UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SIMULADOR PARA ESTACIONAMENTO DE CARROS AUTÔNOMOS NÃO ARTICULADOS USANDO LÓGICA DIFUSA

EWERTON ROCHA MACHADO

BLUMENAU
2009

2009/2-07

EWERTON ROCHA MACHADO

SIMULADOR PARA ESTACIONAMENTO DE CARROS AUTÔNOMOS NÃO ARTICULADOS USANDO LÓGICA DIFUSA

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos, Doutor - Orientador

BLUMENAU
2009

2009/2-07

SIMULADOR PARA ESTACIONAMENTO DE CARROS AUTÔNOMOS NÃO ARTICULADOS USANDO LÓGICA DIFUSA

Por

EWERTON ROCHA MACHADO

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof. Mauro Marcelo Mattos, Dr. – Orientador, FURB
	_____
Membro:	Prof. Roberto Heinzle, Ms. – FURB
	_____
Membro:	Prof. Miguel Alexandre Wisintainer, Ms. – FURB

Blumenau, 15 de dezembro de 2009

Dedico este trabalho à minha família que sempre acreditou em minha capacidade.

AGRADECIMENTOS

Agradeço em primeiro lugar a minha família por sempre direcionar-me no caminho dos estudos, prestando muito apoio e confiança.

Ao meu orientador Mauro Marcelo Mattos por toda a paciência e suporte, mostrando-se sempre motivado pela realização e conclusão do presente trabalho.

A todos os professores e demais colaboradores da Universidade que contribuíram para meu crescimento acadêmico e também pessoal.

Grandes realizações não são feitas por impulso, mas por uma soma de pequenas realizações.

Vincent Van Gogh

RESUMO

Este trabalho tem como objetivo apresentar o desenvolvimento de um simulador de estacionamento para carros não articulados de forma autônoma. Para o mesmo se faz necessário à utilização da lógica difusa, tornando possível o ato de estacionar através de sua capacidade de distinguir os conceitos imprecisos e tratar incertezas provenientes de informações parciais ou incompletas. Por fim, para a construção do mundo virtual do simulador, apresenta a utilização da *engine* gráfica JMonkey Engine e o *framework* JME Physics.

Palavras-chave: Lógica difusa. Simulador autônomo. JMonkey Engine.

ABSTRACT

This paper aims to presents the development of a simulator of parking for cars without a link autonomously. For the same is necessary the use of fuzzy logic, making possible the act of parking through their ability to discriminate between inaccurate and treat uncertainties from partial or incomplete information. Finally, for the construction of the virtual world simulator, the use of graphics engine jMonkey Engine and the framework JME Physics.

Key-words: Fuzzy logic. Standalone simulator. JMonkey Engine.

LISTA DE ILUSTRAÇÕES

Figura 1 – Modelo da VW Touran 2007 equipada com sistema automático de *park-assist* 16
Quadro 1 – Estacionamento automático de um automóvel BMW 18
Figura 2 – Representação dos valores na lógica difusa 21
Figura 3 – Intervalo difuso 21
Figura 4 – Operação complemento 22
Figura 5 – Operação união 23
Figura 6 – Operação intersecção 23
Figura 7 – Sistema de controle difuso 24
Figura 8 – Arquitetura da JME 28
Figura 9 – Diagrama de classe do renderizador LWJGL 29
Figura 10 – Exemplo de grafo de cena 29
Figura 11 – Diagrama de classes do componente de geometria da JME 30
Quadro 2 – Principais classes da *engine* JME e do *framework* JME Physics 31
Figura 12 – Simulador modelado 33
Figura 13 – Diagrama de casos de uso do ator usuário 35
Figura 14 – Diagrama de casos de uso do ator veículo 35
Figura 15 – Diagrama das classes responsável pela criação do ambiente virtual e estacionamento 36
Figura 16 – Diagrama das classes responsável pela criação do veículo 37
Quadro 3 – Trecho de código da classe `TestAdvancedVehicle` 41
Quadro 4 – Método `simpleInitGame` 42
Quadro 5 – Método `createFloor` 42
Quadro 6 – Trecho de código da classe `Car` 43
Quadro 7 – Criação do chassi e da suspensão do veículo 43
Quadro 8 – Trecho de código da classe `TestAdvancedVehicle` 44
Quadro 9 – Método `startPark` 45
Quadro 10 – Trecho de código do método `ParkingSimulation` 46
Quadro 11 – Método `setVars` 46
Quadro 12 – Método `carMoviment` 47
Quadro 13 – Métodos `moverEsquerda`, `moverDireita` e `moverReto` 47

Quadro 14 – Método `simulationFinished` 47
Figura 17 – Parâmetros de visualização do simulador 48
Figura 18 – Posição inicial do veículo 48
Figura 19 – Posição ideal para iniciar o estacionamento 49
Figura 20 – Veículo em movimento 49
Figura 21 – Estacionamento completo 50

LISTA DE SIGLAS

API – *Application Programming Interface*

JME – *jMonkey Engine*

JNI – *Java Native Interface*

JOAL – *Java OpenAL*

LWJGL – *Light Weight* Java *Game Library*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 OBJETIVOS DO TRABALHO 14
1.2 ESTRUTURA DO TRABALHO 14
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 TECNOLOGIAS PARA DIREÇÃO ASSISTIDA 15
2.1.1 Simulação 19
2.2 LÓGICA DIFUSA 20
2.2.1 Operadores dos conjuntos difusos 21
2.2.1.1 Complemento 22
2.2.1.2 União 22
2.2.1.3 Intersecção 23
2.2.2 Variáveis lingüísticas 23
2.2.3 Base de regras 24
2.2.4 Fuzzyficação 25
2.2.5 Inferência 25
2.2.6 Defuzzyficação 26
2.2.7 Aplicação da lógica difusa 26
2.3 JMONKEY ENGINE (JME) 27
2.3.1 Principais classes e interfaces 31
2.4 TRABALHOS CORRELATOS 32
3 DESENVOLVIMENTO 34
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 34
3.2 ESPECIFICAÇÃO 34
3.2.1 Diagrama de casos de uso 35
3.2.2 Diagrama de classes 36
3.2.2.1 Classe `BaseGame` 37
3.2.2.2 Classe `TestAdvancedVehicle` 38
3.2.2.3 Classe `Building` 38
3.2.2.4 Classe `Park` 39
3.2.2.5 Classe `Car` 39
3.2.2.6 Classe `Suspension` 40

3.2.2.7 Classe `Wheel` 40
3.3 IMPLEMENTAÇÃO 41
3.3.1 Tecnologias e ferramentas usadas 41
3.3.2 Implementação gráfica do simulador 41
3.3.3 Estratégia para estacionamento do veículo 44
3.3.4 Operacionalidade da implementação 48
3.4 RESULTADOS E DISCUSSÃO 50
4 CONCLUSÕES 52
4.1 EXTENSÕES 52
REFERÊNCIAS BIBLIOGRÁFICAS 54

1 INTRODUÇÃO

A indústria automobilística, juntamente com a área de automação, vem em um processo de união cada vez mais intenso. Atualmente, robôs e simuladores autônomos já participam ativamente desde o processo de montagem dos automóveis até nas tecnologias embarcadas nestes veículos.

Projetos futuros apontam para a automação completa do ato de dirigir, onde basta o condutor do veículo informar o seu destino, que o veículo o conduzirá de forma totalmente independente até o local escolhido. “Em um futuro muito próximo carros se guiarão sozinhos, a voz é o comando.” (SOARES, 2005).

Fatores de pressão advindos de órgãos reguladores também introduzem um fator de promoção ao desenvolvimento de novas tecnologias. Segundo The Auto Channel (2008), nos Estados Unidos, modelos de carros ano 2010 deverão ser submetidos a uma nova bateria de *crash-tests* frontais e laterais. Esta bateria de testes é a base para a obtenção do selo 5 estrelas da New Car Assessment Program (NCAP) e o órgão regulador é o U.S. National Highway Traffic Safety Administration (NHTSA).

Muitos motoristas enfrentam dificuldades ao estacionar seus carros em vagas apertadas ou pequenos espaços, chegando a causar danos a seus veículos ou provocar pequenos acidentes. São muito comuns já nos dias de hoje, carros com sensores de estacionamento, sensores de chuva, sensores de acendimento automático dos faróis e muitas outras funcionalidades que facilitam o processo de condução de um veículo.

Tavares (2004) afirma que robôs móveis autônomos são capazes de tomar decisões de forma independente frente a situações diversas impostas pelo ambiente sobre o qual operam. Diante do exposto, surge a idéia de simular o processo de estacionamento de um veículo de forma autônoma.

O simulador, para a realização das manobras necessárias, utiliza lógica difusa[1] como diferencial para traduzir em termos matemáticos a informação imprecisa do ambiente. Segundo Tanscheit (2004), um sistema difuso é um sistema de inferência baseado em regras que fornece o ferramental matemático para se lidar com tais regras lingüísticas.

O algoritmo desenvolveu-se utilizando a linguagem Java, fazendo uso da *engine*[2]

---

[1] Também chamada de lógica *fuzzy* ou nebulosa.

[2] *Engine* gráfica é usada no texto para descrever um *middleware* para construção de aplicações gráficas 3D.

gráfica JMonkey Engine para criação da interface gráfica.

1.1 OBJETIVOS DO TRABALHO

O objetivo do trabalho é desenvolver uma aplicação que simule o processo de estacionamento de um veículo autônomo.

Os objetivos específicos do trabalho são:

a) construir uma base de regras difusas que possibilite a simulação do ato de estacionar;
b) disponibilizar uma interface gráfica que permita o acompanhamento do processo de simulação.

1.2 ESTRUTURA DO TRABALHO

O primeiro capítulo apresenta uma introdução do trabalho, os objetivos a serem apresentados e a estrutura do trabalho.

O segundo capítulo contempla a fundamentação teórica do trabalho e descreve as tecnologias para direção assistida, os conceitos da lógica difusa e as principais características da *engine* gráfica JMonkey Engine (JME). Neste capítulo também será mostrado os trabalhos correlatos.

O terceiro capítulo apresenta a especificação e implementação do simulador.

O quarto capítulo descreve as considerações finais sobre o trabalho, incluindo sugestões para extensões em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta de forma geral as tecnologias para direção assistida e também os conceitos da lógica difusa. Explica também sobre a *engine* JMonkey Engine e as principais características.

2.1 TECNOLOGIAS PARA DIREÇÃO ASSISTIDA

A direção assistida, além de propiciar ao motorista maior segurança em condições críticas adversas, também pode auxiliá-lo na condução do veículo, onde uma ou mais tarefas podem ser automatizadas, tais como: seguir a pista mantendo a faixa correta, manter uma distância segura entre veículos, regular automaticamente a velocidade do veículo conforme as condições de trânsito e características da rodovia, fazer ultrapassagens seguras e evitar obstáculos, achar o caminho mais curto (JUNG et al., 2005).

Osório et al. (2006) diz que os veículos autônomos têm atraído a atenção de um grande número de pesquisadores, devido ao desafio que este novo domínio de pesquisas propõe: dotar sistemas de uma capacidade de raciocínio inteligente e de interação com o meio em que estão inseridos.

Jung et al. (2005) também diz que a indústria tem investido intensamente em sistemas eletrônicos embarcados. Com o intuito de auxiliar os condutores, diversas soluções eletrônicas vêm sendo desenvolvidas e implementadas nos veículos nas últimas décadas, como freios ABS e sistema de estabilização ESP, que atuam de forma autônoma, quando o veículo encontra-se em condições extremas, procurando minimizar, dessa forma, a ocorrência de acidentes.

Por exemplo, a tecnologia de estacionamento assistido (ou *self-parking*) não é uma tecnologia nova. A Volkswagen demonstrou um sistema totalmente automático em 1992 de um protótipo conceito que podia estacionar um automóvel sem a presença de motorista. A Toyota disponibilizou algumas unidades para exportação do modelo Prius em 2003 com a opção de *self-parking*. A Volvo também demonstrou a funcionalidade em um carro conceito em 2004. A Siemens VDO desenvolveu um sistema que detecta o tamanho de um espaço de estacionamento e auxilia o motorista, mas não de forma automática. A Mercedes também

oferece um sistema não automático para auxiliar o motorista baseado em sinais sonoros e visuais (HAMPTON, 2006).

De acordo com Hampton (2006), a empresa Hella KGaA Hueck & Co.[3] deve lançar em 2009 um sistema de estacionamento assistido. A tecnologia é similar àquela adotada no Lexus 2007 modelo LS460. O sistema é baseado em um sensor ultra-sônico simplificado posicionado em cada lado do veículo (um sistema mais barato que aqueles baseados em câmeras). Os sensores medem o espaço disponível e a distancia para o carro de trás enquanto a manobra é realizada. A manobra inicia com um sinal do motorista de que o espaço é suficiente. O sistema movimenta a direção enquanto o motorista opera os freios e o acelerador. A base da tecnologia é um sistema elétrico de direção e atuação no freio que é incorporado ao sistema ABS do veículo. A companhia possui uma *joint-venture* com a Volkswagen AG e já disponibilizou o sistema do sistema em uma minivan VW Touran 2007. A Figura 1 ilustra o VW Touran 2007.

Fonte: adaptado de Hampton (2006).

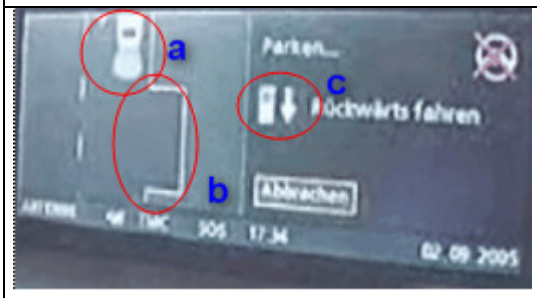
Figura 1 – Modelo da VW Touran 2007 equipada com sistema automático de *park-assist*

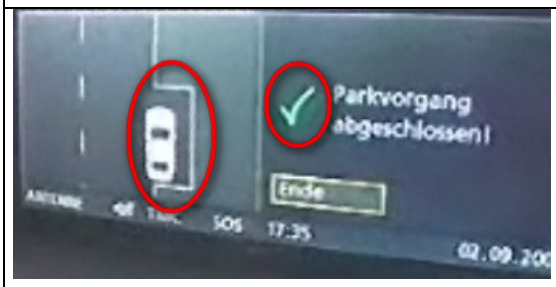
O sistema da VW Touran usa sensores frontais e traseiros para calcular o ângulo ideal de torção da direção para o veículo entrar na vaga. Sinais acústicos e visuais indicam ao motorista a correta posição de início da operação. Uma vez que o veículo esteja na posição inicial, a marcha é automaticamente engatada para ré e a direção movimenta-se autonomamente. O motorista controla o acelerador e os freios e o automóvel estaciona em 15 segundos (HAMPTON, 2006).

A seqüência de imagens do Quadro 1 demonstra o processo de localização de uma vaga e posteriormente o processo de estacionamento automático de um automóvel BMW (METACAFE, 2009).

---

3 Hella KGaA Hueck & Co. desenvolve e produz componentes eletrônicos e de iluminação para a indústria automotiva. Hella está entre os 50 maiores fornecedores de peças automotivas no mundo e uma das 100 maiores indústrias da Alemanha (HELLA, 2009).

IMAGEM	DESCRIÇÃO
	Quando o motorista deseja estacionar o veículo ele aciona um botão no painel que ativa um conjunto de sensores laterais.
	Durante o percurso sensores laterais vão procurando um espaço suficiente para estacionar o automóvel.
	Neste exemplo, o automóvel localizou um espaço de tamanho suficiente.
	No painel do automóvel, um display LCD indica a posição do automóvel (a) em relação ao espaço localizado (b). Ao lado, um sistema de orientação (c) indica ao motorista a direção que o automóvel vai tomar no sentido de estacionar na vaga.
	Neste momento um sistema autônomo assume o controle de direção e aceleração.

	O automóvel então inicia um movimento de marcha-ré com as rodas inicialmente viradas para a direita e, a medida em que o carro avança para trás, o sistema de direção vai alterando a trajetória da direção de tal forma que as rodas da frente são giradas para a esquerda fazendo com que a traseira do automóvel alinhe-se dentro da vaga.
	O sistema então apresenta ao motorista um feedback visual no painel indicando a posição do automóvel dentro da vaga e um indicativo de que a operação foi concluída.

Quadro 1 – Estacionamento automático de um automóvel BMW

De acordo com The Auto Channel (2008), outras tecnologias desenvolvidas pela Hella envolvem:

a) Lane Departure Warning (LDW): sistema com câmera instalada na base do espelho retrovisor que avisa o motorista quando o mesmo troca de faixa numa pista. O processador de sinais do sistema filtra fotos tiradas pela câmera procurando por linhas e estruturas longitudinais para identificar faixas delimitadoras das pistas. Graças a um algoritmo especial e monitorando os movimentos das rodas, o sistema somente emite sinais de aviso em situações de perigo. O dispositivo que funciona em velocidades acima de 30 milhas por hora equipa os modelos Opel Insignia 2009. A Opel é uma marca da General Motors comercializada na Europa;

b) Adaptive Cruise Control (ACC): em produção no Chrysler 300 modelo 2009, o ACC utiliza tecnologia de Light Detection and Ranging (LIDAR), a qual envia sinais infravermelhos para detectar veículos à frente. Ela permite que o controle de velocidade permaneça ativo em condições de trânsito leves a moderadas sem a necessidade de constantes ajustes. A tecnologia auxilia os motoristas a manterem distâncias de segurança automaticamente ativando os freios em caso de perigo;

c) Adaptive Bend-Lighting (ABL): disponível como item opcional em veículos BMW Serie 5 juntamente com o sistema Static Cornering Light (SCL), esta tecnologia quase dobram o range de luz baixa quando o veículo contorna uma esquina. O sistema que pode também funcionar para luz alta, é controlado pelo ângulo do volante, velocidade do veículo e guinadas;

d) Adaptive Cut-Off Light (ACOL): este dispositivo ajusta o range de um sistema de iluminação frontal do veículo para permitir uma melhor visibilidade fazendo com que os faróis atinjam a maior distância possível. O range é alterado dependendo das condições de tráfego;
e) Alertness Assistant (AA): inserindo uma câmera na direção do automóvel voltada para o motorista, o sistema detecta piscada de olhos. Se os olhos do motorista fecharem por mais de 1.5 segundos, o sistema de alerta dispara um aviso sonoro para despertar o motorista;
f) Distance Warning (DD): utilizando um sensor de radar de 24GHz, esta tecnologia para evitar acidentes determina a velocidade relativa e a distância dos veículos à frente e soa um alarme se a distância tornar-se muito pequena;
g) Lane Change Assist (LCA): usando um sistema de radar de 24GHz, o sistema monitora as faixas delimitadoras de pista a direita e a esquerda até uma distância de 200 pés. O sistema avisa se outros veículos estão em pontos cegos para prevenir acidentes;
h) Marking Light (ML): outra tecnologia de luz frontal adaptativa detecta pessoas e pontos de perigo e ilumina-os com luzes de marcação especiais;
i) Traffic-Sign Recognition (TSR): tecnologia encontrada no novo modelo da GM na Europa, o Opel Insignia usa uma câmera frontal e software para detector e identificar sinais de trânsito. A tecnologia pode ser utilizada para avisar motoristas se eles estão andando muito rápido e assim evitar acidentes devido a velocidades inapropriadas.

2.1.1 Simulação

Historicamente a simulação, como técnica, originou-se dos estudos de Von Neumann e Ulan. Estes estudos ficaram conhecidos como análise ou técnica de Monte Carlo. A simulação começou a ser mais utilizada como técnica para solução de problemas, principalmente para o tratamento dos problemas eminentemente probabilísticos, cuja solução analítica é, geralmente, muito mais árdua e difícil, senão impossível (CORNÉLIO FILHO, 1998, sem paginação).

A simulação autônoma inteligente vem se desenvolvendo bastante nos últimos anos. Simuladores e robôs têm auxiliado o homem em diversas tarefas, tais como a exploração

espacial e o auxílio a pessoas com deficiências físicas. Com o grande desenvolvimento das tecnologias de informática, se tratando de hardware e software, potentes computadores podem ser usados para desenvolver novas técnicas de controle robótico (HEINEN, 1999).

2.2 LÓGICA DIFUSA

Segundo Tarig (2001), Aristóteles, filósofo grego, foi o fundador da ciência lógica, e estabeleceu um conjunto de regras rígidas para que conclusões pudessem ser aceitas como logicamente válidas. A utilização da lógica de Aristóteles levava a uma linha de raciocínio lógico baseado em premissas e conclusões.

Desde então, a lógica ocidental, assim chamada, tem sido binária, isto é, uma declaração é falsa ou verdadeira, não podendo ser ao mesmo tempo parcialmente verdadeira e parcialmente falsa. A lógica difusa viola estas suposições (Tarig, 2001).

Segundo Correa (1999), a teoria dos conjuntos nebulosos foi desenvolvida a partir de 1965 por Lotfi Zadeh, para tratar do aspecto do vago da informação. A teoria dos conjuntos pode ser vista então como um caso particular desta teoria mais geral. A teoria dos conjuntos nebulosos, quando utilizada em um contexto lógico, como o de sistemas baseados em conhecimento, é conhecida como lógica difusa.

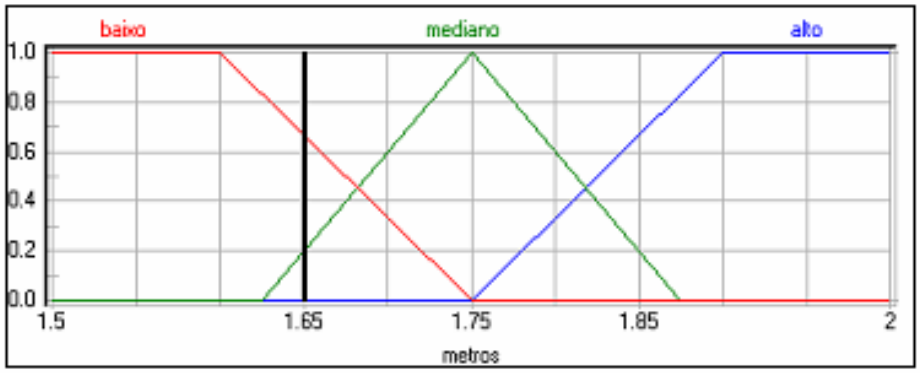
> O enfoque dos sistemas de lógica difusa é voltado para produtos e soluções que incorporam a capacidade humana de distinguir os conceitos imprecisos e tratar incertezas oriundas de informações parciais ou incompletas. Assim sendo, lógica difusa, por definição, é um ramo da lógica que utiliza graus de pertinência em conjuntos ao invés de utilizar uma referência de pertinência estritamente verdadeira ou falsa. Ou seja, a idéia básica é que os valores que representam verdades são representados dentro do intervalo entre, sendo que o valor 0.0 representa o valor lógico absolutamente falso e, o valor 1.0 representa o valor lógico absolutamente verdadeiro. (MATTOS, 2004, p. 6).

A lógica difusa é uma das tecnologias atuais bem sucedidas para o desenvolvimento de sistemas para controlar processos sofisticados. Com sua utilização, requerimentos complexos podem ser implementados em controladores simples, de fácil manutenção e baixo custo. O uso de sistemas construídos desta maneira, chamados de controladores nebulosos, é especialmente interessante quando o modelo matemático está sujeito a incertezas (CORREA, 1999).

Correa (1999) ainda afirma que um controlador nebuloso é um sistema nebuloso a base de regras que definem ações de controle em função de diversas faixas de valores que as

variáveis de estado do problema podem assumir. Estas faixas (usualmente mal definidas) de valores são modeladas por conjuntos nebulosos e denominados de termos lingüísticos. A maior dificuldade na criação de sistemas nebulosos, e de controladores nebulosos em particular, encontra-se na definição dos termos lingüísticos e das regras.

Uma representação gráfica convencional de valores na lógica difusa é ilustrada na Figura 2, em que a altura das pessoas é representada na abscissa e três funções (baixo, mediano e alto) representam a classificação das pessoas quanto à altura.



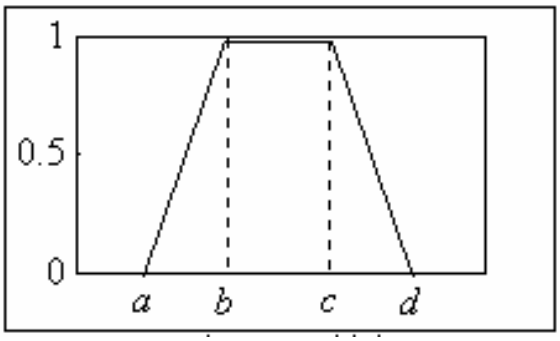
Fonte: adaptado de Bilobrovec (2005).

Figura 2 – Representação dos valores na lógica difusa

2.2.1 Operadores dos conjuntos difusos

Como nos conjuntos tradicionais, existem operações especificas que servem para combinar e alterar os conjuntos difusos. Estas operações são as ferramentas básicas da lógica difusa. A teoria originária dos conjuntos difusos foi fundamentada nos termos das três operações realizadas com conjuntos (complemento, união e interseção) que são equivalentes às operações da lógica booleana (“negação”, “ou”, e “e”) (COX, 1994).

A Figura 3 ilustra um intervalo difuso entre dois números quaisquer.



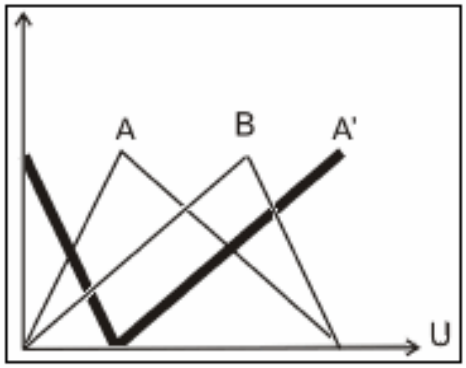
Fonte: adaptado de Hernandes (2009).

Figura 3 – Intervalo difuso

2.2.1.1 Complemento

Segundo Cox (1994), a operação complemento é utilizada para definir a função de pertinência oposta de um subconjunto, ou seja, o complemento do subconjunto A, definido como $\bar{A}$, é formado pelos pontos opostos de A de dentro do intervalo [0, 1]. Essa operação, quando tratada nos extremos desse intervalo, é equivalente à operação "negação" da lógica booleana. A representação formal da operação complemento é descrita na equação $\mu\bar{A}(x) = 1-\mu A(x)$.

A sua representação gráfica é ilustrada na Figura 4.



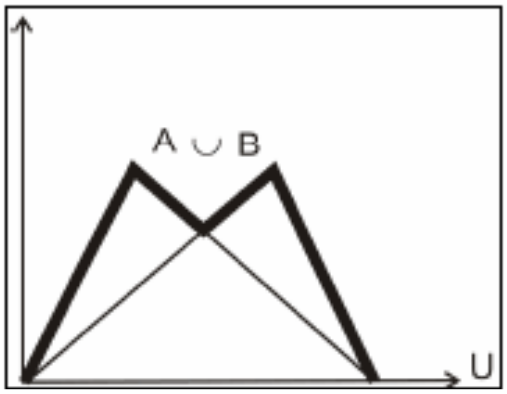
Fonte: adaptado de Hernandes (2009).

Figura 4 – Operação complemento

2.2.1.2 União

A operação união é utilizada para associar dois subconjuntos, ou seja, a união do subconjunto A com B resulta em um subconjunto abrangendo os pontos máximos dos dois subconjuntos unidos. Essa operação, quando tratada nos extremos do intervalo [0,1], é equivalente à operação "ou" da lógica booleana. A representação formal da operação união é descrita na equação: $A \cup B \rightarrow \mu A(x) \cup \mu B(x) = \max(\mu A(x), \mu B(x))$ (COX, 1994).

A Figura 5 ilustra a operação união.



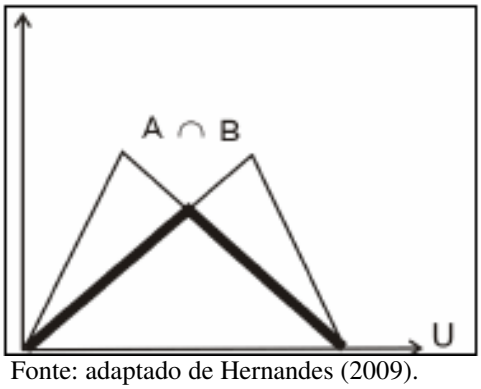
Fonte: adaptado de Hernandes (2009).

Figura 5 – Operação união

2.2.1.3 Intersecção

Cox (2004) afirma que a operação intersecção é utilizada para definir a região comum entre dois subconjuntos, ou seja, a intersecção do subconjunto A com B resulta em um subconjunto abrangendo os pontos que pertencem tanto ao subconjunto A quanto ao subconjunto B. Essa operação, quando tratada nos extremos do intervalo [0, 1], é equivalente à operação "e" da lógica booleana. A representação formal da operação intersecção é descrita na equação $A \cap B \rightarrow \mu A(x) \cap \mu B(x) = \min(\mu A(x), \mu B(x))$.

A operação intersecção é ilustrada na Figura 6.



Fonte: adaptado de Hernandes (2009).

Figura 6 – Operação intersecção

2.2.2 Variáveis lingüísticas

Para a expressão do conceito é muito comum o uso de elementos qualitativos ao invés de valores quantitativos. Elementos típicos incluem "mais ou menos", "alto", "baixo",

"médio", entre outros. Estes elementos são ditos pela definição de variável lingüística. Uma variável lingüística tem por característica assumir valores dentro de um conjunto de termos lingüísticos, ou seja, palavras ou frases (TARIG, 2001).

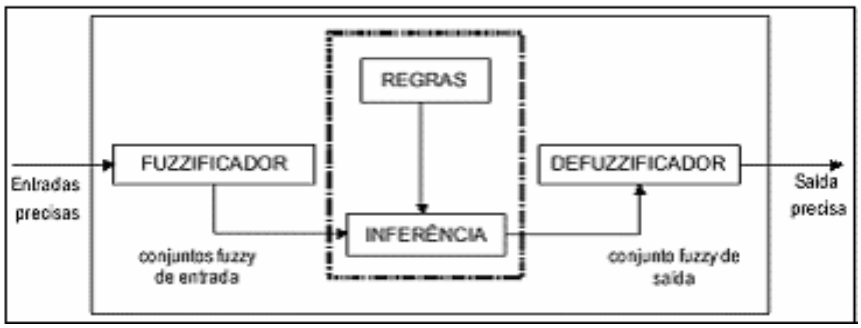
Segundo Pereira (1995), variáveis lingüísticas são variáveis cujos valores são palavras em linguagem natural representadas em conjuntos difusos. Por exemplo, uma variável lingüística altura poderá assumir um dos membros do conjunto {muito alto, alto, médio, médio alto, baixo, muito baixo}. Para se atribuir um significado aos termos lingüísticos, associa-se a cada um deles um conjunto difuso definido sobre um universo de discurso comum.

Para Pacheco (1991) o raciocínio humano é por natureza "aproximado", e qualquer técnica de modelá-lo diferentemente está desprezando a principal vantagem humana, a de tratar diretamente com conceitos inexatos. Daí a importância das variáveis lingüísticas.

2.2.3 Base de regras

A base de regras tem por objetivo representar de forma sistemática a maneira como o controlador gerenciará o sistema sob sua supervisão (DRIANKOV et al, 1996).

A forma mais geral de uma regra lingüística é: Se premissa então conseqüência. Para Driankov et al (1996), as premissas também chamadas de antecedentes são associadas com as entradas do controlador difuso e formam a parte das regras, enquanto as conseqüências, que também são conhecidas como ações, estão associadas às saídas dos controladores. A Figura 7 ilustra um controlador difuso.



Fonte: adaptado de Tarig (2001).

Figura 7 – Sistema de controle difuso

2.2.4 Fuzzyficação

A *fuzzyficação* é o processo de associar ou calcular um valor para representar um grau de pertinência da entrada em um ou mais grupos qualitativos, chamados de conjuntos difusos. O grau de pertinência é determinado por uma função de pertinência que foi definida com base na experiência ou intuição. Funções de pertinência são o meio pelo qual um controlador é sintonizado para alcançar respostas desejadas a determinadas entradas (CABRAL, 1994).

Conforme Fernandes (1997), o tipo e a quantidade de funções de pertinência usadas em um sistema dependem de precisão, estabilidade, facilidade de implementação, manipulação e manutenção.

Portanto, nessa etapa, Mattos (2001), cita que os valores numéricos são transformados em graus de pertinência e associados a uma variável lingüística. Permitindo uma ligação entre os termos lingüísticos (frio, próximo, quente, dentre outros) e as funções de pertinência.

Segundo Mattos (2001), o número e a forma das funções de pertinência em conjuntos difusos são escolhidos dependendo da exatidão, resposta, estabilidade, facilidade de implementação, manipulação e manutenção requeridas pelo sistema.

As funções de pertinência triangulares e trapezóides são as mais comuns, e têm provado serem boas em efetividade e eficiência. Os conjuntos difusos devem abranger o eixo X, cobrindo todo o intervalo, ou o universo de discurso, para uma entrada de um sistema, mapeando para o intervalo de 0 a 1 do eixo Y as pertinências de uma entrada. Sobreposição entre limites de conjuntos é desejável e a chave para a operação suave do controlador. São permitidas pertinências em múltiplos conjuntos (CABRAL, 1994).

2.2.5 Inferência

Depois das variáveis lingüísticas serem interpretadas, por meio da *fuzzyficação*, a próxima etapa é a descrição das situações nas quais há reações, ou seja, a determinação das regras “se-então“.

O lado “se” (*if*) de uma regra contém uma ou mais condições, chamadas antecedentes que constituem uma premissa. O lado “então” (*then*) contém uma ou mais ações chamadas conseqüentes (CABRAL, 1994).

O antecedente da regra contém uma ou mais condições, o conseqüente contém uma ou

mais ações.

O antecedente corresponde diretamente aos graus de pertinência calculados durante o processo de *fuzzyficação*. Cada antecedente tem um grau de pertinência indicado para ele como resultado da *fuzzyficação*. Durante a avaliação das regras (a inferência), a intensidade é calculada com base em valores dos antecedentes e estão indicadas para saídas difusas da regra (FERNANDES, 1997).

2.2.6 Defuzzyficação

A *defuzzyficação* converte um conjunto difuso de saída de um sistema em um valor clássico correspondente. Este processo é importante para decifrar o significado das ações difusas usando funções de pertinência e também para resolver conflitos entre ações de competição (MATTOS, 2001).

A avaliação das regras associa potências (intensidade) para cada ação específica na atividade de inferência. Contudo, um outro processamento, ou *defuzzyficação*, é necessário que seja executado por duas razões: a primeira é decifrar o significado de ações vagas (difusas), utilizando funções de pertinência; a segunda é resolver os conflitos entre ações conflitantes, que podem ter sido acionadas durante certas condições na avaliação das regras (CABRAL, 1994).

2.2.7 Aplicação da lógica difusa

Zadeh (1987), criador da lógica difusa, demonstra em seus trabalhos, a capacidade de tal teoria interpretar os fenômenos não exatos do nosso dia-a-dia. Daí sua aplicabilidade. Apesar disso, os matemáticos não consideram a lógica difusa como uma lógica matemática nos padrões atuais, ou seja, ela não é, ainda, uma teoria matemática perfeitamente consistente e completa, uma vez que ela deixa de satisfazer algumas propriedades da lógica clássica, principal responsável para ditar a veracidade de uma teoria matemática. Isso, no entanto, não inviabiliza a lógica difusa como uma teoria matemática perfeitamente aplicada, segundo Cruz (1996).

Segundo Ivanqui (2005), existem diversas áreas que estão sendo beneficiadas pelo uso da lógica difusa, a exemplo:

a) em câmeras de vídeo, são aplicados ao foco automático e ao controle da íris da câmera;
b) máquinas de lavar com a utilização de sensores de temperatura da água, concentração de detergente, peso das roupas, nível de água, etc;
c) em fornos de microondas, com informações obtidas a partir de sensores;
d) aparelhos de ar-condicionado com o controle da umidade e temperatura;
e) manutenção de motores elétricos, verificação das condições de vibração dos motores com a finalidade de estabelecer procedimentos de manutenção.

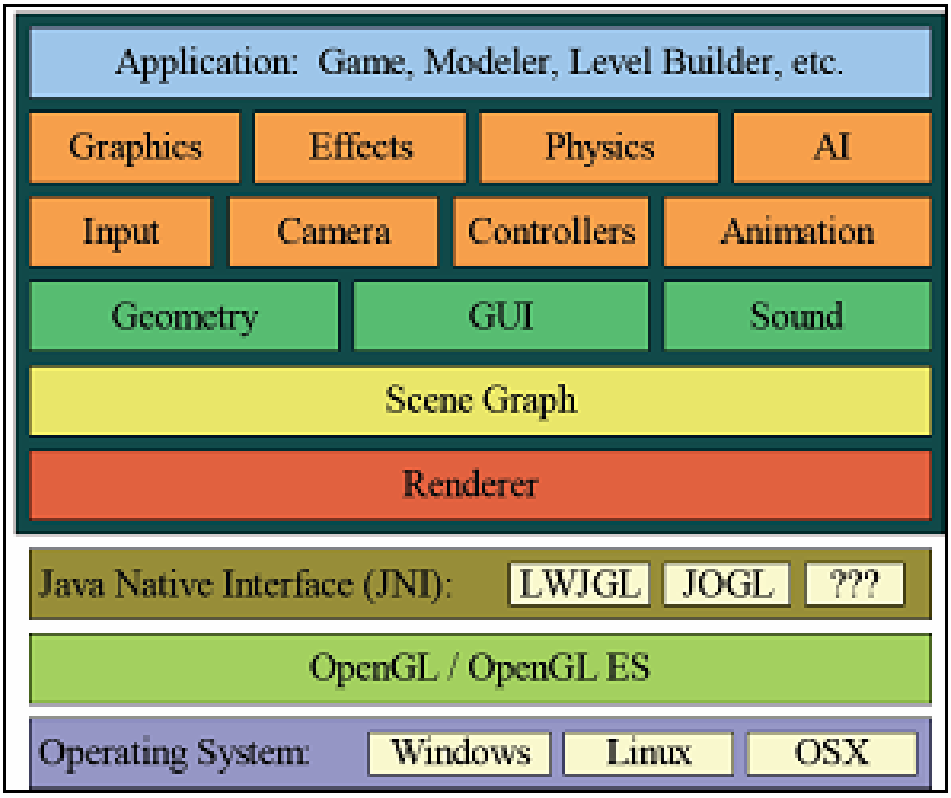
Para Barbosa (1995), a aplicação da lógica difusa no controle de motores CC, facilitou, sensivelmente, o desenvolvimento de tais aparelhos, tornando-os mais precisos, capazes de interpretar com mais exatidão os fenômenos elétricos de um motor.

Tanaka e Mizumoto (1974) consideram a lógica difusa, no que diz respeito aos modelos matemáticos, muito mais adequada a programação, sugerindo softwares difusos de fácil entendimento, capazes de serem usados em várias simulações reais.

2.3 JMONKEY ENGINE (JME)

A JME é uma *engine* utilizada para o desenvolvimento de aplicações gráficas 3D. Possui a característica de ser *freeware* e *open-source*, e propõe um nível alto de abstração ao desenvolvedor, fazendo com que o mesmo não se preocupe com aspectos de baixo nível como chamada de funções da biblioteca nativa. Por ser totalmente escrita em Java, a JME possui a característica de ser interpretada. Sendo assim, pode-se considerar a JME como multiplataforma.

A Figura 8 exibe as camadas da arquitetura JME.



Fonte: adaptado de Patton (2004).

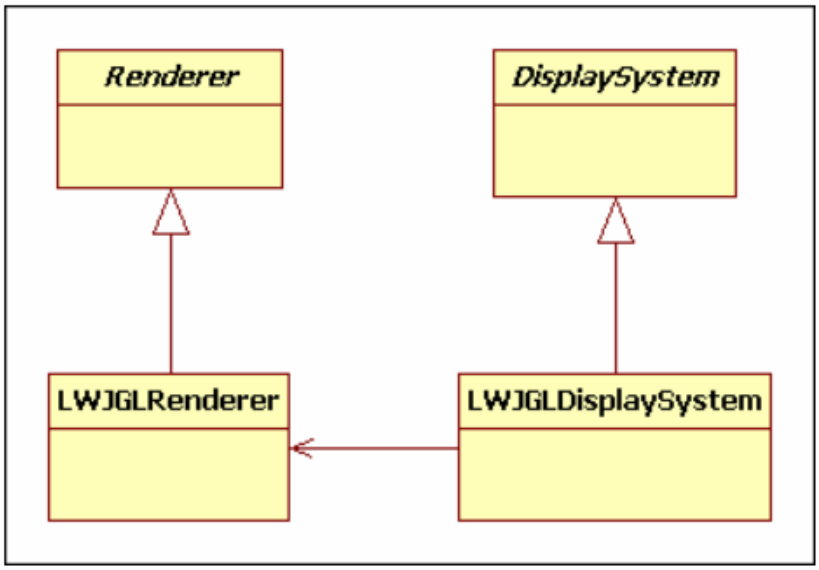
Figura 8 – Arquitetura da JME

A camada OpenGL/OpenGL ES abrange a biblioteca de *renderização*. Atualmente a única biblioteca suportada é a OpenGL. Escrita na linguagem de programação C, consiste em uma biblioteca de alta performance para construção de aplicações gráficas interativas 2D ou 3D (OPENGL OVERVIEW, 2008).

Fronza (2008) diz que a interface nativa Java (Java *Native Interface* – JNI) acopla a biblioteca nativa de *renderização* a *engine* JME. Pela JNI, é possível que um código escrito em Java utilize a implementação de uma biblioteca em outra linguagem qualquer, desde que essa possa gerar bibliotecas nativas (arquivos binários). Para a biblioteca OpenGL, as implementações de JNIs mais conhecidas são *a Light Weight Java Game Library* (LWJGL) e a JOGL. Porém atualmente a JME opera somente com a primeira opção.

A camada de *Renderer* é possui os dados da cena representados por coordenadas de mundo para coordenadas de espaço. Essa é a primeira camada que faz parte efetivamente da *engine* JME, interagindo diretamente com a biblioteca JNI de OpenGL para desenhar na tela os dados transformados em coordenadas de espaço. Esta camada também possuí a responsabilidade de desconsiderar objetos que estão fora da visão do observador (câmera). Esta técnica é chamada de *culling*. Outro tipo de otimização, chamado de *clipping*, divide um objeto em pequenas porções para desconsiderar as que não estão correntemente visíveis na perspectiva de visão do observador (FRONZA, 2008).

A Figura 9 ilustra a especialização *renderizador* LWJGL.

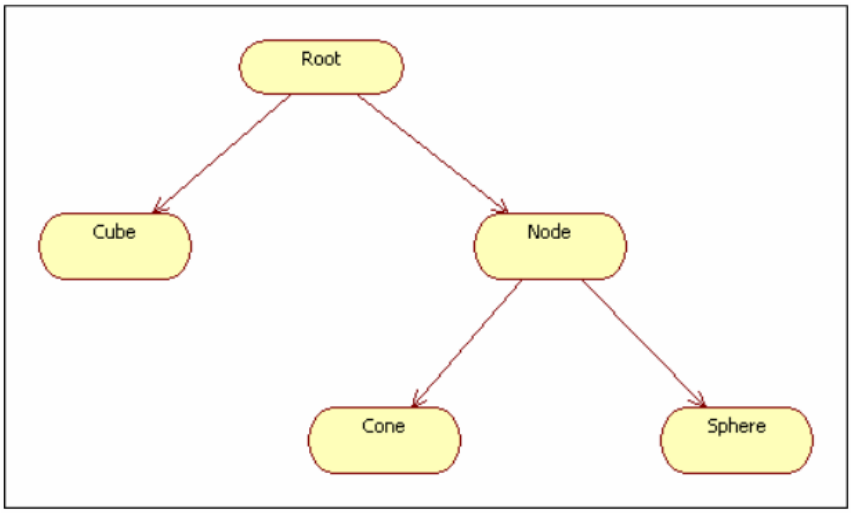


Fonte: adaptado de Patton (2004).

Figura 9 – Diagrama de classe do renderizador LWJGL

Através das classes `Renderer` e `DisplaySystem` que acoplamento entre a *engine* JME e a camada de JNI é garantido. Dessa forma, quando um novo *renderizador* precisar ser suportado pela JME, basta criar as respectivas especializações das classes abstratas. Mais adiante é mostrado como e em qual momento essas classes concretas são instanciadas.

A camada de grafo de cena (do inglês, *scene graph*) provê uma maneira de representar os dados do ambiente virtual de forma hierárquica. Patton (2004) afirma que utilizar estruturas de grafos de cena simplifica o gerenciamento de estruturas globais. A Figura 10 apresenta graficamente um exemplo de grafo de cena, envolvendo objetos simples como cubos, cones e esferas. Ainda na Figura 10, os nós denominados `Root` e `Node` são considerados nós de agrupamento, sendo que não são objetos visuais.



Fonte: adaptado de Patton (2004).

Figura 10 – Exemplo de grafo de cena

A camada seguinte, composta por Geometria (*Geometry*), Interface de Usuário (GUI) e Sonorização (*Sound*), provê o penúltimo nível de abstração para o desenvolvedor da aplicação. O componente de geometria, representado pela classe base `Geometry`, define um nó folha utilizado para representar objetos geométricos no grafo de cena. No diagrama de classes da Figura 11, podem ser vistas as classes estendidas de `Geometry` que acompanham a JME por padrão. O componente de interface de usuário é a implementação padrão da JME para a criação de janelas, botões, inputs, entre outros componentes padrões de interface. Por último, o componente de sonorização funciona da mesma forma que a camada de *renderização* (*Renderer*), ou seja, conectada a uma biblioteca nativa. Atualmente esse componente da JME opera com as bibliotecas LWJGL e JOAL[4] (FRONZA, 2008).

A última camada de abstração é composta por oito componentes. Os componentes de gráficos e de efeitos são responsáveis, respectivamente, pela importação de modelos de ferramentas externas e tratamento de texturas, e pela criação e gerenciamento de sistemas de partículas, tal como efeitos de fogo, fumaça e explosão. O componente Input é responsável por prover uma interface dos comandos de teclado e mouse com a aplicação. A câmera é utilizada para definir o campo de visão do usuário. O componente controlador é utilizado para gerenciar objetos animados, que freqüentemente são importados de ferramentas externas, tal como 3D Studio Max e Blender.

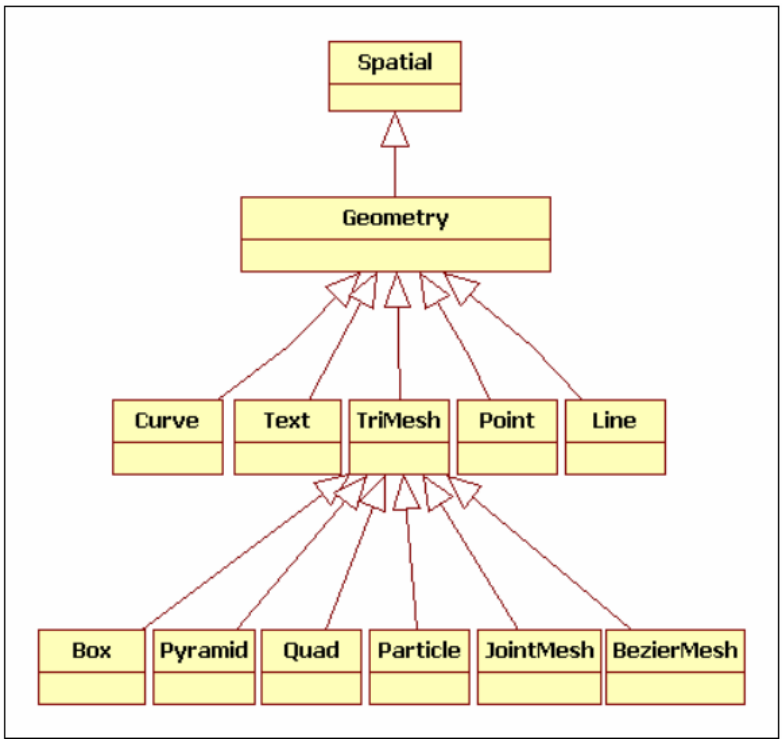


Figura 11 – Diagrama de classes do componente de geometria da JME

[4] JOAL é uma implementação JNI para utilização da biblioteca nativa OpenAL.

Ainda tratando-se da última camada de abstração, Fronza (2008) diz que o componente mais relevante é o que controla a física dos objetos de uma aplicação JME. Questões de física não são diretamente tratadas pela *engine* JME, essa tarefa é delegada a um *framework* de física específico, chamado JME Physics. Esse *framework* é responsável por prover uma interface de alto nível entre a *engine* JME e a Open Dinamics Engine (ODE), sendo essa uma biblioteca de alta performance escrita na linguagem de programação C, utilizada para simulação de corpos rígidos dinâmicos.

2.3.1 Principais classes e interfaces

Além das classes já citadas, a *engine* JME e o *framework* JME Physics é composto de várias classes que visam facilitar o desenvolvimento de aplicações gráficas escritas em Java, assim como pode ser visto no Quadro 2.

CLASSE/INTERFACE	**DESCRIÇÃO/FUNCIONALIDADES**
`BaseGame`	Principal classe da aplicação JME. O *loop* principal da aplicação está nesta classe.
`DisplaySystem`	Interface para a criação do sistema. Através dela pode ser obtida a referência para o *renderizador*.
`StatisticsGameState`	Provê um *game state* com estatísticas de *frames* por segundo, número de vértices e número de triângulos.
`PhysicsGameState`	Provê um *game state* responsável pelo controle da física dos objetos estáticos e dinâmicos.
`PhysicsSpace`	Classe central do *framework* JME Physics. Utilizado para criar objetos estáticos e dinâmicos.
`Node`	Provê um nó de agrupamento para o grafo de cena.
`DynamicPhysicsNode`	Mesmas funcionalidades da classe `Node`, porém essa é controlada pela física. Contém tratamento de colisão e é afetada pela gravidade do mundo.
`StaticPhysicsNode`	Mesmas funcionalidades da classe `DynamicPhysicsNode`, porém essa não é afetada pela gravidade do mundo.
`Vector3f`	Provê métodos para manipulação de uma coordenada 3D.
`InputHandler`	Utilizado para manipular eventos de teclado e mouse.
`ChaseCamera`	Classe herdada de `InputHandler`. Câmera utilizada para seguir um determinado objeto da cena. Possui eventos de mouse e teclado pré-definidos para rotação e zoom.
`Texture`	Classe que representa uma textura na *engine* JME.
`TerrainPage`	Classe utilizada para gerar malha de polígonos para terrenos, baseado em um "mapa de alturas".

Quadro 2 – Principais classes da *engine* JME e do *framework* JME Physics

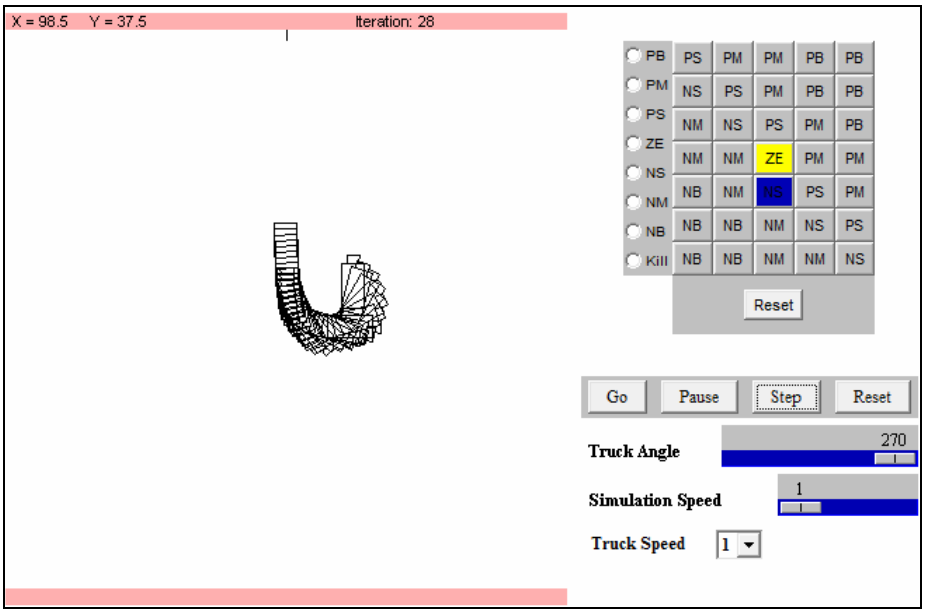
2.4 TRABALHOS CORRELATOS

Existem alguns projetos semelhantes ao atual trabalho desenvolvido. Dentre eles, foram escolhidos três cujas características se enquadram nas três principais áreas de estudos desse trabalho.

Em Lima (2005) é mostrado um sistema *fuzzy* otimizado baseado em modelos de controle previamente desenvolvidos. O sistema permite guiar um robô simulado através de um mundo virtual, que contém obstáculos randomicamente distribuídos. O objetivo principal é investigar o desempenho do controlador estudando sua estabilidade e robustez.

Tan e Seng (2007) desenvolveram um protótipo de um veículo que também faz o processo de estacionamento autonomamente (Figura 1). Foi utilizada a arquitetura do *kit* LEGO Mindstorms e o algoritmo foi desenvolvido no ambiente de desenvolvimento de aplicações que acompanha o *kit* LEGO. O veículo anda em linha reta a procura de um espaço mínimo em sua direita. Esse espaço mínimo é de aproximadamente cinco centímetros ou mais do tamanho do protótipo. Ao encontrar um espaço entre dois objetos, o veículo tem que decidir se é suficiente para estacionar ou não. Caso o espaço encontrado seja suficiente, o veículo inicia a manobra autonomamente. Caso contrário, ele continua sua procura.

Britton (2000) desenvolveu um algoritmo para estacionar de ré um caminhão articulado. Um cenário virtual foi criado e o caminhão deve estacionar em um ponto pré-definido no mapa. O algoritmo utiliza técnicas de lógica difusa para movimentação e controle do veículo. O cenário é baseado em uma matriz bidimensional e o algoritmo leva em consideração o posicionamento e o ângulo do caminhão em relação ao ponto onde o caminhão deve estacionar.



Fonte: Adaptado de Britton (2000).

Figura 12 – Simulador modelado

3 DESENVOLVIMENTO

Para detalhar o processo de desenvolvimento serão abordados os temas a seguir:

a) análise e especificação dos requisitos;
b) especificação através de diagramas de classe e casos de uso;
c) estratégia utilizada para estacionamento do veículo.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O simulador deverá contemplar os seguintes requisitos funcionais:

a) estacionar um veículo autonomamente em espaços paralelos ao veículo;
b) procurar em linha reta um espaço adequado para estacionar;
c) parar o veículo após o mesmo estar estacionado;
d) continuar procurando uma vaga, caso ache uma não adequada;
e) estacionar o carro em vagas no lado direito, excluindo cenários com mão inglesa.

O simulador deverá contemplar os seguintes requisitos não funcionais:

a) ser implementado utilizando o ambiente de desenvolvimento Eclipse Europa;
b) ser implementado utilizando a linguagem de programação Java;
c) utilizar a *engine* gráfica JMonkey Engine para a parte gráfica do simulador.

3.2 ESPECIFICAÇÃO

Nesta seção serão apresentados os diagramas de casos de uso e diagramas de classe. Os diagramas foram desenvolvidos utilizando a ferramenta Enterprise Architect versão 7.5 da empresa Sparx Systems.

3.2.1 Diagrama de casos de uso

Os diagramas de casos de uso apresentados nesta seção tem como ator o usuário que fará a visualização do veículo através do simulador e o veículo ao receber as informações de sua localização no mapa.

Na Figura 13 é descrito o usuário como ator que dará a comando inicial para o veículo iniciar o processo de estacionamento.

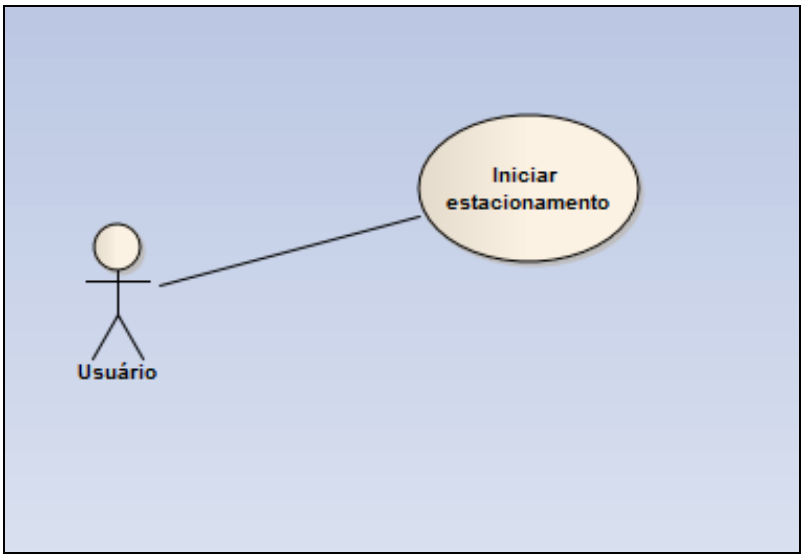


Figura 13 – Diagrama de casos de uso do ator usuário

Na Figura 14 o ator é o próprio veículo que receberá seu posicionamento no mapa e tomará suas próprias decisões.

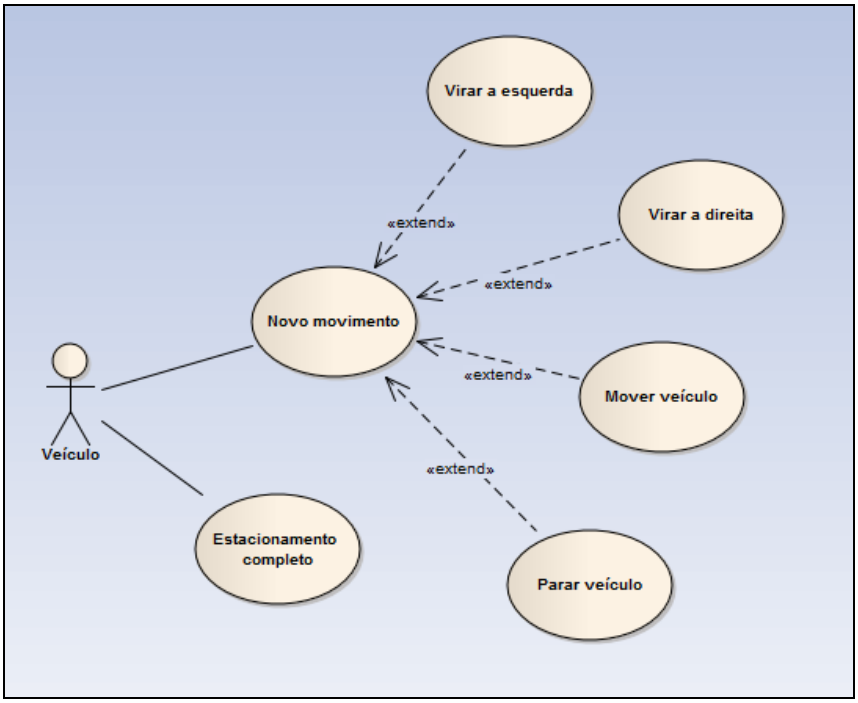


Figura 14 – Diagrama de casos de uso do ator veículo

3.2.2 Diagrama de classes

Os diagramas de classes apresentados nesta seção demonstram a estrutura de dados para a construção do simulador.

Na Figura 15 é apresentado resumidamente o diagrama de classes, com seus atributos e métodos, das classes utilizadas para geração do ambiente simulado e do ato de estacionar o veículo.

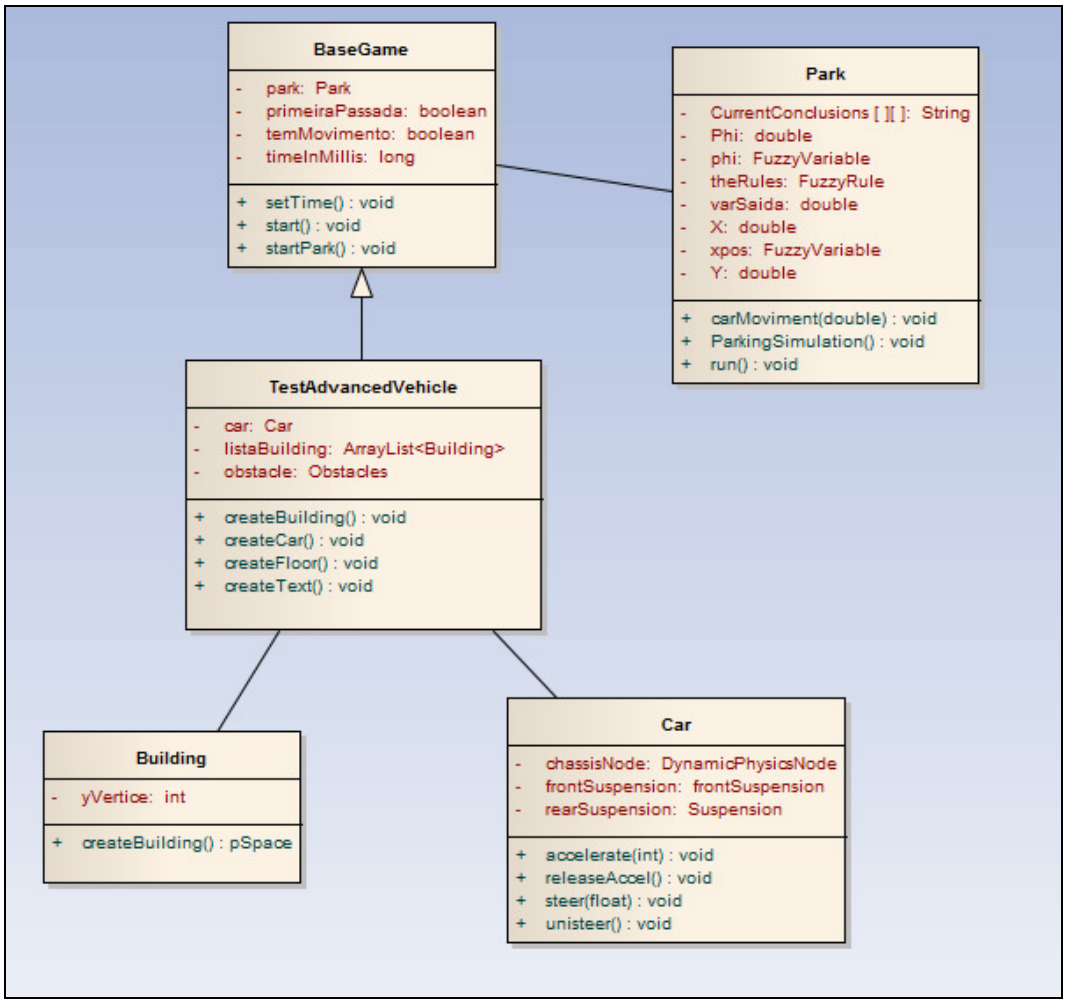


Figura 15 – Diagrama das classes responsável pela criação do ambiente virtual e estacionamento

Na Figura 16 é apresentado o diagrama de classes das classes responsáveis pela criação do protótipo do veículo.

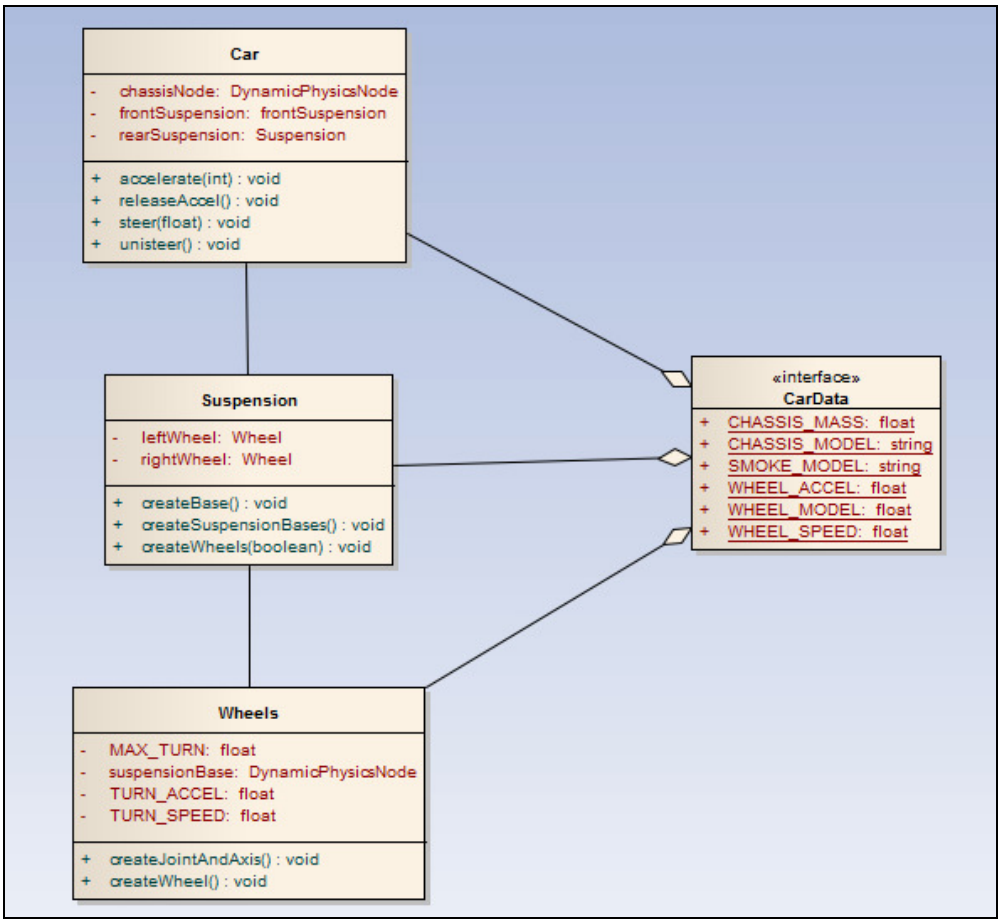


Figura 16 – Diagrama das classes responsável pela criação do veículo

A seguir cada uma das classes é apresentada em detalhes.

3.2.2.1 Classe `BaseGame`

A classe `BaseGame` é utilizada como superclasse da classe principal do simulador. É responsabilidade desta classe o controle do *loop* principal da aplicação, que consiste basicamente em atualizar e *renderizar* o grafo de cena.

Possui como principais atributos:

a) `park`: responsável pelo inicio do estacionamento autônomo baseado na lógica difusa;

b) `primeiraPassada`: dentro do *loop* principal do programa, garante que seja declarado apenas uma vez as variáveis *fuzzy*. As partir da segunda passada no *loop* principal, apenas são gerados novos movimentos para o veículo baseado nas variáveis já instanciadas na primeira passagem;

c) `temMovimento`: dentro do *loop* principal do programa, informa se o veículo está

parado ou em movimento. Só é gerado um novo movimento quando esta variável possui valor falso;

d) `timeInMillis`: recebe o exato momento que foi iniciado um movimento no veículo.

Possui como principais métodos:

a) `start`: início da aplicação JME. Dentro deste método está o *loop* principal da aplicação;

b) `startPark`: este método inicia o ato de estacionar. Instância as variáveis *fuzzy* e retorna qual movimento deve ser feito pelo veículo;

c) `setTime`: atualiza o tempo exato de geração de um novo movimento do veículo.

3.2.2.2 Classe `TestAdvancedVehicle`

A classe `TestAdvancedVehicle` é uma extensão da classe `BaseGame`. É a classe principal no que se refere à criação do ambiente simulado. Através desta classe é criado o terreno, o veículo e obstáculos. Também possui os eventos de teclado e mouse pré-definidos.

Dentre seus atributos, deve-se destacar o `car`, atributo este que é o veículo simulado e o `listaBuilding`, que é uma lista contendo todos os obstáculos gerados aleatoriamente.

Possui como principais métodos:

a) `createBuilding`: cria um novo obstáculo já como objeto físico;

b) `createCar`: responsável pela criação do veículo passando como parâmetro um novo espaço físico;

c) `createFloor`: é através deste método que é criado o terreno onde o veículo e os obstáculos estarão localizados.

3.2.2.3 Classe `Building`

A classe `Building` é responsável pela criação de cada obstáculo gerado na tela. Cada um destes obstáculos é representado fisicamente através de um espaço físico estático. Recebe como parâmetro a posição em que deve estar localizado no mapa (`yVertice`).

O método `createBuilding` cria uma nova `Box` para o obstáculo passa a textura a ser

utilizada para o mesmo.

3.2.2.4 Classe `Park`

A classe `Park` é a classe principal da lógica difusa. É através dela que são passadas as coordenadas para o simulador *fuzzy*, que retorna o próximo movimento a ser realizado pelo veículo.

Possui como principais atributos:

a) `X` e `Y`: responsáveis pela localização do veículo no terreno;
b) `varSaida`: variável de saída da lógica difusa. Recebe a direção em que o carro deve realizar o próximo movimento;
c) `currentConclusions`: vetor de duas dimensões responsável pelo conjunto de regras *fuzzy*.

Possui como principais métodos:

a) `ParkingSimulation`: responsável por instanciar as variáveis *fuzzy* e informar os limites dentro do mapa no qual o simulador *fuzzy* irá atuar;
b) `run`: rotina principal de atuação da lógica difusa. Ao passar como parâmetro a localização do veículo no mapa, o simulador *fuzzy* retorna qual movimento deve ser feito pelo veículo;
c) `carMovement`: recebe como parâmetro a variável *fuzzy* de saída e de acordo com o valor, realiza os movimentos para o lado definido nas regras *fuzzy* (`currentConclusions`).

3.2.2.5 Classe `Car`

A classe `Car` é responsável pela montagem do veículo. É formado basicamente por duas suspensões (`frontSuspension` e `rearSuspension`) e por seu chassi (`chassisNode`).

Possui como métodos:

a) `accelerate`: acelera o veículo para frente ou para trás, dependendo do valor recebido como parâmetro. Recebendo um valor positivo, move o veículo para frente. Caso contrário, para trás;

b) `releaseAccel`: responsável por parar o veículo;
c) `steer`: move o veículo para a esquerda ou para a direita;
d) `unsteer`: corrige o ângulo das rodas para o veículo mover-se apenas reto.

3.2.2.6 Classe `Suspension`

A classe `Suspension` cria duas suspensões para o veículo, onde cada suspensão receberá as rodas. Em cima destas suspensões (`suspensionBase`) é colocado o chassi, dando forma ao veículo.

Como principais atributos, `leftWheel` e `rightWheel` (roda esquerda e direita) se destacam como principais.

3.2.2.7 Classe `Wheel`

A classe `Wheel` é responsável pela criação das rodas do veículo. Cada roda está ligada a uma suspensão descrita anteriormente.

Possui como atributos:

a) `suspensionBase`: suspensão em que a roda esta posicionada;
b) `MAX_TURN`: determina o ângulo máximo de inclinação da roda;
c) `TURN_ACCEL`: aceleração máxima permitida para as rodas;
d) `TURN_SPEED`: velocidade máxima de cada roda e conseqüentemente velocidade máxima também do veículo.

Os métodos existentes nesta classe servem para criar as rodas e anexar as suspensões dianteira e traseira.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as ferramentas e técnicas utilizadas e a operacionalidade da implementação.

3.3.1 Tecnologias e ferramentas usadas

Para a implementação do simulador na linguagem Java, foi utilizado o ambiente de desenvolvimento Eclipse Europa. Também foi usada a ferramenta Blender 2.43 para modelagem do veículo 3D. Para a geração das texturas utilizadas no terreno do cenário 3D foi usada a ferramenta Adobe Photoshop CS.

3.3.2 Implementação gráfica do simulador

O início do aplicativo ocorre através do método *main* da classe `TestAdvancedVehicle`. A única tarefa desse método é invocar o construtor da classe, que inicializa os recursos necessários para exibição do simulador. O Quadro 3 exibe um trecho de código da classe `TestAdvancedVehicle`, evidenciando os passos iniciais do aplicativo.

```
public class TestAdvancedVehicle extends SimplePhysicsTest {

	private Skybox skybox;
	// Objeto carro
	public static Car car;
	protected Obstacles obstacle;
	private InputAction resetAction;
	private InputAction parkingAction;
	public int xTamanhoMapa = 1000;
	public int yTamanhoMapa = 1000;

	public static void main(String[] args) {
		Logger.getLogger("").setLevel(Level.WARNING);
		new TestAdvancedVehicle().start();
	}
	...
}
```

Quadro 3 – Trecho de código da classe `TestAdvancedVehicle`

Para a criação do mundo virtual, o método `simpleInitGame` da classe `TestAdvancedVehicle` invoca os métodos responsáveis pela criação de cada objeto do

cenário. O Quadro 4 exibe o método `simpleInitGame`.

```
protected void simpleInitGame() {

	tunePhysics();
	// Cria o terreno
	createFloor();
	createCar();
	createBuilding();
	// Eventos de teclado
	initInput();
	// Textos de tela
	createText();

	rootNode.updateRenderState();
	resetAction.performAction(null);
}
```

Quadro 4 – Método `simpleInitGame`

A criação do terreno inicia pela invocação do método `createFloor`, onde é criado um objeto do tipo `Box`, onde o tamanho do terreno é passado pelos parâmetros `xTamanhoMapa` e `yTamanhoMapa`.

Para a representação física do terreno, é criado um nó estático (controlado pela física) onde é anexada a malha de triângulos gerada. Nesse nó também é definido um material tipo concreto (`Material.CONCRETE`), que é o material a ser usado para que o *framework* de física possa prover o aspecto de atrito entre o terreno e os demais objetos sobre ele (no caso, as rodas do veículo).

Para a criação do terreno, é executado o código do Quadro 5.

```
private void createFloor() {
	Spatial floorVisual = new Box("floor", new Vector3f(),
		xTamanhoMapa, 0.1f,yTamanhoMapa);
	floorVisual.setModelBound(new BoundingBox());
	floorVisual.updateModelBound();
	StaticPhysicsNode floor = getPhysicsSpace().createStaticNode();
	floor.attachChild(floorVisual);
	floor.generatePhysicsGeometry();
	floor.setMaterial(Material.CONCRETE);
	floor.setLocalTranslation(new Vector3f(0, -0.1f, 0));
	rootNode.attachChild(floor);

	// Anexa a textura ao terreno.
	final TextureState wallTextureState = display.getRenderer()
			.createTextureState();
	wallTextureState.setTexture(TextureManager.loadTexture(

	jmetest.TestChooser.class.getResource("data/texture/dirt.jpg"),
		Texture.MM_LINEAR, Texture.FM_LINEAR));
	wallTextureState.getTexture().setScale(new Vector3f(256, 256, 1));

	wallTextureState.getTexture().setWrap(Texture.WM_WRAP_S_WRAP_T);
	floorVisual.setRenderState(wallTextureState);

	}
```

Quadro 5 – Método `createFloor`

A criação do veículo acontece através do método `createCar`, que nada mais é do que a criação de um objeto `Car`. O Quadro 6 exibe o construtor do veículo na classe `Car`.

```
Public class Car extends Node {

    private static final long serialVersionUID = 1L;

    // Nodo que representa o chassi do carro
    public DynamicPhysicsNode chassisNode;

    // Duas suspensões
    public Suspension rearSuspension, frontSuspension;

    public Car( final PhysicsSpace pSpace ) {
        super( "car" );
        createChassi( pSpace );
        createSuspension( pSpace );
        loadFancySmoke();
    }
    ...
}
```

Quadro 6 – Trecho de código da classe `Car`

A criação do veículo consiste no processo mais demorado que ocorre durante a criação do cenário, pois é nesse momento que são carregados os modelos 3D referentes à estrutura do veículo, chassi e rodas.

A criação do chassi e da suspensão do tanque acontece no código apresentado no Quadro 7.

```
public void createSuspension( final PhysicsSpace pSpace ) {
    frontSuspension = new Suspension( pSpace, chassisNode,
        CarData.FRONT_SUSPENSION_OFFSET, true );
    this.attachChild( frontSuspension );
    rearSuspension = new Suspension( pSpace, chassisNode,
    CarData.REAR_SUSPENSION_OFFSET, false );
    this.attachChild( rearSuspension );
}

private void createChassi( final PhysicsSpace pSpace ) {
    chassisNode = pSpace.createDynamicNode();
    chassisNode.setName( "chassiPhysicsNode" );

    // Modelo de carro importado através da classe carData
    Node chassisModel = Util.loadModel( CarData.CHASSIS_MODEL );
    chassisModel.setLocalScale( CarData.CHASSIS_SCALE );

    chassisNode.attachChild( chassisModel );

    chassisNode.generatePhysicsGeometry( true );
    chassisNode.setMaterial( Material.IRON );
    chassisNode.setMass( CarData.CHASSIS_MASS );
    this.attachChild( chassisNode );
}
```

Quadro 7 – Criação do chassi e da suspensão do veículo

Na criação do chassi dinâmico, nota-se que é obtido o modelo 3D que representa graficamente o chassi do veículo. Isso é feito através da invocação do atributo estático `CHASSIS_MODEL` da classe `CarData`. O método `loadModel` carrega um arquivo com extensão ".jme" e converte para um objeto da classe `Node` que pode ser anexado ao grafo de cena como um objeto nativo criado na JME. Além disso, esse método armazena internamente o conteúdo do arquivo carregado, para que na próxima vez que for invocado, não seja necessário carregá-los outra vez. Ainda na criação do chassi, nota-se também que é definida uma massa para o nó dinâmico (`setMass`). Essa informação é utilizada pelo *framework* JME Physics para controlar as colisões e também afetar o objeto pela gravidade ainda a ser definida.

3.3.3 Estratégia para estacionamento do veículo

A estratégia para o estacionamento do veículo é baseada nas seguintes premissas e conclusões.

a) posicionar o veículo paralelamente ao espaço onde se deseja estacionar;

b) passar as coordenadas de onde está localizado o veículo;

c) realizar as manobras necessárias evitando colisões;

d) parar o veículo ao concluir a manobra.

Com o veículo paralelamente posicionado ao espaço onde ele deve estacionar, um evento de teclado (tecla "M") da ordem ao início do estacionamento autônomo. O Quadro 8 exibe o evento que dá início ao processo de estacionar.

```
private void initInput() {
      ...
      /*(Tecla "M") que dá início ao estacionamento autônomo*/
      parkingAction = new StartParking();
      input.addAction(parkingAction, InputHandler.DEVICE_KEYBOARD,
                        KeyInput.KEY_M, InputHandler.AXIS_NONE, false);
      }

...

private class StartParking extends InputAction {
      public void performAction(InputActionEvent evt) {

            startParking = true;

      }

}
```

Quadro 8 – Trecho de código da classe `TestAdvancedVehicle`

Ao receber o evento do teclado, o atributo `startParking` recebe valor verdadeiro. No *loop* principal da aplicação na classe `BaseGame`, o processo de estacionamento só é iniciado quando este atributo for verdadeiro. Lembrando que a classe `startParking` é uma extensão da classe `BaseGame`.

Ao iniciar o processo de estacionamento, são instanciadas as regras difusas e é calculada a posição inicial do veículo. O Quadro 9 apresenta a criação do objeto `Park`.

```
private void startPark() {

  if (primeiraPassada) {
      primeiraPassada = false;
      park = new Park();
      park.setCar();
      park.ParkingSimulation();
      park.setVars();
  }

  if (temMovimento){
      if (System.currentTimeMillis() > (timeInMilis + 500)) {
            park.parar();
            temMovimento = false;
          }
  } else {
      if (!park.simulationFinished()){
            park.setVars();
            temMovimento = true;
            park.run();
      }
  }
}
```

Quadro 9 – Método `startPark`

No quadro acima, observa-se que se for o primeiro movimento autônomo do veículo, é instanciado o objeto `Park`. No método `setCar`, apenas é criado um objeto `Car` dentro da classe `BaseGame`. Já o método `setVars` grava a posição do momentânea do veículo que após será passada ao simulador difuso. Por fim, o método `ParkingSimulation` define o *range* de valores para cada variável difusa e define as regras difusas.

O Quadro 10 exibe a definição das variáveis difusas de entrada e saída na classe `Park`. Já o Quadro 11 mostra como é obtida a posição do veículo no terreno também na classe `Park`.

```
public void ParkingSimulation () {
      //parent = p;
      int i, j;
      try
      {
        phiFzSets[0] = new RFuzzySet(-10.0, -7.0, new
                                    RightLinearFunction());
        phiFzSets[1] = new TriangleFuzzySet(-7.0, -4.5, -2.0);
        phiFzSets[2] = new TriangleFuzzySet(-2.0, 0.0, 2.0);
        phiFzSets[3] = new TriangleFuzzySet(2.0, 4.5, 7.0);
        phiFzSets[4] = new LFuzzySet(7.0, 10.0, new]
                                    LeftLinearFunction());
        changePhiFzSets[0] = new TriangleFuzzySet(-45.0, -25.0, -5.0);
        changePhiFzSets[1] = new TriangleFuzzySet(-10.0, 0.0, 10.0);
        changePhiFzSets[2] = new TriangleFuzzySet(5.0, 25.0, 45);

        //var de entrada - x
        xposFzsets[0] = new RFuzzySet(420.0, 500.0, new
                                     RightLinearFunction());
        xposFzsets[1] = new TriangleFuzzySet(490.0, 508.0, 525.0);
        xposFzsets[2] = new TriangleFuzzySet(520.0, 530.0, 540.0);
        xposFzsets[3] = new TriangleFuzzySet(535.0, 550.0, 565.0);
        xposFzsets[4] = new LFuzzySet(560.0, 590.0, new
                                     LeftLinearFunction());
        ...
}
```

Quadro 10 – Trecho de código do método `ParkingSimulation`

```
public void setVars() {

      double posDianteira    = 0,
             posCentro       = 0;

      //atualiza as vars de entrada a cada passada meio segundo.
      posDianteira =
      ((double)car.frontSuspension.leftBase.getLocalTranslation().z +
      (double)car.frontSuspension.rightBase.getLocalTranslation().z) / 2;

      posCentro = car.getPositionZ();
      Yt = -1*(int)((double)(posCentro - posDianteira));
      Xt = (int)car.getPositionZ();
}
```

Quadro 11 – Método `setVars`

Existem ao todo duas variáveis *fuzzy* de entrada. A variável `phiFzSets` é a posição do carro em relação ao eixo X do mapa e recebe o valor estático da variável `Xt`. Já a outra variável de entrada, `xposFzsets`, é a angulação do carro em relação ao eixo X do mapa e recebe o valor estático da variável `Yt`. Ao receber os valores estáticos de entrada, a fase de *fuzzyficação* identifica em qual intervalo difuso, descrito no Quadro 10, este valor melhor se encaixa.

Com o processo de estacionamento autônomo iniciado e suas variáveis já instanciadas, a cada meio segundo é passado ao simulador *fuzzy* uma nova atualização das variáveis de entrada, retornando através da variável de saída `changePhiFzSets` um novo movimento a ser feito pelo veículo.

O Quadro 12 descreve a interpretação da variável de saída na classe `Park`.

```
Private void carMoviment(double varSaida) {

      if (varSaida >= -10){
            if (varSaida <= 10){
                  moverReto();
            }

            else {
                  moverEsquerda();
            }
      }
      else {
            moverDireita();

      }
}
```

Quadro 12 – Método `carMoviment`

Cada ação a ser tomada vinda do simulador *fuzzy* determina a direção a ser realizada pelo veículo. Se a variável *fuzzy* de saída retornar um valor entre 10 e -10, significa que o veículo deve mover-se em linha reta. Para valores maiores que 10 o veículo move-se à esquerda, e para valores menores que -10, as rodas são viradas para a direita. O Quadro 13 mostra como cada movimento é gerado.

```
public void moverEsquerda() {
      car.accelerate( -1 );
      car.steer(-1);
}
      public void moverDireita() {
      car.accelerate( -1 );
      car.steer(1);

}
      public void moverReto() {
      car.accelerate( -1 );
}
```

Quadro 13 – Métodos `moverEsquerda`, `moverDireita` e `moverReto`

O processo de geração de movimentos descrito anteriormente só é encerrado quando o veículo está paralelo aos obstáculos e entre os mesmos. O Quadro 14 ilustra um estacionamento encerrado.

```
Public boolean simulationFinished() {
      if (Yt == 0)  {
            if (Xt > 528){
                  if (Xt < 531)
                        return true;
            }
      }
      return false;
}
```

Quadro 14 – Método `simulationFinished`

3.3.4 Operacionalidade da implementação

Esta seção descreve o funcionamento do simulador através de um estudo de caso. Ao iniciar o simulador, o usuário informa quais características de visualização do simulador melhor lhe convém. A Figura 17 demonstra estes parâmetros a serem informados pelo usuário.

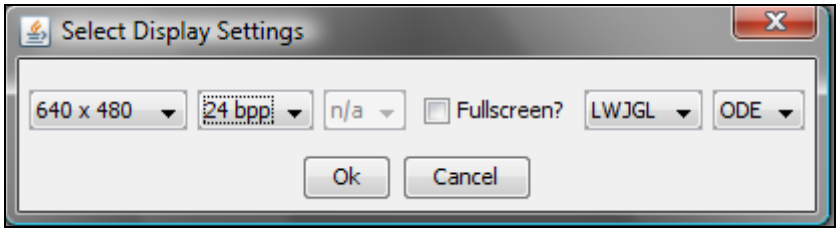


Figura 17 – Parâmetros de visualização do simulador

Após passar as informações descritas acima, é exibido o ambiente 3D do simulador. O veículo encontra-se em uma posição não adequada para iniciar o estacionamento. A Figura 18 ilustra a posição inicial do veículo.

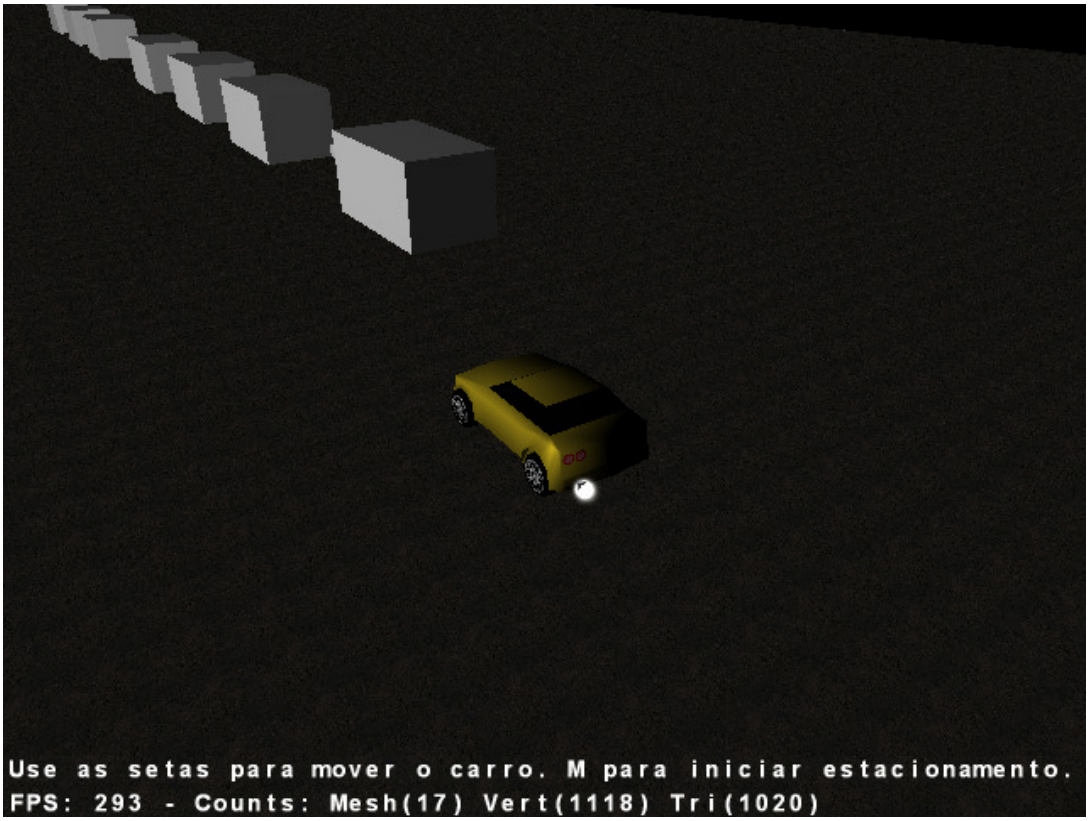


Figura 18 – Posição inicial do veículo

Com o veículo em uma posição paralela aos obstáculos e com espaço suficiente para o estacionamento, o usuário pressiona a tecla “M” do teclado para iniciar o estacionamento autônomo.

Em destaque na Figura 19 uma posição considerada ideal para iniciar o estacionamento.

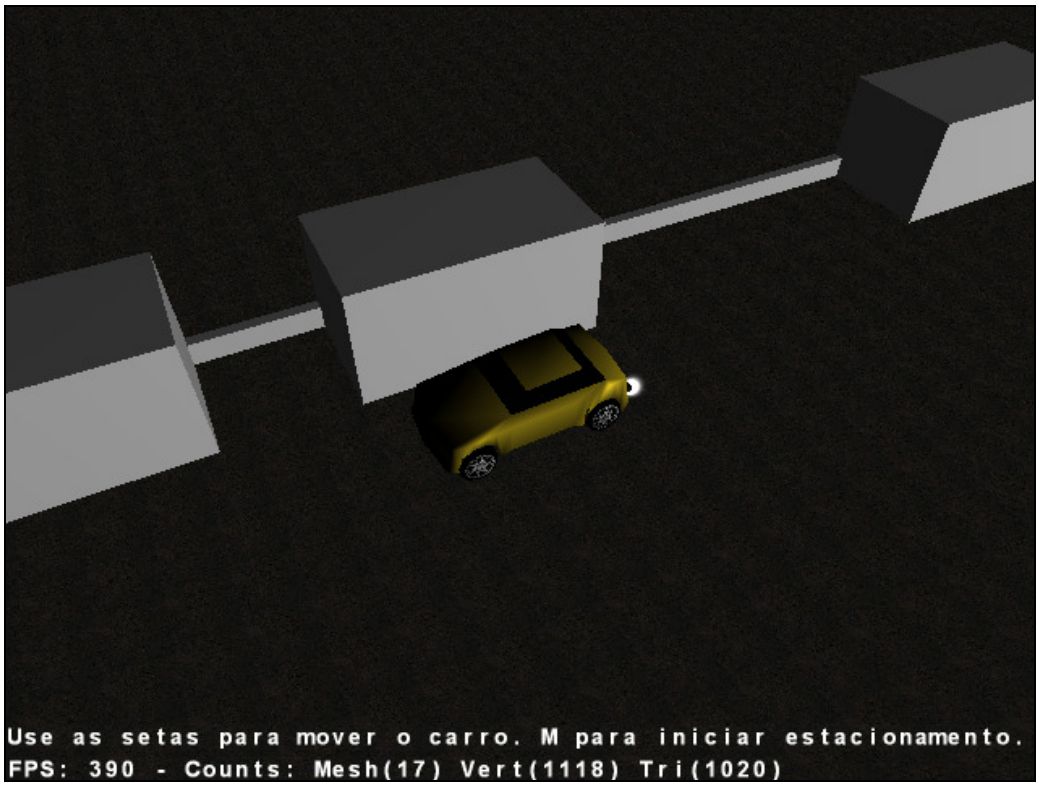


Figura 19 – Posição ideal para iniciar o estacionamento

Com o estacionamento autônomo iniciado, o veículo realiza seus movimentos de forma autônoma. A Figura 20 demonstra o veículo em durante um movimento.

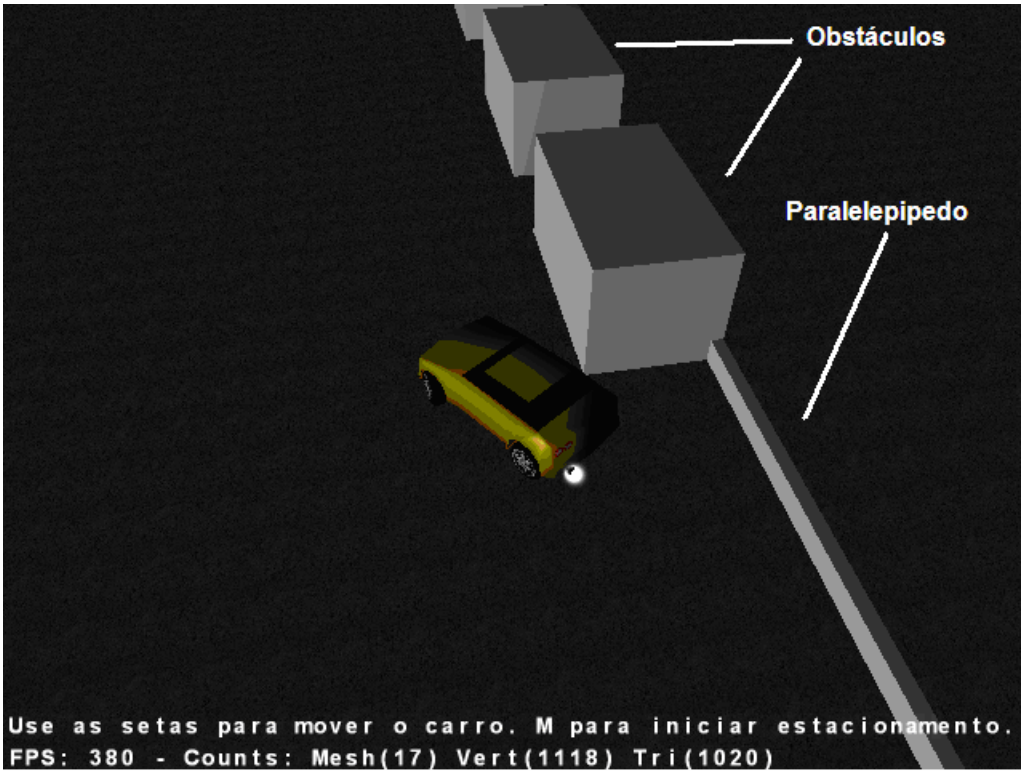


Figura 20 – Veículo em movimento

O processo de geração de movimentos só é encerrado quando o veículo está paralelo aos obstáculos e entre os mesmos. Ao encerrar o processo de estacionar, o veículo para entre os dois obstáculos. A Figura 21 ilustra um estacionamento encerrado.

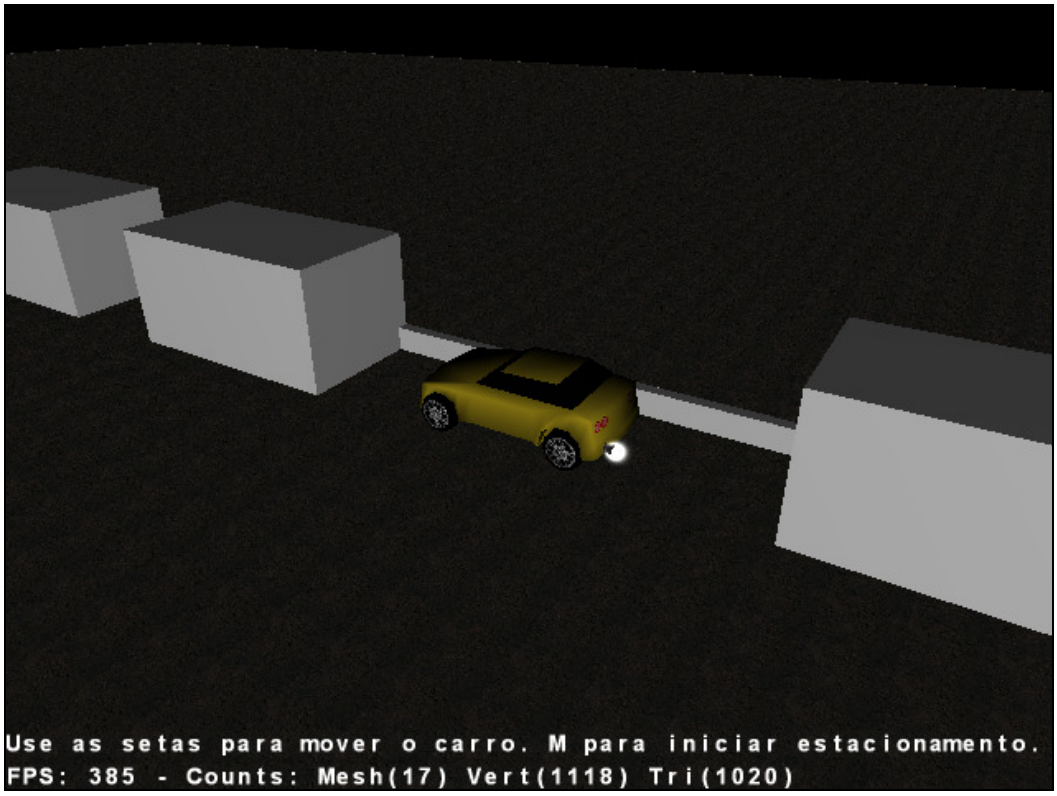


Figura 21 – Estacionamento completo

3.4 RESULTADOS E DISCUSSÃO

Os estudos realizados sobre lógica difusa demonstraram as vantagens da utilização desta técnica em relação a outros métodos de estacionamento autônomo. A imprecisão da informação foi tratada com êxito pela lógica difusa. Outras técnicas tornam o tratamento da informação muito particular, fazendo com que a aplicação se torne complexa e de difícil entendimento.

A *engine* gráfica JME proporcionou um alto nível de abstração para o desenvolvimento do cenário 3D. O principal fato que contribui para essa abstração é que a JME trabalha com o conceito de grafo de cena, onde os elementos do cenário puderam ser estruturados hierarquicamente de forma bastante clara de ser compreendida.

A maioria dos resultados foram notados (obtidos) durante a realização da fase de testes

do simulador. Essa etapa consistiu na execução do simulador em um computador com processador Pentium AMD Turion X2 de 2.1Ghz, 3Gb de memória RAM e placa gráfica ATI Radeon HD 3200 de 128Mb de memória dedicada. Vale mencionar que os testes iniciais foram feitos em um computador com processador Celeron 1.4Ghz e 1.5Gb de memória RAM, porém esta configuração não foi suportou o simulador atual, fazendo necessária a troca do computador. Como configuração mínima, recomenda-se um processador com 1Ghz, 1Gb de memória RAM e uma placa gráfica com memória dedicada.

Fazendo uma relação com os trabalhos correlatos elencados, pode-se perceber que o simulador desenvolvido possui diversas características derivadas deles. Com relação ao projeto de Tan e Seng (2007), apesar de não utilizar lógica difusa, os estudos sobre o movimento a ser realizado pelo veículo foram bastante proveitosos.

Com relação à lógica difusa, o trabalho desenvolvido por Lima (2005) serviu como base para estudos iniciais. A partir do projeto desenvolvido por Britton (2000), foi desenvolvida a base de regras difusas. Apesar de realizar o estacionamento em 90 graus, e não paralelo, serviu como uma base inicial de regras e estudo de variáveis de entrada e saída.

4 CONCLUSÕES

Os objetivos do trabalho foram atingidos, os quais eram desenvolver um simulador para estacionar autonomamente um veículo em uma vaga paralela utilizando lógica difusa e criar um ambiente 3D com um nível adequado de visualização.

Apesar disto o trabalho possui algumas limitações, as quais são:

a) o veículo protótipo apenas estaciona em vagas do lado direito;
b) serve apenas para vagas paralelas;
c) foram aplicadas poucas texturas ao terreno e também poucos tipos de obstáculos estáticos foram implantados, tal como construções e vegetação. Esses elementos ajudariam a aumentar o nível de realidade do cenário 3D.

Observou-se durante o trabalho que a lógica *fuzzy* possui características fundamentais para a solução de determinados tipos de problemas, principalmente aqueles relacionados com tomada de decisão sobre valores imprecisos.

Por último, fez-se uso também da *engine* gráfica JME e do *framework* JME Physics. Com esses recursos, pôde-se criar um cenário 3D com elementos bastante genéricos, caracterizando os veículos estacionados e a guia de meio-fio. Tanto a *engine*, como o *framework* de física apresentou uma boa documentação da API e também aplicativos de exemplos, ajudando principalmente na etapa inicial da especificação e desenvolvimento do simulador.

4.1 EXTENSÕES

Sugerem-se as seguintes extensões para a continuidade do trabalho:

a) criar novos modelos de terrenos para a simulação. Podem ser inseridos também mais elementos estáticos, como vegetação, construções e calçadas;
b) implementar um sistema de sonorização 3D para o cenário. Incluindo ruídos do ambiente, como o barulho do motor. Para prover isso, a *engine* JME contém uma integração com a biblioteca OpenAL, que além de ser multiplataforma, é bastante difundida na comunidade;
c) expandir a base de regras difusas para estacionar também em vagas do lado

esquerdo;

d) implementar um mecanismo para que o protótipo estacione também em vagas de 90 e 45 graus;
e) embarcar a solução desenvolvida em um protótipo real;
f) adaptar o simulador para veículos articulados, como caminhões.

REFERÊNCIAS BIBLIOGRÁFICAS

BARBOSA, Vilmar B. **Aplicação da lógica fuzzy no controle de um motor CC**. São José dos Campos: Instituto Tecnológico da Aeronáutica, 1995.

BILOBROVEC, Marcelo. **Sistema especialista em lógica fuzzy para controle, gerenciamento e manutenção da qualidade em processo de aeração de grãos**. 2005. 75 f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção, Universidade Tecnológica Federal do Paraná, Ponta Grossa.

BRITTON, Christopher. **The fuzzy truck simulator**. [S.l.], 2000. Disponível em: <http://www.gsi.dit.upm.es/~cif/cursos/lssii/pfuzzy/truck/>. Acesso em: 12 out. 2009.

CABRAL, Rodrigo B. **Adaptação computacional de sistemas de inferência difusos**: um caso aplicado. 1994. 81 f. Dissertação (Mestrado em Ciências da Computação) – Universidade Federal de Santa Catarina, Florianópolis.

CORNÉLIO FILHO, Plínio. **O modelo de simulação do GPCP-1:** jogo do planejamento e controle da produção. 1998. 76 f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção e Sistemas, Universidade Federal de Santa Catarina, Florianópolis. Disponível em: <http://www.eps.ufsc.br/disserta98/plinio/index.htm>. Acesso em: 13 maio. 2009

CORREA, Cláudio; SANDRI, Sandra. **Lógica nebulosa**. São José dos Campos, 1999. Disponível em: <http://www.ele.ita.br/cnrn/minicursos-5ern/log-neb.pdf>. Acesso em: 09 mar. 2009.

COX, Earl. **The fuzzy systems handbook**. New York: AP Professional, 1994.

CRUZ, Luiz F. **Sistematizações da teoria fuzzy**. 1996. 72 f. Dissertação (Mestrado em Matemática/Fundamentos) – Instituto de Geociências e Ciências Exatas, Universidade Estadual Paulista, Rio Claro.

DRIANKOV, Dimiter; HELLENDOORN, Hans; REINFRANK, Michael. **An introduction to fuzzy control**. [S.l.]: Springer-Verlag, 1996.

FERNANDES, A. P. S. **Sistema especialista difuso de apoio ao aprendizado do traumatismo Dento-Alveolar utilizando recursos multimídia**. 1997. 59 f. Dissertação (Mestrado em Ciências da Computação) – Programa de Pós-Graduação em Ciências da Computação, Universidade Federal de Santa Catarina, Florianópolis.

FRONZA, Germano. **Simulador de um ambiente virtual distribuído multiusuário para batalhas de tanques 3D com inteligência baseada em agentes em agentes BDI**. 2008. 141 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

HERNANDES, Fábio. **O problema de caminho mínimo com incertezas e imprecisões de tempo**. Rio de Janeiro, 2009. Disponível em:

<http://www.scielo.br/scielo.php?pid=S0101-74382009000200012&script=sci_arttext>. Acesso em: 12 out. 2009.

HAMPTON, Bill. **New Touran adds parking assist**. Bloomfield Hills, 2006. Disponível em: <http://www.autotechdaily.com/pdfs/T11-07-06.pdf>. Acesso em: 10 nov. 2009.

HELLA. [S.l.], 2009. Disponível em: <http://www.hella.com/hella-com-en/assets/media/hh08_gb_einzelseiten.pdf>. Acesso em: 07 nov. 2009.

HEINEN, Farlei J. **Robótica autônoma**: integração entre planificação e comportamento reativo. São Leopoldo, 1999. Disponível em: <http://ncg.unisinos.br/robotica/robotica.html>. Acesso em: 09 mar. 2009.

IVANQUI, Josmar. **Esteira eletrônica com velocidade controlada por lógica fuzzy**. 2005. 104 f. Dissertação (Mestrado em Ciências da Computação) - Programa de Pós Graduação em Informática na Educação, Centro Federal de Educação Tecnológica, Curitiba.

JUNG, Cláudio R. et al. **Computação embarcada**: projeto e implementação de veículos autônomos inteligentes. São Leopoldo, 2005. Disponível em: <http://bibliotecadigital.sbc.org.br/download.php?paper=138>. Acesso em: 02 out. 2009.

LIMA, Cabral et al. Análise de estabilidade e robustez de um sistema de controle fuzzy otimizado desenvolvido para guiar um robô simulado. In: SEMISH, 22.; 2005, São Leopoldo. **Anais eletrônicos...** São Leopoldo: UNISINOS, 2005. p. 1704-1715. Disponível em: <http://www.sbc.org.br/bibliotecadigital/download.php?paper=137>. Acesso em: 09 mar. 2009.

MATTOS, Mauro M. Problemas de segurança em sistemas operacionais: uma questão em aberto há quase 30 anos. In: SEMINCO, 1., 2004, Blumenau. **Anais eletrônicos...** Blumenau: FURB, 2004. p. 9-10. Disponível em: <http://www.inf.furb.br/seminco/2004/artigos/124-vf.pdf>. Acesso em: 09 out. 2009.

MATTOS; Merisandra C. **Sistema difuso de controle da assistência respiratória em neonatos – SARE**. 2001. 102 f. Dissertação (Mestrado em Ciências da Computação) – Programa de Pós-Graduação em Ciências da Computação, Universidade Federal de Santa Catarina, Florianópolis.

METACAFE. [S.l.], 2008. Disponível em: <www.metacafe.com/watch/1027169/auto_parking>. Acesso em: 08 nov. 2009.

OPENGL OVERVIEW. [S.l.], 2008. Disponível em: <http://www.opengl.org/about/overview/>. Acesso em: 26 maio 2009.

OSÓRIO, Fernando et al. Estacionamento de um veículo de forma autônoma utilizando redes neurais artificiais. In: ENRI, 3.; 2006, São Leopoldo. **Anais eletrônicos...** São Leopoldo: UNISINOS, 2006. p. 1-2. Disponível em: <http://www.natalnet.br/sbc2006/pdf/arq0171.pdf>. Acesso em: 09 mar. 2009.

PACHECO, Roberto C. S. **Tratamento de imprecisão em sistemas especialistas**. 1991. 85 f. Dissertação (Mestrado em Engenharia de Produção) – Engenharia de Produção e Sistemas, Universidade Federal de Santa Catarina, Florianópolis.

PATTON, Greeg. **Introduction to the JMonkeyEngine**. [S.l.], 2004. Disponível em: <http://fortworthjug.org:8090/fwjug/resources/topics/IntroTojME.pdf>. Acesso em: 26 out. 2009.

PEREIRA, Cledy Gonçalves. **Um sistema especialista com técnicas difusas para os limites da agência**. 1995. 91 f. Dissertação (Mestrado em Engenharia de Produção) – Engenharia de Produção e Sistemas, Universidade Federal de Santa Catarina, Florianópolis.

SOARES, Valéria D. **Uma genealogia da moral por Eu, Robô**. Porto Alegre, 2005. Disponível em: <http://revistaseletronicas.pucrs.br/ojs/index.php/famecos/article/viewFile/865/652>. Acesso em: 09 nov. 2009.

TAN, Eric; SENG, Ong Poh. **Building blocks**: where nurture meets nature. [Selangor], 2007. Disponível em: <http://www.buildingblocks.com.my/resource.html#BBJeep>. Acesso em: 09 nov. 2009.

TANAKA, Kokichi; MIZUMOTO, Masaharu. Fuzzy programs and their execution. In: _____. **Fuzzy sets and their applications to cognitive and decision processes**. New York: Academic Press, 1974. p. 41-76.

TANSCHEIT, Ricardo. **Sistemas fuzzy**. Rio de Janeiro, 2004. Disponível em: <http://www.inf.ufsc.br/~mauro/ine5377/leituras/ICA-Sistemas%20Fuzzy.pdf>. Acesso em: 09 nov. 2009.

TARIG, Ali A. E. S. **Controle de um braço robótico utilizando uma abordagem de agente inteligente**. 2001. 98 f. Dissertação (Mestrado em Ciências da Computação) – Programa de Pós-Graduação em Informática, Universidade Federal da Paraíba, João Pessoa.

TAVARES, Douglas M. **Ferramentas computacionais para robôs móveis autônomos**. Campinas, 2004. Disponível em: <http://libdigi.unicamp.br/document/?code=vtls000322079>. Acesso em: 09 out. 2009.

THE AUTO CHANNEL. **Hella Electronics help OEMs meet federal safety regulations**. Detroit, 2008. Disponível em: <http://theautochannel.com/news/2008/10/22/190277.html>. Acesso em: 08 nov. 2009.

ZADEH, Lotfi A. **Fuzzy sets and applications**. USA: John Wiley & Sons, 1987. 684 p.