EDER RAFAEL MIYASHIRO
KAREN JULIANA SUGAWARA

SIMULAÇÃO DE UM ROBÔ INDUSTRIAL UTILIZANDO MANUFATURA VIRTUAL

São Paulo
2011

EDER RAFAEL MIYASHIRO
KAREN JULIANA SUGAWARA

SIMULAÇÃO DE UM ROBÔ INDUSTRIAL UTILIZANDO MANUFATURA VIRTUAL

Monografia apresentada a Escola Politécnica da Universidade de São Paulo referente à disciplina PMR2550 – Projeto de Conclusão de Curso II

São Paulo
2011

EDER RAFAEL MIYASHIRO
KAREN JULIANA SUGAWARA

SIMULAÇÃO DE UM ROBÔ INDUSTRIAL UTILIZANDO MANUFATURA VIRTUAL

Monografia apresentada a Escola Politécnica da Universidade de São Paulo referente à disciplina PMR2550 – Projeto de Conclusão de Curso II

Área de Concentração:
Engenharia Mecatrônica

Orientador:
Professor Doutor Fabrício Junqueira

São Paulo
2011

FICHA CATALOGRÁFICA

Miyashiro, Eder Rafael
Simulação de um robô industrial utilizando manufatura virtual / E.R. Miyashiro, K.J. Sugawara. -- São Paulo, 2011.
97 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Realidade virtual 2.Simulação 3.Robótica I.Sugawara, Karen
Juliana II.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos III.t.

Aos meus pais Mariana e Jorge e ao meu irmão Denis.

Eder Rafael Miyashiro

Aos meus pais Elisabeth e Sadami e as minhas irmãs Luciana e Simone.

Karen Juliana Sugawara

AGRADECIMENTOS

Ao Professor Dr. Fabrício Junqueira pela orientação e atenção dadas durante o desenvolvimento do projeto.

Ao colega de curso Rafael Freitas da Silva pela ajuda na elaboração deste trabalho.

Aos amigos Daniel Cadario de Azevedo Centini, Eduardo Morita Ishihara, Fernando Martins Pedro, Gabriel Kimito Kiyohara, Henrique Yuji Sugimoto, João Vitor Tomotani, José Miguel Manzanares Chirinos, Martha Veronika Kozaka, Pedro Ferrari Abud e Vinicius Ito Iwasso que colaboraram direta ou indiretamente conosco.

RESUMO

O crescimento na utilização da automação e robótica nas indústrias, visando aumentar a produtividade e baixar os custos, faz com que seja importante o desenvolvimento de ferramentas para mitigar as dificuldades encontradas nessa nova realidade e otimizar os processos já existentes. Utilizando conceitos de realidade virtual, softwares de manufatura virtual permitem uma visualização gráfica 3D de um produto e de um processo de fabricação para o reconhecimento de possíveis erros e problemas na sequência de comandos de um equipamento industrial por meio de simulações, sem que sejam necessárias interrupções nas máquinas de chão de fábrica para efetuar testes. O objetivo deste trabalho é modelar a dinâmica e cinemática de um robô industrial utilizando conceitos de realidade virtual, gerando um software com uma visualização gráfica em ambiente 3D de seus movimentos. Para o desenvolvimento do projeto, o software Blender foi utilizado na modelagem gráfica do robô industrial, que então foi importada para o Microsoft XNA integrado ao Microsoft Visual Studio no qual, por meio da linguagem de programação C#, foram implementadas a modelagem dinâmica e a interface para controle do robô.

Palavras-chave: Manufatura Virtual, Realidade Virtual, Simulação, Robótica Industrial.

ABSTRACT

The increase in the use of automation and robotics in the industries aiming to increase productivity and lower costs, makes it important to develop tools to mitigate the difficulties found in this new reality and optimize the already existing processes. Using virtual reality concepts, virtual manufacturing softwares allow 3D graphical visualizations of a product or a manufacturing process to identify problems in the sequence of commands of industrial equipments by means of simulations without interruptions in the machines of the shop floor to perform tests. This works aims to model the dynamics and the kinematics of a industrial robot using concepts of virtual reality developing a software with a graphic visualization of its movements in a 3D environment. To develop this work, Blender was used to create the robot model, which was imported to Microsoft XNA integrated to Microsoft Visual Studio and then the dynamic model and user interface were created using the C# programming language.

Keywords: Virtual Manufacturing, Virtual Reality, Simulation, Industrial Robotics.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1– INTRODUÇÃO

A robótica é um campo extremamente dinâmico e junto com outras tecnologias emergentes como biotecnologia e nanotecnologia vai afetar futuras gerações com substanciais impactos sociais e econômicos (Bekey *et al.*, 2008). Um mundo cada vez mais exigente em qualidade, produtividade e baixo custo na produção industrial e com maiores automações e uso de robôs faz com que ferramentas que auxiliem as indústrias a atender essas exigências sejam de grande valor.

A recente combinação da tecnologia de realidade virtual com sistemas de manufatura virtual gera exemplos de ferramentas que aumentam a competitividade em áreas industriais como a automotiva e a naval. Projetistas podem controlar máquinas virtuais dentro de uma fábrica virtual e avaliar e verificar produtos digitais, o que ajuda a eliminar erros inesperados e, portanto, reduzir os custos gerais e o tempo. Antes da manufatura virtual ser utilizada era necessário entender formas 2D e imaginá-las no mundo real, o que era suscetível a erros. Considerável experiência e alto custo eram necessários para resultados precisos (Kim; Yang; Han, 2006).

Sistemas de manufatura virtual permitem simular corretamente a fabricação de produtos sem a necessidade de testes reais no chão de fábrica. Usando um sistema virtual, menos material é gasto, e interrupções na máquina real no chão de fábrica podem ser evitadas. A capacidade dos programas tem evoluído de simples simulações para previsões completas (análise do tempo de vida de ferramenta, análise topográfica de superfície, simulação de cavaco, vibrações, temperatura, etc.) (Kadir *et al.,* 2010).

Além dessas vantagens de projeto de produtos, também é possível utilizar a manufatura virtual para construir ambientes virtuais de aprendizado que podem dar aos alunos percepções muito próximas da realidade e acumular experiências por meio das interações com o computador (Liang, 2007).

1.1 – Objetivo do trabalho

Neste contexto, pretende-se modelar a dinâmica e cinemática de um robô industrial num ambiente de manufatura virtual utilizando conceitos de realidade virtual. Assim, é desejável desenvolver uma solução em *software* que simule os

movimentos de um braço robótico no qual seja possível a entrada dos comandos desejados ao robô e obter uma saída em ambiente gráfico 3D.

2 – REVISÃO BIBLIOGRÁFICA

2.1 – Realidade Virtual

No início, o termo Realidade Virtual (RV) gerava grandes expectativas pois se pensava que seria possível criar ambientes virtuais indistinguíveis do mundo real. Hoje, este conceito esta ligado à reprodução aceitável de objetos e ambientes do mundo real, para treinamento, entretenimento, ou auxílio em projetos (Gutiérrez *et al.*, 2008).

A RV utiliza computadores para criar ambientes 3D nos quais é possível navegar, ou seja, movimentar-se e explorar o ambiente, e com os quais pode-se interagir, ou seja, manipular os objetos presentes no ambiente (Gutiérrez *et al.*, 2008). Assim, é possível reproduzir situações cotidianas comparáveis àquelas do mundo real (Chan *et al.,* 2009), e que em alguns casos possuem um grande potencial para que os usuários aceitem como substitutas de experiências reais (Guttentag, 2010), como acontece, por exemplo, em simuladores utilizados em treinamentos militares e na medicina.

A interação do usuário com o ambiente de RV deve ser o mais próxima possível daquela com o sistema real (Potkonjak *et al*., 2010), para que a experiência torne-se mais parecida com a realidade. Esta interação pode ser feita tanto com teclado e mouse de um computador, como com um conjunto de sensores para rastrear os movimentos do usuário. Deve-se observar que a RV permite não somente interações entre usuários e ambientes virtuais, mas também a troca direta de informações entre usuários (Liang, 2008), como acontece com os mundos virtuais.

Já os sistemas de RV podem interagir com os usuários principalmente por meio dos sentidos (Gutiérrez *et al.*, 2008):

- Visão – uso de monitores CRT (*cathode ray tube*), LCD (*liquid crystal display*), plasma, e, mais recentemente, 3D, HMDs (*Head Mounted Displays*), e telas de projeção;
- Audição – métodos de gravação de áudio monoaural (não dá uma sensação espacial do som), estéreo e binaural (dão uma sensação espacial do som, sendo a binaural a que traz resultados mais parecidos com o que o homem

consegue ouvir);

- Tato – uso de luvas virtuais, e interfaces táteis;
- Olfato e paladar ainda não foram bem explorados no contexto da realidade virtual devido à complexidade das tecnologias requeridas.

A importância dos sentidos estimulados varia de acordo com a aplicação do sistema. Por exemplo, na simulação de uma cirurgia, uma sensação tátil de alta qualidade é o fator mais importante, já no caso de simulação de uma orquestra, este seria a qualidade do som (Gutiérrez *et al.*, 2008).

Segundo Liang (2007), um sistema de RV deve ser constituído de três elementos (os chamados “três Is”):

- Imersão (*Immersion*) – o usuário deve estar imerso no ambiente virtual como se estivesse no ambiente real. Uma RV imersiva pode fornecer situações ricas, interativas e atraentes, que seriam custosas, perigosas, ou impossíveis de acontecer no mundo real;
- Interação (*Interaction*) – o sistema deve ser capaz de detectar ações do usuário e produzir reações correspondentes em tempo real;
- Imaginação (*Imagination*) – como a RV é construída sobre a imaginação, pode-se ter situações não possíveis no mundo real.

Já para Gutiérrez *et al.* (2008), há dois fatores que descrevem uma experiência de RV: imersão e presença.

A imersão está relacionada com a interface entre o sistema de RV e o usuário. Um sistema pode ser totalmente imersivo (HMD), semi-imersivo (telões de projeção), e não imersivo (monitores). No início, os sistemas de RV eram totalmente imersivos, pois acreditava-se que um isolamento total de usuário em relação ao mundo real daria maior credibilidade à simulação. Sistemas semi-imersivos proporcionam gráficos de alta resolução e sensação espacial do som, além de muitas vezes permitirem que vários usuários compartilhem a simulação. Sistemas não imersivos ganharam popularidade devido ao baixo custo e facilidade de uso.

A presença está ligada ao psicológico do usuário, quando todas as informações fornecidas pela RV (imagens, sons, sensações táteis, etc) são processadas pelo usuário e entendidas como um ambiente com o qual ele pode

interagir.

Outro conceito dentro da RV é o *Continuum Realidade-Virtualidade* (Figura 1), no qual se tem de um lado a RV “pura”, onde tudo é artificial, e o usuário fica isolado da realidade; do outro lado, a Realidade Real, sem intervenções do computador, e entre elas, tem-se a Realidade Misturada, que compreende a Realidade Aumentada e a Virtualidade Aumentada, combinando imagens reais e virtuais (Gutiérrez *et al.*, 2008).

Existem inúmeras aplicações para a RV nas mais diversas áreas do conhecimento: aviação, militar, medicina, indústria, turismo, etc. (Liang, 2008). Assim, o desenvolvimento de um único sistema que atenda aos requisitos de todas essas áreas torna-se muito complicado, resultando no desenvolvimento de sistemas otimizados para cada aplicação, e com reusabilidade limitada (Gutiérrez *et al.*, 2008).

Deste modo, existem, por exemplo, diferenças significativas entre arquiteturas de diferentes mundos virtuais (como Second Life, OpenSimulator, Unity3D, etc). Para que se possa expandir além das aplicações que se têm hoje novas arquiteturas suficientemente padronizadas serão necessárias (Thompson, 2011).

Como já dito anteriormente, o uso da RV se aplica às mais diversas áreas, algumas nas quais já existem aplicações, e outras nas quais elas ainda estão sendo desenvolvidas:

- O uso de Realidade Virtual para treinamentos por simulação tem sido recomendado como um processo de certificação e credenciamento para aplicação de *stent* em carótidas, para que permita que médicos adquiriram novas habilidades sem afetar a segurança de pacientes. Estudos como os de Seymour *et al.* (2002) mostram a eficiência de treinamentos utilizando simulação com RV sobre os métodos de treinamento tradicionais (Cates, 2007).
- RV também tem muito sido utilizada na área de esportes, como nos saltos ornamentais, ginástica, trampolim acrobático, basquete, etc (Li; Sun, 2009).

  Manufatura virtual possibilita a manufatura de um produto com a redução de testes e experimentos no chão de fábrica. Não há gasto de material, nem a necessidade de parar uma máquina para realizar os testes, e não há problemas de segurança (Kadir *et al.*, 2010).

- O ensino à distância tem se desenvolvido rapidamente nos últimos anos. No entanto, percebe-se que em áreas do conhecimento mais técnicas, como é o caso da engenharia, surge uma dificuldade devido à ausência de aulas práticas. Torna-se então necessário o desenvolvimento de laboratórios virtuais. Em Potkonjak *et al.* (2010) tem-se como estudo de caso o desenvolvimento de um laboratório virtual de robótica.

Figura 1: *Continuum Realidade-Virtualidade* (Adaptado de Milgram *et al*., 2004)

2.2 – Robótica Industrial

A forte necessidade por alta produtividade e entrega de produtos finais uniformes e de qualidade faz com que cada vez mais a indústria tenha uma automação baseada em computadores. Atualmente, muitos processos de manufatura são executados por máquinas específicas que foram projetadas para executar operações pré-definidas. Isso gera certa inflexibilidade e em geral altos custos dessas máquinas. Logo isso gerou um interesse em se usar robôs com capacidade de executar uma grande variedade de processos de manufatura em ambientes mais flexíveis de trabalho e com custos de produção mais reduzidos (Fu; Gonzalez; Lee, 1987). Além dessas vantagens também determinou o aumento do uso da robótica nas indústrias, como pode ser observado na Figura 2, a melhora dos padrões de qualidade e a possibilidade de eliminar tarefas perigosas aos operadores humanos num sistema de manufatura (Siciliano *et al.*, 2009).

Um robô industrial pode ser usado para uma variedade de tarefas e é controlado por computador. Ele consiste de diversos corpos rígidos (“ligamentos”) conectados por juntas prismáticas ou de revolução. Esta estrutura mecânica é constituída de um braço (que garante mobilidade), um punho (que confere destreza)

e um efetuador (ferramenta para as tarefas do robô). A estrutura de um robô industrial manipulador é uma cadeia cinemática aberta, ou seja, a sequência de ligamentos não forma uma volta (*loop*). (Fu; Gonzalez; Lee, 1987; Siciliano *et al.*, 2009).

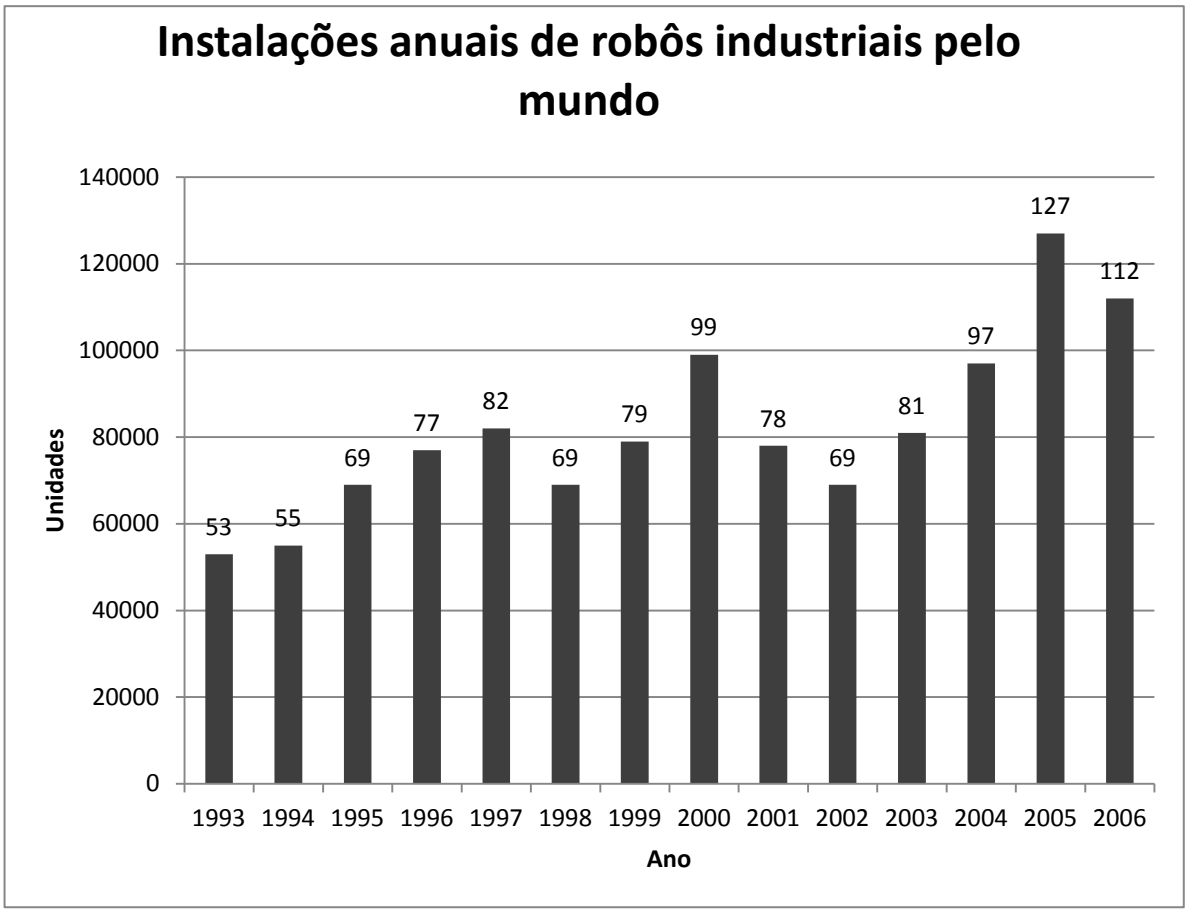

Figura 2: Instalações anuais de robôs industriais pelo mundo (Siciliano *et al*., 2009)

O volume de trabalho de um robô industrial representa o espaço do ambiente que seu efetuador pode alcançar. A forma e o volume dependem da estrutura do manipulador bem como os limites mecânicos de suas juntas. Existem classificações dos manipuladores em cartesianos, cilíndricos, esféricos, SCARA e antropomórfico que têm diferentes volumes de trabalho (Siciliano *et al.*, 2009).

Os robôs industriais disponíveis comercialmente (pode-se observar alguns exemplos na Figura 3) são amplamente utilizados em processos de manufatura e montagem como para (Fu; Gonzalez; Lee, 1987; Siciliano *et al.*, 2009):

- Manipulação de material;
- Soldagem;

- Fresamento e furação;
- Montagem de peças;
- Montagem de placas de circuito eletrônico;
- Fixação de parafusos;
- Pintura;
- Abastecendo máquinas de controle numérico;
- Manipulação em ambientes ou de materiais de risco;
- Exploração espacial e submarina.

Figura 3: Exemplos de robôs industriais (Siciliano *et al*., 2009).

O estudo do movimento dos robôs industriais envolve duas grandes áreas: cinemática e dinâmica. A cinemática trata de um estudo analítico da geometria do movimento em relação a um referencial fixo de coordenadas em função do tempo, não considerando as forças e momentos que causam o movimento. Já a dinâmica trata do movimento com equações e formulações matemáticas (mecânica de Newton e Lagrange) que descrevem o comportamento dinâmico do robô. Tais equações são muito úteis para simulações do movimento, do projeto e estrutura dos robôs (Fu; Gonzalez; Lee, 1987).

2.3 – Manufatura Virtual

O termo manufatura virtual tem sido utilizado em um grande número de contextos, uma definição é de que ele se refere abrangentemente a modelagem de sistemas de manufatura e seus componentes com o uso efetivo de ferramentas

audiovisuais e sensoriais para simular e criar alternativas para um ambiente de manufatura real, principalmente por meio do uso de computadores (Banerjee; Zetu, 2001). Os sistemas de manufatura representam precisamente toda a estrutura de um ambiente real e simulam os comportamentos físicos e lógicos dele em operação. (Iwata *et al*., 1995).

Com o desenvolvimento do rápido processamento de dados, gráficos de alta resolução, aparelhos de interação com usuários, a realidade virtual surgiu como uma das principais tecnologias dos últimos anos (Banerjee; Zetu, 2001) e tem papel importante nos sistemas de manufatura virtual, sendo uma ferramenta para visualização e interação entre o usuário e o sistema (Marinov, 2004).

Trabalho pioneiro na área foi realizado por várias grandes organizações, principalmente no campo aeroespacial, na indústria automobilística e também em grupos acadêmicos de pesquisa (Banerjee; Zetu, 2001).

Sistemas de manufatura virtual podem ser divididos em três categorias (Lawrence Associates Inc., 1994):

- Sistemas com foco no projeto do produto que usam protótipos virtuais para definição do processo a ser utilizado;
- Sistemas com foco na produção que avaliam o processo e o planejamento do tipo de usinagem a ser utilizado;
- Sistemas com foco no controle que avaliam os aspectos dinâmicos do controle de um processo produtivo;

Algumas das principais vantagens na utilização da manufatura virtual são:

- Entender e simular o comportamento de um sistema de manufatura específico (Kadir *et al.*, 2010);
- Reduzir a quantidade de teste e experimentos no chão de fábrica (Kadir *et al.*, 2010) que traz como consequências direta a economia de capital e de tempo, já que menos material é desperdiçado e ocorrem menos interrupções na máquina real;
- Menos preocupação com segurança (Kadir *et al.*, 2010) já que processos considerados perigosos ou que se ocorrerem falhas criam situações de risco podem ser simulados antes de realmente executados;

- Prever problemas e ineficiências nas funcionalidades e na usinabilidade do produto antes que a manufatura real ocorra. Protótipos virtuais podem ser montados, testados e inspecionados virtualmente. (Banerjee; Zetu, 2001);
- Redução do tempo, pois a manufatura virtual tem sido usada por companhias como General Motors e Caterpillar para construir protótipos eletrônicos de veículos, ao invés de protótipos físicos e esse processo reduz significativamente o tempo de desenvolvimento. (Banerjee; Zetu, 2001);
- Aumento da qualidade do produto dado o fato de que se pode otimizar os processos produtivos;
- Aumento da flexibilidade pois possibilita rápidas alterações na sequência de comandos e no processo produtivo utilizado, além de análises de diferentes produtos e projetos ao mesmo tempo (Dépincé *et al*., 2004);
- Melhora da relação com o consumidor, pois este pode visualizar seu produto durante a fase de projeto (Dépincé *et al*., 2004) e analisar se os resultados desejados estão sendo alcançados;
- Colaborações à distância, pois avanços recentes em redes de banda larga estão também permitindo novas aplicações para ambientes virtuais como esta onde os membros de um time de desenvolvimento podem se encontrar num sistema de projeto virtual para discutir e implementar mudanças nos protótipos virtuais. Recentes desenvolvimentos desses ambientes podem incluir projeção de gestos e movimentos de múltiplos projetistas que ativam por voz avatares que os auxilia na explicação das intenções dele para os o restante do time em tempo real. (Banerjee; Zetu, 2001);
- Facilitar a monitoração e gerenciamento de plantas de fábricas por meio das informações fornecidas e simuladas em sistemas de manufatura (Banerjee; Zetu, 2001).
- Treinamento por meio da visualização de processos e interação com os mesmos virtualmente (Banerjee; Zetu, 2001). Além de poder ocorrer a verificação se na manutenção ou na construção dos mesmos as peças terão acessibilidade como mostra a Figura 4.

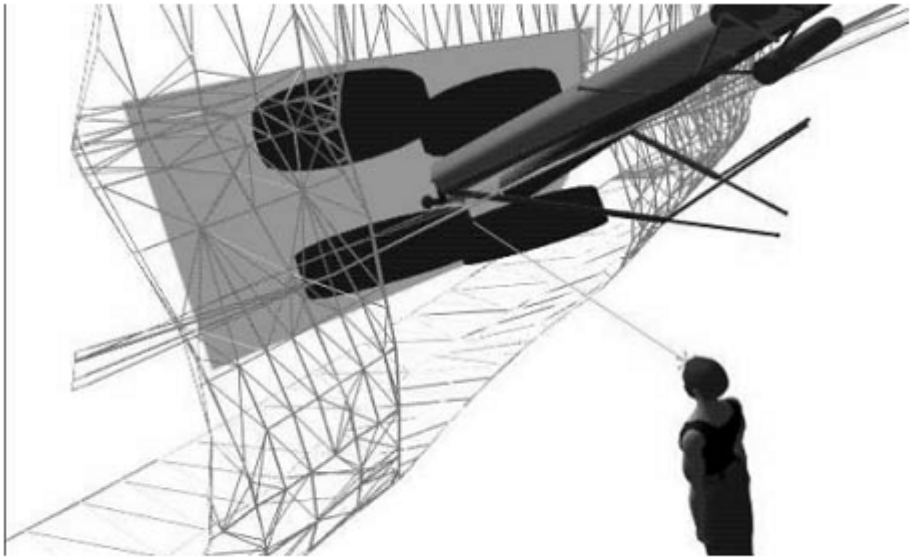

Figura 4: Verificando a acessibilidade visual embaixo de uma aeronave (Dépincé *et al*., 2004).

Algumas das principais áreas que podem se beneficiar do desenvolvimento de manufatura virtual e das consequentes vantagens citadas anteriormente, incluem:

- Projeto de produtos;
- Modelagens de operações perigosas;
- Modelagem da produção;
- Modalgem do processo;
- Treinamento;
- Educação;
- Visualização de informações;
- Telecomunicações, teleoperação colaboração à distância.

Algumas das funcionalidades que podem ser encontradas nas simulações nos sistemas de manufatura são (Kadir *et al.,* 2010):

- Verificação de código de programação;
- Geração da trajetória da ferramenta;
- Simulação da remoção de material;
- Simulação do cavaco;
- Previsão de erros de usinagem;
- Vibrações;

- Ruído;
- Análise de distribuição de temperatura;
- Otimização de processos;
- Detecção de colisão;
- Análise do tempo de vida da ferramenta;
- Análise da topologia da superfície da peça;
- Estimativa de forças;
- Análise do tempo gasto para cada processo;
- Análise de custos;

Além dessas citadas também se tem:

- Análise da utilização de fluidos;
- Visualização do volume de trabalho da ferramenta;
- Rastro do caminho percorrido pela ferramenta;

Um exemplo de aplicação das vantagens da manufatura virtual na redução de custos e otimização do projeto do produto é um caso da Boeing na área aeroespacial. O problema era que durante o processo de formação da chapa externa metálica do painel de aviões ocorriam várias deformações que desviavam a peça do seu formato desejado (criava-se ondulações) e isto resultava em alto consumo de combustível além do uso de diversos calços para montagem das peças. Utilizou-se então *softwares* de manufatura virtual para simular o processo produtivo em questão (Figura 5) e prever exatamente a deformação e otimizar para se obter as condições para alcançar o formato final desejado e nisso um processo de tentativa e erro é necessário. Logo, ao invés de executar esse processo em modelos reais, este foi feito virtualmente, o que gerou economia de tempo e custos. Com a correção do problema na fabricação e otimizando o processo, o total dos gastos de manufatura por peça foi reduzido para cerca de um terço bem como a eliminação dos calços, e também o tempo de instalação das peças, que foi reduzido de vários dias. A diminuição das deformações indesejadas trouxe uma economia de combustível enorme ao longo da vida da aeronave (MSC Software Corporation, 2001).

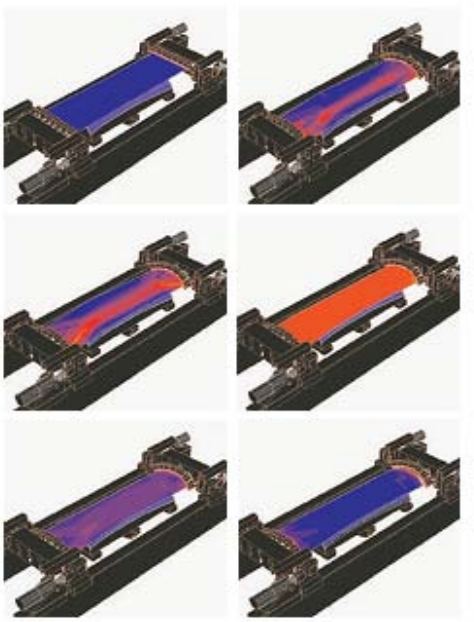

Figura 5: Simulação de peça de aeronave (MSC Software Corporation, 2001).

2.4 – Unified Modelling Language (UML)

A Unified Modelling Language (UML) é uma linguagem gráfica que auxilia na visualização e modelagem de um *software*. Segundo Booch *et al.* (1998), uma empresa de *software* bem sucedida é aquela cujo *software* atende as necessidades de seus usuários. A modelagem é uma parte central de todas as atividades que levam a um bom *software*, pois:

- Modelos ajudam na visualização do sistema como ele é ou como se desejada que ele seja;
- Modelos permitem uma especificação da estrutura e/ou comportamento de um sistema;
- Modelos documentam as decisões tomadas ao longo do desenvolvimento;
- Modelos guiam na construção de um sistema.

Uma visão geral da UML é apresentada na Figura 6.

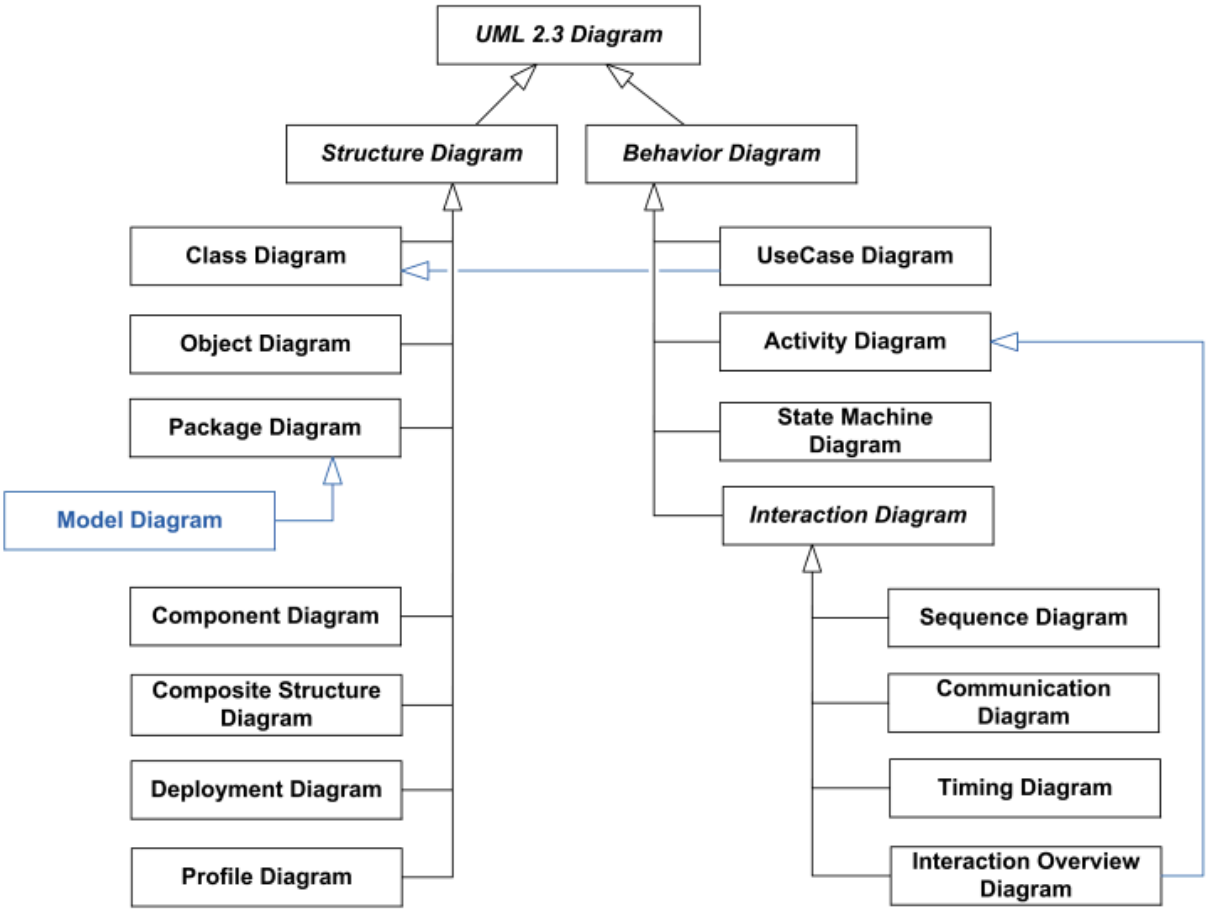

Figura 6: Visão Geral dos diagramas da UML (UML Diagrams Org, 2011).

Existem diagramas comportamentais e diagramas estruturais. Os diagramas comportamentais são utilizados para visualizar, especificar, construir e documentar os aspectos dinâmicos de um sistema. Tem-se com exemplo um diagrama de casos de uso apresentado na Figura 7. Os diagramas estruturais consideram os aspectos estáticos do sistema como classes, interfaces, componentes, objetos, etc. (Booch *et al.*, 1998).

Segundo Booch *et al.* (1998) essa linguagem tem sido empregada de maneira efetiva em diversas áreas como:

- Sistemas de informações corporativos;
- Serviços bancários e financeiros;
- Telecomunicações;
- Transportes;
- Defesa/espaço aéreo;
- Serviços distribuídos baseados na Web;
- Entre outros.

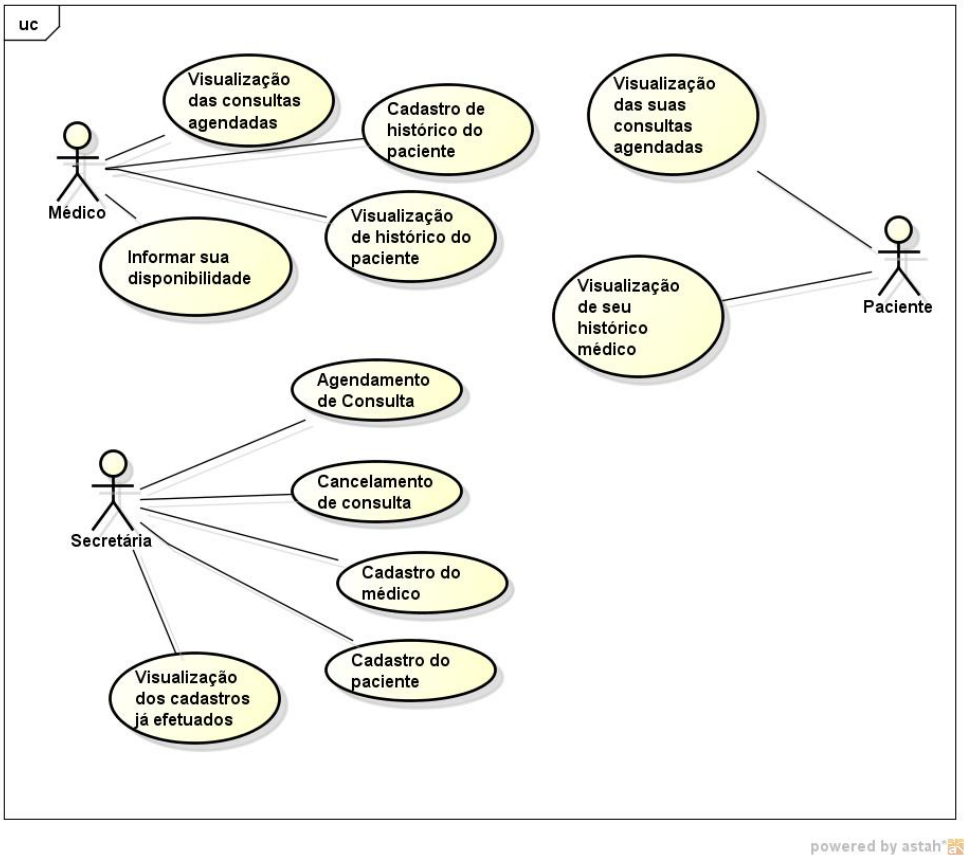

Figura 7: Exemplo de diagrama de casos de uso para confecção de um *software* de agendamento de consultas médicas.

2.5 – Blender

Blender é um *software* de modelagem e simulação 3D com as seguintes funcionalidades (Flavell, 2010):

- Possui ferramentas de texturização para pintura de superfícies de modelos;
- Possui funções para criação de esqueletos (elementos que podem ser conectados entre si e permitem a movimentação do modelo) e animação;
- Possui seu próprio motor gráfico, responsáveis pela renderização;
- Possui seu próprio módulo de composição, para combinar diferentes animações; assim como um editor de sequências de vídeos;
- Possui seu próprio motor de jogos.

É um *software* gratuito e de código aberto, ou seja, é possível baixar o programa gratuitamente e ainda ter acesso ao seu código-fonte, de modo que ele

pode ser aperfeiçoado por desenvolvedores e usuários. No entanto, o Blender nem sempre foi gratuito e de código-livre. Originalmente, foi criado como uma ferramenta de produção interna para uma companhia de animação holandesa chamada NeoGeo, fundada por Ton Roosendaal. Em 1998, Roosenal criou uma nova companhia, a Not a Number (NaN), distribuindo tanto uma versão gratuita do Blender, como uma paga, contando com algumas características adicionais. Mesmo com o Blender ganhando popularidade, a NaN não estava conseguindo juntar dinheiro suficiente para satisfazer seus investidores, o que a fez fechar as portas em 2002, e parar de trabalhar no desenvolvimento do Blender. Mesmo assim, o Blender já havia desenvolvido uma grande comunidade de usuários, de modo que Roosendaal criou uma organização sem fins lucrativos, a Blender Foundation, para a qual, por €100,000 angariados pela comunidade de usuários, os investidores da NaN liberaram o código do Blender (Gumster, 2009). Já foi recomendado pela Peugeot em seu concurso *Peugeot Design Contest* e utilizado no filme “Homem-Aranha 2” para animações e esboços de cenas.

Na Figura 8 tem-se um exemplo de modelagem de um braço robótico no Blender.

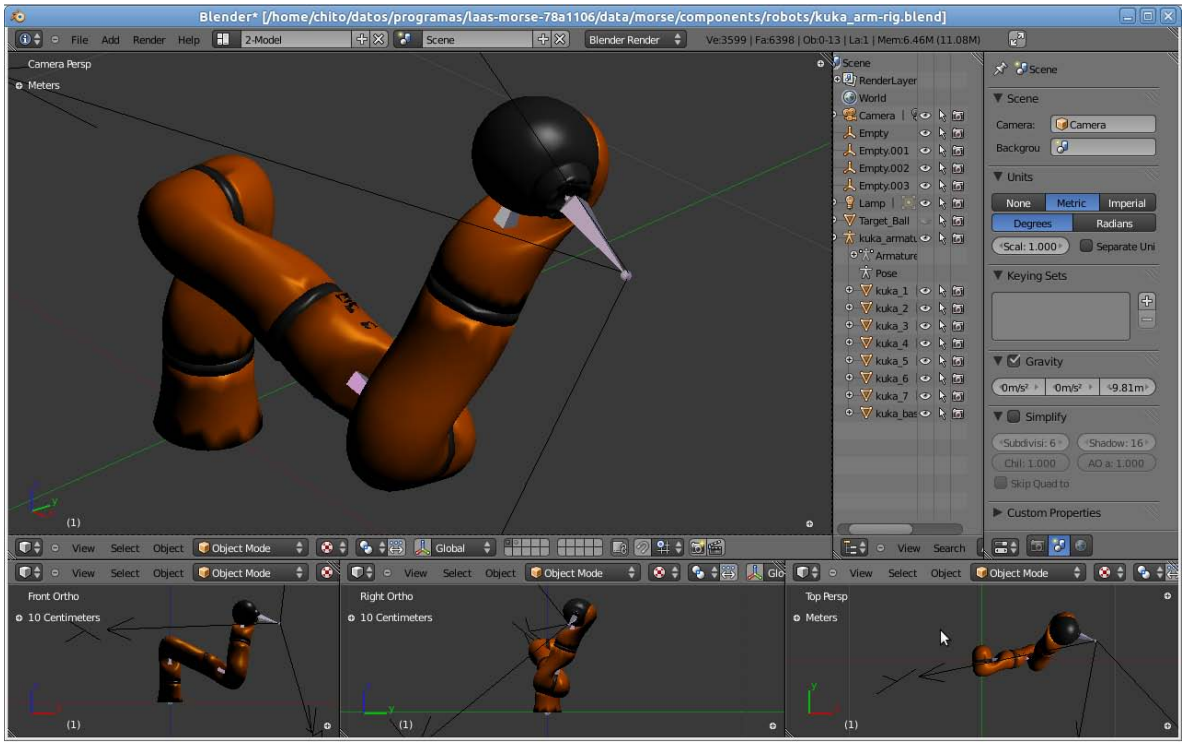

Figura 8: Exemplo de modelagem de braço robótico no *software* Blender (Laboratory for Systems Theory and Automatic Control, 2011)

2.6 – Microsoft XNA

No início, o desenvolvimento da maioria dos jogos para computadores pessoais era orientado ao MS-DOS. Assim, era necessário escrever códigos de baixo nível para interagir diretamente com placas de som, placas de vídeo, e dispositivos de entrada. Os jogos resultantes eram propícios a erros, pois diferentes fabricantes lidavam com diferentes interrupções de *BIOS*, portas de *I/O*, e bancos de memória, de modo que os jogos acabavam funcionando em um sistema e não funcionando em outro. Com o lançamento do Windows 95, surgiu o problema de que este impedia que os programadores acessassem as interrupções de baixo nível do *hardware*. A solução da Microsoft foi a criação de uma tecnologia chamada DirectX, que permitiu aos programadores criarem *softwares* que funcionassem em qualquer computador, independentemente do *hardware*. Isto porque os desenvolvedores de *hardware* passaram a desenvolver *drivers* de dispositivo, realizando todo o trabalho que antes tinha que ser feito pelos programadores. Em 2002, a Microsoft lançou o Managed DirectX, que levou as funcionalidades essenciais do DirectX para programadores do .NET *Framework* (Carter, 2008).

Enquanto o Managed DirectX 2 ainda estava em fase beta de desenvolvimento, ele foi cancelado e o Microsoft XNA foi introduzido em seu lugar. O XNA (acrônimo para "*XNA's Not Acronymed*") é composto pelo XNA *Framework*, um conjunto de bibliotecas para realizar tarefas gráficas, sonoras, entre outras; e pelo XNA Game Studio, uma extensão do Visual Studio que inclui modelos de projetos para uso do XNA *Framework*. Tais modelos incluem o *loop* principal do jogo, métodos rápidos e de fácil uso para exibir gráficos, suporte a modelos 3D, e acesso a múltiplos tipos de dispositivos de entrada (Jaegers, 2010). Além do Visual Studio, o XNA Game Studio pode ser utilizado como uma extensão do Visual C# Express, que por ser gratuito assim como o XNA, permite a criação de jogos sem custos para o programador.

O Microsoft XNA é utilizado para desenvolvimento de jogos para computadores com sistema operacional Windows, console Xbox 360, e para Windows Phone 7. Ele permite a importação de arquivos criados em *softwares* de modelagem gráfica, sendo possível importar modelos criados, por exemplo, no Blender, e utilizá-los nos jogos a serem desenvolvidos.

3 – DEFINIÇÃO DO PROJETO

Será projetado um *software* não imersivo que será baseado num sistema de manufatura virtual com foco no processo produtivo realizado por um robô industrial. *O software* terá como principais vantagens: reduzir a quantidade de teste e experimentos na máquina real, prever problemas e ineficiências nas funcionalidades e na usinabilidade do produto antes que a manufatura real ocorra, redução do tempo de projeto da peça na versão final, alta flexibilidade.

O *software* terá como principais funcionalidades: verificação de código de programação, detecção de colisão e visualização da movimentação do robô.

Para iniciar a definição de requisitos do *software*, inicialmente criou-se um esquema simplificado com os comandos de entrada, parâmetros controláveis e as respostas e visualizações geradas pelo *software* como pode ser observado na Figura 9.

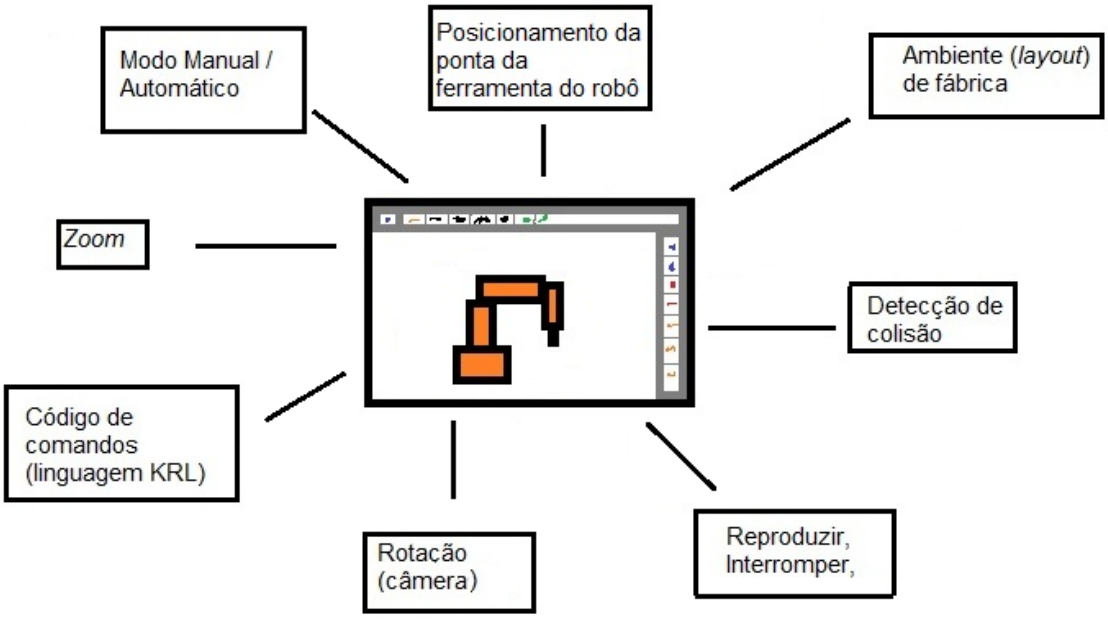

Figura 9: Diagrama esquemático dos requisitos do *software*

Com o esquema da Figura 9, foram definidos os seguintes requisitos:

- Modo Manual / Automático: o usuário pode escolher movimentar o braço robótico em modo automático, onde o computador executa a sequência de comandos definida no código de programação, ou em modo manual, onde

cada junta do robô pode ser rotacionada independentemente ou a ponta do efetuador pode ser transladada;

- *Zoom*: o usuário pode ampliar ou reduzir a visualização da tela;
- Códigos de programação: o usuário pode carregar e editar códigos de movimentação do braço robótico que definem a sequência de movimentos a ser executada e também a rotação, velocidade de avanço, etc.;
- Rotação de câmeras: o usuário pode rotacionar a visualização;
- Comandos de reprodução: apenas em modo automático, o usuário pode iniciar e interromper a execução da sequência de comandos;
- Posicionamento atual da ponta da ferramenta do robô: visualização das informações referentes a coordenadas atuais da ponta do robô;
- Ambiente: visualização de um ambiente de fábrica;
- Detecção de colisão: caso o robô encoste em algum objeto, haverá a visualização e identificação de colisão;
- Som: por meio de sinais sonoros será indicado, por exemplo, o funcionamento, a movimentação, a colisão e a usinagem do robô;

A partir da definição de requisitos anterior foram desenvolvidos diagramas UML. O diagrama de casos de uso pode ser observado na Figura 10.

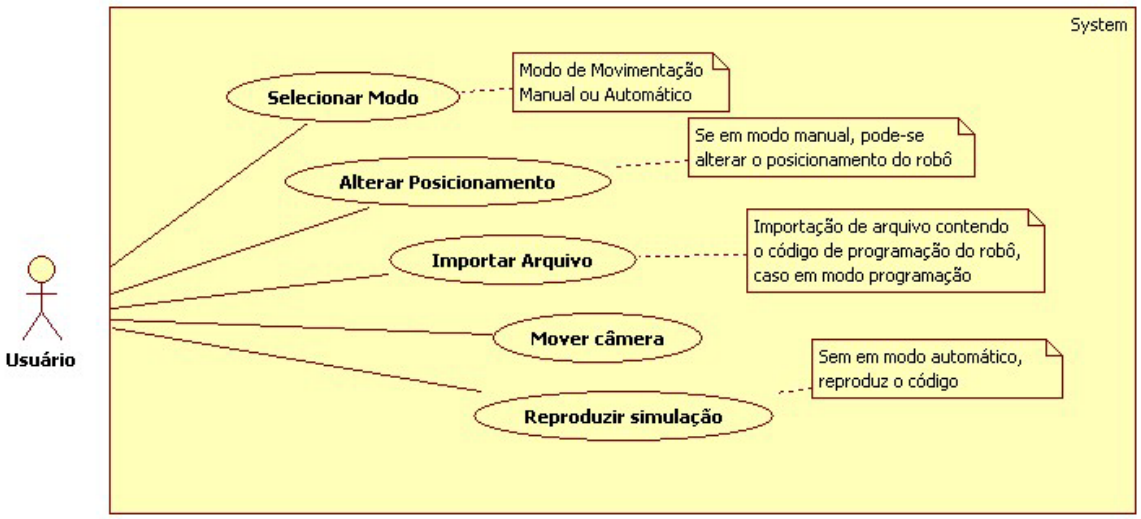

Figura 10: Diagrama de casos de uso do projeto

A seguir pode-se observar os diagramas de atividades:

Na Figura 11, a atividade “Selecionar modo”, na qual pode-se escolher entre “Manual” (movimentação manual do robô), ou “Automático” (interpretação de código).

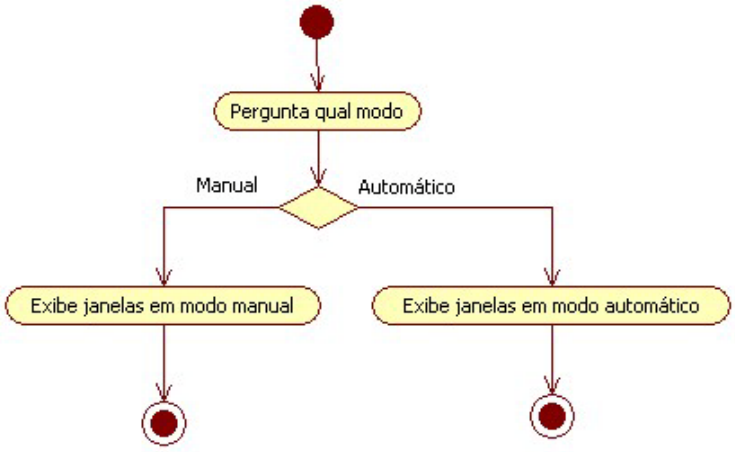

Figura 11: Diagrama de atividade - Selecionar modo

Na Figura 12, a atividade “Alterar posicionamento”, na qual verifica-se caso a nova posição desejada gera uma colisão e, em caso negativo, realiza a movimentação para esta nova posição.

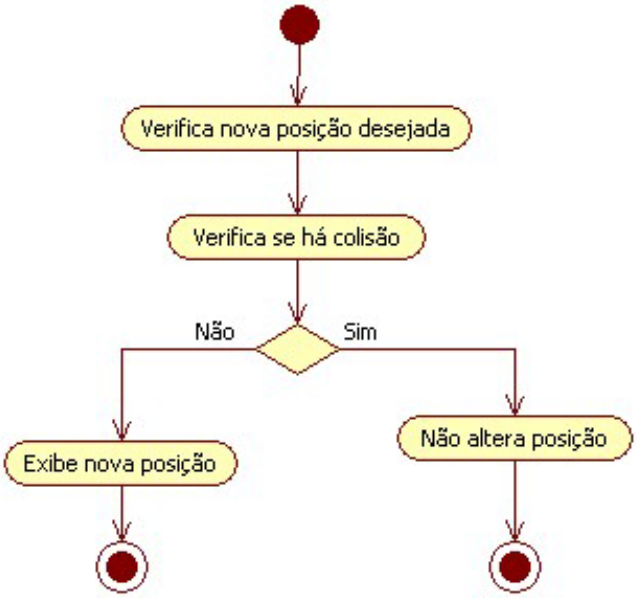

Figura 12: Diagrama de atividade - Alterar posicionamento

Na Figura 13, a atividade “Importar arquivo”, na qual altera-se para a janela de programação e verifica erros no código, exibindo-os caso existam, ou habilitando o início da simulação em caso contrário.

Figura 13: Diagrama de atividade - Importar arquivo

Na Figura 14, a atividade “Mover câmera”, na qual verifica-se a nova posição desejada para a câmera e move-se ela.

Figura 14: Diagrama de atividade - Mover câmera

Na Figura 15, a atividade “Reprodução”, na qual o código é interpretado. Caso haja movimentação, verifica-se a ocorrência de colisão, se esta existir, o programa para, em caso contrário, continua executando até os comandos até o fim do código.

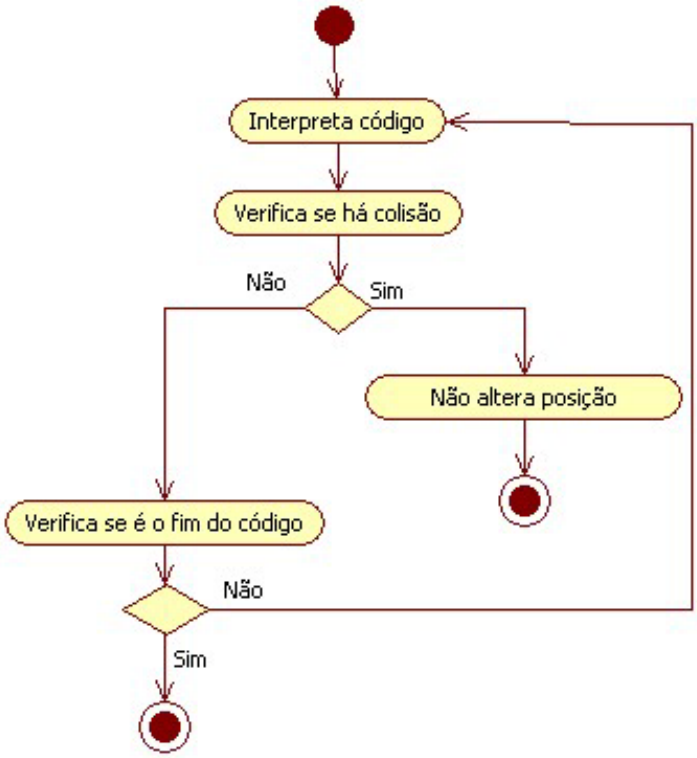

Figura 15: Diagrama de atividade - Reprodução

Na Figura 16, pode-se observar o diagrama de classes do projeto, tendo em vista as atividades e casos de uso.

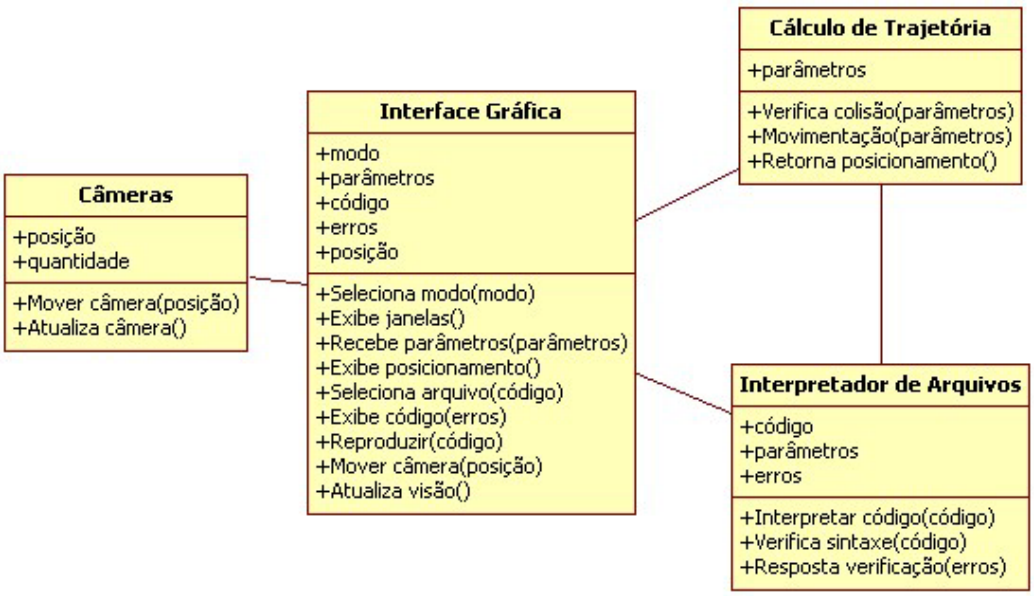

Figura 16: Diagrama de classes do projeto

A partir do diagrama de classes foram desenvolvidos os diagramas de sequência:

Na Figura 17, para a atividade “Seleciona modo”, o usuário utiliza a função “Seleciona modo” da “Interface Gráfica”, e esta devolve o desejado com “Exibe janelas”.

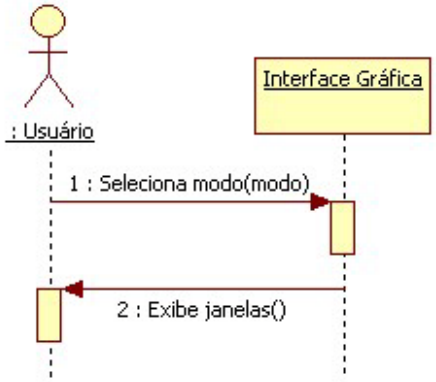

Figura 17: Diagrama de sequência - Seleciona modo

Na Figura 18, para a atividade “Altera posicionamento”, o usuário utiliza a função “Recebe parâmetros” da “Interface Gráfica”, mandando parâmetros da posição desejada. O “Cálculo de trajetória” verifica se há colisão e calcula a posição desejada, retornando o posicionamento para a “Interface Gráfica”, que exibe o resultado ao usuário.

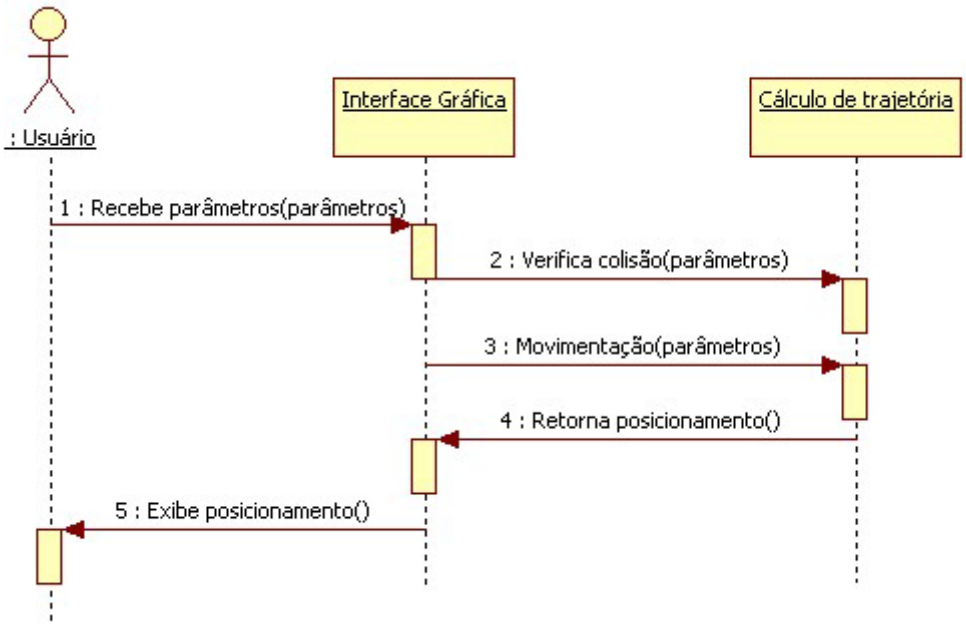

Figura 18: Diagrama de sequência - Alterar posicionamento

Na Figura 19, para a atividade “Importar arquivo”, o usuário seleciona o arquivo a ser utilizado, que é verificado pelo “Interpretador de Arquivos” e retorna erros caso estes existam, senão ele exibe o código na tela.

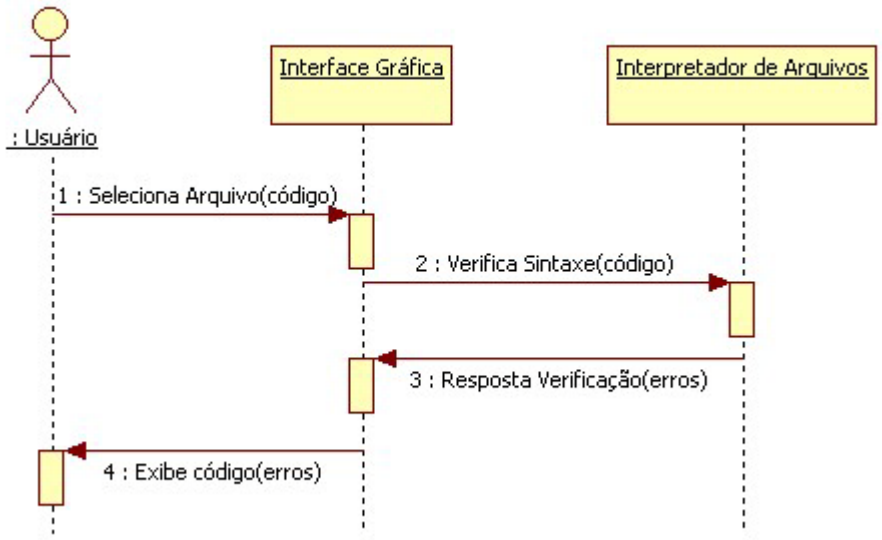

Figura 19: Diagrama de sequência - Importar arquivo

Na Figura 20, para a atividade “Mover câmera”, manda para a “Interface Gráfica” o novo posicionamento desejado, e esta repassa a informação para “Câmeras”, que atualiza o posicionamento da câmera.

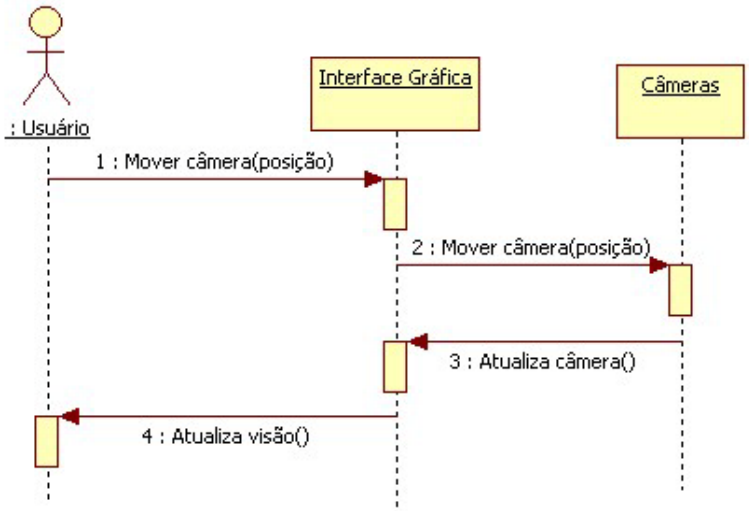

Figura 20: Diagrama de sequência - Mover câmera

Na Figura 21, para a atividade “Reprodução”, o usuário manda o código ser reproduzido, o que é feito pelo “Interpretador de Arquivos”, que utilizando “Cálculo de trajetória”, verifica se ocorrem colisões e realiza as movimentações desejadas.

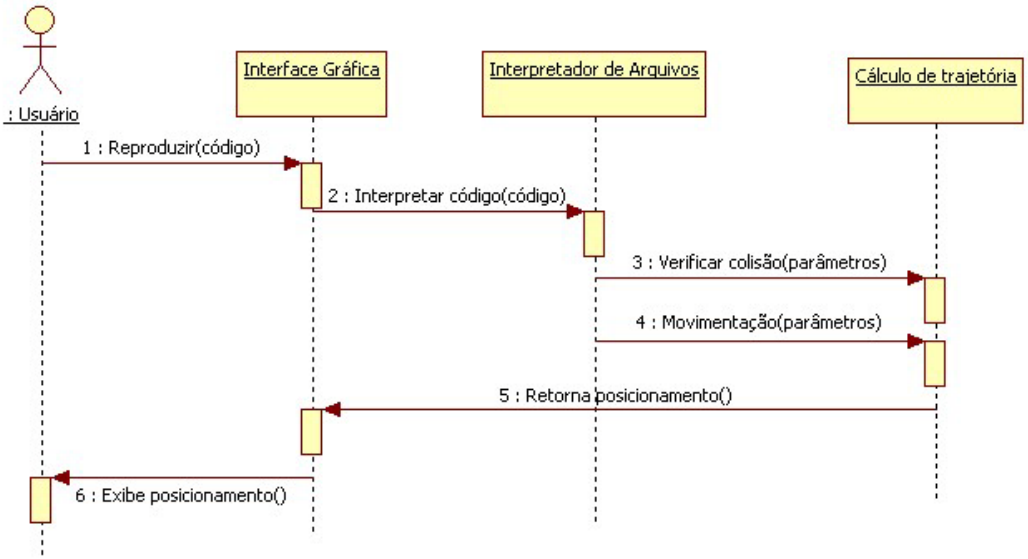

Figura 21: Diagrama de sequência - Reprodução

O robô industrial implementado é baseado no robô modelo 15/2 fabricado pela Kuka. As principais áreas de aplicação deste são: manuseamento, montagem, aplicação de colas e vedantes, soldadura MIG/MAG, etc. Este modelo de robô pode ser programado por meio da linguagem *Kuka Robot Language* (KRL) que contém a sequência de comandos a ser seguida pelo robô.

A modelagem gráfica do robô será feita no *software* Blender com a utilização das medidas aproximadas do real. O modelo gerado será então importado para o Microsoft XNA e, a partir daí, sua dinâmica e os requisitos desejados para o *software* serão programados em linguagem C#.

4 – IMPLEMENTAÇÃO

4.1 – Modelagem Blender

Um recurso do *software* que foi utilizado é a possibilidade de se colocar imagens de fundo durante a modelagem, que pode ser utilizada como referência para se criar um objeto na forma desejada e analisar as medidas do tamanho do modelo, assim como pode ser observado na Figura 22. Essa figura mostra uma imagem que foi colocada ao fundo do Blender para ser utilizada como referência durante a modelagem.

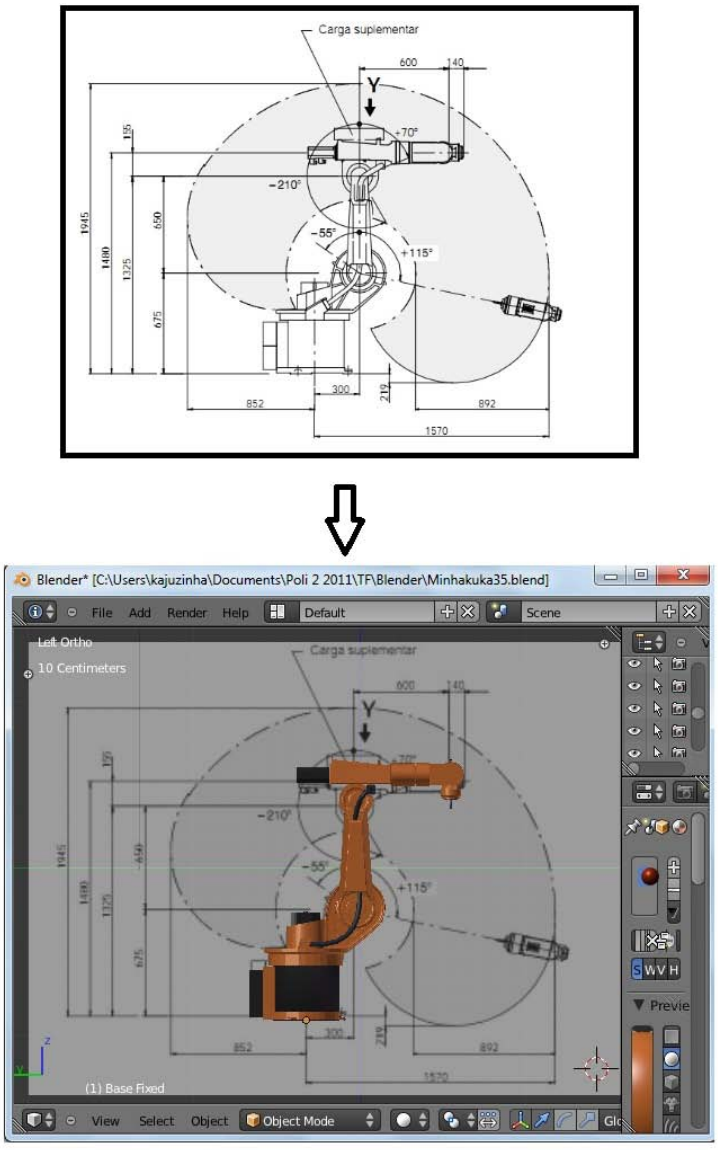

Figura 22: Imagem de fundo no Blender

Existem duas principais formas de se criar, editar e alterar objetos dentro do Blender: modo objeto e modo de edição. No modo objeto pode-se fazer diversas alterações de tamanho, forma, cor, entre outros na peça vista como um todo, enquanto no modo de edição tem-se acesso ao posicionamento de cada vértice, aresta e face. Na Figura 23 pode-se observar uma peça sendo vista no modo de edição (peça visualizada a partir de arestas e vértices) enquanto o restante do modelo é visualizado no modo objeto.

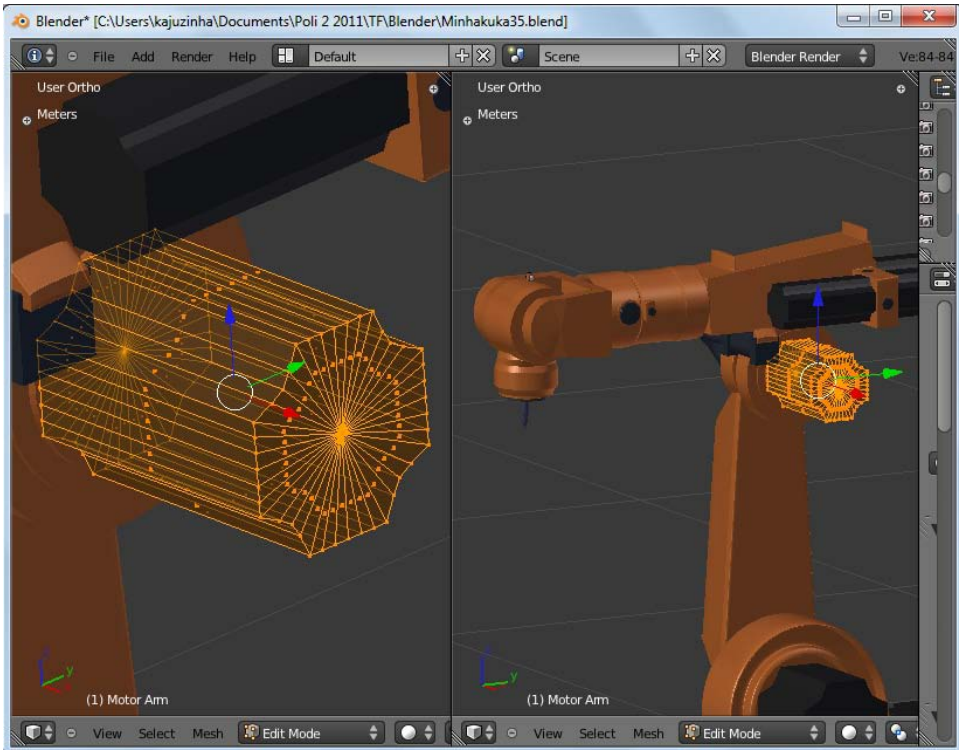

Figura 23: Modo de edição

O modo de edição é importante neste projeto, uma vez que para que o robô modelado no Blender fosse importado com sucesso no Microsoft XNA, ele teve que ser criado seguindo algumas regras:

- Todos os objetos e armadura tinham que ter seu centro numa mesma posição no modo objeto, por exemplo na origem dos eixos (X=0, Y=0, Z=0);
- Todos os objetos tinham que ter valor 1.0 no campo escala do modo objeto;
- Todos os objetos não poderiam ter rotação no modo objeto, ou seja, rotações em todos os eixos iguais a zero.

Ou seja, dentro do modo objeto fica-se sujeito a essas limitações e todas as alterações de posição, tamanho e rotação deveriam ser feitas no modo de edição para que o modelo seguisse as regras anteriores e fosse importado de modo adequado no Microsoft XNA.

O modelo obtido no Blender do braço robótico é observado na Figura 24, onde também visualiza-se um recurso do *software* para se ter mais de uma vista do objeto.

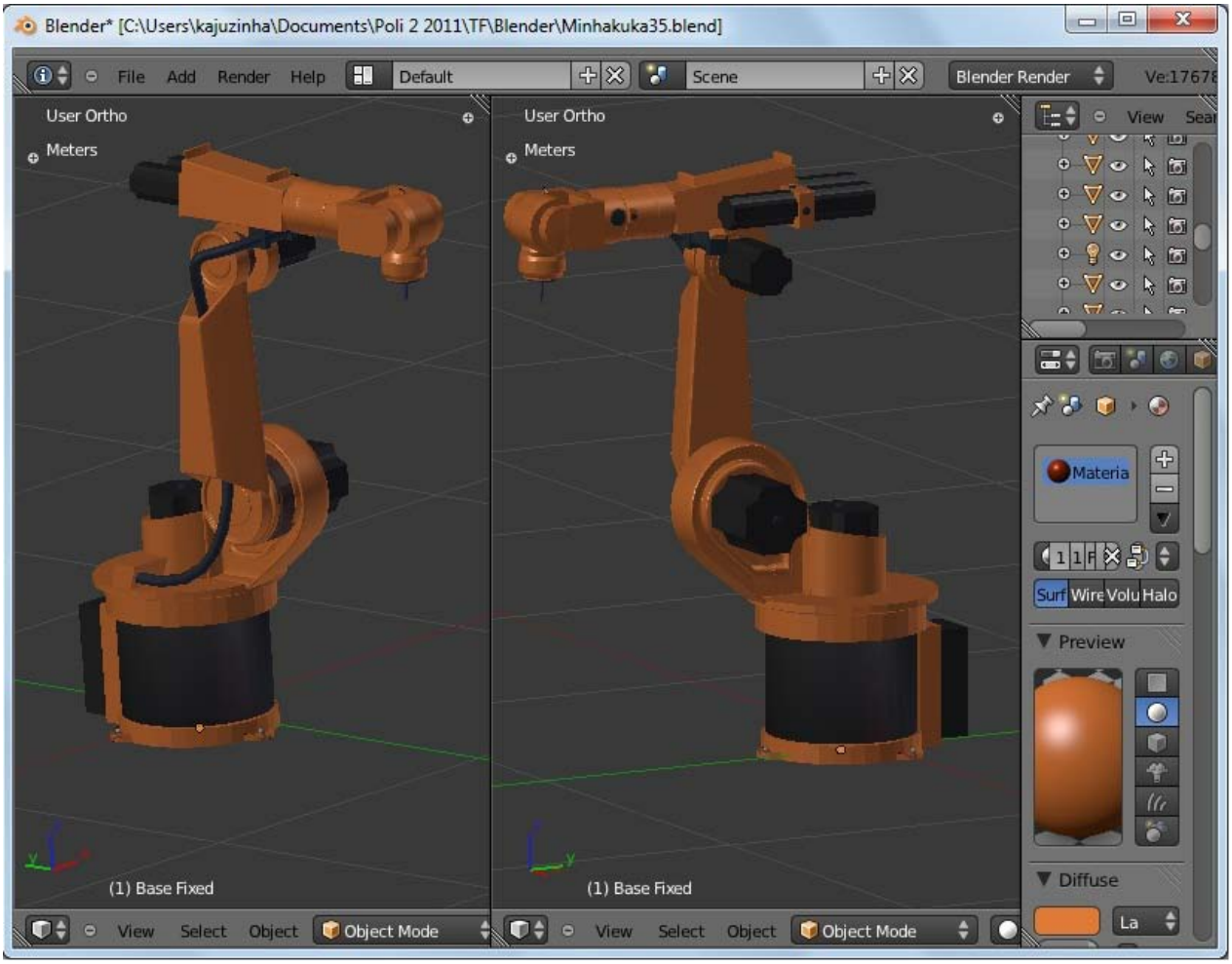

Figura 24: Modelo final no Blender

Para possibilitar a movimentação do robô no Microsoft XNA escolheu-se o método de criar uma armadura no modelo. Uma armadura no Blender é uma estrutura parecida com um esqueleto formado por ossos ligados entre si que ao se movimentarem, movimentam e deformam os objetos relacionados a eles. (Gumster, 2009).

Existem quatro tipos de ossos dentro do Blender (Figura 25), cada um tem propriedades diferentes em relação ao outro, como por exemplo o da Figura 25 (c) que permite movimentos suaves deformando como se fosse uma curva, no caso deste projeto precisa-se basicamente de movimentos de rotação, por isso foi utilizado o tipo octaedro como o da Figura 25(a) que é o tipo de osso padrão.

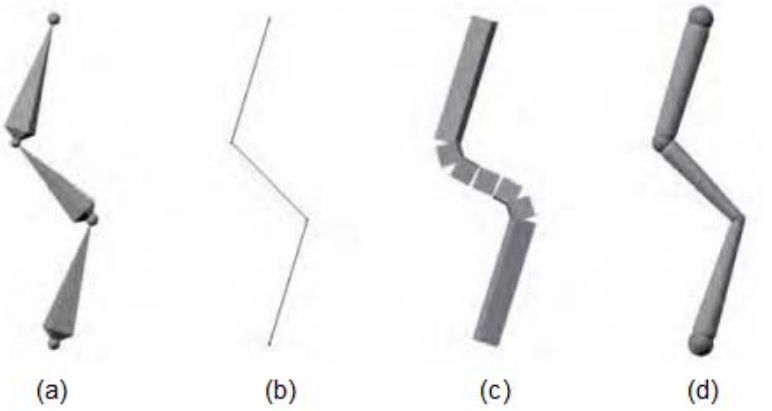

Figura 25: Tipos de ossos no Blender (Flavell, 2010)

A Figura 26 mostra o exemplo de alguns dos ossos colocados no modelo. No lado esquerdo desta figura pode-se observar o posicionamento de três ossos no robô, apresentado na visualização *Bounding Box*, e do lado direito pode-se observar o robô com seus ossos numa visualização com o robô em modo sólido.

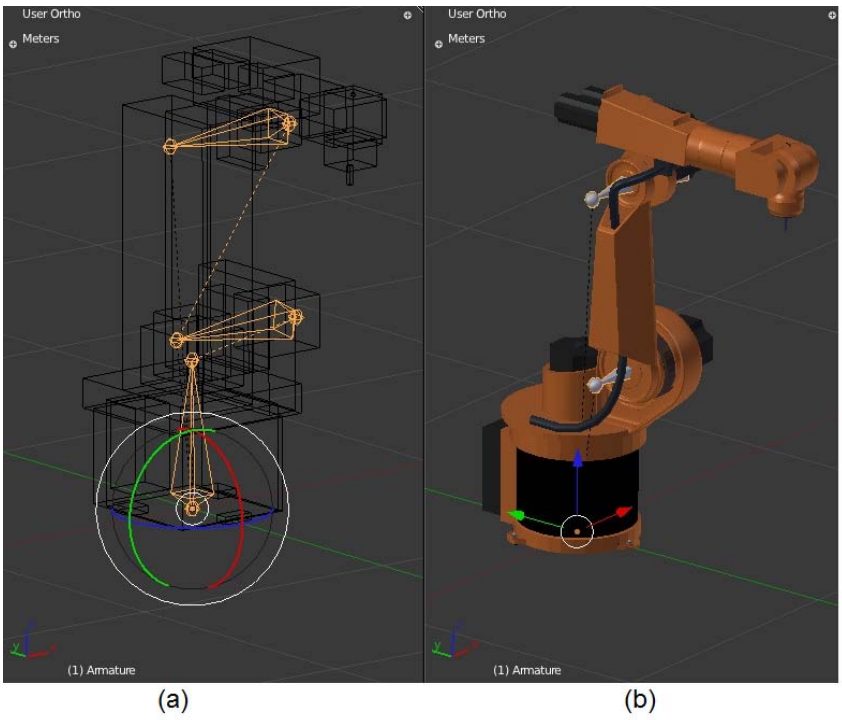

Figura 26: Armadura no Blender: (a) *Bounding Box* ; (b) Modo sólido

Dois exemplos da utilização dos ossos no próprio Blender para mudar o posicionamento do robô podem ser observados na Figura 27.

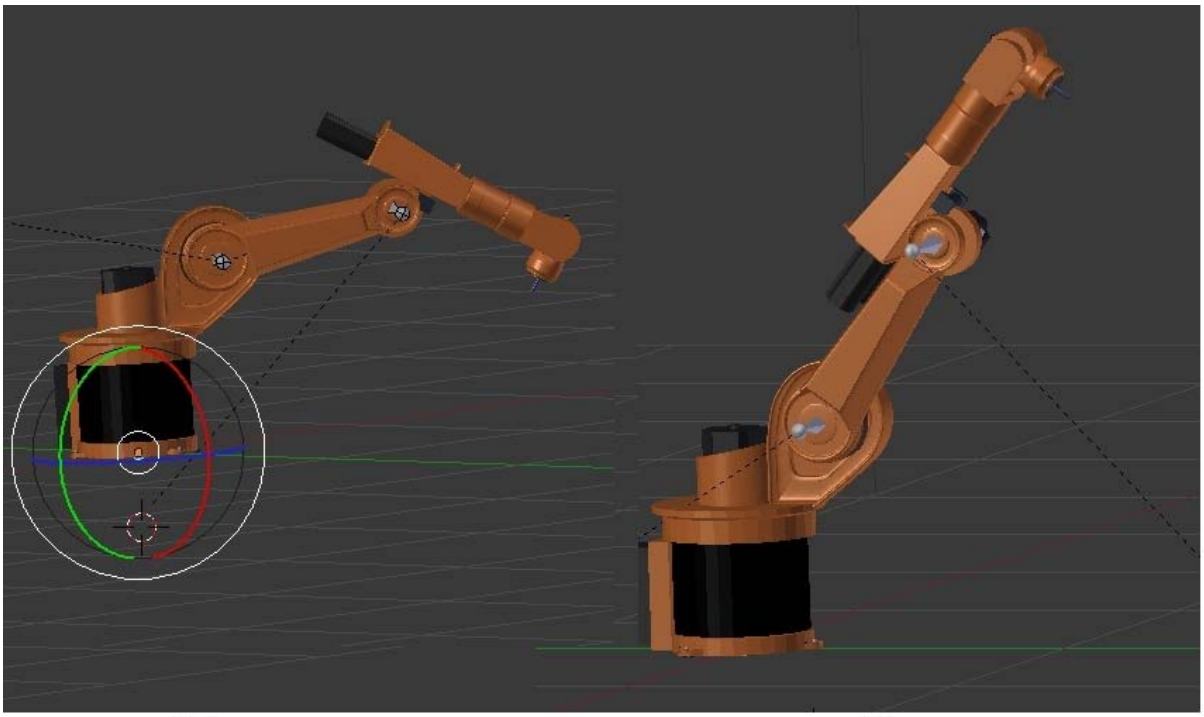

Figura 27: Diferentes posições do robô: (a) recuado , (b) extendido

Para a criação de um cenário para o robô durante a animação no Microsoft XNA também foi utilizado o método de *cube mapping* que consiste em colocar imagens nos 6 lados de um cubo para criar um ambiente desejado. Se for desejado um ambiente que tenha a aparência de ser bem maior do que ele realmente é, pode-se projetar imagens com objetos ao longe nas faces do cubo, este método é chamado de *skybox* e é bastante utilizado no desenvolvimento de cenário para jogos. O modelo criado para o programa de testes foi nomeado de *skybox* e foi desenvolvido no Blender a partir de um cubo e pode ser visualizado na Figura 28.

No cenário incluiu-se uma mesa dentro do ambiente de trabalho do robô para a simulação de um objeto com o qual ele pudesse interagir e possivelmente colidir se movimentarmos o robô com erros no código ou na movimentação manual. A mesa foi modelada no Blender, e a partir da mesma técnica utilizada anteriormente para o *skybox,* o UV Mapping, ela obteve a aparência de ser feita de madeira. Nessa técnica, coloca-se uma mesma imagem de madeira nas várias faces do objeto. Obtendo o resultado que pode ser visualizado na Figura 29.

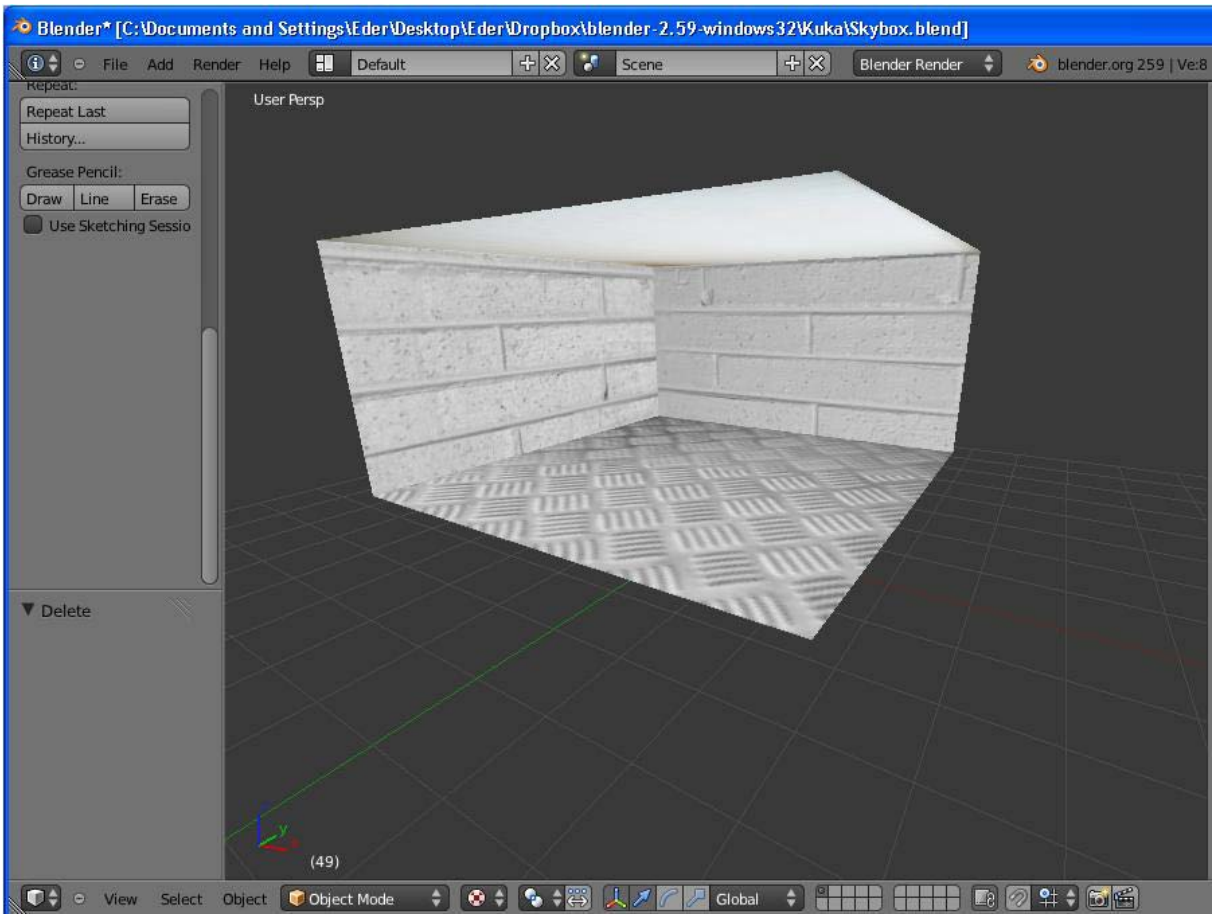

Figura 28: *Skybox*

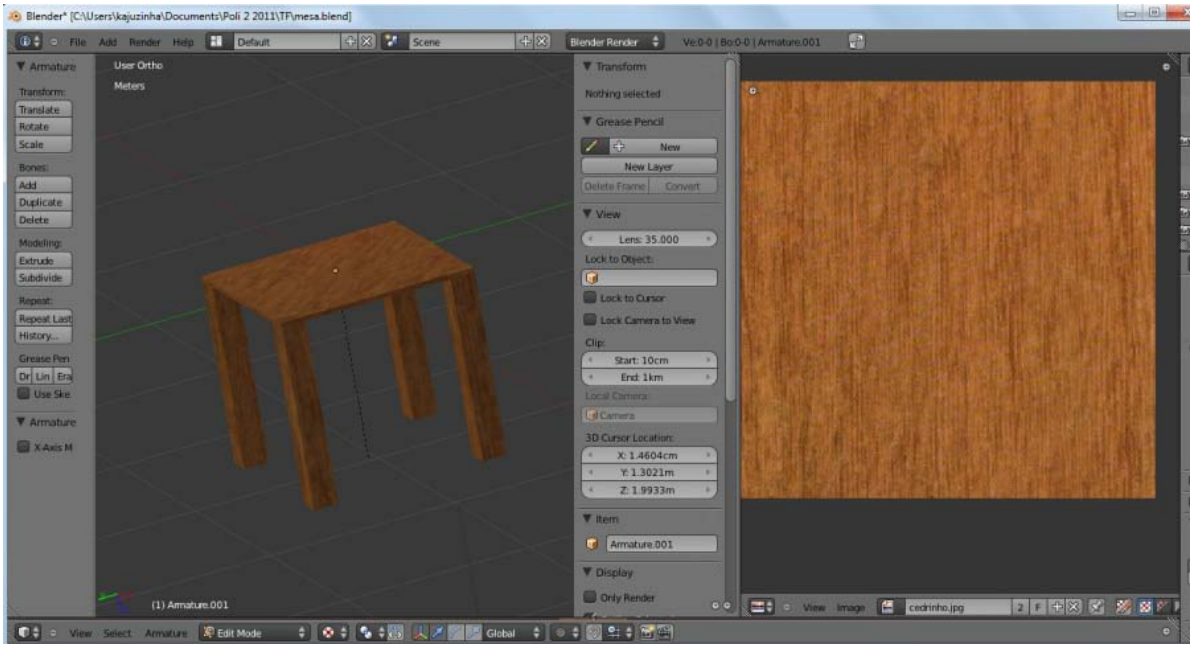

Figura 29: Mesa sendo modelada no Blender

Considerando que o *software* gera visualizações gráficas tridimensionais da movimentação do robô, e na tela são fornecidos valores do posicionamento no espaço do efetuador, decidiu-se a representação no espaço do sistema de coordenadas adotado. Com este propósito, foram modelados no Blender os eixos do sistema (Figura 30).

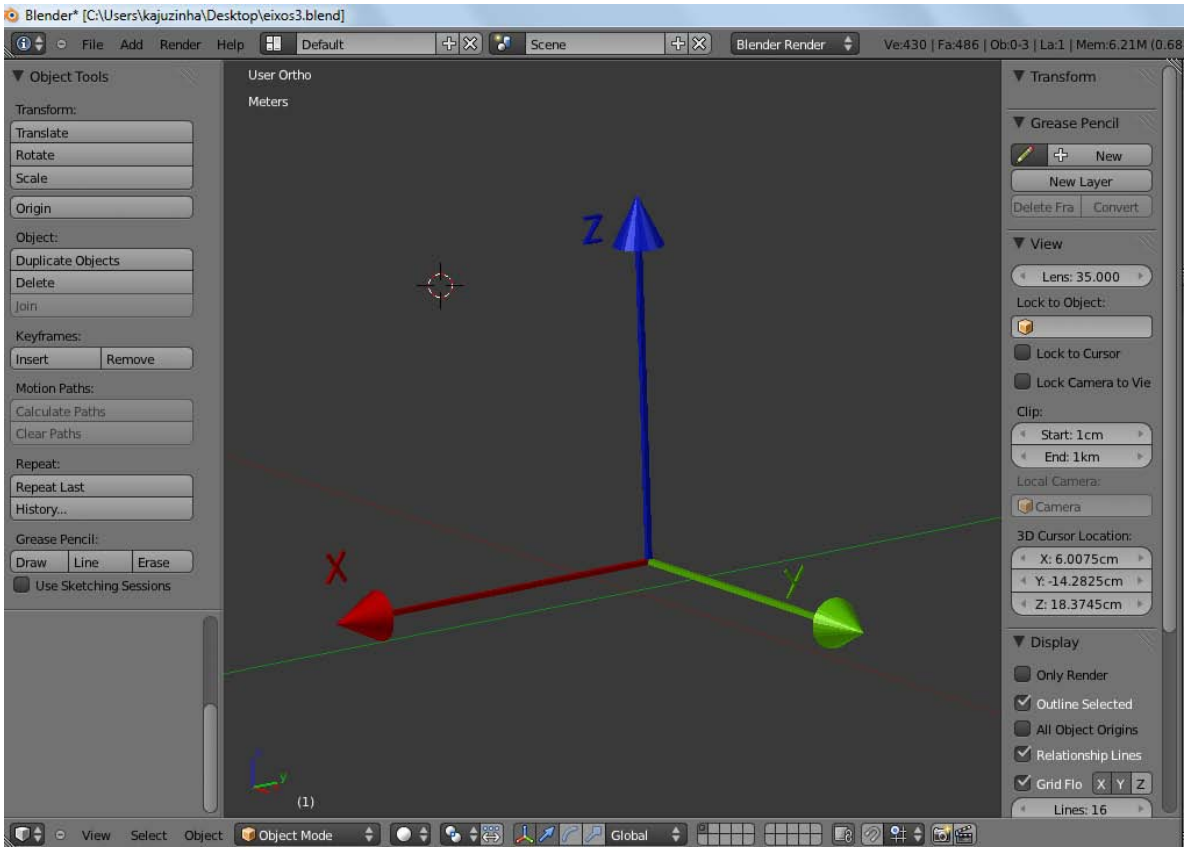

Figura 30: Modelo no Blender para representação do sistema de coordenadas

No *skybox*, na mesa e no sistema de coordenadas, também foram adicionados um osso em cada um, para permitir o posicionamento dos mesmos no ambiente tridimensional.

4.2 – Importação para o XNA

Após realizar a modelagem no Blender, é necessário exportar o modelo para um formato reconhecível pelo XNA. O Blender possui um *script* de exportação com as características necessárias para um funcionamento correto dentro do XNA 4.0 (Figura 31), dado que as regras de definidas na seção anterior sejam seguidas.

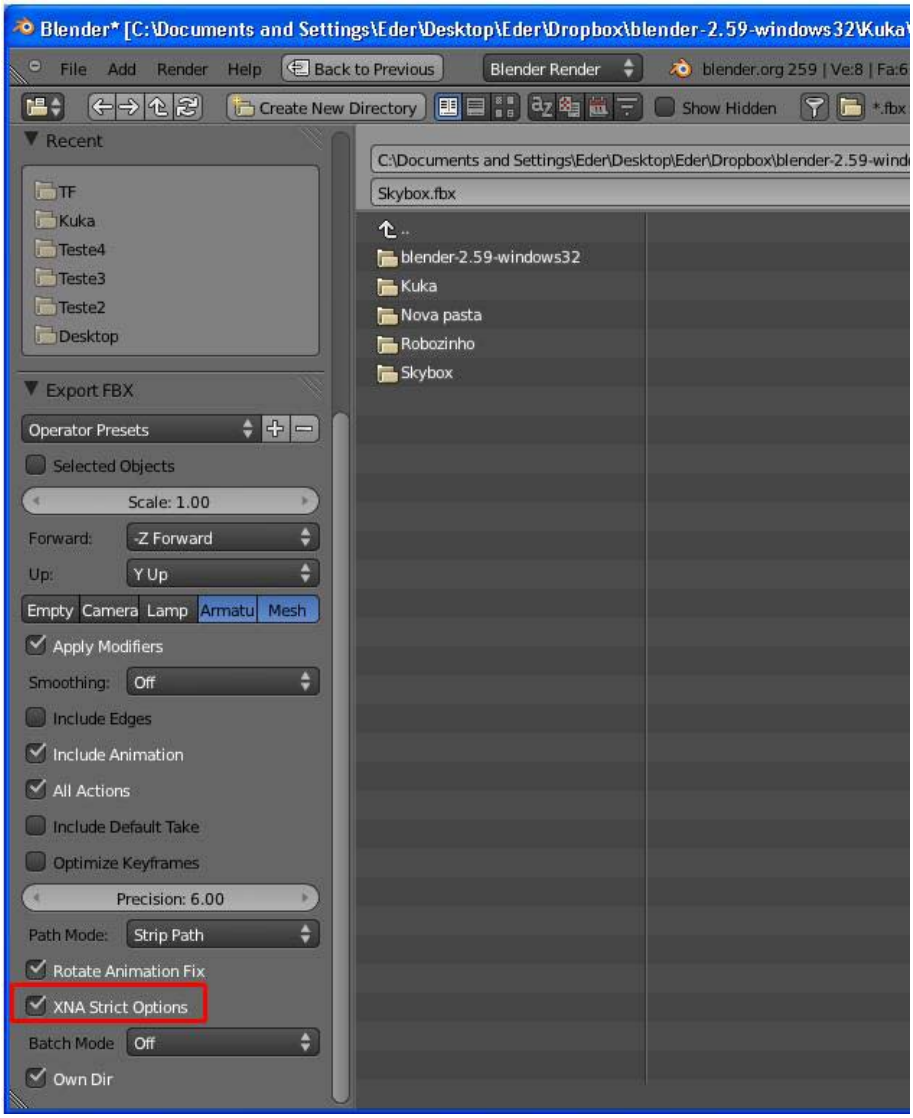

Figura 31: Tela de exportação do Blender

Porém, realizando testes com modelos simples, verificou-se que a movimentação não acontecia do modo desejado. Movendo-se um osso, os objetos associados a ele não se moviam, sendo necessário atuar diretamente sobre os objetos, o que acaba com a estrutura hierárquica criada pela armadura.

Para resolver tal problema, o arquivo importado do Blender foi aberto e verificou-se que, em vez de os objetos estarem ligados aos ossos da armadura, eles estavam ligados a um objeto “Scene” (Figura 32). Assim, o arquivo foi alterado para que cada objeto ficasse ligado ao osso correto (Figura 33). Então verificou-se que o modelo passou a se movimentar do modo desejado. Os objetos que não apresentarem movimentos podem continuar ligados ao objeto “Scene”.

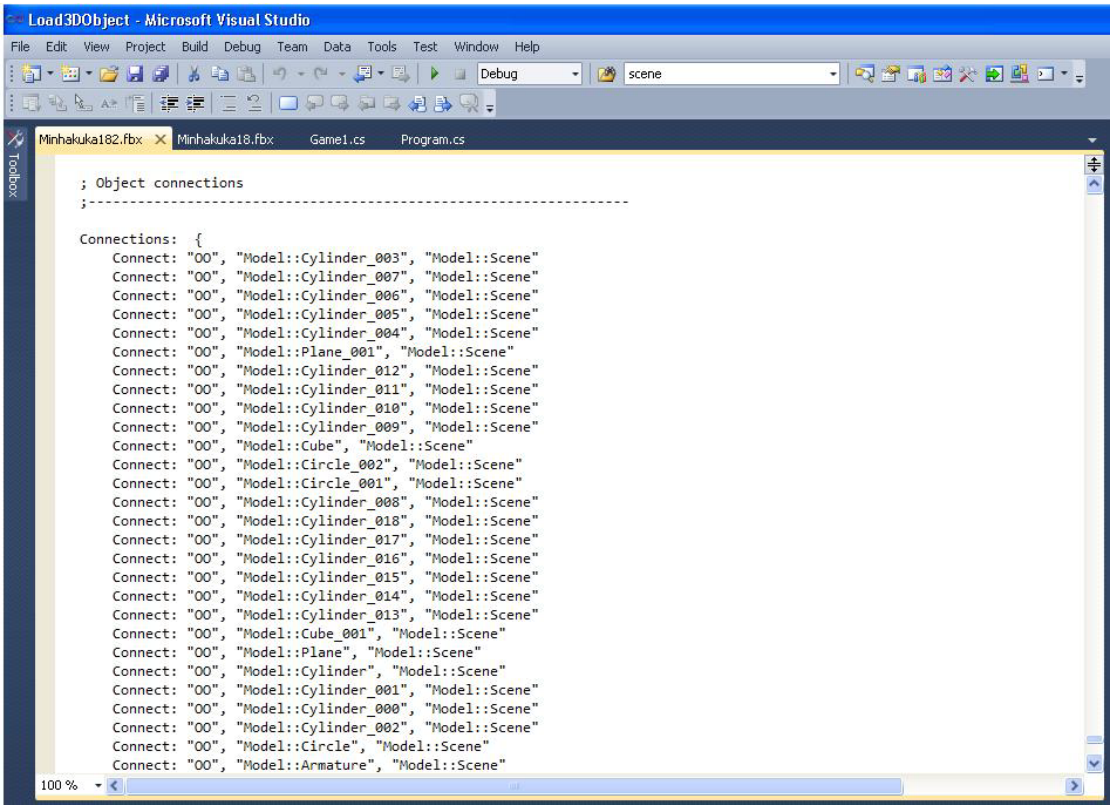

Figura 32: Arquivo exportado não modificado

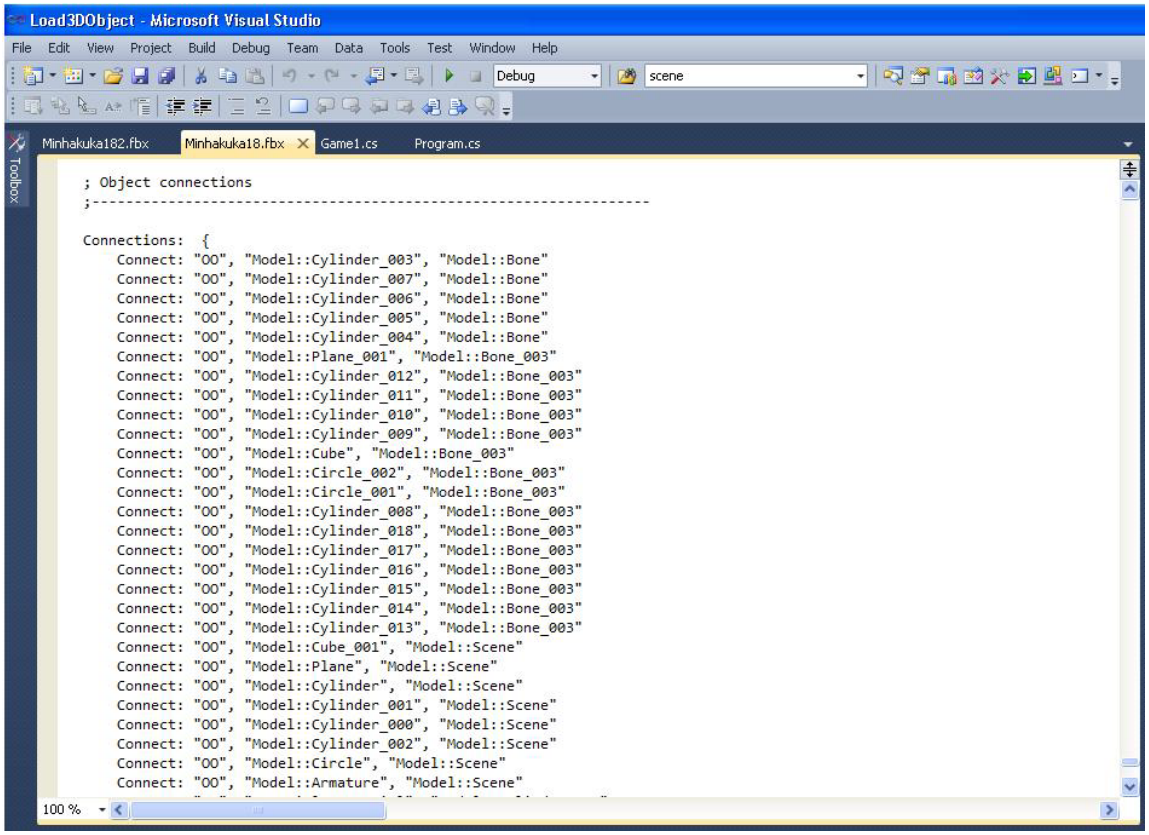

Figura 33: Arquivo exportado modificado

Para as funcionalidades que envolviam áudio e imagens bidimensionais (botões, tela inicial, etc.) foram importados também para o XNA arquivos de som e de imagem a serem carregados no *software*. Para o áudio, utilizou-se um som de motor a ser executado durante a movimentação do robô e um som de colisão a ser executado caso a ferramenta do robô colidisse com a mesa do cenário.

4.3 – Programação no XNA

XELibrary

Para a programação no XNA, a referência (Carter, 2008) foi amplamente utilizada. Seguindo seus exemplos, foi criada uma biblioteca “XELibrary” contendo quatro classes necessárias para auxiliar no desenvolvimento do projeto.

A classe “FPS” calcula a quantidade de quadros por segundo (*frames per second* - FPS) exibida, ou seja, a quantidade de vezes que as imagens são atualizadas por segundo. Normalmente considera-se um valor de 60 FPS como aceitável para que uma animação seja vista sem perda de qualidade. Com esta classe, pode-se verificar durante todo o processo de desenvolvimento do projeto qual o FPS, e assim verificar se o programa desenvolvido está eficiente ou se é necessário melhorá-lo para realizar tarefas em menos tempo.

Para interpretar as entradas do usuário, tem-se a classe “InputHandler”. Ela verifica se um usuário está gerando alguma entrada pelo teclado ou *mouse* do computador e consegue interpretá-la, como por exemplo, qual tecla foi pressionado ou a posição do *mouse*.

Por fim, as classes “Camera” e “FirstPersonCamera”, utilizando a classe “InputHandler”, permitem a manipulação da câmera por meio da entrada fornecida pelo usuário. Assim a câmera pode rotacionar (“Camera”) e transladar (“FirstPersonCamera”) pelo cenário.

Foi desenvolvido um programa para teste. Nele é carregado o modelo de robô industrial e a *skybox* de teste desenvolvidos, e é utilizada a biblioteca XELibrary. Utilizando comandos por teclado é possível movimentar a câmera para aproximar ou afastar e andar em volta do robô, e também movimentar algumas de suas juntas (Figuras 34).

Figura 34: Modelo de testes desenvolvido utilizando o XNA

Limitação dos ângulos máximos

No manual do usuário do Kuka (KUKA) os ângulos máximos definidos pelo *software* do robô para cada junta são:

Junta	Ângulos
A1	± 185^{o}
A2	+115^{o} a -55^{o}
A3	+70^{o} a -210^{o}
A4	± 350^{o}
A5	± 135^{o}
A6	± 350^{o}

Figura 35: Ângulos máximos de cada junta

Estes ângulos máximos foram implementados na lógica para não permitir ângulos fora desta faixa apresentada.

Seleção de opções da interface gráfica

Para interpretar cliques do *mouse* nos botões da tela, utilizou-se a classe MouseState que fornece as coordenadas do clique do *mouse* (em pixels em X e em Y) na tela. Para reconhecer qual dos botões da tela foi escolhido, foram criados

retângulos fictícios (não desenhados na tela, só utilizados na lógica) em volta da área que deseja-se que o botão seja reconhecido e então verifica-se se há intersecção entre os retângulos que representam cada área de cada botão e a posição do clique do mouse. Assim, pode-se reconhecer qual botão foi escolhido. Como o mouse também é utilizado para rotacionar a câmera na tela, foi definida na biblioteca que trata da câmera a área na qual ela faz efeito, retirando as áreas com os botões e imagens bidimensionais da área de atuação do movimento da câmera por meio do *mouse*.

Colisão

A detecção de colisão consiste em verificar continuamente se os objetos estão colidindo com outros objetos. Existem vários tipos de algoritmos para se fazer essa verificação, estes podem ser simples ou complexos e podem diminuir a taxa de frames por segundo (Carter, 2008) apresentada anteriormente.

Para o projeto foram escolhidos métodos que considerassem que os modelos utilizados são tridimensionais e que a colisão de interesse a ser identificada pelo *software* seria entre a ponta da ferramenta do robô e a mesa presente na cena. Estes métodos são baseados em criar volumes fictícios ao redor dos objetos considerados e analisar se há intersecção entre estes volumes. Os volumes fictícios escolhidos tem a forma de uma caixa para a mesa, e de uma esfera para a ponta da ferramenta.

Para cada objeto do modelo importado, o XNA cria uma *Bounding Sphere* que é uma esfera definida pela posição de seu centro e seu raio. Utilizou-se esta esfera para envolver a ferramenta, porém como o modelo se movimenta no espaço, para se utilizar essa esfera, ela precisa ser movimentada junto com o objeto, ou seja, ter sua posição atualizada conforme a movimentação do robô.

Considerando a geometria da mesa, conclui-se que uma esfera ao redor da mesa identificaria colisões muito distantes da peça real, como pode ser melhor compreendido por meio da representação da Figura 36

Portanto, para uma melhor detecção da colisão, definiu-se para a mesa uma *bounding box* que é similar a *bounding sphere*, porém o volume fictício tem o formato de uma caixa ao redor do objeto (Figura 37).

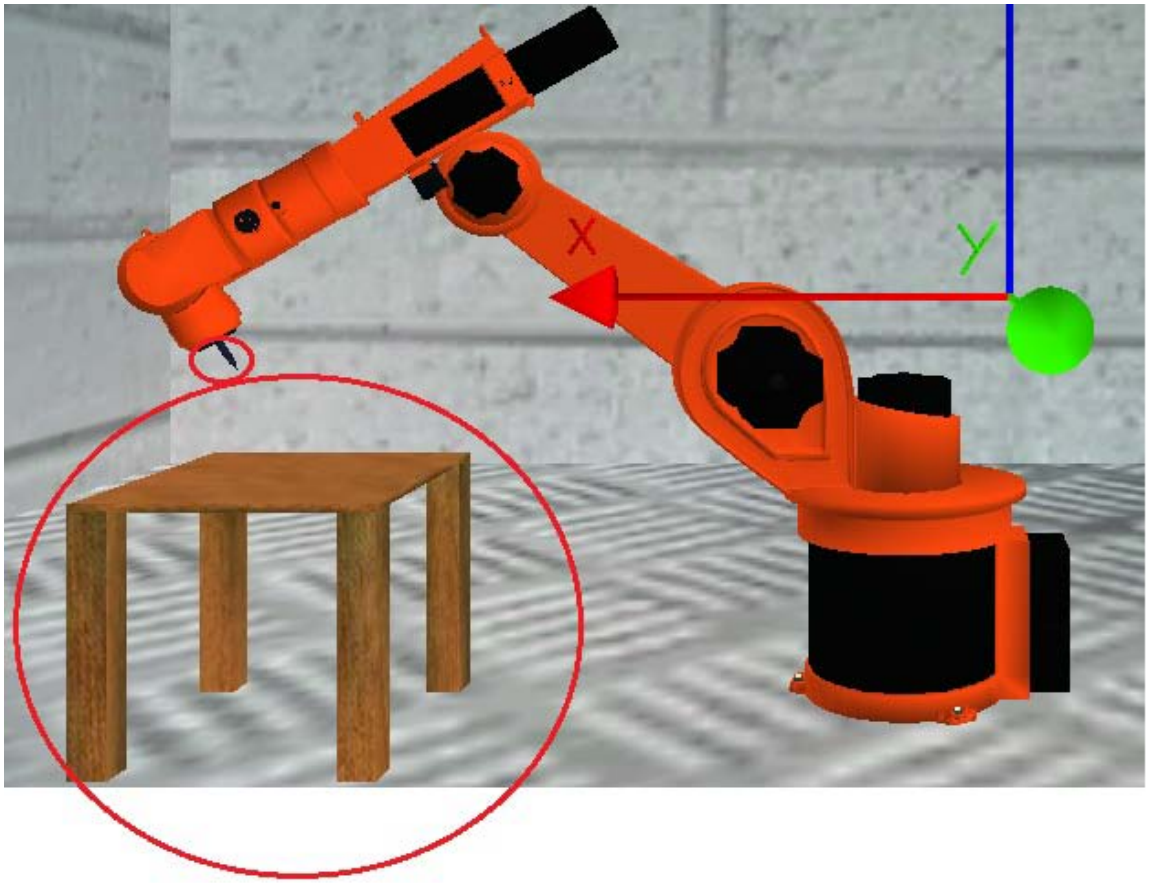

Figura 36: Detecção de colisão por meio de duas *bounding spheres*

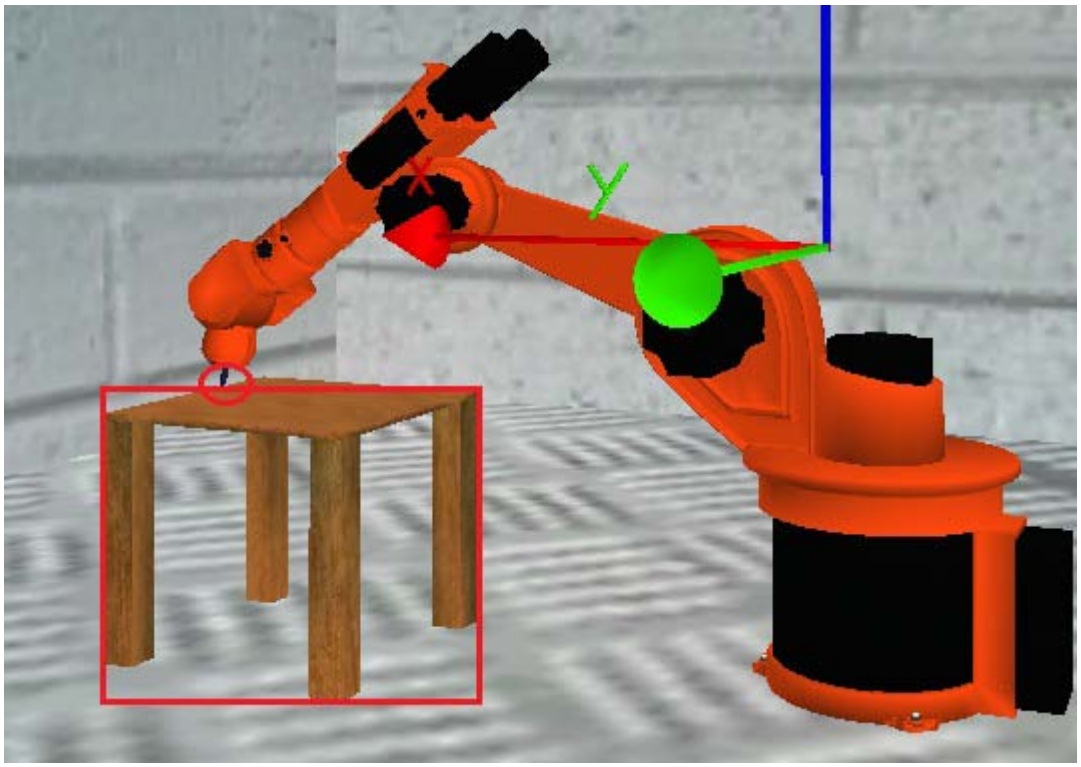

Figura 37: Detecção de colisão por meio de uma *bounding box* e uma *bounding sphere*

Após a identificação da colisão, implementou-se a execução de um som de colisão e uma lógica para que o robô parasse sua movimentação de modo a não atravessar a mesa.

Áudio

Tanto no modo manual quanto no automático, ao se movimentar o robô, o som de um motor importado no XNA é reproduzido. Ao parar o robô, o som também para. Um som também é reproduzido caso seja detectada uma colisão. Ele é reproduzido somente uma vez, representando um único choque. Se forem forçadas outras colisões, o som é reproduzido novamente.

Cinemática Direta

Para os cálculos de cinemática direta (encontrar posição do efetuador em função aos ângulos de cada junta), poderiam ser utilizadas a notação de Denavit-Hartenberg e matrizes de transformação homogênea (Siciliano *et al.*, 2009). No entanto, para evitar cálculos desnecessários, foi utilizada uma funcionalidade do próprio XNA, que permite encontrar a posição de qualquer osso presente no modelo. Como um dos ossos está na ponta da ferramenta, sabendo a posição dele, tem-se a posição da ferramenta.

Cinemática Inversa

Além da cinemática direta, o cálculo de cinemática inversa também é muito importante em robôs industriais. Este consiste em, a partir da posição e orientação do efetuador do robô, encontrar o ângulo de cada uma de suas juntas.

Devido à presença de punho esférico no robô modelado (três articulações mais próximas do efetuador), é possível resolver o problema do cálculo da cinemática inversa graficamente, utilizando um desacoplamento de posição e orientação, no qual a posição do efetuador permite encontrar os ângulos das três primeiras juntas, e sua orientação, os das três últimas (Siciliano *et al.*, 2009).

Na Figura 38, tem-se uma vista superior esquemática do robô, com a qual é possível encontrar o ângulo da junta A1 (θ_1).

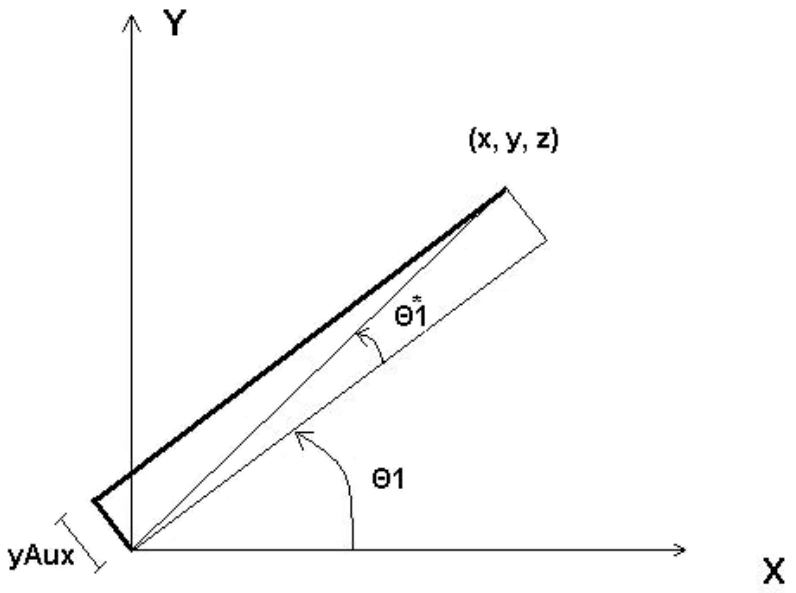

Figura 38: Vista superior esquemática do robô para cálculo de $\boldsymbol{\theta_1}$

Assim,

$$sen(\theta_1^*) = \frac{yAux}{\sqrt{x^2 + y^2}} \tag{1}$$

$$\theta_1 = atan2(y, x) - \theta_1^* \tag{2}$$

onde atan2(y, x) é definida por:

$$\operatorname{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

Nas Figuras 39 e 40, tem-se vistas no plano Zr, sendo r na direção da projeção dos ligamentos 2 e 3 do robô no plano XY. Com estas vistas, é possível encontrar o ângulo das juntas A2 (θ_2) e A3 (θ_3).

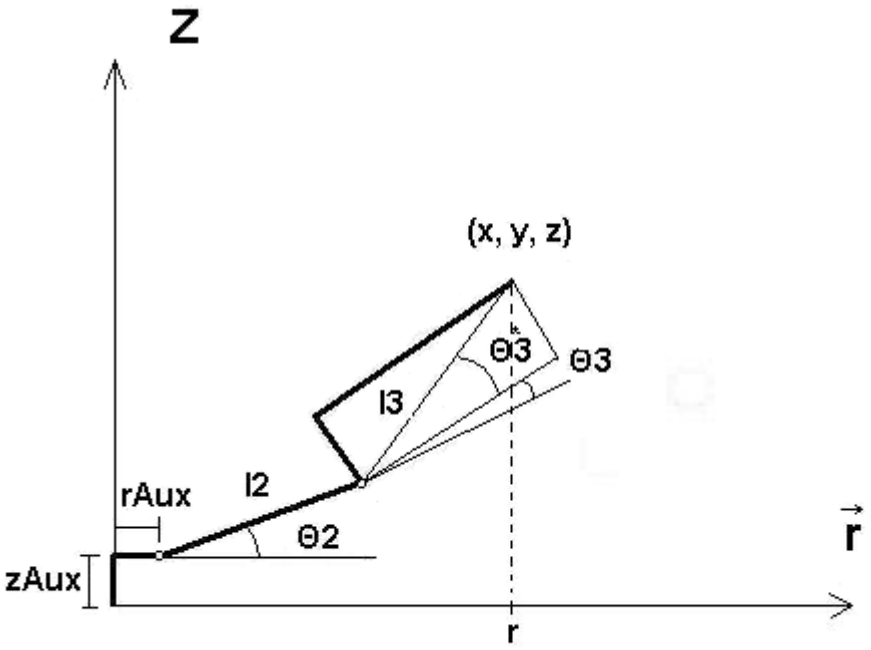

Figura 39: Vista lateral do robô para cálculo de $\boldsymbol{\theta_3}$

A partir da Figura 39, tem-se da lei dos cossenos:

$$(r - rAux)^2 + (z - zAux)^2 = {l_2}^2 + {l_3}^2 + 2l_2 l_3 \cos(\theta_3 + \theta_3^*) \tag{3}$$

Isolando-se θ_3 na equação (3), chega-se em:

$$\theta_3 = \arccos\left(\frac{(r - rAux)^2 + (z - zAux)^2 - {l_2}^2 - {l_3}^2}{2l_2 l_3}\right) - \theta_3^* \tag{4}$$

Finalmente, da Figura 40, pode-se encontrar:

$$\propto = atan2(z - zAuz, r - rAux) \tag{5}$$

$$\gamma = atan2(l_3 sen(\theta_3 + \theta_3^*), l_2 + l_3 cos(\theta_3 + \theta_3^*)) \tag{6}$$

A partir dos quais se calcula θ_2:

$$\theta_2 = \alpha - \gamma \tag{7}$$

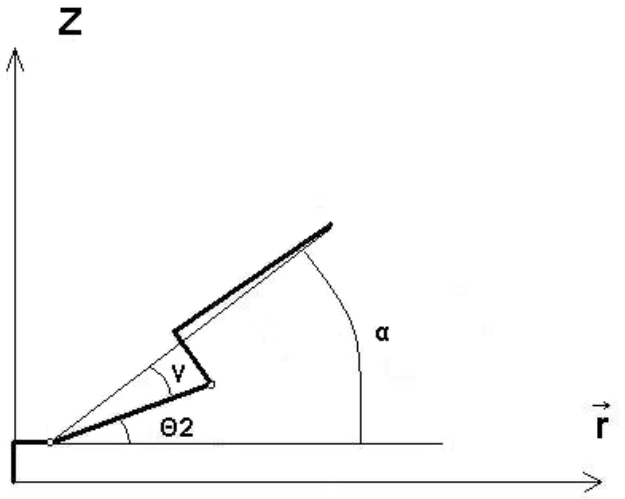

Figura 40: Vista lateral do robô para cálculo de $\boldsymbol{\theta_2}$

Assim, pode-se saber qual o ângulo das juntas do robô para uma dada posição.

ANTLR

Uma funcionalidade do projeto é a interpretação de comandos em KRL (Kuka *Robot Language*). Essa linguagem foi criada pelo fabricante e tem sua própria síntaxe e semântica. Para interpretá-la, foram utilizados recursos obtidos do ANTLR (*Another Tool for Language Recognition)* que é uma ferramenta que auxilia na construção de, por exemplo, interpretadores e compiladores a partir da descrição gramatical de linguagens para ser utilizada em outras linguagens diferentes (ANTLR, 2011).

Para utilizar essa ferramenta, foi utilizado o software disponível no site do ANTLR (ANTLR, 2011), o ANTLRWorks versão 1.43. Nele foi importada a gramática do KRL, disponível no próprio site do ANTLR, e foi gerado o código escrito em C# para interpretar o KRL. Entretanto, houve alguns problemas, pois alguns dos comandos gerados estavam na linguagem Java (padrão do ANTLRWorks), e não C#, e tiveram que ser alterados manualmente para os comandos correspondentes. O ANTLRWorks gerou dois arquivos: KRLParser e KRLLexer.

Então foi criada uma classe KRLManager, para interpretar comandos em KRL. Como a quantidade de comandos KRL existentes é extensa, apenas comandos básicos foram implementados. Também é possível verificar se há erros no código de programação que não compilariam no robô real.

Interface Gráfica

Ao executar o *software*, a primeira tela apresentada pode ser visualizada (Figura 41). Nela pode-se decidir iniciar a utilização do mesmo ou sair (fecha o *software*). Para sair, pode-se sempre que desejado também utilizar a tecla “ESC”.

Após selecionar “iniciar”, apresenta-se o ambiente com o robô (Figura 42). Nesta tela, apresenta-se o modo manual, e pode-se visualizar o modelo do robô, o modelo da mesa, o modelo dos eixos, e o modelo do cenário, um *menu* na parte superior esquerda e outro na parte da direita. Ao rotacionar e transladar a câmera, todos os modelos também se movimentam de modo tridimensional. A câmera pode ser movimentada pelo teclado: teclas A (esquerda), S (afastar), D (direita), W (aproximar), setas para cima, lados e para baixo (rotacionam).

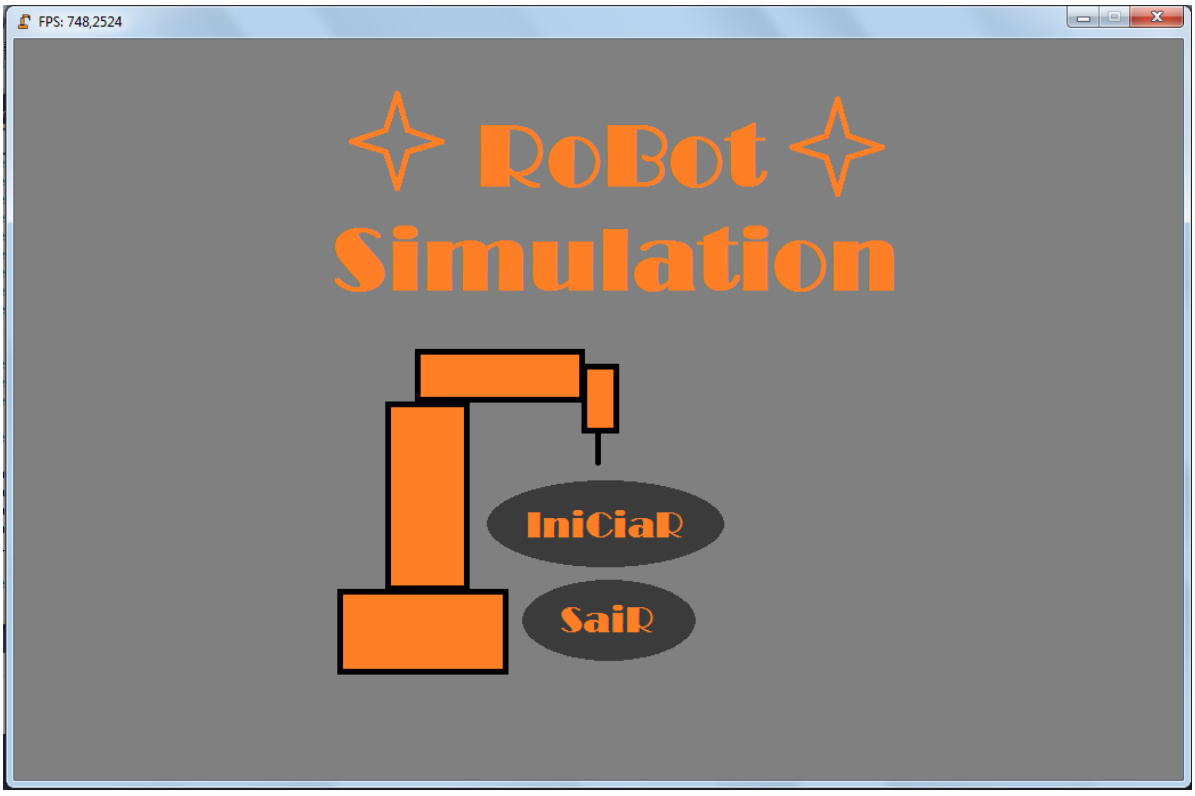

Figura 41: Tela inicial do software

Como definido nos requisitos do projeto, o robô pode ser movimentado por meio do modo manual ou do automático. Implementou-se portanto a opção de escolha desse modo por meio da seleção nas opções do *menu* visível no canto superior esquerdo. Ao clicar no círculo ao lado da opção Manual, este fica selecionado (estado representado pela circunferência preenchida em laranja), e um *menu* é desenhado do lado direito da tela, contendo três áreas (Figura 43).

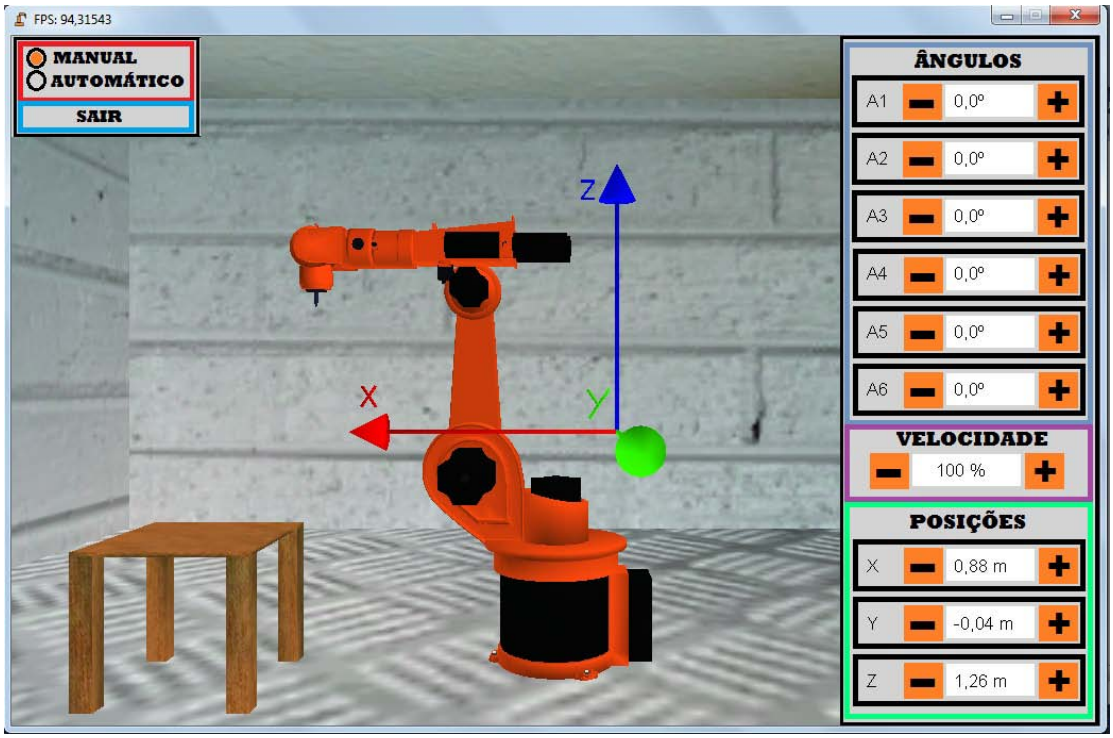

Figura 42: Tela com modelo e menus

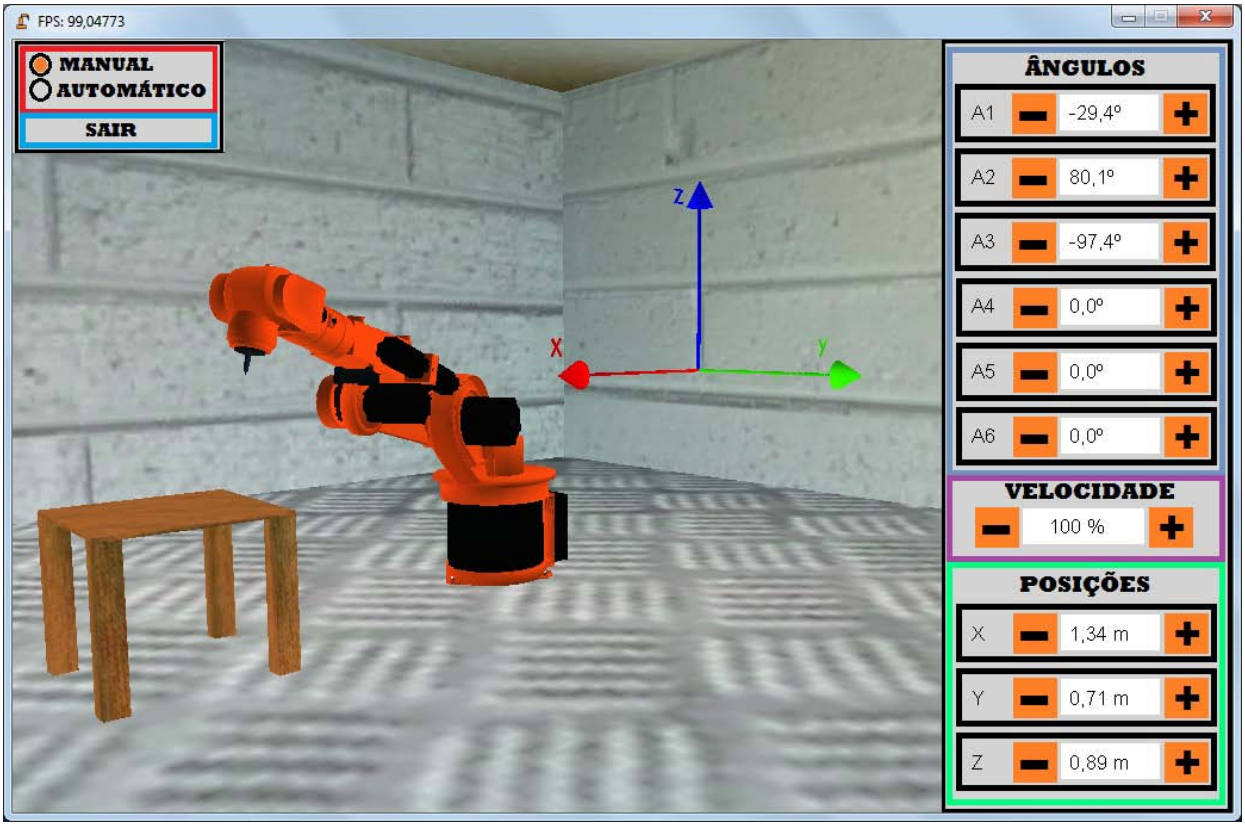

Figura 43: Tela em modo Manual

A primeira área (superior) permite a visualização e alteração dos ângulos das juntas dos robôs. As juntas são numeradas e tem suas direções definidas, assim como no robô real, de A1 a A6 como pode ser visualizado na Figura 44. O ângulo zero de cada junta é definido na posição do robô assim como mostrado na Figura 42.

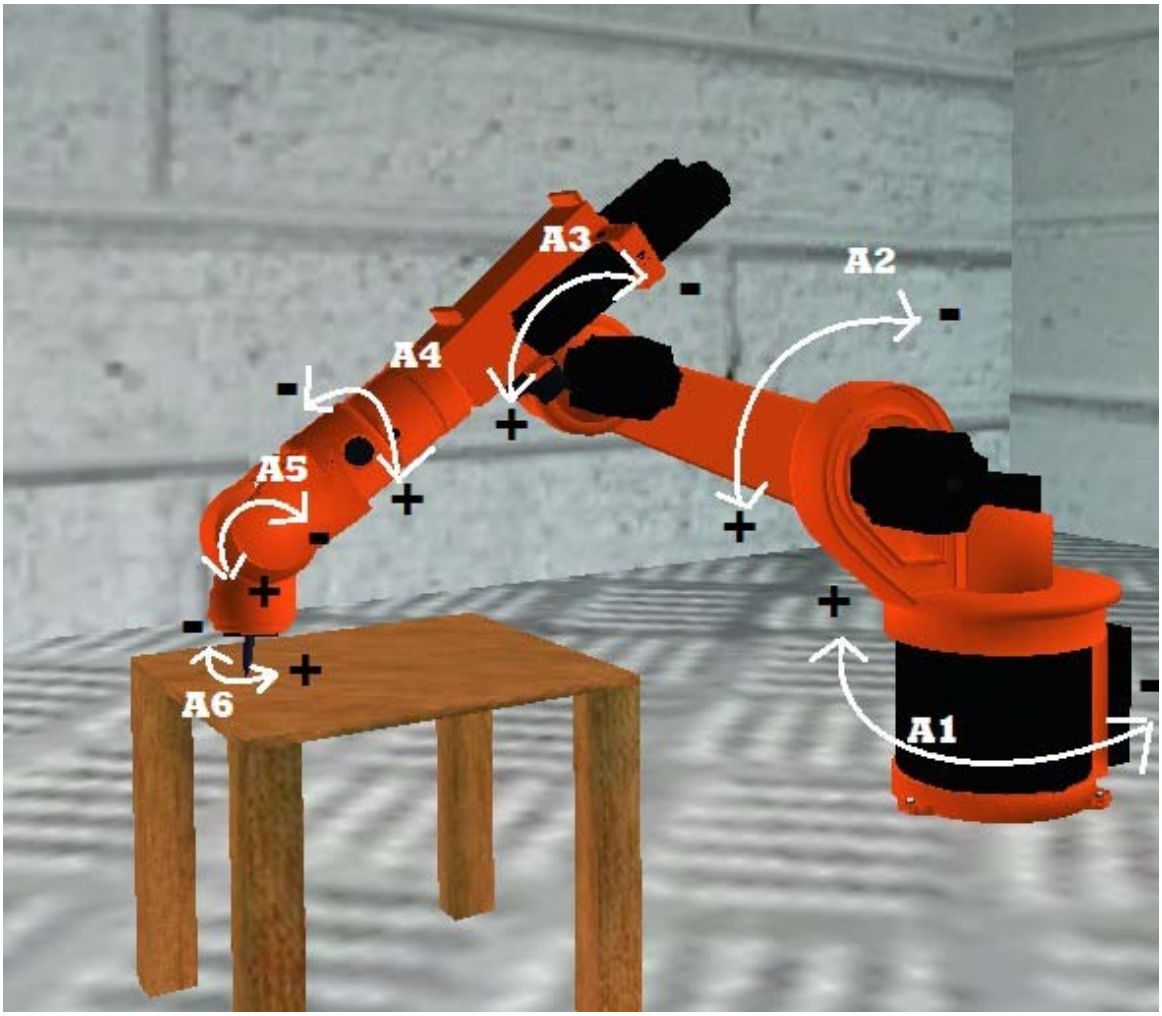

Figura 44: Representação da nomenclatura e sentido dos ângulos

Para alterar os ângulos e movimentar o robô pode-se clicar nos botões representados pelo sinal de mais (sentido positivo do ângulo) e pelo sinal de menos (sentido negativo do ângulo). Pode-se também utilizar comandos do teclado para a movimentação, primeiro seleciona-se a junta pressionando teclas de 1 a 6 e depois pressionando a tecla “Z” para o sentido positivo ou a tecla “X” para o sentindo negativo. Se for selecionada a movimentação tanto por meio do teclado quanto por meio dos botões da tela, o botão em questão na tela fica selecionado por meio do desenho de uma borda preta ao redor do botão, como pode ser visualizado no

detalhe da Figura 45, onde o sentido positivo da junta A1 está sendo selecionado pelo teclado.

A segunda área (central) possibilita a escolha de diferentes velocidades para a movimentação do robô. Assim como no robô real, ela é apresentada em valores percentuais. O *software* não possui correspondência com a velocidade do robô real, porém foram definidas velocidades percentuais de 10 a 100 por cento para permitir visualização de movimentos mais lentos ou mais rápidos na tela. A velocidade pode ser aumentada (sinal de mais) ou reduzida (sinal de menos) por meio dos botões na tela.

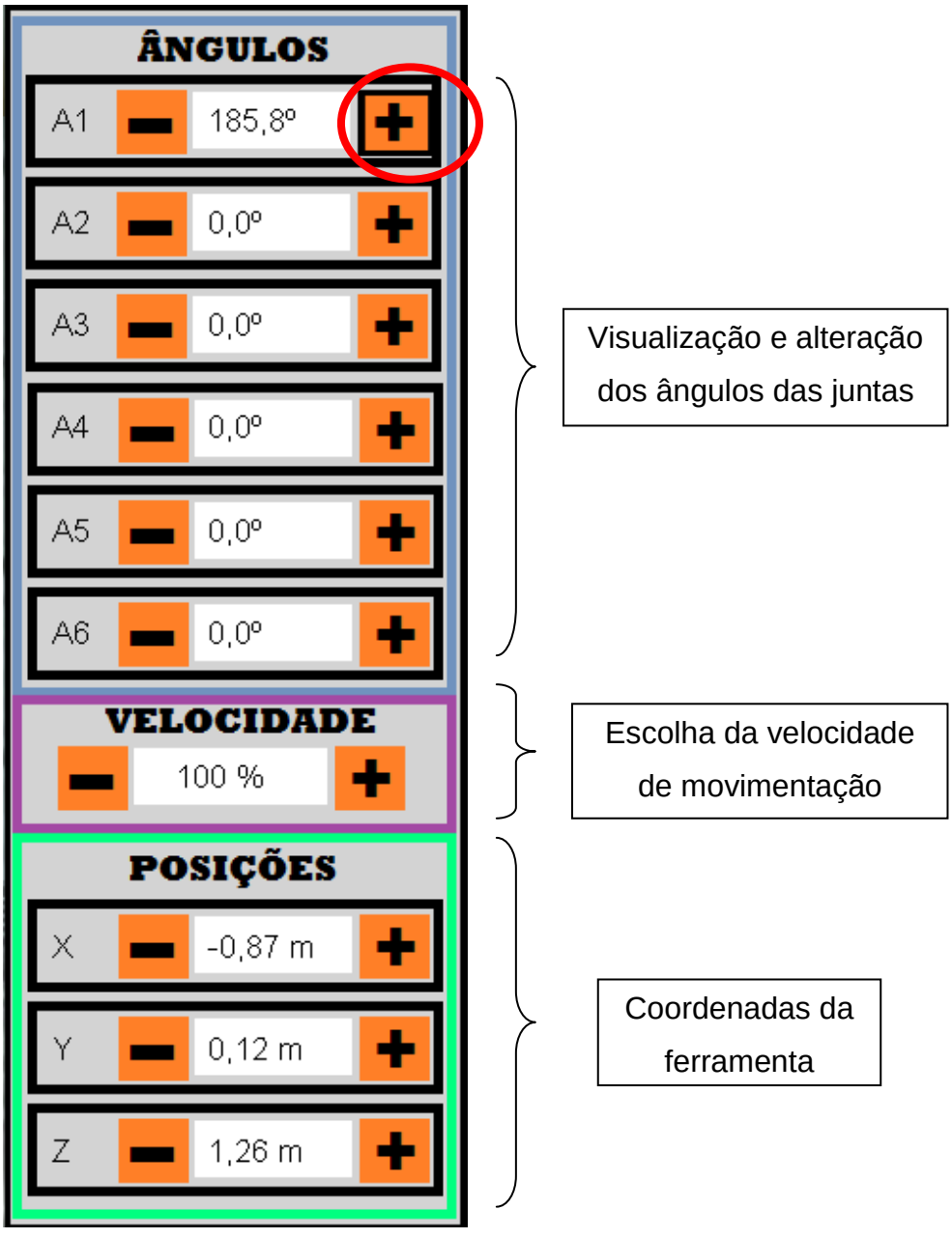

Figura 45: *Menu* do modo manual

A terceira área apresenta a posição da ponta do efetuador em coordenadas X, Y, Z conforme o sistema de coordenadas definido na tela. Ao pressionar os botões na tela, pode-se movimentar o robô por meio das posições desejadas do efetuador. Para esta movimentação foi implementada a cinemática inversa da posição do robô.

Ao clicar no círculo ao lado da opção Automático, este fica selecionado. Neste modo é apresentado, no lado direito da tela, um novo menu com três áreas (Figura 46).

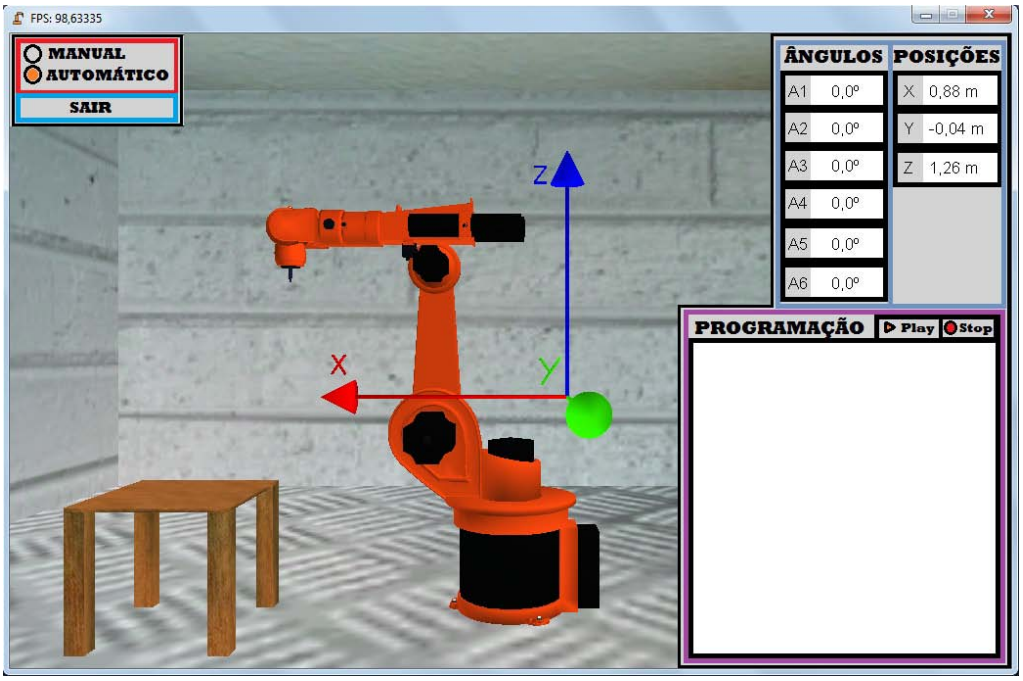

Figura 46: Tela do modo automático

Uma das áreas apresenta os ângulos das juntas e outra a posição do efetuador da mesma maneira que em modo manual, porém sem a opção dos botões de movimentação. Neste modo, porém, visualiza-se outra área destinada a programação. Nesta área pode-se pressionar o botão de *Play* (reproduzir) ou de *Stop* (interromper reprodução). Ao optar pela reprodução, o *software* movimenta o robô de forma automática conforme o código em KRL previamente carregado pelo usuário em um arquivo de texto salvo na pasta do programa e apresenta tal código na tela do *software*. Ao optar por interromper a reprodução, o *software* para a movimentação automática do robô. Também foi necessário a utilização da lógica da cinemática inversa implementada para estas movimentações.

5 – CONCLUSÃO

O projeto desenvolvido pode auxiliar as indústrias a atender a exigência atual por qualidade, produtividade e baixo custo, uma vez que teve como foco a modelagem de um braço robótico, cada vez mais utilizado em processos de manufatura.

O ambiente virtual criado simula os movimentos de um braço robótico com uma interface de fácil utilização pelo usuário, para que este possa entender o funcionamento do robô e realizar testes com o mesmo sem utilizar uma máquina real, não interrompendo seu funcionamento no chão de fábrica, o que economiza tempo e reduz gastos com material.

O *software* tem como principais vantagens a detecção de colisão da ferramenta do robô com outro objeto presente no ambiente, a verificação da movimentação do robô utilizando tanto a cinemática direta quanto a cinemática inversa, e a verificação de código de programação em linguagem KRL. Assim é possível realizar testes tanto em modo manual quanto automático, com uma visualização em tempo real dos valores dos ângulos das juntas e posição da ponta da ferramenta dentro de um ambiente que permite diferentes ângulos e posições de visualização pelo usuário, por meio de comandos do teclado e *mouse*.

Verificou-se que o Blender e o Microsoft XNA são excelentes ferramentas que podem ser integradas para o desenvolvimento de projetos de simulação gráfica 3D.

5.1 – Pontos a desenvolver

Mesmo com os resultados satisfatórios do trabalho, alguns pontos ainda podem ser melhorados, de modo a se obter um *software* que permita uma simulação mais real de um robô industrial e que permita tirar conclusões mais concretas sobre sua utilização.

Um ponto de melhoria seria aumentar a quantidade de comandos na linguagem KRL que são interpretados e executados. Não apenas comandos de movimentação, o KRL também define comandos de lógica, como criação de *loops,* e situações condicionais, que são bastante úteis e muito utilizadas.

Também pode ser feita uma melhora no algoritmo de detecção de colisão, detectando colisões não somente da ponta da ferramenta, mas de qualquer um dos objetos que compõe o robô.

No caso de reproduções de códigos, também poderia ser disponibilizado o tempo gasto para se executar um programa, para se ter uma ideia se o processo realizado está lento ou não.

Além de todas estas funcionalidades, também pode-se fazer um estudo mais aprofundado da dinâmica e velocidades reais do Kuka para se obter um resultado mais preciso dos movimentos do robô.

6 – REFERÊNCIAS BIBLIOGRÁFICAS

ANTLR; Disponível em <www.antlr.org>.
Último acesso em 16 nov 2011

Banerjee P.; Zetu D. Virtual Manufacturing. United States of America: John Wiley and Sons, Inc., 2001. 321 p.

Bekey, G. et al. Robotics: state of the art and future challenges. London: Imperial College Press, 2008. 144 p.

Blender Features; Disponível em <http://www.blender.org/features-gallery/features/>. Último acesso 18 abr 2011.

Booch, G.; Jacobson, I.; Rumbaugh, J. The Unified Modeling Language User Guide.USA: Addison-Wesley, 1999.

Carter C. Microsoft XNA Unleashed: Graphics and Game Programming for Xbox 360 and Windows. United States of America: Sams, 2008.

Cates, C. U.; Virtual Reality Simulation in Carotid Stenting: A New Paradigm for Procedural Training in Nature Publishing Group, 2007.

Chan, C. L. F.; Ngai E. K. Y.; Leung, P. K. H.; Wong, S.; Effect of the Adapted Virtual Reality Cognitive Training Program among Chinese Older Adults with Chronic Schizophrenia: A Pilot Study in: Wiley InterScience, 2009.

Dépincé *et al*. The Virtual Manufacturing Concept: Scope, Socio-economic aspects and future trends in ASME 2004 Design Engineering Technical Conferences and Computers and Information in Engineering Conference. United States of America: 2004.

Flavell, L. Beginning Blender: Open Source 3D Modeling, Animation, and Game Design. United States of America: Apress, 2010. 449 p.

Fu, F.S.; Gonzalez, R. C., Lee, C.S.G.; Robotics: Control, Sensing, Vision, and Intelligence. Singapore: McGraw_Hill, 1987. 589 p.

Gumster, J. V. Blender for Dummies. Wiley Publishing, Inc.: United States of America, 2009. 412 p.

Gutiérrez, M. A.; Vexo, F.; Thalmann, D.; Stepping into Virtual Reality. London: Springer, 2008

Guttentag, D. A.; Virtual Reality: Applications and Implications for Tourism, in: Tourism Management 31, 2010.

Iwata K; Onosato M.; Teramoto K; Osaki S. A Modelling and Simulation Architecture for Virtual Manufacturing Systems, 1995.

Jaegers K. XNA 4.0 Game Development by Example – Beginner's Guide. United Kingdom: Packt Publishing, 2010. 428 p.

Kadir, A. A.; Xu, X.; Hammerle, E.; Virtual Machine Tools and Virtual Machining – A Technological Review.

Kim, Y. S.; Yang, J.; Han, S. A multichannel visualization module for virtual manufacturing. 2006.

KUKA. Robô KR15 – Dados técnicos

Lawrence Associates Inc. Virtual Manufacturing user workshop. Technical report. Ohio, 1994. Disponível em <http://www.isr.umd.edu/Labs/CIM/vm/lai2/tfinal7.html>. Último acesso em 22 jun 2011.

Laboratory for Systems Theory and Automatic Control – 3D online simulation of a 7-DOF robotic manipulator. Disponível em <http://ifatwww.et.uni-magdeburg.de/syst/education/projects/pro_desc/lwr_3donline_sim.shtml> . Último acesso em 15 jun 2011.

Li, S.; Sun, J.; Application of Virtual Reality Technology in the Field of Sports, in: 2009 First International Workshop on Education Technology and Computer Science

Liang, J.; Generation of a Virtual Reality-Based Automotive Driving Training System for CAD Education, in: Wiley Periodicals, Inc Computer Applications in Engineering Education, 2008.

Marinov V., S. Seetharamu. Virtual machining operation: a concept and an example. In Proceeding of the SPIE Conference on Intelligent Systems in Design and Manufacturing, 25-26 October 2004, Philadelphia, PA, Proc. of SPIE Vol. #5605, 2004.

Milgram, P.; Takemura, H.; Utsumi, A.; Kishino, F.; Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum, in: Telemanipulator and Telepresence Technologies, 1994.

MSC Software Corporation - Virtual Manufacturing: The Next Revolution on Global Manufacturing, 2001. Disponível em < http://www.mscsoftware.com/assets/1776_vm_2001.pdf>. Último acesso em 15 jun 2011.

Netto *et al.* Realidade virtual e suas aplicações na área de manufatura, treinamento, simulação e desenvolvimento de produto. Gestão e Produto. São Carlos, 1998.

Potkonjak, V.; Vukobratovic, M.; Jovanovic, K.; Medenica, M.; Virtual Mechatronic/Robotic Laboratory – A Step Further in Distance Learning, in: Computers & Education 55, 2010.

Seymour, N. E.; Gallagher, A. G.; Roman, S. A.; O'Brien, M. K.; Bansal, V. K.; Andersen, D. K.; Satava, R. M.; Virtual Reality Training Improves Operating Room Performance.

Siciliano B.; Sciavicco L.; Vilani L.; Oriolo G. Robotics: Modelling, Planning and Control. London: Springer, 2009. 632 p.

Thompson, C. W.; Next Generation Virtual Worlds: Architecture, Status and Directions, in: IEEE Internet Computing, Jan/Feb 2011.

UML Diagramas Org. Disponível em <http://www.uml-diagrams.org/uml-23-diagrams.html1>. Último acesso em 25 mai 2011.

ANEXO

Arquivo Program.cs

```
using System;

using System.Windows.Forms;
using System.Drawing;

namespace KukaSimulation
{
#if WINDOWS || XBOX
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
#endif
}
```

Arquivo Game1.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using XELibrary; // Usando a biblioteca XELibrary

namespace KukaSimulation
{
    /// <summary>
    /// Esta e' a classe principal do programa
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        #region Definicao de Variaveis
        // Uso de objetos das classes da biblioteca XELibrary
        private FPS fps; // Calculo de FPS
        private FirstPersonCamera camera; // Movimentacao da camera
        private InputHandler input; // Interpretacao de comandos
```

```
// Objetos que conterao os modelos
private Model model; // Modelo do robo
private Model modelSkybox; // Modelo da skybox
private Model modelMesa; // Modelo da mesa
private Model modelEixos; //Modelo dos eixos
private BoundingBox boundingBoxMesa; //Bounding box da mesa
private Vector3 boundingSphereFerramenta; // Bounding sphere da ferramenta

// Matrizes contendo as transformacoes no modelo
Matrix modelTransform;
Matrix modelTransform1; // Primeira junta do robo
Matrix modelTransform2; // Segunda junta do robo
Matrix modelTransform3; // Terceira junta do robo
Matrix modelTransform4;
Matrix modelTransform5;
Matrix modelTransform6;
Matrix modelTransformSkybox; // Transformacao da skybox
Matrix modelTransformMesa;
Matrix modelTransformEixos;
Matrix[] transform; // Transformacao total do robo
Matrix[] boneOriginal; // Posicao original do robo
Matrix[] transformSkybox; // Transformacao total da skybox
Matrix[] boneOriginalSkybox; // Posicao original da skybox
Matrix[] transformMesa;
Matrix[] boneOriginalMesa;
Matrix[] transformEixos;
Matrix[] boneOriginalEixos;
Matrix position;
Matrix positionIK;

// Indices contendo o ID dos ossos
int bone;
int boneID;
int boneID1;
int boneID2;
int boneID3;
int boneID4;
int boneID5;
int boneID6;
int boneIDSkybox;
int boneIDMesa;
int boneIDEixos;

// Fonte do texto
SpriteFont font;

// Variaveis auxiliares para movimentacao do robo
int joint = 1; // Para selecao da junta
double velocidade = 1; // Variavel com o valor da velocidade
string vel; // Variavel para impressao da velocidade na tela
float[] angulo =  new float [6]; // Angulos das juntas
string[] angulostring = new string[6]; // Variavel para impressao das juntas na tela
bool collision = false; // Para deteccao de colisao em cinematica inversa
int minAngle; // Minimo angulo para movimentacao de cada junta
int maxAngle; // Maximo angulo para movimentacao de cada junta
// Identificar cliques
bool xplus = false;
bool xminus = false;
bool yplus = false;
```

```
bool yminus = false;
bool zplus = false;
bool zminus = false;
int xm;
int ap = 0;
int am = 0;
int[] chaves = new int[6];
int[] leds = new int[6];
bool[] junta = new bool[6];
bool positivo = false;
bool negativo = false;

// Para a tela inicial
private Texture2D startButton; // Botao de iniciar
private Texture2D exitButton; // Botao de sair
private Texture2D logo; // Logo inicial
private Texture2D barralateral; // Barra lateral direita modo Manual
private Texture2D kukinha; // Simbolo do kuka
private Texture2D but; // Retangulo de selecao dos botoes
private Texture2D barralateralesq; // Barra lateral esquerda
private Texture2D bolinhaM; // Selecao do modo automatico
private Texture2D bolinhaA; // Selecao do modo manual
private Texture2D barralateralA; // Barra lateral direita modo Automatico
private Texture2D led; // Leds apagados
private Texture2D ledA; // Leds acesos
private Texture2D chave0; // chave em zero
private Texture2D chave1; // chave em 1
// Posicoes na tela
private Vector2 startButtonPosition;
private Vector2 exitButtonPosition;
private Vector2[] angulosPosition = new Vector2[6];
private Vector2[] asPosition = new Vector2[6];
private Vector2[] angulosPositionA = new Vector2[6];
private Vector2[] asPositionA = new Vector2[6];

// Status do mouse
MouseState mouseState;
MouseState previousMouseState;
// Status do jogo (Playing/ Menu inicial)
GameState gameState;
GameState previousGameState;
// Modo manual automatico
GameMode gameMode;

// Variaveis para efeitos sonoros
SoundEffect hitTable; // Colisao com a mesa
SoundEffectInstance hitTableInstance;
SoundEffect soundEngine; // Motores
SoundEffectInstance soundEngineInstance;

// Areas para deteccao de clique
Rectangle a1PlusRect = new Rectangle(974, 54, 31, 26);
Rectangle a1PlusRectPress = new Rectangle(966, 45, 40, 35);
Rectangle a2PlusRect = new Rectangle(974, 108, 31, 26);
Rectangle a2PlusRectPress = new Rectangle(966, 99, 40, 35);
Rectangle a3PlusRect = new Rectangle(974, 162, 31, 26);
Rectangle a3PlusRectPress = new Rectangle(966, 153, 40, 35);
Rectangle a4PlusRect = new Rectangle(974, 216, 31, 26);
Rectangle a4PlusRectPress = new Rectangle(966, 207, 40, 35);
Rectangle a5PlusRect = new Rectangle(974, 271, 31, 26);
```

```
Rectangle a5PlusRectPress = new Rectangle(966, 262, 40, 35);
Rectangle a6PlusRect = new Rectangle(974, 325, 31, 26);
Rectangle a6PlusRectPress = new Rectangle(966, 316, 40, 35);
Rectangle a1MinusRect = new Rectangle(849, 54, 31, 26);
Rectangle a1MinusRectPress = new Rectangle(838, 45, 40, 35);
Rectangle a2MinusRect = new Rectangle(849, 108, 31, 26);
Rectangle a2MinusRectPress = new Rectangle(838, 99, 40, 35);
Rectangle a3MinusRect = new Rectangle(849, 162, 31, 26);
Rectangle a3MinusRectPress = new Rectangle(838, 153, 40, 35);
Rectangle a4MinusRect = new Rectangle(849, 216, 31, 26);
Rectangle a4MinusRectPress = new Rectangle(838, 207, 40, 35);
Rectangle a5MinusRect = new Rectangle(849, 271, 31, 26);
Rectangle a5MinusRectPress = new Rectangle(838, 262, 40, 35);
Rectangle a6MinusRect = new Rectangle(849, 325, 31, 26);
Rectangle a6MinusRectPress = new Rectangle(838, 316, 40, 35);
Rectangle VelMinusRect = new Rectangle(812, 403, 31, 26);
Rectangle VelMinusRectPress = new Rectangle(805, 391, 40, 35);
Rectangle VelPlusRect = new Rectangle(960, 403, 31, 26);
Rectangle VelPlusRectPress = new Rectangle(953, 391, 40, 35);
Rectangle xPlusRect = new Rectangle(974, 490, 31, 26);
Rectangle yPlusRect = new Rectangle(974, 544, 31, 26);
Rectangle zPlusRect = new Rectangle(974, 598, 31, 26);
Rectangle xMinusRect = new Rectangle(849, 490, 31, 26);
Rectangle yMinusRect = new Rectangle(849, 544, 31, 26);
Rectangle zMinusRect = new Rectangle(849, 598, 31, 26);
Rectangle manualRect = new Rectangle(23, 18, 10, 15);
Rectangle automaticRect = new Rectangle(23, 38, 10, 15);
Rectangle sairRect = new Rectangle(11, 68, 100, 20);
Rectangle loadRect = new Rectangle(889, 292, 60, 18);
Rectangle chave10 = new Rectangle(920, 246, 5, 13);
Rectangle chave11 = new Rectangle(920, 230, 5, 13);
Rectangle chave20 = new Rectangle(940, 246, 5, 13);
Rectangle chave21 = new Rectangle(940, 230, 5, 13);
Rectangle chave30 = new Rectangle(960, 246, 5, 13);
Rectangle chave31 = new Rectangle(960, 230, 5, 13);
Rectangle chave40 = new Rectangle(980, 246, 5, 13);
Rectangle chave41 = new Rectangle(980, 230, 5, 13);
Rectangle chave50 = new Rectangle(1000, 246, 5, 13);
Rectangle chave51 = new Rectangle(1000, 230, 5, 13);
Rectangle xPlusRectPress = new Rectangle(966, 480, 40, 35);
Rectangle yPlusRectPress = new Rectangle(966, 534, 40, 35);
Rectangle zPlusRectPress = new Rectangle(966, 588, 40, 35);
Rectangle xMinusRectPress = new Rectangle(838, 480, 40, 35);
Rectangle yMinusRectPress = new Rectangle(838, 534, 40, 35);
Rectangle zMinusRectPress = new Rectangle(838, 588, 40, 35);

// Objeto para interpretar krl
KRLManager krl = new KRLManager();

// Variaveis auxiliares para cinematica inversa
double theta1 = 0;
double theta2 = 0;
double theta3 = 0;
double theta1Aux = 0;
double theta3Aux = 0;
double yAux = 0;
double rAux = 0;
double zAux = 0;
double r;
double l2;
```

```
        double l3;
        double z;
        double alfa;
        double gama;
        double deltaAngle;
        double auxX;
        double auxY;
        double auxZ;
        int indexAux;
        bool interpret;
        int numNode = 0;

        // Alternar entre a tela inicial e a tela de movimentacao do kuka
        enum GameState
        {
            StartMenu,
            Playing
        }

        // Alternar entre modo manual e automatico
        enum GameMode
        {
            Manual,
            Automatic
        }

        #endregion

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content = new ContentManager(Services);
            Content.RootDirectory = "Content";

            // Adicao dos componentes da biblioteca XELibrary
            input = new InputHandler(this);
            Components.Add(input);
            camera = new FirstPersonCamera(this);
            Components.Add(camera);
#if DEBUG
            fps = new FPS(this, true, false); // Desenha na tela com o max de FPS possivel para verificar o
desempenho
#else
            fps = new FPS(this, true, false); // Desenha com 60 FPS
#endif
            Components.Add(fps);
            this.Window.Title = "Kuka Robot Simulation"; // titulo na barra superior
            // tamanho da tela
            graphics.PreferredBackBufferWidth = 1024;
            graphics.PreferredBackBufferHeight = 648;
            graphics.ApplyChanges();
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content.  Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
```

```
{

    // Defininido a posicao dos objetos 2D da tela
    startButtonPosition = new Vector2((GraphicsDevice.Viewport.Width / 2) - 100, 380);
    exitButtonPosition = new Vector2((GraphicsDevice.Viewport.Width / 2) - 70 , 470);
    angulosPosition[0] = new Vector2(887, 51);
    angulosPosition[1] = new Vector2(887, 105);
    angulosPosition[2] = new Vector2(887, 159);
    angulosPosition[3] = new Vector2(887, 213);
    angulosPosition[4] = new Vector2(887, 268);
    angulosPosition[5] = new Vector2(887, 322);
    asPosition[0] = new Vector2(805, 51);
    asPosition[1] = new Vector2(805, 105);
    asPosition[2] = new Vector2(805, 159);
    asPosition[3] = new Vector2(805, 213);
    asPosition[4] = new Vector2(805, 268);
    asPosition[5] = new Vector2(805, 322);
    angulosPositionA[0] = new Vector2(840, 49);
    angulosPositionA[1] = new Vector2(840, 87);
    angulosPositionA[2] = new Vector2(840, 125);
    angulosPositionA[3] = new Vector2(840, 163);
    angulosPositionA[4] = new Vector2(840, 205);
    angulosPositionA[5] = new Vector2(840, 245);
    asPositionA[0] = new Vector2(796, 49);
    asPositionA[1] = new Vector2(796, 87);
    asPositionA[2] = new Vector2(796, 125);
    asPositionA[3] = new Vector2(796, 163);
    asPositionA[4] = new Vector2(796, 205);
    asPositionA[5] = new Vector2(796, 245);

    // Define que o estado inicial é o da tela inicial
    gameState = GameState.StartMenu;
    previousGameState = GameState.StartMenu;
    gameMode = GameMode.Manual;

    // Obtem o status do mouse
    mouseState = Mouse.GetState();
    previousMouseState = mouseState;

    base.Initialize();
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // Carrega os arquivos de imagem / som / fonte  necessarios
    font = Content.Load<SpriteFont>("myFont");
    startButton = Content.Load<Texture2D>(@"start");
    exitButton = Content.Load<Texture2D>(@"exit");
    logo = Content.Load<Texture2D>(@"simbolo");
    kukinha = Content.Load<Texture2D>(@"kukinha");
    barralateral = Content.Load<Texture2D>(@"barralateral");
    but = Content.Load<Texture2D>(@"but");
    barralateralesq = Content.Load<Texture2D>(@"barralateralesq");
```

```
    bolinhaM = Content.Load<Texture2D>(@"bolinhaM");
    bolinhaA = Content.Load<Texture2D>(@"bolinhaA");
    barralateralA = Content.Load<Texture2D>(@"barralateralA");
    led = Content.Load<Texture2D>(@"led");
    ledA = Content.Load<Texture2D>(@"ledA");
    chave0 = Content.Load<Texture2D>(@"chave0");
    chave1 = Content.Load<Texture2D>(@"chave1");
    model = Content.Load<Model>(@"Models\KukaFinal");
    modelSkybox = Content.Load<Model>(@"Models\Skybox");
    modelMesa = Content.Load<Model>(@"Models\Mesa");
    modelEixos = Content.Load<Model>(@"Models\Eixos");

    hitTable = Content.Load<SoundEffect>("bang_1");
    soundEngine = Content.Load<SoundEffect>("machine_1");
    soundEngineInstance = soundEngine.CreateInstance();
    hitTableInstance = hitTable.CreateInstance();

}

protected void LoadGame()
{
    // Cria um SpriteBatch utilizado para o desenho das imagens 2D
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // Verificacao dos IDs dos ossos
    boneID = model.Bones["Bone0"].Index;
    modelTransform = model.Bones[boneID].Transform;
    boneID1 = model.Bones["Bone1"].Index;
    modelTransform1 = model.Bones[boneID2].Transform;
    boneID2 = model.Bones["Bone2"].Index;
    modelTransform2 = model.Bones[boneID3].Transform;
    boneID3 = model.Bones["Bone3"].Index;
    modelTransform3 = model.Bones[boneID3].Transform;
    boneID4 = model.Bones["Bone4"].Index;
    modelTransform4 = model.Bones[boneID3].Transform;
    boneID5 = model.Bones["Bone5"].Index;
    modelTransform5 = model.Bones[boneID3].Transform;
    boneID6 = model.Bones["Bone6"].Index;
    modelTransform6 = model.Bones[boneID3].Transform;
    boneIDSkybox = modelSkybox.Bones["Bone"].Index;
    modelTransformSkybox = modelSkybox.Bones[boneIDSkybox].Transform;
    boneIDMesa = modelMesa.Bones["Bone"].Index;
    modelTransformMesa = modelMesa.Bones[boneIDMesa].Transform;
    boneIDEixos = modelEixos.Bones["Bone"].Index;
    modelTransformEixos = modelEixos.Bones[boneIDEixos].Transform;

    // Criacao das matrizes de transformacao dos modelos
    transform = new Matrix[model.Bones.Count];
    boneOriginal = new Matrix[model.Bones.Count];
    transformSkybox = new Matrix[modelSkybox.Bones.Count];
    boneOriginalSkybox = new Matrix[modelSkybox.Bones.Count];
    transformMesa = new Matrix[modelMesa.Bones.Count];
    boneOriginalMesa = new Matrix[modelMesa.Bones.Count];
    transformEixos = new Matrix[modelEixos.Bones.Count];
    boneOriginalEixos = new Matrix[modelEixos.Bones.Count];
    model.CopyAbsoluteBoneTransformsTo(transform);


    // Transformacoes iniciais para que os modelos fiquem nas posicoes corretas
    // Obs.: Blender e XNA possuem sistemas de coordenadas diferentes e por isso
```

```
            //      sao necessarias rotacoes para os modelos ficarem nas posicoes corretas
            model.Root.Transform = Matrix.CreateTranslation(0, 0, -1) *
                Matrix.CreateRotationX(-MathHelper.PiOver2) *
                Matrix.CreateRotationY(-MathHelper.PiOver2);
            model.CopyAbsoluteBoneTransformsTo(boneOriginal);
            modelSkybox.Root.Transform = Matrix.CreateTranslation(0, 0, 1) *
                Matrix.CreateRotationX(-MathHelper.PiOver2) *
                Matrix.CreateRotationY(-MathHelper.PiOver2);
            modelSkybox.CopyAbsoluteBoneTransformsTo(boneOriginalSkybox);
            modelMesa.Root.Transform = Matrix.CreateTranslation((float)-1.3, 0, (float)0.5) *
                Matrix.CreateRotationX(MathHelper.PiOver2);
            modelMesa.CopyAbsoluteBoneTransformsTo(boneOriginalMesa);
            modelEixos.Root.Transform = Matrix.CreateTranslation((float)1.0, (float)0, (float)0) *
                Matrix.CreateRotationY(-MathHelper.PiOver2) *
                Matrix.CreateRotationZ(-MathHelper.PiOver2);
            modelEixos.CopyAbsoluteBoneTransformsTo(boneOriginalEixos);

            model.CopyAbsoluteBoneTransformsTo(transform);
            position = transform[boneID6] * Matrix.CreateTranslation(0, 1, 0);

            // M41 = -X, M42 = Z, M43 = Y
            positionIK = transform[boneID5] * Matrix.CreateTranslation(0, 1, 0);

            boundingBoxMesa = new BoundingBox(new Vector3((float)-1.6, (float)-1.0, (float)-0.31),
                new Vector3((float)-1.0, (float)-0.52, (float)0.31));
            boundingSphereFerramenta = model.Meshes["Pen"].BoundingSphere.Center;

            theta1Aux = Math.Atan2(positionIK.M43, -positionIK.M41);
            yAux = positionIK.M43;
            rAux = -(transform[boneID2] * Matrix.CreateTranslation(0, 1, 0)).M41;
            zAux = (transform[boneID2] * Matrix.CreateTranslation(0, 1, 0)).M42;

            l2 = Math.Sqrt(Math.Pow((transform[boneID3] * Matrix.CreateTranslation(0, 1, 0)).M42 -
(transform[boneID2] * Matrix.CreateTranslation(0, 1, 0)).M42, 2) +
                Math.Pow((transform[boneID3] * Matrix.CreateTranslation(0, 1, 0)).M41 -
(transform[boneID2] * Matrix.CreateTranslation(0, 1, 0)).M41, 2));
            l3 = Math.Sqrt(Math.Pow((transform[boneID5] * Matrix.CreateTranslation(0, 1, 0)).M42 -
(transform[boneID3] * Matrix.CreateTranslation(0, 1, 0)).M42, 2) +
                Math.Pow((transform[boneID5] * Matrix.CreateTranslation(0, 1, 0)).M41 -
(transform[boneID3] * Matrix.CreateTranslation(0, 1, 0)).M41, 2));
            theta3Aux = Math.Atan2((transform[boneID5] * Matrix.CreateTranslation(0, 1, 0)).M42 -
(transform[boneID3] * Matrix.CreateTranslation(0, 1, 0)).M42,
            -(transform[boneID5] * Matrix.CreateTranslation(0, 1, 0)).M41 + (transform[boneID3] *
Matrix.CreateTranslation(0, 1, 0)).M41);
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// all content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
```

```
        protected override void Update(GameTime gameTime)
        {
            // Para o jogo sair caso seja pressionado ESC
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            // Aguarda clique do mouse
            mouseState = Mouse.GetState();
            if (previousMouseState.LeftButton == ButtonState.Pressed )
            {
                MouseClicked(mouseState.X, mouseState.Y);
            }

            previousMouseState = mouseState;

            // Se o jogo esta na tela de movimentacao do Kuka
            if (gameState == GameState.Playing)
            {
                if (!(previousGameState == GameState.Playing))
                {
                    LoadGame();
                    previousGameState = GameState.Playing;
                }

                if (gameMode == GameMode.Manual)
                {
                    // Pressionando as teclas 1 a 6 ou pelo clique do mouse, seleciona-se a junta a ser
movida
                    if (input.KeyboardState.IsKeyDown(Keys.D1) || junta[0])
                    {
                        joint = 1;
                        bone = boneID1;
                        junta[0] = false;
                        minAngle = -185;
                        maxAngle = 185;
                    }
                    else if (input.KeyboardState.IsKeyDown(Keys.D2) || junta[1])
                    {
                        joint = 2;
                        bone = boneID2;
                        junta[1] = false;
                        minAngle = -55;
                        maxAngle = 115;
                    }
                    else if (input.KeyboardState.IsKeyDown(Keys.D3) || junta[2])
                    {
                        joint = 3;
                        bone = boneID3;
                        junta[2] = false;
                        minAngle = -210;
                        maxAngle = 70;
                    }
                    else if (input.KeyboardState.IsKeyDown(Keys.D4) || junta[3])
                    {
                        joint = 4;
                        bone = boneID4;
                        junta[3] = false;
                        minAngle = -350;
                        maxAngle = 350;
                    }
```

```
            else if (input.KeyboardState.IsKeyDown(Keys.D5) || junta[4])
            {
                joint = 5;
                bone = boneID5;
                junta[4] = false;
                minAngle = -225;
                maxAngle = 45;
            }
            else if (input.KeyboardState.IsKeyDown(Keys.D6) || junta[5])
            {
                joint = 6;
                bone = boneID6;
                junta[5] = false;
                minAngle = -350;
                maxAngle = 350;
            }

            // Pressionando as teclas "Z" ou "X" ou pelo clique do mouse, move-se a junta
            // escolhida acima nos sentidos horario ou anti-horario
            if (((input.KeyboardState.IsKeyDown(Keys.Z) && (joint != 0) ) ||
                (positivo)) && (angulo[joint - 1] < maxAngle))
            {
                MoveJoint(90, gameTime.ElapsedGameTime.TotalSeconds);
                positivo = false;
                boundingSphereFerramenta =
Vector3.Transform(model.Meshes["Pen"].BoundingSphere.Center,
                    transform[model.Meshes["Pen"].ParentBone.Index]);
                soundEngineInstance.Play();
                if (boundingBoxMesa.Contains(boundingSphereFerramenta) ==
ContainmentType.Contains ||
                    boundingBoxMesa.Contains(boundingSphereFerramenta) ==
ContainmentType.Intersects)
                {
                    hitTableInstance.Volume = 0.75f;
                    hitTableInstance.Play();
                    MoveJoint(-90, gameTime.ElapsedGameTime.TotalSeconds);
                }
            }


            if (((input.KeyboardState.IsKeyDown(Keys.X) && (joint != 0)) ||
                (negativo) == true) && (angulo[joint - 1] > minAngle))
            {
                MoveJoint(-90, gameTime.ElapsedGameTime.TotalSeconds);
                negativo = false;
                boundingSphereFerramenta =
Vector3.Transform(model.Meshes["Pen"].BoundingSphere.Center,
                    transform[model.Meshes["Pen"].ParentBone.Index]);
                soundEngineInstance.Play();
                if (boundingBoxMesa.Contains(boundingSphereFerramenta) ==
ContainmentType.Contains ||
                    boundingBoxMesa.Contains(boundingSphereFerramenta) ==
ContainmentType.Intersects)
                {
                    hitTableInstance.Volume = 0.75f;
                    hitTableInstance.Play();
                    MoveJoint(90, gameTime.ElapsedGameTime.TotalSeconds);
                }
            }
        }
```

```
            // Se o jogo esta em modo Automatico
            if (gameMode == GameMode.Automatic)
            {
                if(interpret == true)
                    KRLInterpretor();
            }

            // Pausar o som do motor quando o kuka nao estiver em movimento
            if ((input.KeyboardState.IsKeyDown(Keys.X) == false && soundEngineInstance.State ==
SoundState.Playing)
                && input.KeyboardState.IsKeyDown(Keys.Z) == false && mouseState.LeftButton !=
ButtonState.Pressed)
            {
              soundEngineInstance.Pause();
            }
            // Movimentacao da bouding sphere da ferramenta conforme a movimentacao da mesma
            boundingSphereFerramenta =
Vector3.Transform(model.Meshes["Pen"].BoundingSphere.Center,
                    transform[model.Meshes["Pen"].ParentBone.Index]);
        }
        base.Update(gameTime);
    }

    /// <summary>
    /// This is called when the game should draw itself.
    /// </summary>
    /// <param name="gameTime">Provides a snapshot of timing values.</param>
    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.Gray);
        spriteBatch.Begin();

        // Se o jogo esta na tela de movimentacao do Kuka
        if (gameState == GameState.Playing)
        {
            Matrix world = Matrix.CreateTranslation(new Vector3(0, 0, 0));
            GraphicsDevice.DepthStencilState = new DepthStencilState() { DepthBufferEnable = true };
            GraphicsDevice.BlendState = BlendState.Opaque;
            GraphicsDevice.DepthStencilState = DepthStencilState.Default;
            GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;
            DrawModel(ref model, ref world); // Desenha o robo na tela
            DrawModel(ref modelSkybox, ref world); // Desenha a skybox na tela
            DrawModel(ref modelMesa, ref world); // Desenha a mesa na tela
            DrawModel(ref modelEixos, ref world); // Desenha os eixos na tela
            spriteBatch.Draw(barralateralesq, new Rectangle(0, 0, 178, 95), Color.White);

            // Se o jogo esta em modo manual
            if (gameMode == GameMode.Manual)
            {
                // Desenho das imagens 2D
                spriteBatch.Draw(bolinhaM, new Rectangle(0, 0, 178, 95), Color.White);
                spriteBatch.Draw(barralateral, new Vector2(780, 0), Color.White);
                //Desenho do retangulo de selecao dos botoes
                if ((ap == 1) || ((input.KeyboardState.IsKeyDown(Keys.Z)) && (joint == 1)))
                {
                    spriteBatch.Draw(but, a1PlusRectPress, Color.White);
                    ap = 0;
                }
                if ((ap == 2) || ((input.KeyboardState.IsKeyDown(Keys.Z)) && (joint == 2)))
                {
```

```
    spriteBatch.Draw(but, a2PlusRectPress, Color.White);
    ap = 0;
}
if ((ap == 3) || ((input.KeyboardState.IsKeyDown(Keys.Z)) && (joint == 3)))
{
    spriteBatch.Draw(but, a3PlusRectPress, Color.White);
    ap = 0;
}
if ((ap == 4) || ((input.KeyboardState.IsKeyDown(Keys.Z)) && (joint == 4)))
{
    spriteBatch.Draw(but, a4PlusRectPress, Color.White);
    ap = 0;
}
if ((ap == 5) || ((input.KeyboardState.IsKeyDown(Keys.Z)) && (joint == 5)))
{
    spriteBatch.Draw(but, a5PlusRectPress, Color.White);
    ap = 0;
}
if ((ap == 6) || ((input.KeyboardState.IsKeyDown(Keys.Z)) && (joint == 6)))
{
    spriteBatch.Draw(but, a6PlusRectPress, Color.White);
    ap = 0;
}
if ((am == 1) || ((input.KeyboardState.IsKeyDown(Keys.X)) && (joint == 1)))
{
    spriteBatch.Draw(but, a1MinusRectPress, Color.White);
    am = 0;
}
if ((am == 2) || ((input.KeyboardState.IsKeyDown(Keys.X)) && (joint == 2)))
{
    spriteBatch.Draw(but, a2MinusRectPress, Color.White);
    am = 0;
}
if ((am == 3) || ((input.KeyboardState.IsKeyDown(Keys.X)) && (joint == 3)))
{
    spriteBatch.Draw(but, a3MinusRectPress, Color.White);
    am = 0;
}
if ((am == 4) || ((input.KeyboardState.IsKeyDown(Keys.X)) && (joint == 4)))
{
    spriteBatch.Draw(but, a4MinusRectPress, Color.White);
    am = 0;
}
if ((am == 5) || ((input.KeyboardState.IsKeyDown(Keys.X)) && (joint == 5)))
{
    spriteBatch.Draw(but, a5MinusRectPress, Color.White);
    am = 0;
}
if ((am == 6) || ((input.KeyboardState.IsKeyDown(Keys.X)) && (joint == 6)))
{
    spriteBatch.Draw(but, a6MinusRectPress, Color.White);
    am = 0;
}
if (ap == 7)
{
    spriteBatch.Draw(but, VelPlusRectPress, Color.White);
    ap = 0;
}
if (ap == 8)
{
```

```
            spriteBatch.Draw(but, VelMinusRectPress, Color.White);
            ap = 0;
        }

        if (xm == 1)
        {
            spriteBatch.Draw(but, xMinusRectPress, Color.White);
            xm = 0;
        }
        if (xm == 2)
        {
            spriteBatch.Draw(but, yMinusRectPress, Color.White);
            xm = 0;
        }
        if (xm == 3)
        {
            spriteBatch.Draw(but, zMinusRectPress, Color.White);
            xm = 0;
        }
        if (xm == 4)
        {
            spriteBatch.Draw(but, xPlusRectPress, Color.White);
            xm = 0;
        }
        if (xm == 5)
        {
            spriteBatch.Draw(but, yPlusRectPress, Color.White);
            xm = 0;
        }
        if (xm == 6)
        {
            spriteBatch.Draw(but, zPlusRectPress, Color.White);
            xm = 0;
        }
    }

    // Se em modo automatico
    if (gameMode == GameMode.Automatic)
    {
        // Desenho das imagens 2D
        spriteBatch.Draw(bolinhaA, new Rectangle(0, 0, 178, 95), Color.White);
        spriteBatch.Draw(barralateralA, new Vector2(683, 0), Color.White);
        krl = new KRLManager();
        // Texto do codigo na tela
        spriteBatch.DrawString(font, krl.treeString, new Vector2(700, 315), Color.Black);
    }

    // Impressao das variaveis na tela
    for (int i = 0; i <= 5; i++)
    {
        angulostring[i] = angulo[i].ToString("N1");
        // Se em modo manual
        if (gameMode == GameMode.Manual)
        {
            spriteBatch.DrawString(font, "A" + (i + 1), asPosition[i], Color.Black);
            spriteBatch.DrawString(font, angulostring[i] + "º", angulosPosition[i], Color.Black);
        }
        // Se em modo automatico, desenha tambem leds e chaves
        if (gameMode == GameMode.Automatic)
        {
```

```
spriteBatch.DrawString(font, "A" + (i + 1), asPositionA[i], Color.Black);
spriteBatch.DrawString(font, angulostring[i] + "º", angulosPositionA[i], Color.Black);
// Alterando status dos leds e das chaves
if (leds[1] == 0)
{
    spriteBatch.Draw(led, new Vector2(910, 180), Color.White);
}
if (leds[1] == 1)
{
    spriteBatch.Draw(ledA, new Vector2(910, 180), Color.White);
}
if (chaves[1] == 0)
{
    spriteBatch.Draw(chave0, new Vector2(912, 225), Color.White);
}
if (chaves[1] == 1)
{
    spriteBatch.Draw(chave1, new Vector2(912, 225), Color.White);

}
if (leds[2] == 0)
{
    spriteBatch.Draw(led, new Vector2(932, 180), Color.White);
}
if (leds[2] == 1)
{
    spriteBatch.Draw(ledA, new Vector2(932, 180), Color.White);
}
if (chaves[2] == 0)
{
    spriteBatch.Draw(chave0, new Vector2(932, 225), Color.White);
}
if (chaves[2] == 1)
{
    spriteBatch.Draw(chave1, new Vector2(932, 225), Color.White);
}
if (leds[3] == 0)
{
    spriteBatch.Draw(led, new Vector2(952, 180), Color.White);
}
if (leds[3] == 1)
{
    spriteBatch.Draw(ledA, new Vector2(952, 180), Color.White);
}
if (chaves[3] == 0)
{
    spriteBatch.Draw(chave0, new Vector2(952, 225), Color.White);
}
if (chaves[3] == 1)
{
    spriteBatch.Draw(chave1, new Vector2(952, 225), Color.White);
}
if (leds[4] == 0)
{
    spriteBatch.Draw(led, new Vector2(972, 180), Color.White);
}
if (leds[4] == 1)
{
    spriteBatch.Draw(ledA, new Vector2(972, 180), Color.White);
}
```

```
        if (chaves[4] == 0)
        {
            spriteBatch.Draw(chave0, new Vector2(972, 225), Color.White);
        }
        if (chaves[4] == 1)
        {
            spriteBatch.Draw(chave1, new Vector2(972, 225), Color.White);
        }
        if (leds[5] == 0)
        {
            spriteBatch.Draw(led, new Vector2(992, 180), Color.White);
        }
        if (leds[5] == 1)
        {
            spriteBatch.Draw(ledA, new Vector2(992, 180), Color.White);
        }
        if (chaves[5] == 0)
        {
            spriteBatch.Draw(chave0, new Vector2(992, 225), Color.White);
        }
        if (chaves[5] == 1)
        {
            spriteBatch.Draw(chave1, new Vector2(992, 225), Color.White);
        }
    }
}

// Se em modo manual, permite selecao da velocidade
// e imprime valores da posicao do efetuador
if (gameMode == GameMode.Manual)
{
    vel = Convert.ToString(velocidade * 100);
    spriteBatch.DrawString(font, vel + " %", new Vector2(870, 398), Color.Black);

    spriteBatch.DrawString(font, "X", new Vector2(805, 488), Color.Black);
    spriteBatch.DrawString(font, (position.M41 * (-1)).ToString("N2") + " m",
        new Vector2(887, 488), Color.Black);
    spriteBatch.DrawString(font, "Y", new Vector2(805, 542), Color.Black);
    spriteBatch.DrawString(font, position.M43.ToString("N2") + " m",
        new Vector2(887, 542), Color.Black);
    spriteBatch.DrawString(font, "Z", new Vector2(805, 596), Color.Black);
    spriteBatch.DrawString(font, position.M42.ToString("N2") + " m",
        new Vector2(887, 596), Color.Black);
}

// Se em modo automatico,  imprime valores da posicao do efetuador
if (gameMode == GameMode.Automatic)
{
    spriteBatch.DrawString(font, "X", new Vector2(914, 49), Color.Black);
    spriteBatch.DrawString(font, (position.M41 * (-1)).ToString("N2") + " m",
        new Vector2(940, 49), Color.Black);
    spriteBatch.DrawString(font, "Y", new Vector2(914, 87), Color.Black);
    spriteBatch.DrawString(font, position.M43.ToString("N2") + " m",
        new Vector2(940, 87), Color.Black);
    spriteBatch.DrawString(font, "Z", new Vector2(914, 127), Color.Black);
    spriteBatch.DrawString(font, position.M42.ToString("N2") + " m",
        new Vector2(940, 127), Color.Black);
}
GraphicsDevice.DepthStencilState = new DepthStencilState() { DepthBufferEnable = true };
```

```
    }

    // Se o jogo esta na tela inicial, desenha as imagens necessarias
    if (gameState == GameState.StartMenu)
    {
        spriteBatch.Draw(kukinha, new Vector2(270, 250), Color.White);
        spriteBatch.Draw(logo, new Vector2(270, 40), Color.White);
        spriteBatch.Draw(startButton, startButtonPosition, Color.White);
        spriteBatch.Draw(exitButton, exitButtonPosition, Color.White);
    }

    spriteBatch.End();
    RasterizerState state = new RasterizerState();
    state.CullMode = CullMode.None;
    graphics.GraphicsDevice.RasterizerState = state;
    base.Draw(gameTime);
}

// Essa função lida com o clique do mouse
void MouseClicked(int x, int y)
{
    // Cria um retangulo ficticio na area onde o mouse foi clicado
    Rectangle mouseClickRect = new Rectangle(x, y, 10, 10);

    // Na tela inicial
    if (gameState == GameState.StartMenu)
    {
        Rectangle startButtonRect = new Rectangle((int)startButtonPosition.X,
            (int)startButtonPosition.Y, 200, 70);
        Rectangle exitButtonRect = new Rectangle((int)exitButtonPosition.X,
            (int)exitButtonPosition.Y, 150, 65);

        if (mouseClickRect.Intersects(startButtonRect))
        {
            gameState = GameState.Playing;
        }
        else if (mouseClickRect.Intersects(exitButtonRect))
        {
            Exit();
        }
    }
    // Na tela de movimentacao do Kuka
    if (gameState == GameState.Playing)
    {
        if (gameMode == GameMode.Manual)
        {

            if (mouseClickRect.Intersects(a1PlusRect))
            {
                ap = 1;
                junta[0] = true;
                positivo = true;
            }
            if (mouseClickRect.Intersects(a2PlusRect))
            {
                ap = 2;
                junta[1] = true;
                positivo = true;
            }
```

```
if (mouseClickRect.Intersects(a3PlusRect))
{
    ap = 3;
    junta[2] = true;
    positivo = true;
}
if (mouseClickRect.Intersects(a4PlusRect))
{
    ap = 4;
    junta[3] = true;
    positivo = true;
}
if (mouseClickRect.Intersects(a5PlusRect))
{
    ap = 5;
    junta[4] = true;
    positivo = true;
}
if (mouseClickRect.Intersects(a6PlusRect))
{
    ap = 6;
    junta[5] = true;
    positivo = true;

}
if (mouseClickRect.Intersects(a1MinusRect))
{
    am = 1;
    junta[0] = true;
    negativo = true;
}
if (mouseClickRect.Intersects(a2MinusRect))
{
    am = 2;
    junta[1] = true;
    negativo = true;
}
if (mouseClickRect.Intersects(a3MinusRect))
{
    am = 3;
    junta[2] = true;
    negativo = true;
}
if (mouseClickRect.Intersects(a4MinusRect))
{
    am = 4;
    junta[3] = true;
    negativo = true;
}
if (mouseClickRect.Intersects(a5MinusRect))
{
    am = 5;
    junta[4] = true;
    negativo = true;
}
if (mouseClickRect.Intersects(a6MinusRect))
{
    am = 6;
    junta[5] = true;
    negativo = true;
```

```
}
if (mouseClickRect.Intersects(VelPlusRect))
{
    ap = 7;
    if (mouseState.LeftButton == ButtonState.Released)
    {
        if (velocidade != 1.0)
            velocidade = velocidade + 0.1;
    }
}
if (mouseClickRect.Intersects(VelMinusRect))
{
    ap = 8;
    if (mouseState.LeftButton == ButtonState.Released)
    {
        if (velocidade != 0.1 && velocidade > 0.15)
            velocidade = velocidade - 0.1;
    }
}
if (mouseClickRect.Intersects(xMinusRect) || xminus)
{
    xm = 1;
    positionIK.M41 = positionIK.M41 + (float)(0.007 * velocidade + 0.008);
    InverseKinematics();
    if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
    {
        bone = boneID1;
        MoveJoint(theta1, 1);
        bone = boneID2;
        MoveJoint(theta2, 2);
        bone = boneID3;
        MoveJoint(theta3, 3);
        // KAREN Adicionei !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        soundEngineInstance.Play();
        if (collision == true)
        {
            xplus = true;
            collision = false;
        }
        xminus = false;
    }
}
if (mouseClickRect.Intersects(yMinusRect) || yminus)
{
    xm = 2;
    positionIK.M43 = positionIK.M43 - (float)(0.007 * velocidade + 0.008);
    InverseKinematics();
    if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
    {
        bone = boneID1;
        MoveJoint(theta1, 1);
        bone = boneID2;
        MoveJoint(theta2, 2);
        bone = boneID3;
        MoveJoint(theta3, 3);
        // KAREN Adicionei !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        soundEngineInstance.Play();
        if (collision == true)
        {
            yplus = true;
```

```
            collision = false;
        }
        yminus = false;
    }
}
if (mouseClickRect.Intersects(zMinusRect) || zminus)
{
    xm = 3;
    positionIK.M42 = positionIK.M42 - (float)(0.007 * velocidade + 0.008);
    InverseKinematics();
    if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
    {
        bone = boneID1;
        MoveJoint(theta1, 1);
        bone = boneID2;
        MoveJoint(theta2, 2);
        bone = boneID3;
        MoveJoint(theta3, 3);
        // KAREN Adicionei !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        soundEngineInstance.Play();
        if (collision == true)
        {
            zplus = true;
            collision = false;
        }
        zminus = false;
    }
}
if (mouseClickRect.Intersects(xPlusRect) || xplus)
{
    xm = 4;
    positionIK.M41 = positionIK.M41 - (float)(0.007 * velocidade + 0.008);
    InverseKinematics();
    if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
    {
        bone = boneID1;
        MoveJoint(theta1, 1);
        bone = boneID2;
        MoveJoint(theta2, 2);
        bone = boneID3;
        MoveJoint(theta3, 3);
        // KAREN Adicionei !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        soundEngineInstance.Play();
        if (collision == true)
        {
            xminus = true;
            collision = false;
        }
        xplus = false;
    }
}
if (mouseClickRect.Intersects(yPlusRect) || yplus)
{
    xm = 5;
    positionIK.M43 = positionIK.M43 + (float)(0.007 * velocidade + 0.008);
    InverseKinematics();
    if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
    {
        bone = boneID1;
        MoveJoint(theta1, 1);
```

```
            bone = boneID2;
            MoveJoint(theta2, 2);
            bone = boneID3;
            MoveJoint(theta3, 3);
            // KAREN Adicionei !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            soundEngineInstance.Play();
            if (collision == true)
            {
                yminus = true;
                collision = false;
            }
            yplus = false;
        }
    }
    if (mouseClickRect.Intersects(zPlusRect) || zplus)
    {
        xm = 6;
        positionIK.M42 = positionIK.M42 + (float)(0.007 * velocidade + 0.008);
        InverseKinematics();
        if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
        {
            bone = boneID1;
            MoveJoint(theta1, 1);
            bone = boneID2;
            MoveJoint(theta2, 2);
            bone = boneID3;
            MoveJoint(theta3, 3);
            // KAREN Adicionei !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            soundEngineInstance.Play();
            if (collision == true)
            {
                zminus = true;
                collision = false;
            }
            zplus = false;
        }
    }
}
if (mouseClickRect.Intersects(manualRect))
{
    gameMode = GameMode.Manual;
}
if (mouseClickRect.Intersects(automaticRect))
{
    gameMode = GameMode.Automatic;
}
if (mouseClickRect.Intersects(sairRect))
{
    this.Exit();
}
if (gameMode == GameMode.Automatic)
{
    if (mouseClickRect.Intersects(chave10))
    {
        chaves[1] = 1;
        leds[1] = 1;
    }
    if (mouseClickRect.Intersects(chave11))
    {
        chaves[1] = 0;
```

```
                leds[1] = 0;
            }
            if (mouseClickRect.Intersects(chave20))
            {
                chaves[2] = 1;
                leds[2] = 1;
            }
            if (mouseClickRect.Intersects(chave21))
            {
                chaves[2] = 0;
                leds[2] = 0;
            }
            if (mouseClickRect.Intersects(chave30))
            {
                chaves[3] = 1;
                leds[3] = 1;
            }
            if (mouseClickRect.Intersects(chave31))
            {
                chaves[3] = 0;
                leds[3] = 0;
            }
            if (mouseClickRect.Intersects(chave40))
            {
                chaves[4] = 1;
                leds[4] = 1;
            }
            if (mouseClickRect.Intersects(chave41))
            {
                chaves[4] = 0;
                leds[4] = 0;
            }
            if (mouseClickRect.Intersects(chave50))
            {
                chaves[5] = 1;
                leds[5] = 1;
            }
            if (mouseClickRect.Intersects(chave51))
            {
                chaves[5] = 0;
                leds[5] = 0;
            }
        }

    }
    // Se em modo automatico
    if (gameMode == GameMode.Automatic)
    {
        if (mouseClickRect.Intersects(loadRect))
        {
            interpret = true;
        }
    }
}

// Funcao para desenhar na tela os objetos tridimensionais
private void DrawModel(ref Model m, ref Matrix world)
{
    m.CopyAbsoluteBoneTransformsTo(transform);
    foreach (ModelMesh mesh in m.Meshes)
```

```
            {
                foreach (BasicEffect e in mesh.Effects)
                {
                    e.EnableDefaultLighting();
                    e.World = transform[mesh.ParentBone.Index] * world;
                    e.View = camera.View;
                    e.Projection = camera.Projection;
                    e.AmbientLightColor = Color.LightBlue.ToVector3();
                }
                GraphicsDevice.SamplerStates[0] = SamplerState.LinearClamp;

                mesh.Draw();
            }
        }
        // Funcao para movimentacao das juntas conforme a direcao e o tempo
        private void MoveJoint(int rot, double elapsedTime)
        {
            angulo[joint - 1] = angulo[joint - 1] + (float)rot *
                (float)(elapsedTime * velocidade);
            model.Bones[bone].Transform = Matrix.CreateRotationY(MathHelper.ToRadians((float)rot *
                (float)(elapsedTime * velocidade))) *
                model.Bones[bone].Transform;
            model.CopyAbsoluteBoneTransformsTo(transform);
            position = transform[boneID6] * Matrix.CreateTranslation(0, 1, 0);
            positionIK = transform[boneID5] * Matrix.CreateTranslation(0, 1, 0);
        }
        // Funcao para movimentacao das juntas conforme o angulo e qual junta
        private void MoveJoint(double angle, int joint)
        {
            deltaAngle = angle - angulo[joint - 1];
            angulo[joint - 1] = angulo[joint - 1] + (float)deltaAngle;
            model.Bones[bone].Transform =
Matrix.CreateRotationY((float)MathHelper.ToRadians((float)deltaAngle)) *
                model.Bones[bone].Transform;
            model.CopyAbsoluteBoneTransformsTo(transform);
            position = transform[boneID6] * Matrix.CreateTranslation(0, 1, 0);
            positionIK = transform[boneID5] * Matrix.CreateTranslation(0, 1, 0);
            boundingSphereFerramenta =
Vector3.Transform(model.Meshes["Pen"].BoundingSphere.Center,
                        transform[model.Meshes["Pen"].ParentBone.Index]);
            if (boundingBoxMesa.Contains(boundingSphereFerramenta) == ContainmentType.Contains ||
                boundingBoxMesa.Contains(boundingSphereFerramenta) == ContainmentType.Intersects)
            {
                hitTableInstance.Volume = 0.75f;
                hitTableInstance.Play();
                collision = true;
            }

        }
        // Lida com a cinematica inversa
        private void InverseKinematics()
        {
            theta1Aux = Math.Asin(yAux / Math.Sqrt(Math.Pow(positionIK.M43, 2) +
Math.Pow(positionIK.M41, 2)));
            theta1 = -(Math.Atan2(positionIK.M43, -positionIK.M41) - theta1Aux);
            r = Math.Sqrt(Math.Pow(positionIK.M43, 2) + Math.Pow(positionIK.M41, 2)) *
Math.Cos(theta1Aux);
            z = positionIK.M42;
            theta3 = Math.Acos((Math.Pow(r - rAux, 2) + Math.Pow(z - zAux, 2) - l2 * l2 - l3 * l3) / (2 * l2 *
l3)) + theta3Aux - Math.PI / 2;
```

```
            alfa = Math.Atan2(z - zAux, r - rAux);
            gama = Math.Atan2(l3 * Math.Cos(-theta3 + theta3Aux), l2 + l3 * Math.Sin(-theta3 +
theta3Aux));
            theta2 = -(alfa + gama - Math.PI / 2);
            theta1 = Math.Round(MathHelper.ToDegrees((float)theta1), 7);
            theta2 = Math.Round(MathHelper.ToDegrees((float)theta2), 7);
            theta3 = Math.Round(MathHelper.ToDegrees((float)theta3), 7);
        }
        // Interpretacao do KRL
        private void KRLInterpretor()
        {
            if (krl.tree.GetChild(numNode) != null)
            {
                switch (krl.tree.GetChild(numNode).Type)
                {
                    case KRLParser.IDENTIFIER:
                        if ((krl.tree.GetChild(numNode + 1).Text == ".") && (krl.tree.GetChild(numNode +
2).Text == "X"))
                        {
                            auxX = Convert.ToDouble(krl.tree.GetChild(numNode + 4).Text) / 10;
                        }
                        else if ((krl.tree.GetChild(numNode + 1).Text == ".") && (krl.tree.GetChild(numNode +
2).Text == "Y"))
                        {
                            auxY = Convert.ToDouble(krl.tree.GetChild(numNode + 4).Text) / 10;
                        }
                        else if ((krl.tree.GetChild(numNode + 1).Text == ".") && (krl.tree.GetChild(numNode +
2).Text == "Z"))
                        {
                            auxZ = Convert.ToDouble(krl.tree.GetChild(numNode + 4).Text) / 10;
                        }
                        numNode = numNode + 1;
                        break;
                    case KRLParser.LIN:
                        if (auxX > -positionIK.M41)
                        {

                            positionIK.M41 = positionIK.M41 - (float)0.01;
                            InverseKinematics();
                            if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
                            {
                                bone = boneID1;
                                MoveJoint(theta1, 1);
                                bone = boneID2;
                                MoveJoint(theta2, 2);
                                bone = boneID3;
                                MoveJoint(theta3, 3);
                                indexAux++;
                                numNode = numNode - 1;
                            }
                            if (auxX <= -positionIK.M41)
                                numNode = numNode + 2;
                        }
                        if (auxY > positionIK.M43)
                        {
                            positionIK.M43 = positionIK.M43 + (float)0.01;
                            InverseKinematics();
                            if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
                            {
                                bone = boneID1;
```

```
                        MoveJoint(theta1, 1);
                        bone = boneID2;
                        MoveJoint(theta2, 2);
                        bone = boneID3;
                        MoveJoint(theta3, 3);
                        indexAux++;
                        numNode = numNode - 1;
                    }
                    if (auxY <= positionIK.M43)
                        numNode = numNode + 2;
                }
                if (auxZ > positionIK.M42)
                {
                    positionIK.M42 = positionIK.M42 + (float)0.01;
                    InverseKinematics();
                    if (!Double.IsNaN(theta1) && !Double.IsNaN(theta2) && !Double.IsNaN(theta3))
                    {
                        bone = boneID1;
                        MoveJoint(theta1, 1);
                        bone = boneID2;
                        MoveJoint(theta2, 2);
                        bone = boneID3;
                        MoveJoint(theta3, 3);
                        indexAux++;
                        numNode = numNode - 1;
                    }
                    if (auxZ <= -positionIK.M42)
                        numNode = numNode + 2;
                }
                break;
            default:
                numNode++;
                break;
        }
    }
    else
    {
        interpret = false;
    }

  }

 }
}
```

Arquivo KRLManager.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

using Antlr.Runtime;
using Antlr.Runtime.Tree;
using Antlr.Runtime.Debug;

using Antlr.StringTemplate;
using Antlr.StringTemplate.Language;
```

```
using RewriteRuleITokenStream = Antlr.Runtime.Tree.RewriteRuleTokenStream;


namespace KukaSimulation
{
   class KRLManager
   {
      private String _treeString;
      private CommonTree _tree;

      public KRLManager()
      {
         // Leitura do arquivo com o codigo KRL definido pelo usuario
         ICharStream file = new ANTLRFileStream(@"kuka.txt");
         KRLLexer lexer = new KRLLexer(file);
         CommonTokenStream tokens = new CommonTokenStream(lexer);
         KRLParser parser = new KRLParser(tokens);
         _tree = (CommonTree)parser.module().Tree;
         _treeString = _tree.ToStringTree();
      }

      public ITree GetChild(int i)
      {
         return _tree.GetChild(i);
      }

      public String treeString
      {
         get { return _treeString; }
      }
      public CommonTree tree
      {
         get { return _tree; }
      }


   }
}
```

Arquivo Câmera.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;


namespace XELibrary
{
   /// <summary>
   /// This is a game component that implements IUpdateable.
```

```
/// </summary>
public class Camera : Microsoft.Xna.Framework.GameComponent
{

    protected IInputHandler input;
    private GraphicsDeviceManager graphics;

    private Matrix projection;
    private Matrix view;
    private Vector3 cameraPosition = new Vector3(0.0f, 0.0f, 3.0f);
    private Vector3 cameraTarget = Vector3.Zero;
    private Vector3 cameraUpVector = Vector3.Up;
    private Vector3 cameraReference = new Vector3(0.0f, 0.0f, -1.0f);
    private float cameraYaw = 0.0f;
    private const float spinRate = 30.0f;
    private float cameraPitch;
    protected Vector3 movement = Vector3.Zero;
    private const float moveRate = 5.0f;

    private Viewport? viewport;

    public Camera(Game game)
        : base(game)
    {
        // TODO: Construct any child components here
        graphics = (GraphicsDeviceManager)Game.Services.GetService(
            typeof(IGraphicsDeviceManager));
        input = (IInputHandler)game.Services.GetService(typeof(IInputHandler));
    }

    /// <summary>
    /// Allows the game component to perform any initialization it needs to before starting
    /// to run.  This is where it can query for any required services and load content.
    /// </summary>
    public override void Initialize()
    {
        // TODO: Add your initialization code here

        base.Initialize();
        InitializeCamera();
    }

    /// <summary>
    /// Allows the game component to update itself.
    /// </summary>
    /// <param name="gameTime">Provides a snapshot of timing values.</param>
    public override void Update(GameTime gameTime)
    {
        // TODO: Add your update code here

        float timeDelta = (float)gameTime.ElapsedGameTime.TotalSeconds;
        if (input.KeyboardState.IsKeyDown(Keys.Left))
            cameraYaw += spinRate * timeDelta;
        if (input.KeyboardState.IsKeyDown(Keys.Right))
            cameraYaw -= spinRate * timeDelta;

        if ((input.PreviousMouseState.X > input.MouseState.X) &&
            (input.MouseState.LeftButton == ButtonState.Pressed) &&
            (((input.MouseState.X > 178) && (input.MouseState.Y >= 0) &&
            (input.MouseState.X < 780) && (input.MouseState.Y <= 648)) ||
```

```
    ((input.MouseState.X >= 0) && (input.MouseState.Y > 95) &&
    (input.MouseState.X < 780) && (input.MouseState.Y <= 648))))
{
    cameraYaw += spinRate * timeDelta;
}
else if ((input.PreviousMouseState.X < input.MouseState.X) &&
    (input.MouseState.LeftButton == ButtonState.Pressed) &&
    (((input.MouseState.X > 178) && (input.MouseState.Y >= 0) &&
    (input.MouseState.X < 780) && (input.MouseState.Y <= 648)) ||
    ((input.MouseState.X >= 0) && (input.MouseState.Y > 95) &&
    (input.MouseState.X < 780) && (input.MouseState.Y <= 648))))
{
    cameraYaw -= spinRate * timeDelta;
}

if (cameraYaw > 360)
    cameraYaw = 0;
else if (cameraYaw < 0)
    cameraYaw = 360;

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

if ((input.PreviousMouseState.Y > input.MouseState.Y) &&
    (input.MouseState.LeftButton == ButtonState.Pressed) &&
    (((input.MouseState.X > 178) && (input.MouseState.Y >= 0) &&
    (input.MouseState.X < 780) && (input.MouseState.Y <= 648)) ||
    ((input.MouseState.X >= 0) && (input.MouseState.Y > 95) &&
    (input.MouseState.X < 780) && (input.MouseState.Y <= 648))))
    cameraPitch += spinRate * timeDelta;
else if ((input.PreviousMouseState.Y < input.MouseState.Y) &&
    (input.MouseState.LeftButton == ButtonState.Pressed) &&
    (((input.MouseState.X > 178) && (input.MouseState.Y >= 0) &&
    (input.MouseState.X < 780) && (input.MouseState.Y <= 648)) ||
    ((input.MouseState.X >= 0) && (input.MouseState.Y > 95) &&
    (input.MouseState.X < 780) && (input.MouseState.Y <= 648))))
    cameraPitch -= spinRate * timeDelta;

if (cameraPitch > 89)
    cameraPitch = 89;
if (cameraPitch < -89)
    cameraPitch = -89;

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)) *
    rotationMatrix;
// Create a vector pointing the direction the camera is facing.
```

```
    Vector3 transformedReference;
    Vector3.Transform(ref cameraReference, ref rotationMatrix,
        out transformedReference);
    // Calculate the position the camera is looking at.
    Vector3.Add(ref cameraPosition, ref transformedReference, out cameraTarget);
    Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget, ref cameraUpVector,
        out view);

    base.Update(gameTime);
}

private void InitializeCamera()
{
    float aspectRatio = (float)graphics.GraphicsDevice.Viewport.Width /
        (float)graphics.GraphicsDevice.Viewport.Height;
    Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4, aspectRatio,
        0.0001f, 1000.0f, out projection);

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

}

public Matrix View
{
    get { return view; }
}

public Matrix Projection
{
    get { return projection; }
}

public Vector3 Position
{
    get { return (cameraPosition); }
    set { cameraPosition = value; }
}

public Vector3 Orientation
{
    get { return (cameraReference); }
    set { cameraReference = value; }
}

public Vector3 Target
{
    get { return (cameraTarget); }
    set { cameraTarget = value; }
}

public Viewport Viewport
{
    get
    {
        if (viewport == null)
            viewport = graphics.GraphicsDevice.Viewport;
        return ((Viewport)viewport);
    }
    set
```

```
        {
            viewport = value;
            InitializeCamera();
        }
    }

  }
}
```

Arquivo FirstPersonCamera.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;


namespace XELibrary
{
    /// <summary>
    /// This is a game component that implements IUpdateable.
    /// </summary>
    public class FirstPersonCamera : Camera
    {



        public FirstPersonCamera(Game game)
            : base(game)
        {
            // TODO: Construct any child components here
        }

        /// <summary>
        /// Allows the game component to perform any initialization it needs to before starting
        /// to run.  This is where it can query for any required services and load content.
        /// </summary>
        public override void Initialize()
        {
            // TODO: Add your initialization code here

            base.Initialize();
        }

        /// <summary>
        /// Allows the game component to update itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        public override void Update(GameTime gameTime)
        {
            // TODO: Add your update code here
            movement = Vector3.Zero;
            if (input.KeyboardState.IsKeyDown(Keys.A))
```

```
            {
                movement.X--;
            }
            if (input.KeyboardState.IsKeyDown(Keys.D))
            {
                movement.X++;
            }
            if (input.KeyboardState.IsKeyDown(Keys.S))
            {
                movement.Z++;
            }
            if (input.KeyboardState.IsKeyDown(Keys.W))
            {
                movement.Z--;
            }
            //make sure we don't increase speed if pushing up and over (diagonal)
            if (movement.LengthSquared() != 0)
                movement.Normalize();



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

Arquivo FPS.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;


namespace XELibrary
{
    /// <summary>
    /// This is a game component that implements IUpdateable.
    /// </summary>
    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) { }

    /// <summary>
    /// Allows the game component to perform any initialization it needs to before starting
    /// to run.  This is where it can query for any required services and load content.
    /// </summary>
    public override void Initialize()
    {
        // TODO: Add your initialization code here

        base.Initialize();
    }

    /// <summary>
    /// Allows the game component to update itself.
    /// </summary>
    /// <param name="gameTime">Provides a snapshot of timing values.</param>
    public override void Update(GameTime gameTime)
    {
        // TODO: Add your update code here
        base.Update(gameTime);
    }

    public sealed override void Draw(GameTime gameTime)
    {
        float elapsed = (float)gameTime.ElapsedGameTime.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);
    }
  }
}
```

Arquivo InputHandler.cs

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;


namespace XELibrary
{
    /// <summary>
    /// This is a game component that implements IUpdateable.
    /// </summary>
    public class InputHandler : Microsoft.Xna.Framework.GameComponent,
IInputHandler
    {

        private KeyboardHandler keyboard;

        private MouseState mouseState;
        private MouseState prevMouseState;

        public InputHandler(Game game)
            : base(game)
        {
            // TODO: Construct any child components here
            game.Services.AddService(typeof(IInputHandler), this);

            keyboard = new KeyboardHandler();
            Game.IsMouseVisible = true;
            prevMouseState = Mouse.GetState();

        }

        /// <summary>
        /// Allows the game component to perform any initialization it needs to
before starting
        /// to run.  This is where it can query for any required services and load
content.
        /// </summary>
        public override void Initialize()
        {
            // TODO: Add your initialization code here

            base.Initialize();
        }

        /// <summary>
        /// Allows the game component to update itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        public override void Update(GameTime gameTime)
        {
            // TODO: Add your update code here
            keyboard.Update();
            if (keyboard.IsKeyDown(Keys.Escape))
                Game.Exit();
```

```
            prevMouseState = mouseState;
            mouseState = Mouse.GetState();

            base.Update(gameTime);
        }

        public KeyboardHandler KeyboardState
        {
            get { return (keyboard); }
        }

        public MouseState MouseState
        {
            get { return (mouseState); }
        }

        public MouseState PreviousMouseState
        {
            get { return (prevMouseState); }
        }

        }
    public class KeyboardHandler
    {
        private KeyboardState prevKeyboardState;
        private KeyboardState keyboardState;
        public KeyboardHandler()
        {
            prevKeyboardState = Keyboard.GetState();
        }
        public bool IsKeyDown(Keys key)
        {
            return (keyboardState.IsKeyDown(key));
        }
        public bool IsHoldingKey(Keys key)
        {
            return (keyboardState.IsKeyDown(key) &&
                prevKeyboardState.IsKeyDown(key));
        }
        public bool WasKeyPressed(Keys key)
        {
            return (keyboardState.IsKeyDown(key) &&
        prevKeyboardState.IsKeyUp(key));
        }
        public bool HasReleasedKey(Keys key)
        {
            return (keyboardState.IsKeyUp(key) &&
                prevKeyboardState.IsKeyDown(key));
        }
        public void Update()
        {
            //set our previous state to our new state
            prevKeyboardState = keyboardState;
            //get our new state
            keyboardState = Keyboard.GetState();
        }
    }
}
```

Arquivo IinputHandler.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework.Input;

namespace XELibrary
{
    public interface IInputHandler
    {
        KeyboardHandler KeyboardState { get; }

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

    }
}
```