absoluta de cada *bone*, para facilitar a operação de desenho, denominado **m_cmxWorldTransforms**.

Além dos vetores de matrizes, também são criadas dois vetores de índices. O primeiro armazena os índices do pai de cada *bone*, que é intensivamente utilizado ao pesquisar pelos vetores de matrizes acima, denominado **m_ciBoneParents**. O segundo, por sua vez, armazena os índices dos *bones* do modelo que serão afetados pela técnica de cinemática inversa (a mão do modelo, por exemplo, não faz parte das iterações do processo, somente suas estruturas anteriores), denominado **m_ciBoneChain**. Para facilitar a

legibilidade do código, o índice do efetuador nos vetores de matrizes é indicado pela variável **m_iIndiceEndEffector**.

Para animar o movimento de abrir e fechar a mão do braço mecânico foi necessária a criação de variáveis para armazenar o índice de seus respectivos *bones*, da garra superior e inferior, e para controlar o grau de abertura dessas peças. Os índices dos *bones* são armazenados em **m_iIndiceGarraSuperior** e **m_iIndiceGarraInferior**, enquanto a abertura máxima e atual dessas peças são armazenadas em **m_fAnguloAmplitude** (igual a 43°, constante) e **m_fAnguloAmplitudeAtual** (inicializada com zero).

A inicialização dessas variáveis ocorre no método **InicializaBraco( )**, chamado logo após o carregamento do modelo 3D. Neste método é inicializado o tamanho e posição inicial do braço mecânico. Também são preenchidos os vetores **m_cmxBindPose**, **m_ciBoneParents** e **m_ciBoneChain**, e as variáveis **m_iIndiceEndEffector**, **m_iIndiceGarraSuperior** e **m_iIndiceGarraInferior**, que necessitam essa operação apenas uma vez. Depois de feita a inicialização é imediatamente chamado o método **Reset( )**, que inicializa os vetores variáveis **m_cmxBoneTransforms**, **m_cmxLocalTransforms** e **m_cmxWorldTransforms**, responsáveis por armazenar a posição atual dos *bones* durante a animação. O método **Reset( )** também é chamado pela aplicação sempre que pressionada a tecla R do teclado, para reinicializar a posição do braço mecânico.

No método **Update( )**, primeiramente faz-se a animação de abertura ou fechamento da mão do braço mecânico sempre que detectado que o botão A do *Wiimote* está, ou não, pressionado. Para isso, sempre que detectado o pressionar do botão, incrementa a variável **m_fAnguloAmplitudeAtual** de acordo com um pequeno offset de movimento circular em torno do eixo X local (*pitch*), definido pela constante **ROTATION_RATE**, sempre respeitando o limite imposto pela variável **m_fAnguloAmplitude**. Em seguida calcula-se uma matriz de rotação para cada garra baseando-se neste ângulo e as aplica às respectivas matrizes armazenadas em **m_cmxBoneTransforms**. De maneira análoga, sempre que detectado o soltar do botão decrementa **m_fAnguloAmplitudeAtual** até chegar à posição inicial novamente.

Em seguida, ainda em **Update( )**, faz a leitura da entrada de dados fornecida por **WiimoteInput**. Aplica a rotação obtida diretamente ao *bone* do efetuador, e em seguida, caso informado um vetor de deslocamento não nulo, itera pelos *bones* contidos em **m_ciBoneChain** aplicando progressivamente a técnica de cinemática inversa ao chamar os

métodos **UpdateBone( )** e **UpdateTransforms( )** da classe **InverseKinematics**, usando como parâmetro as variáveis descritas anteriormente. Por fim, inverte as rotações realizadas com este processo e aplica-a à mão do braço mecânico (efetuador), para que esta mantenha sua orientação original, anterior ao movimento.

Finalmente, no método **Draw( )**, são desenhados os modelos 3D, em especial o braço mecânico de acordo com o vetor **m_cmxWorldTransforms**.

O código-fonte com a implementação desta classe pode ser conferido no Apêndice III, Seção 3.2.

3.3.3. Classe Program

A classe estática **Program** consiste apenas de um ponto de entrada para o aplicativo. Possui apenas um método público **Main( )** que instancia **SimulacaoBracoMecanico** e inicia sua execução ao invocar o método **Run( )**. O código desta classe é gerado automaticamente pelo XNA™ Framework, e pode ser conferido no Apêndice III, Seção 3.3.

4. UTILIZAÇÃO DO SOFTWARE

Para utilizar o software desenvolvido, é necessário um computador com sistema operacional Microsoft® Windows™ XP, Vista ou 7, com placa gráfica que suporte os requisitos mínimos de execução do XNA™ Framework 3.1 e conectividade Bluetooth (ou um adaptador que a ofereça); e um *Wiimote* com a extensão Wii Motion Plus™ acoplada.

4.1. CONECÇÃO VIA BLUETOOTH

Para conectar via Bluetooth, o *Wiimote* deve estar em modo de descoberta. Para isto, basta manter pressionados simultaneamente os botões **1** e **2**. As luzes azuis, na parte inferior do controle, devem estar piscando. Manter pressionados os botões durante todo o processo de conexão do *Wiimote* ao PC.

No caso específico do Windows XP™, o procedimento de conexão deve ser realizado através do item “Dispositivos Bluetooth”, normalmente disponível como ícone na bandeja do sistema. Se não houver ícone na barra de tarefa, a entrada “Dispositivos Bluetooth” pode ser acessada através do “Painel de controle”. Ao clicar com o botão direito em cima do ícone do Bluetooth, o item “Mostrar dispositivos Bluetooth” deve se selecionado. Aparecerá uma janela, onde o botão “Adicionar” deve ser clicado.

Uma nova janela surgirá, na qual o item “Meu dispositivo está configurado e pronto para ser encontrado” deve ser selecionado, para em seguida clicar no botão “Avançar”. Quando for procurar, os botões **1** e **2** devem estar pressionados e as luzes do *Wiimote* devem estar piscando. Após encontrar o *Wiimote,* normalmente com um nome similar a “Nintendo RVL-CNT-01” o botão “Avançar” deve ser pressionado.

Por fim, como opção de pareamento, o item “Não usar uma senha” deve ser selecionado (com os botões do *Wiimote* ainda pressionados e as luzes ainda piscando). Em seguida basta pressionar o botão “Avançar” e depois “Concluir”.

O procedimento é similar para as outras versões do Windows™, com telas e nomes de opções levemente diferentes.

4.2. INICIALIZAÇÃO DO PROGRAMA

Com o *Wiimote* conectado ao PC, é necessário deixar o controle em cima da mesa, totalmente parado (com os botões para cima) para inicializar o programa.

O programa calibra automaticamente no início da execução. Então ao iniciar o software, basta esperar dois segundos para que o controle seja calibrado. Após esse tempo o *Wiimote* pode ser manuseado. Caso o controle, durante todo o período de calibração, for movimentado ou não estiver sobre a mesa, basta colocá-lo novamente em posição, pressionar a tecla **Espaço** e esperar por dois segundos pelo processo de recalibração.

4.3. CONTROLE DO BRAÇO MECÂNICO

O controle do braço mecânico foi separado em três modos distintos de funcionamento (sendo os modos 2 e 3 apenas para demonstração dos resultados obtidos na pesquisa do Apêndice II). Esses modos de funcionamento determinam como ocorre o deslocamento do braço mecânico. O modo pode ser alterado pressionando o botão **1** do *Wiimote*, e funcionam como descrito a seguir:

- Modo 1 – Movimento Digital (modo *default*): Para rotacionar a mão do braço mecânico, deixar pressionado o botão **B** enquanto movimenta-se o *Wiimote.* Para mover o braço mecânico, usar os botões **direcionais** para ir para frente, para trás, esquerda e direita, e os botões **+** (mais) e **–** (menos) para subir e descer, respectivamente. O botão **2** alterna entre as três velocidades de movimentação possíveis (iniciando pela mais baixa).
- Modo 2 - Acelerômetro com integração simples: Deixar o **B** pressionado para detectar rotação e translação feita no controle (botões direcionais deixam de funcionar).
- Modo 3 - Acelerômetro com integração dupla e técnicas de correção: Deixar o **B** pressionado para detectar rotação e translação feita no controle (botões direcionais deixam de funcionar).

O modo de operação atual é indicado pelos dois primeiros LEDs do *Wiimote*, como representado pela figura a seguir:

Figura 19: Disposição dos dois primeiros LEDs do *Wiimote* para cada modo de operação.

De maneira análoga, enquanto em modo 1, a velocidade de movimentação do braço mecânico é indicado pelos dois últimos LEDs do *Wiimote*, como representado pela figura a seguir:

Figura 20: Disposição dos dois últimos LEDs para cada velocidade do modo 1 (*default*) de operação.

Para fechar a garra, deixar o botão **A** do *Wiimote* pressionado. Para retornar o braço mecânico à posição original, pressionar **R** no teclado.

Para fazer a extração (*dump*) de informações dos sinais emitidos pelo *Wiimote* e dos resultados obtidos pelo programa (interpretando-os a cada instante), basta deixar o botão **Home** do *Wiimote* pressionado enquanto se mantém pressionado o botão **B**. Será gerado um arquivo do tipo CSV com as medições do acelerômetro e do giroscópio, a velocidade e o deslocamento calculados pelo programa. Este arquivo CSV pode ser aberto facilmente pelo Microsoft® Excel™ (ou ferramenta similar de planilhas) para a geração de gráficos para análise.

4.4. CONTROLE DA CÂMERA

Para mover a câmera, usar as teclas **W**, **A**, **S**, **D** para se deslocar, e os **direcionais** (Cima, Baixo, Esquerda e Direita, do teclado) para mudar a direção da visão.

4.5. SAIR DO PROGRAMA

Para sair do programa, deve-se pressionar a tecla **Esc**.

5. RESULTADOS OBTIDOS

Neste capítulo será feita uma análise dos resultados obtidos com o projeto. Será analisada, principalmente, a precisão obtida na orientação do controle, que utiliza o giroscópio do Wii Motion Plus™. Por fim, será feito um balanço geral dos resultados obtidos neste projeto.

5.1. PRECISÃO OBTIDA NA ORIENTAÇÃO DO CONTROLE

A precisão obtida com os resultados do processo de interpretação do sinal do giroscópio do Wii Motion Plus™ foi extremamente satisfatória. Durante a medição de movimentos de rotação mais lentos, o software acompanha com uma margem de erros muito pequena, quase imperceptível, que não chega a 5°. Entretanto, a precisão cai em duas situações: primeiramente, quando são realizados movimentos mais rápidos, sugerindo que o processo de normalização do sinal do Wii Motion Plus™ quando em modo rápido precisa de aperfeiçoamentos; e também durante a continuidade muito longa da execução do processo, que sugere um erro cumulativo inerente do processo de integração numérica.

Em ambos os casos, o desvio pode ser corrigido pelo usuário ao manualmente reiniciar a execução do monitoramento de rotação ao soltar e pressionar novamente o botão B do *Wiimote*. O desvio criado pode ser manualmente removido com esta técnica, e também o erro cumulativo do processo de integração numérica é zerado toda vez que esta ação ocorre.

5.2. AVALIAÇÃO GERAL DOS RESULTADOS

A implementação do controle de deslocamento do objeto através dos botões do *Wiimote*, combinada com a orientação obtida pelo monitoramento do sinal do giroscópio,

permitiu que fosse concluído o objetivo final deste projeto: a criação de uma nova forma de controle, mais intuitiva, de rotação e deslocamento de um braço mecânico.

A combinação do monitoramento da orientação com o deslocamento através de botões, em seguida aplicados ao braço mecânico através da técnica de cinemática inversa adotada, trouxe um resultado bastante satisfatório e preciso no que tange o controle de tal dispositivo. A proposta inicial, de utilizar o *Wiimote* para criar uma nova forma, mais amigável, de interagir com este tipo de dispositivo foi atingida com sucesso.

O controle do braço mecânico pode ser realizado com facilidade. A orientação da mão deste dispositivo pode ser obtida com a interpretação da rotação do controle. Desta maneira ocorre a tradução em tempo real do monitoramento de três graus de liberdade de rotação da mão do usuário diretamente para a mão do braço mecânico. O deslocamento dos membros, controlado pelos botões do *Wiimote*, configuram outros três graus de liberdade (de translação) e permitem a realização de movimentos rápidos e precisos (com velocidade configurável) necessários para a operação deste tipo de dispositivo.

CONCLUSÃO

Neste trabalho foi desenvolvida uma técnica para controle de braços mecânicos utilizando o *Wiimote* e o Wii Motion Plus™. Foi possível monitorar em tempo real a orientação do controle, em três graus de liberdade de rotação, transferindo esta informação diretamente para o efetuador do braço mecânico. Além disso, o deslocamento obtido pelo pressionar de botões do controle, mapeado em um movimento de velocidade constante pré-configurada em três níveis e transferida para o efetuador através de cinemática inversa, permite a realização rápida e intuitiva de movimentos precisos com o equipamento (na simulação).

Além disso, foi realizado um estudo sobre interpretação do sinal dos sensores para monitoramento em tempo real de seis graus de liberdade de movimento, combinando informações do acelerômetro e do giroscópio. Este estudo e seus resultados são apresentados no Apêndice II deste trabalho.

A pesquisa realizada neste trabalho foi viabilizada pelo desenvolvimento de uma aplicação de testes que deu suporte à interpretação do sinal, usando muita tentativa, erro e experimentação, permitindo verificar a efetividade do método pesquisado. A aplicação final foi construída sobre a biblioteca implementada neste trabalho, e permite o controle da simulação de um braço mecânico tridimensional, de maneira simples e intuitiva, utilizando o *Wiimote* em conjunto com o Wii Motion Plus™.

O principal percalço encontrado no presente trabalho ocorreu durante a pesquisa apresentada no Apêndice II, sobre o monitoramento da movimentação do controle através de seu acelerômetro. Na ocasião detectou-se que os resultados obtidos divergiam muito do esperado, e desta maneira eram insuficientes para a execução verossímil da simulação. Na tentativa de resolver o problema foram feitas análises gráficas do sinal, em especial do comportamento de amostras com um movimento pré-determinado, e a conclusão a que se chega é que o erro foi causado pela utilização de técnicas impróprias, ou exageradas, para a

manipulação e filtragem do sinal do acelerômetro. Isso causou uma distorção muito grande no sinal, mudando sua forma fundamental e impossibilitando a interpretação da aceleração fornecida pelo dispositivo.

A principal conquista deste trabalho foi, sem dúvida, a implementação do monitoramento em tempo real da variação de rotação fornecida pelo giroscópio do Wii Motion Plus™, extensão acoplada ao *Wiimote*. O monitoramento da rotação é feito com precisão. Sua margem de erro é pequena, e quando ocorre pode ser facilmente corrigida pelo usuário. Esta técnica, quando combinada com o deslocamento através dos botões, criou uma forma mais intuitiva de controlar um braço mecânico.

A modulação do código da aplicação na biblioteca **WiimoteInputLib** também trouxe inúmeras vantagens. A principal é a versatilidade e reusabilidade do código, e também a possibilidade de portabilidade (necessitando apenas de algumas adaptações aos métodos inerentes do XNA™). A classe **WiimoteInput**, responsável pela interpretação do *Wiimote* e do Wii Motion Plus™, possui uma interface unificada e simplificada de acesso. Desta maneira, uma vez corrigido o problema do acelerômetro (em trabalhos futuros), basta aplicar o algoritmo ao código, sem a necessidade de reescrever a classe.

Como trabalho futuro, primeiramente, é necessária uma pesquisa mais aprofundada sobre o acelerômetro do *Wiimote*, complementando o trabalho apresentado no Apêndice II. O entendimento aprimorado do funcionamento deste dispositivo é vital para a criação de filtros mais eficientes para o melhoramento de sua precisão. Ou, constatada a inviabilidade de obter esta precisão, obter os resultados gerados para compará-los com valores de movimentos pré-definidos e, desta forma, obter um movimento que melhor se adéqua ao movimento realizado. A biblioteca desenvolvida neste projeto está preparada para a situação em que, uma vez resolvido o problema de interpretação do acelerômetro, o algoritmo pode ser facilmente incorporado à classe, e diretamente utilizado na aplicação desenvolvida de controle do braço mecânico.

Outra possibilidade de trabalho futuro é o aumento da precisão da estimativa de orientação do controle ao utilizar o Wii Motion Plus™. A aplicação de um filtro de Kalman, como descrita em (Luinge, et al., 2005), combinando o giroscópio com as informações de inclinação detectadas pelo acelerômetro, pode aumentar muito a precisão da técnica e, consequentemente, reduzir o erro cumulativo inerente da integração numérica. Também é necessária uma pesquisa mais aprofundada sobre a normalização do sinal do giroscópio do

Wii Motion Plus™, uma vez que a técnica desenvolvida utiliza somente valores experimentais e estimativas obtidas em uma fonte que fez a engenharia reversa do controle.

Por fim, a pertinência de tal estudo é de vital importância para a ciência da computação e as áreas consultadas. O processamento de sinais é uma área crescente. Diversos tipos de dados que utilizamos hoje em dia podem ser considerados um sinal, como uma imagem, por exemplo. O desenvolvimento de técnicas e filtros mais avançados trarão benefícios imensos ao desenvolvimento tecnológico futuro. Também a utilização de dispositivos de controle mais intuitivos é muito relevante e uma necessidade crescente, principalmente no que tange ao manuseamento e controle de inventos cada vez mais complexos. Esse tipo de controle permitirá mais precisão e melhor aproveitamento de recursos, humanos ou não humanos, em suas aplicações. Desta maneira, esse tipo de dispositivo tende a ser adotado mais facilmente pela indústria, e popularizado. A combinação entre o processamento de sinais e a criação de novas técnicas de interação é uma tendência recente, entretanto que se solidifica a cada dia.

REFERÊNCIAS BIBLIOGRÁFICAS

AiLive. 2008. Wii MotionPlus and AiLive's LiveMove 2: Motion recognition and tracking. *YouTube.* [Online] Google, Inc., 28 de 07 de 2008. [Citado em: 28 de 01 de 2010.] http://www.youtube.com/watch?v=acND4sO3pJs.

Aveline, Sofia Sabbado. 2009. Iwata Asks - Wii MotionPlus Pt. 1. *Wii Brasil.* [Online] 08 de 2009. [Citado em: 10 de 02 de 2010.] http://www.wii-brasil.com/lite/lerartigo.php?id=212.

Billas, S. 2002. A data-driven game object system. *Talk at the Game Developers Conference.* 2002.

Carter, Chad. 2008. *Microsoft® XNA™ Unleashed: Graphics and Game Programming for Xbox 360 and Windows.* Indianapolis : Sams, 2008. ISBN 0-672-32964-6.

Guimarães, Renato. 2004. Você tem componentes COM e quer aproveitá-los em .NET? *Linha de Código.* [Online] 01 de 05 de 2004. [Citado em: 05 de 02 de 2010.] http://www.linhadecodigo.com.br/Artigo.aspx?id=309.

Habib, Ismail. 2008. Cyclic Coordinate Descent. *A Computer Geek's Blog.* [Online] 29 de 04 de 2008. [Citado em: 29 de 01 de 2010.] http://www.geekyblogger.com/2008/04/cyclic-coordinate-descent-ccd.html.

Hand, Chris. 1997. A survey 3D interaction techniques. *Computer Graphics Forum.* 5, 1997, Vol. 16, 269-281.

Intuitive Surgical. 2005. Da Vinci Surgical System. *Intuitive Surgical.* [Online] Intuitive Surgical, 2005. [Citado em: 04 de 02 de 2010.] http://www.intuitivesurgical.com/products/davinci_surgicalsystem/index.aspx.

Lee, Johnny Chung. 2009. Hacking the Nintendo Wii Remote. *SCHOOL OF COMPUTER SCIENCE, Carnegie Mellon.* [Online] 13 de 08 de 2009. [Citado em: 06 de 02 de 2010.] http://www.cs.cmu.edu/~15-821/CDROM/PAPERS/lee08.pdf.

Luinge, H. J. e Veltnik, P. H. 2005. Measuring Orientation of Human Body Segments Using Miniature Gyroscopes and Accelerometers. *Med. Biol. Eng. Comput.* 2005. Vol. 43, 273-282.

NASA. 2001. The Amazing Canadarm2. *Science@NASA.* [Online] NASA, 18 de 04 de 2001. [Citado em: 01 de 02 de 2010.] http://science.nasa.gov/headlines/y2001/ast18apr_1.htm.

OSHA. 2008. OSHA Technical Manual. *OSHA.* [Online] United States Department of Labor, 18 de 11 de 2008. [Citado em: 01 de 02 de 2010.] http://www.osha.gov/dts/osta/otm/otm_toc.html.

Peek, Brian. 2007. Managed Library for Nintendo's Wiimote. *Coding4Fun.* [Online] Microsoft, 14 de 03 de 2007. [Citado em: 06 de 02 de 2010.] http://blogs.msdn.com/coding4fun/archive/2007/03/14/1879033.aspx.

—. 2009. Status of Wii MotionPlus Support for WiimoteLib. *Brian Peek's Blog - BrianPeek.com.* [Online] 19 de 06 de 2009. [Citado em: 05 de 02 de 2010.] http://www.brianpeek.com/blog/archive/2009/06/19/status-of-wii-motionplus-support-for-wiimotelib.aspx.

—. 2009. WiimoteLib 1.8 Beta 1 Posted. *Brian's Blog - BrianPeek.com.* [Online] 20 de 07 de 2009. [Citado em: 06 de 02 de 2010.] http://www.brianpeek.com/blog/archive/2009/07/20/wiimotelib-1-8-beta-1-posted.aspx.

PIXART. 1998. PixArt Imaging. *PixArt Imaging Inc.* [Online] Julho 1998. [Cited: Fevereiro 02, 2010.] http://www.pixart.com.tw.

Redell, Dave. 1998. Thinking about accelerometers and gravity. *LUNAR'clips - Newsletter of the Livermore Unit of the National Association of Rocketry.* 1998. Vol. 5, 1. http://www.lunar.org/docs/LUNARclips/v5/v5n1/Accelerometers.html.

2009. rosseto.wordpress.com. *Quiver.* [Online] 11 14, 2009. [Cited: fevereiro 10, 2010.] http://rosseto.wordpress.com/2009/11/14/utilizando-o-controle-wii-remote-no-pc/.

Russell, Stuart J. e Norvig, Peter. 2004. *Inteligência Artificial: tradução da segunda edição.* Rio de Janeiro : Elsevier, 2004.

Santos, Alfranci Freitas, Lopes, Heitor Silvério e Junior, Munif Gebara. 2005. Cinemática Inversa de Trajetórias de Manipuladores Robóticos Redundantes Utilizando Algorítmos Genéticos Com Redução Progressiva do Espaço de Busca. 2005. http://www.munif.com.br/munif/arquivos/artigoAG.pdf?id=36.

Sturman, David J. e Zeltzer, David. 1994. A survey of glove-based input. *IEEE Computer Graphics and Applications.* 1994.

The Guardian. 2009. Toyota ponders UK production cuts. *Guardian.co.uk.* [Online] The Guardian, 26 de 08 de 2009. [Citado em: 03 de 02 de 2010.] http://www.guardian.co.uk/business/2009/aug/26/toyota-suspends-japanese-production-line.

Tolani, Deepak, Goswami, Ambarish e Balder, Norman I. 2000. Real-Time inverse Kinematics Techniques for Anthropomorphic Limbs. s.l. : Ideal Library, 2000. http://www.ideallibrary.com.

Wiibrew. 2009. Wiimote. *WiiBrew.* [Online] 09 de 12 de 2009. [Citado em: 05 de 02 de 2010.] http://wiibrew.org/wiki/Wiimote.

WiiBrew. 2010. Wiimote/Extension Controllers. *WiiBrew.* [Online] 18 de 01 de 2010. [Citado em: 04 de 02 de 2010.] http://www.wiibrew.org/wiki/Wiimote/Extension_Controllers#Wii_Motion_Plus.

Wikipedia. 2010. Degrees of freedom (mechanics). *Wikipedia, the free encyclopedia.* [Online] Wikipedia, 12 de 01 de 2010. [Citado em: 19 de 02 de 2010.] http://en.wikipedia.org/wiki/Degrees_of_freedom_%28mechanics%29.

—. 2010. Industrial Robot. *Wikipedia, the free encyclopedia.* [Online] Wikipedia, 14 de 01 de 2010. [Citado em: 03 de 02 de 2010.] http://en.wikipedia.org/wiki/Industrial_robot.

—. 2009. Integração Numérica. *Wikipedia, the free encyclopedia.* [Online] Wikipedia, 18 de 11 de 2009. [Citado em: 14 de 02 de 2010.] http://pt.wikipedia.org/wiki/Integração_numérica.

—. 2010. Integral. *Wikipedia, the free encyclopedia.* [Online] Wikipedia, 29 de 01 de 2010. [Citado em: 14 de 02 de 2010.] http://pt.wikipedia.org/wiki/Integral.

—. 2008. Wii MotionPlus. *Wikipedia, the free encyclopedia.* [Online] Wikipedia, 17 de julho de 2008. [Citado em: 10 de fevereiro de 2010.] http://en.wikipedia.org/wiki/Wii_MotionPlus.

—. 2006. Wii Remote. *Wikipedia, the free encyclopedia.* [Online] Wikipedia, 08 de 12 de 2006. [Citado em: 01 de 02 de 2010.] http://en.wikipedia.org/wiki/Wii_Remote.

XNA Creators Club. 2009. Inverse Kinematics Sample. *XNA Creators Club Online.* [Online] Microsoft, 09 de 12 de 2009. [Citado em: 28 de 01 de 2010.] http://creators.xna.com/en-US/sample/inversekinematics.

APÊNDICE

I. ARQUITETURA DE UMA APLICAÇÃO DO XNA™ FRAMEWORK

Uma aplicação desenvolvida no XNA™ possui sempre a mesma característica: funciona através de um *Game Loop*. Um *Game Loop* consiste de um conjunto de métodos chamados continuamente, em sequência, para atualização e renderização [18] dos componentes da aplicação em tempo real.

Para implementar o *Game Loop*, o XNA™ provê uma classe chamada **Game**, que provê a estrutura básica para a aplicação. Toda aplicação criada utilizando o XNA™ framework vai possuir uma, e somente uma, classe principal derivada de **Game**. Os métodos e o funcionamento desta classe são explicados abaixo.

A inicialização da aplicação consiste, inicialmente, da invocação do construtor. Em seguida, é executado o método **Initialize( )**, responsável pela inicialização dos atributos da classe, e então executado o método **LoadContent( )**, responsável pelo carregamento de recursos para o projeto, através de seu *Content Pipeline* (sistema interno de conversão de formatos de arquivos para compatibilização de seus conteúdos com a aplicação).

O interior do *Game Loop* em si consiste dos métodos **Update( )** e **Draw( )**, que são chamados continuamente durante a execução da aplicação, sempre em sequência. Em **Update( )** são realizadas atualizações dos atributos de controle da classe, como, por exemplo, interpretação de *input*, posicionamento de objetos e detecção de colisões. No método **Draw( )**, por outro lado, é feita apenas a interpretação das informações contidas nos atributos de controle da classe e as utiliza para "pintar" a cena (*frame*) da simulação, fazendo uso dos recursos carregados. Este processo, de chamar estes dois métodos em

[18] Entende-se por renderização a plotagem na tela de elementos gráficos de uma aplicação em tempo real, notadamente simulações ou jogos.

sequência, ocorre em *loop* (por isso o termo *Game Loop*), continuamente durante toda a duração da execução da aplicação.

Por fim, ao finalizar a aplicação, são chamados os métodos **UnloadContent( )**, responsável por descarregar recursos que não foram carregados através do *Content Pipeline*, e o destrutor da classe, para remover da memória os objetos alocados.

Outra característica importante da arquitetura de funcionamento do XNA™ é a utilização de componentes. Componentes são classes que imitam o funcionamento da classe **Game**, e desta forma permitem uma melhor organização, reaproveitamento e modularização do código.

Um componente do XNA™ é uma classe, derivada de **GameComponent** ou **DrawableGameComponent**, que possui uma estrutura muito similar à do *Game Loop* principal, da classe **Game**. Cada componente possui seus métodos de inicialização, seu método **Update( )** e, no caso de **DrawableGameComponent**, seus métodos de carregamento de recursos e **Draw( )**. O que os torna tão versáteis é que basta instanciá-los na classe principal (derivada de **Game**) e registrá-los como componentes que toda vez que um dos métodos principais da classe **Game** for executado, a versão análoga destes de seus componentes serão chamadas logo em seguida. Por exemplo, após executar o método **Update( )** da classe principal, o mesmo método de cada componente registrado será chamado, como se fossem uma extensão direta de seu funcionamento.

Serviços são componentes com uma característica especial: eles implementam uma interface única de acesso que é registrada na classe principal durante sua criação. Desta maneira, quando um de seus componente precisa se comunicar com outro (considerando que ambos estão registrados na mesma classe principal), eles o fazem através da aquisição de interfaces de serviço. Para realizar a comunicação, uma vez obtida a Interface do componente que se deseja utilizar, basta fazer a coerção[19] para a classe padrão do componente e utilizá-la normalmente.

[19] Operação mais conhecida pelo termo *cast* ou *type cast*.

II. PESQUISA SOBRE *TRACKING* UTILIZANDO O ACELERÔMETRO EM CONJUNTO COM O GIROSCÓPIO

1. Visão geral

Neste apêndice será detalhada a pesquisa feita de uma técnica para realizar o monitoramento (*tracking*), em tempo real, do *Wiimote*, através da combinação de informações fornecidas pelo acelerômetro com o giroscópio.

A implementação parte do ponto em que a orientação do controle é obtida através da interpretação do sinal do giroscópio, como detalhado no Capítulo 3, Seção 3.2.2. Em seguida, executam-se duas etapas:

1) A estimativa da força normal exercida sobre o acelerômetro;
2) A estimativa do deslocamento através da integração dupla da aceleração.

Em termos gerais, a ideia é que, uma vez estimada a orientação que o controle se encontra, é possível estimar o vetor da força normal sendo exercida sobre este e então descontá-la do sinal do acelerômetro. Desta maneira isola-se apenas a aceleração exercida pelo usuário, que pode então ser integrada para obter o deslocamento espacial do controle. Com orientação e deslocamento, teríamos monitoramento em tempo real de seis graus de liberdade.

A estimativa do vetor de força normal que incide sobre o controle é uma etapa que consiste em utilizar a orientação absoluta do controle, obtida anteriormente, como parâmetro de uma operação de transformação de coordenadas. Sabendo que a força normal é um vetor de direção vertical, com sentido “para cima” e intensidade conhecida (igual à gravitacional), transformamos este vetor para coordenadas relativas ao controle, uma vez que todas as medidas dos sensores deste são baseadas num sistema de coordenadas relativo à si próprio. Para aumentar a precisão nesta etapa, foi desenvolvido um mecanismo de autoajuste que se baseia no estado corrente do acelerômetro, descrito a seguir.

A última etapa, de estimar o deslocamento do controle, consiste em descontar o vetor de força normal obtido na etapa anterior da medição gerada pelo acelerômetro. O objetivo é isolar o sinal do acelerômetro apenas com as acelerações exercidas por fatores externos (pelo usuário, principalmente), fazendo então seus devidos cálculos matemáticos de integração numérica para transformar a informação de aceleração medida em velocidade

e, em seguida, deslocamento. Com este deslocamento seria possível rastrear com precisão a posição espacial do controle, que combinada com sua orientação obtida na segunda etapa, permitiria atingir o objetivo central deste projeto, monitorando seis graus de liberdade em tempo real.

Entretanto, esta última etapa não funcionou como esperado. O sinal fornecido pelo acelerômetro, depois da aplicação dos filtros e do desconto da força normal, não correspondeu à uma aceleração como se esperava. Desta maneira o deslocamento obtido pelo algoritmo, refletido na simulação, não correspondia ao deslocamento do controle. Com resultados incoerentes, após muita tentativa e erro, utilização de técnicas de correção e análise gráfica de informações extraídas da aplicação, constatou-se que os resultados obtidos não foram suficientemente satisfatórios, sendo necessárias pesquisas aprofundadas em trabalhos futuros. A análise dos resultados obtidos é realizada na seção 4 deste Apêndice.

2. Estimativa da força normal exercida sobre o acelerômetro

A primeira etapa do processo de interpretação do acelerômetro do *Wiimote* consiste da estimativa da força normal exercida sobre o controle. O objetivo desta etapa é obter um vetor tridimensional, com coordenadas relativas à orientação do controle, contendo o valor da força normal sendo exercida sobre o dispositivo. Utilizando este vetor, é possível isolar as medições do sinal do acelerômetro da perturbação exercida pela gravidade, para que este meça somente as movimentações geradas pelo usuário. Esta etapa é fundamental para o correto funcionamento desse processo.

O algoritmo desenvolvido se baseia na ideia de que, uma vez conhecido o vetor de força normal para o sistema de coordenadas absoluto, aplica-se uma transformação de coordenadas para obter o vetor final. O sistema de coordenadas para o qual deve ser feita a conversão é o sistema relativo ao plano do *Wiimote*, que está rotacionado em relação ao plano absoluto. Esta rotação foi mensurada na etapa anterior, quando estimou-se a orientação do controle. Então, utilizando a orientação do controle, é feita a transformação do sistema de coordenadas de um vetor de força normal estimado durante o processo de calibração.

Sabendo que a força normal é uma força de intensidade e direção iguais à gravitacional, porém de sentido oposto, e sabendo que o acelerômetro do *Wiimote* utiliza uma escala em termos de Força G para realizar suas medições (Wiibrew, 2009 p. Acelerômetro), é válido afirmar que o vetor de força normal a ser descontado do acelerômetro em uma situação ideal (controle virado para cima, sobre uma mesa nivelada, com seu plano perfeitamente paralelo ao plano da mesa), como quando calibrando, é **(0, 1, 0)**, ou seja, um vetor unitário direcionado para cima, que representa **+1.0G** de perturbação no eixo **Y**. Combinando essa informação com os valores de gravidade obtidos durante o processo de calibração é possível determinar o valor inicial para a estimativa da força normal no início da simulação.

Levando em consideração que durante o processo de calibração o controle está perfeitamente paralelo ao plano da mesa (ou muito próximo disso), no decorrer da simulação o que ocorre é a variação da orientação do controle. Essa variação, relativa ao plano da mesa (também considerado o plano absoluto do ambiente tridimensional), é medida pelos *quaternions* obtidos no passo anterior do processo, em especial **m_quaRotation**. Sendo assim, o que os *quaternions* obtidos medem nada mais é do que a variação angular (orientação) do controle em relação ao plano absoluto. De maneira análoga, olhando por outra perspectiva, pode-se afirmar então que a orientação do plano absoluto é equivalente à inversa de **m_quaRotation** em relação ao plano do controle. Através desta analogia infere-se que a transformação do sistema de coordenadas do vetor da força normal, inicialmente conhecido, consiste apenas de transformar o vetor de força normal inicial **(0, 1, 0)** de acordo com a inversa do *quaternion* **m_quaRotation**.

Para melhorar a precisão deste algoritmo, foi criado um mecanismo de autocorreção para o vetor de força normal inicial. Sabendo-se que o acelerômetro, quando parado, mede apenas a força normal exercida sobre si (pois não sofre outro tipo de perturbação externa nessa situação), podemos utilizá-lo como um sensor de inclinação. Baseando-se nesta informação, uma vez que o controle seja detectado em posição constante, seu valor é utilizado para ajustar o vetor de força normal inicial para a força normal mensurada atualmente. Entretanto, ao realizar esta correção, seria incorreto continuar aplicando a inversa de **m_quaRotation**, já que este mantinha inalterado seu ponto de referência inicial (sua inicialização junto com o vetor inicial de força normal). Mas como **m_quaRotation** tem como propósito principal informar a orientação atual do objeto sendo movimentado pelo

controle, ele não poderia ser reinicializado juntamente com a força normal. Desta forma, foi criado um segundo *quaternion*, **m_quaGravity**, que por sua vez acumula a inversa dos **m_quaDeltaRotation** e é utilizado para fazer a conversão da força normal para o sistema do *Wiimote*. E desta maneira, uma vez que o vetor de força normal seja reinicializado, **m_quaGravity** também é reinicializado para um *quaternion* identidade. Isto porque uma vez que **m_quaGravity** apenas acumula os deltas de rotação a partir de um determinado instante de tempo, quando decidimos reiniciar o vetor de força normal podemos fazer o mesmo com este *quaternion*, pois ocorre no mesmo instante temporal.

Esta implementação trás dois grandes benefícios. O primeiro é que o controle pode ser utilizado em qualquer orientação absoluta pelo usuário, uma vez que apenas as variações instantâneas terão efeito sobre os *quaternions* e a força normal exercida sobre o controle é descontada automaticamente com autocorreção, sem importar em qual posição o controle está sendo utilizado (de cabeça para baixo, por exemplo). A segunda é que a técnica utilizada para detectar que o controle se encontra em posição constante provê uma redução automática de ruído, como visto a seguir.

Durante o processo de calibração obtêm-se informações sobre a intensidade da força normal sendo exercida sobre o controle durante um período de dois segundos. Como o controle permanece parado, verifica-se que o desvio padrão dessas amostras é muito pequeno (da ordem de 10^{-5}), de acordo com valores experimentais. Desta maneira, pode-se verificar que o controle permanece parado, ao comparar, com uma determinada tolerância, os valores de calibração com os valores obtidos ao monitorar-se as medições atuais do acelerômetro em um vetor. Este vetor consiste das cem[20] amostras mais recentes fornecidas pelo acelerômetro, e sobre elas é calculada a média e o desvio padrão. Se a média atual estiver próxima suficiente, de acordo com uma tolerância determinada por valores experimentais, da média de calibração (obtida com o controle parado) e o desvio padrão atual for suficientemente pequeno (de ordem semelhante à de calibração), o controle é considerado como parado e a autocorreção do vetor de força normal é ativada. Uma vez que a autocorreção consiste de igualar a força normal à medição do acelerômetro, o ruído do acelerômetro é anulado pela tolerância aplicada ao algoritmo.

[20] Este valor corresponde a meio segundo de amostras, e foi obtido através de experimentação.

Com esta técnica de autocorreção, foi possível estimar um vetor de força normal, chamado de **m_v3Gravity**, que então pode ser descontado da medição do acelerômetro. Note que uma vez que a autocorreção estiver ativa, **m_v3Gravity** é igual à medição do acelerômetro, o que significa que ao descontar este vetor desta medição teremos uma aceleração nula. Isso configura um resultado válido e que reduz o ruído de medição do dispositivo (também gerada por imprecisões e trepidações causadas pela mão do usuário), uma vez que esta autocorreção só é ativada quando o controle é considerado como parado.

3. Estimativa do deslocamento usando o acelerômetro

Uma vez executada a etapa anterior e descontada a força normal das medições do sinal do acelerômetro, as medições resultantes deveriam configurar a aceleração devida apenas ao deslocamento executado pelo controle, o que seria possível interpretar com um processo duplo de integração, análogo ao que foi feito com o sinal de velocidade angular do giroscópio na etapa de estimativa da orientação.

O processo de integração deve ser duplo porque, nesse caso, se trata de um sinal de aceleração, e para obter o deslocamento a partir da aceleração deve-se proceder integrando esta por duas vezes, tal como referenciado em (Wikipedia, 2010). Da mesma maneira como foi feito com o sinal do giroscópio, o processo de integração será numérico e a técnica utilizada será a integração em tempo real através da regra trapezoidal, como indicado na equaç (2). Em termos mais simples, o processo consiste do seguinte: A integral em tempo real da aceleração fornece a variação instantânea da velocidade, que deve ser acumulada na velocidade absoluta do controle. Uma vez obtida essa velocidade absoluta, esta também deve ser integrada em tempo real para obter-se o deslocamento instantâneo, que por sua vez deve ser acumulado na posição absoluta do objeto.

O processo inicia-se então ao inicializar a velocidade (**m_v3Velocity**) e posição absoluta (**m_v3Position**) do controle com um vetor zero. Em seguida, no método **Update( )**, após a realização do desconto da força normal do sinal do acelerômetro, integra-se a velocidade através dos termos **m_v3_fB_Acceleration** e **m_v3_fA_Acceleration**. A velocidade instantânea obtida é armazenada em **v3DeltaVelocity**, que é por sua vez somada a **m_v3Velocity**. Uma vez obtida a velocidade absoluta o processo se repete de maneira análoga, integrando-a através dos termos **m_v3_fB_Velocity** e **m_v3_fA_Velocity**, em

seguida armazenando o deslocamento instantâneo em **m_v3DeltaPosition**, e por fim somando este à **m_v3Position**. Como visto anteriormente, todas as medições realizadas pelo *Wiimote* são relativas à seu próprio eixo de coordenadas. Desta maneira, para obter a posição absoluta do controle de maneira correta, o vetor **m_v3DeltaPosition** deve ser transformado de acordo com a orientação armazenada em **m_quaRotation**. Para compensar essa característica em aplicações que não necessitam do deslocamento absoluto, e sim somente o deslocamento relativo, é armazenado em **m_v3UntransformedDeltaPosition** o deslocamento instantâneo não transformado.

Com este processo obtêm-se o deslocamento do controle de acordo com as medições do acelerômetro. Da mesma maneira como quando integrando o sinal do giroscópio, este processo tem um erro inerente, dessa vez duas vezes maior já que se trata de uma dupla integração, devido à incertezas do processo de integração numérica pela regra trapezoidal, como visto na fórmula (3). As consequências geradas por este erro serão analisadas na seção seguinte.

4. Análise dos resultados com o sinal do acelerômetro

Nesta seção serão analisados os resultados obtidos com a técnica descrita nas seções anteriores, e o sinal resultante do acelerômetro.

Após a conclusão e testes da estimativa do deslocamento usando o acelerômetro, verificou-se que os resultados obtidos divergiam muito do esperado. Para analisar esta divergência foi usado como exemplo um movimento curto, retilíneo, que inicia em repouso, acelera em um eixo (neste caso, acelera negativamente no eixo Z, configurando um movimento para frente), desacelera e em seguida volta a ficar em repouso, esperava-se encontrar uma curva de aceleração ao menos semelhante à de uma senoidal, com um pico e um vale de intensidades proporcionais, uma vez que o movimento deve acelerar e desacelerar na mesma proporção.

O gráfico que inicialmente se esperava como saída produzida pelo acelerômetro, e os resultados obtidos pelo processo de integração dupla, é exemplificado com a **Figura 21** a seguir:

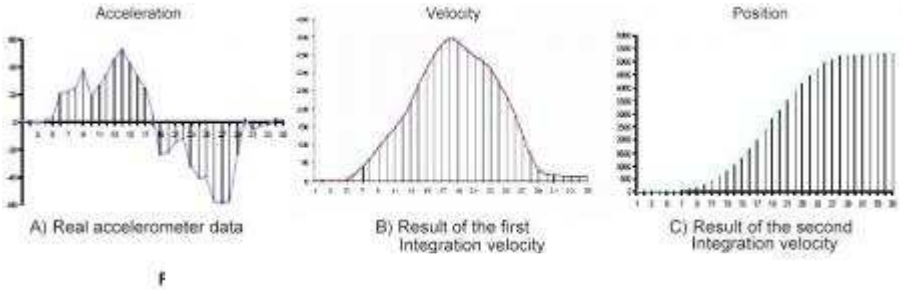


Figura 21: Exemplo de saída esperada do acelerômetro, e o resultado esperado do processo de integração dupla para obter o deslocamento.

Entretanto verificou que o acelerômetro não produz esse tipo de saída, e iniciou-se a pesquisa para tentar explicar a causa deste problema.

O resultado obtido tem duas características estranhas: a velocidade nunca zera (mesmo após o fim do movimento, com o controle parado), o que faz o objeto na tela se deslocar continuamente até desaparecer; e ocorre uma perturbação muito grande em eixos que normalmente não fariam parte daquele movimento (ao menos não em tamanha intensidade), o que faz o objeto se deslocar em direções estranhas ao movimento realizado.

Para tentar resolver o problema da velocidade foi desenvolvido um algoritmo de autorredução. Primeiramente a técnica utilizada consistia de reduzir constantemente a velocidade absoluta do objeto. Entretanto, esta técnica afetava diretamente o gráfico de velocidade e o segundo passo de integração, gerando resultados ainda mais imprecisos. Então foi desenvolvida uma técnica um pouco mais sofisticada, que é ativada ao detectar que o controle está parado. Para fazer esta detecção, utiliza-se um método bastante semelhante ao da autocorreção do vetor de força normal descrito na seção 2 deste apêndice. Monitora-se os cem valores de velocidade mais recentes em um vetor, e ao calcular sua média e desvio padrão, verifica-se que o controle está parado ao detectar uma velocidade constante (ou seja, com desvio padrão muito pequeno, quando comparado com um valor experimental obtido). Desta maneira, sempre que detectada a velocidade constante, a velocidade é gradualmente reduzida até ficar totalmente nula.

Devido ao objeto não mais se manter em velocidade constante quando com o controle parado, como anteriormente, é possível afirmar que esta solução trouxe resultados mais razoáveis, entretanto o problema da perturbação indevida de eixos que não pertenciam ao movimento se manteve. Para tentar explicar este problema e o da velocidade, e tentar solucioná-los através de uma técnica que interferisse menos diretamente nos valores calculados, foi criado um código que extrai, durante um período

contínuo definido pelo usuário ao manter um botão pressionado, as informações fornecidas pelos sensores do *Wiimote* e os valores calculados pelo processo para um arquivo CSV (*comma-separated values*). Este arquivo então pode ser aberto em uma ferramenta de planilhas (como o Microsoft® Excel™) para a fácil geração de gráficos para análise do sinal. Ao realizar este procedimento, para o exemplo citado anteriormente, o gráfico obtido para uma das amostras foi o seguinte (**Gráfico 2)**:

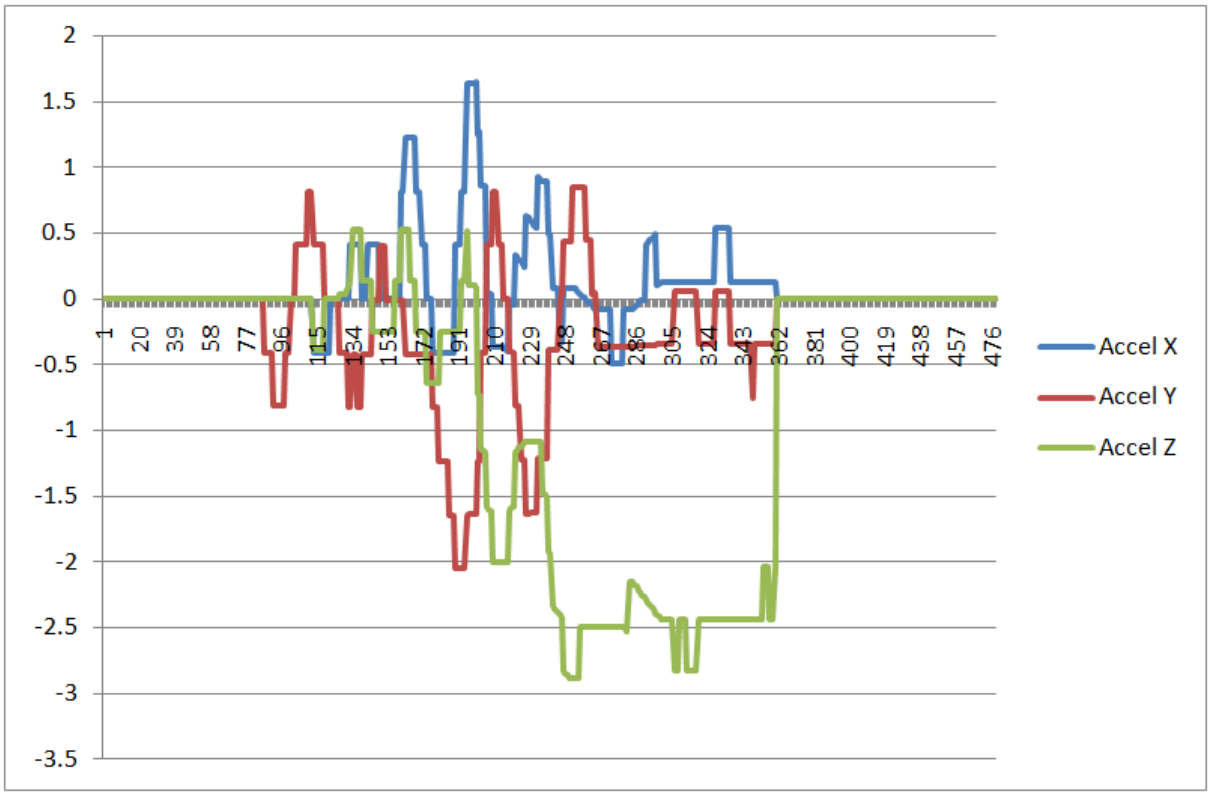


Gráfico 2: Amostra de sinal do acelerômetro para movimento curto, retilíneo, em direção dianteira (cuja perturbação deveria se concentrar principalmente no sentido negativo do eixo Z). Não foi realizada a normalização das curvas, e o giroscópio não influencia este sinal.

Este gráfico mostra claramente que, em um movimento que deveria perturbar principalmente o eixo Z, há uma perturbação de amplitude desproporcional, semelhante à amplitude encontrada em Z, nos eixos X e Y. Essa perturbação causa um erro muito grande na estimativa do deslocamento, que passa a não refletir o movimento feito pelo usuário.

Inicialmente foi verificado o erro causado pela regra trapezoidal na integração dupla, entretanto isto não causaria a perturbação medida nos outros eixos. Um erro no processo de anulação da força normal também não causaria este efeito.

Indo além, para tentar explicar o efeito indevido sobre a velocidade, o **Gráfico 3** demonstra o sinal obtido apenas pelo eixo Z, eixo que deveria concentrar grande parte das medições do acelerômetro para o movimento de exemplo utilizado:

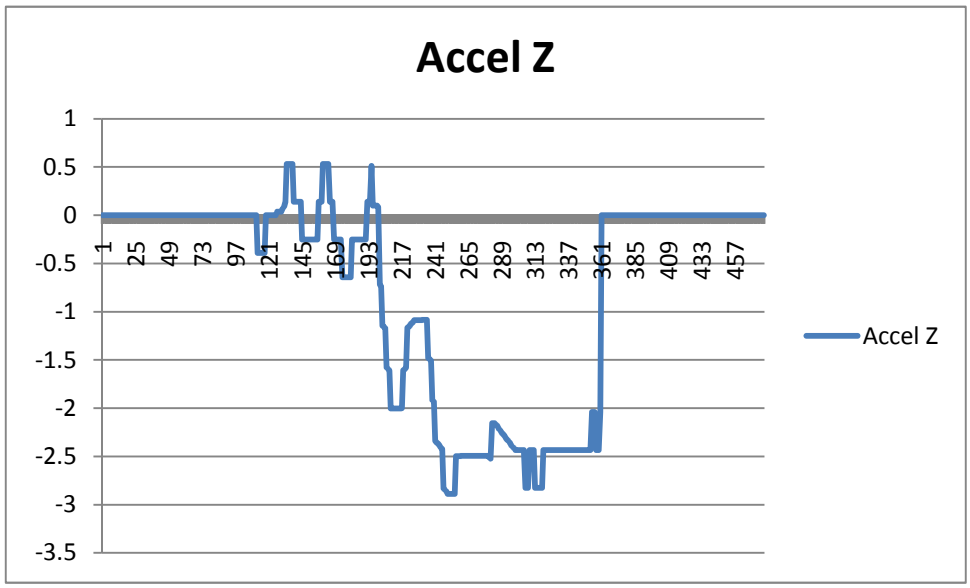


Gráfico 3: Amostra do mesmo exemplo obtida somente para o eixo Z.

Se analisado cuidadosamente, verifica-se que o sinal acima não condiz com um sinal de aceleração para um movimento que se inicia e em seguida para. O sinal acima descreve uma aceleração no sentido negativo de Z (para frente), mas em momento algum ocorre a desaceleração no sentido oposto. Desta maneira a velocidade se mantém constante após o fim do movimento. Isso explica o porquê da velocidade nunca retornar a zero mesmo após o controle ficar parado.

Uma hipótese para este problema foi que este dispositivo estaria gerando um sinal de velocidade, ao invés de aceleração. Com isto em mente, foi implementada uma solução alternativa com apenas uma integração feita sobre o sinal do acelerômetro. Entretanto, o resultado obtido também não foi satisfatório, e foi ainda mais impreciso que os resultados obtidos com a técnica de dupla integração combinada com a autocorreção da força normal e autorredução da velocidade. Este resultado pode ser verificado ao utilizar o software desenvolvido no modo 2.

Com a técnica de autorredução da velocidade, a integração dupla sobre o sinal mostrado no **Gráfico 3** apresenta o seguinte comportamento (**Gráfico 4**):

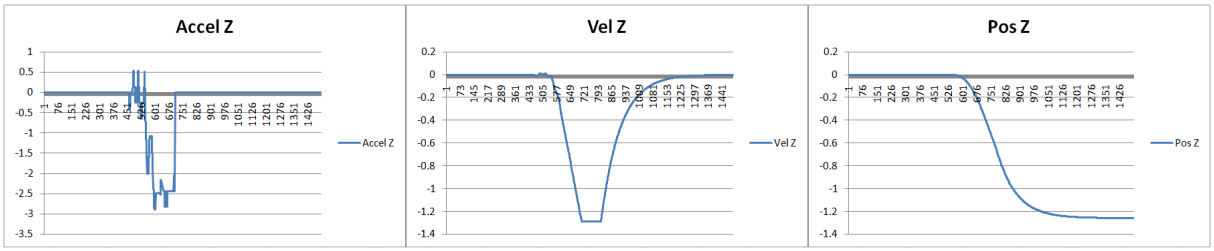


Gráfico 4: Gráficos obtidos para o resultado do movimento exemplificado, quando analisado somente o eixo Z, utilizando a técnica de autocorreção da gravidade (com redução de ruído) e autorredução de velocidade.

A mais provável hipótese para a causa desse problema é que ele ocorre devido às intervenções feitas no sinal pela técnica desenvolvida neste trabalho. Que as técnicas de eliminação de força normal, autocorreção gravitacional e redução de ruído, por não serem refinadas o suficiente, devem estar interferindo de maneira muito significativa no comportamento do sinal. Entretanto, devido à limitação de escopo deste trabalho, a análise mais cuidadosa da intervenção sobre o sinal do acelerômetro do *Wiimote* deve ser realizada em trabalhos futuros.

Também verificou-se que o resultado obtido com esta técnica apresenta um tempo de resposta muito lento, como pode ser conferido no **Gráfico 5** a seguir. Quando o usuário já está por terminar de fazer o movimento (indicado pelas medições do acelerômetro) é que o posicionamento do objeto começa a ser alterado. A razão para este problema é o atraso no início da integração numérica da velocidade com relação à aceleração (uma vez que é necessário esperar, de início, mais de um ciclo para poder obter duas medições de aceleração, para então iniciar o processo de integração), e o mesmo para o deslocamento em relação à velocidade (que, neste caso, é necessário esperar mais de três ciclos para obter-se duas medições de velocidade para usar como entrada para a integração).

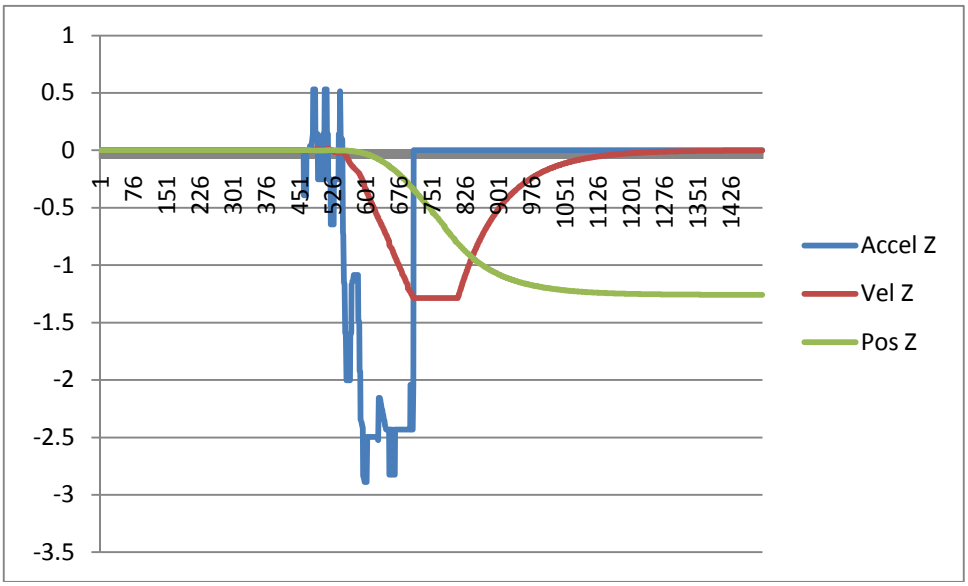


Gráfico 5: Gráfico combinado dos resultados obtidos de aceleração, velocidade e posicionamento. O movimento aplicado para este exemplo é o mesmo descrito anteriormente.

Uma hipótese foi que o dispositivo utilizado estaria com defeito, e por esta razão enviava medições de aceleração inadequadas. Entretanto, após testar a aplicação com dois outros dispositivos diferentes, o resultado obtido foi o mesmo, descartando esta possibilidade.

Em resumo, o sinal produzido pelo acelerômetro do *Wiimote* aparentemente não caracteriza um sinal ideal (ou próximo ao) de aceleração. Além disso, há uma perturbação muito grande em eixos que não fazem parte do movimento realizado. Devido essas causas são necessárias pesquisas mais aprofundadas em trabalhos futuros para permitir o rastreamento em tempo real de todos os movimentos realizados pelo usuário com um *Wiimote* equipado com a extensão Wii Motion Plus™.

Para efeitos de demonstração a implementação da detecção de movimentos com o acelerômetro foi mantida no sistema. O software então foi dividido em três modos de funcionamento: O modo 1, ativado por padrão, é o modo digital que funciona como descrito acima. O modo 2 é o modo de acelerômetro com integração simples, que ilustra a tentativa de interpretar o sinal do acelerômetro como velocidade, fazendo apenas um processo de integração numérica; e o modo 3 é o modo de acelerômetro com integração dupla e técnicas de autocorreção, que ilustra todas as técnicas e tentativas de interpretar o sinal do acelerômetro como um sinal de aceleração, com dupla integração numérica e autorredução da velocidade, como descrito nas seções anteriores.

Para indicar o modo corrente de operação do controle, utiliza-se os dois primeiros LEDs do *Wiimote*. Quando operando em modo 1, o primeiro LED se encontra aceso e o segundo apagado; quando em modo 2, o primeiro LED se encontra apagado e o segundo aceso; e, por fim, quando operando em modo 3, ambos os primeiro e segundo LEDs se encontram acesos. De maneira análoga, quando operando em modo 1, os dois últimos LEDs do *Wiimote* são utilizados para indicar a velocidade atual de operação. Quando na velocidade mais baixa, o terceiro LED se encontra aceso e o quarto apagado; quando em velocidade média, o terceiro LED se encontra apagado e o quarto aceso; e, por fim, quando operando em velocidade alta, ambos os terceiro e quarto LEDs se encontram acesos.

III. CÓDIGO-FONTE DA SOLUÇÃO

1. Biblioteca WiimoteLib

A classe mais relevante da biblioteca **WiimoteLib** para este projeto é a **Wiimote**. A declaração simplificada desta classe está contida no **Código 1** abaixo.

```
////////////////////////////////////////////////////////////////////////////
///////
//Wiimote.cs
//Managed Wiimote Library
//Written by Brian Peek (http://www.brianpeek.com/)
//for MSDN's Coding4Fun (http://msdn.microsoft.com/coding4fun/)
//Visit http://blogs.msdn.com/coding4fun/archive/2007/03/14/1879033.aspx
// and http://www.codeplex.com/WiimoteLib
//for more information
////////////////////////////////////////////////////////////////////////////
///////

using ( ... )

namespace WiimoteLib
{
   public class Wiimote : IDisposable
   {
      public event EventHandler<WiimoteChangedEventArgs> WiimoteChanged;
      public event EventHandler<WiimoteExtensionChangedEventArgs>
WiimoteExtensionChanged;

      public Wiimote()
      ( ... )
      public void Connect()
      public void InitializeMotionPlus()
      public void Disconnect()

      public void SetReportType(InputReport type, bool continuous)
      public void SetReportType(InputReport type, IRSensitivity
irSensitivity, bool continuous)
```

```
        public void SetLEDs(bool led1, bool led2, bool led3, bool led4)
        public void SetLEDs(int leds)

        public void SetRumble(bool on)
        public void GetStatus()

        public byte[] ReadData(int address, short size)
        public void WriteData(int address, byte data)
        public void WriteData(int address, byte size, byte[] data)

        public WiimoteState WiimoteState    { get; }
        public Guid ID    { get; }
        public string HIDDevicePath   { get; }
        public LastReadStatus LastReadStatus { get; private set; }

        #region IDisposable Members
        ( ... )
        #endregion
    }

        [Serializable]
        public class WiimoteNotFoundException : ApplicationException

        [Serializable]
        public class WiimoteException : ApplicationException
}
```

Código 1: Declaração simplificada da classe **Wiimote**. Contém apenas seus métodos públicos.

2. Biblioteca WiimoteInputLib

2.1. Interface **IInputHandler**

```
public interface IInputHandler
{
    KeyboardState KeyboardState { get; }

    MouseState MouseState { get; }
    MouseState PreviousMouseState { get; }

    WiimoteState WiimoteState { get; }
}
```

Código 2: Declaração da Interface **IInputHandler.**

2.2. Classe **InputHandler**

```
public class InputHandler : Microsoft.Xna.Framework.GameComponent,
IInputHandler
{
    // Variáveis membro
    private KeyboardState keyState;
    public KeyboardState KeyboardState { get { return keyState; } }

    private MouseState mouseState;
    public MouseState MouseState { get { return mouseState; } }

    private MouseState prevMouseState;
    public MouseState PreviousMouseState { get { return prevMouseState; } }
```

```
   private WiimoteState wiimoteState;
   public WiimoteState WiimoteState { get { return wiimoteState; } }

   private Wiimote wiimote;

   public InputHandler( Game game ) : base( game )
   {
      game.Services.AddService( typeof( IInputHandler ), this );
      game.IsMouseVisible = true;
   }

   public override void Initialize( )
   {
      // Inicializa o Wiimote
      wiimote = new Wiimote( );
      InitWiimote( );

      wiimoteState = wiimote.WiimoteState;
      mouseState = Mouse.GetState( );

      base.Initialize( );
   }

   public bool InitWiimote( )
   {
      try
      {
         wiimote.Connect( );
         wiimote.SetReportType( InputReport.ExtensionAccel, true );
         wiimote.SetLEDs( true, false, false, false );
         wiimote.InitializeMotionPlus( );
      }
      catch( WiimoteNotFoundException ex )
      {
         MessageBox.Show( ex.Message, "Wiimote not found error",
MessageBoxButtons.OK, MessageBoxIcon.Error );
      }

      return true;
   }

   public override void Update( GameTime gameTime )
   {
      // Pega o estado atual do teclado
      keyState = Keyboard.GetState( );

      // Pega o estado atual do mouse
      prevMouseState = mouseState;
      mouseState = Mouse.GetState( );

      // Pega o estado atual do wiimote conectado
      wiimoteState = wiimote.WiimoteState;

      // Fecha o jogo se a tecla ESC for pressionada
      if( keyState.IsKeyDown( Keys.Escape ) )
      {
         Game.Exit( );
      }

      base.Update( gameTime );
   }
```

```
   public void SetWiimoteLEDs( bool _bLED1, bool _bLED2, bool _bLED3, bool
_bLED4 )
   {
         wiimote.SetLEDs( _bLED1, _bLED2, _bLED3, _bLED4 );
      }
   }
}
```

Código 3: Implementação da classe **InputHandler.**

2.3. Interface **IWiimoteInput**

```
public interface IWiimoteInput
{
   Quaternion AbsoluteOrientation
   { get; }

   Quaternion DeltaOrientation
   { get; }

   Vector3 AbsolutePosition
   { get; }

   Vector3 DeltaPosition
   { get; }

   Vector3 UntransformedDeltaPosition
   { get; }
}
```

Código 4: Declaração da interface **IWiimoteInput.**

2.4. Classe **WiimoteInput**

```
public class WiimoteInput : DrawableGameComponent, IWiimoteInput
{
   enum OperatingMode
   {
      Manual = 0,
      AccelerometerSingleIntegration = 1,
      AccelerometerDoubleIntegration = 2
   }

   #region Constants

   private const float CALIBRATION_TIME = 2.0f; // Tempo de calibração, em
segundos
   private const float ANGULAR_ERROR_THRESHOLD = 0.015f; // Threshold de
corte para o erro de drift do sinal do giroscópio (motion plus)
   private const int MAX_SAMPLES = 10000; // Máximo de amostras para o
processo de calibração
   private const int MONITORING_BUFFER_SIZE = 100; // Como a aplicação roda
a aproximadamente 200fps, dá um tempo de resposta de aproximadamente
0.5sec. Qto maior, mais precisa é a análise e mais tempo demora a resposta
   private const int MOTIONPLUS_DIVISION_RATE = 100; // Descarta os dois
últimos dígitos de informação fornecida pelo motion plus, por ter muito
drift
   private const float ACCELERATION_CORRECTION_THRESHOLD = 0.03f; // Valor
de tolerância para ativar a auto-correção da gravidade
   private const float ACCELERATION_RATE = 9.8f * 1.0f; // Gravity
acceleration * Pixel/meter multiplier
```

```
   private const float VELOCITY_SD_THRESHOLD = 0.0002f; // Valor de
tolerância para o desvio padrão (SD) da velocidade, para verificar se esta
é constante e pode ser reduzida
   private const float VELOCITY_REDUCTION_RATE = 0.9f; // Valor que
multiplica-se à velocidade a cada update para reduzi-la progressivamente
   private const float VELOCITY_ZERO = 0.001f; // Limite inferior para
velocidade não nula
   private const float DIGITAL_MAX_MOVE_RATE = 0.01f; // Taxa maxima
(velocidade) de deslocamento do objeto
   private const float DIGITAL_MED_MOVE_RATE = 0.005f; // Taxa media
(velocidade) de deslocamento do objeto
   private const float DIGITAL_MIN_MOVE_RATE = 0.0025f; // Taxa minima
(velocidade) de deslocamento do objeto

   #endregion // Constants

   #region Control Variables

   // Operating mode of the controller
   OperatingMode m_opMode = OperatingMode.Manual; // Uses manual operating
mode for Default

   // Rotation variables
   Quaternion m_quaRotation = Quaternion.Identity; // Quaternion with
absolute orientation of the wiimote
   Quaternion m_quaDeltaRotation = Quaternion.Identity; // Quaternion with
instantaneous (delta) orientation of the wiimote for the current frame

   // Movement variables
   Vector3 m_v3Position = Vector3.Zero; // Vector with absolute postion
   Vector3 m_v3DeltaPosition = Vector3.Zero; // Vector with instantaneous
(delta) postion for the current frame
   Vector3 m_v3UntransformedDeltaPosition = Vector3.Zero; // Vector with
instantaneous (delta) postion for the current frame, untransformed

   // For accelerometer integration modes
   Quaternion m_quaGravity = Quaternion.Identity; // Quaternion used to
discount the gravity from the acceleration
   Vector3 m_v3Gravity = new Vector3( 0, 1, 0 ); // Vector containing the
current gravity vector affecting the accelerometer
   Vector3 m_v3InitialGravity = new Vector3( 0, 1, 0 ); // Vector
containing the initial gravity vector affecting the accelerometer
   Vector3 m_v3Velocity = Vector3.Zero; // Vector containing the absolute
velocity of the wiimote

   float m_fDigitalMoveRate = DIGITAL_MIN_MOVE_RATE; // Rate of move when
in manual operation mode

   // Controlling the wiimote buttons
   bool m_bOnePressed = false;
   bool m_bTwoPressed = false;

   #endregion // Control Variables

   #region Calculation Variables

   #region Calibration Variables

   // General calibration flags and settings
   bool m_bCalibrating = true;
   bool m_bCalculateCalibrationStats = true;
```

```
    float m_fCalibrationEnd = CALIBRATION_TIME;

    // Motion plus calibration variables
    Vector3[ ] m_cv3CalibrationMotionPlus = new Vector3[ MAX_SAMPLES ];
    int m_iCalibrationIndexMotionPlus = 0;

    // Motion plus statistics variables
    int maiorX = 0;
    int menorX = 99999;
    int maiorDiferencaX = 0;
    int menorDiferencaX = 99999;
    int mediaX = 0;

    int maiorY = 0;
    int menorY = 99999;
    int maiorDiferencaY = 0;
    int menorDiferencaY = 99999;
    int mediaY = 0;

    int maiorZ = 0;
    int menorZ = 99999;
    int maiorDiferencaZ = 0;
    int menorDiferencaZ = 99999;
    int mediaZ = 0;

    // Accelerometer calibration variables
    Vector3[ ] m_cv3CalibrationAccelerometer = new Vector3[ MAX_SAMPLES ];
    int m_iCalibrationIndexAccelerometer = 0;

    // Accelerometer Statistics variables
    float m_fAccelerometerMean = 0.0f;
    float m_fAccelerometerStandardDeviation = 0.0f;

    bool m_bAutoCorrectingGravity = false;
    bool m_bAutoReducingVelocity = false;

    #endregion // Calibration Variables

    #region Integration Variables

    // Motion plus angular velocity integration
    Vector3 m_v3_fA_AngularVelocity = Vector3.Zero;
    Vector3 m_v3_fB_AngularVelocity = Vector3.Zero;

    // Accelerometer double integration
    Vector3 m_v3_fA_Acceleration = Vector3.Zero;
    Vector3 m_v3_fB_Acceleration = Vector3.Zero;

    Vector3 m_v3_fA_Velocity = Vector3.Zero;
    Vector3 m_v3_fB_Velocity = Vector3.Zero;

    #endregion // Integration Variables

    #endregion // Calculation Variables

    #region Movement Monitoring Variables

    // Monitoring array of vectors to keep track of the Wii Motion Plus
angular velocity
    Vector3[ ] m_cv3AngularVelocity = new Vector3[ MONITORING_BUFFER_SIZE ];
    int m_iAngularVelocityVectorIndex = 0;
```

```
   int m_iAngularVelocityVectorCount = 0;

   // Monitoring arrays of vectors to keep track of the accelerometer
acceleration and velocity (calculated)
   Vector3[ ] m_cv3Acceleration = new Vector3[ MONITORING_BUFFER_SIZE ];
   int m_iAccelerationVectorIndex = 0;
   int m_iAccelerationVectorCount = 0;

   Vector3[ ] m_cv3Velocity = new Vector3[ MONITORING_BUFFER_SIZE ];
   int m_iVelocityVectorIndex = 0;
   int m_iVelocityVectorCount = 0;

   #endregion // Movement Monitoring Variables

   #region Information Dump Variables

   bool m_bDumping = false;
   string m_strFilePath = "dump.csv";
   System.IO.StreamWriter m_fileDump;
   int m_nDumpCount = 1;

   #endregion // Information Dump Variables

   #region Necessary components and services

   private ContentManager content;              // Game content manager
   private InputHandler m_cmpInput;      // Input service

   #endregion // Necessary components and services

   #region IWiimoteInput Members

   public Quaternion AbsoluteOrientation
   {
      get { return m_quaRotation; }
   }

   public Quaternion DeltaOrientation
   {
      get { return m_quaDeltaRotation; }
   }

   public Vector3 AbsolutePosition
   {
      get { return m_v3Position; }
   }

   public Vector3 DeltaPosition
   {
      get { return m_v3DeltaPosition; }
   }

   public Vector3 UntransformedDeltaPosition
   {
      get { return m_v3UntransformedDeltaPosition; }
   }

   #endregion

   public WiimoteInput( Game game ) : base( game )
   {
```

```
      // Registra o serviço
      game.Services.AddService( typeof( IWiimoteInput ), this );

      // Obtêm os outros serviços necessários
      content = game.Content;
      m_cmpInput = ( InputHandler )game.Services.GetService( typeof(
IInputHandler ) );
   }

   public override void Initialize( )
   {
      ResetPositionOrientation( );
      ResetDeltas( );
      ResetMonitoringBuffers( );

      SetLeds( );

      base.Initialize( );
   }

   public void ResetPositionOrientation( )
   {
      // Reset position, velocity and orientation to the initial state
      m_v3Position = Vector3.Zero;
      m_v3Velocity = Vector3.Zero;

      m_quaRotation = Quaternion.Identity;
   }

   public void ResetDeltas( )
   {
      m_v3DeltaPosition = Vector3.Zero;
      m_v3UntransformedDeltaPosition = Vector3.Zero;

      m_quaDeltaRotation = Quaternion.Identity;
   }

   private void ResetMonitoringBuffers( )
   {
      m_iAngularVelocityVectorIndex = 0;
      m_iAngularVelocityVectorCount = 0;
      m_iAccelerationVectorIndex = 0;
      m_iAccelerationVectorCount = 0;
      m_iVelocityVectorIndex = 0;
      m_iVelocityVectorCount = 0;
   }

   private void SetLeds( )
   {
      bool bLED1 = false;
      bool bLED2 = false;
      bool bLED3 = false;
      bool bLED4 = false;

      if( m_opMode == OperatingMode.Manual )
      {
         bLED1 = true;
         bLED2 = false;
      }
      else if( m_opMode == OperatingMode.AccelerometerSingleIntegration )
      {
```

```
            bLED1 = false;
            bLED2 = true;
        }
        else // m_opMode == OperatingMode.AccelerometerDoubleIntegration
        {
            bLED1 = true;
            bLED2 = true;
        }

        if( m_fDigitalMoveRate == DIGITAL_MIN_MOVE_RATE )
        {
            bLED3 = true;
            bLED4 = false;
        }
        else if( m_fDigitalMoveRate == DIGITAL_MED_MOVE_RATE )
        {
            bLED3 = false;
            bLED4 = true;
        }
        else // m_fDigitalMoveRate == DIGITAL_MAX_MOVE_RATE
        {
            bLED3 = true;
            bLED4 = true;
        }

        m_cmpInput.SetWiimoteLEDs( bLED1, bLED2, bLED3, bLED4 );
    }

    private void CalibrateWiimote( )
    {
        ResetPositionOrientation( );
        ResetDeltas( );
        ResetMonitoringBuffers( );

        // Reset calibration statistics variables
        maiorX = 0;
        menorX = 99999;
        maiorDiferencaX = 0;
        menorDiferencaX = 99999;
        mediaX = 0;

        maiorY = 0;
        menorY = 99999;
        maiorDiferencaY = 0;
        menorDiferencaY = 99999;
        mediaY = 0;

        maiorZ = 0;
        menorZ = 99999;
        maiorDiferencaZ = 0;
        menorDiferencaZ = 99999;
        mediaZ = 0;

        m_fAccelerometerMean = 0.0f;
        m_fAccelerometerStandardDeviation = 0.0f;

        // Reset the sampling buffers
        m_iCalibrationIndexMotionPlus = 0;
        m_iCalibrationIndexAccelerometer = 0;
```

```
      // Set the flags for the start of the sampling process and further
statistics calculations over them
      m_bCalibrating = true;
      m_bCalculateCalibrationStats = true;
   }

   private void CalculateCalibrationStats( )
   {
      // Calcula estatísticas de calibração do Motion Plus
      for( int i = 0; i < m_iCalibrationIndexMotionPlus; i++ )
      {
         // X
         if( m_cv3CalibrationMotionPlus[ i ].X > maiorX )
            maiorX = ( int )m_cv3CalibrationMotionPlus[ i ].X;

         if( m_cv3CalibrationMotionPlus[ i ].X < menorX )
            menorX = ( int )m_cv3CalibrationMotionPlus[ i ].X;

         mediaX += ( int )m_cv3CalibrationMotionPlus[ i ].X;

         // Y
         if( m_cv3CalibrationMotionPlus[ i ].Y > maiorY )
            maiorY = ( int )m_cv3CalibrationMotionPlus[ i ].Y;

         if( m_cv3CalibrationMotionPlus[ i ].Y < menorY )
            menorY = ( int )m_cv3CalibrationMotionPlus[ i ].Y;

         mediaY += ( int )m_cv3CalibrationMotionPlus[ i ].Y;

         // Z
         if( m_cv3CalibrationMotionPlus[ i ].Z > maiorZ )
            maiorZ = ( int )m_cv3CalibrationMotionPlus[ i ].Z;

         if( m_cv3CalibrationMotionPlus[ i ].Z < menorZ )
            menorZ = ( int )m_cv3CalibrationMotionPlus[ i ].Z;

         mediaZ += ( int )m_cv3CalibrationMotionPlus[ i ].Z;
      }

      mediaX /= m_iCalibrationIndexMotionPlus;
      mediaY /= m_iCalibrationIndexMotionPlus;
      mediaZ /= m_iCalibrationIndexMotionPlus;

      // Calcula estatísticas de calibração do acelerômetro

      // Média (controle parado)
      for( int i = 0; i < m_iCalibrationIndexAccelerometer; i++ )
      {
         m_fAccelerometerMean += m_cv3CalibrationAccelerometer[ i ].Length(
);
      }

      m_fAccelerometerMean /= m_iCalibrationIndexAccelerometer;

      // Desvio padrão
      for( int i = 0; i < m_iCalibrationIndexAccelerometer; i++ )
      {
         m_fAccelerometerStandardDeviation += ( (
m_cv3CalibrationAccelerometer[ i ].Length( ) - m_fAccelerometerMean ) * (
m_cv3CalibrationAccelerometer[ i ].Length( ) - m_fAccelerometerMean ) );
      }
```

```
      m_fAccelerometerStandardDeviation = ( float )Math.Sqrt( ( double )(
m_fAccelerometerStandardDeviation / ( m_iCalibrationIndexAccelerometer - 1
) ) );


      // Fim
      m_bCalculateCalibrationStats = false;
   }

   protected override void LoadContent( )
   {
      base.LoadContent( );
   }

   public override void Update( GameTime gameTime )
   {
      #region Interpretation of pressed keys

      if( m_cmpInput.KeyboardState.IsKeyDown( Keys.Space ) ) // Space Key:
Re-Calibrate the Wiimote
      {
         m_fCalibrationEnd = CALIBRATION_TIME + ( float
)gameTime.TotalGameTime.TotalSeconds;
         CalibrateWiimote( );
      }

      if( m_cmpInput.KeyboardState.IsKeyDown( Keys.R ) ) // R Key: Reset
the positioning
      {
         ResetPositionOrientation( );
         ResetDeltas( );
         ResetMonitoringBuffers( );
      }

      if( !m_bOnePressed && m_cmpInput.WiimoteState.WiimoteButtonState.One)
      {
         m_opMode = ( OperatingMode )( ( ( int )m_opMode + 1 ) % 3 );
         SetLeds( );
      }
      m_bOnePressed = m_cmpInput.WiimoteState.WiimoteButtonState.One;

      if( !m_bTwoPressed && m_cmpInput.WiimoteState.WiimoteButtonState.Two)
      {
         if( m_fDigitalMoveRate == DIGITAL_MIN_MOVE_RATE )
         {
            m_fDigitalMoveRate = DIGITAL_MED_MOVE_RATE;
         }
         else if( m_fDigitalMoveRate == DIGITAL_MED_MOVE_RATE )
         {
            m_fDigitalMoveRate = DIGITAL_MAX_MOVE_RATE;
         }
         else // m_fDigitalMoveRate == DIGITAL_MAX_MOVE_RATE
         {
            m_fDigitalMoveRate = DIGITAL_MIN_MOVE_RATE;
         }
         SetLeds( );
      }
      m_bTwoPressed = m_cmpInput.WiimoteState.WiimoteButtonState.Two;

      #endregion //Interpretation of pressed keys
```

```
      #region Calibration of the Wiimote

      if( m_bCalibrating )
      {
         if( ( float )gameTime.TotalGameTime.TotalSeconds >=
m_fCalibrationEnd )
         {
            m_bCalibrating = false;
         }
         else
         {
            m_cv3CalibrationMotionPlus[ m_iCalibrationIndexMotionPlus++ ] =
new Vector3(
   m_cmpInput.WiimoteState.MotionPlusState.RawValues.X /
MOTIONPLUS_DIVISION_RATE,
   m_cmpInput.WiimoteState.MotionPlusState.RawValues.Y /
MOTIONPLUS_DIVISION_RATE,
   m_cmpInput.WiimoteState.MotionPlusState.RawValues.Z /
MOTIONPLUS_DIVISION_RATE );

            m_cv3CalibrationAccelerometer[
m_iCalibrationIndexAccelerometer++ ] = new Vector3(
               -m_cmpInput.WiimoteState.AccelState.Values.X,
                m_cmpInput.WiimoteState.AccelState.Values.Z,
                m_cmpInput.WiimoteState.AccelState.Values.Y );
         }
      }
      else
      {
         if( m_bCalculateCalibrationStats )
         {
            CalculateCalibrationStats( );
         }

         #endregion //Calibration of the Wiimote

         if( m_cmpInput.WiimoteState.WiimoteButtonState.B ) // Holding
Wiimote B Button: track the wiimote movement change
         {
            // Get current motion plus state (angular velocity), and
convert to the XNA coordinate system (inverting the Z axis)
            m_v3_fB_AngularVelocity = new Vector3( (
m_cmpInput.WiimoteState.MotionPlusState.RawValues.X /
MOTIONPLUS_DIVISION_RATE ) - mediaX,
               ( m_cmpInput.WiimoteState.MotionPlusState.RawValues.Y /
MOTIONPLUS_DIVISION_RATE ) - mediaY,
              -( ( m_cmpInput.WiimoteState.MotionPlusState.RawValues.Z /
MOTIONPLUS_DIVISION_RATE ) - mediaZ ) );

            // Store current motion plus state into the angular velocity
monitoring buffer and update its index and counter
            m_cv3AngularVelocity[ m_iAngularVelocityVectorIndex ] =
m_v3_fB_AngularVelocity;

            m_iAngularVelocityVectorIndex = ( m_iAngularVelocityVectorIndex
+ 1 ) % MONITORING_BUFFER_SIZE; // Circular Buffer

            if( m_iAngularVelocityVectorCount < MONITORING_BUFFER_SIZE )
               m_iAngularVelocityVectorCount++;
```

```
            // If more than one term has already been aquired, we can start
integrating
            if( m_iAngularVelocityVectorCount > 1 )
            {
               // Numerical integration of the Motion Plus angular velocity
vector (trapezoidal rule, subject to cumulative error)

               // These multipliers are the convertion rate from motion
plus raw data to degree angles
               float fYawMultiplier = 5.0f;
               float fRollMultiplier = 5.0f;
               float fPitchMultiplier = 5.0f;

               // Motion plus particularity: if *Angle*Fast is enabled, the
convertion rate is 5x greater.
               if( m_cmpInput.WiimoteState.MotionPlusState.PitchFast )
                  fPitchMultiplier = 25.0f;

               if( m_cmpInput.WiimoteState.MotionPlusState.YawFast )
                  fYawMultiplier = 25.0f;

               if( m_cmpInput.WiimoteState.MotionPlusState.RollFast )
                  fRollMultiplier = 25.0f;

              // This is the noise cut threshold
              float fYawError = ANGULAR_ERROR_THRESHOLD * fYawMultiplier;
              float fRollError = ANGULAR_ERROR_THRESHOLD * fRollMultiplier;
              float fPitchError = ANGULAR_ERROR_THRESHOLD *
fPitchMultiplier;

               // Integrate the terms in the three axis
               float fDeltaYaw = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( ( float )(
m_v3_fB_AngularVelocity.X + m_v3_fA_AngularVelocity.X ) / 2.0f ) *
fYawMultiplier );
               if(( fDeltaYaw <= fYawError ) && ( fDeltaYaw >= -fYawError))
                  fDeltaYaw = 0;

               float fDeltaRoll = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( ( float )(
m_v3_fB_AngularVelocity.Y + m_v3_fA_AngularVelocity.Y ) / 2.0f ) *
fRollMultiplier );
               if( ( fDeltaRoll <= fRollError ) && ( fDeltaRoll >= -
fRollError ) )
                  fDeltaRoll = 0;

               float fDeltaPitch = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( ( float )(
m_v3_fB_AngularVelocity.Z + m_v3_fA_AngularVelocity.Z ) / 2.0f ) *
fPitchMultiplier );
               if( ( fDeltaPitch <= fPitchError ) && ( fDeltaPitch >= -
fPitchError ) )
                  fDeltaPitch = 0;

               // Create the delta rotation quaternion with the angles
moved this frame, and then multiplies it with the absolute rotation
quaternion (because this measure is the rotation around itself)
               m_quaDeltaRotation = Quaternion.CreateFromAxisAngle(
Vector3.Right, MathHelper.ToRadians( fDeltaPitch ) ) *
Quaternion.CreateFromAxisAngle( Vector3.Up, MathHelper.ToRadians( fDeltaYaw
```

```
) ) * Quaternion.CreateFromAxisAngle( Vector3.Backward,
MathHelper.ToRadians( fDeltaRoll ) );
                m_quaRotation *= m_quaDeltaRotation;


                #region Accelerometer Single and Double Integration mode
calculations

                // Continue with the integration processes if not operating
in the manual mode
                if( m_opMode != OperatingMode.Manual )
                {
                    // Estimates the gravity vector affecting the
accelerometer
                    m_quaGravity *= Quaternion.Inverse( m_quaDeltaRotation );
                    m_v3Gravity = Vector3.Transform( m_v3InitialGravity,
m_quaGravity );

                    // Get the current accelerometer state and converts it to
the XNA coordinate system
                    m_v3_fB_Acceleration = new Vector3(
                        -m_cmpInput.WiimoteState.AccelState.Values.X,
                         m_cmpInput.WiimoteState.AccelState.Values.Z,
                        -m_cmpInput.WiimoteState.AccelState.Values.Y );

                    // Store current accelerometer state into the
acceleration monitoring buffer
                    m_cv3Acceleration[ m_iAccelerationVectorIndex ] =
m_v3_fB_Acceleration;

                    #region Real-time auto-correction of the gravity vector

                    m_bAutoCorrectingGravity = false; // For debug purposes
only
                    if(m_iAccelerationVectorCount == MONITORING_BUFFER_SIZE )
                    {
                        // Calculate current state statistics for auto-
correction
                        float fCurrentAccelerometerMean = 0.0f;
                        float fCurrentAccelerometerStandardDeviation = 0.0f;

                        // Current mean
                        for( int i = 0; i < m_iAccelerationVectorCount; i++ )
// The order doesn't matter because we're calculating mean and standard
deviation
                        {
                            fCurrentAccelerometerMean += m_cv3Acceleration[ i
].Length( );
                        }

                        fCurrentAccelerometerMean /=
m_iAccelerationVectorCount;

                        // Current standard deviation
                        for( int i = 0; i < m_iAccelerationVectorCount; i++ )
                        {
                            fCurrentAccelerometerStandardDeviation += ( (
m_cv3Acceleration[ i ].Length( ) - fCurrentAccelerometerMean ) * (
m_cv3Acceleration[ i ].Length( ) - fCurrentAccelerometerMean ) );
                        }
```

```
                    fCurrentAccelerometerStandardDeviation = ( float
)Math.Sqrt( ( double )( fCurrentAccelerometerStandardDeviation / (
m_iAccelerationVectorCount - 1 ) ) );


                    // Auto-correction analysis

                    // Compares the current mean with the calibrated one,
to check if the controller seems to be still
                    if( ( fCurrentAccelerometerMean < (
m_fAccelerometerMean + ACCELERATION_CORRECTION_THRESHOLD ) ) && (
fCurrentAccelerometerMean > ( m_fAccelerometerMean -
ACCELERATION_CORRECTION_THRESHOLD ) ) )
                    {
                       // Verify if the current standard deviation is
small enough to consider the controller still
                       if( fCurrentAccelerometerStandardDeviation <
ACCELERATION_CORRECTION_THRESHOLD )
                       {
                          // Apply the auto-correction
                          m_v3InitialGravity = m_v3_fB_Acceleration;
                          m_v3Gravity = m_v3_fB_Acceleration;
                          m_quaGravity = Quaternion.Identity;

                          m_bAutoCorrectingGravity = true;
                       }
                    }
                 }

                 #endregion // Real-time auto-correction of the gravity
vector

                 m_v3_fB_Acceleration -= m_v3Gravity; // Subtracts the
gravity vector estimate from the accelerometer signal

                 m_v3_fB_Acceleration *= ACCELERATION_RATE; // Converts
the accelerometer signal from Gs/sec2 to Pixels/sec2

                 // Update the acceleration monitoring buffer index and
counter
                 m_iAccelerationVectorIndex = ( m_iAccelerationVectorIndex
+ 1 ) % MONITORING_BUFFER_SIZE; // Circular Buffer

                 if( m_iAccelerationVectorCount < MONITORING_BUFFER_SIZE )
                    m_iAccelerationVectorCount++;

                 // If more than one term has already been aquired, we can
start integrating
                 if( m_iAccelerationVectorCount > 1 )
                 {
                    if( m_opMode ==
OperatingMode.AccelerometerSingleIntegration )
                    {
                       // In the accelerometer single integration mode, we
integrate the acceleration directly into the displacement
                       // (treating it as it was a velocity signal) and
sum it to the absolute position of the object
                       // This mode don't work properly because the signal
obtained from the accelerometer, even after filters and corrections,
                       // don't caracterize a velocity signal (curve).
```

```
                        // This is just for test and verification purposes.
Try dumping this info with the wiimote Home button, and then analyzing it
with Excel graphics.

                        // Integrate the acceleration just once to obtain
the displacement
                        float fDisplacementX = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )(
m_v3_fB_Acceleration.X + m_v3_fA_Acceleration.X ) / 2.0f );
                        float fDisplacementY = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )(
m_v3_fB_Acceleration.Y + m_v3_fA_Acceleration.Y ) / 2.0f );
                        float fDisplacementZ = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )(
m_v3_fB_Acceleration.Z + m_v3_fA_Acceleration.Z ) / 2.0f );

                        m_v3UntransformedDeltaPosition = new Vector3(
fDisplacementX, fDisplacementY, fDisplacementZ );

                        // Now transforms the displacement obtained
acording to the absolute orientation of the wiimote
                        m_v3DeltaPosition = Vector3.Transform(
m_v3UntransformedDeltaPosition, m_quaRotation );

                        // Adds the displacement to the absolute position
                        m_v3Position += m_v3DeltaPosition;
                    }
                    else // m_opMode ==
OperatingMode.AccelerometerDoubleIntegration
                    {
                        // In the accelerometer double integration mode, we
integrate the acceleration first to obtain the delta velocity, sum it to
the absolute velocity,
                        // then integrate it again to obtain the
displacement, and sum it to the absolute position of the object.
                        // This was the initial idea, but didn't work
properly because the signal obtained from the accelerometer, even after
filters and corrections,
                        // didn't caracterize an acceleration signal
(curve).
                        // This is just for test and verification purposes.
Try dumping this info with the wiimote Home button, and then analyzing it
with Excel graphics.

                        // Integrates acceleration
                        float fVelocityX = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )(
m_v3_fB_Acceleration.X + m_v3_fA_Acceleration.X ) / 2.0f );
                        float fVelocityY = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )(
m_v3_fB_Acceleration.Y + m_v3_fA_Acceleration.Y ) / 2.0f );
                        float fVelocityZ = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )(
m_v3_fB_Acceleration.Z + m_v3_fA_Acceleration.Z ) / 2.0f );

                        // Sum the delta velocity obtained to the absolute
velocity
                        Vector3 v3DeltaVelocity = new Vector3( fVelocityX,
fVelocityY, fVelocityZ );
                        m_v3Velocity += v3DeltaVelocity;
```

```
                        #region Real-time velocity auto-reduction

                        // Velocity reduction based in statistical analysis
of current velocity samples. Reduces velocity to zero when it's constant
                        m_bAutoReducingVelocity = false; // For debug
purposes only
                        if(m_iVelocityVectorCount== MONITORING_BUFFER_SIZE)
                        {
                           // Calculate current statistics
                           float fCurrentVelocityMean = 0.0f;
                           float fCurrentVelocityStandardDeviation = 0.0f;

                           // Current mean
                           for( int i = 0; i < m_iVelocityVectorCount; i++
) // The order doesn't matter because we're calculating mean and standard
deviation
                           {
                              fCurrentVelocityMean += m_cv3Velocity[ i
].Length( );
                           }

                           fCurrentVelocityMean /= m_iVelocityVectorCount;

                           // Current standard deviation
                           for(int i = 0; i < m_iVelocityVectorCount; i++ )
                           {
                              fCurrentVelocityStandardDeviation += ( (
m_cv3Velocity[ i ].Length( ) - fCurrentVelocityMean ) * ( m_cv3Velocity[ i
].Length( ) - fCurrentVelocityMean ) );
                           }

                           fCurrentVelocityStandardDeviation = ( float
)Math.Sqrt( ( double )( fCurrentVelocityStandardDeviation / (
m_iVelocityVectorCount - 1 ) ) );


                           // Análise da auto-redução

                           // If the current standard deviation is under
the accepted threshold, we consider it constant
                           if( fCurrentVelocityStandardDeviation <
VELOCITY_SD_THRESHOLD )
                           {
                              // Apply the velocity auto-reduction
                              m_v3Velocity *= VELOCITY_REDUCTION_RATE;
                              if( m_v3Velocity.Length( ) < VELOCITY_ZERO )
                              {
                                 m_v3Velocity = Vector3.Zero;
                              }

                              m_bAutoReducingVelocity = true;
                           }
                        }
                        #endregion

                        //Now integrates the velocity to obtain the
position
                        m_v3_fB_Velocity = m_v3Velocity;

                        m_cv3Velocity[ m_iVelocityVectorIndex ] =
v3DeltaVelocity; // Adds only the delta to the monitoring vector because
```

```
the auto-reduction would disturb itself if the absolute velocity was to be
considered

                        // Updates the velocity monitoring vector index and
counter
                        m_iVelocityVectorIndex = ( m_iVelocityVectorIndex +
1 ) % MONITORING_BUFFER_SIZE; // Circular Buffer

                        if(m_iVelocityVectorCount < MONITORING_BUFFER_SIZE)
                           m_iVelocityVectorCount++;

                        // If more than one term has already been aquired,
we can start integrating
                        if( m_iVelocityVectorCount > 1 )
                        {
                           // Integrate the velocity to obtain the
displacement
                           float fDisplacementX = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )( m_v3_fB_Velocity.X +
m_v3_fA_Velocity.X ) / 2.0f );
                           float fDisplacementY = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )( m_v3_fB_Velocity.Y +
m_v3_fA_Velocity.Y ) / 2.0f );
                           float fDisplacementZ = ( float
)gameTime.ElapsedGameTime.TotalSeconds * ( ( float )( m_v3_fB_Velocity.Z +
m_v3_fA_Velocity.Z ) / 2.0f );

                           m_v3UntransformedDeltaPosition = new Vector3(
fDisplacementX, fDisplacementY, fDisplacementZ );

                           // Now transforms the displacement obtained
acording to the absolute orientation of the wiimote
                           m_v3DeltaPosition = Vector3.Transform(
m_v3UntransformedDeltaPosition, m_quaRotation );

                           //Adds the displacement to the absolute position
                           m_v3Position += m_v3DeltaPosition;
                        }

                        //Updates the variable for the next integration
step
                        m_v3_fA_Velocity = m_v3_fB_Velocity;
                     }
                  }

                  // Updates the variable for the next integration step
                  m_v3_fA_Acceleration = m_v3_fB_Acceleration;
               }

               #endregion // Accelerometer Single and Double Integration
mode calculations
            }

            // Updates the variable for the next integration step
            m_v3_fA_AngularVelocity = m_v3_fB_AngularVelocity;

            #region Wiimote Information Dump

            if( m_cmpInput.WiimoteState.WiimoteButtonState.Home )
            {
               if( !m_bDumping )
```

```
                {
                    while( System.IO.File.Exists( m_strFilePath ) )
                    {
                        m_strFilePath = "dump" + m_nDumpCount++.ToString( ) +
".csv";
                    }

                    m_fileDump = System.IO.File.CreateText( m_strFilePath );

                    m_fileDump.WriteLine( "Accelerometer X,Accelerometer
Y,Accelerometer Z,Velocity X,Velocity Y,Velocity Z,Position X,Position
Y,Position Z,,MotionPlus X,MotionPlus Y,MotionPlus Z," );
                    m_bDumping = true;
                }

                m_fileDump.Write( "{0},", m_v3_fB_Acceleration.X );
                m_fileDump.Write( "{0},", m_v3_fB_Acceleration.Y );
                m_fileDump.Write( "{0},", m_v3_fB_Acceleration.Z );
                m_fileDump.Write( "{0},", m_v3_fB_Velocity.X );
                m_fileDump.Write( "{0},", m_v3_fB_Velocity.Y );
                m_fileDump.Write( "{0},", m_v3_fB_Velocity.Z );
                m_fileDump.Write( "{0},", m_v3Position.X );
                m_fileDump.Write( "{0},", m_v3Position.Y );
                m_fileDump.Write( "{0},,", m_v3Position.Z );
                m_fileDump.Write( "{0},", m_v3_fB_AngularVelocity.X );
                m_fileDump.Write( "{0},", m_v3_fB_AngularVelocity.Y );
                m_fileDump.WriteLine( "{0},", m_v3_fB_AngularVelocity.Z );
            }
            else
            {
                if( m_bDumping )
                {
                    m_bDumping = false;
                    m_fileDump = null;
                }
            }

            #endregion // Wiimote Information Dump
        }
        else
        {
            // When B button is not pressed, the absolute orientation
remains static and the control variables are reset
            ResetDeltas( );
            ResetMonitoringBuffers( );

            #region Reset control variables for the non-Manual modes of
operation

            m_v3InitialGravity = new Vector3(
                -m_cmpInput.WiimoteState.AccelState.Values.X,
                 m_cmpInput.WiimoteState.AccelState.Values.Z,
                -m_cmpInput.WiimoteState.AccelState.Values.Y );

            m_quaGravity = Quaternion.Identity;
            m_v3Velocity = Vector3.Zero;

            m_bAutoCorrectingGravity = false;
            m_bAutoReducingVelocity = false;
```

```
            #endregion // Reset control variables for the non-Manual modes
of operation
         }

         // When operating in manual mode, the position of the object is
altered by pressing the wiimote's digital directional
         if( m_opMode == OperatingMode.Manual )
         {
            Vector3 v3Displacement = new Vector3( 0.0f, 0.0f, 0.0f );

            if( m_cmpInput.WiimoteState.WiimoteButtonState.Up )
            {
               v3Displacement.Z -= m_fDigitalMoveRate;
            }

            if( m_cmpInput.WiimoteState.WiimoteButtonState.Down )
            {
               v3Displacement.Z += m_fDigitalMoveRate;
            }

            if( m_cmpInput.WiimoteState.WiimoteButtonState.Left )
            {
               v3Displacement.X -= m_fDigitalMoveRate;
            }

            if( m_cmpInput.WiimoteState.WiimoteButtonState.Right )
            {
               v3Displacement.X += m_fDigitalMoveRate;
            }

            if( m_cmpInput.WiimoteState.WiimoteButtonState.Minus )
            {
               v3Displacement.Y -= m_fDigitalMoveRate;
            }

            if( m_cmpInput.WiimoteState.WiimoteButtonState.Plus )
            {
               v3Displacement.Y += m_fDigitalMoveRate;
            }

            m_v3UntransformedDeltaPosition = v3Displacement;

            // Transforms the displacement acording to the current absolute
orientation of the wiimote and adds it to the absolute position
            m_v3DeltaPosition = Vector3.Transform(
m_v3UntransformedDeltaPosition, m_quaRotation );
            m_v3Position += m_v3DeltaPosition;
         }
      }

      base.Update( gameTime );
   }

   public override void Draw( GameTime gameTime )
   {
      base.Draw( gameTime );
   }
}
```

Código 5: Implementação da classe **WiimoteInput**.

2.5. Classe **InverseKinematics**

```
public class InverseKinematics
{
   private InverseKinematics( )
   {
      // Private constructor so this class can't be instantiated. Supposed
to be a static class.
   }

   /// <summary>
   /// This is the primary function for updating inverse kinematics. Here,
   /// we implement the Cyclic Coordinate Decsent algorithm. In a nutshell
   /// this is what we are trying to do: Given a goal position, end
   /// effector (often the end bone in a chain), and a current bone, we
   /// want to rotate the current bone such that it will bring the end
   /// effector closer to the goal position.
   /// We do this iteratively for every bone in the chain.
   ///
   /// Here's a very basic idea of how the algorithm works:
   /// 1) Compute vector directions for the bone to
   ///    the goal and the bone to end effector.
   /// 2) Compute a matrix that rotates the end effector vector onto
   ///    the goal vector.
   /// 3) Rotate the current bone by this rotation matrix.
   /// 4) Repeat steps 1-3 for each bone in the chain.
   /// </summary>
   /// <param name="curBone">The index into the bone chain for the current
   /// bone to update.</param>
   /// <param name="endEffector">The index of the end effector in bone
   /// chain. The end effector is the end bone that we want to move toward
   /// the cat.</param>
   /// <param name="goal">The world position of the object the IK chain is
   /// trying to move to.</param>
   /// <param name="rootWorldTransform">The world transform for the root of
   /// the bone chain</param>
   /// <param name="bindPose">The bind pose of the IK objects or the
   /// default position and orientation of the bones in the IK
   /// chain</param>
   /// <param name="transforms">The list of rotational offsets from
   /// the bind pose</param>
   /// <param name="parentBones">The list of parent bones for each
   /// bone index.</param>
   static public Matrix UpdateBone( ref Matrix[ ] transforms, int curBone,
                  int endEffector, Vector3 goal, Matrix[ ] worldTransforms,
                  Matrix[ ] localTransforms )
   {
      // RETIRADO E ADAPTADO DE
      // http://creators.xna.com/en-US/sample/inversekinematics
      // Implementação do algorítmo Cyclic Coordinate Descent (CCD) para
      // Cinemática Inversa

      // We first compute the vector directions for the current bone to the
      // goal and the current bone to end effector. We will do all this in
      // the current bone's local coordinates which makes it easier to
      // generate our final rotation matrix for the bone.

      // Get the world transform of the current bone
      Matrix curBoneWorld = worldTransforms[ curBone ];

      // Transform the goal into coordinate system of current bone. To do
```

```
      // this, we transform the goal position by the inverse world
      // transform of the current bone.
      Vector3 goalInBoneSpace = Vector3.Transform( goal,
                                   Matrix.Invert( curBoneWorld ) );

      // Transform the end effector into coordinate system of the current
      // bone. To do this, we first compute the current world transform of
      // the end effector then we multiply it by the Inverse of the current
      // bone's world transform. We then store the position.
      Matrix endEffectorWorld = worldTransforms[ endEffector ];
      Vector3 endEffectorInBoneSpace = Matrix.Multiply( endEffectorWorld,
                               Matrix.Invert( curBoneWorld ) ).Translation;

      // After we normalize, we will have unit vectors that represent the
      // direction to the end effector and the goal in the local coordinate
      // space of the current bone.
      if( endEffectorInBoneSpace != Vector3.Zero )
         endEffectorInBoneSpace.Normalize( );

      if( goalInBoneSpace != Vector3.Zero )
         goalInBoneSpace.Normalize( );

      // Next we build the rotatian matrix that rotates the end effector
      // onto the goal and apply that rotation to the current bone.

      // Compute axis of rotation: the cross product of the two vectors.
      Vector3 axis = Vector3.Cross(endEffectorInBoneSpace,goalInBoneSpace);

      // Use TransformNormal to orient the axis by the local coordinate
      // transform of the current bone
      axis = Vector3.TransformNormal( axis, localTransforms[ curBone ] );
      if( axis != Vector3.Zero )
         axis.Normalize( );

      // Compute the angle we will be rotating by which is just the angle
      // between the vectors
      float dot = Vector3.Dot( goalInBoneSpace, endEffectorInBoneSpace );

      //Clamp to -1 and 1 to avoid any possible floating point precision
      //errors
      dot = MathHelper.Clamp( dot, -1, 1 );

      //Compute the angle.
      float angle = ( float )Math.Acos( dot );
      angle = MathHelper.WrapAngle( angle );

      // Create the rotation matrix
      Matrix rotation = Matrix.CreateFromAxisAngle( axis, angle );

      // Rotate the current bone by the new rotation matrix
      transforms[ curBone ] *= rotation;

      return rotation;
   }

   /// <summary>
   /// Updates the list of world transforms for a bone hierarchy. The
   /// hierarchy must be sorted by bone depth where the parent bone is at
   /// the head of the list
   /// </summary>
   /// <param name="worldTransforms">The list of world transforms to
```

```
   /// update</param>
   /// <param name="rootWorldTransform">The root world transform of
   /// the root bone</param>
   /// <param name="bindPose">The defaulft pose of the bone
   /// hierarchy</param>
   /// <param name="animationTransforms">The transform offsets from the
   /// bind pose</param>
   /// <param name="parentBones">The list of parent bones for each bone
   /// index.</param>
   static public void UpdateTransforms( ref Matrix[ ] worldTransforms,
                  ref Matrix[ ] localTransforms, Matrix rootWorldTransform,
                  Matrix[ ] bindPose, Matrix[ ] animationTransforms,
                  int[ ] parentBones )
   {
      // RETIRADO E ADAPTADO DE
      // http://creators.xna.com/en-US/sample/inversekinematics
      // Implementação do algorítmo Cyclic Coordinate Descent (CCD) para
      // Cinemática Inversa

      //Set the parent bone transform
      localTransforms[ 0 ] = Matrix.Multiply( animationTransforms[ 0 ],
bindPose[ 0 ] );
      worldTransforms[ 0 ] = Matrix.Multiply( localTransforms[ 0 ],
                                              rootWorldTransform );

      // Loop all of the bones.
      // Since the bone hierarchy is sorted by depth
      // we will transform the parent before any child.
      for( int curBone = 1; curBone < worldTransforms.Length; curBone++ )
      {
         //calculate the local transform of the bone
         Matrix local = Matrix.Multiply( animationTransforms[ curBone ],
                                                 bindPose[ curBone ] );

         // Find the transform of this bones parent.
         // If this is the first bone use the world matrix used on the
avatar
         int iParentBoneIndex = parentBones[ curBone ];
         Matrix parentMatrix;
         if( iParentBoneIndex != -1 )
            parentMatrix = worldTransforms[ iParentBoneIndex ];
         else
            parentMatrix = Matrix.Identity;

         // Calculate this bones world space position
         localTransforms[ curBone ] = local;
         worldTransforms[ curBone ] = Matrix.Multiply(local, parentMatrix);
      }
   }
}
```

Código 6: Implementação da classe **InverseKinematics.**

2.6. Interface **ICamera**

```
public interface ICamera
{
   Matrix Projection
   { get; }

   Matrix View
```

```
   { get; }
}
```

Código 7: Declaração da interface **ICamera**.

2.7. Classe **Camera**

```
public class Camera : Microsoft.Xna.Framework.GameComponent, ICamera
{
   // Constantes
   private const float spinRate = 120.0f;
   private const float moveRate = 40.0f;

   // Variáveis membro
   protected IInputHandler input; // Serviço de entrada de dados
   private GraphicsDeviceManager graphics; // Classe de acesso ao
dispositivo gráfico

   // Matrizes da câmera
   private Matrix projection; // Matriz de projeção (perspectiva)
   public Matrix Projection
   {
      get { return projection; }
   }

   private Matrix view; // Matriz de visão (posição e orientação da câmera)
   public Matrix View
   {
      get { return view; }
   }

   // Vetores da câmera
   private Vector3 cameraPosition = new Vector3( 0.0f, 10.0f, 15.0f ); //
Posição da câmera
   private Vector3 cameraTarget = Vector3.Zero; // Ponto para o qual a
câmera está olhando
   private Vector3 cameraUpVector = Vector3.Up; // Vetor que indica qual é
o sentido "para cima" da câmera (orientação)

   // Posicao padrao da camera
   private Vector3 cameraReference = new Vector3( 0.0f, -0.4f, -1.0f );

   // Vetor de movimentação da câmera
   protected Vector3 movement = Vector3.Zero;

   // Ângulos de rotação da câmera
   private float cameraYaw = 0.0f; // Yaw = rotação em torno do eixo Y
   private float cameraPitch = 0.0f; // Pitch = rotação em torno do eixo X

   public Camera( Game game ) : base( game )
   {
      // Registra o serviço de câmera
      game.Services.AddService( typeof( ICamera ), this );

      //Obtêm os serviços necessários
      graphics = ( GraphicsDeviceManager )game.Services.GetService( typeof(
IGraphicsDeviceManager ) );
      input = ( IInputHandler )game.Services.GetService( typeof(
IInputHandler ) );
   }
```

```
   public override void Initialize( )
   {
      base.Initialize( );
      InitializeCamera( );
   }

   private void InitializeCamera( )
   {
      float aspectRatio = ( float )graphics.GraphicsDevice.Viewport.Width /
                          ( float )graphics.GraphicsDevice.Viewport.Height;

      // Cria uma matriz de projeção com visão perspectiva, com ângulo de
45 graus e os limites de frustrum de 1 e 10000.
      Matrix.CreatePerspectiveFieldOfView( MathHelper.PiOver4, aspectRatio,
1.0f, 10000.0f, out projection );

      // Cria uma matriz de visão com os atributos atuais da câmera
      Matrix.CreateLookAt( ref cameraPosition, ref cameraTarget,
                                       ref cameraUpVector, out view );
   }

   public override void Update( GameTime gameTime )
   {
      float timeDelta = ( float )gameTime.ElapsedGameTime.TotalSeconds;

      // Atualiza a rotação no eixo Y (Yaw) de acordo com o teclado
      if( input.KeyboardState.IsKeyDown( Keys.Left ) )
      {
         cameraYaw += spinRate * timeDelta;
      }

      if( input.KeyboardState.IsKeyDown( Keys.Right ) )
      {
         cameraYaw -= spinRate * timeDelta;
      }

      // Atualiza a rotação no eixo X (Pitch) de acordo com o teclado
      if( input.KeyboardState.IsKeyDown( Keys.Up ) )
      {
         cameraPitch += spinRate * timeDelta;
      }

      if( input.KeyboardState.IsKeyDown( Keys.Down ) )
      {
         cameraPitch -= spinRate * timeDelta;
      }

      // Atualiza a rotação no eixo Y (Yaw) de acordo com o mouse
      if( ( input.PreviousMouseState.X > input.MouseState.X ) && (
input.MouseState.LeftButton == ButtonState.Pressed ) )
      {
         cameraYaw += spinRate * timeDelta;
      }
      else if( ( input.PreviousMouseState.X < input.MouseState.X ) && (
input.MouseState.LeftButton == ButtonState.Pressed ) )
      {
         cameraYaw -= spinRate * timeDelta;
      }

      // Atualiza a rotação no eixo X (Pitch) de acordo com o mouse
```

```
      if( ( input.PreviousMouseState.Y > input.MouseState.Y ) && (
input.MouseState.LeftButton == ButtonState.Pressed ) )
      {
         cameraPitch -= spinRate * timeDelta;
      }
      else if( ( input.PreviousMouseState.Y < input.MouseState.Y ) && (
input.MouseState.LeftButton == ButtonState.Pressed ) )
      {
         cameraPitch += spinRate * timeDelta;
      }

      // Ajusta o ângulo de rotação para que esteja sempre entre 0 e 360
      if( cameraYaw > 360 )
      {
         cameraYaw -= 360;
      }
      else if( cameraYaw < 0 )
      {
         cameraYaw += 360;
      }

      // Não permite que a rotação no eixo X passe de 90 graus
      if( cameraPitch > 89 )
      {
         cameraPitch = 89;
      }
      if( cameraPitch < -89 )
      {
         cameraPitch = -89;
      }


      // Rotaciona e translada a matriz de acordo com a entrada do usuário
      movement *= ( moveRate * timeDelta );

      Matrix rotationMatrix;
      Matrix.CreateRotationY( MathHelper.ToRadians( cameraYaw ), out
rotationMatrix );

      if( movement != Vector3.Zero )
      {
         Vector3.Transform(ref movement, ref rotationMatrix, out movement);
         cameraPosition += movement;
      }

      rotationMatrix *= Matrix.CreateRotationX( MathHelper.ToRadians(
cameraPitch ) );

      // Cria um vetor com a posição para qual a câmera está apontando
      Vector3 transformedReference;
      Vector3.Transform( ref cameraReference, ref rotationMatrix, out
transformedReference );

      // Calcula a posição para qual a câmera está olhando
      Vector3.Add( ref cameraPosition, ref transformedReference, out
cameraTarget );

      Matrix.CreateLookAt( ref cameraPosition, ref cameraTarget, ref
cameraUpVector, out view );

      base.Update( gameTime );
```

```
    }
}
```

Código 8: Implementação da classe **Camera** e da interface de acesso **ICamera**.

2.8. Classe **FirstPersonCamera**

```
public class FirstPersonCamera : Camera
{
    public FirstPersonCamera( Game game ) : base( game )
    {

    }

    public override void Update( GameTime gameTime )
    {
        movement = Vector3.Zero;

        if( input.KeyboardState.IsKeyDown( Keys.A ) )
        {
            movement.X--;
        }

        if( input.KeyboardState.IsKeyDown( Keys.D ) )
        {
            movement.X++;
        }

        if( input.KeyboardState.IsKeyDown( Keys.W ) )
        {
            movement.Z--;
        }

        if( input.KeyboardState.IsKeyDown( Keys.S ) )
        {
            movement.Z++;
        }

        if( movement.LengthSquared( ) != 0 )
        {
            movement.Normalize( );
        }

        base.Update( gameTime );
    }
}
```

Código 9: Implementação da classe **FirstPersonCamera,** extensão da classe **Camera.**

2.9. Classe **FPS**

```
public class FPS : Microsoft.Xna.Framework.DrawableGameComponent
{
    private float fps;
    private float updateInterval = 1.0f;
    private float timeSinceLastUpdate = 0.0f;
    private float frameCount = 0;

    public FPS( Game game )
        : this( game, false, false, game.TargetElapsedTime )
    {
```

```
   }

   public FPS( Game game, bool synchWithVerticalRetrace, bool
isFixedTimeStep, TimeSpan targetElapsedTime )
      : base( game )
   {
      GraphicsDeviceManager graphics = ( GraphicsDeviceManager
)Game.Services.GetService( typeof( IGraphicsDeviceManager ) );

      graphics.SynchronizeWithVerticalRetrace = synchWithVerticalRetrace;
      Game.IsFixedTimeStep = isFixedTimeStep;
      Game.TargetElapsedTime = targetElapsedTime;
   }

   public FPS( Game game, bool synchWithVerticalRetrace, bool
isFixedTimeStep )
      : this( game, synchWithVerticalRetrace, isFixedTimeStep,
game.TargetElapsedTime )
   {

   }

   public override void Initialize( )
   {
      base.Initialize( );
   }

   public override void Update( GameTime gameTime )
   {
      base.Update( gameTime );
   }

   public override void Draw( GameTime gameTime )
   {
      float elapsed = ( float )gameTime.ElapsedRealTime.TotalSeconds;
      frameCount++;
      timeSinceLastUpdate += elapsed;
      if( timeSinceLastUpdate > updateInterval )
      {
         fps = frameCount / timeSinceLastUpdate;
         #if XBOX360
            System.Diagnostics.Debug.WriteLine("FPS: " + fps.ToString());
         #else
            Game.Window.Title = "FPS: " + fps.ToString( );
         #endif
         frameCount = 0;
         timeSinceLastUpdate -= updateInterval;
      }
      base.Draw( gameTime );
   }
}
```

Código 10: Implementação da classe **FPS.**

3. Aplicação Simulacao

3.1. Classe **SimulacaoBracoMecanico**

```
public class SimulacaoBracoMecanico : Microsoft.Xna.Framework.Game
{
```

```
   GraphicsDeviceManager graphics;
   SpriteBatch spriteBatch;

   // Performance measurement (as component)
   private FPS m_cmpFpsCounter;

   // Input entry (as service and component)
   private InputHandler m_cmpInput;

   //Camera (as component)
   private FirstPersonCamera m_cmpFirstPersonCamera;

   // Interpretador de input do Wiimote
   private WiimoteInput m_cmpWiimoteInput;

   // Componente que desenha e controla o braço mecânico
   private BracoMecanico m_cmpBraco;

   public SimulacaoBracoMecanico( )
   {
      graphics = new GraphicsDeviceManager( this );
      Content.RootDirectory = "Content";

      m_cmpFpsCounter = new FPS( this );
      Components.Add( m_cmpFpsCounter );

      // Create input handling component first because the other components
use it
      m_cmpInput = new InputHandler( this );
      Components.Add( m_cmpInput );

      m_cmpFirstPersonCamera = new FirstPersonCamera( this );
      Components.Add( m_cmpFirstPersonCamera );

      m_cmpWiimoteInput = new WiimoteInput( this );
      Components.Add( m_cmpWiimoteInput );

      m_cmpBraco = new BracoMecanico( this );
      Components.Add( m_cmpBraco );
   }

   protected override void Initialize( )
   {
      base.Initialize( );
   }

   protected override void LoadContent( )
   {
      // Create a new SpriteBatch, which can be used to draw textures.
      spriteBatch = new SpriteBatch( GraphicsDevice );
   }

   protected override void UnloadContent( )
   {

   }

   protected override void Update( GameTime gameTime )
   {
      if( m_cmpInput.KeyboardState.IsKeyDown( Keys.R ) )
      {
```

```
            m_cmpBraco.Reset( );
        }

        base.Update( gameTime );
    }

    protected override void Draw( GameTime gameTime )
    {
        GraphicsDevice.Clear( Color.CornflowerBlue );

        base.Draw( gameTime );
    }
}
```

Código 11: Implementação da classe **SimulacaoBracoMecanico**, classe principal da aplicação.

3.2. Classe **BracoMecanico**

```
class BracoMecanico : DrawableGameComponent
{
    const float ROTATION_RATE = 1.0f; // Velocidade de rotação do movimento
abrir/fechar mão

    private ContentManager content; // Classe de acesso ao dispositivo
gráfico
    private FirstPersonCamera camera; // Serviço de Câmera
    private InputHandler input; // Serviço de input genérico
    private WiimoteInput wiimote; // Serviço de input do Wiimote

    Model m_mdlBraco;
    Matrix[ ] m_cmxBindPose; // Armazena as transformações da Bind Pose
(posição inicial dos bones do modelo)
    Matrix[ ] m_cmxBoneTransforms; // Armazena as transformações relativas
(offset) dos bones à Bind Pose (posição inicial dos bones do modelo)
    Matrix[ ] m_cmxWorldTransforms; // Armazena as transformações absolutas
dos bones (em relação à origem)
    Matrix[ ] m_cmxLocalTransforms; // Armazena as transformações relativas
dos bones (em relação ao pai)
    int[ ] m_ciBoneParents; // Armazena os índices do pai de cada bone
    int[ ] m_ciBoneChain; // Armazena os índices dos bones que serão
afetados pela cinemática inversa
    int m_iIndiceEndEffector; // Armazena o índice do End Effector

    // Movimento de fechar e abrir a mão
    float m_fAnguloAmplitude = 43.0f; // A mão abre e fecha 43 graus, para
cima e para baixo
    float m_fAnguloAmplitudeAtual = 0.0f;
    int m_iIndiceGarraSuperior;
    int m_iIndiceGarraInferior;

    // Debug Information - Show with the red ball where the hand should go
    Model m_mdlTarget;
    Vector3 m_v3Target = Vector3.Zero;

    public BracoMecanico( Game game ) : base( game )
    {
        //Obtêm os serviços necessários
        content = game.Content;
        camera = ( FirstPersonCamera )game.Services.GetService( typeof(
ICamera ) );
```

```
      wiimote = ( WiimoteInput )game.Services.GetService( typeof(
IWiimoteInput ) );
      input = ( InputHandler )game.Services.GetService( typeof(
IInputHandler ) );
   }

   public override void Initialize( )
   {
      base.Initialize( );
   }

   protected override void LoadContent( )
   {
      // Carrega o modelo do braço e ajusta sua posição e tamanho inicial
      m_mdlBraco = content.Load<Model>( "Models\\Braco" );
      InicializaBraco( );

      // Carrega modelo de debug
      m_mdlTarget = content.Load<Model>( "Models\\Bola" );

      base.LoadContent( );
   }

   void InicializaBraco( )
   {
      // Ajusta a posição inicial do braço
      m_mdlBraco.Root.Transform *= Matrix.CreateScale( 5.0f ) *
Matrix.CreateRotationX( MathHelper.ToRadians( -90 ) );

      // Inicializa os parâmetros necessários para a Cinemática Inversa
      m_cmxBindPose = new Matrix[ m_mdlBraco.Bones.Count ];
      m_mdlBraco.CopyBoneTransformsTo( m_cmxBindPose );

      m_cmxBoneTransforms = new Matrix[ m_mdlBraco.Bones.Count ];
      m_cmxWorldTransforms = new Matrix[ m_mdlBraco.Bones.Count ];
      m_cmxLocalTransforms = new Matrix[ m_mdlBraco.Bones.Count ];
      m_ciBoneParents = new int[ m_mdlBraco.Bones.Count ];

      for( int i = 0; i < m_mdlBraco.Bones.Count; i++ )
      {
         if( m_mdlBraco.Bones[ i ].Parent != null )
            m_ciBoneParents[ i ] = m_mdlBraco.Bones[ i ].Parent.Index;
         else
            m_ciBoneParents[ i ] = -1;
      }

      m_ciBoneChain = new int[ 3 ];
      m_ciBoneChain[ 0 ] = m_mdlBraco.Bones[ "mao" ].Index; // End effector
      m_ciBoneChain[ 1 ] = m_mdlBraco.Bones[ "antebraco" ].Index;
      m_ciBoneChain[ 2 ] = m_mdlBraco.Bones[ "braco" ].Index;

      m_iIndiceEndEffector = m_ciBoneChain[ 0 ];

      m_iIndiceGarraSuperior = m_mdlBraco.Bones[ "garra_superior" ].Index;
      m_iIndiceGarraInferior = m_mdlBraco.Bones[ "garra_inferior" ].Index;

      Reset( );
   }

   public void Reset( )
   {
```

```
      for( int i = 0; i < m_mdlBraco.Bones.Count; i++ )
      {
         m_cmxBoneTransforms[ i ] = Matrix.Identity;
         m_cmxWorldTransforms[ i ] = Matrix.Identity;
         m_cmxLocalTransforms[ i ] = Matrix.Identity;
      }

      // Preenche os valores iniciais de WorldTransforms e LocalTransforms
com a primeira chamada à UpdateTransforms
      InverseKinematics.UpdateTransforms( ref m_cmxWorldTransforms, ref
m_cmxLocalTransforms, Matrix.Identity, m_cmxBindPose, m_cmxBoneTransforms,
m_ciBoneParents );
   }

   public override void Update( GameTime gameTime )
   {
      // Anima o abrir e fechar da mão do braço mecânico ao pressionar "A"
no Wiimote
      if( input.WiimoteState.WiimoteButtonState.A )
      {
         if( m_fAnguloAmplitudeAtual < m_fAnguloAmplitude )
         {
            m_cmxBoneTransforms[ m_iIndiceGarraSuperior ] =
Matrix.CreateRotationX( MathHelper.ToRadians( ROTATION_RATE ) ) *
m_cmxBoneTransforms[ m_iIndiceGarraSuperior ];
            m_cmxBoneTransforms[ m_iIndiceGarraInferior ] =
Matrix.CreateRotationX( MathHelper.ToRadians( -ROTATION_RATE ) ) *
m_cmxBoneTransforms[ m_iIndiceGarraInferior ];

            m_fAnguloAmplitudeAtual += ROTATION_RATE;
         }
      }
      else
      {
         if( m_fAnguloAmplitudeAtual > 0.0f )
         {
            m_cmxBoneTransforms[ m_iIndiceGarraSuperior ] =
Matrix.CreateRotationX( MathHelper.ToRadians( -ROTATION_RATE ) ) *
m_cmxBoneTransforms[ m_iIndiceGarraSuperior ];
            m_cmxBoneTransforms[ m_iIndiceGarraInferior ] =
Matrix.CreateRotationX( MathHelper.ToRadians( ROTATION_RATE ) ) *
m_cmxBoneTransforms[ m_iIndiceGarraInferior ];

            m_fAnguloAmplitudeAtual -= ROTATION_RATE;
         }
      }

      // Obtem e aplica a orientação do wiimote na mão do braço mecânico
      m_cmxBoneTransforms[ m_iIndiceEndEffector ] =
Matrix.CreateFromQuaternion( wiimote.DeltaOrientation2 ) *
m_cmxBoneTransforms[ m_iIndiceEndEffector ];
      InverseKinematics.UpdateTransforms( ref m_cmxWorldTransforms, ref
m_cmxLocalTransforms, Matrix.Identity, m_cmxBindPose, m_cmxBoneTransforms,
m_ciBoneParents );

      m_v3Target = m_cmxWorldTransforms[ m_iIndiceEndEffector
].Translation;

      // Obtem o deslocamento da posição da mão (obtido do Wiimote, não
transformado)
      Vector3 v3Deslocamento = wiimote.UntransformedDeltaPosition2;
```

```
      if( v3Deslocamento != Vector3.Zero )
      {
         // Transforma o deslocamento de acordo com a posição e orientação
da mão
         v3Deslocamento = Vector3.Transform( v3Deslocamento,
m_cmxWorldTransforms[ m_iIndiceEndEffector ] );
         m_v3Target = v3Deslocamento;

         Matrix mxRotacaoTotal = Matrix.Identity;
         // Aplica o algorítmo de Cinemática Inversa para os bones
pertinentes (bones contidos no vetor BoneChain)
         for( int i = 1; i < m_ciBoneChain.Length; i++ )
         {
            // Calcula a Cinemática Inversa para o bone atual.
            mxRotacaoTotal = InverseKinematics.UpdateBone( ref
m_cmxBoneTransforms, m_ciBoneChain[ i ], m_iIndiceEndEffector,
v3Deslocamento, m_cmxWorldTransforms, m_cmxLocalTransforms ) *
mxRotacaoTotal;

            // Atualiza WorldTransforms e LocalTransforms para a cadeia de
cinemática inversa, agora que os bones foram movidos
            InverseKinematics.UpdateTransforms( ref m_cmxWorldTransforms,
ref m_cmxLocalTransforms, Matrix.Identity, m_cmxBindPose,
m_cmxBoneTransforms, m_ciBoneParents );
         }

         // Inverte a rotação total necessária para mover o braço, e aplica
na mão, para mantê-la na orientação original.
         m_cmxBoneTransforms[ m_iIndiceEndEffector ] = Matrix.Invert(
mxRotacaoTotal ) * m_cmxBoneTransforms[ m_iIndiceEndEffector ];
         InverseKinematics.UpdateTransforms( ref m_cmxWorldTransforms, ref
m_cmxLocalTransforms, Matrix.Identity, m_cmxBindPose, m_cmxBoneTransforms,
m_ciBoneParents );
      }

      base.Update( gameTime );
   }

   public override void Draw( GameTime gameTime )
   {
      // Desenha o braço
      foreach( ModelMesh mesh in m_mdlBraco.Meshes )
      {
         foreach( BasicEffect be in mesh.Effects )
         {
            be.EnableDefaultLighting( );
            be.World = m_cmxWorldTransforms[ mesh.ParentBone.Index ];
            be.View = camera.View;
            be.Projection = camera.Projection;
         }

         mesh.Draw( );
      }

      // Desenha a bola de debug (em vermelho)
      foreach( ModelMesh mesh in m_mdlTarget.Meshes )
      {
         foreach( BasicEffect be in mesh.Effects )
         {
            be.EnableDefaultLighting( );
```

```
            be.World = Matrix.CreateScale( 0.5f ) *
Matrix.CreateTranslation( m_v3Target );
            be.View = camera.View;
            be.Projection = camera.Projection;
            be.DiffuseColor = Color.Red.ToVector3( );
         }

         mesh.Draw( );
      }

      base.Draw( gameTime );
   }
}
```

Código 12: Implementação da classe **BracoMecanico.**

3.3. Classe **Program**

```
static class Program
{
   /// <summary>
   /// The main entry point for the application.
   /// </summary>
   static void Main( string[ ] args )
   {
      using( SimulacaoBracoMecanico game = new SimulacaoBracoMecanico( ) )
      {
         game.Run( );
      }
   }
}
```

Código 13: Implementação da classe estática **Program.**