UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORDENAÇÃO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

RAFAEL DOS PASSOS CANTERI

SOLUÇÃO PARA CONTROLE DE VEÍCULOS AUTÔNOMOS EM UM JOGO DE CARRO EM AMBIENTE URBANO

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2011

RAFAEL DOS PASSOS CANTERI

SOLUÇÃO PARA CONTROLE DE VEÍCULOS AUTÔNOMOS EM UM JOGO DE CARRO EM AMBIENTE URBANO

Trabalho de conclusão de curso de graduação, apresentado à disciplina Trabalho de Diplomação, do curso superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Coordenação de Análise e Desenvolvimento de Sistemas – COADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para a obtenção do título de Tecnólogo.

Orientador: Prof. Dr. André Koscianski

PONTA GROSSA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa

Diretoria de Graduação e Educação Profissional
Coordenação de Análise e Desenvolvimento de Sistemas
Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

SOLUÇÃO PARA CONTROLE DE VEÍCULOS AUTÔNOMOS EM UM JOGO DE CARRO EM AMBIENTE URBANO

por

RAFAEL DOS PASSOS CANTERI

Este Trabalho de Conclusão de Curso foi apresentado em 10 de novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

____
André Koscianski
Prof. Orientador

____
Gleifer Vaz Alves
Membro titular

____
Marcus Vinicius Drissen Silva
Membro titular

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

À minha Nica, que sempre acreditou no meu sonho de
trabalhar na maravilhosa área dos jogos eletrônicos.

AGRADECIMENTOS

Vou tentar, neste espaço, agradecer a todos que de alguma forma contribuíram para a realização deste trabalho e a conclusão deste curso de graduação.

Agradeço primeiramente aos meus pais, que me proporcionaram a oportunidade de estudar e de poder me dedicar aos estudos. Sou grato ainda ao meu irmão, que, sempre que precisei, tentou me ajudar em exercícios e dúvidas ao longo do curso.

Quero agradecer também ao meu orientador, o professor André Koscianski, que me deu a incrível oportunidade de trabalhar no desenvolvimento de um jogo e com quem aprendi muito. Sem ele esse projeto não teria sido possível.

Sou grato também aos meus grandes amigos que sempre estiveram comigo, mesmo estando mais distantes hoje em dia, devido à falta de tempo de todos. Em especial, ao Rodrigo que criou a imagem do teclado.

Por fim, agradeço à UTFPR, que proporcionou esse excelente curso, que, espero eu, será a alavanca da minha vida profissional e a todos aqueles que colaboraram direta ou indiretamente com esse trabalho.

What if you walk along and everything that you see is more than what you see— the person in the T-shirt and slacks is a warrior, the space that appears empty is a secret door to an alternate world? What if, on a crowded street, you look up and see something appear that should not, given what we know, be there? You either shake your head and dismiss it or you accept that there is much more to the world than we think. Perhaps it really is a doorway to another place. If you choose to go inside you might find many unexpected things. (SHEFF apud MIYAMOTO, 1994).

E se você caminhar e tudo o que você ver é mais do que aquilo que você vê - a pessoa de camiseta e calças é um guerreiro, o espaço que parece vazio é uma porta secreta para um mundo alternativo? E se, em uma rua cheia, aparecer algo que não deveria, dado ao que sabemos, estar ali? Ou você ignora o fato ou aceita que existem mais coisas no mundo do que você imagina. Talvez exista mesmo uma porta, e, se você escolher passar por ela, encontrará muitas coisas surpreendentes.
(SHEFF apud MIYAMOTO, 1994).

RESUMO

CANTERI, Rafael dos Passos. **Solução para Controle de Veículos Autônomos em um Jogo de Carro em Ambiente Urbano**. 2011. 75 f. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) – Coordenação de Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2011.

Jogos eletrônicos são *softwares* extremamente complexos e, cada vez mais, existem estudos de técnicas diferentes a serem aplicadas no seu desenvolvimento. Este trabalho trata do desenvolvimento de um jogo simulador de carros em ambiente urbano. Para o isso, foi necessária a utilização de um motor gráfico 3D, de estruturas de dados e bibliotecas específicas. O projeto objetiva simular uma cidade com carros, ruas, prédios e cruzamentos. Estão presentes no jogo, carros controlados pelo computador que trafegam nas ruas, de acordo com o sentido das vias. São apresentados métodos de criação de cidades virtuais tridimensionais, além de técnicas de controle de personagens autônomos para jogos. Foi desenvolvida uma estrutura de dados específica para mapeamento da cidade e orientação dos veículos do computador.

Palavras-chave: Simulador de carro, cidade virtual, jogo, Irrlicht, NPC.

ABSTRACT

CANTERI, Rafael dos Passos. **Solution for Autonomous Vehicles Control in a Car Game at Urban Environment**. 2011. 75 f. Final Paper (Technology in Systems' Analysis and Development) – Direction of Systems' Analysis and Development, Federal Technological University of Paraná. Ponta Grossa, 2011.

Video games are extremely complex software and constitute a field of increasing interest, with several studies of techniques to be applied in their implementation. This work deals with the development of a car simulator game in urban areas. The project required a 3D graphics engine, definition of data structures and selection of specific libraries. The project aims to simulate a town with cars, streets, buildings and crossroads. The game contains computer-controlled cars that travel on the streets, following the direction of the roads. Methods to create three-dimensional virtual cities are discussed, as well as control techniques applied to guide autonomous characters in games. A specific data structure was developed for the city mapping and orientation of the computer guided vehicles.

Key-words: Car simulator, virtual city, game, Irrlicht, NPC.

LISTA DE FIGURAS

Figura 1 - XBOX 360, Playstation 3 e Wii, Consoles da Geração Atual 19
Figura 2 - Gran Turismo 5 21
Figura 3 - Camadas de Arquitetura de um Jogo.. 22
Figura 4 - Exemplo de Máquina de Estados de um Jogo 27
Figura 5 - Cidade Virtual Procedural Gerada em Tempo Real 29
Figura 6 - Exemplos de Estruturas de Prédios Possíveis.................................... 31
Figura 7 - Exemplo de Grafo 34
Figura 8 - Diagrama de Classes do Jogo 38
Figura 9 - Jogo em Execução com a Primeira Cidade Desenvolvida.................. 40
Figura 10 - Height Field 41
Figura 11 - Exemplo de Divisão de Ruas da Cidade 44
Figura 12 - Arquivo CityConfig.xml que Contém as Informações dos Spots 46
Figura 13 - Arquivo que Contém Informações Sobre as Ruas 47
Figura 14 - Tabela de Navegação 48
Figura 15 - Modo de Visualização do Grafo 49
Figura 16 - Diagrama de Estados - Movimento do Carro Principal 51
Figura 17 - Câmera Aérea 53
Figura 18 - Diagrama da Máquina de Estados – Veículos Autônomos 55
Figura 19 - Estrutura Switch da Máquina de Estados 56
Figura 20 - Carros Autônomos Trafegando Pela Cidade 57
Figura 21 - Método de Detecção de Colisão do Carro Principal.......................... 60
Figura 22 - Menu Principal do Jogo.. 62
Figura A1 - Código de Criação de Terreno 71
Figura A2 - Código do Método de Criação de Prédios 72
Figura A3 - Exemplos de Pontos de Decisão dos NPCs 74

LISTA DE QUADROS

Quadro 1 - Divisão de Lotes - Centro e Bairro 30
Quadro 2 - Estruturas de Dados Usadas para Representar as Vias 45
Quadro 3 - O Mesmo Momento no Jogo, Visto com Duas Câmeras Diferentes .. 52

LISTA DE SIGLAS

API	Application Programming Interface
DLL	Dynamic-Link Library
FPS	First Person Shooter
IA	Inteligência Artificial
NPC	Non-Playable Character
PC	Personal Computer
PPGECT	Programa de Pós-Graduação em Ensino Ciência e Tecnologia
RPG	Role Playing Game
RTS	Real-Time Strategy
UML	Unified Modeling Language
XML	eXtensible Markup Language

LISTA DE ACRÔNIMOS

GUI	Graphical User Interface
MIT	Massachusetts Institute of Technology
RAM	Random Access Memory
SICITE	Seminário de Iniciação Científica e Tecnológica

SUMÁRIO

1 INTRODUÇÃO ... 15
1.1 PROBLEMA ... 15
1.2 OBJETIVOS ... 16
1.2.1 Objetivo Geral ... 16
1.2.2 Objetivos Específicos ... 17
1.3 ORGANIZAÇÃO DO TRABALHO ... 17
2 EMBASAMENTO TEÓRICO ... 18
2.1 VISÃO GERAL DE JOGOS ... 18
2.1.1 História dos Jogos Eletrônicos ... 18
2.1.2 Gêneros de Jogos ... 20
2.1.2.1 Jogos de carros ... 20
2.2 ARQUITETURA DE JOGOS ... 21
2.2.1 Análise de Jogos Eletrônicos ... 22
2.2.2 Engenharia de Software ... 23
2.2.3 Programação de Jogos ... 24
2.2.3.1 APIs e Bibliotecas ... 25
2.2.3.2 Engine Irrlicht ... 25
2.3 TÉCNICAS DE IA EM JOGOS ... 26
2.3.1 Máquina de Estados Finitos ... 26
2.3.2 Estratégias de Busca ... 27
2.3.3 Scripts ... 28
2.4 CIDADES VIRTUAIS ... 28
2.4.1 Cidades Pseudo-Infinitas ... 29
2.4.2 Citygen ... 30
2.4.3 Modelagem Contínua de Prédios em Nível de Detalhe ... 31
2.5 TECNOLOGIAS AUXILIARES ... 32
2.5.1 XML ... 32
2.6 ESTRUTURA DE DADOS ... 33
2.6.1 Grafo ... 33
3 DESENVOLVIMENTO ... 35
3.1 IMPLEMENTAÇÃO ... 35
3.1.1 Metodologia de Desenvolvimento ... 35
3.1.2 Paradigma de Desenvolvimento ... 36
3.2 LEVANTAMENTO DE CLASSES ... 37
3.3 REPRESENTAÇÃO DA CIDADE ... 39
3.3.1 Cidade Procedural ... 39
3.3.2 Cidade Modelada ... 40
3.3.3 Relevo ... 41
3.4 BIBLIOTECA DE SONS ... 42
3.5 MAPEAMENTO DA CIDADE ... 43
3.5.1 Tabela de Navegação ... 46
3.5.1.1 Arquivo de dados ... 47
3.5.1.2 Classe CTable ... 48
3.5.1.3 Classe CLaserBeam ... 49
3.6 MOVIMENTAÇÃO DO CARRO ... 50

3.6.1 Visão Geral de Movimento 50
3.6.2 Controles 51
3.6.3 Câmera.. 52
3.6.4 Movimento dos Modelos 3D 53
3.7 CONTROLE DE NPCS 54
3.7.1 Orientação dos Carros do Computador 54
3.8 CINEMÁTICA DOS MOVIMENTOS 57
3.9 COLISÃO 58
3.9.1 Colisão entre NPCs 59
3.9.2 Carro Principal e Prédios 59
3.10 INTERFACE DE USUÁRIO... 61
4 CONCLUSÃO 63
5 CONSIDERAÇÕES FINAIS 64
5.1 TRABALHOS FUTUROS 64
REFERÊNCIAS... 66
APÊNDICE A 70
APÊNDICE B 73

1 INTRODUÇÃO

Jogos digitais se tornaram uma das áreas de entretenimento mais importantes atualmente. Os jogos são uma atividade humana que assumiu, ao longo da história, diversos significados e definições antropológicas, culturais, filosóficas, psicológicas, educacionais, dentre outras (LARANJEIRA; PORTO; ALVES, 2001). O faturamento anual da indústria mundial de jogos eletrônicos já superou o de todas as outras áreas do entretenimento, inclusive o da indústria cinematográfica.

Os jogos vêm conquistando públicos de todas as idades, criando, dessa maneira, novos problemas para os profissionais da área. Contudo, a criação e produção de jogos eletrônicos acontecem, quase que totalmente, nos Estados Unidos e no Japão. Como prova disso, tem-se o fato de que uma das empresas com maior valor de mercado de todo o Japão é a Nintendo, uma empresa voltada totalmente à produção de jogos e consoles. Com investimento, planejamento e muito estudo por parte dos profissionais da área essa situação pode ser mudada e países em desenvolvimento podem se tornar grandes produtores dessa área no futuro.

O desenvolvimento de jogos eletrônicos envolve vários profissionais, mas uma parte significante cabe ao pessoal da área de Computação. Dentre as grandes preocupações para esses profissionais está a escolha de uma linguagem de programação adequada, associada a bibliotecas que satisfaçam a necessidade do *software* em questão.

1.1 PROBLEMA

Desenvolver jogos é trabalhoso e envolve várias tarefas. Em jogos ambientados em cidades, um dos principais problemas que se tem a resolver é qual metodologia utilizar para criar o ambiente do jogo. Outra questão importante

a se levar em consideração no desenvolvimento de qualquer jogo é como controlar os personagens autônomos presentes.

Este trabalho trata da implementação de um jogo simulador de carro. O problema a ser resolvido é a criação e o controle de veículos autônomos que trafegam por um ambiente tridimensional que representa uma cidade. A criação do cenário e a preparação do *software* para futuro refinamento de modelos físicos são outros pontos importantes do trabalho. São abordadas também maneiras de tratar sons e músicas em jogos, como testar colisões entre objetos e como criar uma interface gráfica para menus.

1.2 OBJETIVOS

O projeto pretende utilizar uma linguagem de programação associada a bibliotecas específicas de desenvolvimento de jogos tridimensionais para construir um jogo de carros.

Os objetivos do projeto se desdobram em uma série de objetivos específicos. Quando atingidos os objetivos específicos, satisfaz-se também o objetivo geral.

1.2.1 Objetivo Geral

Esse trabalho vem de um projeto de iniciação científica, realizado no Grupo de Pesquisas Edutainment, pertencente ao PPGECT, Programa de Pós-Graduação em Ensino Ciência e Tecnologia. O projeto consiste no desenvolvimento de um jogo simulador de veículo, que contém carros controlados pelo computador, em um ambiente que representa uma cidade.

1.2.2 Objetivos Específicos

- Adquirir conhecimento sobre desenvolvimento de jogos eletrônicos.
- Conhecer as principais técnicas usadas no desenvolvimento de jogos, através do estudo de materiais específicos.
- Estudar as diferentes possibilidades e tecnologias para se criar uma cidade virtual.
- Desenvolver uma forma para representação de uma cidade em três dimensões.
- Desenvolver uma solução para controle de veículos, pilotados pelo computador, presentes no jogo.

1.3 ORGANIZAÇÃO DO TRABALHO

No capítulo 2 são apresentados conceitos referentes aos jogos eletrônicos. São abordados os gêneros e uma breve história da indústria de jogos, técnicas de inteligência artificial para controle de personagens, além de arquitetura de jogos. Por fim, são mostrados e explicados alguns métodos de criação de cidades 3D, tecnologias e estruturas de dados usadas.

No capítulo 3 é descrito o desenvolvimento do projeto. Nesse capítulo estão expostas as etapas mais relevantes, desde o levantamento de requisitos do jogo, passando pela criação da cidade, implementação da solução de controle de veículos autônomos, implementação do carro principal do jogador e solução para detecção de colisões.

O capítulo 4 trata das conclusões e dos resultados obtidos com este trabalho.

Por fim, o capítulo 5 discorre sobre as considerações finais e os trabalhos futuros que podem dar continuidade a este projeto.

2 EMBASAMENTO TEÓRICO

Este capítulo trata de conceitos importantes sobre a indústria, a análise e desenvolvimento de jogos eletrônicos. São discutidos também métodos de criação de cidades virtuais.

2.1 VISÃO GERAL DE JOGOS

Antes de começar a desenvolver jogos é importante conhecer a indústria para saber o que fazer e o que evitar. É útil também entender que existem vários estilos diferentes e, de uma forma ou de outra, qualquer jogo acaba sendo classificado em alguma categoria.

2.1.1 História dos Jogos Eletrônicos

Antes da década de 70, começaram a surgir os primeiros indícios de criação de jogos eletrônicos. Uma das principais empresas da indústria, a Nintendo, começou como uma empresa que fabricava cartas de Hanafuda (um tipo de baralho japonês) em 1889 (SHEFF, 1994) e no início da década de 80 adentrou no mercado de produção de jogos e consoles, onde se encontra até hoje.

O primeiro jogo eletrônico interativo criado na história foi o Spacewar, um jogo onde duas pessoas controlavam dois tipos diferentes de espaçonave que deveriam competir entre si. O Spacewar foi programado por um estudante do MIT (Instituto de Tecnologia de Massachusetts), Steve Russell, em um computador PDP-1, no ano de 1961.

Em 1970, Nolan Bushnell começou a trabalhar em uma versão fliperama do jogo Spacewar, chamada Computer Space. No ano seguinte, a empresa

Nutting Associates comprou o jogo de Bushnell, e assim colocou no mercado a primeira máquina de fliperama da história. Em 1978, a empresa fundada por Bushnell e comprada pela Warner, a Atari, lança o console Atari 2600, que se tornou um sucesso absoluto (COHEN, 1987).

As máquinas de fliperama estavam em seu auge nos anos 80, ao mesmo tempo em que surgiram os primeiros consoles 8-bit após o Atari 2600: Famicom, da Nintendo e Master System, da SEGA. Na área de jogos para PC, houve também grandes lançamentos de jogos e criação de empresas. A década de 90 foi marcada pelo lançamento de consoles de 16-bit e, anos depois, pelos consoles da geração 32/64-bit. Entre 2000 e 2001, Sony, Nintendo e, a novata no mercado, Microsoft divulgaram seus novos consoles de vídeo game de 128-bit, cuja geração durou até o final do ano de 2006.

Atualmente, figuram no mercado, os consoles de alta definição da Sony e da Microsoft, além do console inovador com sensor de movimentos, o Nintendo Wii. Ao lado destes, estão presentes os portáteis da Sony e da Nintendo juntamente com os jogos para computador, que continuam a acompanhar a evolução dos jogos de consoles caseiros, graças às novas tecnologias de placas de vídeo e processadores. Além disso, estão crescendo muito na indústria, jogos para celulares ou *smartphones* e para redes sociais.

Figura 1 - XBOX 360, Playstation 3 e Wii, Consoles da Geração Atual
Fonte: Top News (2008)

2.1.2 Gêneros de Jogos

Assim como na indústria cinematográfica, existem diversos gêneros na indústria de jogos atualmente. Esses gêneros servem tanto para guiar o consumidor na hora da compra, quanto para orientar a equipe em uma direção de desenvolvimento.

Entre os gêneros de jogos pode-se citar: RTS (Jogos de Estratégia em Tempo Real), aventura, RPG (Jogo de Interpretação de Papéis), ação e simulação (BETHKE, 2003). Existem também outros gêneros como jogos de tiro em primeira e terceira pessoa, esporte, quebra-cabeça ou *puzzle*, coletâneas de *mini-games*, etc.

2.1.2.1 Jogos de carros

Desde a época do Atari 2600 existem jogos de carro à disposição nas mais diversas plataformas. Ao longo da história, eles sofreram diversas evoluções, tanto gráficas quanto de jogabilidade. Atualmente jogos de carro são divididos entre jogos de corrida *arcade* e jogos simuladores de carro, a diferença entre as duas classificações é que os de corrida *arcade* não se preocupam com aspectos avançados de Física e fidelidade à realidade como os simuladores.

Simuladores de carros se preocupam rigidamente com a representação da Física e do comportamento dos veículos. Um dos simuladores mais aclamados na indústria é o Gran Turismo (GRAN TURISMO, 2011), nele aspectos de simulação são bastante fiéis à realidade e até mesmo o som dos veículos é obtido a partir da versão real dos carros.

Figura 2 - Gran Turismo 5
Fonte: Gran-Turismo.com (2011)

A figura 2 contém uma imagem do jogo Gran Turismo 5, lançado para Playstation 3 em 2010, a foto foi tirada da câmera interna do carro.

2.2 ARQUITETURA DE JOGOS

A arquitetura de um jogo é geralmente dividida em duas partes: abstração de jogo e abstração de interface de *hardware* (PERUCIA et al. 2007).

A responsabilidade da abstração de *hardware* é tratar do uso de recursos e dispositivos como gráficos, entradas (teclado, mouse, *joysticks*), som, e tudo o que é relacionado com acesso ao *hardware* da plataforma de desenvolvimento. Já a abstração de jogo é utilizada para se ter uma visão global de como o *software* funcionará, com seus subsistemas, sem se preocupar com o *hardware*.

Figura 3 - Camadas de Arquitetura de um Jogo
Fonte: Adaptada de Perucia et al. (2007)

A figura 3 exibe as camadas de arquitetura de um jogo e as relações entre elas. A camada de abstração de *hardware*, vista na figura 3, é representada principalmente por bibliotecas específicas para construção de jogos.

2.2.1 Análise de Jogos Eletrônicos

A análise de um jogo eletrônico é equivalente à fase de análise do desenvolvimento de um sistema de *software* qualquer, quando são definidas as características e requisitos do produto (PERUCIA et al., 2007). Existem diferenças entre a análise de um sistema tradicional e a modelagem de um jogo, entre elas está a dificuldade em coletar os requisitos do sistema neste último, principalmente no início do desenvolvimento, devido à falta de algo concreto e às constantes mudanças e adaptações causadas pelo avanço tecnológico ou pelo lançamento de outros jogos concorrentes.

Jogos digitais são *softwares* comerciais, não um sistema que um cliente deseja para informatizar a sua empresa, por exemplo. Assim fica difícil definir exatamente o que fazer e o que irá agradar o público-alvo, que nesse caso são os jogadores. Dessa forma, processos rígidos das fases tradicionais de análise de sistemas podem ser substituídos por técnicas e diretivas maleáveis. Entretanto, existem muitos jogos que são desenvolvidos utilizando os paradigmas de

desenvolvimento mais comuns, como o modelo Cascata. O ciclo de vida clássico ou modelo Cascata é o paradigma mais antigo e mais amplamente utilizado da Engenharia de Software (PRESSMAN, 2006). Esse modelo possui as fases: Levantamento de Requisitos, Projeto, Implementação, Verificação e Manutenção.

Em geral, o *design* é testado e os requisitos são coletados durante todo o processo de desenvolvimento do jogo, permitindo que mudanças sejam feitas durante o tempo em que se está trabalhando sobre o mesmo. Porém, mudanças profundas ou complexas devem ser detectadas e executadas nos estágios iniciais da concepção. O estágio de análise de requisitos é o mais crítico para o sucesso do projeto e, em muitos aspectos, é o mais difícil (BETHKE, 2003).

Assim como em sistemas tradicionais, caso os requisitos não sejam analisados corretamente, o desenvolvimento pode ser prejudicado por atrasos, pois será necessário retrabalho elevado.

2.2.2 Engenharia de Software

Engenharia de Software é a criação e a utilização de sólidos princípios de engenharia a fim de obter *softwares* econômicos que sejam confiáveis e que trabalhem eficientemente em máquinas reais (PRESSMAN, 2006).

Em jogos eletrônicos a Engenharia de Software também está presente, pois é imprescindível que o jogo seja um *software* de qualidade, sem custos desnecessários e que seja desenvolvido de acordo com uma metodologia organizada. Testes de *software* são essenciais nessa área, pois além de testar falhas, deve-se testar também se o jogo não ficou desbalanceado ou maçante para o jogador.

Algumas formas de testes de jogos conhecidas são os *beta tests,* que consistem em disponibilizar o jogo, ainda incompleto, aos jogadores, para que estes reportem os defeitos do produto para a empresa desenvolvedora. Outra forma de teste, ainda mais utilizada que a anterior, é ter na empresa, funcionários específicos para teste de jogo.

Os principais objetivos da Engenharia de Software são melhorar a qualidade do *software* e aumentar a produtividade e a satisfação profissional de todos os envolvidos no processo de desenvolvimento.

2.2.3 Programação de Jogos

O processo de codificação de um jogo geralmente requer amplos conhecimentos e capacidade técnica por parte do programador. Algumas características específicas exigidas de um programador de jogos eletrônicos são: capacidade de se adaptar a novas tecnologias, trabalhar em projetos com requisitos não detalhados, aperfeiçoar rotinas para melhorar desempenho e relacionar-se bem com outros profissionais envolvidos no projeto, como engenheiros de som, artistas, *game designers* e roteiristas.

A programação apresenta vastos desafios, e a utilização de bibliotecas complexas, como as bibliotecas gráficas e de física, sempre se mostra necessária.

Programadores de jogos precisam de dois elementos centrais. Eles necessitam de um profundo entendimento de como enfrentar os desafios 3D de programação (código, matemática, e física), e como injetar vida através de código aos elementos do jogo (MEIGS, 2003).

Uma das maiores preocupações de todo desenvolvedor de jogos é a questão do desempenho. Jogos são aplicativos que, em geral, exigem muito dos computadores, dessa forma, a programação precisa ser bastante otimizada visando o aproveitamento do *hardware* e evitando desperdícios de processamento em procedimentos desnecessários.

2.2.3.1 APIs e Bibliotecas

Há várias APIs (Interfaces de Programação de Aplicações), conjunto de rotinas, protocolos e ferramentas para construção de aplicações, usadas no desenvolvimento de jogos. As mais conhecidas e utilizadas para esse propósito em computadores são DirectX (WINDOWS DIRECTX, 2011) e OpenGL (OPENGL, 2011), a maioria dos *engines* (motores) de jogos são criados em cima dessas APIs. Consoles possuem APIs próprias, mas sua programação é semelhante às APIs de computadores para facilitar a portabilidade.

Dentre as ferramentas para desenvolvimento de jogos estão as bibliotecas, APIs e *game engines* (BINA, 2010). Nas bibliotecas geralmente existem ferramentas específicas para gráficos, sons, entradas, IA, física, etc. Algumas bibliotecas são multiplataforma e com o mesmo código é possível criar jogos para diferentes sistemas operacionais e consoles.

2.2.3.2 Engine Irrlicht

O *engine* Irrlicht é um motor 3D de código aberto, escrito e utilizável em C++ (IRRLICHT ENGINE, 2011), também utilizável em linguagens .NET. É uma ferramenta multiplataforma e utiliza Direct3D, OpenGL e renderizador de *software* próprio.

A maior parte dos recursos gráficos de motores comerciais está presente no Irrlicht, que é completamente gratuito, esse foi um dos motivos mais significativos na escolha desse motor para o projeto. Além disso, não é difícil encontrar melhorias e adições de recursos para o motor na *web*.

2.3 TÉCNICAS DE IA EM JOGOS

Inteligência Artificial é o estudo de como fazer os computadores realizarem coisas que, até então, as pessoas fazem melhor (RICH; KNIGHT, 1994).

É um ramo da ciência da computação que se propõe a elaborar dispositivos que simulem a capacidade humana de raciocinar, perceber, tomar decisões e resolver problemas.

Em jogos são utilizadas diversas técnicas de IA, porém elas se diferenciam um pouco da chamada Inteligência Artificial Acadêmica. A Inteligência Artificial Acadêmica é a IA pura, baseada nas regras de lógica, enquanto a Game AI *(*Inteligência Artificial de Jogos*)* se baseia nas técnicas aplicando-as de forma, em geral, mais simplificada. O objetivo da IA em jogos não é saber como pensar, mas sim desencadear uma sequência de ações aos personagens, que deem a sensação de inteligência ao jogador, independente de como eles chegam a esses resultados. Entretanto, atualmente existem jogos no mercado que possuem técnicas de IA ainda mais complexas que a IA acadêmica. Um exemplo são os jogos eletrônicos de futebol, nos quais o computador aprende com as ações do jogador e se adapta às situações, melhorando constantemente o seu comportamento.

Desde que os jogos eletrônicos existem foi implementado algum tipo de inteligência artificial. Os métodos de IA mais utilizados para esse propósito são as máquinas de estados finitos, os scripts e as estratégias de busca tradicionais (TATAI, 2003).

2.3.1 Máquina de Estados Finitos

Uma máquina de estados finitos é composta de um número finito de estados que podem ser alterados através de algumas regras de transição. Essas

regras são ativadas dependendo de alguma situação encontrada no jogo e do estado atual do objeto.

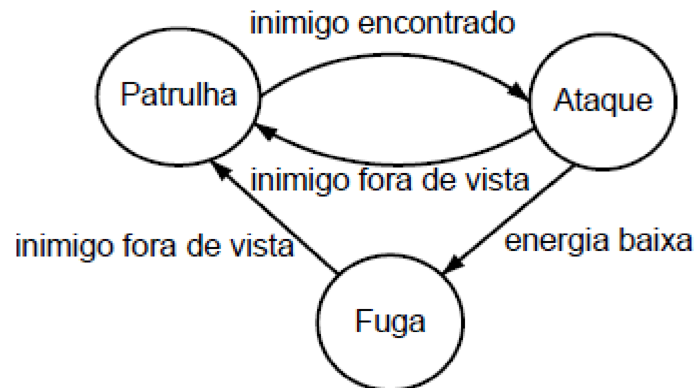


Figura 4 - Exemplo de Máquina de Estados de um Jogo
Fonte: Técnicas de Sistemas Inteligentes Aplicadas ao Desenv. de Jogos (2003)

A figura 4 traz um exemplo simples de máquina de estado para jogos. Nela, existem três estados: patrulha, ataque e fuga; e quatro eventos que causam a transição entre estes estados.

2.3.2 Estratégias de Busca

As estratégias de busca são normalmente utilizadas em uma árvore onde a raiz é seu estado inicial e os demais nós são possíveis estados finais, a busca deseja encontrar um caminho qualquer do estado inicial até um determinado estado final. Este método é muito utilizado em jogos clássicos como xadrez e damas, mas também é utilizado para busca por caminhos em jogos modernos.

Outro método de busca bastante empregado é de otimização de caminho. Ele serve, por exemplo, para guiar NPCs dentro de um labirinto. Exemplos desse método são o algoritmo de Dijkstra e o algoritmo A* (POZZO; KOSCIANSKI, 2010).

Um NPC ou Non-Playable Character é um objeto que pode ter comportamentos próprios dentro do jogo, podendo ser comparado com a ideia de agente inteligente (POZZO; KOSCIANSKI, 2010).

2.3.3 Scripts

Um *script* é uma estrutura que especifica uma sequência de eventos. É similar a uma sequência de pensamentos ou situações que devem acontecer (PRATSCHKE; PELIZZONI; ALUÍSIO, 2011).

Scripts têm amplo aspecto de aplicação, pois eventos da vida real tendem a acontecer segundo certos padrões ou de maneiras bem conhecidas; eventos de interesse geralmente guardam relações causais entre si; geralmente existem condições de entrada (pré-requisitos) para que um evento aconteça.

2.4 CIDADES VIRTUAIS

Diversos trabalhos propõem a criação automática de cidades, por exemplo, usando algoritmos, ideias como fractais ou mesmo fotos aéreas de cidades reais. Em (KELLY; MCCABE, 2006) são avaliadas diversas técnicas de geração procedural de cidades e as vantagens das diversas abordagens.

Técnicas de geração procedural de cenários têm sido usadas por mais de 20 anos na área da computação gráfica (EBERT et al., 2003). Cidades virtuais são utilizadas em jogos digitais, filmes de animação 3D, simulações em projetos de governos, entre outras aplicações.

Existem artigos que tratam de técnicas de geração de mapas em jogos, um deles demonstra a criação de mapas para Tile/Map-Based Games (MICHAEL, 1999), no entanto, a sua utilização é para os gêneros RTS e RPGs, onde a câmera fica diretamente em cima da ação ou em perspectiva isométrica, o que não é o caso de um jogo de carros. Ainda assim, o artigo é útil para esse projeto, pois revelou a necessidade de uma grande quantidade de memória RAM para a criação e utilização de um terreno gerado por código.

2.4.1 Cidades Pseudo-Infinitas

Uma possibilidade que a geração procedural permite é a criação de cidades que dão a sensação de serem infinitas. Nessa abordagem as cidades contém prédios com geometrias variadas que são geradas de acordo com a necessidade, isso é chamado de cidades pseudo-infinitas (GREUTER et al., 2003).

Basicamente, nessa abordagem as plantas para cada edifício são criadas pela combinação de polígonos gerados aleatoriamente em um processo iterativo, isso evita que os edifícios sejam sempre iguais. Existe uma entidade que gerencia os recursos do sistema para que o algoritmo não acabe com o desempenho. O gerenciamento é realizado através de uma lista de exibição de edifícios, só é preciso exibir o que o jogador pode ver. Os prédios que não estão mais visíveis no momento ficam armazenados em outra lista.

A principal vantagem dessa abordagem é o fato de se ter um cenário “ilimitado”. Muitos jogos utilizam a famosa “parede invisível” para limitar o ambiente do jogo, isso acaba com o realismo do jogo e, muitas vezes, irrita os jogadores. Com a técnica de cidades pseudo-infinitas é possível aumentar o realismo do jogo e agradar mais os jogadores.

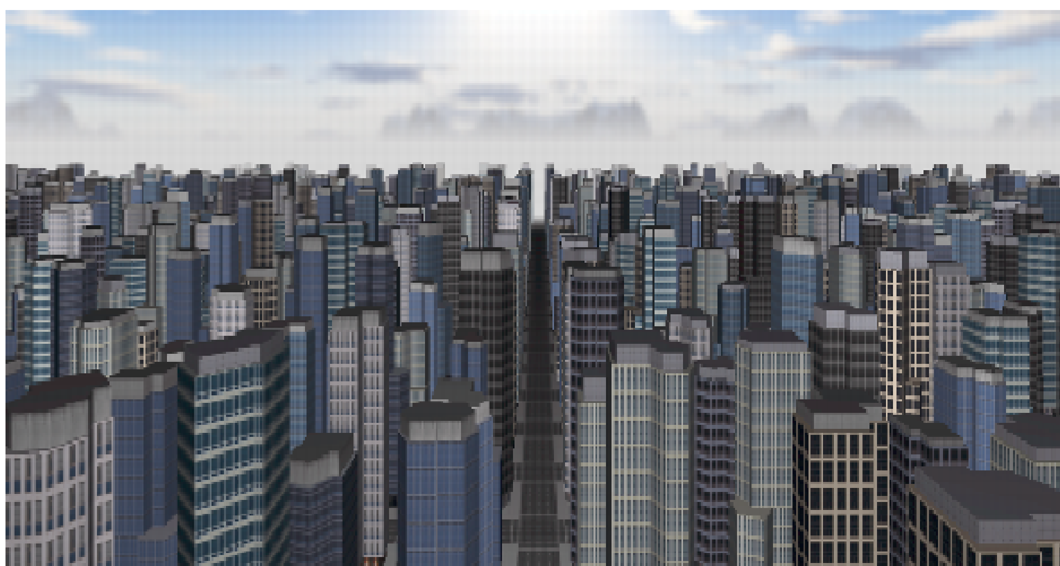


Figura 5 - Cidade Virtual Procedural Gerada em Tempo Real
Fonte: Real-time Procedural Generation of ‘Pseudo Infinite’ Cities (2003)

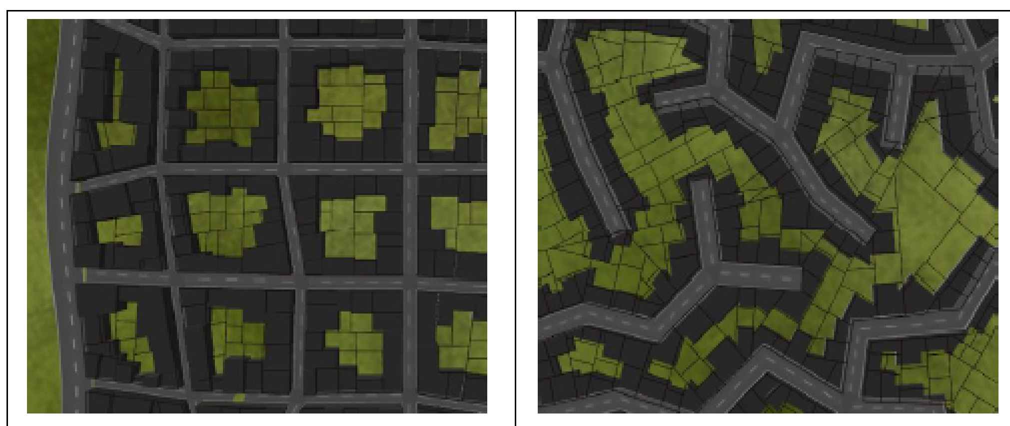
A figura 5 apresenta um exemplo de uma grande cidade criada com a técnica de geração de cidades pseudo-infinitas.

2.4.2 Citygen

Citygen é uma aplicação interativa que provê um espaço integrado para geração de cidade (KELLY; MCCABE, 2007). Ele divide o processo em três estágios: Geração de Estrada Primária, Geração de Estrada Secundária e Geração de Prédios.

Para se criar as estradas primárias, usa-se um algoritmo para computar as trajetórias das estradas, que seguem o terreno de forma natural e convincente. As estradas secundárias são geradas através de um algoritmo de subdivisão. Por fim, os prédios são construídos nos lotes criados pelo algoritmo de divisão de lotes da cidade. Assim, os prédios são gerados nas áreas delimitadas pelas estradas secundárias.

O quadro 1 mostra duas possibilidades de divisão de lotes do algoritmo. Na esquerda a região central de uma cidade, e na direita uma região de periferia.



Quadro 1 - Divisão de Lotes - Centro e Bairro
Fonte: Citygen: An Interactive System for Procedural City Generation (2007)

2.4.3 Modelagem Contínua de Prédios em Nível de Detalhe

A abordagem de nível de qualidade contínua unifica a representação de conjuntos heterogêneos de edifícios, que ocorrem na maioria dos modelos virtuais de cidades em 3D (DÖLLNER; BUCHHOLZ, 2005).

O objetivo desse método é basicamente construir edifícios procedurais bastante detalhados e evitar que existam edifícios com a mesma aparência na cidade virtual. Nessa ideia o edifício é construído a partir do piso da base (andar térreo), então são construídos os outros pisos, as paredes, partes específicas das paredes e, por último, o telhado. Só então são colocadas as texturas, diferenciadas para cada elemento.

É uma abordagem bastante interessante para jogos que têm uma grande necessidade de diferenciação na cidade, pois os edifícios são distinguidos não apenas por tamanho ou texturas, mas também por toda a estrutura que os compõem. A figura 6 mostra alguns dos muitos tipos de estrutura que podem ser utilizadas nessa técnica de modelagem.

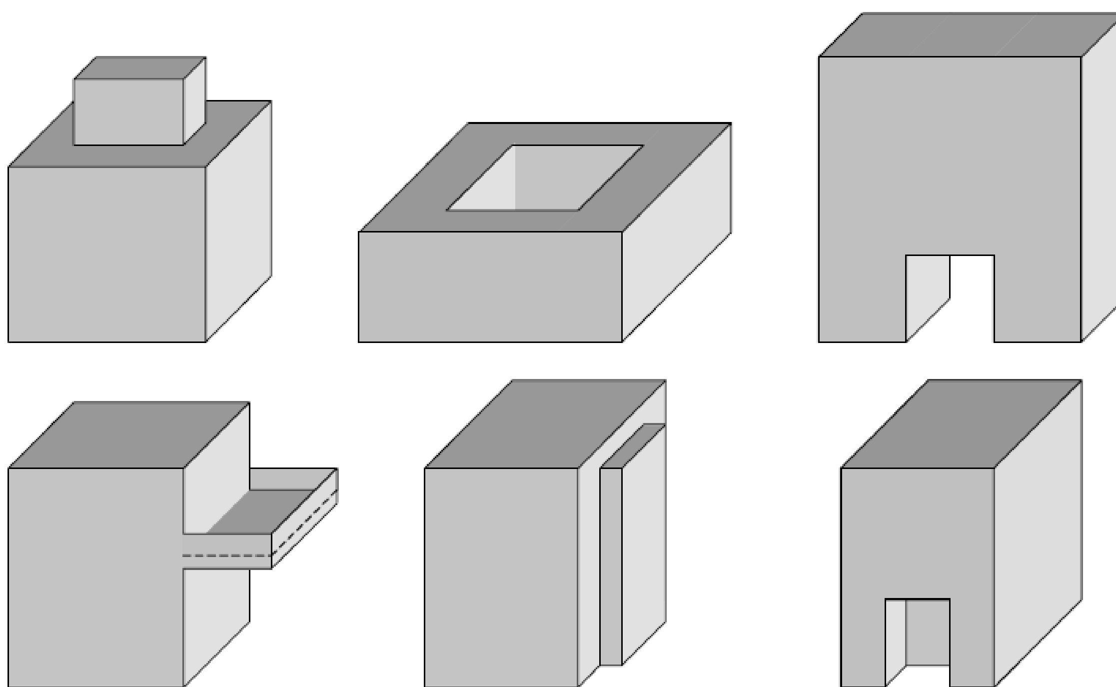


Figura 6 - Exemplos de Estruturas de Prédios Possíveis
Fonte: Continuous Level-of-Detail Modeling of Buildings in 3D City Models (2005)

2.5 TECNOLOGIAS AUXILIARES

Jogos são *softwares* complexos que utilizam as mais diversas ferramentas para as mais distintas utilidades. Como existe muita diversidade na área e bastante necessidade de inovação, diferentes tecnologias e estruturas de dados são usadas em jogos para várias finalidades.

Existem jogos que possuem vários módulos e cada módulo é desenvolvido em uma linguagem de programação diferente, isso pode ocorrer, por exemplo, em jogos *multiplayer*, onde as questões relacionadas à comunicação podem ser desenvolvidas em uma linguagem diferente à do núcleo do jogo. Por esse motivo, é preciso conhecer várias tecnologias e linguagens de programação diferentes.

Nesta seção serão abordada tecnologias relevantes para a implementação do projeto proposto.

2.5.1 XML

Extensible Markup Language (XML) ou Linguagem de Marcação Extensível se tornou um padrão muito utilizado na representação e troca de dados em aplicações (BRAY; PAOLI; SPERBERG-MCQUEEN, 1998). Resumidamente, o XML é um formato para a criação de documentos com dados organizados de forma hierárquica.

Mas como isso se aplica a jogos eletrônicos? É possível existirem arquivos de configuração externos, que não fazem parte do código fonte, dessa forma se torna uma tarefa mais fácil reconfigurar e efetuar mudanças no jogo sem precisar mudar o código ou trabalhar com a linguagem de programação propriamente dita (KOSCIANSKI, 2011). Assim qualquer pessoa com conhecimentos em XML pode customizar o jogo a seu gosto (claro que com limitações) sem precisar ter conhecimentos em linguagem de programação.

2.6 ESTRUTURA DE DADOS

Entender como funciona o computador, como armazenar dados, e como trabalhar de forma eficiente com os dados são questões essenciais para a programação de jogos (PENTON, 2003). Estes utilizam desde estruturas de dados mais simples como *arrays*, até as mais complexas como grafos e árvores.

2.6.1 Grafo

Um grafo consiste em um conjunto de nós ou vértices e em um conjunto de arcos ou arestas (TENEMBAUM; LANGSAM; AUGENSTEIN, 1995). Em um grafo, cada nó está ligado a, pelo menos, mais um nó, por meio de uma aresta.

Grafos possuem diversas aplicações práticas como cálculo de caminho de custo mínimo entre dois pontos, desenho de redes de transporte, de telecomunicações e elétricas, entre outras.

Essa estrutura de dados é também utilizada em jogos eletrônicos para o mapeamento do ambiente e movimento de personagens controlados pelo computador. Em jogos de estratégia em tempo-real, por exemplo, quando o jogador dá ordens às suas unidades, o caminho que elas irão percorrer e como irão contornar possíveis obstáculos, como montanhas, é definido por um grafo.

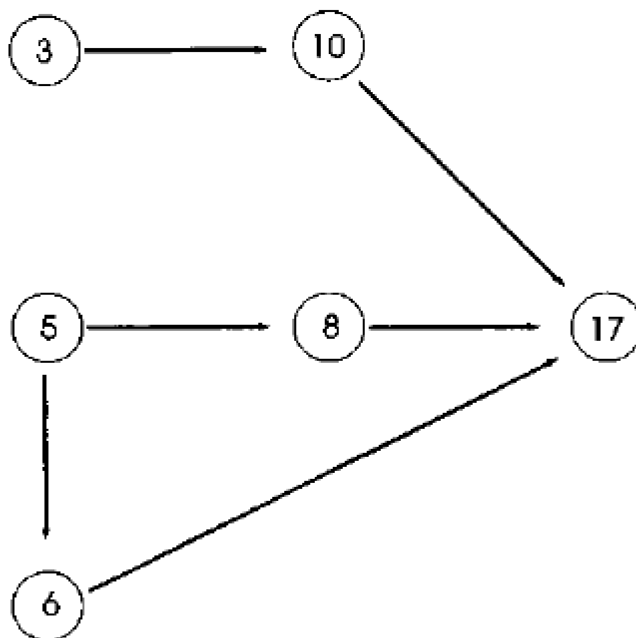


Figura 7 - Exemplo de Grafo
Fonte: Estrutura de Dados Usando C (1995)

No caso deste projeto, a estrutura grafo é útil para mapeamento do ambiente de jogo, divisão de ruas e identificação de cruzamentos.

3 DESENVOLVIMENTO

O presente capítulo trata do desenvolvimento do projeto proposto. São descritas as etapas do desenvolvimento, as ferramentas utilizadas e os processos que envolveram a construção efetiva do jogo.

3.1 IMPLEMENTAÇÃO

Inicialmente, o foco do projeto era desenvolver um simulador de carros, com equações de dinâmica e efeitos que deixassem o veículo mais real. Contudo, o responsável pela parte de cálculos no desenvolvimento precisou desligar-se do projeto, por motivos particulares. Então, o foco foi reajustado para um jogo de carros com modelos físicos menos complexos e com enfoque no controle dos veículos autônomos e na criação de uma cidade em três dimensões.

3.1.1 Metodologia de Desenvolvimento

O desenvolvimento do jogo foi dividido em três etapas principais.

Inicialmente houve a criação da base do *software*, concepção ou obtenção de modelos 3D, definição de classes. As principais são: CCar, CComputerCar, CSpot e CTable.

Em seguida, foram discutidas maneiras para representar em um grafo as ruas de uma cidade. Foi implementada uma versão inicial e, em seguida, foram criados os NPCs.

Por fim, desenvolveu-se a estrutura de dados que armazena os dados relevantes da cidade. Modelou-se a cidade tridimensional e então o desenvolvimento passou a se focar no melhoramento do controle dos veículos dirigidos pelo computador.

3.1.2 Paradigma de Desenvolvimento

O paradigma de desenvolvimento orientado a objetos é largamente utilizado atualmente, e no desenvolvimento de jogos não poderia ser diferente (PENTON 2003). Programar um jogo estruturado hoje em dia é um atraso. As vantagens conhecidas de orientação a objetos: reuso, organização do código, menor custo para manutenções futuras e ampliação do ciclo de vida do *software* são atributos essenciais em um jogo eletrônico. Por esse motivo, o jogo foi desenvolvido em C++, que é uma linguagem de programação O.O. bastante utilizada no desenvolvimento de jogos e que possui sintaxe semelhante a C e Java.

Mas por que não desenvolver em Java, já que é uma linguagem orientada a objetos muito comentada e apreciada?

Jogos eletrônicos são divididos em jogos “*hardcore*”, mais elaborados e complexos, que tem como público-alvo jogadores mais assíduos; e jogos “casuais”, que têm como objetivo distração rápida e sem compromissos. Jogos “*hardcore*” desenvolvidos para consoles e computadores são, em sua maioria, desenvolvidos em C++. Alguns motivos são o fato dessa linguagem ser compilada e possuir em geral um desempenho maior que Java, uma linguagem interpretada; o fato de permitir acesso direto a recursos do sistema operacional; como acessar DLLs e recursos de *hardware*; e como inserir programação Assembly dentro do código fonte C++.

Os avanços incríveis no desempenho dos anos 80 e 90 do século passado tornaram a interpretação viável para muitas aplicações importantes, mas um fator de atraso de 10 vezes, em comparação com programas em C compilados tradicionalmente, tornou o Java pouco atraente para algumas aplicações (PATTERSON; HENNESSY, 2005).

3.2 LEVANTAMENTO DE CLASSES

Como já explicado anteriormente, o levantamento de requisitos em jogo eletrônico consegue ser mais complexo do que em um *software* comum, por esse motivo, algumas classes só foram adicionadas ao projeto durante a fase de implementação e levou um tempo considerável para chegar ao modelo de classes atual.

Na figura 8 pode-se visualizar o diagrama de classes final do projeto. Nele, a classe CComputerCar cuida do comportamento dos carros comandados pelo computador, enquanto que a classe CCar é responsável pelo carro controlado pelo jogador. CCity é a responsável pela cidade na qual o jogo é ambientado. Por sua vez, CEventReceiver, que é uma classe filha da IEventReceiver, classe do próprio Irrlicht, trata os eventos disparados pelo teclado. Finalmente, na CGlobals estão contidas todas as variáveis globais, utilizadas por praticamente todas as classes do jogo. A classe CSemaphore cuida dos semáforos do jogo que serão implementados no futuro. As outras classes importantes serão explicadas nas seções seguintes.

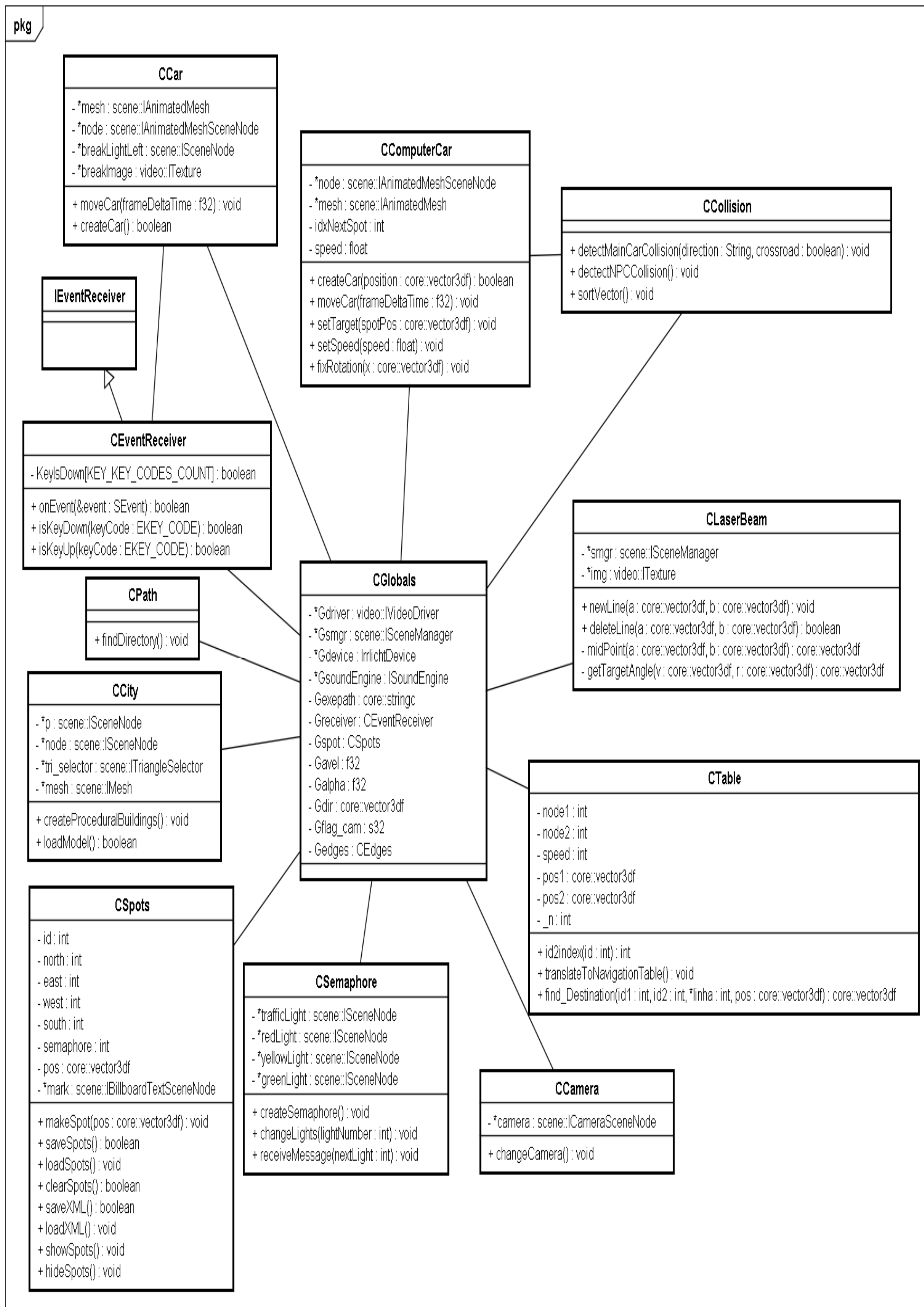


Figura 8 - Diagrama de Classes do Jogo
Fonte: Autoria Própria

3.3 REPRESENTAÇÃO DA CIDADE

Uma das principais e mais complexas etapas do projeto, a criação da cidade 3D, demandou várias pesquisas de métodos de geração de terrenos e ambientes em jogos.

3.3.1 Cidade Procedural

Primeiramente, a ideia era a geração procedural de uma cidade através de código e preenchimento dessa cidade com texturas. Em sua dissertação de mestrado, Illangovan (2009) demonstra o processo de criação de uma ferramenta que gera layout de uma cidade com renderização em tempo real, em síntese, o autor escreve scripts na linguagem Python, para que o *software* Houdini (HOUDINI, 2011) interprete e gere uma cidade automaticamente, a partir dos *scripts*. O processo é extremamente complexo e não caberia nesse projeto, porém em aplicações com grandes equipes e orçamentos a ideia é bastante interessante, pois é uma das maneiras mais interativas, entre as estudadas, para geração procedural.

Quando a ideia da montagem da cidade ainda era procedural, pensou-se em uma solução que leria um arquivo XML com dados sobre cada prédio, como altura, largura, posição e então o jogo geraria a cidade a partir desse arquivo.

A ideia de prédios por XML foi abandonada e seguiu-se com prédios gerados a partir de figuras geométricas do motor Irrlicht, por fim foram colocadas texturas nessas figuras, obtendo assim o visual de prédios. Na figura 9 tem-se a primeira cidade construída, a cidade procedural, com o jogo em execução.

Figura 9 - Jogo em Execução com a Primeira Cidade Desenvolvida
Fonte: Autoria Própria

3.3.2 Cidade Modelada

Outra forma possível de se criar um ambiente urbano em um jogo é a utilização de uma ferramenta de modelagem 3D para construir a cidade. Atualmente, existe uma infinidade de ferramentas de modelagem para computação gráfica no mercado e, muitas destas, geram formatos de arquivos compatíveis com o *engine* Irrlicht.

Nenhum gerador procedural pronto para uso foi localizado. Desenvolver uma solução desse tipo implica uma escolha cuidadosa das estruturas e dos algoritmos associados, além de atribuir valores adequados a parâmetros controlando a geometria da cidade e fatores como ramificação de vias. Dado o requisito de modelar uma paisagem buscando algum realismo gráfico, decidiu-se adotar uma estratégia diferente. Foi escolhido usar uma ferramenta de desenho

3D para criar o cenário e, posteriormente, alimentar uma estrutura de dados com informações sobre a localização de ruas e cruzamentos.

Portanto, a escolha final por uma cidade pré-renderizada deveu-se ao fato de ser mais prático criar uma cidade com diversidade visual e estrutural com o auxílio de uma ferramenta de modelagem.

3.3.3 Relevo

Com a intenção de tornar a cidade virtual mais realista, verificou-se a necessidade de variar o relevo do ambiente. Uma técnica de criação de relevo para cenários de jogos compatível com Irrlicht é a Height Field (Mapa de Altura).

Mapa de Altura é basicamente uma imagem bidimensional, na qual são armazenados valores relativos à elevação de determinada superfície. Esses valores são interpretados e utilizados para a exibição de gráficos tridimensionais. As imagens são mostradas em escala de cinza, onde o preto representa a menor altitude e o branco a maior, dessa forma, quando forem interpretadas poderão mostrar as diferenças no relevo em três dimensões (GARLAND; HECKBERT, 1995).

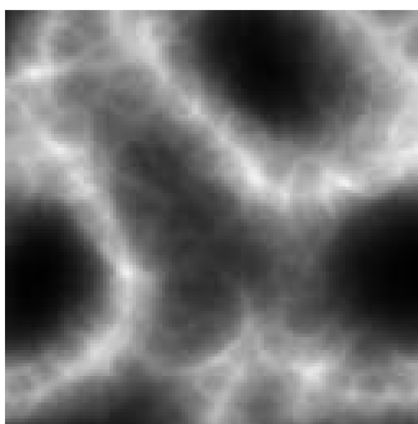


Figura 10 - Height Field
Fonte: Hardware Rendering Competition (2005)

Na primeira representação de cidade criada para o trabalho utilizou-se o Height Field para se obter o relevo. O *engine* Irrlicht, em seu método de criação

de terrenos, possui como um dos parâmetros o caminho de um arquivo de imagem que representa o Height Field, isso facilita o trabalho com esse tipo de relevo. O arquivo foi criado em um simples editor de imagens e carregado com o método de geração de terrenos. Dessa forma, obteve-se relevo para a primeira cidade construída para o jogo.

Contudo, para a cidade final o Height Field não se aplicava, porque todo o terreno foi modelado em ferramenta própria, inclusive com variações de relevo. O Height Field foi, portanto, descartado junto com a primeira cidade.

3.4 BIBLIOTECA DE SONS

Desde os primeiros jogos da história, sempre estiveram presentes efeitos sonoros. Inicialmente, eram simples sons, com pequenas diferenças entre si, pois as placas de sons dos *arcades*, consoles e PCs estavam começando a surgir, no entanto, desde aquela época, efeitos sonoros e música são responsáveis por causar mais imersão nos jogadores.

Neste jogo de simulação de carro não poderia ser diferente, sons e música precisavam estar presentes. Para isso foi utilizada a IrrKlang, biblioteca de áudio multiplataforma para desenvolvimento de jogos eletrônicos (IRRKLANG, 2011).

IrrKlang é uma biblioteca de fácil entendimento e utilização, mas nem por isso deixa de ser completa e possuir funcionalidades suficientes para satisfazer as necessidades dos desenvolvedores dessa área. Essa biblioteca suporta uma quantidade considerável de formatos de arquivos de áudio, além de possuir métodos para execução de sons e músicas em 2D e 3D.

Usar a biblioteca é relativamente simples, basicamente são usados os métodos:

```
soundEngine->play2D("../../media/sound.wav", true);
soundEngine->play3D("../../media/music.mp3", true);
```

Esses dois métodos são responsáveis por executar todos os sons, tanto músicas quanto efeitos sonoros. A diferença entre os dois métodos é que sons 3D possuem maior sensação de imersão. O primeiro parâmetro dos métodos é o caminho do arquivo a ser tocado, o segundo parâmetro especifica se o arquivo será tocado em *loop* ou não. Tocar som em *loop* é vantajoso quando se deseja que a música não pare de tocar enquanto o jogo estiver em execução.

Foram incluídos no simulador, efeitos como som de partida no carro, buzina e sons simples na interface de usuário. Além disso, foram colocadas músicas para o jogo em execução e para o menu inicial.

3.5 MAPEAMENTO DA CIDADE

Um arquivo XML foi utilizado para registro de informações de pontos específicos da cidade para que eles pudessem ser utilizados na criação de um grafo. Esse grafo é uma estrutura de dados essencial nesse jogo, pois é utilizada para que carros controlados pelo computador possam se guiar pelas ruas e saber por onde podem ou não trafegar.

A figura 11 demonstra como deve ser a divisão de ruas entre os quarteirões. Em cada final de quarteirão tem-se um nó do grafo e cada aresta representa uma via. A figura é meramente ilustrativa, pois é possível ter mais que duas vias para cada rua da cidade.

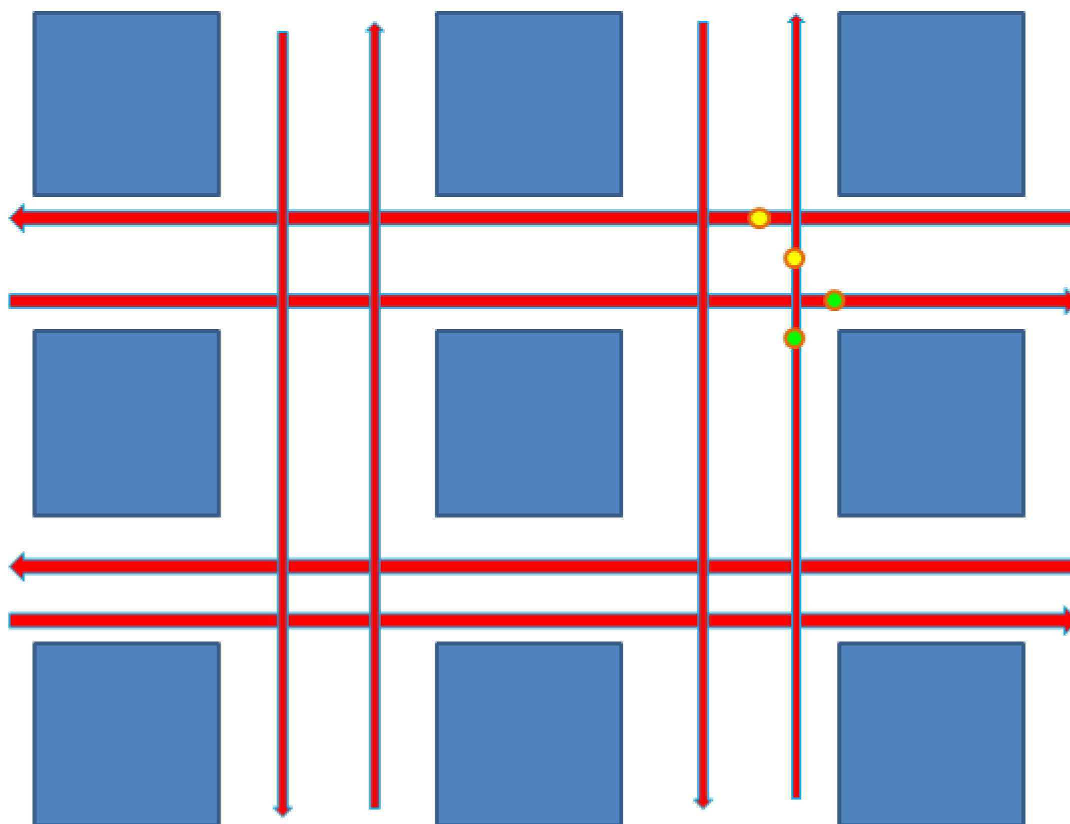


Figura 11 - Exemplo de Divisão de Ruas da Cidade
Fonte: Autoria Própria

Para a criação do grafo, foi aproveitada uma ideia desenvolvida em um trabalho de iniciação científica anterior (POZZO; KOSCIANSKI, 2010). Criou-se a possibilidade de navegar pela cidade criando marcas nos cruzamentos, dessa maneira obtendo facilmente as coordenadas 3D (os nós do grafo possuem coordenadas X, Y e Z, assim, é possível armazenar a topografia do cenário) desses pontos para alimentar o grafo que representa o mapa de ruas. Esses pontos podem ser marcados ao andar pela cidade com o carro e apertar a tecla M. Os pontos são arquivados em formato XML e o grafo corresponde a duas estruturas de dados, apresentadas no quadro 2.

class CSpots {	struct s_Table {
int id ,	
north, west ,	int node1, node2;
east , south,	
semaphore ;	float speed;
core::vector3df pos;	irr::core::vector3df pos1, pos2;
scene::IBillboardTextSceneNode *mark;	};
};	

Quadro 2 - Estruturas de Dados Usadas para Representar as Vias
Fonte: Autoria Própria

A classe CSpots contém a posição de cada cruzamento na cidade, identificados numericamente, além da identificação de pontos de saída. Essa estrutura é empregada por uma máquina de estados finitos para escolher de forma aleatória o próximo destino. Com essa informação, uma tabela contendo a estrutura S_Table é percorrida e uma via é selecionada. A tabela permite registrar linhas paralelas correspondendo a ruas e avenidas com várias faixas.

```xml
<?xml version="1.0"?>
<Spot>
    <attributes>
        <int name="ID" value="1" />
        <float name="PosX" value="102.152992" />
        <float name="PosY" value="4.000000" />
        <float name="PosZ" value="100.000000" />
        <int name="north" value="2" />
        <int name="west" value="10" />
        <int name="east" value="-1" />
        <int name="south" value="-1" />
        <int name="semaphore" value="0" />
    </attributes>
</Spot>
<Spot>
    <attributes>
        <int name="ID" value="2" />
        <float name="PosX" value="207.899277" />
        <float name="PosY" value="4.000000" />
        <float name="PosZ" value="100.000000" />
        <int name="north" value="3" />
        <int name="west" value="9" />
        <int name="east" value="-1" />
        <int name="south" value="1" />
        <int name="semaphore" value="0" />
    </attributes>
</Spot>
<Spot>
    <attributes>
        <int name="ID" value="3" />
        <float name="PosX" value="316.451080" />
        <float name="PosY" value="4.000000" />
        <float name="PosZ" value="100.000000" />
```

Figura 12 - Arquivo CityConfig.xml que Contém as Informações dos Spots
Fonte: Autoria Própria

Na figura acima está o arquivo XML gerado a partir da marcação dos cruzamentos da cidade, com o carro do jogador. Os atributos "*north*", "*south*", "*east*" e "*west*" contém o número identificador dos nós vizinhos ao nó em questão. O valor -1 identifica que não existe nó vizinho para essa determinada coordenada.

3.5.1 Tabela de Navegação

A solução final desenvolvida para o desafio do mapeamento da cidade foi a utilização de uma tabela de navegação. Após serem marcados os pontos no cenário utilizando o carro principal e, consequentemente, ser gerado o arquivo XML, é a hora da criação do arquivo base para a tabela.

3.5.1.1 Arquivo de dados

O arquivo base para a tabela contém dados sobre as ruas. Ele possui informações sobre o nó de origem e nó de destino de determinada rua, a largura total da rua, o número de pistas de cada rua e informações de velocidade e sentido de cada pista (se a velocidade for negativa, o sentido é invertido). O símbolo # é interpretado como comentário e, portanto a linha é ignorada pelo método leitor do arquivo. Quando o leitor encontra uma linha que começa com o valor -1, ele sabe que o arquivo acabou.

```
1  #node1, node2, largura, numero de pistas, velocidade cada uma
2  1,2,40,4,-60,-60,60,60,
3  2,3,40,4,-60,-60,60,60,
4  3,4,40,4,-60,-60,60,60,
5  4,5,40,4,-60,-60,60,60,
6  #-----------------------------------------
7  10,9,40,4,-60,-60,60,60,
8  9,8,40,4,-60,-60,60,60,
9  8,7,40,4,-60,-60,60,60,
10 7,6,40,4,-60,-60,60,60,
11 #-----------------------------------------
12 1,10,40,4,-60,-60,60,60,
13 2,9,40,4,-60,-60,60,60,
14 3,8,40,4,-60,-60,60,60,
15 4,7,40,4,-60,-60,60,60,
16 5,6,40,4,-60,-60,60,60,
17 -1,5,40,4,60,60,-60,-60,
```

Figura 13 - Arquivo que Contém Informações Sobre as Ruas
Fonte: Autoria Própria

A figura 13 mostra um exemplo do arquivo base para a navegação. Embora, neste exemplo, todas as ruas tenham sido dividas em quatro pistas, o formato definido permite que cada rua tenha um número diferente de pistas se necessário.

As velocidades indicadas em cada pista são usadas para controlar a velocidade máxima dos carros pilotados pelo computador. Atualmente os NPCs

trafegam a uma velocidade fixa, mas o teste já foi implementado pensando em trabalhos futuros para ampliar o controle e a inteligência artificial dos NPCs.

A estrutura projetada permite também indicar faixas de estacionamento. Para isso bastará definir a velocidade como zero e incluir o controle necessário na lógica dos veículos.

3.5.1.2 Classe CTable

A leitura do arquivo base para a tabela é feita através do método *readTable()* pertencente a classe CTable. Em seguida, é chamado o método *translateToNavigationTable()*, que é o responsável por converter os dados do arquivo base para informações que possam ser efetivamente utilizadas pelos carros.

O método *translateToNavigationTable()* é o responsável por criar uma tabela em memória para os carros autônomos poderem se basear. Esse método também efetua a divisão justa de largura entre as pistas, pois ele divide a largura total da rua pelo número de pistas e organiza cada pista em seu devido lugar.

A ideia principal por trás da tabela é o fato de ela funcionar com alocação estática de memória. Dessa forma evita possibilidades de erros com ponteiros perdidos e, adicionalmente, permite acelerar acesso.

A figura 14 mostra uma representação de como a tabela de navegação parece após ser construída pelo jogo.

Tabela				
node1	node2	posxyz_origem	posxyz_destino	velocidade
1	2	102,15299; 4; 100	207,15299; 4; 100	60
1	2	112,89927; 4; 100	217,89927; 4; 100	60
2	3	207,89927; 4; 100	316,45108; 4; 100	60
2	3	217,89927; 4; 100	326,45108; 4; 100	60
2	3	227,45108; 4; 100	336,89927; 4; 100	60
2	3	237,45108; 4; 100	346,89927; 4; 100	60

Figura 14 - Tabela de Navegação
Fonte: Autoria Própria

A tabela de navegação reside apenas na memória durante a execução do jogo; essa estrutura não é gravada em disco.

3.5.1.3 Classe CLaserBeam

Para poder verificar como ficaram as ruas e suas respectivas pistas, foi utilizada a classe CLaserBeam. Dado um ponto inicial e um ponto final, essa classe cria linhas coloridas entre esses dois pontos. Em um laço de repetição, todas as linhas da tabela de navegação foram utilizadas para criar os "*laser beams*" que representam as vias.

A figura 15 exemplifica a representação das ruas utilizando a classe CLaserBeam.

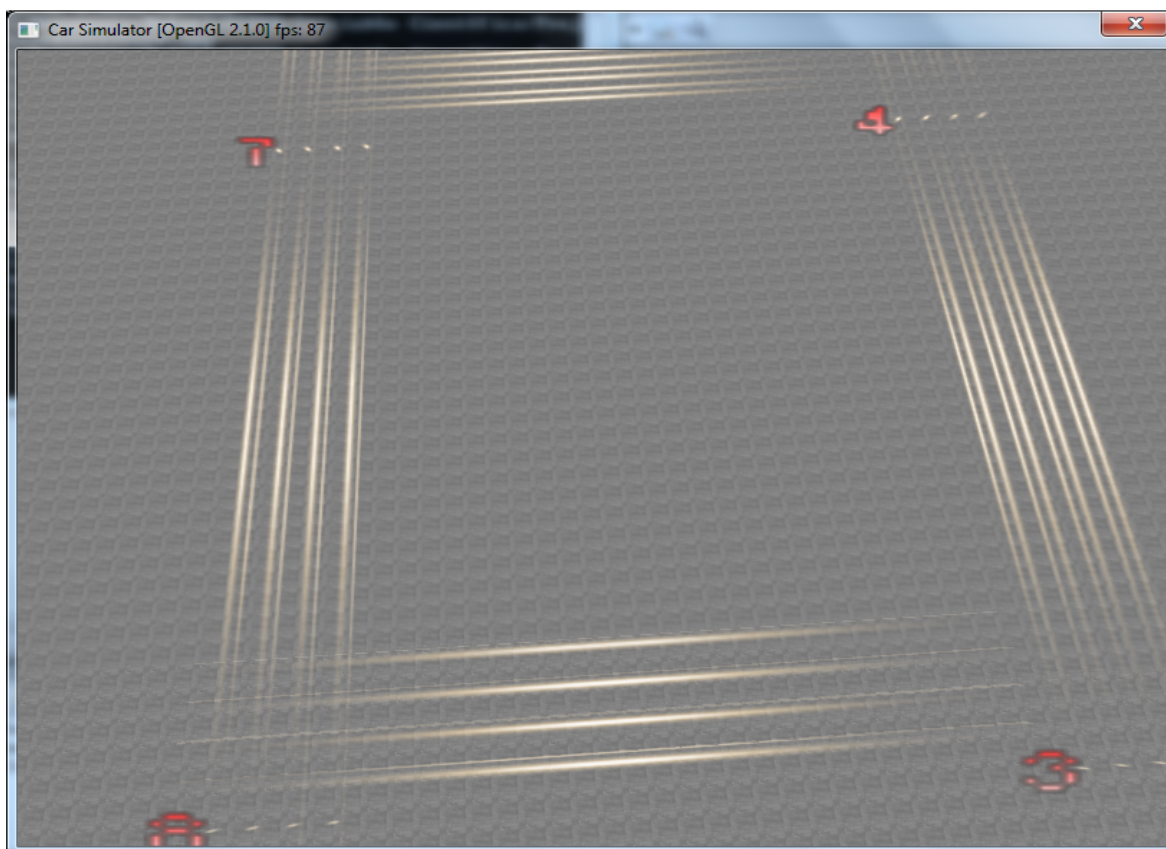


Figura 15 - Modo de Visualização do Grafo
Fonte: Autoria Própria

Estão visíveis na figura 15 também os nós (*spots*) marcados, com seus respectivos números identificadores. As linhas amarelas representam as vias, e os números vermelhos representam as esquinas ou cruzamentos.

3.6 MOVIMENTAÇÃO DO CARRO

A ideia principal em qualquer jogo computacional é proporcionar interação agradável aos jogadores, em um jogo de carros essa interação se dá de maneira que o jogador tenha a sensação de estar no controle total do veículo.

3.6.1 Visão Geral de Movimento

O movimento do veículo controlado pelo jogador ocorre através de uma sequência de passos:

1. O jogador pressiona uma tecla de direção.
2. A classe receptora de eventos captura o evento.
3. A classe controladora do carro verifica o evento e decide que resultado produzir com esse evento.
4. Cálculos e processamento de código são efetuados.
5. O resultado é gerado na tela para o jogador.

Em qualquer jogo digital existe uma entidade que lê e trata os eventos adequadamente, em um jogo de carro ou de simulação, no entanto, há alguns passos a mais, que são os cálculos de física e matemática, pois sem eles o jogo ficaria com aspecto muito artificial e os movimentos não fariam sentido.

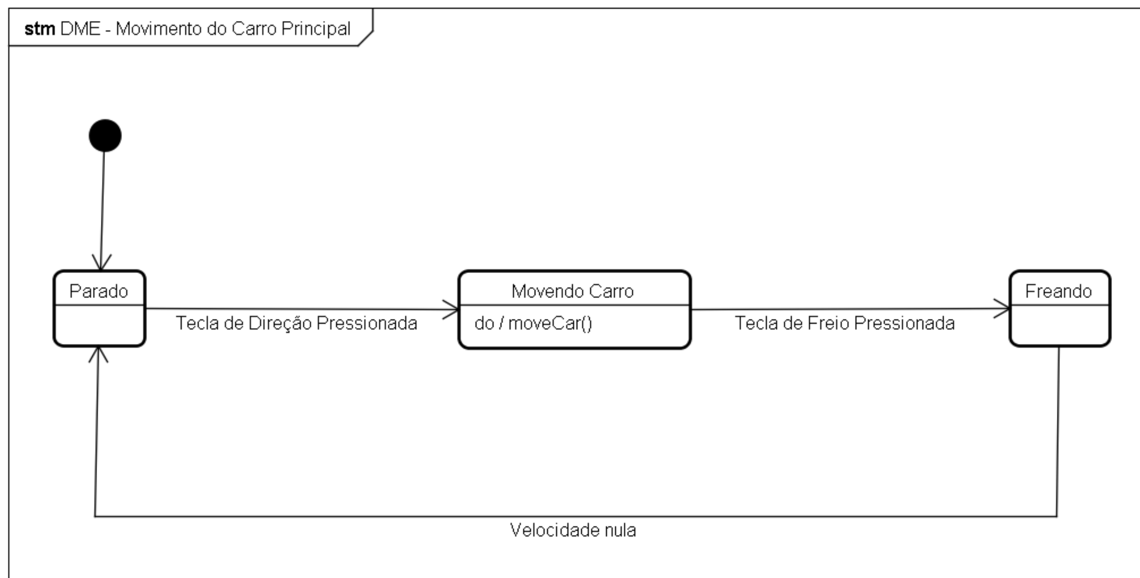


Figura 16 - Diagrama de Estados - Movimento do Carro Principal
Fonte: Autoria Própria

A aceleração, a marcha ré e as curvas que o carro principal faz estão dentro do estado “Movendo Carro” e é nele que o processamento é efetuado.

3.6.2 Controles

No jogo estão presentes comandos que realizam as mais diversas ações. Há comandos que fazem o carro frear, acelerar, dar marcha ré, deslocar-se para os lados, buzinar, além de teclas que efetuam as trocas entre as câmeras.

Todos esses comandos são executados por meio do teclado, porém o *engine* permite a adição de outras formas de controle de jogo, como *joysticks* e *gamepads*, que podem ser ofertados como opção ao jogador no futuro.

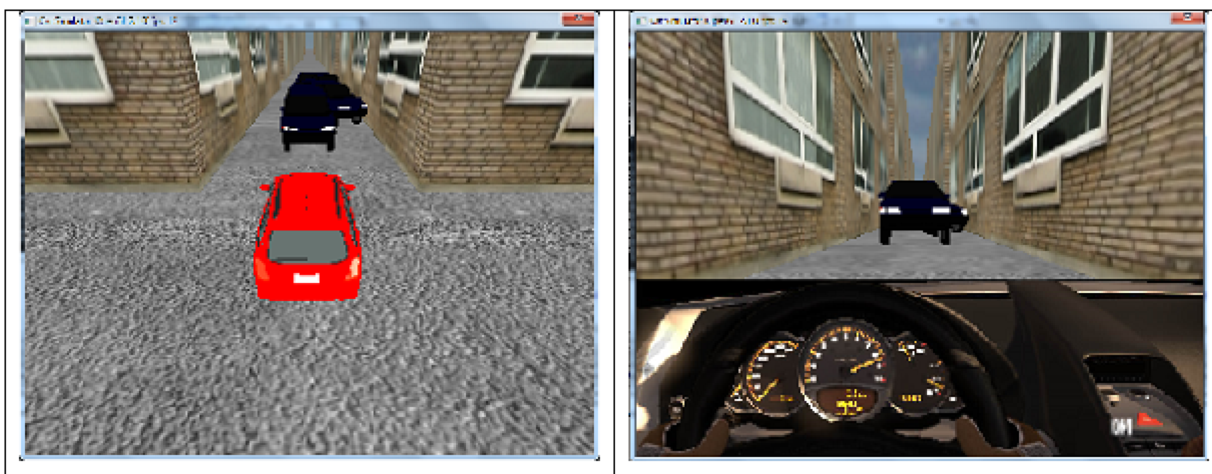
Além dos comandos utilizados pelo jogador, existem teclas específicas para criação ou edição do grafo. Para isso existem teclas que marcam *spots*, escondem ou mostram os *spots* marcados e teclas que escondem ou mostram as linhas que ligam esses pontos.

3.6.3 Câmera

O conceito de câmera em jogos eletrônicos é semelhante ao presente em um filme, ou seja, é através das “lentes” da câmera que o telespectador ou jogador visualiza toda a ação, portanto elas agem como os olhos do jogador.

No motor Irrlicht existem diversos tipos de câmeras nativas. As câmeras mais comuns do Irrlicht são a Normal Camera e FPS Camera. A Normal Camera é ajustada para que sempre “olhe” em direção a determinada parte da cena. Já a FPS Camera é baseada e utilizada em jogos em primeira pessoa, como se a câmera fosse os olhos do personagem sendo utilizado no jogo.

É comum que exista mais de uma câmera em jogos de corrida, geralmente uma câmera externa que segue o veículo e uma interna que dá a sensação de que o jogador está realmente dirigindo o carro e que gera mais imersão. Neste simulador de carros, os dois modelos de câmera citados estão presentes.



Quadro 3 - O Mesmo Momento no Jogo, Visto com Duas Câmeras Diferentes
Fonte: Autoria Própria

No decorrer do desenvolvimento, verificou-se que a câmera estava se tornando complexa e tendo bastante responsabilidade, então criou-se uma classe específica para manipulação das câmeras, a CCamera.

Além da câmera interna ao veículo e da câmera externa que segue o automóvel a uma distância constante, foi implementado um terceiro tipo de

câmera. Essa câmera possui uma visão aérea da cidade, semelhante à câmera de um helicóptero, e com ela é possível visualizar toda a cidade.

Figura 17 - Câmera Aérea
Fonte: Autoria Própria

A câmera da figura acima aceita comandos através das teclas de direção e também do *mouse*, por isso, é possível percorrer a cidade e ver o comportamento dos carros controlados pelo computador, através dela.

3.6.4 Movimento dos Modelos 3D

Após ser desenvolvida a estrutura de dados utilizada no mapeamento da cidade, a implementação seguiu com o refinamento do controle dos carros presentes no jogo.

Para posicionar os carros no jogo foi necessário rotacionar os modelos 3D. Primeiramente é corrigida a orientação de cada modelo e sua escala para coincidir com os eixos X, Y e Z dentro do *software*. Em seguida, é aplicada uma rotação para fazer a direção do modelo coincidir com um vetor paralelo à via percorrida. Esse cálculo foi realizado facilmente com duas funções presentes no Irrlicht:

```
irr::core::vector3df requiredRotation = direction.getHorizontalAngle();
node->setRotation (requiredRotation + _fixRotation);
```

3.7 CONTROLE DE NPCS

No caso desse jogo de carros, os NPCs são os veículos autônomos que compõem o trânsito da cidade e, como foi dito anteriormente, a orientação desses personagens é um dos focos principais deste trabalho.

3.7.1 Orientação dos Carros do Computador

Além do carro comandado pelo teclado, o jogo contém carros controlados pelo computador. A tabela de navegação construída corresponde a um grafo representando as ruas dentro de uma cidade. Uma vez carregado em memória, o grafo é usado pelos NPCs para tomar decisões. Uma máquina de estados foi empregada para controle dos veículos. Ela contém dois estados básicos: eGO, que apenas faz o carro se mover; eCHOOSE, que escolhe um novo destino, a partir do grafo, quando o carro chega a um cruzamento.

Cada aresta do grafo é modelada como uma linha de S_Table, contendo referências aos carros que nela transitam. Esse objeto pode ser consultado pelos veículos para verificar a possibilidade de colisões. Com esse particionamento do espaço do jogo (KOSCIANSKI, 2011), é possível reduzir o número de testes de

colisão e, consequentemente, reduzir o gasto de CPU, pois os testes são feitos somente com os objetos presentes no espaço particionado em que o jogador se encontra em determinado instante e não com todos os objetos presentes na cidade.

A máquina de estados que controla os carros do computador funciona da seguinte forma:

1. O carro anda por uma rua até a esquina.
2. Ao chegar ao final, consulta-se o *spot* para saber se é possível ir para esquerda, frente ou direita.
3. Escolhe-se um desses três caminhos aleatoriamente.
4. Ao escolher, o carro recebe o índice do próximo *spot*.
5. Com o número do *spot*, ele obtém a próxima via e todos os dados necessários sobre ela.
6. Se o *spot* escolhido está à esquerda ou à direita da posição atual do carro, ele passa para o estado eCURVE, que é quando o carro é rotacionado de forma suave, evitando movimentos bruscos e irreais.

As figuras 18 e 19 mostram a máquina de estados do controle dos carros. Na figura 18 tem-se um diagrama de máquina de estados da UML e na figura 19 um trecho de código.

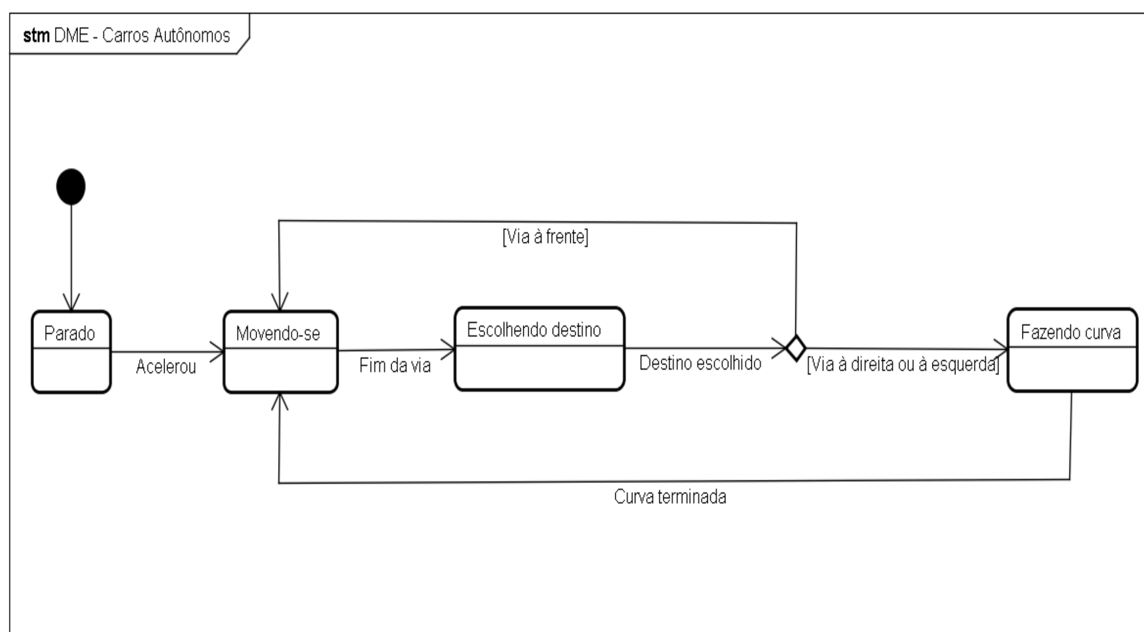


Figura 18 - Diagrama da Máquina de Estados – Veículos Autônomos
Fonte: Autoria Própria

```
switch (_state) {

        case eSTOP: {

        }

        case eGO: {

        }

        case eCHOOSE: {

        }

        case eCURVE: {

        }

    } //switch
```

Figura 19 - Estrutura Switch da Máquina de Estados
Fonte: Autoria Própria

O estado eSTOP é quando os carros estão parados por algum motivo; estado eGO é aquele no qual os NPCs passam mais tempo, ou seja, é quando estão se deslocando; eCHOOSE é o estado da escolha pelo próximo *spot*, ou seja, é quando o carro consulta a tabela de navegação para escolher a próxima via; e eCURVE ocorre quando os carros precisam fazer curvas.

Na figura 20, têm-se os veículos autônomos trafegando pelas vias, representadas pelas linhas amarelas.

Figura 20 – Carros Autônomos Trafegando Pela Cidade
Fonte: Autoria Própria

3.8 CINEMÁTICA DOS MOVIMENTOS

O projeto original previa a simulação de equações de dinâmica dos veículos; entretanto, como foi dito, a pessoa responsável pelo estudo e desenvolvimento dessa parte do projeto precisou deixar a equipe. Assim, para a movimentação dos veículos optou-se por cálculos de cinemática simples.

O carro pilotado pelo usuário é acelerado ao ser pressionada uma tecla, simulando um pedal de acelerador. Como em outros jogos, o carro perde aceleração aos poucos caso a tecla seja solta, simulando o que acontece no mundo real. Outra tecla funciona como freio, que não desacelera o carro imediatamente, mas reduz a velocidade do carro até chegar a zero. O movimento do carro é controlado por duas equações:

$$pos_new = pos_old + dir * vel * \Delta t \qquad (1)$$

$$vel = vel + acc * \Delta t \qquad (2)$$

As equações (1) e (2) são calculadas dentro de um laço de animação. São usadas tanto no carro do jogador quanto nos controlados pelo computador. A variável Δt é dada pelo tempo gasto na última iteração do laço. Isso faz com que os carros andem na mesma velocidade independente do computador usado para executar o programa. A essa técnica é dado o nome de "*frameDeltaTime*", com ela é possível ter velocidade independente da taxa de frames (*frame rate*) do jogo.

Todas as equações de Física pertinentes ao carro do jogador estão implementadas na classe CCar, e o método *moveCar*() é o principal responsável por elas.

3.9 COLISÃO

Em praticamente todos os jogos do mercado existe algum tipo de detecção e tratamento de colisões entre as entidades existentes. Como foi dito anteriormente, optou-se por particionar o espaço do jogo com o intuito de evitar testes desnecessários de colisão.

No *engine* Irrlicht existem métodos de colisão prontos para uso, entretanto usá-los nesse projeto acabaria com o desempenho, pois os testes do Irrlicht sempre testam a colisão entre todos os objetos do jogo. Isso tornaria o jogo inutilizável, visto que ficaria absurdamente lento.

Neste trabalho, verificou-se a necessidade de detectar dois tipos de colisão: entre os veículos autônomos e entre o carro do jogador e os prédios. Não é necessário verificar colisão entre os veículos autônomos e os prédios, pois eles andam de acordo com o grafo e este, se criado corretamente, não invade os prédios. Os dois tipos de colisão citados são verificados pela classe CCollision.

3.9.1 Colisão entre NPCs

A solução para detectar colisão entre carros autônomos foi realizada utilizando-se um vetor global de carros do computador. Esse vetor global contém os automóveis autônomos do jogo. Para detectar a colisão, primeiramente ordena-se esse vetor, de acordo com a posição dos veículos, e, em seguida, testam-se os carros que estão próximos e têm perigo de colidirem. Essas tarefas são realizadas com os métodos *dectectNPCCollision*() e *sortVector*() da classe CCollision.

É importante ressaltar que os processos de detecção de colisão e de ordenação do vetor são realizados uma vez a cada 1 segundo, assim economiza-se uma grande parcela de processamento.

Para tratar a colisão detectada, um dos carros que colidiu é obrigado a permanecer parado por determinado intervalo de tempo, enquanto o outro continua sua trajetória, isso faz com que os dois carros se separem.

3.9.2 Carro Principal e Prédios

Para detectar colisão do carro do jogador com os prédios utilizou-se um artifício diferente. Ao invés de fazer o teste de colisão entre os modelos dos prédios e do automóvel, optou-se por testar se o carro saiu da rua e invadiu o espaço da calçada.

Isso ocorre da seguinte maneira:

1. Primeiramente, verifica-se se o carro está ou não em um cruzamento.
2. Caso esteja em um cruzamento o teste não é efetuado, visto que não há perigo de colisão com prédios nessa situação. Se não estiver em um cruzamento, procura-se o *spot* mais próximo e obtêm-se as coordenadas X e Z desse *spot*.

3. Em seguida, é preciso identificar se o sentido do carro é norte/sul ou leste/oeste.

4. No caso de o carro estar em sentido norte/sul, obtém-se o desvio deste em relação ao eixo Z, baseando-se na posição Z do *spot* mais próximo, e, se o desvio for maior que a largura da rua, o carro colidiu. Caso o sentido seja leste/oeste a única diferença é a obtenção do desvio em relação ao eixo X.

5. Se a colisão for detectada, uma mensagem aparece na tela para o jogador em uma caixa de mensagem.

A figura 21 mostra o código do método de detecção de colisão do carro principal.

```
29  void CCollision::detectMainCarCollision(char* going, float coordinate) {
30      //! Verificação de colisão com prédios
31      int detour;
32
33      //! Carro movendo-se sentido Norte/Sul
34      if (strcmp(going,"NorthSouth") == 0) {
35          detour = (GmainCar.node->getPosition().Z - coordinate);
36      }
37
38      //! Carro movendo-se sentido Leste/Oeste
39      if (strcmp(going,"EastWest") == 0) {
40          detour = (GmainCar.node->getPosition().X - coordinate);
41      }
42
43      if ((detour <= -8) || (detour > 8))  {
44          if (window == NULL)
45              window = Gdevice->getGUIEnvironment()->addMessageBox(L"COLLISION", L"You got off the street and collided!");
46      }
47
48      else {
49          if (window != NULL) {
50              window->remove();
51              window = NULL;
52          }
53      }
54  }
```

Figura 21 - Método de Detecção de Colisão do Carro Principal
Fonte: Autoria Própria

A desvantagem dessa solução para colisão é que ela funciona apenas para ruas retas. Para ruas curvas, seria preciso utilizar outros artifícios.

3.10 INTERFACE DE USUÁRIO

Interfaces de usuário são elementos importantes em praticamente qualquer *software*. O crescente interesse no projeto de interfaces do usuário é bastante claro nos mais variados tipos de sistemas (BARANAUSKAS; ROCHA, 2003). Em jogos digitais isso também é uma realidade. Por esse motivo, foi decidido que, para o simulador de carro ficar mais atraente, seria necessário que ele possuísse uma interface de usuário amigável.

O *engine* Irrlicht possui bibliotecas próprias para trabalhar com Interfaces Gráficas de Usuário (GUI), entretanto, esse não é o ponto mais forte do Irrlicht e geralmente as GUIs feitas utilizando o Irrlicht não possuem *designs* coloridos e arrojados, ideais para jogos.

Após verificar outras alternativas para a concepção de uma interface adequada para o jogo, chegou-se à conclusão que utilizando um bom *software* de edição de imagens seria possível gerar a interface certa para os menus. Como o Irrlicht trabalha bem com os mais variados formatos de imagens, após os desenhos das interfaces estarem prontos, seria necessário apenas carregar os arquivos do disco e exibi-los na tela quando necessário. Desse modo foram criadas as telas do menu principal, a tela de opções e a tela de créditos (muito comum em jogos digitais).

A título de curiosidade, uma alternativa para criar ótimas interfaces de usuários é utilizar o Adobe Flash (ADOBE FLASH, 2011). Com o Adobe Flash é possível criar interfaces agradáveis compatíveis com o Irrlicht.

Na figura 22 é possível visualizar o menu principal, criado com o *software* livre de edição de imagens, GIMP (GIMP, 2011). A seta à esquerda é utilizada para sinalizar ao jogador a opção selecionada no momento. Para escolher determinada opção o jogador utiliza a tecla “*enter*”.

Figura 22 - Menu Principal do Jogo
Fonte: Autoria Própria

4 CONCLUSÃO

Este trabalho apresentou o desenvolvimento de um jogo simulador de carros. Foram mostradas várias técnicas que podem ser utilizadas para representar uma cidade virtual. Além disso, foi criado um sistema de mapeamento do ambiente de jogo, que torna possível orientar os NPCs pelo ambiente. Esse sistema pode ser reaproveitado por outros jogos.

O resultado do trabalho foi um jogo de carros ambientado em um cenário urbano. Nele é possível dirigir por uma cidade onde estão presentes outros carros controlados pelo computador. O jogador pode dirigir utilizando três tipos diferentes de câmeras: câmera interna ao carro, externa e aérea.

A cidade a ser utilizada pode ser criada em um editor de modelos 3D e posteriormente importada pelo jogo. O grafo representando as ruas é criado dentro do jogo, marcando-se os cruzamentos e, depois, indicando as ruas existentes (ou arestas do grafo) e, para cada rua, as pistas. As informações armazenadas em formato XML e “txt” podem ser ajustadas fora do jogo.

A maneira como foi construído o sistema de mapeamento da cidade permite que o cenário tridimensional possa ser trocado, ou que mais cidades sejam adicionadas ao jogo, pois o mapeamento pode ser refeito em tempo de execução.

Os carros controlados pelo computador se orientam pelas ruas mapeadas; ao chegarem a um cruzamento, decidem aleatoriamente qual será sua nova rua, baseada nas possibilidades; e fazem testes para evitar colisões. Foram utilizados três modelos 3D para os carros autônomos: uma viatura de polícia, uma caminhonete e um carro de passeio comum, entretanto mais modelos podem ser adicionados com facilidade, precisando somente ajustar suas rotações e escalas.

O trabalho também demonstrou maneiras de tratar colisões entre os objetos em jogos, como empregar músicas e efeitos sonoros e como criar interfaces gráficas para jogos digitais.

O projeto do simulador rendeu também um artigo científico (CANTERI; KOSCIANSKI, 2011), que foi publicado no décimo sexto SICITE, Seminário de Iniciação Científica e Tecnológica da UTFPR.

5 CONSIDERAÇÕES FINAIS

O desenvolvimento do jogo simulador de carros foi trabalhoso. Exigiu conhecimento de programação e estruturas de dados, física, matemática e até mesmo modelagem 3D.

Com esse projeto, é possível partir para um curso de pós-graduação na área de desenvolvimento de jogos, na área de simulação ou mesmo inteligência artificial. Outra possibilidade é a utilização educacional desse jogo, após melhorias e adição de funcionalidades pertinentes à simulação de trânsito.

5.1 TRABALHOS FUTUROS

Sem dúvidas existem muitas possibilidades de melhoramentos para esse *software*. Como trabalhos futuros têm-se a pesquisa sobre dinâmica dos veículos e o teste de técnicas de inteligência artificial para controle dos veículos autônomos.

Entre outras possibilidades estão: utilizar modelos físicos mais realistas, ou seja, desenvolver um trabalho para detalhar mais a dinâmica e a cinemática dos movimentos dos veículos; colocar à disposição do jogador a possibilidade de escolher o carro que ele deseja utilizar, podendo ser tanto carros que funcionam de forma diferente, quanto modelos 3D diferentes; inserir pessoas andando na cidade; expandir a cidade na qual o simulador é ambientado; e oferecer ao jogador a possibilidade de salvar o estado do jogo, para isso, seria preciso definir uma estrutura de dados capaz de salvar toda a informação de estado do jogo e adaptá-lo juntamente com todos os objetos para que possam se tornar persistentes.

Todo o código do programa está comentado em português e documentado, isso facilitará a realização de trabalhos futuros, mesmo que por outros desenvolvedores.

É possível transformar esse jogo em um simulador de tráfego, pois a classe controladora de semáforos foi parcialmente codificada, e o método para criação do modelo 3D dos semáforos foi finalizado. Dessa forma, com a finalização dessa classe, é possível usar o jogo para essa finalidade.

REFERÊNCIAS

ADOBE FLASH. Disponível em: <http://www.adobe.com/br/products/flash.html>. Acesso em: 5 out. 2011.

BARANAUSKAS, Maria Cecília Calani; ROCHA, Heloísa Vieira da. **Design e Avaliação de Interfaces Humano-Computador**. Campinas, SP. Editora Unicamp, 2003.

BINA, Diego Ribeiro. **Simulador de Boliche 3D com OpenGL**. 2010. 43f. Trabalho de Conclusão de Curso (Graduação) – Curso de Engenharia Eletrônica. Universidade Federal do Rio de Janeiro. Rio de Janeiro, 2010.

BETHKE, Erik. **Game Development and Production (Wordware Game Developer's Library)**. Editora Wordware Publishing, 2003.

BRAY, T.; PAOLI, J.; SPERBERG-MCQUEEN, C. M. **Extensible Markup Language (XML) 1.0**. W3C Recommendation, 1998. Disponível em: <http://www.w3.org/TR>. Acesso em: 16 jul. 2011.

CANTERI, Rafael dos Passos; KOSCIANSKI, André. **Desenvolvimento de um Simulador de Carro**. Seminário de Iniciação Científica e Tecnológica da UTFPR, Ponta Grossa, PR. SICITE 2011.

COHEN, Scott. **Zap: Rise and Fall of Atari**. New York, USA. Editora McGraw-Hill Osborne Media, 1987.

DÖLLNER, J.; BUCHHOLZ, H. **Continuous Level-of-Detail Modeling of Buildings in 3D City Models**. 13th annual ACM international workshop on Geographic information systems, 2005.

EBERT, David S.; MUSGRAVE, F. Kenton; PEACHY, Darwyn; PERLIN, Ken; WORLEY, Steven; **Texturing & Modeling - A Procedural Approach**. Morgan Kaufmann, 2003.

GARLAND, Michael; HECKBERT, Paul S. **Fast Polygonal Approximation of Terrains and Height Fields**. School of Computer Science Carnegie Mellon University Pittsburgh, 1995.

GIMP, the GNU Image Manipulation Program. Disponível em: <http://www.gimp.org>. Acesso em: 5 out. 2011.

GRAN-TURISMO.COM. Disponível em: <http://us.gran-turismo.com/us>. Acesso em: 4 set. 2011.

GREUTER, Stefan; PARKER, Jeremy; STEWART, Nigel; LEACH, Geoff. **Real-time Procedural Generation of 'Pseudo Infinite' Cities**. Proceedings of the First International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia, 2003.

HOUDINI. Disponível em: <http://www.sidefx.com>. Acesso em: 18 maio 2011.

ILANGOVAN, Praveen. **Procedural City Generator**. 2009. 37f. Dissertação (Mestrado em Computer Animation and Visual Effects), Bournemouth University, 2009.

IRRKLANG. Disponível em: <http://www.ambiera.com/irrklang/index.html>. Acesso em: 18 maio 2011.

IRRLICHT ENGINE. Disponível em: <http://irrlicht.sourceforge.net>. Acesso em: 10 jun. 2011.

KELLY, George; MCCABE, Hugh. **A Survey of Procedural Techniques for City Generation**. ITB Journal, 2006.

KELLY, George; MCCABE, Hugh. **Citygen: An Interactive System for Procedural City Generation**. Fifth International Conference on Game Design and Technology, 2007.

KOSCIANSKI, André. **Game Programming with Irrlicht**. Editora CreateSpace, 2011.

LARANJEIRA, Pablo; PORTO, Ed; ALVES, Lynn. **Garena e DOTA como plataformas digitais de comunicação para usuários de jogos eletrônicos on-line.** VII Seminário Jogos Eletrônicos, Educação e Comunicação. Fortaleza, CE. SJE, 2011.

MEIGS, Tom. **Ultimate Game Design: Building Game Worlds**. Editora McGraw-Hill Osborne Media, 2003.

MICHAEL, David. **Tile/Map-Based Game Techniques: Base Data Structures**. GameDev.net, 1999. Disponível em: <http://www.gamedev.net/page/resources/_/reference/programming/isometric-and-tile-based-games/298/tilemap-based-game-techniques-base-data-structures-r837> Acesso em: 12 abr. 2011.

OPENGL. Disponível em: <http://www.opengl.org>. Acesso em: 4 ago. 2011.

PATTERSON, David A.; HENESSY, John L. **Organização e Projeto de Computadores**. 3ª edição. São Paulo, SP. Editora Campus, 2005.

PENTON, Ron. **Data Structures for Game Programmers**. 1ª edição. Ohio, USA. Editora Premier Press, 2003.

PERUCIA, Alexandre; BERTHÊM, Antonio de; BERTSCHINGER, Guilherme; CASTRO, Roberto R. **Desenvolvimento de Jogos Eletrônicos: Teoria e Prática**. 2ª edição. São Paulo, SP. Editora Novatec, 2007.

POZZO, Luiz Gustavo; KOSCIANSKI, André. **Arquitetura de Jogos 3D**. Seminário de Iniciação Científica e Tecnológica da UTFPR, Cornélio Procópio, PR. SICITE 2010.

PRATSCHKE, Anja; PELIZZONI, Jorge Marques; ALUÍSIO, Sandra Maria. **Introdução à Inteligência Artificial**. Instituto de Ciências Matemáticas e de Computação. Disponível em: <http://www.icmc.sc.usp.br/~sandra/G9_t5/scripts.html>. Acesso em: 2 jun. 2011.

PRESSMAN, Roger S. **Engenharia de Software**. 6ª edição. Editora McGraw-Hill Osborne Media, 2006.

RICH, Elaine; KNIGHT, Kevin. **Inteligência Artificial**. Editora Makron Books. 2ª. Edição. São Paulo, 1994.

SHEFF, David. **Game Over: How Nintendo Conquered The World**. 1ª edição. Estados Unidos da América. Editora Vintage, 1994.

TATAI, Victor Kazuo. **Técnicas de Sistemas Inteligentes Aplicadas ao Desenvolvimento de Jogos de Computador**. 2003. 129f. Dissertação (Mestrado em Engenharia Elétrica) - Programa de Pós-graduação da Faculdade de Engenharia Elétrica e de Computação, Unicamp, 2003.

TENENBAUM, Aaron; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de Dados Usando C**. 1ª Edição. Editora Makron, 1995.

TOP NEWS. Disponível em: <http://www.topnews.in/usa/xbox-360-beats-nintendo-wii-sony-playstation3-sales-2367>. Acesso em: 2 set. 2011.

WINDOWS DIRECTX GRAPHICS DOCUMENTATION. Disponível em: <http://msdn.microsoft.com/en-us/library/ee663301%28v=VS.85%29.aspx>. Acesso em: 4 ago. 2011.

WELTER, Michael. **Geometry Clipmap Terrain Render System**. Hardware Rending Competition, 2005.

APÊNDICE A – Como Construir uma Cidade

Como Construir uma Cidade

Para se criar a cidade gerada por código do jogo utilizou-se os seguintes passos: criação e ajustes do terreno, e construção e ajustes dos prédios da cidade.

Primeiramente, gerou-se o terreno do cenário com o método mostrado na figura A1.

```
1  //! Adição do scene node do terreno
2      scene::ITerrainSceneNode *terrain = Gsmgr->addTerrainSceneNode((Gexepath + "\\medias\\terrain-heightmap.bmp"),
3                                              0,                                      //! parent node
4                                              -1,                                     //! node id
5                                              core::vector3df(0.f, 0.f, 0.f),         //! position
6                                              core::vector3df(0.f, 0.f, 0.f),         //! rotation
7                                              core::vector3df(4.f, 4.4f, 4.f),        //! scale
8                                              video::SColor (255, 255, 255, 255),     //! vertexColor
9                                              5,                                      //! maxLOD
10                                             scene::ETPS_17,                         //! patchSize
11                                             4                                       //! smoothFactor
12                                             );
13
14     terrain->setMaterialFlag    (video::EMF_LIGHTING, false);
15     terrain->setMaterialTexture (1, Gdriver->getTexture(Gexepath + "\\medias\\terraintexture.jpg"));
16     terrain->setMaterialTexture (0, Gdriver->getTexture(Gexepath + "\\medias\\terraintexture2.jpg"));
17     terrain->setMaterialType    (video::EMT_SOLID_2_LAYER);
18     terrain->scaleTexture       (30.0f, 50.0f);
```

Figura A1 - Código de Criação de Terreno
Fonte: Autoria Própria

Os parâmetros mais importantes do método *addTerrainSceneNode()* são: caminho do arquivo de Height Field, identificador do objeto de tela pai desse objeto (no caso de herança), identificador do terreno, posição 3D do terreno, rotação e escala.

Em seguida, são chamados os métodos que determinam a iluminação do terreno; a textura principal e a secundária; o tipo de material do terreno; e a escala das texturas, respectivamente.

Só após o terreno ter sido gerado é que são criados os prédios procedurais. Isso é feito através do método *createProceduralBuildings()* da classe CCity.

```
1  //! Criação dos prédios
2  void CCity::createProceduralBuildings () {
3
4      for (int i = 0; i < NBUILDINGS; i++)
5          for (int k = 0; k < NBUILDINGS; k++) {
6              p = Gsmgr->addCubeSceneNode(1);
7              if (p) {
8
9                  #define SCALE 90
10                 #define DISTANCE (SCALE+20)
11
12                 if (!(k + i))
13                     continue;
14                 p->setMaterialTexture   (0, Gdriver->getTexture(Gexepath + "\\medias\\building2.bmp"));
15                 p->setMaterialFlag      (video::EMF_LIGHTING, false);
16                 p->setScale             (core::vector3df(SCALE, SCALE, SCALE));
17                 p->setPosition          (core::vector3df((k * DISTANCE) + (SCALE / 2), (SCALE / 2), (i * DISTANCE) + (SCALE / 2)));
18             }
19         }
20 }
```

Figura A2 - Código do Método de Criação de Prédios
Fonte: Autoria Própria

O método apresentado na figura A2 contém duas estruturas de repetição que servem para colocar os prédios nas posições corretas, assim eles compõem os quarteirões da cidade.

Os cubos recebem textura de prédios, em seguida suas iluminações são desativadas, suas escalas recebem valores padrões e, por fim, suas posições são ajustadas de acordo com uma fórmula que distribui os prédios de forma padronizada.

APÊNDICE B – Tratando Curvas

Tratando Curvas

Foi explicado, durante o trabalho, que um dos estados dos carros autônomos é o estado eCURVE. Mas por que fazer um estado específico para as curvas do carro? O fato é que não é possível simplesmente rotacionar (no ângulo correto) o modelo 3D do veículo autônomo na situação de uma curva, pois se isso fosse feito, resultaria em um movimento anômalo: o carro seria girado instantaneamente e o resultado seria totalmente irreal.

Para resolver o problema da curva instantânea, desenvolveu-se um artifício. Em vez de esperar chegar ao cruzamento para tomar a decisão do seu destino, o NPC toma essa decisão um pouco antes, equivalente a 5 metros antes. Então, se o NPC precisar fazer uma curva, ele começa a realizá-la 5 metros antes, de forma suave e aos poucos. A velocidade do carro autônomo na hora da curva também é reduzida, dessa forma assemelhando-se ao que acontece no mundo real, onde os carros reduzem suas velocidades ao realizarem curvas.

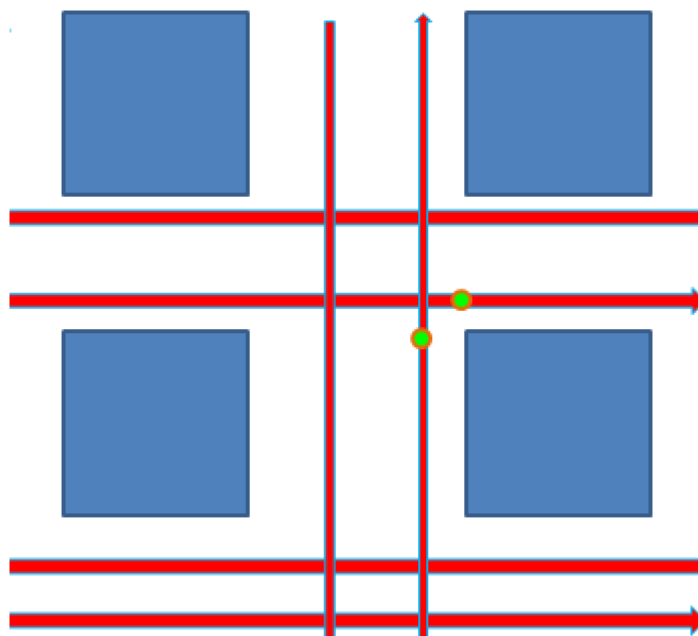


Figura A3 - Exemplos de Pontos de Decisão dos NPCs
Fonte: Autoria Própria

Os pontos verdes da figura A3 marcam as posições onde o NPC toma a sua decisão antes de chegar ao cruzamento. Deste modo, se for o caso de uma

curva, é traçado um arco ou semicírculo entre a posição atual do carro e a posição localizada a 5 metros de distância do cruzamento na via de destino, que é representado na figura pelo próximo ponto verde. Esse semicírculo criado é percorrido pelo carro, gerando uma curva suave e relativamente realista.