

Rodrigo Marques Almeida da Silva

GRIFO – GAME RESOURCES AND INTERACTION FRAMEWORK

Vitória – ES, Brasil

12 de Dezembro de 2007

Rodrigo Marques Almeida da Silva

GRIFO - GAME RESOURCES AND INTERACTION

FRAMEWORK

Monografia apresentada para obtenção do Grau de Engenheiro de Computação pela Universidade Federal do Espírito Santo.

Orientador:
Laércio Ferracioli

DEPARTAMENTO DE INFORMÁTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Vitória – ES, Brasil

19 de Dezembro de 2007

Monografia de Projeto Final de Graduação sob o título “*GRIFO – Game Resources and Interaction Framework*”, defendida por Rodrigo Marques Almeida da Silva e aprovada em 19 de Dezembro de 2007, em Vitória, Estado do Espírito Santo, pela banca examinadora constituída pelos membros:

Prof. Laércio Ferracioli, Ph. D.

Orientador

Prof. Hans-Jorg Andreas Schneebeli, D. Sc.

Examinador

Prof. Magnos Martinello, D. Sc.

Examinador

RESUMO

O mercado atual de desenvolvimento de jogos é um dos que mais movimenta investimentos em pesquisa, desenvolvimento e produção. Uma das principais subáreas de desenvolvimento de jogos é a de criação de motores – do inglês engines - para os mesmos.

Esse trabalho apresenta o projeto e implementação de um motor completo de jogos de Futebol 3D. Para tal fim, foram estudadas todas as áreas e subáreas da informática que interagem com o desenvolvimento do projeto. Dessa forma, a arquitetura do motor é apresentada e cada subsistema descrito detalhadamente, analisando-se cada item do funcionamento do mesmo.

Por fim, a projeto do jogo é apresentado e os resultados da utilização das várias técnicas e funcionalidades do mesmo são discutidas.

ABSTRACT

The current market of game development is one that most puts investments in production and research. One of the main sub-areas of game development is the games engines creation.

This work presents the project and implementation of a complete 3D soccer game engine. For such end, a study was made of all areas and sub-areas of the computer science that interact with the development of the project. For this, the architecture of the engine is presented and each subsystem traced, analyzing each item of the functionality of the engine.

Finally, the project of the game is presented and the results of the use of the several techniques and functionalities are argued.

DEDICATÓRIA

Dedico este trabalho primeiramente ao meu avô Francisco Antônio de Almeida, que sempre lutou pela felicidade de minha família, aos meus pais, Clésio Marques da Silva e Maria do Carmo Almeida Silva, e aos meus irmãos, Murilo e Filipe Marques Almeida da Silva, que sempre me incentivaram e ajudaram. Além desses, aos meus avós e a Deus, que sempre estiveram ao meu lado para iluminar meu caminho.

Semeia um pensamento e colherás um desejo; semeia um desejo e colherás a ação; semeia a ação e colherás um hábito; semeia o hábito e colherás o carácter.

(Tihamer Toth)

AGRADECIMENTOS

Agradeço a todos os meus amigos e colegas, principalmente à Cristina Klippel Dominicini, que sempre me ajudou, e aos meus tios e tias.

Aos meus amigos de Afonso Cláudio.

Aos Professores do DI e da Elétrica, em especial a Cláudia Boeres, Rosane, Cristina Vale, Thomas, Magnos, Arlindo, que me mostram a importância do estudo e da pesquisa.

Ao Prof. Hebert, pelos conselhos dados nas aulas de Aspectos Legais.

Ao Prof. Hans, pelos complicados trabalhos que me ensinaram muito.

Ao Prof. Dr. Laércio Ferracioli, que pelas críticas, elogios, oportunidades de desenvolvimento de projetos e aos incentivos a buscar coisas novas e a sua visão empreendedora.

Não poderia me esquecer da equipe maravilhosa do ModeLab, em especial o Msc. Thiéberson Gomes, que sempre me deu força.

E aos tantos amigos e familiares que não foram citados, mas que fizeram e fazem parte da minha história.

SUMÁRIO

Resumo.....	4
Abstract	5
Dedicatória	6
Agradecimentos	7
Sumário	8
Lista de Figuras.....	10
Lista de Tabelas	13
Lista de Equações	14
1 Introdução	15
1.1 Motivação.....	15
1.2 Objetivos e Metodologia.....	15
1.3 Revisão Bibliográfica	17
1.4 Estrutura da Monografia	18
2 História e Mercado de Jogos	19
2.1 História dos Jogos.....	19
2.1.1 O Império da Atari.....	22
2.1.2 A corrida dos videogames	24
2.2 Mercado de Jogos	26
2.2.1 TV Digital	27
2.2.2 Brasil	28
3 Arquitetura do Grifo.....	29
3.1 Especificação e Análise de Requisitos	29
3.2 Camadas e Arquitetura do Grifo	30
3.2.1 Descrição das Camadas e Arquiteturas	31
4 Desenvolvimento do Grifo	75

4.1	Visão Geral da OpenGL.....	75
4.2	Desenvolvimento das Camadas	76
4.2.1	Visão Geral	76
4.2.2	Resource Manager	79
4.2.3	Scene Manager.....	89
5	Desenvolvimento do Jogo Robotics Soccer.....	93
5.1	Contexto do jogo	93
5.1.1	StoryBoard.....	94
5.2	Regras do jogo	95
5.3	Mecanismos do jogo	96
5.4	Projeto Artístico	97
5.5	Projeto Técnico.....	105
5.6	Resultados	110
6	Conclusões e Trabalhos Futuros	113
	Trabalhos Futuros	114
	Anexos.....	115
	Anexo 1 - Arquivos de Configuração	115
	Anexo 2 - Programa de Carga e Instalador.....	117
	Glossário.....	118
	Referências Bibliográficas	122

LISTA DE FIGURAS

Figura 01: Modelo Cascata.....	16
Figura 02: Exemplo de Diagrama em UML para o Padrão Abstract Factory.....	16
Figura 03: Tennis Programming 1958, O primeiro vídeo game	20
Figura 04: Graetz, Kotok y Steve Russell e Imagem de Spacewar no original PDP-1.....	21
Figura 05: Logotipo da Atari.....	22
Figura 06: Tela do Jogo Pong.....	23
Figura 07: O famoso Atari 2600 VCS	24
Figura 08: Logotipo dop Grifo	29
Figura 09: BrainStorm do Grifo	30
Figura 10: Camadas do Grifo.....	31
Figura 11: Arquitetura de pacotes, Visão Superior	31
Figura 12: Arquitetura do Resource Manager	32
Figura 13: Arquitetura do Rendering System.....	33
Figura 14: Diagrama de Classes do Sistema de LoD Gráfico	35
Figura 15: Mapeamento de Textura 1D, 2D.....	36
Figura 16: a) Espaço de Mapeamento de textura. b) Replicação de textura.....	37
Figura 17: Coordenadas Esféricas	37
Figura 18: a) Método Clamp e Repeat. b) Método Nearest e Linear. c) Sem e Com Mipmaps..	38
Figura 19: a) Sphere Map. b) Cube Map.	39
Figura 20: Representação de Objetos 3D, Vértices e Arestas.....	40
Figura 21: Diagrama de Classes de um modelo 3D	40
Figura 22: Modelos de Reflexão a) Ambiente b) Difusa c) Especular	42
Figura 23: a) Luz Pontual. b) Luz Spot. c) Luz Direcional.....	45
Figura 24: Diagrama de Classes do Sistema de Luzes	45
Figura 25: a) Visualização da Câmera. b) Câmera inclinada a direita	46
Figura 26: Diagrama de Classes do sistema de Câmeras	46
Figura 27: a) Câmera de Foco. b) Câmera Walk.....	47
Figura 28: Ângulos de Rotação.....	47
Figura 29: Efeito Tilt na Câmera Walk.....	48
Figura 30: a) Modelo Original b) Modelo com Bump Mapping c) Textura do Modelo.....	48
Figura 31: Arquitetura de uma GPU	50
Figura 32: Sistema de Shaders	52
Figura 33: Arquitetura do Sistema de Física.....	53
Figura 34: Diagrama de Classes do ODE Solver.....	54
Figura 35: Modelo Físico e Modelo Gráfico	54

Figura 36: Diagrama de Classes dos Modelos Físicos.....	55
Figura 37: Momentos de Inércia de Objetos (HALLIDAY, et al., 2004).....	56
Figura 38: Colisão entre esferas	57
Figura 39: Colisão entre corpos rígidos	60
Figura 40: Atrito	62
Figura 41: Curva $C_A \times Re$	63
Figura 42: Separação da Camada Limite de uma Esfera	63
Figura 43: Separação da Camada Limite de Uma Bola Girando.....	65
Figura 44: Diagrama de Classes do Sistema de Som	66
Figura 45: Efeito Doppler	67
Figura 46: Diagrama de Classes do Sistema Matemático	68
Figura 47: Diagrama de Classes do Sistema de Scripts	70
Figura 48: Diagrama de Classes do Scene Manager.....	71
Figura 49: Diagrama de Classes do Tradutor	73
Figura 50: Árvore Vermelha e Preta.....	74
Figura 51: Diagrama de Atividades do Grifo	77
Figura 52: Diagrama de Seqüência do Grifo.....	78
Figura 53: Diagrama de Fluxo do sistema de renderização	79
Figura 54: Diagrama de Seqüência do Compilador de Shaders	80
Figura 55: Diagrama de Fluxo do Sistema de Física	84
Figura 56: Formato do Arquivo WAV	85
Figura 57: Funcionamento do Streaming	86
Figura 58: Teste de pertinência dos widgets.....	89
Figura 59: Transformação de sistemas de coordenadas.....	89
Figura 60: Máquina de Estados do Botão	90
Figura 61: Máquina de Estados da Caixa de Texto.....	91
Figura 62: Máquina de Estados do Modo Edição do EditBox	92
Figura 63: StoryBoard do Jogo Lohan.....	94
Figura 64: StoryBoard do RoS.....	95
Figura 65: Lista de Comandos e Teclas do RoS	97
Figura 66: a) Esboço da personagem jogador b) Modelo do Jogador.....	98
Figura 67: Modelagem da Personagem	98
Figura 68: Grafo do modelo hierárquico da personagem	99
Figura 69: a) Esboço dos ângulos de rotação α, β b) Ângulo α c) Ângulo β	100
Figura 70: Linhas do Campo	100
Figura 71: Dimensões Oficiais do Campo	101
Figura 72: a) Esboço do cenário b) Modelo do cenário	101
Figura 73: a) Modelo em construção do cenário b) Modelo em construção do cenário com luzes.....	102
Figura 74: Texturas da camisa do BotFogo e do FlaBot	102
Figura 75: Textura do resto e da bola	103
Figura 76: Posicionamento dos jogadores	103
Figura 77: Máquina de Estados das Cenas	104
Figura 78: Diagrama de Classes do RoS.....	105
Figura 79: Tipos de chutes.....	108

Figura 80: a) Efeito com o Vertex Shader b) Efeito com o Pixel Shader	109
Figura 81: Efeitos de Motion Blur e Fog.....	109
Figura 82: Mapa do jogo	110
Figura 83: Arquivo de Configuração	115
Figura 84: Arquivo de configuração do time do flamengo.....	116
Figura 86: Telas de configuração.....	117
Figura 85: Tela principal do programa de carga.....	117

LISTA DE TABELAS

Tabela 01: Renderização de Modelo 3D	41
Tabela 02: Exemplo de conteúdo de arquivo OBJ que contém a descrição de uma taca	44
Tabela 03: Exemplo de arquivo MTL (taca2.mtl),correspondente ao arquivo OBJ da Tabela 1.	44
Tabela 04: Algoritmo de Teste de Colisão em Retângulos.....	57
Tabela 05: Colisão entre Retângulos.....	58
Tabela 06: Tabela de Termos de Tradução	72
Tabela 07: Algoritmo de Verificação de Colisão de Esferas e de Impulso	81
Tabela 08: Algoritmo de Resolução de Colisão	82
Tabela 09: Algoritmo de Atualização (EDO)	83
Tabela 10: Formatos de dados suportados pelo motor.....	84
Tabela 11: Exemplo de chamada de variável LUA	87
Tabela 12: Exemplo de chamada de função LUA	88
Tabela 13: Tabela de decisão do tipo de falta.....	96
Tabela 14: Cenas do jogo	104
Tabela 15: Transição da FSM dos Zagueiros	106
Tabela 16: Ações dos estados dos zagueiros	106
Tabela 17: Funcionamento do script de IA	107
Tabela 18: Tabela de funções de IA do RoS	108
Tabela 19: Tabela de Imagens dos Modelos do RoS	111
Tabela 20: Imagens do Jogo em Funcionamento.....	112

LISTA DE EQUAÇÕES

Equação 01: Coordenadas Esférica para cartesianas.....	37
Equação 02: Coordenadas de textura de uma esfera em modo cartesiano.....	38
Equação 03: Coordenadas de textura de uma esfera	38
Equação 04: Matriz de rotação em torno de um eixo qualquer	48
Equação 05: Momento de Inércia Linear e Angular.....	58
Equação 06: Força e Torque médio durante um impulso	59
Equação 07: Conservação do Momento Linear	59
Equação 08: Energia Cinética Linear e Angular.....	59
Equação 09: Coeficiente de Restituição	60
Equação 10: Velocidade do ponto de colisão	60
Equação 11: Movimento Linear dos Corpos (1) e (2).....	60
Equação 12: Momento de Inércia Angular dos Corpos (1) e (2)	61
Equação 13: Magnitude do Impulso da Colisão	61
Equação 14: Velocidades Lineares e Angulares após colisão.....	61
Equação 15: Força de Arrasto Aerodinâmico.....	62
Equação 16: Número de Reynolds	62
Equação 17: Forma decomposta da força de Arrasto Aerodinâmico	64
Equação 18: Forma Vetorial da força de Arrasto Aerodinâmica.....	64
Equação 19: Formulação Analítica do Coeficiente de Arrasto	64
Equação 20: Velocidade Terminal.....	64
Equação 21: Força Magnus	65
Equação 22: Ganho de uma fonte dependente da distância.....	67
Equação 23: Efeito Doppler.....	68
Equação 24: Transformação de sistemas de coordenadas	90
Equação 25: Cálculo do Fator de Colisão	96

CAPÍTULO I

INTRODUÇÃO

Este trabalho destina-se a elucidar a integração da informática com os jogos, mostrando os vários aspectos do desenvolvimento de softwares e games.

1.1 Motivação

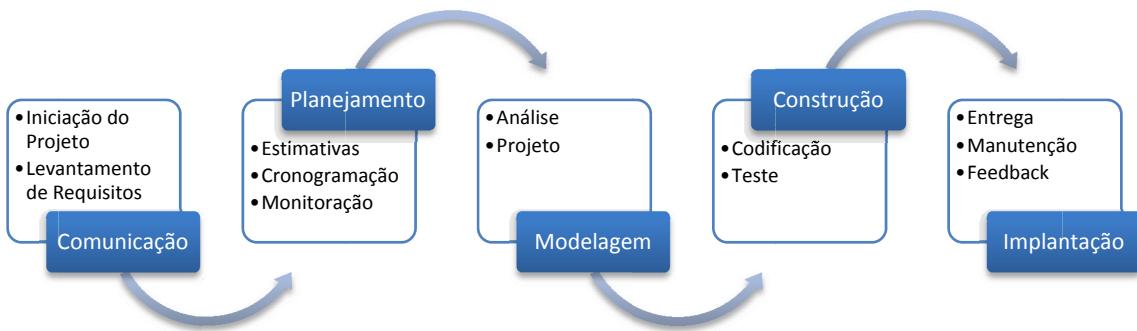
A principal motivação do trabalho foi desenvolver um sistema que contemplasse o maior número de áreas da computação, de forma a ser um projeto que “resumisse” todo o conhecimento aprendido durante o curso de Engenharia de Computação. Além disso, apesar da atividade de criação de jogos ser uma tarefa difícil, é, acima de tudo, muito empolgante.

1.2 Objetivos e Metodologia

Este trabalho tem como escopo, apresentar todo o projeto e desenvolvimento do *Grifo*, focando nos detalhes relevantes de cada subárea da computação envolvidas no processo. Contudo, apesar de grande importância, muitos passos da engenharia de software não serão amplamente discutidos, apenas mencionados, pois tais assuntos delongariam a discussão do texto, ofuscando áreas prioritárias de interesse.

A metodologia utilizada é o *modelo em cascata* (ROYCE, 1970). A escolha de tal modelo deve ao fato de haver apenas um membro na equipe desenvolvedora e também às poucas mudanças necessárias ao projeto. Para o autor, um modelo mais adequado para o tipo de desenvolvimento seria o *modelo RAD* ou de *prototipagem*, esse sendo largamente utilizado no mercado de jogos (AZEVEDO, 2005).

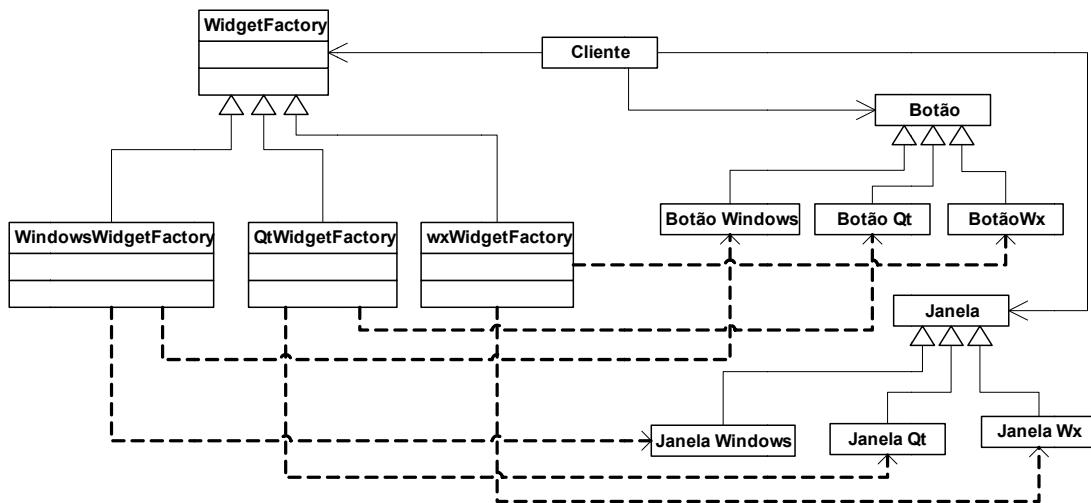
A Figura 01 mostra a seqüência de passos utilizados no modelo cascata.

**Figura 01:** Modelo Cascata

O modelo cascata, também chamado de *ciclo de vida clássico*, fornece uma abordagem sistemática e seqüencial do desenvolvimento de softwares, tendo início no levantamento de requisitos pelo cliente até a implantação. Esse tipo de modelo não é conveniente para o desenvolvimento de um jogo comercial, *serious game*, o qual deve se adequar tanto às mudanças funcionais, de requisitos, quanto às tecnológicas.

Para a idealização do projeto utilizou-se como base o padrão criacional de projeto *Abstract Factory* (KUCHANA, 2004). Esse padrão permite a criação de famílias de objetos relacionados ou dependentes, através de uma única interface e sem que a classe concreta seja especificada.

O Abstract Factory é muito utilizado na criação de *toolkits* para interfaces gráficas. Um exemplo de utilização é o da biblioteca de Widgets¹, QT. A Figura 02 mostra um exemplo de modelagem de um sistema baseado nessa metodologia.

**Figura 02:** Exemplo de Diagrama em UML para o Padrão Abstract Factory

¹ Elementos de Interface Gráfica: Botões, Caixas de Texto e outros.

Essa abordagem permite a criação de sistemas bastante gerais e com grande capacidade de suporte a modificação.

1.3 Revisão Bibliográfica

No início do projeto, foi feita uma pesquisa bibliográfica sobre o problema e verificou-se a existência de um amplo material para estudo. Esta seção apresenta brevemente os trabalhos que foram considerados mais relevantes.

Para o desenvolvimento de um motor de jogos, mesmo que com poucas funcionalidades, uma ampla quantidade de informação deve ser coletada, pois, esses sistemas possuem, naturalmente, uma abordagem multidisciplinar.

FEIJÓ, et al. (2006) apresentam uma descrição dos principais componentes de um motor de jogos 3D, mostrando a importância de cada componente. AZEVEDO (2005) apresenta o processo de desenvolvimento de jogos, expondo a atuação de um motor de jogos no processo.

Para a criação dos sistemas gráficos, a bibliografia consultada foi extensa, mas a fundamentação teórica necessária para o entendimento dos processos de renderização² e rasterização³ de imagens foi baseada em FOLEY, et al. (1997). COHEN, et al. (2006) colaborou enormemente para a implementação do motor, mostrando de forma bastante didática a forma de utilização das bibliotecas do *OpenGL*.

Para a simulação da física e dos sistemas matemáticos, os principais livros consultados foram HALLIDAY, et al. (2004), que mostra os fenômenos pela visão física, e BOURG (2002) que analisa a implementação lógica dos mesmos. O artigo de AGUIAR, et al. (2004) auxiliou na criação e entendimento dos conceitos aerodinâmicos para jogos de futebol.

Usando os princípios do manual Creative Technology (2005), foi possível construir o sistema de som, permitindo que as funcionalidades de sonorização 3D fossem adicionais no motor. Outras fontes de suma importância foram JUNG, et al. (2006) e BUCKLAND (2005), que permitiram o correto entendimento da linguagem LUA e as técnicas de IA para jogos de computador.

Como a construção de qualquer software necessita de uma padronização e boas especificações, o livro de PRESSMAN (2005) mostra detalhadamente todo o processo de software. Sem o uso da engenharia de software a qualidade e o tempo de desenvolvimento

² Desenho de um modelo 3D

³ Conversão de uma definição geométrica para pixels

seriam muito prejudicados. Além disso, o livro de SALEM, et al. (2004) foi utilizado para delinear o processo de projeto e produção de jogos.

Além dessas referências citadas, muitas outras foram utilizadas para a produção desse motor, tal como CORMEN, et al. (2002) de onde algoritmos foram extraídos. As demais referências são citadas ao longo do texto.

1.4 Estrutura da Monografia

O restante deste trabalho está dividido em quatro capítulos:

- O capítulo 2 resgata o contexto da história e do mercado atual de jogos de computador, abordando os fatos mais relevantes desse processo e desmistificando o desenvolvimento dessa nova área de pesquisa.
- O capítulo 3 descreve toda a arquitetura do Grifo, mostrando os detalhes de cada camada e subcamada do sistema. Além disso, são apresentados os conceitos de computação gráfica, simulação em tempo real de física e tecnologias gráficas. Uma seção especial foi escrita para apresentar as arquiteturas atuais de programação de Shaders e *GPGPU - General-Purpose Computing on Graphics Processing Units*.
- O capítulo 4 mostra os algoritmos, as considerações e sistemas externos utilizados no desenvolvimento do motor, além de exibir todo o fluxo de execução do mesmo.
- O capítulo 5 é a apresentação do projeto e desenvolvimento do jogo RoS - Robotics Soccer, um jogo de futebol de robôs, feito com o auxílio do motor Grifo. Nesse capítulo são mostrados desde os detalhes de design até os detalhes de inteligência desenvolvidos.
- Finalmente são apresentadas as considerações finais, conclusões e trabalhos futuros.

CAPÍTULO II

HISTÓRIA E MERCADO DE JOGOS

Para entendermos a situação atual e as perspectivas futuras da área de jogos e entretenimento digital, é necessário entendermos a história desse mercado e aprendermos com os erros e com os acertos dos autores de outrora, fazendo da experiência de outros uma lição de vida.

Em 1931, Aldous Huxley fez uma audaciosa previsão do futuro nas páginas do famoso *Admirável Mundo Novo*. Analisando o desenvolvimento de interfaces 3D para plataformas virtuais, o conto Science Fiction dos irmãos Wachowski é o mais próximo da visão do futuro para quem entende a magnitude do mercado dos games dos dias de hoje.

Esse é um setor dinâmico, onde a cada jornal publicado ou newsletter recebida, os paradigmas são alterados e, novos modelos emergentes podem mudar toda a perspectiva do desenvolvedor. Portanto, a contínua leitura desse tipo de texto permite a compreensão das mudanças curto e médio prazo.

2.1 História dos Jogos

Ao fim da Segunda Guerra Mundial, o planeta viu-se dividido e bipolar, restando sobre a face da Terra duas potências fortalecidas a ponto de serem chamadas Superpotências. A corrida tecnológica, impulsionada pela divisão, levou o mundo a um tempo de evolução sem precedentes, com inovações surgindo em todos os campos do conhecimento, já que os embates também se davam no campo cultural. Muitas das revoluções tecnológicas militares sofreram extensões na população civil e trouxeram benefícios dos quais desfrutamos até hoje, entre eles o vídeo game.

Sobre a origem dos *games*, dois laboratórios norte-americanos reivindicam o título, contudo, depois de muitas discussões definiu-se, o ano de 1958 e o estado de Nova York como data e

local dos mesmos. O principal responsável por essas máquinas seria o físico Willy Higinbotham (1910 - 1995).



Figura 03: Tennis Programming 1958, O primeiro vídeo game

Willy, depois de sua passagem pelo Projeto Manhattan, trabalhou no Brookhaven National Laboratories, o qual possuía poderosos computadores analógicos. Com o objetivo de criar uma maneira de entreter o público, Willy, ajudado por Rovert Dvorak, criou um jogo de tênis bastante simples, que era mostrado em um osciloscópio e processado pelas máquinas do laboratório conforme mostra a Figura 03.

Em 1995 o físico faleceu sem nunca ter lucrado com sua invenção, apesar de seu investimento em melhorias e no aperfeiçoamento do "Tennis Programming", que depois de ter sido adaptada para um monitor de 15 polegadas teria seu nome alterado para "Tennis for Two".

Em 1965, um executivo da Intel, Gordon Moore (1929-1965) previu que a capacidade de um chip de computador dobraria anualmente nas décadas seguintes, já em 75 a previsão se mostrou válida e ele voltou a projetar que a capacidade duplicaria a cada dois anos, num efeito que ainda se mantém até hoje. Essa previsão é conhecida pelos engenheiros como a "Lei de Moore", e uma de suas consequências foram os avanços na área de jogos.

Quatro anos depois dos acontecimentos do Brookhaven, uma nova equipe retomou as pesquisas que dariam origem ao mundo virtual dos dias de hoje. Dessa vez, nas salas do Massachusetts Institute of Technology (MIT).

Utilizando um computador modelo DEC PDP-1, a equipe liderada pelo designer Stephen Russell trouxe à vida um sonho adormecido desde o descobrimento do "Tennis for Two". Esses

acadêmicos são reconhecidos como os "verdadeiros pais do videogame". O PDP-1 era uma maravilha tecnológica da sua época, com memória de 4 Kbytes e processador de 18 bits, era composto por cartões perfurados, um monitor e uma caneta ótica e custava a fortuna de US\$120 mil. Em vez de válvulas, o PDP-1 usava transistores, o que permitia ser ligado instantaneamente, ao contrário dos computadores valvulados que demoravam horas para aquecerem.

Em 30 de julho de 1961, Russell e sua equipe testavam o "Spacewar!", desenvolvido em linguagem "Assembly" de PDP-1. O jogo, que utilizava metade da capacidade do PDP-1 (2 KB), foi inspirado nos livros de ficção científica do americano E.E. "Doc" Smith e recriava uma batalha entre duas naves.

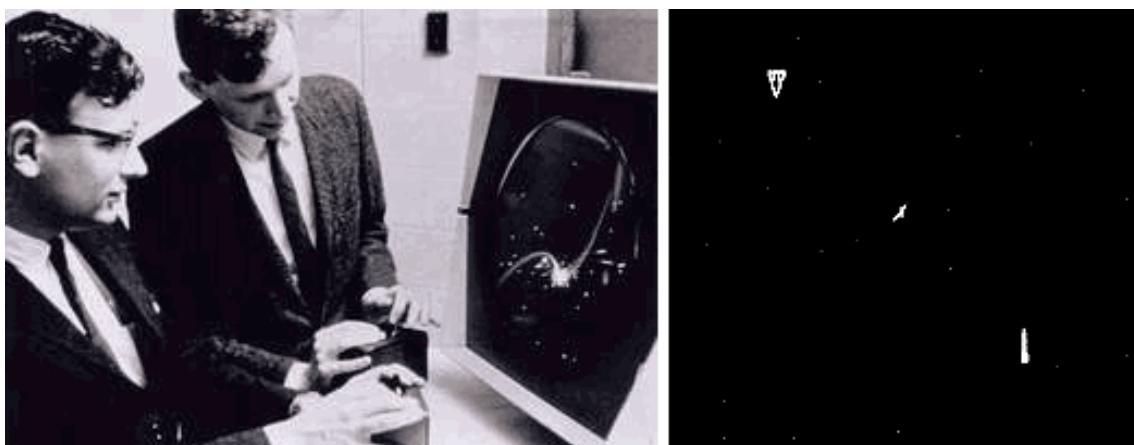


Figura 04: Graetz, Kotok y Steve Russell e Imagem de Spacewar no original PDP-1.

O piloto tinha como obstáculo uma estrela no meio do campo de batalha, a qual gerava um campo gravitacional. Dependendo da habilidade do jogador, esse campo poderia ajudar ou atrapalhar. O campo possuía ainda inércia simulada e, assumindo o controle da nave, era preciso movimentar-se em diferentes direções, tentando alvejar o inimigo com torpedos. Apesar de simples, o jogo surpreendia pelo uso realista das propriedades da Física. Novamente os criadores subestimaram o valor da sua criação e não ganharam um centavo pela sua obra. Apenas serviram de inspiração para que outro engenheiro construísse um império. A idéia novamente mudaria de mãos rumo ao definitivo sucesso comercial.

Em 1968 surge o que podemos dizer de o primeiro console de jogos. A história desse importante invento remonta à Alemanha Oriental e começa em 1922 quando Ralph Baer nasceu. Devido à forte pressão nazista de Hitler, em 1938 o jovem tentou escapar do início da Segunda Guerra Mundial mudando-se para os Estados Unidos. Três anos depois, em 7 de

dezembro de 1941, os Estados Unidos, após sofrerem o ataque japonês em Pearl Harbor, entraram no conflito e levaram Ralph com eles.

Após regressar da guerra, Ralph formou-se em Engenharia de Televisão (*Television Engineering*), e trabalhou em algumas empresas de Eletrônica e de telecomunicações até ser convidado a trabalhar na Sander Associates em 1966. Em 1967, nasce o "chasing game", um jogo no qual dois quadrados moviam-se pela tela controlados pelo usuário, posteriormente intitulado "Brown Box".

Em 1968 Ralph Baer patenteia o protótipo *Brown Box*, que já rodava alguns jogos de ping-pong, futebol, voleibol e tiros. Em 1969, após uma corrida por financiamento para seu projeto, Ralph consegue apoio junto à subsidiária da Holandesa Philips, a Magnavox, sendo esse acordo fechado em Março de 1971.

Em Maio de 1972 é lançada comercialmente o console *Magnavox Odyssey* que teve origem do protótipo *Brown Box*. O console⁴ era ligado à televisão, e inicialmente foram lançados 12 títulos, além de ser possível usar uma espingarda em alguns jogos. Devido ao baixo processamento gráfico, os utilizadores precisavam de colocar cartões de plásticos na tela da televisão para simular o campo de jogo: por exemplo, num jogo de tênis era preciso colocar um cartão verde para simular o campo. O único jogo que não usava os cartões de plástico era o *Table Tennis*.



Figura 05: Logotipo da Atari

2.1.1 O Império da Atari

O nome *Atari* vem do japonês, que é o grito que os jogadores de "Go" dão ao encurralarem seus adversários durante uma partida e que é equivalente ao "Cheque Mate!", do Xadrez. Devido, talvez, a complexidade de suas regras, o "Go" nunca se popularizou fora do Japão e o nome Atari tornou-se sinônimo de Games.

Em 1962, Nolan Bushnell, um estudante de Engenharia Eletrônica da universidade de Utah, em visita aos laboratórios do MIT, faz um contato que jamais iria esquecer, ele conhece o *Spacewar*.

Durante os anos que se seguiram à sua passagem pela faculdade, Nolan tentou aperfeiçoar o *Spacewar*. Contudo, o jogo demandava poderosos mainframes para rodar e, além disso, era um jogo muito complexo.

⁴ Videogame.

Após abandonar a companhia onde trabalhava, com um investimento de 250 dólares e uma grande pretensão na cabeça, Nolan funda sua própria empresa. Contam as lendas que o nome escolhido para o projeto seria *Syzygy*, além de muito difícil de ser pronunciado, já estava registrado para uma companhia que trabalhava com telhados. Assim, em 1972, a Atari foi fundada. A inspiração japonesa não se ateve apenas ao nome, o logotipo escolhido foi inspirado numa estilização do Monte Fuji.

Em Maio de 1972, Nolan visita o *The Magnavox Profit Caravan* onde assina o livro de convidados para assistir a uma demonstração do Magnavox Odyssey e joga o *Odyssey Table Tennis*.

Mais tarde, contando com a parceria do colega Al Alcorn, a Atari partiu para a estruturação de seu primeiro produto, chamado simplesmente de "Pong", que era uma versão arcade⁵, da máquina de jogo de moedas do Table Tennis.

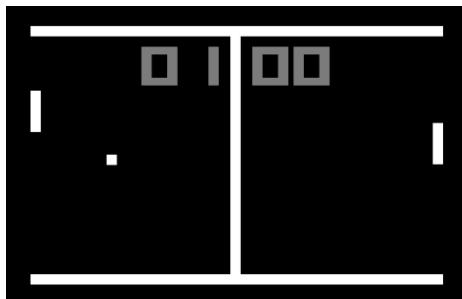


Figura 06: Tela do Jogo Pong

Basicamente, a jogabilidade do Pong era constituída por dois traços nas laterais da tela e um ponto, que representava uma bola. O jogador deveria rebater a bola movimentando as pretensas raquetes para cima ou para baixo, vide Figura 06. O jogo foi testado num

modelo Fliperama (Arcade), em um bar chamado Andy Capp's. Já na estréia conseguiu-se medir o sucesso que o projeto teria pelo excedente de fichas depositadas na máquina, foram tantas que elas transbordaram invadindo os circuitos e geraram o primeiro problema da Atari. A máquina, chamada pelos criadores de Computer Space, ganhou ainda o status de primeiro Arcade.

Movido pelo sucesso do Pong, Nolan lançou a idéia de criar uma versão doméstica, o *Home Pong*. Ele sabia que a simplicidade do brinquedo, que não tinha a pretensão de seu antecessor Odyssey, poderia ser um diferencial, já que a simplicidade do projeto poderia estar refletida em preços populares. Em 1974 a idéia de criar um sistema caseiro que rodasse o "Pong" saiu das pranchetas por meio de uma parceria estratégica com a cadeia de lojas Sears e vendeu 150.000 unidades consolidando a *Pongmania* por todo o país. A sinergia dos mercados ocidentais e orientais se fortalecia e, já em 1973, a Atari estabelece um importante contato com a Nanco do Japão.

⁵ Arcade é um videogame profissional usado em estabelecimentos de entretenimento, por vezes também chamado no Brasil de fliperama.

2.1.2 A corrida dos videogames

Deu-se início a uma corrida no mercado e nos anos que se seguiram. Dezenas de empresas lançaram produtos similares para concorrer nesse novo mercado que surgia. A melhoria era contínua e as empresas incorporavam novas tecnologias a cada novo lançamento de consoles entre 1975 e 1979.

Em 1977 a Atari lança o Atari 2600 VCS (Video Computer System) que se torna o console doméstico mais popular daqueles tempos. Os seus 128 bytes de memória e 1.19 Mhz de velocidade do processador, e uma placa de vídeo, marcavam uma nova geração de consoles domésticos.

Empresas produtoras de jogos como a Sega, Konami entre outras viram no Atari 2600 VCS uma oportunidade e desenvolveram alguns títulos para o mesmo.



Figura 07: O famoso Atari 2600 VCS

Sete anos depois do lançamento o console Atari 2600 VCS naufragou devido à grande quantidade de títulos ruins, arrastando todo o mercado de consoles, esse acontecimento histórico ficou conhecido como o crash dos videogames de 1984.

Em 1983 começam a surgir os consoles de 8 bits, formando a terceira geração de consoles de videogames. Embora a geração anterior usasse também processadores de 8 bits, foi nesta altura que os consoles domésticos foram rotulados pelos seus bits.

Nesta geração que se deu a primeira “guerra” entre a Nintendo com o Nintendo Entertainment System (NES), popular nintendinho, e a Sega com o Master System. Na América do Norte e Japão a Nintendo levou a melhor enquanto que na Europa e Brasil o console da Sega era a mais popular.

Em 1983, a Nintendo lança no Japão o console Famicom (Family Computer), começando assim o domínio japonês da indústria de videogames. Com receio da concorrência da Atari a Nintendo tenta vender a comercialização de seu console, perante a recusa, a Nintendo introduz em 1985 pelos seus próprios meios o console no mercado americano com o nome de NES (Nintendo Entertainment System) e com um novo aspecto mais parecido com um computador do que com um brinquedo como acontecia com a Famicom.

Nos tempos iniciais da comercialização do NES a Nintendo lança dois acessórios “revolucionários” a Power Glove que permite ao jogador controlar o jogo movendo o braço e dedos, e o Robotic Operating Buddy um robô que jogava.

Em 1986, a gigante japonesa dos flipers Sega, decide entrar no mercado dos videogames e em 1984 lança o Mark III que serviu de base ao Master System que chegou aos EUA em 1986, ano em que 90% do mercado americano era detido pelo NES e os restantes 10% pelo Atari 7800 ProSystem e o Intellivision. Apesar do insucesso do Master System, a Sega foi inovadora ao lançar para a Master System óculos 3D que permitiam obter a sensação de profundidade nos jogos que o usavam. Os óculos foram impopulares devido à falta de bons jogos compatíveis, e ao cansaço que eles causavam à vista.

Em 1988, atrás do NES/Famicom da Nintendo, a Sega lança o Mega Drive, o primeiro console de 16 bits. Com um aspecto futurista, um processador de 16 bits o Motorola 68000 que rodava a 7.67 Mhz e tinha capacidade para exibir 64 cores em simultâneo, conseguiu impressionar os jogadores com os seus gráficos. Nos EUA ele foi lançado em 1989 com o nome de Genesis. Um destaque de jogo produzido para o Mega Drive foi a série Sonic.

Com o lançamento do Mega Drive/Genesis, as perspectivas para o NES da Nintendo não eram as melhores. Por isso, em 1990, a Nintendo lança seu console de 16 bits o Super Famicom, e em 1991, lança nos EUA com o nome de Super Nintendo Entertainment System (SNES).

Tempos depois, a Nintendo tinha contratado a Sony para desenvolver o SNES CD, mas cancelou o contrato ficando a Sony com a tecnologia desenvolvida. Desse deslize da Nintendo nasce o Playstation.

Em 1994 é lançado no Japão o Playstation sendo um sucesso imediato. Com um processador gráfico (*Graphics Processing Unit*) com capacidade para desenhar mais de meio milhão de superfícies triangulares planas por segundo, a Sony tenta assegurar jogos que condizem ao investimento, e compra a produtora de jogos britânica Psygnosis, que tinha começado a ser conhecida com o jogo Lemmings.

Após um grande sucesso, em 2000 o console Playstation 2 é lançado no mercado Japonês e Americano. Apoiado pela maioria das produtoras de jogos, e conseguindo com muitas produtoras exclusividade absoluta ou exclusividade de títulos chave e mantendo compatibilidade dos jogos do primeiro Playstation, o Playstation 2 torna-se muito popular.

Em 2001, A Microsoft faz o lançamento do XBOX e marca a entrada da gigante da informática no mercado de consoles.

Em 2005 é lançado nos EUA o XBOX 360, o segundo console da Microsoft. O fato de o lançamento ser quase simultâneo para todo o mundo é uma novidade no mundo dos consoles. O XBOX 360 é fruto da cooperação entre a IBM, ATI, Samsung e SiS e marca o início da geração atual de consoles.

O XBOX 360 utiliza um drive de DVDs que permite ver filmes em DVD, CDs de música, CDs com MP3. Ele pode ser ligado em rede com computares com Windows XP através da porta USB ou por rede sem fios Wi-Fi, comportando-se como uma estação para reproduzir os dados armazenados no computador, câmera digital ou leitor de MP3.

Em 2006, a Nintendo lança o Nintendo Wii. Apesar da Nintendo anunciar que o seu público alvo é diferente, inevitavelmente ela entra em competição com o XBOX 360 e com o Playstation 3.

Uma característica que distingue o Wii de qualquer outro console, até o momento, é seu controle sem fios BlueTooth, que pode ser usado como um dispositivo de mão para apontar, capaz de detectar movimentos e rotações em três dimensões e capaz de emitir som e vibrar.

Ainda em 2006, a Sony lança o Playstation 3, conhecida também por PS3. Além de uma elevada capacidade de processamento, o console dispõe de disco rígido. O PS3 utiliza a tecnologia Blu-Ray que pode armazenar seis vezes mais que um DVD.

2.2 Mercado de Jogos

As perspectivas de crescimento Indústria do Entretenimento Interativo apontam para um enorme crescimento do setor, sendo as previsões para esse setor são as melhores possíveis (AZEVEDO, 2005). Em 2003, a indústria dos games faturou US\$31 bilhões em todo o mundo. Comparativamente, a indústria dos videogames faturou, em 2001, quase três vezes mais que a indústria do cinema de Hollywood: foram US\$21 bilhões contra US\$8,4 bilhões.

Os US\$31 bilhões dessa indústria é um número difícil de ser calculado pela pluralidade de subprodutos, pela forte presença da indústria da pirataria, que não paga impostos, não participa de pesquisas e nem contabiliza seus lucros.

Atualmente os grandes mercados consumidores e produtores de videogames estão concentrados nos Estados Unidos, Europa e Ásia. Cada mercado têm particularidades culturais.

Enquanto o fator preço é bastante relevante no processo de seleção de um usuário de jogos no Brasil, os alemães são grandes consumidores de jogos de administração e gerenciamento como Sim City, Civilization e outros ainda mais complexos. Enquanto o Brasil e a Coréia do Sul são grandes jogadores on-line, em Lan Houses e Ciber Cafés, os norte-americanos são adoradores de jogos de esportes e os japoneses preferem as histórias mais sofisticadas dos RPGs.

A principal associação da indústria de games, a Digital Software Association, apresentou na maior feira do mundo desse setor, E3: Eletronic Entertainment Expo, os impactantes resultados de uma pesquisa nacional realizada em 2003.

Na análise feita, pelo terceiro ano consecutivo, 35% dos norte-americanos, uma população de 280 milhões de pessoas, afirmaram terem nos jogos eletrônicos por computador sua principal fonte de lazer. De acordo com uma pesquisa divulgada pela ESA (Entertainment Software Association, 2007) a média de idade do norte-americano que gosta de jogar é de 29 anos, enquanto a faixa etária do comprador de jogos gira em torno de 36 anos.

Dessa pesquisa, sessenta por cento dos norte-americanos (168 milhões de pessoas) afirmaram jogar video games. Dos que jogam em computadores, 28% têm menos de 18 anos de idade, 30% têm entre 18 e 35 anos de idade e 42% têm mais de 35 anos de idade. Dos que jogam em aparelhos de video game, 42% têm menos de 18 anos de idade, 37% têm entre 18 e 35 anos de idade e 21 % têm mais de 35 anos de idade. As pesquisas apontaram ainda que 61% dos jogadores norte-americanos são homens e 39% mulheres.

2.2.1 TV Digital

Um ramo emergente no mercado é o de jogos para a TV Digital. No caso específico das aplicações que envolvem qualquer tipo de risco financeiro, como comércio eletrônico, e-banking, pagamento de serviço de acesso a conteúdos ou jogos on-line, a TV Digital pode ser a esperança para a adoção do brasileiro que ainda desconfia da segurança na Internet.

A TV digital traz não apenas melhora na qualidade de recepção de imagem e som. Com ela, surgem novas aplicações fundamentadas na alta velocidade de transmissão de dados, no aumento dos canais disponíveis e na possibilidade de interação. A combinação de TV digital com as tecnologias de internet permite a seleção de programação, acesso à Web, o *t-commerce* (shopping on-line por TV) e o *t-banking* (t de television).

Além dos celulares, a base de aparelhos de entretenimento eletrônico, excluindo os computadores pessoais dessa contagem, crescerá de 415 milhões em 2004 para 2,6 bilhões

em 2010. Assim como a introdução de novas plataformas dentro do mercado, como tocadores de MP3, PDAs, brinquedos infantis e até mesmo aparelhos de ginástica. Em 2010, um bilhão de pessoas terão acesso a telefones móveis capazes de rodar conteúdo multimídia como filmes e jogos.

2.2.2 Brasil

Em 1996, o mercado de Games oficial no Brasil gerava US\$ 250 milhões. Naquele ano ficou conhecida no mercado a previsão de um executivo da área que especulava dentro de uma perspectiva bastante realista para os números da época, que o setor iria bater a casa do bilhão de dólar em 2005. Esse é o valor médio da indústria de países como Espanha ou Canadá. Atualmente faturamos menos de US\$ 50 milhões, (AbraGames, 2004), com o mercado legal de jogos, um explícito retrocesso que nos devolveu à periferia do globo em relação a essas tecnologias.

As empresas instaladas aqui e que investiram no mercado nacional não ganharam dinheiro e os comerciantes regulares que cumprem os seus compromissos legais e tributários também não obtiveram lucro. Por consequência, o governo não ganhou quase nada em impostos.

Acabamos nos tornando um país 100% voltado aos jogos para os computadores, caminhando no sentido contrário ao do restante do mundo. Os maiores beneficiados com a falta da distribuição oficial dos games foram os produtos falsificados e o mercado informal.

De acordo com a Associação Brasileira das Empresas de Software (ABES), de cada dez games instalados em computadores brasileiros, nove são falsificados. O índice é superior à pirataria em software de aplicativos, que chega a 56%.

Terminado essa contextualização histórica e mercadológica passa-se a apresentar a arquitetura do motor GRIFO.

CAPÍTULO III

ARQUITETURA DO GRIFO

Nesse capítulo será discutida a arquitetura e o funcionamento básico do sistema. Para o desenvolvimento do jogo foi necessário a criação de um sistema de base, um *Framework* ou, no caso, um motor. Esse framework, intitulado de Grifo é a base do desenvolvimento e concentra todo o trabalho computacional do jogo. A Figura 08 mostra o logotipo do motor.

A separação da parte funcional (computacional) da parte de diversão de um jogo é um procedimento muito utilizado na indústria de jogos (AZEVEDO, 2005), visto que as equipes desenvolvedoras, normalmente, são heterogêneas, possuindo colaboradores de vários ramos de trabalho. Grandes jogos como Half Life, Winning Eleven, são produzidos por times com mais de 100 profissionais.



Figura 08: Logotipo do Grifo

Além dos aspectos organizacionais, essa separação provê o encapsulamento do desenvolvimento de jogos, aumentando assim a produtividade, o aproveitamento de código e reduzindo erros (PRESSMAN, 2005).

3.1 Especificação e Análise de Requisitos

O levantamento dos requisitos do Grifo foi construído a partir de (FEIJÓ, et al., 2006) e de *BrainStorms* com pessoas ligadas ao desenvolvimento de jogos, jogadores de futebol e computador.

Usando o estudo feito por (FEIJÓ, et al., 2006), foi possível elencar uma grande quantidade de funcionalidades necessárias a esse tipo de aplicação. Essas funcionalidades foram classificadas e agrupadas em subsistemas e serão discutidas ao longo do capítulo.

A Figura 09 mostra um diagrama que representa a compilação dos BrainStorms feitos para a explicitação dos requisitos gráfico-funcionais do projeto.

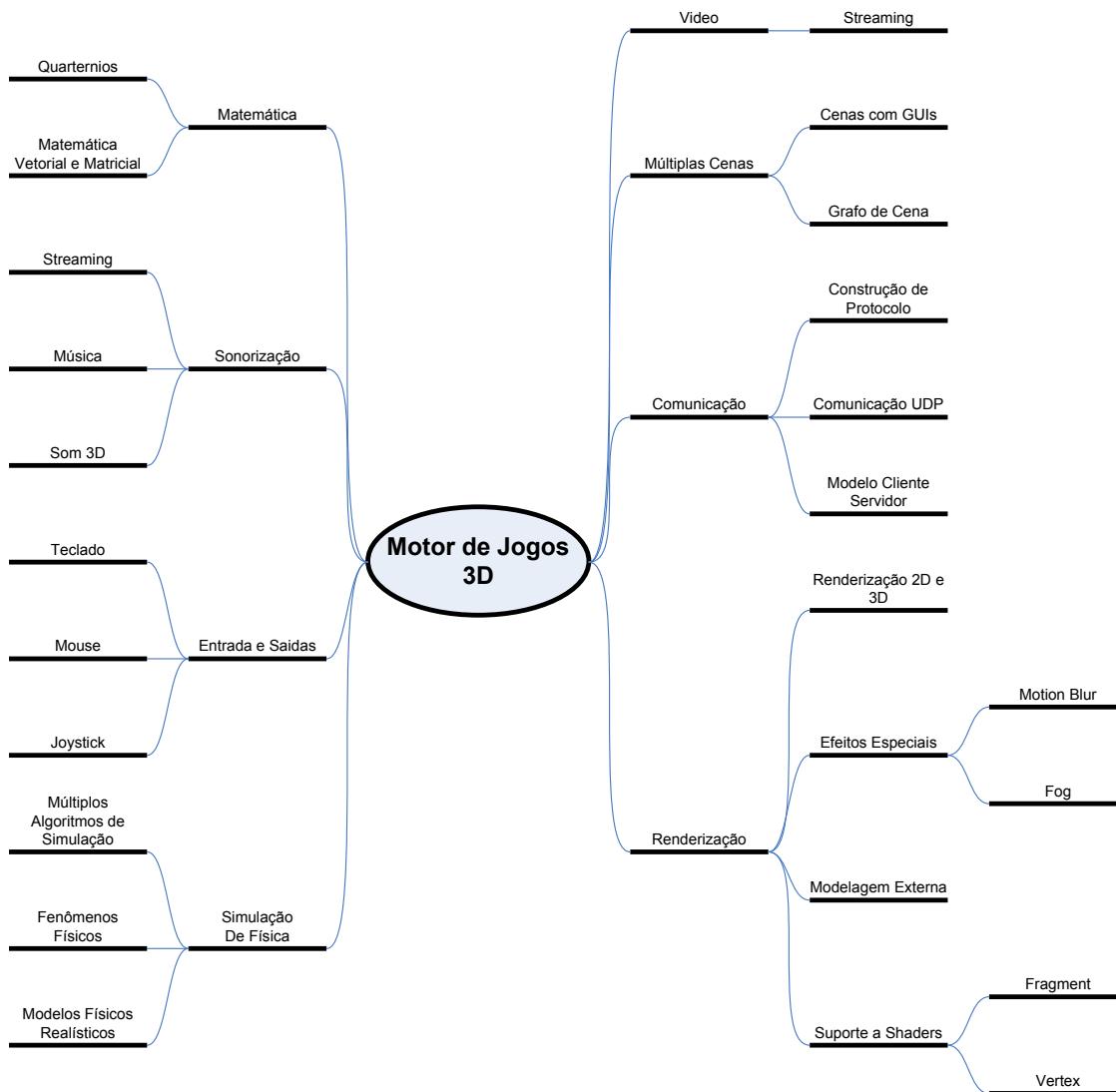


Figura 09: BrainStorm do Grifo

Diante dessa análise, iniciou-se o projeto do motor e do jogo. Cabe ressaltar que a etapa de especificação e análise de requisitos é de suma importância no projeto de qualquer empreendimento, pois é através de uma boa análise que se obtém o sucesso do mesmo.

3.2 Camadas e Arquitetura do Grifo

Para o projeto Grifo decidiu-se trabalhar usando camadas e a analogia com as redes de computadores. O motivo dessa estratégia é a independência obtida entre cada camada, focalizando a especificação e o desenvolvimento das funcionalidades de cada uma delas, além

do alto grau de alteração (modificabilidade) proporcionado. Além disso, a adoção do padrão de projeto Abstract Factory possibilita a generalização do sistema.

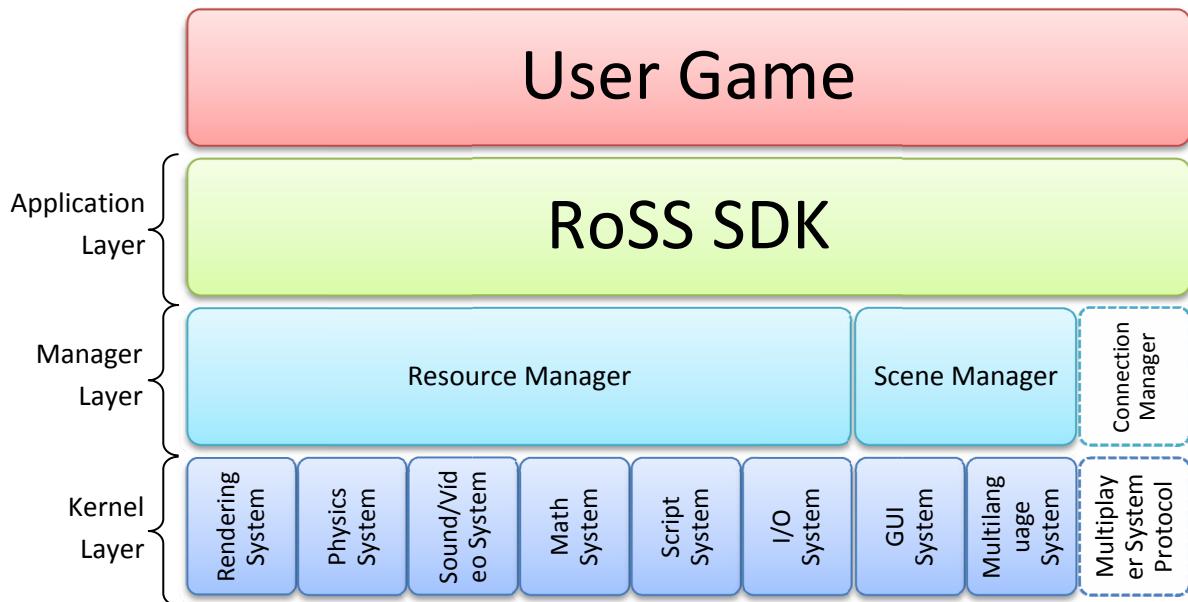


Figura 10: Camadas do Grifo

A Figura 10 mostra a arquitetura das camadas do Grifo. A proposta dele é ser provedor de serviços às necessidades do usuário isentando-o de ter que implementar e gerenciar todos os recursos do jogo. A camada de comunicação não foi devidamente projetada e por isso não será discutida.

3.2.1 Descrição das Camadas e Arquiteturas

Da mesma forma que na arquitetura de redes, cada camada do Grifo provê serviços e funções às camadas superiores. As camadas inferiores preocupam-se em implementar os algoritmos básicos de leitura de arquivo de entrada, tais como imagens, texturas, modelos 3D, XML⁶ e outros, além de especificar as classes mais básicas e gerais do sistema.

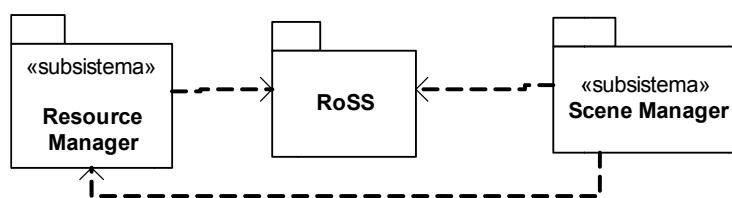


Figura 11: Arquitetura de pacotes, Visão Superior

⁶ eXtensible Markup Language

A Figura 11 mostra a dependência do sistema de gerência de cenas com o sistema de gerência de recursos, isso se dá, pois, pela definição do Grifo, uma cena, seja ela 2D ou 3D, é composta por recursos.

3.2.1.1 Resource Manager(RM)

O Resource Manager é o subsistema responsável por gerenciar todos os recursos, que podem ser Imagens, Sons, Vídeos, Modelos 3D, arquivos de configuração e até scripts, utilizados pelo usuário.

Dentre as funcionalidades providas pelo Resource Manager, podemos destacar:

- Reduzir as duplicações de recursos (em algumas situações a duplicação é necessária).
- Impedir o Vazamento de Memória (Memory Leak).
- Fornecer fácil acesso à criação, modificação e remoção dos recursos.
- Otimizar o processo de carregamento de dados na memória.

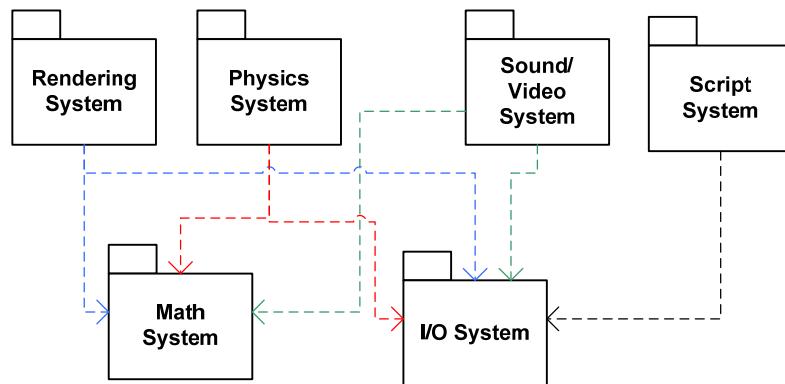


Figura 12: Arquitetura do Resource Manager

Observado a Figura 12, vê-se que tanto o sistema de interface, quanto o sistema de matemática são bases para os demais sistemas. O RM gerencia separadamente cada tipo de recurso, tratando cada um de acordo com as necessidades do seu tipo.

Para cada tipo de recurso, existe uma árvore vermelha e preta (RBTree) (CORMEN, et al., 2002), na qual os elementos são inseridos. A inserção é baseada no nome dado ao recurso e na ordem de solicitação. Essa metodologia permite um rápido acesso ao recurso buscado, pois muitas vezes uma cena possui vários elementos e a rapidez de acesso deve ser levada em conta.

O único subsistema que não possui recursos a serem salvos é o sistema de matemática, contudo, alguns algoritmos devem ser solicitados à mesma, para que esses sejam carregados

no sistema. Isso é possível através do uso de bibliotecas dinâmicas (Wikipédia, a enclopédia livre.), disponíveis em todos os sistemas operacionais.

3.2.1.1.1 Rendering System (ReSys)

O subsistema de renderização, ou desenho, é o cerne da visualização do Grifo, é através dele que as janelas são criadas, as imagens desenhadas, os modelos exibidos.

O ReSys modela e implementa todos os recursos para modelagem de objetos 3D, texturização, filtros, câmeras, luzes dinâmicas, shaders⁷ e outros recursos gráficos. Internamente, ele possui um sistema de análise das capacidades da máquina cliente, de modo a verificar se determinados recursos são possíveis de serem visualizados. Esse tipo de funcionalidade é denominada de *LoD (Level of Detail)*, e é implementada em quase todos os motores comerciais de jogos.

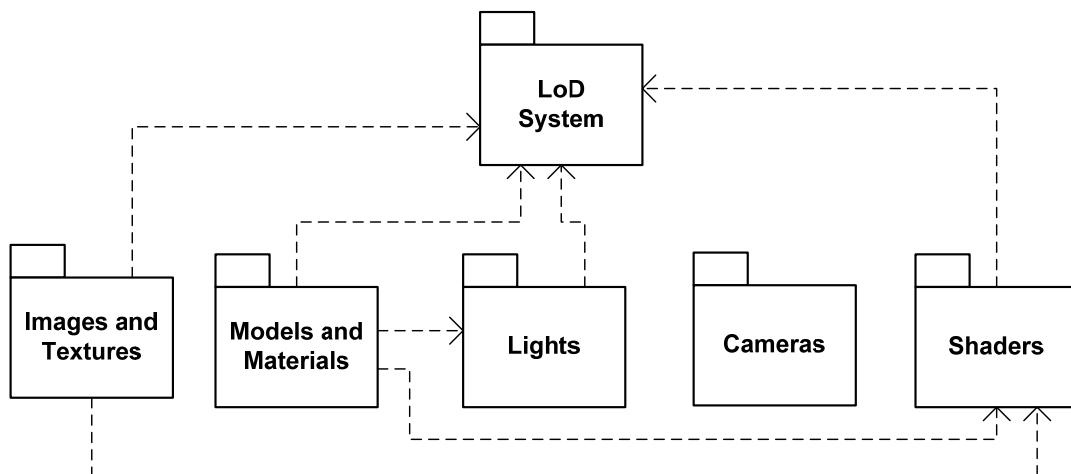


Figura 13: Arquitetura do Rendering System

Na Figura 13 é mostrada a arquitetura desse subsistema e é fácil perceber que todos os subsistemas, exceto as câmeras, dependem do LoD. Pode parecer estranho que as imagens dependam do sistema de LoD, mas, alguns hardwares gráficos não trabalham com transparência e outros não suportam determinados níveis de resolução, sendo necessário ao sistema LoD detectar tais problemas e converter, ou remover, as imagens afetadas.

⁷ Shaders serão explicados na seção sobre shaders

3.2.1.1.1.1 LoD System

O sistema LoD, conforme já mencionado, tem a função de observar a plataforma de hardware e software no qual o Grifo está sendo executado e selecionar/adaptar os recursos para tal plataforma.

Esse sistema é muito criticado pelos desenvolvedores de jogos e motores para consoles. O principal argumento é que ele torna a aplicação dependente da máquina cliente, fazendo com que a experiência de jogo para o jogador seja diferente, podendo ajudá-lo ou atrapalhá-lo. Por outro lado, muitas pessoas não possuem e não têm interesse em um console de jogo, de forma que o computador é utilizado como tal. Os computadores, ao contrário do que ocorre com os videogames, que possuem hardware fixo, são extremamente heterogêneos e, por isso, esse sistema é necessário para o correto funcionamento do jogo.

O LoD é bastante complexo para ser implementado de forma a oferecer robustez à aplicação, para isso ele deve ser bastante genérico. No Grifo o LoD é rodado no início da execução da aplicação e nesse momento ele verifica as funções disponíveis pela máquina cliente. Isso é feito através de primitivas do sistema operacional e das API⁸ gráficas 3D. Após a verificação, um conjunto de flags do LoD é fixado, e toda vez que um recurso é solicitado ao Resource Manager, ele carrega o mesmo e invoca o LoD para adaptá-lo às características do ambiente do usuário. Quando um recurso não é possível de ser utilizado, o Resource Manager libera os dados dele, contudo, mantém a informação de recursos carregado, dessa forma, ao ser solicitado novamente não será necessário refazer a análise.

Um ponto importante de discussão é o fato de que cada subsistema pode apresentar sistema de LoD, de forma que, o sistema de LoD completo é uma composição dos subsistemas de LoD específicos de cada área.

⁸ Application Program Interface, interface para programação de algum recurso.

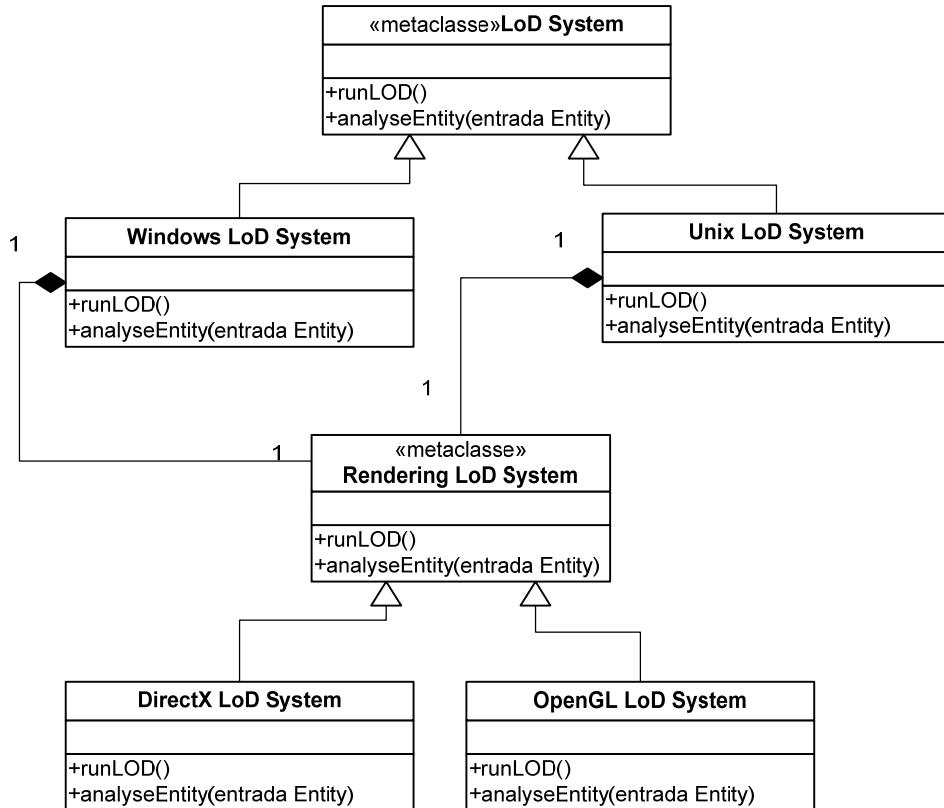


Figura 14: Diagrama de Classes do Sistema de LoD Gráfico

Na Figura 14, observa-se a presença de classes abstratas **LoD System**, que é o sistema geral de LoD, e **Rendering LoD System**, que é a versão para as funcionalidades gráficas, e classes concretas que determinam o sistema operacional e a API de programação 3D escolhida.

Observe que a API Microsoft DirectX só é instanciada no caso do sistema operacional Microsoft Windows. Além disso, o sistema de LoD deve verificar a versão do DirectX instalado, ou do OpenGL⁹, para avaliar as funções disponíveis.

Os sistemas de LoD das outras áreas do Grifo não serão discutidas, pois apresentam a mesma análise do LoD de renderização.

3.2.1.1.2 *Images And Textures*

A imensa maioria dos jogos necessita de imagens e texturas e, portanto, é de vital importância prover meios fáceis de utilização delas.

⁹ APIs de programação 3D. Serão explicadas nos capítulos seguintes.

Para muitas pessoas imagens e texturas são conceitos idênticos, contudo, no contexto de jogos essas são extremamente diferentes. A principal diferença está na forma de utilização da mesma, as imagens são, simplificadamente, matrizes de pixels que são armazenadas na memória principal do computador e rasterizadas, ou seja, desenhadas na tela quando solicitadas.

As texturas, por outro lado, são também, simplificadamente, matrizes de pixels, contudo são armazenadas na memória de textura da placa aceleradora de vídeo, ou na memória principal caso não haja mais espaço, e devem ser mapeadas sob alguma superfície 3D ou 2D, ou seja, são como papéis de paredes aplicados sob determinadas superfícies. Essas superfícies podem ser curvas planas, splines, curvas de Bezier (WATT, 2000), B-Splines, ou qualquer outra superfície modelável¹⁰. A Figura 15 mostra o processo de mapeamento de texturas.

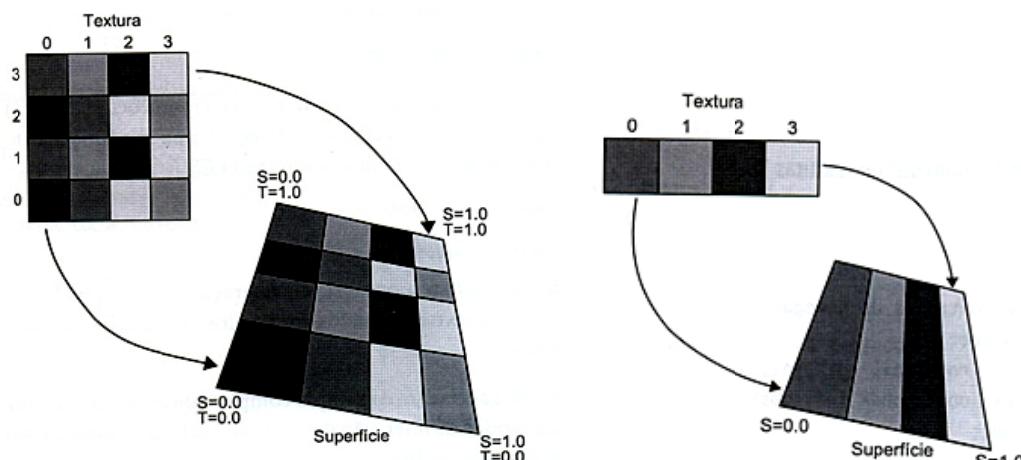


Figura 15: Mapeamento de Textura 1D, 2D

Basicamente, as texturas podem ter de uma a três dimensões, sendo utilizadas para mapear objetos irregulares, contudo texturas de três dimensões, normalmente, não são utilizadas. Toda textura é mapeada para um espaço normalizado, conforme a Figura 16a. As coordenadas são nomeadas de **s,t,r** ou na linguagem comercial, **u,v,w**.

¹⁰ Uma superfície modelável é um Poliedro Convexo, ou seja, obedece a Relação de Euler.

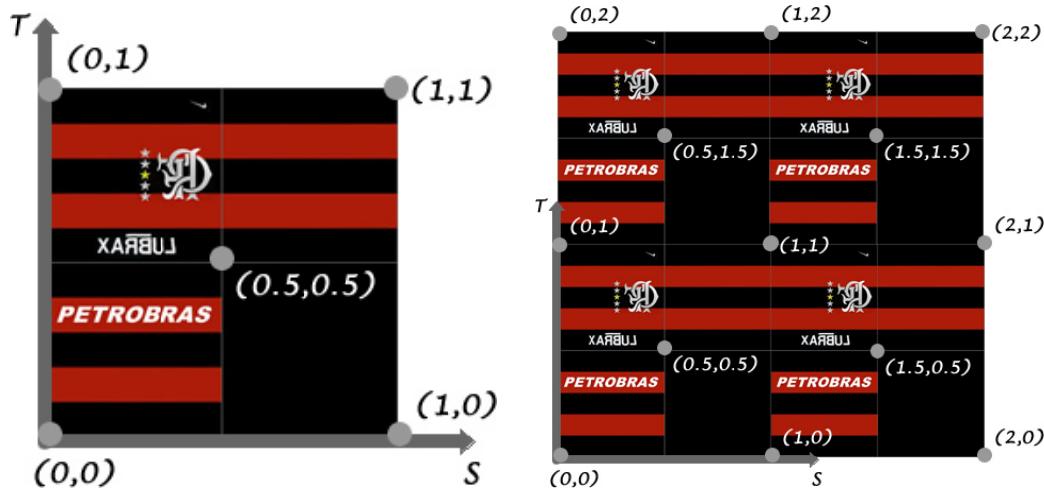


Figura 16: a) Espaço de Mapeamento de textura. b) Replicação de textura

A figura 16b mostra o que ocorre quando acessamos valores maiores do que 1, nesse caso, a imagem é replicada e o mapeamento aumentado. Para podermos manipular com mais precisão uma determinada textura podemos modificar a matriz de textura (COHEN, et al., 2006) do sistema de visualização, utilizando as operações afins (FOLEY, et al., 1997) (escalamento, translação, rotação), dessa forma podemos aumentar a escala da textura, rotacioná-la e transladá-la.

No caso do mapeamento de uma esfera, sua parametrização é necessária. Uma possível forma de fazê-la é utilizar as coordenadas esféricas. Dessa forma baseado no esquema da Figura 17, obtem-se as equações subsequentes.

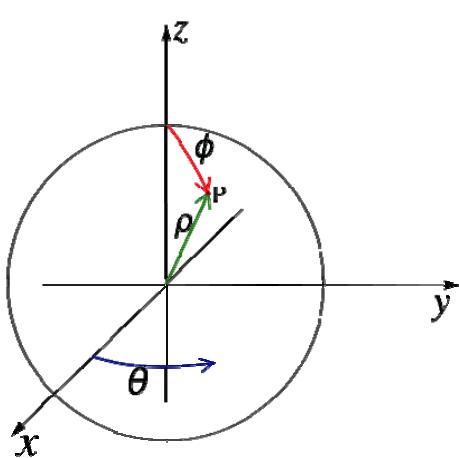


Figura 17: Coordenadas Esféricas

$$\begin{aligned}x &= \rho \sin(\theta) \cos(\phi) \\y &= \rho \sin(\theta) \sin(\phi) \\z &= \rho \cos(\theta)\end{aligned}$$

Equação 01: Coordenadas Esférica para cartesianas

Nas expressões acima, ρ é o raio, θ é o ângulo a partir do eixo *x* ($0 \leq \theta \leq \pi$), e ϕ é o ângulo a partir do eixo *z* ($0 \leq \phi \leq 2\pi$). Para a conversão em coordenadas de textura (*s*, *t*), utilizamos as expressões abaixo ($0 \leq s, t \leq 1$).

$$x = \rho \sin(t \cdot \pi) \cos(s \cdot 2\pi)$$

$$y = \rho \sin(t \cdot \pi) \sin(s \cdot 2\pi)$$

$$z = \rho \cos(t \cdot \pi)$$

Equação 02: Coordenadas de textura de uma esfera em modo cartesiano

Resolvendo as equações acima, obtemos:

$$s = \frac{\cos^{-1}\left(\frac{z}{\rho}\right)}{\pi}, t = \frac{\cos^{-1}\left(\frac{x}{\rho \sin(v \cdot \pi)}\right)}{2\pi}$$

Equação 03: Coordenadas de textura de uma esfera

Com isso, dado um ponto (x,y,z) na superfície da esfera, podemos calcular as respectivas coordenadas de texturas.

Muitas vezes é necessário replicar ou ampliar uma textura quando estamos visualizando uma cena 3D. Quando chegamos próximo à parede percebemos que a textura é ampliada, da mesma forma, quando estamos sobrevoando um cenário percebemos que a textura está sendo replicada. A forma como a repetição da textura deve ocorrer é um parâmetro que diz como cada coordenada de textura deve ser repetida: por exemplo, o *método Repeat*, repete a textura, ou o *método Clamp*, que repete a última seqüência de pixels. Veja a Figura 18a.

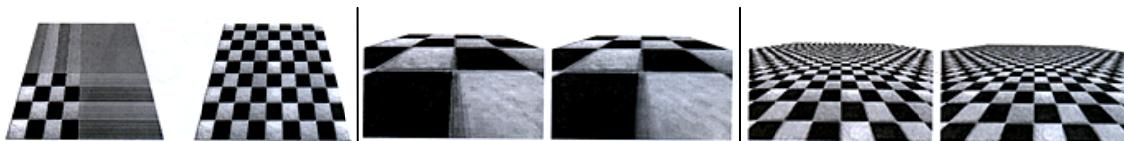


Figura 18: a) Método Clamp e Repeat. b) Método Nearest e Linear. c) Sem e Com Mipmaps

Em diversas situações é necessário especificar ao sistema de visualização a forma como o mesmo irá tratar das situações de ampliação e replicação. Para isso no Grifo, cada textura carrega tais informações.

Quando uma porção da textura é mapeada à um conjunto de pixels é normal que os elementos não estejam alinhados exatamente com a posição dos pixels da tela (HEARN, et al., 2004). Isso ocorre porque a superfície e a textura usualmente têm tamanhos diferentes. Uma forma de solucionar esse problema é copiar o pixel vizinho, essa técnica é chamada de *Nearest Neighbour*, outra forma, é fazer uma interpolação entre os pixels envolvidos, se for uma textura 1D, usa-se interpolação linear, se for 2D usa-se interpolação bilinear. Veja a Figura 18b.

Outra técnica mais eficiente é a do uso de *mipmaps* (CONCI, et al., 2003). O termo *mipmap* vem do latim *multum in parvo*, que significa “muitos em um pequeno espaço”. O mipmap é uma versão da textura de tamanho reduzido. É possível construir vários mipmaps a partir da mesma textura, normalmente reduzindo cada dimensão original pela metade, sucessivamente. O funcionamento é simples: ao se aplicar uma textura, emprega-se sempre o mipmap mais próximo possível ao tamanho da superfície em pixels. Imagine que o observador esteja distante: nesse caso, será utilizado um mipmap pequeno, o que pode inclusive acelerar o desenho. À medida que o observador se aproxima, os mipmaps maiores são progressivamente selecionados até chegarmos à uma distância em que a textura original é empregada. Veja a Figura 18c.

Além desses detalhes, as texturas podem ter suas intensidades modificadas através do parâmetro *Module*, que será influenciado pelas luzes do ambiente.

Muitas placas de vídeo suportam ainda duas formas especiais de geração de coordenadas de textura, denominadas *sphere mapping* e *cube mapping*. Ambas têm o objetivo de simular a reflexão do ambiente em torno de um objeto, porém usando técnicas um pouco diferentes. É importante também observar que ambas são aproximações razoáveis de reflexões reais, porém inferiores às técnicas como *ray tracing* (WATT, 2000), por exemplo. A Figura 19a mostra a aplicação de um sphere map e a 19b a de um cube map.

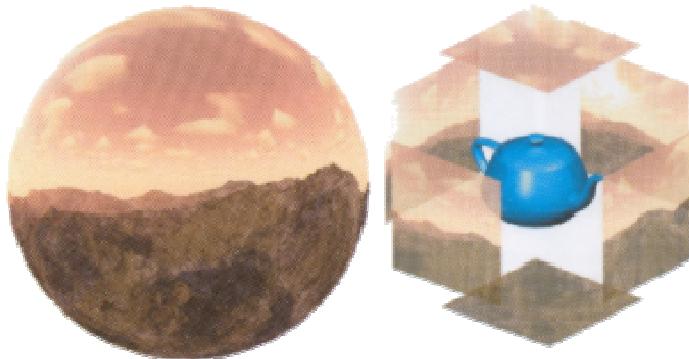


Figura 19: a) Sphere Map. b) Cube Map.

Existem outras técnicas de mapeamento de textura, contudo, elas fogem do escopo desse texto.

Para agrupar todas as funcionalidades necessárias para cada textura, em cada contexto que a mesma pode aparecer, o sistema de texturas deve suportar vários parâmetros e funcionalidades. É importante observar que uma mesma textura pode ser mapeada de várias

formas, logo a forma de mapeamento não é característica da textura e sim do objeto texturizado.

Esse sistema consiste basicamente na classe Texture, sendo que a parte de aplicação da mesma pertence aos modelos. Na maioria dos sistemas, a matriz de pixels ($n \times m$) é convertida em um vetor de dimensão igual ao produto $n * m$, isso é devido à forma como a textura é colocada na memória.

3.2.1.1.1.3 Models And Materials

A forma de representação de objetos 3D mais utilizada em Computação Gráfica consiste na especificação de uma malha de faces poligonais. Nesse caso, uma malha de polígonos representa uma superfície discretizada por faces planas, que podem ser triângulos (preferencialmente) ou quadrados. A estrutura de dados mais usada para armazenar essa malha é uma tabela de vértices e uma tabela de faces que podem ser armazenadas em dois vetores. Conforme ilustra a Figura 20, os dados armazenados na tabela de faces correspondem à índices para a tabela de vértices, dessa forma, evita-se a duplicação de informações.

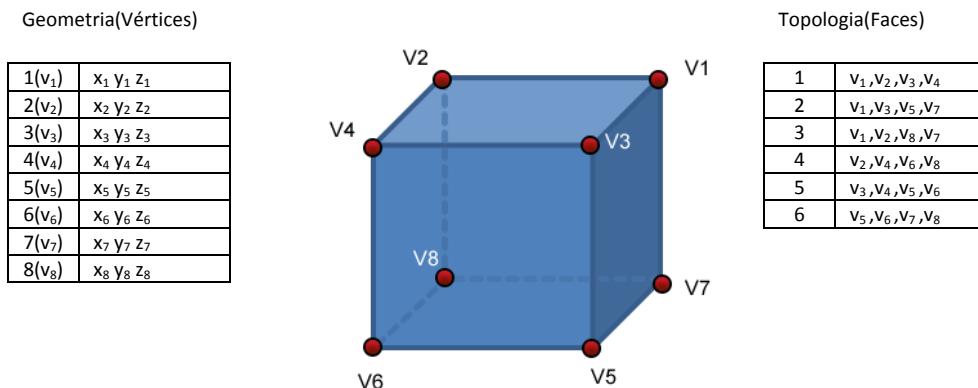


Figura 20: Representação de Objetos 3D, Vértices e Arestas

A Figura 21 mostra o diagrama de classes, simplificado, de um modelo 3D.

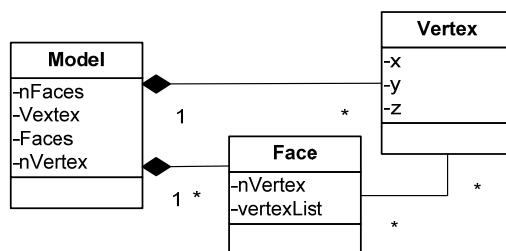


Figura 21: Diagrama de Classes de um modelo 3D

As justificativas para a utilização dessa abordagem são: primeiro e mais importante, separar o modelo 3D do código que desenha o mesmo, e, portanto, quando for necessário trocar o modelo não será necessário modificar o código. Segundo, aumentar a eficiência no armazenamento, pois, se substituirmos as estruturas por um conjunto de chamadas de desenho, provavelmente vários vértices serão repetidamente informados. Isso também criaria um problema considerável no caso da necessidade de alterar o modelo, pois, todas as ocorrências de cada vértice teriam que ser modificadas. A Tabela 01 mostra o pseudocódigo para renderização de um objeto 3D usando a estrutura proposta.

Tabela 01: Renderização de Modelo 3D

```
void render3DModel(Model *model3D) {
    Model *mdl = model3D;

    for(int f = 0; f < mdl->CountFaces(); f++)
    {
        RENDER_API_BEGIN_FACE();
        for(int v = 0; v < mdl->getFace(f).CountVertex(); v++)
        {
            RENDER_PUT_VERTEX(mdl->getVertex(mdl->getFace(f).getVertexIndex(v)).getX(),
                              mdl->getVertex(mdl->getFace(f).getVertexIndex(v)).getY(),
                              mdl->getVertex(mdl->getFace(f).getVertexIndex(v)).getZ());
        }
        RENDER_API_END_FACE();
    }
}
```

É claro que outras informações devem ser armazenadas em cada face para uma melhor representação do modelo. Dentre essas informações, podemos selecionar, para as faces, por exemplo, os vetores normais, as coordenadas de textura, a textura utilizada, e o material.

Um material é uma especificação do modelo de reflexão da luz sobre uma face, ou superfície. Um modelo de reflexão descreve a interação dos raios de luz com uma superfície, considerando as propriedades da superfície e a natureza da fonte de luz incidente, por isso o mesmo conceito será utilizado nas luzes (seção 3.2.1.1.4). O principal objetivo é exibir os objetos tridimensionais na tela bidimensional de maneira que os mesmos se aproximem da realidade. Além disso, por meio de um modelo de reflexão é possível fazer com que objetos do tipo espelho apresentem em sua superfície a imagem de outros objetos do universo. Tal efeito pode ser obtido pela simulação do acompanhamento dos raios refletidos para verificar a cor, ou as cores, que estes trazem de outros objetos.

Diversas informações são necessárias para que um modelo de reflexão processe a intensidade da cor de cada ponto a ser exibido, tais como: cor do objeto, cor da fonte de luz, posição da

fonte de luz, posição na cena 3D do ponto a ser exibido e posição do observador virtual (a posição da câmera é importante).

O tipo de reflexão mais simples é a ambiente, também chamado de luz ambiente. Nela, é considerada a existência de uma fonte de luz não-direcional, resultante de múltiplas reflexões da luz com as superfícies da cena. Outra forma de interpretar a luz ambiente é como a luz que está presente no ambiente, mas que a origem não pode ser precisamente determinada. Portanto, esse tipo de reflexão permite a visibilidade de superfícies que não estejam recebendo diretamente raios de luz.

A reflexão difusa (ou reflexão *Lambertiana*) ocorre na superfície da maioria dos objetos que não emitem luz. Todo objeto absorve a luz do Sol e a luz emitida de uma fonte artificial, refletindo parte desta luz, sendo assim, a reflexão difusa deve-se ao fato de haver uma interação entre a luz incidente e o material da superfície. Por exemplo, supondo um objeto azul que está sendo iluminado por uma fonte de luz branca: o objeto absorve os raios de luz brancos e reflete apenas o componente azul da luz incidente. Uma superfície perfeitamente difusa reflete os raios de luz igualmente em todas as direções, esse tipo de reflexão depende da cor do objeto e da posição da fonte de luz, contudo, a quantidade de luz refletida percebida pelo observador não depende da sua posição. Essa reflexão cria o efeito de dégradé nos objetos.

A reflexão especular é a responsável pela geração do "ponto de brilho" dos objetos. Essa reflexão é processada de acordo com a cor do objeto, a posição da luz e a posição do observador, e a cor do brilho normalmente é a mesma da fonte de luz. A Figura 22 ilustra os três tipos.

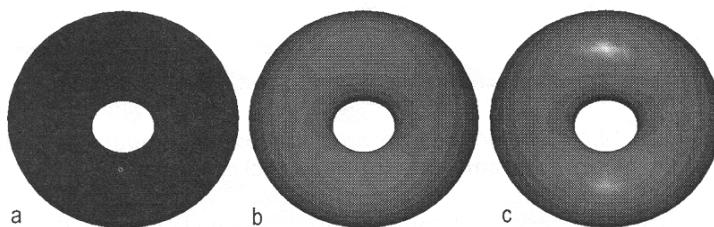


Figura 22: Modelos de Reflexão a) Ambiente b) Difusa c) Especular

3.2.1.1.3.1 Formato de Arquivo OBJ

O formato de arquivo OBJ (Alias/Wavefront object file format) é utilizado pelo Grifo para armazenar um modelo de objeto 3D. Nesse arquivo, além de malhas de faces poligonais, também é possível armazenar outros modelos neste formato, como, por exemplo, uma

superfície NURBS. Diversas ferramentas de modelagem, tais como o 3D Studio Max (<http://www.autodesk.com/3dsmax>) e o Blender (<http://www.blender3d.org>) permitem importar e exportar modelos neste formato.

Dados de geometria e outras propriedades dos objetos, como as texturas (se houver), são armazenados no arquivo OBJ, cujo nome deve ter obrigatoriamente a extensão obj. Informações sobre o material que compõe o objeto são armazenadas em um arquivo MTL correspondente, cujo nome deve ter a extensão mtl. Ambos os arquivos são armazenados como texto ASCII, o que facilita a sua manipulação.

Entre os vários tipos de dados que podem ser incluídos em um arquivo OBJ, destacam-se os listados a seguir. As letras que aparecem entre parênteses são usadas para identificar o conteúdo de cada linha do arquivo.

- vértices da geometria (v);
- vértices de textura (vt);
- normais aos vértices de textura (vn);
- face (f);
- nome do objeto (o).

A Tabela 02 exemplifica o conteúdo de um arquivo OBJ. Foram apresentadas apenas as informações de seis vértices, seis normais e seis faces - as demais linhas foram suprimidas. As duas primeiras linhas do arquivo contêm comentários sobre a sua geração, e por isso iniciam com #.

Na seqüência, é descrito o nome do objeto (o), seguido pelas informações das coordenadas x, y e z de cada vértice (v) do objeto. Depois da listagem dos vértices, começa a listagem dos vetores normais (vn) e das faces (f), nesta ordem. Cada face é composta por três vértices, e um espaço em branco separa as informações para cada um deles. Em geral, uma face composta por três vértices (v1, v2 e v3) é representada da seguinte maneira:

```
f v1/vt/vn v2/vt/vn v3/vt/vn
```

Entretanto, se não houver informações de textura (vt) para a face, são utilizadas duas barras (//). Nesse caso, uma face composta por três vértices (v1, v2 e v3) é representada da seguinte maneira:

```
f v1//vn v2//vn v3//vn
```

O conteúdo do arquivo MTL para o objeto da Tabela 02 é apresentado na Tabela 03. Na primeira linha aparece a informação que um novo material está sendo especificado (newmtl), seguido pelo nome do material. Na seqüência, em cada linha é apresentada uma informação: Ns, para o coeficiente especular (0 a 1000), Kd, para o componente difuso; Ka, para o componente ambiente; Ks para o componente especular e d, para opacidade.

Tabela 02: Exemplo de conteúdo de arquivo OBJ que contém a descrição de uma taca

```
# Blender OBJ File: taca2.obj
# www.blender.org
mtllib taca2.mtl
o Curv_Mesh
v -0.279468148947 0.434727907181 2.95124181093e-07
v 0.0217014085501 -0.323404312134 0.00898867100477
v -0.287736922503 0.3540815413 3.3212688777e-07
v 0.278044521809 0.101156517863 0.115169532597
v -0.202977135777 -0.565464437008 0.202977597713
v -0.306754589081 0.255400657654 2.95576626286e-07
... (coordenadas dos demais vértices)
vn -0.911379754543 0.163915902376 0.377516239882
vn -0.990027487278 0.140874445438 -1.35354475308e-16
vn -0.954665005207 0.297682374716 -1.30519783298e-16
vn 0.906372725964 -0.193769291043 0.375422269106
vn 0.693705320358 -0.193767488003 0.693705320358
vn 0.672276139259 -0.30998313427 0.672276139259
... (coordenadas dos demais vetores normais)
usemtl Vidro
f 79//1 40//2 42//3
f 119//4 184//5 174//6
f 4//7 41//8 291//9
f 399//10 8//11 10//12
f 506//13 27//14 357//15
f 384//16 36//17 323//18
... (informações das demais faces)
```

Tabela 03: Exemplo de arquivo MTL (taca2.mtl),correspondente ao arquivo OBJ da Tabela 1.

```
newmtl Vidro
Ns 727.450980392
Kd 0.34952968359 0.53215277195 0.873049259186
Ka 0.10000000149 0.10000000149 0.10000000149
Ks 1. 0 1. 0 1. 0
d 0.415606021881
illum 2
```

Uma informação importante é que o Grifo ainda não suporta animação 3D, essa simplificação foi necessária para reduzir o tempo de desenvolvimento, dado que o sistema necessário para prover tal suporte, é de alta complexidade.

3.2.1.1.4 Lights

Um modelo de iluminação define a natureza da luz que emana de uma fonte e a interação da mesma com todos os objetos de uma cena. Essa natureza diz respeito à fonte de luz utilizada. Normalmente, consideram-se três tipos diferentes, o Pontual, o Direcional e o Spot, conforme Figura 23.

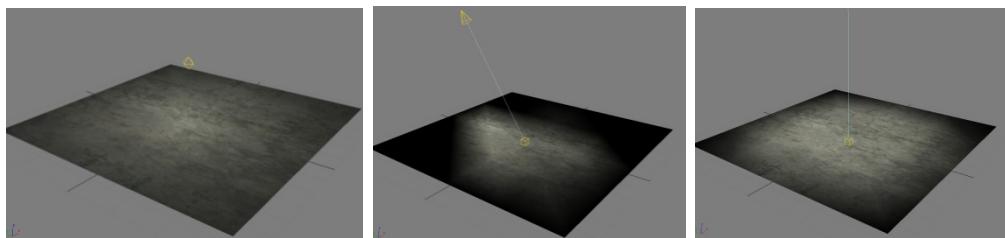


Figura 23: a) Luz Pontual. b) Luz Spot. c) Luz Direcional.

A Figura 24 ilustra os tipos de luzes dinâmicas suportados pelo Grifo. Perceba que tipo direcional não é implementado, isso é devido à falta de mecanismos para modelagem dessa luz nas APIs 3D conhecidas, para conseguir tal efeito é necessário o uso dos shaders.

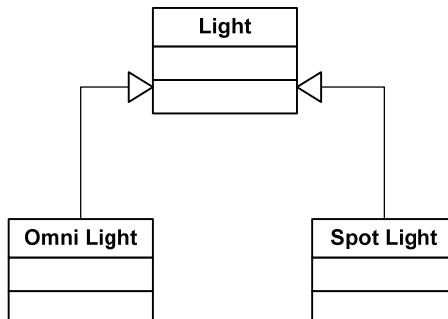


Figura 24: Diagrama de Classes do Sistema de Luzes

Cada luz possui dados como posição, intensidade, cor, decaimento linear, decaimento quadrático, direção (no caso da spot), tipos de emissão (difusa, ambiente, e outras descritas nos materiais).

3.2.1.1.5 Cameras

Para visualizar uma cena 3D é necessário especificar um observador virtual. Na maioria das APIs o Modelo de Câmera utilizado é o de observador-objeto (Figura 25). Nesse modelo basta

dizermos a posição do observador, a posição do objeto observado e a orientação do observador, usado para indicar, por exemplo, se o mesmo encontra-se de cabeça para baixo ou não.

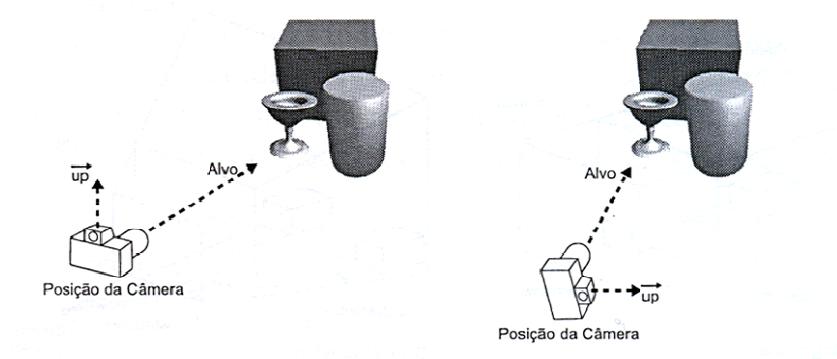


Figura 25: a) Visualização da Câmera. b) Câmera inclinada a direita

Contudo, esse tipo de câmera é muito restrita, pois muitos jogos, como os de 1^a e 3^a pessoa, utilizam efeitos como caminhada que não são muito bem apresentadas usando esse modelo. A fim de resolver o problema, foram criados três tipos de câmeras: A Câmera convencional, descrita acima, a câmera de foco, ou também chamada de Câmera de órbita, e a câmera de caminhada (Figuras 26 e 27).

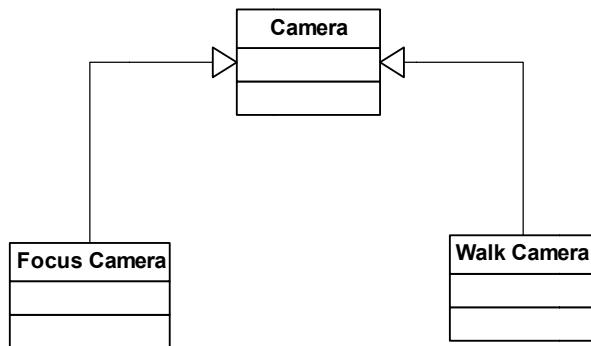


Figura 26: Diagrama de Classes do sistema de Câmeras

A câmera de Foco tem uma posição e um alvo, assim como a câmera convencional, contudo ela implementa os efeitos de Tilt, Pan, Roll, mostrados na Figura 28. Quando ela executa uma movimentação ou esses efeitos, ela ainda mantém o alvo, logo se ela utilizar o efeito Pan em 360°, ela terá orbitado em torno do objeto alvo.

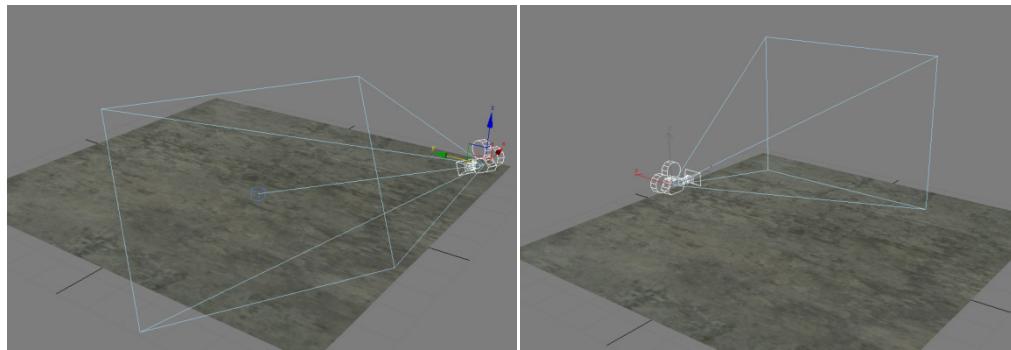


Figura 27: a) Câmera de Foco. b) Câmera Walk.

A implementação é muito simples, basta mudar a posição e manter o alvo. Os efeitos Tilt, Pan, Roll são simplesmente rotações em torno dos eixos cartesianos

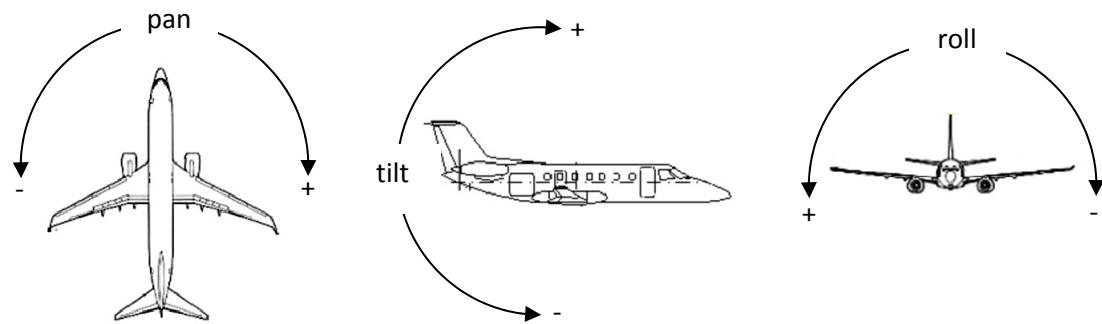


Figura 28: Ângulos de Rotação

No caso da Câmera Walk, são informados a posição, a direção da câmera e a orientação.

Durante a movimentação da câmera, quando o efeito Pan é utilizado, gira-se o vetor de direção em torno do eixo Y, quando o efeito Roll é chamado, gira-se o vetor de orientação em torno do eixo Z, para o efeito Tilt, é necessário girar o vetor de direção em torno do vetor $(0,1,0) \times \text{direção}$, conforme Figura 29.

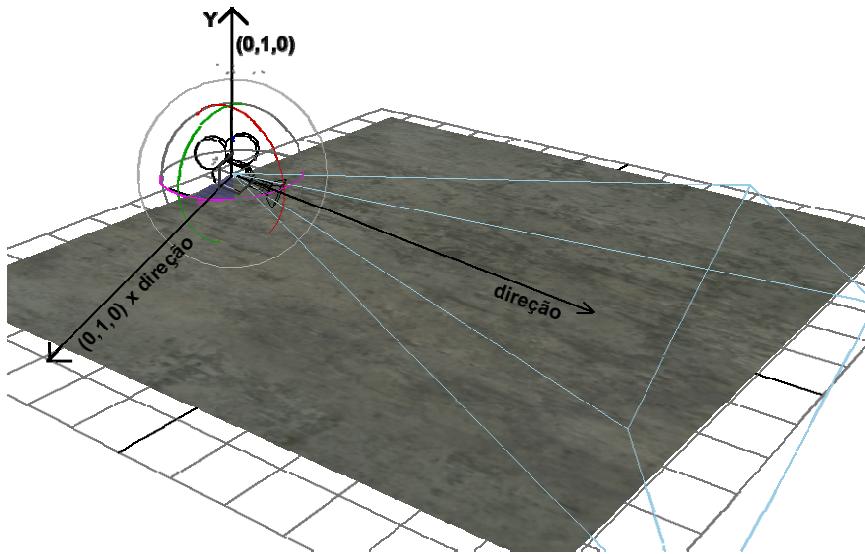


Figura 29: Efeito Tilt na Câmera Walk

Para essa rotação em torno de um vetor qualquer, fez-se uma dedução da matriz que realiza tal transformação linear. Para isso, considere e uma rotação de α unidades angulares em torno de \mathbf{u} .

Seja a matriz , obtém-se a Matriz M de rotação como sendo:

Equação 04: Matriz de rotação em torno de um eixo qualquer

3.2.1.1.6 Shaders

Uma característica essencial do hardware gráfico moderno é a sua programabilidade: graças a isso, é possível produzir efeitos mais sofisticados de iluminação e texturas, e de forma geral, imagens com mais qualidade. Um exemplo de shader é o *bump mapping*, a figura 30 ilustra o efeito.



Figura 30: a) Modelo Original b) Modelo com Bump Mapping c) Textura do Modelo

A figura 30a não apresenta os víncos bem ressaltados, mas a figura 30b, os ressalta sem adicionar mais polígonos, esse efeito é o *bump mapping*.

3.2.1.1.6.1 Arquitetura e Programação de GPUs¹¹

Hoje os hardwares que mais avançam na indústria são as GPUs, ou também chamadas de GPGPUs (General-Purpose Computing on Graphics Processing Units). Desta forma, é difícil apresentar uma arquitetura geral. Mas, há elementos comuns entre todas as placas aceleradoras gráficas.

O componente mais importante de uma GPU é a sua memória, que é utilizada para várias finalidades e é dividida em várias partes. A primeira delas é o *frame buffer*, que é uma região da memória onde serão escritos os valores dos pixels que serão mostrados na tela. Pode-se assumir que a tela é um espelho do frame buffer: tudo o que for escrito nesta memória será mostrado no monitor. Conectada ao frame buffer está a controladora de vídeo, que irá converter o sinal digital presente nesta memória para o sinal analógico que será enviado ao monitor, se o mesmo for analógico.

Ao renderizar uma imagem, os polígonos são plotados¹² seqüencialmente, logo, se estes forem desenhados diretamente no frame buffer, a aplicação gráfica apresentará imagens que vão sendo formadas aos poucos. Outro fato é que muitos dos polígonos serão logo em seguida sobrepostos por outros que estão à sua frente, ocorrendo o desaparecimento dos primeiros, dessa forma, se o desenho for efetuado no frame buffer o usuário irá perceber polígonos "piscando". Assim sendo, outro componente da GPU é o *back buffer*, toda renderização será feita nesta área de memória, apenas ao terminar uma imagem por completo é que o conteúdo desta será transferido para o frame buffer principal, também conhecido como *front buffer*.

No desenho de polígonos no back buffer, pode ocorrer que um polígono se sobrescreva a outro, pois na descrição da cena, o segundo está na frente do primeiro. Assim, é necessário que haja uma memória que armazene a profundidade do último polígono desenhado em cada pixel. Antes de plotar um novo polígono, será feito um teste para ver se sua profundidade é maior ou menor que o valor escrito nesta memória. Caso seja maior, este polígono será totalmente descartado do pipeline. Esta memória de profundidade é denominada *Z-buffer* (WATT, 2000).

¹¹ GPU – Graphical Processor Unit

¹² Desenhados

Outro componente é o *stencil buffer*, esse é uma área da memória usada para operações de máscara.

Existe ainda mais uma área de memória chamada de *accumulation buffer*. Essa permite que várias imagens sejam desenhadas ao mesmo tempo, possibilitando que haja sobreposição entre elas. Assim, nesta área serão compostas imagens formadas a partir de duas ou mais, fundamental para criar efeitos como o *motion blur* ou *depth of field*.

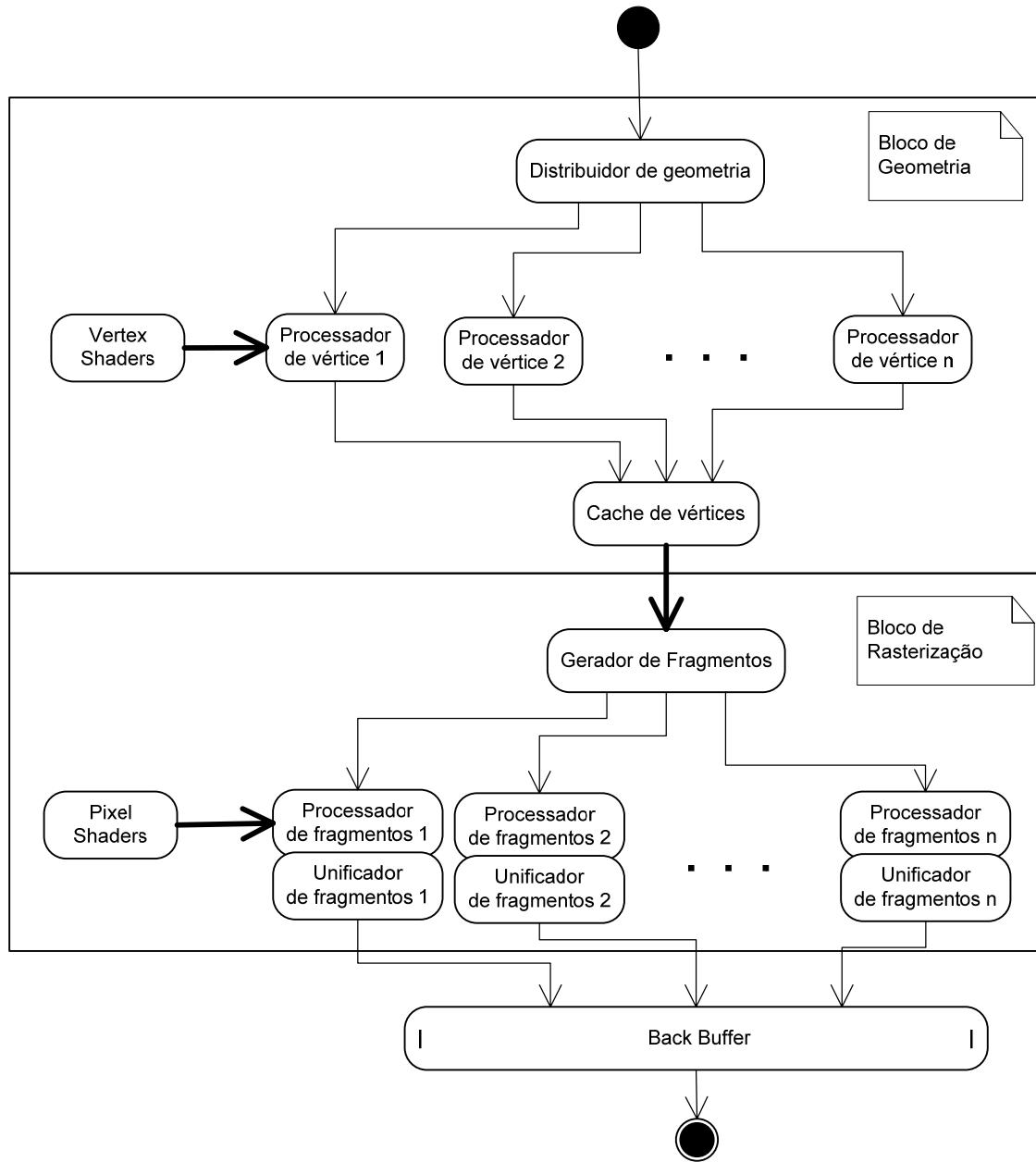


Figura 31: Arquitetura de uma GPU

Além da memória, o hardware gráfico é dividido em duas regiões: o bloco de geometria e o bloco de rasterização. De forma simplificada, o primeiro bloco irá tratar os vértices e o

segundo cuidará dos fragmentos, que são os pixels que ainda não foram mostrados na tela. Uma das principais razões da eficácia das GPUs consiste na sua arquitetura de processadores paralelos. Vários vértices podem ser tratados simultaneamente, bem como vários pixels também o são.

A Figura 31 ilustra a arquitetura básica de uma GPU, nela, o distribuidor de geometria se encarregará de distribuir os vértices para cada um dos processadores de vértices. O processador de vértice fará, em operações de hardware, as operações do pipeline gráfico que corresponde ao estágio de geometria: transformação, iluminação de vértice, projeção, recorte e transformação para coordenadas de tela. Através dos vertex shaders o programador pode escrever um pequeno programa que alterará as operações implementadas no hardware.

O *cache de vértices* é uma memória que recebe os vértices já processados e colocados no plano de projeção. O gerador de fragmentos irá distribuir polígonos para serem preenchidos e rasterizados. Isto será efetuado por outros processadores, denominados de processadores de fragmentos. Chama-se um programa de pixel shader, aquele que alterará o processo de rasterização padrão do hardware, permitindo que o programador insira uma série de efeitos. O unificador de fragmentos irá fazer um teste antes de escrever o candidato a pixel no front buffer, verificando através do Z-buffer, se este fragmento está escondido por outro pixel pertencente à um polígono que está na frente do polígono que deu origem ao fragmento.

Atualmente, as etapas de processamento de vértice e processamento de fragmento podem ser alteradas várias vezes durante a síntese de uma mesma imagem, permitindo que diferentes tratamentos e efeitos sejam dados durante a renderização. Para desenvolver estes programas há diversas linguagens de alto nível, sendo as mais conhecidas a Cg (C for Graphics) da Nvidia, a HLSL (High Level Shader Language) da Microsoft e a OpenGL Shader Language (ROST, 2004).

3.2.1.1.6.1.1 Vertex Shaders

O processador de vértices é responsável por efetuar principalmente as seguintes operações: transformação da posição do vértice, geração de coordenadas de textura para a posição do vértice, iluminação sobre o vértice, operações para determinar o material a ser aplicado ao vértice. Os vertex shaders são programas que irão interferir e alterar de alguma maneira todas ou algumas destas tarefas. Assim sendo, todo vertex shader possui como entrada um vértice e alguns de seus atributos e produz como saída este mesmo vértice com os atributos modificados.

Dentro deste programa, serão utilizados: a normal do vértice, matrizes de modelagem, projeção e de texturização, a coordenada de textura, a iluminação referente ao vértice e algumas variáveis globais configuradas previamente. Efeitos que tipicamente exigem o uso de vertex shaders são os de geração de texturas procedurais, efeitos de iluminação per-vertex e outros.

3.2.1.1.6.1.2 Pixel Shaders

A principal função dos pixel shaders consiste em computar a cor de um fragmento. Como já foi discutido, um fragmento é um candidato a pixel. A GPU irá pintar um fragmento na tela, mas este ainda pode ser sobreescrito por algum outro fragmento tratado posteriormente e que pertença a um polígono que está na frente daquele que deu origem ao primeiro. Um pixel shader não pode alterar as coordenadas na tela do fragmento sendo tratado, visto que no vertex shader era devolvido o valor da posição, que corresponde ao vértice devidamente projetado. A entrada de um pixel shader, por outro lado, não é um vértice, mas sim um pixel, assim, quando um pixel shader recebe a posição não significa que esteja recebendo as coordenadas de um vértice, mas sim o valor interpolado correspondente à posição geométrica da parte do polígono ao qual o pixel pertence. Isso ocorre também com as demais variáveis provindas do vertex shader. As coordenadas do fragmento na tela podem ser acessadas através de variáveis pré-definidas.

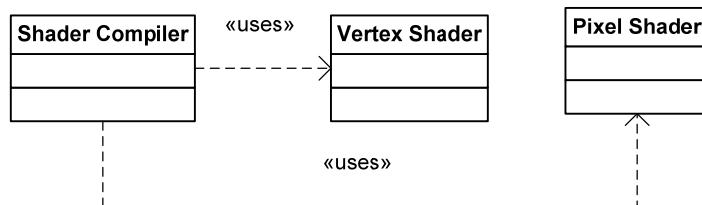


Figura 32: Sistema de Shaders

O sistema de shaders, vide Figura 32, conta com as classes específicas de Shader e com o compilador de shader. Como mencionado anteriormente, os shaders são programas escritos em linguagem de alto nível, logo, para funcionarem nas placas eles devem ser compilados. O compilador de shader após receber o ponteiro do arquivo do shader, faz o tratamento léxico, sintático e semântico e por fim gera os bytes de compilação. Esses bytes serão passados para a placa e quando solicitados serão usados para modificar a renderização das imagens.

3.2.1.1.2 Physics System

O sistema de física do Grifo foi constituído de forma a prover os elementos necessários para modelagem física e simulação numérica de fenômenos da Física. Contudo, se modelarmos o sistema físico com o mesmo número de detalhes do mundo gráfico, as simulações se tornariam muito onerosas ao sistema e inviabilizariam a execução da aplicação. Por essas razões, o modelo físico é uma simplificação do modelo gráfico. A Figura 33 mostra a estrutura do sistema de física.

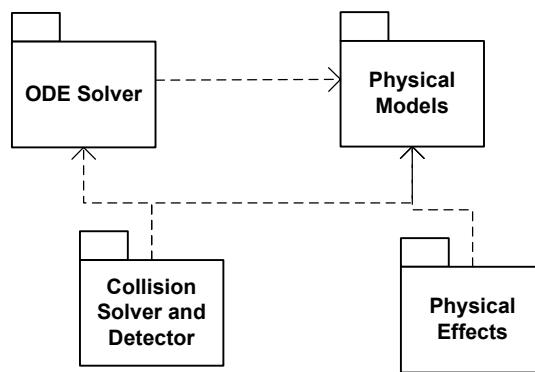


Figura 33: Arquitetura do Sistema de Física

Na figura, vemos o subsistema de simulação das equações da física, o de detecção e resolução de colisão e um repositório de modelos físicos, os quais serão associados a entidades gráficas.

3.2.1.1.2.1 ODE Solver

O sistema de simulação das equações diferenciais trata de resolver as equações diferenciais advindas do conjunto de Modelos Físicos na Cena e dos Efeitos Físicos, gerar o valor futuro dessas equações e atualizar os modelos para que o sistema de visualização desenhe os objetos em uma nova posição.

O Grifo descreve quatro tipos de simuladores, os clássicos Método de Solução de equações diferenciais ordinárias (EDO ou ODE) de Euler, Runge-Kuta 2 e Runge-Kuta 4 (BURDEN, et al., 2003), e o Método de Monte Carlo, contudo o último não foi implementado por questões de prazo e complexidade. A Figura 34 mostra o diagrama de Classes do ODE Solver.

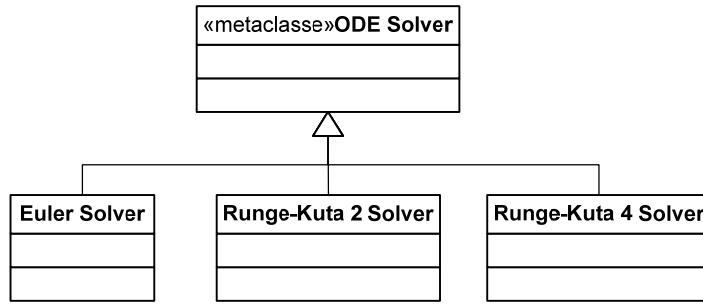


Figura 34: Diagrama de Classes do ODE Solver

Os algoritmos desses métodos estão disponíveis na maioria dos livros de cálculo numérico e, também, na internet. No capítulo 4 será mostrado um dos algoritmos utilizados.

3.2.1.1.2.2 Physical Models

Os modelos físicos são as representações simplificadas de objetos gráficos. Um exemplo simples é o modelo de um ônibus. O modelo gráfico possui detalhes como o ar condicionado, entrada das rodas, portas e janelas, esses detalhes não são relevantes para a simulação física, apenas aumentam a complexidade e o tempo de cálculo, dessa forma, um modelo físico equivalente poderia ser um paralelepípedo com as dimensões de contorno do ônibus. Essa simplificação reduz significativamente a complexidade da simulação e não compromete muito o realismo.

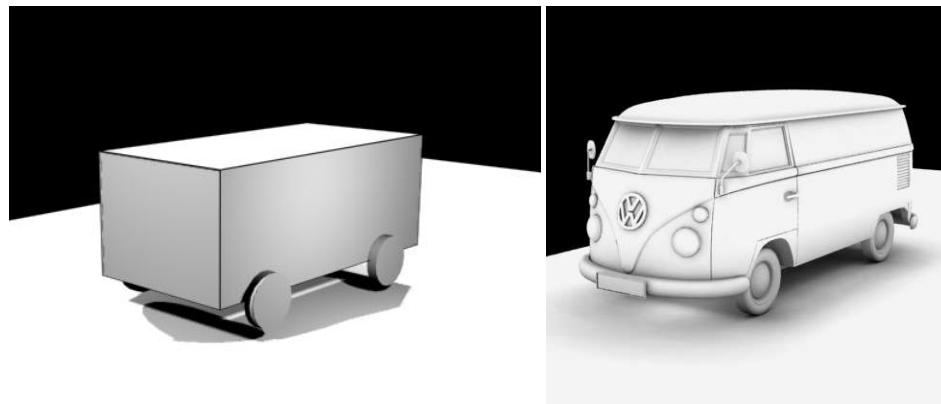


Figura 35: Modelo Físico e Modelo Gráfico

Tendo um modelo geométrico especificado, podemos calcular as propriedades físicas do objeto de forma analítica. Essas propriedades são, por exemplo, o centro de massa, momentos de inércia, nos três eixos, e outras.

O Grifo trabalha com quatro modelos físicos específicos, o cubo, a esfera, o paralelepípedo e o cilindro, além disso, o usuário pode especificar uma malha, um modelo genérico, e as massas

associadas a cada vértice do objeto e solicitar ao motor a criação de um modelo físico para o objeto.

Esse modo é muito mais custoso, não pelo processo inicial de cálculo dos momentos de inércia e centro de massa, mas pela complexidade associada à detecção e à resolução de colisão dessas malhas, esses sim são realizados várias vezes durante o jogo. Para calcularmos o momento de inércia ($I_{xx} = \int r_x^2 dm = \int (y^2 + z^2)dm$), usamos a forma discreta do mesmo e no caso de eixos paralelos, o cálculo é feito via teorema dos eixos paralelos ($I = I_0 + md^2$).

Após o cálculo do centro de massa, todos os vértices do modelo são recalculados para ficarem relativos ao centro de massa. Esse processo é feito apenas na iniciação da cena.

O motor foi projetado para trabalhar apenas com corpos rígidos, dessa forma, não são considerados as partículas e outros objetos que de algum modo perdem partes. O motivo da simplificação é a não necessidade e a complexidade associada. A Figura 36 mostra as classes associadas aos corpos rígidos.

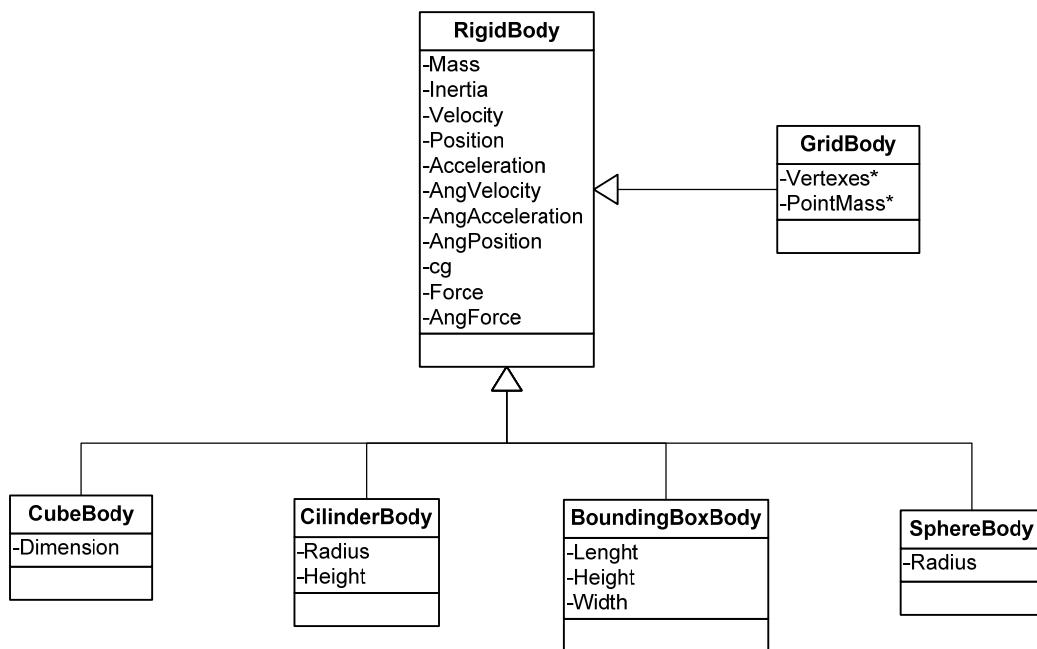


Figura 36: Diagrama de Classes dos Modelos Físicos

Excluindo o **GridBody**, todos os modelos possuem os cálculos de Momento de Inércia e Centro de Massa tabelados (HALLIDAY, et al., 2004), além disso, o método de resolução e detecção de colisão é mais simples. Abaixo são mostradas as fórmulas dos momentos de Inércia.

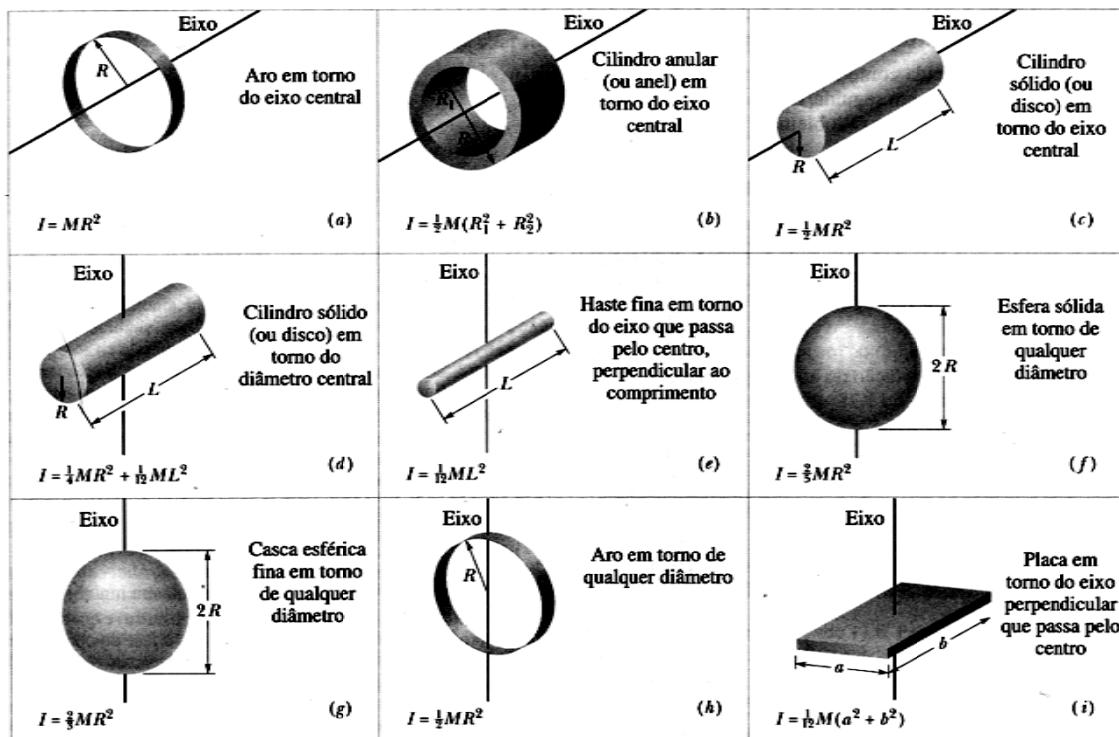


Figura 37: Momentos de Inércia de Objetos (HALLIDAY, et al., 2004)

3.2.1.1.2.3 Collision Solver and Detector

Para a detecção e resolução das colisões devem-se levar em consideração os conceitos físicos de Quantidade de Movimento ou Momento Linear, Energia Cinética e Impulso. O primeiro passo do sistema é descobrir se há colisão entre dois objetos e onde a mesma ocorre, isso permite calcular efeitos de rotação. Na sequência é realizada a resolução da colisão, que consiste em aplicar impulso nos objetos envolvidos até que os mesmos parem de colidir, observe que esse processo não é mostrado para o usuário final, isso ocorre antes da renderização da cena. Um algoritmo completo de resolução de colisão será mostrado no capítulo 4, além disso, os detalhes da resolução de colisão em malhas não serão abordados.

Para a detecção de uma colisão, deve-se saber a geometria dos objetos envolvidos, sendo que para os objetos com geometrias definidas pelo motor, pode-se desenvolver uma metodologia de teste.

As colisões de Esfera – Esfera, Esfera – Cilindro e Cilindro – Esfera são fáceis de serem desenvolvidas, pois, conforme figura abaixo, elas dependem somente do cálculo da distância euclidiana entre os centros geométricos das estruturas.

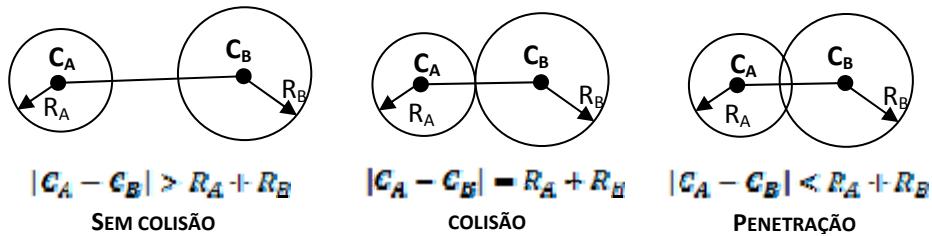


Figura 38: Colisão entre esferas

Para as combinações de casos de Esfera e Cilindro com o paralelepípedo, e paralelepípedo com paralelepípedo, a complexidade das expressões aumenta consideravelmente. Para colisão entre paralelepípedos (aqui mostrado somente em um plano), tem-se o algoritmo mostrado na Tabela 04.

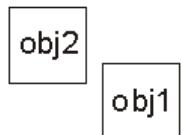
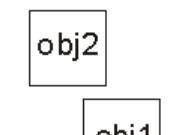
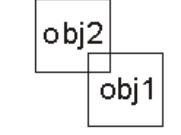
Tabela 04: Algoritmo de Teste de Colisão em Retângulos

```
// No caso x,y são as posições dos centros e l,a são as dimensões largura e
// altura
bool Colision(Rect &obj1, Rect &obj2)
{
    if (obj1.x > obj2.x + obj2.l) return false; //obj1 está a direita de obj2
    if (obj1.y > obj2.y + obj2.a) return false; //obj1 está abaixo de obj2
    if (obj1.x + obj1.l < obj2.x) return false; //obj1 está a esquerda de obj2
    if (obj1.y + obj1.a < obj2.y) return false; //obj1 está acima de obj2

    return true; //existe colisão, ou seja, obj1 está tocando em obj2
}
```

Essa construção da rotina é muito eficaz e clara, pois se, por exemplo, o primeiro teste for verdadeiro, significa que no plano x, o objeto 1 está à direita do objeto 2 mais a sua largura, então cai fora dizendo que não há colisão, o mesmo acontece para outros testes. Veja os exemplos da Tabela 05.

Tabela 05: Colisão entre Retângulos

	obj1 está a direita de obj2 obj1 não está abaixo de obj2 obj1 não está a esquerda de obj2 obj1 não está acima de obj2 (não existe colisão devido ao primeiro caso)
	obj1 não está a direita de obj2 obj1 está abaixo de obj2 obj1 não está a esquerda de obj2 obj1 não está acima de obj2 (não existe colisão devido ao segundo caso)
	obj1 não está a direita de obj2 obj1 não está abaixo de obj2 obj1 não está a esquerda de obj2 obj1 não está acima de obj2 (existe colisão pois nenhum dos testes foram verdadeiros)

O tratamento da resolução de colisão abordado pelo motor é baseado nos princípios clássicos (Newtonianos) de Impacto. Neles, os corpos não se partem, não sofrem deformações, compressões e outros efeitos. Do ponto de vista prático, pode parecer irreal, contudo, a idealização desse modelo consegue resolver com grande realismo muitas colisões, além de simplificar consideravelmente a análise envolvida.

O princípio fundamental do tratamento das colisões está na ferramenta física do Impulso e na Quantidade de Movimento. O Impulso é definido como uma força que age durante um curto período de tempo, infinitesimal. Mais especificamente, o impulso é uma entidade vetorial igual à variação do momento, ou quantidade de movimento (teorema da Impulsão – quantidade de movimento linear (HALLIDAY, et al., 2004)).

$$\mathbf{J}_L = \int_{t_-}^{t_+} \mathbf{F} dt = \Delta \mathbf{p} = m(\mathbf{v}_+ - \mathbf{v}_-)$$

$$\mathbf{J}_A = \int_{t_-}^{t_+} \boldsymbol{\tau} dt = \Delta \boldsymbol{\ell} = I(\boldsymbol{\omega}_+ - \boldsymbol{\omega}_-), \text{ onde } \boldsymbol{\ell} = \mathbf{r} \times \mathbf{p}$$

Equação 05: Momento de Inércia Linear e Angular

Nas equações, \mathbf{F} é a força de impulsão, τ é o torque de impulsão, t é o tempo, v é a velocidade, o subscrito – significa o instante anterior ao impacto e + o instante posterior ao impacto. O cálculo da força média aplicada resulta nas expressões:

$$\mathbf{F} = \frac{m(v_+ - v_-)}{t_+ - t_-}$$

$$\tau = \frac{I(\omega_+ - \omega_-)}{t_+ - t_-}$$

Equação 06: Força e Torque médio durante um impulso

Além do impulso, outro ponto importante, e fundamental, é o Princípio Newtoniano de Conservação da Quantidade de Movimento, isso implica que em uma colisão de dois corpos com massas m_1 e m_2 e velocidades v_1 e v_2 , respectivamente, as velocidades após o impacto devem obedecer à relação:

$$\Delta p = 0 \Rightarrow m_1 v_{1-} + m_2 v_{2-} = m_1 v_{1+} + m_2 v_{2+}$$

Equação 07: Conservação do Momento Linear

O princípio crucial assumido é que durante a colisão, somente a força de impulsão é considerada, sendo as demais negligenciadas, devido ao fato de seu elevado valor em um curto intervalo de tempo na colisão.

Quando uma colisão ocorre e os corpos envolvidos sofrem deformações, o que realmente ocorre é a transformação da energia cinética em energia de compressão, calor e outras formas de energia. A energia cinética é a energia associada ao movimento dos corpos, sua expressão para o caso linear e angular é:

$$K_L = \frac{1}{2} mv^2$$

$$K_A = \frac{1}{2} I\omega^2$$

Equação 08: Energia Cinética Linear e Angular

Colisões que envolvem perda de energia cinética são ditas colisões *inelásticas*, ou *plásticas*. Dentre essas, temos a colisão *perfeitamente inelástica*, que, após a colisão, considera que os corpos envolvidos caminham juntos e com a mesma velocidade. Outro tipo de colisão muito importante é a colisão *perfeitamente elástica*, que considera que a energia cinética do sistema será conservada, não havendo perdas. Apesar do forte escopo teórico dessa, em casos reais sabe-se que a colisão será algo entre a situação perfeitamente inelástica e a perfeitamente elástica.

Para simular esse efeito, deve-se utilizar a razão de separação relativa dos corpos após a colisão. Dessa relação é criado um fator chamado de *coeficiente de restituição* (e), e sua expressão é dada por:

$$e = \frac{v_{1+} - v_{2+}}{v_{1-} - v_{2-}}$$

Equação 09: Coeficiente de Restituição

Esse coeficiente é determinado experimentalmente em cenários específicos de colisão. Para colisões perfeitamente inelásticas, $e = 0$, para colisões perfeitamente elásticas, $e = 1$.

Além dos efeitos lineares, dependendo do ponto de impacto e da geometria dos corpos, efeitos rotacionais surgem da colisão. A análise desse processo é bem mais complexa do que aquela feita para movimento linear.

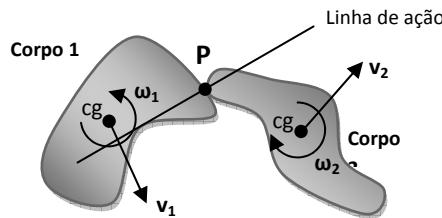


Figura 39: Colisão entre corpos rígidos

Utilizando a Figura 39, vamos supor que dois corpos (1) e (2) sofram uma colisão em um ponto P, e que tenham velocidades lineares v_1 e v_2 , respectivamente, e velocidades angulares ω_1 e ω_2 . Dessa situação, podemos escrever as seguintes relações:

$$\mathbf{v}_p = \mathbf{v}_g + (\boldsymbol{\omega} \times \mathbf{r})$$

Equação 10: Velocidade do ponto de colisão

Nessa relação, \mathbf{r} é o vetor que tem início no centro de gravidade do corpo e fim no ponto de contato P. Usando essa expressão, podem-se escrever as equações de velocidade linear para os corpos envolvidos, considerando o impulso.

$$\text{Corpo 1: } \mathbf{v}_{1g+} + (\boldsymbol{\omega}_{1+} \times \mathbf{r}_1) = \frac{\mathbf{J}}{m_1} + \mathbf{v}_{1g-} + (\boldsymbol{\omega}_{1-} \times \mathbf{r}_1)$$

$$\text{Corpo 2: } \mathbf{v}_{2g+} + (\boldsymbol{\omega}_{2+} \times \mathbf{r}_2) = -\frac{\mathbf{J}}{m_2} + \mathbf{v}_{2g-} + (\boldsymbol{\omega}_{2-} \times \mathbf{r}_2)$$

Equação 11: Movimento Linear dos Corpos (1) e (2)

Fazendo uma análise do movimento angular, podemos adicionar mais duas equações, que torna possível a solução do problema. Sendo assim, escrevemos:

$$\text{Corpo 1: } (\mathbf{r}_1 \times \mathbf{J}) = I_1(\omega_{1+} - \omega_{1-})$$

$$\text{Corpo 2: } (\mathbf{r}_2 \times -\mathbf{J}) = I_2(\omega_{2+} - \omega_{2-})$$

Equação 12: Momento de Inércia Angular dos Corpos (1) e (2)

Combinando as expressões e a definição do coeficiente de restituição, é possível calcular o impulso aplicado nos corpos. O resultado dessa manipulação algébrica gera a equação:

$$J = \frac{-\mathbf{v}_r(e + 1)}{\frac{1}{m_1} + \frac{1}{m_2} + \mathbf{n} \cdot \left[\frac{\mathbf{r}_1 \times \mathbf{n}}{I_1} \right] \times \mathbf{r}_1 + \mathbf{n} \cdot \left[\frac{\mathbf{r}_2 \times \mathbf{n}}{I_2} \right] \times \mathbf{r}_2}$$

Equação 13: Magnitude do Impulso da Colisão

Na fórmula, \mathbf{v}_r é a velocidade relativa dos corpos, \mathbf{n} é o vetor unitário, versor, que passa pelos centros de massa dos objetos. Com a fórmula para J , pode-se calcular as expressões de velocidades para os corpos após a colisão, resultando em:

$$\mathbf{v}_{1+} = \mathbf{v}_{1-} + \frac{J\mathbf{n}}{m_1}, \quad \boldsymbol{\omega}_{1+} = \boldsymbol{\omega}_{1-} + \frac{\mathbf{r}_1 \times J\mathbf{n}}{I_{cg}}$$

$$\mathbf{v}_{2+} = \mathbf{v}_{2-} + \frac{-J\mathbf{n}}{m_1}, \quad \boldsymbol{\omega}_{2+} = \boldsymbol{\omega}_{2-} + \frac{\mathbf{r}_2 \times -J\mathbf{n}}{I_{cg}}$$

Equação 14: Velocidades Lineares e Angulares após colisão

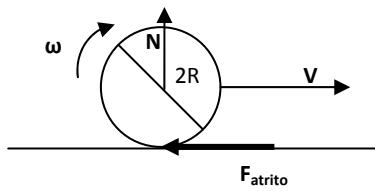
Essas são umas das expressões utilizadas pelo sistema de física para resolver as colisões dos jogos.

3.2.1.1.2.4 Physical Effects

A simulação de efeitos físicos é feita através de uma classe com funções que geram as forças provocadas pelos tais efeitos. Uma dessas funções é a função `gravity(void)`, que devolve o vetor $\mathbf{g} = (0, -\text{value}, 0)$, onde `value` é o modulo do valor da gravidade em m/s^2 . O usuário pode adicionar esse feito apenas somando o vetor \mathbf{g} com vetor força do `RigidBody`.

O Grifo implementa os efeitos de Gravidade, Atrito cinético e estático entre superfícies, o arrasto aerodinâmico, força Magnus, vento.

Os atritos são triviais, pois, sabendo a expressão $F_{at\ estatico} = \mu_{estatico} \mathbf{N}$, $F_{at\ cinetico} = \mu_{cinetico} \mathbf{N}$, onde a \mathbf{N} é a normal ao plano (em muitos casos o peso ($P = m \cdot g$) do objeto), apenas deve-se dar como entrada as informações de μ , *coeficiente de atrito*, e os objetos envolvidos. No caso da rotação de objetos, considerou-se o esquema da figura 40 (no caso objetos esféricos):

**Figura 40:** Atrito

Com isso, temos $\omega = F_{\text{atrito}} \times R$, onde R é o vetor de raio, que vai do centro de gravidade até o ponto de ação da força.

O arrasto aerodinâmico é um efeito mais complexo (PALMER, 2005), o conceito básico dele é que, da mesma forma que o atrito de superfícies, ele atua na direção oposta ao deslocamento. Essa força contém duas componentes, a pressão de arrasto e o arrasto de fricção, ou arrasto de pele (*skin drag*), portanto, $F_D = F_{D,\text{pressure}} + F_{D,\text{friction}}$.

A magnitude da força de arrasto aerodinâmico em um objeto é função de sua geometria, da *densidade do fluido*, ρ , no qual se está atravessando, e o quadrado da velocidade. Normalmente, essa expressão é escrita em função do coeficiente de arrasto, C_D , daí, temos:

$$F_D = \frac{1}{2} \rho v^2 A C_D$$

Equação 15: Força de Arrasto Aerodinâmico

Na equação, A é a área frontal de atuação do arrasto, no caso de esferas $A = \pi r^2$.

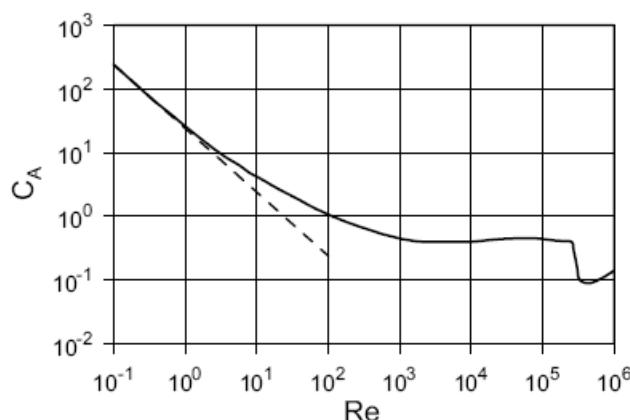
O coeficiente de arrasto é um número adimensional que depende da quantidade conhecida como *Número de Reynolds* (SISSOM, et al., 1979), Re . $C_D = C_D(Re)$.

O número de Reynolds é outra quantidade adimensional utilizada para caracterizar a natureza do fluxo do fluido. Ele é definido em função da densidade do fluido, ρ , da velocidade, v , do comprimento do objeto, L , e da viscosidade do fluido, μ .

$$Re = \frac{\rho v L}{\mu}$$

Equação 16: Número de Reynolds

A relação entre as entidades C_D , ou C_A , e Re são empiricamente estabelecidas pelo gráfico abaixo. Valores de Re pequenos (Aguiar, et al., 2004), são considerados de correspondem ao escoamento laminar, enquanto valores grandes estão associados à formação de turbulências.

Figura 41: Curva $C_A \times Re$

Para pequenos números de Reynolds, $Re \ll 1$, o coeficiente de arrasto é dado pela fórmula de

$$\text{Stokes}, C_D = \frac{24}{Re}$$

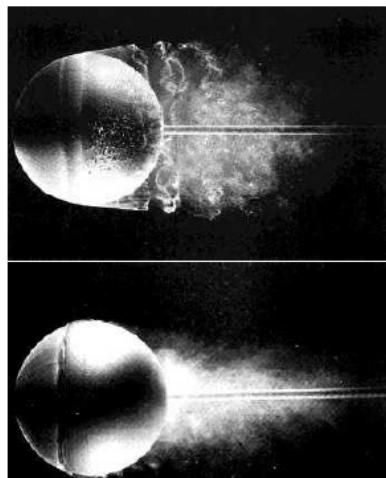


Figura 42: Separação da Camada Limite de uma Esfera

Neste caso a força de arrasto é linearmente proporcional à velocidade. A linha tracejada na Figura 39 mostra a fórmula de Stokes, e vemos que ela é acurada até $Re \ll 1$. Para uma grande faixa de valores de Re , entre aproximadamente 10^3 e 3×10^5 , o coeficiente de arrasto é praticamente constante, mantendo-se em torno de $C_D = 0,5$.

Conseqüentemente, nesta região a força de arrasto é proporcional ao quadrado da velocidade. O aspecto mais curioso da figura 39 é a queda abrupta de C_D (por um fator da ordem de 5) em torno de $Re = 3 \times 10^5$. Esta redução drástica da resistência do ar é chamada de crise do arrasto.

Com esta fórmula é fácil verificar que o arrasto linear ($Re \ll 1$) só ocorre para velocidades irrisórias, bem menores que 0,1 mm/s. Ou seja, a resistência proporcional à velocidade, não tem nenhuma importância para o, no caso do jogo exemplo, futebol, nem para qualquer objeto razoavelmente grande movendo-se no ar.

No contexto de futebol, para uma bola a crise do arrasto ocorre em $V \gg 20 \text{ m/s}$. A região onde a resistência do ar é proporcional a V^2 corresponde a velocidades entre 0,1 m/s e 20 m/s. A velocidade máxima que jogadores profissionais conseguem dar à bola de futebol é da ordem

de 30 m/s. Logo, durante uma partida de futebol a bola deve passar várias vezes pelo ponto de crise.

A crise do arrasto ocorre quando a camada limite torna-se turbulenta. A turbulência permite que a camada resista melhor à tendência de separação, e com isso o ponto de descolamento move-se mais para trás da esfera, diminuindo a área da esteira. A parte de baixo da figura 42 mostra a separação de uma camada limite turbulenta. Está aí a origem da crise — a contração da esteira reduz a área da esfera submetida a baixas pressões, e causa uma diminuição da resistência do ar.

Por fim, as expressões o cálculo da força de arrasto são:

$$F_{Dx} = -F_D \frac{v_x}{v}, F_{Dy} = -F_D \frac{v_y}{v}, F_{Dz} = -F_D \frac{v_z}{v}$$

Equação 17: Forma decomposta da força de Arrasto Aerodinâmico

$$F_D = \frac{1}{2} \rho v^2 A C_D \frac{v}{|v|}$$

Equação 18: Forma Vetorial da força de Arrasto Aerodinâmica

O número C_D é obtido utilizando a aproximação abaixo da curva $C_D \times Re$.

$$C_D = \begin{cases} \frac{24}{Re} & , \text{se } Re \leq 1 \\ \frac{28}{Re} & , \text{se } 1 < Re < 103 \\ 0.8 & , \text{se } 103 \leq Re \leq 3 \times 10^5 \\ 0.1 & , \text{se } Re > 3 \times 10^5 \end{cases}$$

Equação 19: Formulação Analítica do Coeficiente de Arrasto

Durante a queda de um objeto, outro efeito que deve ser considerado, é o da velocidade terminal, esse feito é calculado com base na gravidade e na força de arrasto aerodinâmico, obtendo-se:

$$v_{term} = \sqrt{\frac{2mg}{\rho A C_D}}$$

Equação 20: Velocidade Terminal

A passagem de um objeto por um meio viscoso gera efeitos rotacionais, os quais são calculados conforme a expressão mostrada para o atrito, essa rotação gera um efeito conhecido como Efeito, ou Força, Magnus. No contexto de futebol, quando a bola gira em

torno de seu centro, uma força de sustentação (perpendicular à velocidade) passa a agir sobre ela, essa é a força de Magnus. A expressão dela é:

$$F_M = \frac{1}{2} C_s A v^2 \frac{\omega \times v}{|\omega \times v|}$$

Equação 21: Força Magnus

O coeficiente de sustentação C_s é adimensional, o valor depende da geometria do objeto, no caso de esferas, temos $C_s = \frac{r\omega}{v}$, e para cilindro, $C_s = \frac{2\pi r\omega}{v}$. A Figura 43 ilustra o fenômeno.

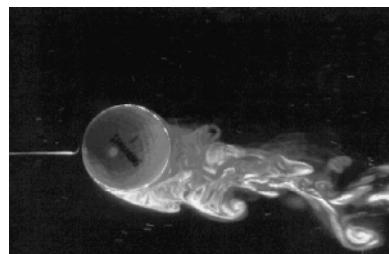


Figura 43: Separação da Camada Limite de Uma Bola Girando

O vento é um efeito simples, ele é apenas um vetor de força apontando para certa direção, pode-se solicitar que o mesmo sofra pequenas oscilações aleatórias.

3.2.1.1.3 Sound/Video System

O sistema de som e vídeo é responsável por manipular toda a multimídia dos jogos. Contudo, apenas a parte de som foi projetada e implementada.

3.2.1.1.3.1 Sistema do Som

O projeto desse sistema foi elaborado de forma a comportar a reprodução de sons 3D e multi-contexto de cenas. O conceito de som 3D não é novo, ele consiste em reproduzir sons, considerando efeitos físicos como a distância da fonte de som até o ouvinte, a orientação e velocidade relativa (efeito Doppler).

A forma de implementar isso é através da amplificação/redução dos som nos canais que melhor simulem o som no espaço. No caso de duas caixas de som, simplificadamente, pode-se dizer que os sons que estiverem à direita do ouvinte (no contexto da cena) serão reproduzidos no canal direito e, por conseguinte, na caixa de som à direita do jogador, o contrário ocorre com os objetos à esquerda.

A sonorização multi-contexto, permite que os sons de uma cena possam ser interrompidos, outra cena com outros sons seja acionada e executada e, quando a cena interrompida for reativada, os sons continuem no ponto em que estavam.

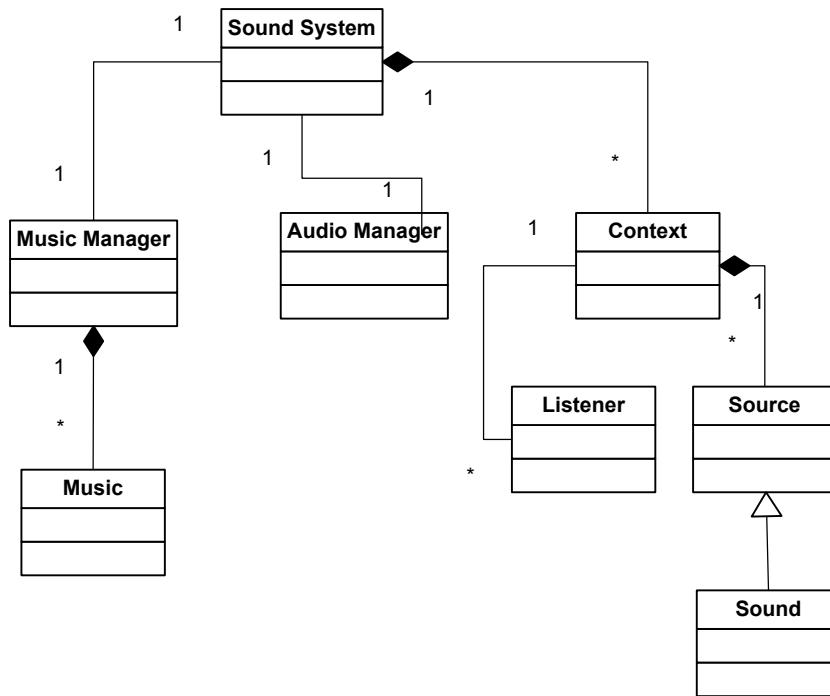


Figura 44: Diagrama de Classes do Sistema de Som

A Figura 44 mostra a arquitetura em alto nível do sistema de som. Nela pode-se perceber que cada contexto pode possuir de um a vários ouvintes (apenas um é o ativo), e uma lista de fontes sonoras. As propriedades básicas dessas classes são posição, velocidade, orientação (em torno do eixo x, para saber se estão de cabeça para baixo), vetor frontal (para saber se estão de frente ou de costas para uma fonte), ganho, timbre, fator de decaimento(Rolloff), taxa de bits, freqüência e nº de canais.

As músicas foram separadas em outra classe, o motivo é que as músicas são sons com um grande tamanho, e, portanto não é possível colocá-las inteiramente na memória das placas de sons, então, deve-se usar streaming com as músicas. Para a utilização de streaming, o Music Manager deve possuir uma thread que de tempos em tempos (dependente da taxa de bits da música) desloca e atualiza o vetor de dados das músicas ativas.

O Audio Manager é a classe com a qual o usuário se comunica para solicitar que um som toque, pare, pause, ou troque de contexto, carregue um som ou uma música.

No contexto de som também são armazenadas as informações como Fator Doppler, velocidade do som, e modelo de distância. Muitos efeitos interessantes são produzidos com a variação desses parâmetros.

3.2.1.1.3.1.1 Modelos de distância

O Grifo implementa dois modelos de distância para som, o de inverso da distância e o inverso da distância cortado.

Para o primeiro, o ganho de uma fonte é calculado como (Src significa Source):

$$G_{dB} = Src.Gain - 20 \log_{10}(1 + Src.RollOff |Listener.pos - Src.pos|)$$

Equação 22: Ganho de uma fonte dependente da distância

Dessa forma, toda fonte contribui para o som final. Em situações de ambientes que possuem muitos sons, os sons mais distantes não são ouvidos e poderiam ser removidas da composição, reduzindo o consumo na placa de som (que muitas vezes é o CPU¹³ quem produz a composição).

O segundo modelo de distância atrela uma distância máxima para o cálculo da composição, dessa forma, se um determinado som estiver a uma distância superior a um dado valor, ele não é contabilizado na composição. O ponto positivo é o descrito anteriormente, contudo, em cenas que existem poucos sons, podem ocorrer situações em que o jogador fica ouvindo o som na zona de rejeição, causando um efeito desagradável.

O Grifo também permite desconsiderar as distâncias e não recalcular o ganho, isso é útil em menus de jogo por exemplo.

3.2.1.1.3.1.2 Efeito Doppler

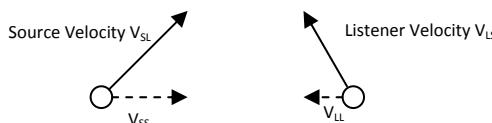


Figura 45: Efeito Doppler

A Figura 45 ilustra a situação em que ocorre o efeito Doppler (HALLIDAY, et al., 2004). Esse efeito faz com que a freqüência dos sons se altere, dependendo das velocidades relativas da fonte e do ouvinte no meio e da velocidade de propagação do som no meio. Para o cálculo da nova freqüência, utiliza-se a expressão abaixo:

¹³ Central Processor Unit

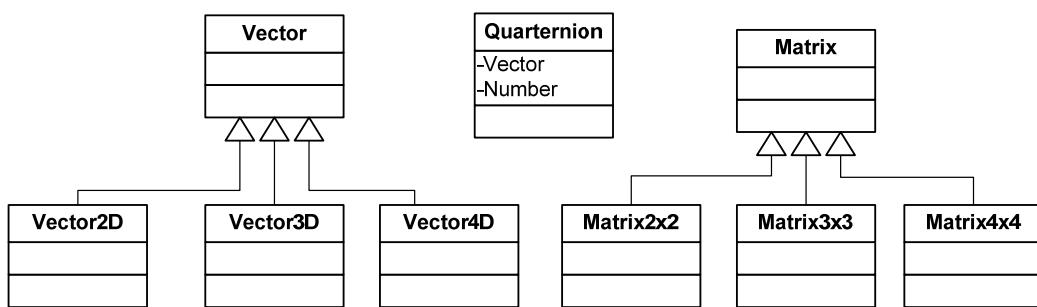
$$f' = f \frac{V_{som} - DF(V_{LL} - V_{SS})}{V_{som} - DF(V_{SS} - V_{LL})}$$

Equação 23: Efeito Doppler

Na expressão, DF é o fator de Doppler e V_{som} é a velocidade do som considerada no meio.

3.2.1.1.4 Math System

O sistema de matemática é o mais simples sistema do Grifo, ele implementa as classes de vetores, quaternions e matrizes. Toda a matemática vetorial é implementada usando os operadores normais da matemática, isso foi possível devido à sobrecarga de operadores.

**Figura 46:** Diagrama de Classes do Sistema Matemático

A Figura 46 mostra as classes do Math System. As classes **Vector4D** e **Matrix4x4** são utilizadas para a manipulação de coordenadas homogêneas (FOLEY, et al., 1997).

3.2.1.1.4.1 Quartenions

Quartenion é uma extensão não comutativa ($a.b = b.a$) dos números complexos. Eles foram inicialmente descritos por (Hamilton, 1853) e muito aplicados na mecânica tridimensional. Os quartenions possuem propriedades muito úteis para as rotações tridimensionais necessárias na computação gráfica e na física. Eles podem ser representados por um vetor de três dimensões e um valor escalar, ou seja, 4 termos, daí quaternion $\mathbf{q} = [s, \mathbf{v}]$. A álgebra dos Quartenions é conhecida como álgebra **H** (de Hamilton), ou Clifford.

A multiplicação de dois quaternions \mathbf{q}_1 e \mathbf{q}_2 é definida como $\mathbf{q}_1 \cdot \mathbf{q}_2 = [s_1, \mathbf{v}_1] \cdot [s_2, \mathbf{v}_2] = [s_1 \cdot s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1 \cdot \mathbf{v}_2 + s_2 \cdot \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2]$. Um quaternion unitário é definido com a condição $s^2 + v_x^2 + v_y^2 + v_z^2 = 1$. Dessa forma, podemos representar uma rotação de um ângulo θ em torno do versor \mathbf{u} como o quaternion unitário $\mathbf{q} = [s, \mathbf{v}] = [\cos(\frac{\theta}{2}), \sin(\frac{\theta}{2}) \mathbf{u}]$, e a rotação inversa \mathbf{q}^{-1} é feita invertendo-se o sinal de s ou \mathbf{v} , mas não ambos.

Para rotacionar um ponto $\mathcal{P} = (x, y, z)$ por um quaternion \mathbf{q} , escreve-se o ponto como $\mathbf{p} = [0, (x, y, z)]$ e efetua-se o produto, gerando $\mathbf{p}' = [0, (x', y', z')] = \mathbf{qpq}^{-1}$ e daí $\mathcal{P}' = (x', y', z')$.

A propriedade de multiplicação é extremamente útil, pois resolve dois problemas da manipulação de matrizes para a rotação. O primeiro, a perda de precisão, e o segundo, do aumento da complexidade de cálculo e armazenamento das matrizes. Por esses motivos os quaternions são cada vez mais usados em jogos.

3.2.1.1.5 Script System

O sistema de script permite ao usuário do Grifo a criação de códigos externos que podem ser modificados ou adicionados mesmo depois de o jogo ser compilado. A vantagem de utilizar esses scripts está na capacidade de extensão que eles permitem.

As linguagens scripts funcionam através do uso de uma máquina virtual, que pode ser entendida como uma *CPU emulada*, simulada, que entende e executa a linguagem script. São exemplos dessa metodologia as linguagens LUA, Java e ActionScript.

Scripts podem ser *interpretados* ou *compilados*. Um script interpretado existe da mesma forma que foi escrito, sendo lido, analisado e executado linha a linha pelo *interpretador*. Isso tende a se tornar um processo lento na execução em tempo real, muitas linguagens scripts compilam automaticamente o script antes de ser interpretado, esse arquivo compilado é chamado de *bytecode*. Outro problema associado à interpretação é a possibilidade dos mesmos serem alterados pelos jogadores, podendo causar erros no jogo, ou facilitar a trapaça.

Os scripts podem ser usados em muitos contextos do desenvolvimento de um jogo, alguns deles, são: servir como inicializador de dados do jogo, provedor de extensões, programação de interfaces, roteiro de jogo, e, principalmente, sistema de inteligência artificial.

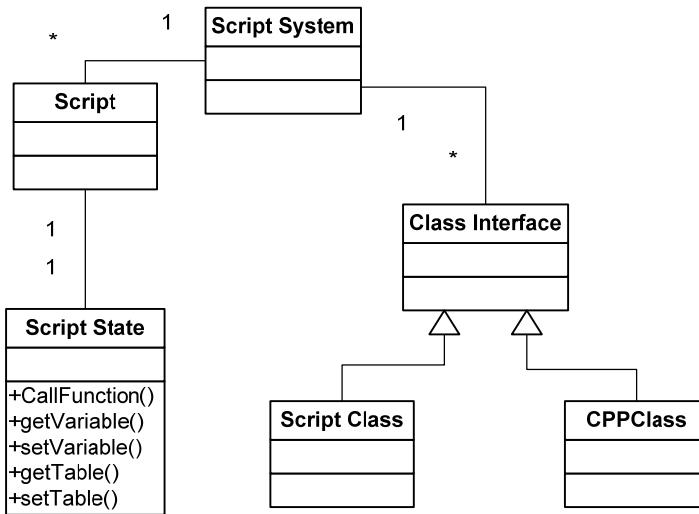


Figura 47: Diagrama de Classes do Sistema de Scripts

O diagrama da Figura 47 mostra a arquitetura do sistema de script, nele é permitida a carga de vários arquivos scripts. Cada script possui um estado, esse estado diz respeito à situação da tabela de variáveis globais e locais, e.g. os valores delas, o estado do heap de memória e outros. Em muitas linguagens script é possível o interfaceamento de classes do script no código do programa e classes do programa com classes do script. Contudo essa parte do sistema não foi desenvolvida devido a problemas com a biblioteca de interface.

3.2.1.1.6 I/O System

O I/O system é responsável por gerenciar os dispositivos de entrada e saída do Grifo, ele é quem captura os eventos do teclado, do mouse, do joystick e repassa para o Resource Manager, que tomará as ações selecionadas para o comando do dispositivo. Ele também sincroniza os eventos do teclado através da classe KeyHelper, usando um procedimento muito simples de indexação de vetor, e.g., se a tecla 'a' é pressionada, durante o tratamento do evento, o vetor de teclas do KeyHelper é indexado na posição `(int)'a'` e setado como 1, se o evento for de ativação e como 0 caso contrário.

Para a aquisição de dados, abertura e salvamento de arquivos o I/O System trabalha como um conversor, transformando arquivos em disco em Objetos do Grifo. Quando é necessário carregar uma imagem, um som ou outro recurso, o I/O lê o arquivo, interpreta o formato de dados e entrega para o sistema usuário o objeto associado àquele tipo de arquivo. O mesmo ocorre em sentido contrário quando se deseja salvar algo. Além disso, ele provê o carregamento e manipulação de arquivos XML, o qual pode carregar e salvar dados específicos do jogo.

3.2.1.2 Scene Manager

O Scene Manager é o sistema responsável por gerenciar, trocar, carregar e descarregar cenas de um jogo. Para gerenciar as cenas, ele trabalha com uma máquina de estados, ou máquina de cenas, que especifica a ordem e como ocorrem as mudanças de cena. É claro que o desenvolvedor pode solicitar diretamente a mudança de cena.

Essa máquina pode ser especificada via programação direta, ou via arquivos de configuração. Na segunda forma, dois arquivos são necessários, o de especificação de estados e transições (XML) e o de programação da transição (script). Essa abordagem ainda não foi implementada, contudo há suporte para a mesma. A Figura 48 apresenta o diagrama de classes do gerenciador.

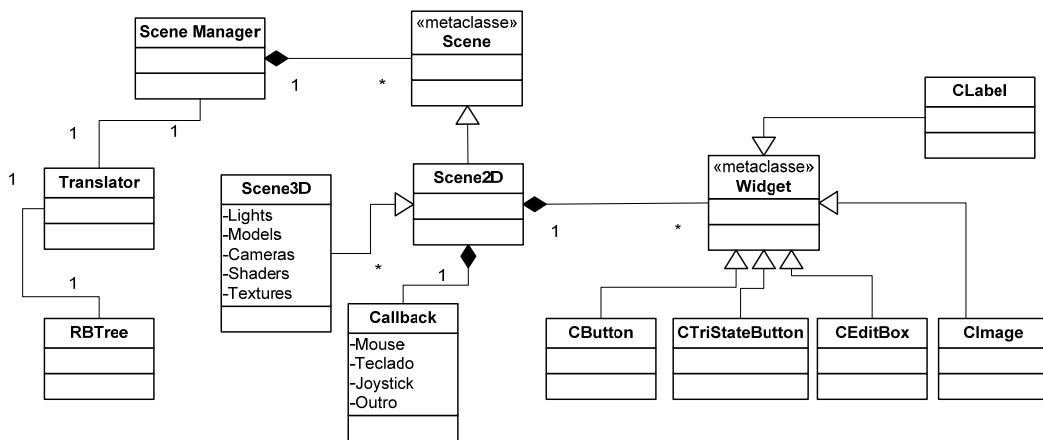


Figura 48: Diagrama de Classes do Scene Manager

Observa-se que existem no projeto duas classes de cenas, as 2D e as 3D. As cenas 2D são compostas de elementos de interface com o usuário e callbacks. As cenas 3D são extensões das cenas 2D, dessa forma, uma cena 3D possui elementos 2D, esses podem ser textos, imagens sobre a tela e outros. No exemplo, o placar é um elemento 2D sobre uma cena 3D.

As cenas 3D possuem cinco listas de elementos, as luzes, os modelos 3D, as câmeras, os shaders e as texturas. O processo de desenho é seqüencial, ativando-se as luzes, selecionando a câmera, ativando os shaders e desenhando os modelos. O Grifo ainda não trabalha com o conceito de grafo de cena, muito usado em aplicativos 3D Java.

3.2.1.2.1 GUI System

O sistema de interface gráfica é o responsável por dar interatividade com o usuário, além de ser entrada de dados como texto e seleção de opções. Cada elemento de interface é chamado de Widget.

Um Widget possui propriedades, métodos e eventos, cada evento está associado, se for utilizado, a um *callback*. Os callbacks são inseridos e acessados de acordo com o tipo de evento, dessa forma, quando um widget, e.g. um botão é clicado, o evento do mouse é passado para o Resource Manager, via I/O System, e encaminhado para a cena corrente. Quando uma cena recebe a informação do evento (Clique do Mouse, Botão Esquerdo, Posição (X,Y)), ela passa a testar todos os callbacks associados ao clique do mouse com botão esquerdo.

Ao testar um callback, verifica-se quem está associado ao mesmo, no caso um botão, e avalia-se se o clique foi efetuado sobre o botão, caso seja verdadeiro, a ação do callback, que pode ter sido programada junto ao programa, ou contida em um script, é executada.

Vários widgets foram projetados e desenvolvidos, a fim de facilitar o desenvolvimento das interfaces. Dentre eles, os mais importantes são os botões, os botões de três estados, as imagens, a caixa de texto e o texto estático (Label). O botão de três estados é um botão que possui três imagens associadas a ele, uma imagem associada ao estado normal, uma ao estado de *mouseOver* (mouse sobre o botão) e uma ao estado de *onClicked* (mouse clicou no botão).

3.2.1.2.2 Multilanguage System

A maioria das aplicações atuais utiliza sistemas multilíngües, logo, é necessária a inserção desse tipo de suporte no desenvolvimento de jogos. O Grifo possui um sistema multilíngüe simples, que utiliza a indexação de termos para traduzir. Esse tipo de sistema utiliza arquivos XML ou arquivos similares aos antigos arquivos de configuração INI. No caso do Grifo utilizou-se o modelo de arquivos INI, desse modo, cada termo lingüístico utilizado no jogo é associado a um número único, e ao se precisar de exibir o termo, o mesmo é chamado pelo seu número.

Vamos supor que existam dois arquivos de configuração, então teríamos:

Tabela 06: Tabela de Termos de Tradução

COD	TERMO	PORUGUES.LNG	ENGLISH.LNG
100	Bem Vindo	100=Bem Vindo	100=Welcome
1001	Passe a bola	1001=Passe a Bola	1001=Pass the Ball

Para exibir uma mensagem no código, teríamos que fazer um chamado do tipo `Imprima(LinguaCorrente(100))` para exibir o bem vindo. A abordagem dificulta a programação, contudo o ganho de simplicidade e eficiência é significativo, além de facilitar a modificação e distribuição. Com a utilização de macros, pode-se melhorar o processo de programação, ficando e.g., `Imprima(LinguaCorrente(BEM_VINDO))`, onde `BEM_VINDO = 100`.

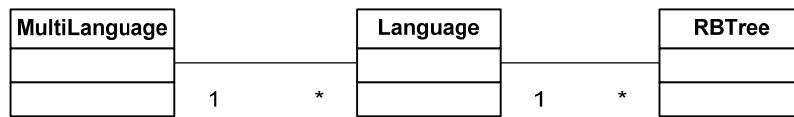


Figura 49: Diagrama de Classes do Tradutor

Para a localização do termo foi utilizada uma árvore vermelha e preta (RBTrees) (CORMEN, et al., 2002), o motivo é que os índices não são contíguos, e se utilizássemos um vetor, teríamos muitas posições vazias, além disso, outros sistemas do Grifo utilizam essa estrutura de dados. As RBTrees são árvores binárias balanceadas de busca, isso as tornam extremamente eficientes nas buscas por termos. A complexidade de inserção não é importante, pois, é um procedimento feito apenas na carga do programa.

Para entendermos esse tipo de árvore, devemos atentar para as propriedades da mesma, pois, são elas que garantem o balanceamento da mesma.

1. Um nó é vermelho ou preto.
2. A raiz é preta.
3. Todas as folhas são pretas (As folhas são os nós nulos).
4. Os filhos de um nó vermelho são pretos.
5. Todo caminho simples, partindo de um nó até as folhas contêm o mesmo número de nós pretos.

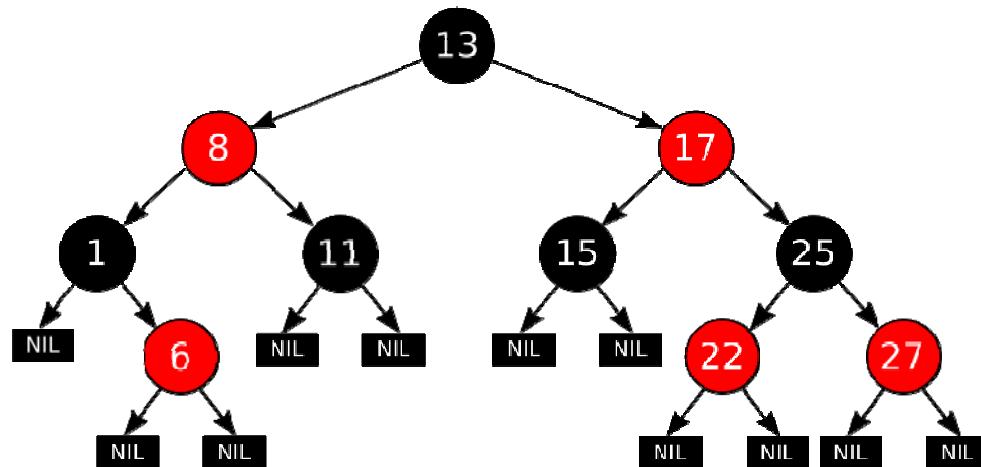


Figura 50: Árvore Vermelha e Preta

Os algoritmos de inserção, remoção e busca, podem ser encontrados em (CORMEN, et al., 2002). A cada solicitação de termo pela chave, é feita a busca na árvore e devolvida o conteúdo do nó procurado.

Terminado a apresentação da Arquitetura do Motor Grifo, pode-se iniciar a análise do desenvolvimento desse.

CAPÍTULO IV

DESENVOLVIMENTO DO GRIFO

Nesse capítulo será discutido o desenvolvimento do Grifo, descrevendo as tecnologias e os recursos utilizados na mesma, além de aspectos específicos do projeto.

A arquitetura do Grifo suporta uma API¹⁴ gráfica genérica, contudo direciona-se o projeto para as APIs OpenGL e DirectX (LUNA, 2003). Para o desenvolvimento da arquitetura Grifo, foi escolhida a API OpenGL 2.0. Dentre os motivos dessa escolha pode-se destacar a facilidade de aprendizagem, a grande disponibilidade de material de estudo, a gratuidade e principalmente a portabilidade e bom desempenho. A API DirectX não foi implementada devido às restrições de tempo.

Além disso, a linguagem de programação utilizada foi o C++ (STROUSTRUP, 1999), por facilitar o processo de projeto, modularização e organização do sistema. Muitas bibliotecas foram utilizadas no projeto, essas serão devidamente citadas ao longo do texto.

4.1 Visão Geral da OpenGL

Segundo (SHREINER, et al., 2005), a OpenGL fornece um conjunto de comandos muito importante para modelagem e visualização de objetos geométricos. Porém, esses comandos são muito primitivos, no sentido de que fornecem um baixo nível de abstração. No caso, toda e qualquer rotina de alto nível para elaboração de desenhos deve ser implementada utilizando tais comandos (a maioria dos comandos é executada diretamente na GPU). Além disso, como a OpenGL não contém funções para gerenciamento de janelas, logo, seria necessário trabalhar com o sistema de janelas disponível no ambiente operacional.

Para facilitar o desenvolvimento das aplicações gráficas, foram criadas bibliotecas que fornecem uma maior abstração, além de proverem funções para a criação de janelas e

¹⁴ Application Program Interface

gerenciamento de eventos. Entre as várias bibliotecas existentes, pode-se considerar que as mais utilizadas são a GLU e a GLUT.

A GLU (OpenGL Utility Library), que é comumente utilizada, é instalada junto com a OpenGL. Essa biblioteca contém uma série de funções que encapsulam comandos OpenGL de mais baixo nível. Entre as várias tarefas que podem ser realizadas utilizando as rotinas da GLU, podemos destacar: a definição de matrizes para projeção e orientação da visualização, as quais permitem o mapeamento de coordenadas entre o espaço de tela e do objeto e o desenho de superfícies quádricas, texturização de quádricas.

Para facilitar o desenvolvimento de interfaces, foi criada a biblioteca GLUT (OpenGL Utility Toolkit), que consiste em um toolkit independente de plataforma, que inclui alguns elementos de interface gráfica com o usuário (GUI - Graphical User Interface). Entre as funcionalidades disponíveis na GLUT estão, a criação de janelas e menus pop-up, além do gerenciamento de eventos de mouse e teclado. A GLUT não é de domínio público, mas é gratuita, e tem como um de seus objetivos ocultar a complexidade das APIs dos vários sistemas de janelas existentes hoje em dia. Assim, programas escritos usando a GLUT são portáveis, podendo ser compilados e executados em diferentes ambientes.

4.2 Desenvolvimento das Camadas

4.2.1 Visão Geral

O projeto Grifo foi desenvolvido, utilizando os recursos da linguagem C++ e os benefícios provindos da orientação a objetos (BOOCH, 1994). O diagrama de execução da Figura 51 mostra o funcionamento dos subsistemas do Grifo.

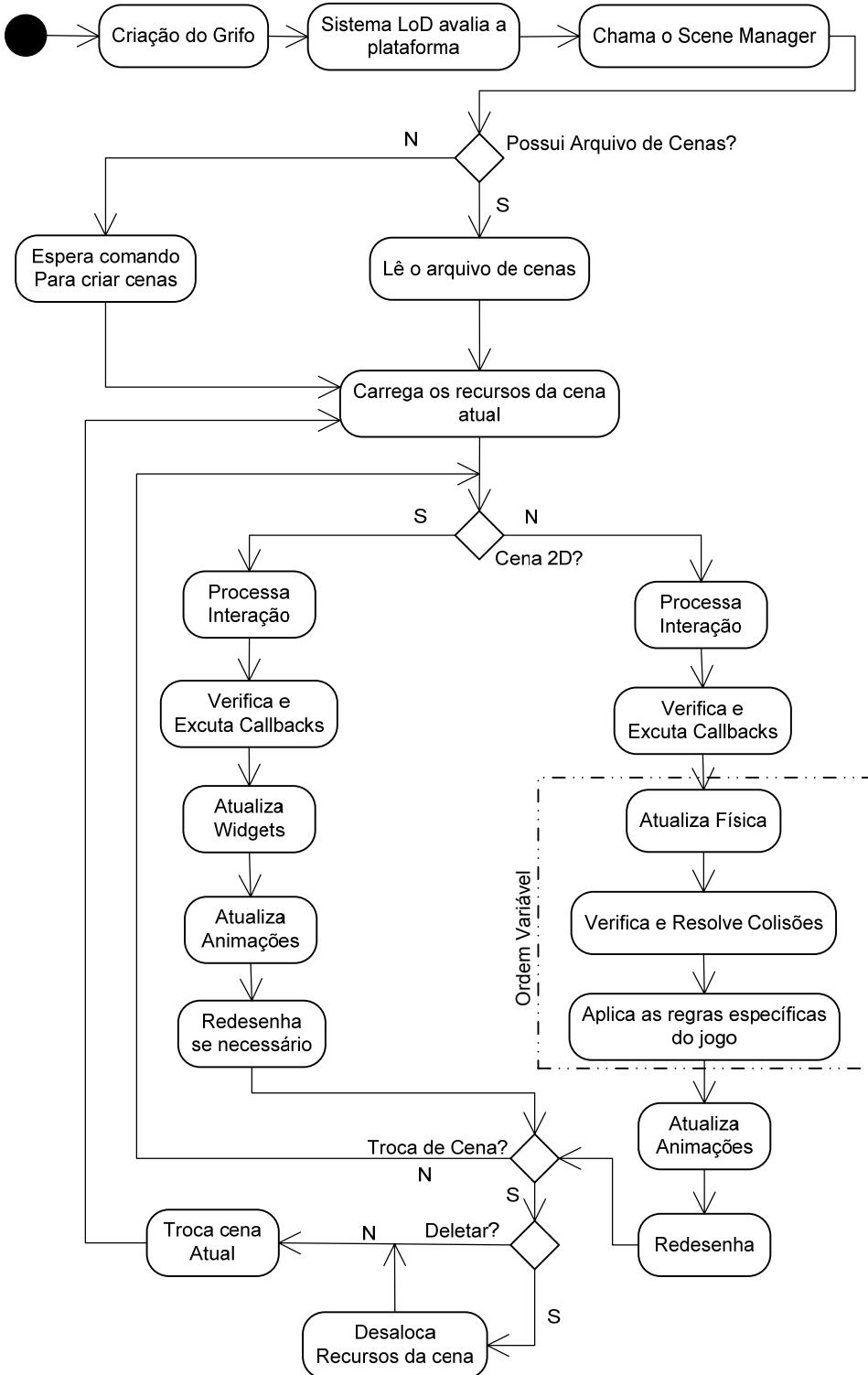


Figura 51: Diagrama de Atividades do Grifo

No diagrama, após a instanciação do motor, o sistema global de LoD analisa toda a arquitetura de hardware e software da estação cliente (o computador do usuário) e prontamente sinaliza os recursos disponíveis. Após tal avaliação, o Grifo aciona o Scene Manager, que, se houver um arquivo de cenas, o importará. Caso contrário, ele esperará o sistema usuário solicitar os

recursos. Para o carregamento dos recursos de uma cena, o I/O System é acionado para realizar tal procedimento.

Quando a carga é concluída, a cena está pronta para ser exibida. Nessa fase é verificado o tipo de cena a exibir, para isso usa-se *downcast*. É importante perceber que é feito um tratamento diferenciado para cenas puramente 2D e cenas 3D. A motivação para tal se atém ao fato de que as cenas exclusivamente 2D, e.g. cenas de menus, podem ser atualizadas, renderizadas somente quando algum evento ocorre, por exemplo, quando o jogador clica em um botão. Isso reduz o consumo da placa de vídeo, além de permitir que dados de cenas mais complexas fiquem acomodados na memória da placa. As cenas 3D necessitam de renderização constante.

No processo principal (*Main Loop*) de execução do Grifo, são analisadas as interações do usuário, logo após, são feitas as verificações e execuções dos callbacks cabíveis. Para o caso de cenas 3D que possuem simulação de física, o sistema usuário pode fazer as chamadas para atualização da física e resolução da colisão de objetos, além de poder criar as regras específicas (SALEM, et al., 2004) do sistema de interseção. A Figura 52 mostra o diagrama de seqüência do processo de carga de uma cena.

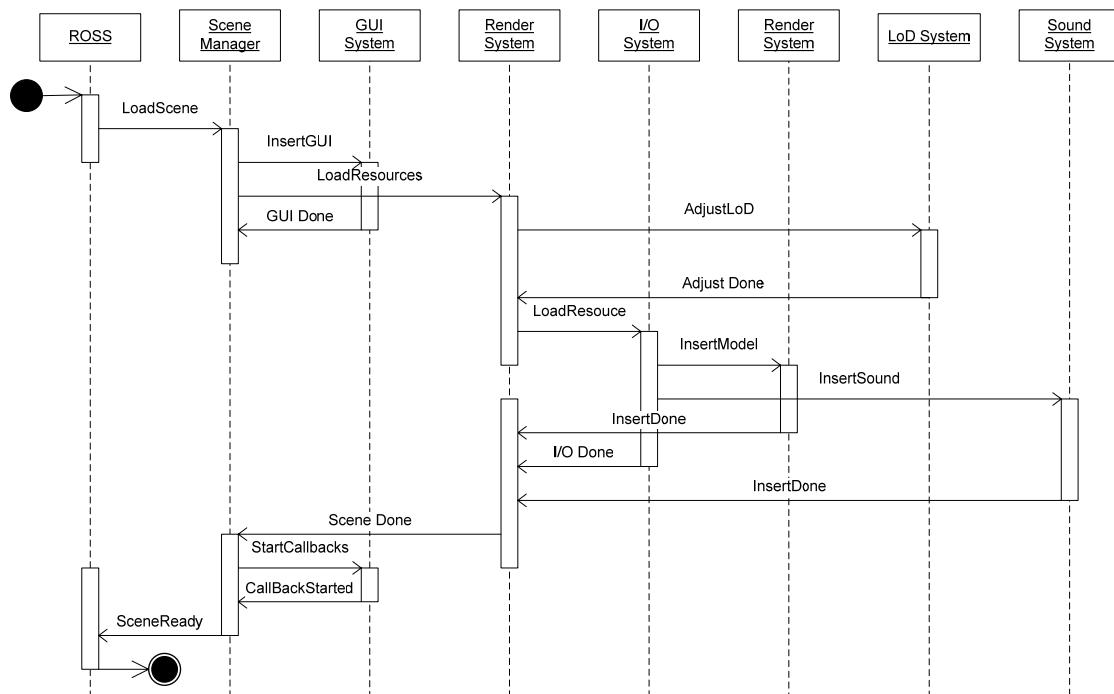


Figura 52: Diagrama de Seqüência do Grifo

4.2.2 Resource Manager

O gerenciador de recursos foi implementado com o auxílio de várias bibliotecas livres, disponíveis na internet. O critério de escolha dessas bibliotecas foi, em primeira instância, a tradição, ou seja, escolheu-se bibliotecas que já estão a muito tempo disponíveis, além disso, os comentários de utilizadores das mesmas foram considerados, e por fim, a gratuidade, pois o projeto não conta com fundos orçamentários.

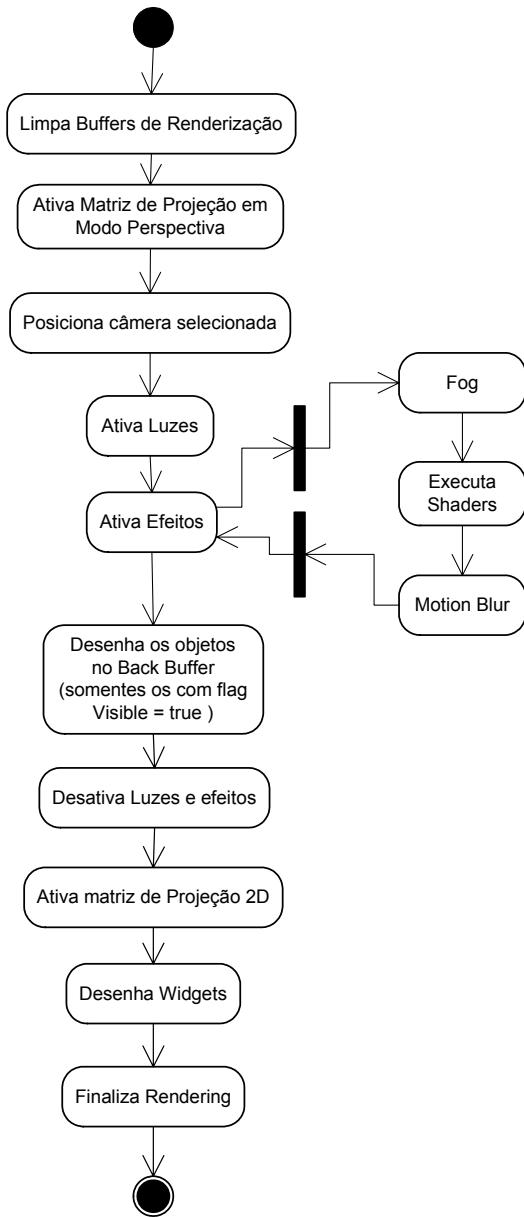


Figura 53: Diagrama de Fluxo do sistema de renderização

O sistema de renderização utiliza as bibliotecas da OpenGL, as quais, fornecem a maioria dos recursos necessários para o funcionamento de cada segmento desse sistema.

A Figura 53 apresenta a seqüência de ações tomadas durante o procedimento de renderização. Esse procedimento é o utilizado para o desenho de uma cena 3D, a qual pode possuir uma parte 2D, observe que há uma mudança de sistema de projeção, a fim de possibilitar o desenho da interface sobre o desenho 3D.

Para o funcionamento do sistema de shaders, foi incorporada a biblioteca GLEW (ROST, 2004). A GLEW é uma biblioteca livre de programação de Vertex Shaders e Pixel, ou também Fragment, Shaders. A linguagem de programação da GLEW é a *OpenGL Shading Language (GLSL)*. A GLSL foi aprovada como uma extensão pelo ARB (Architecture Review Board) em 2003 (ROST, 2004), porém, somente na versão 2.0 da API OpenGL é que a linguagem passou a ser, oficialmente, parte da biblioteca.

A sintaxe da mesma é muito similar à da linguagem C, mas há diferenças e restrições significativas. Primeiramente, existem dois tipos de programas, os de vértice e os de fragmento, apesar de terem a sintaxe igual, o funcionamento é diferente. Outra característica

notável é a introdução de tipos de dados vetoriais como, vec4, vetor float de quatro componentes, e matrizes, mat4, matrix 4x4 e outros. Além disso, diversas variáveis do sistema de renderização são disponibilizadas nos programas, como a matriz de Modelagem e Visualização, a matriz de Projeção, posição das luzes e outros. A GLEW permite que dados do programa sejam passados para o programa GLSL e já possui um compilador de shaders integrado à biblioteca. O diagrama de seqüência (Figura 54) mostra o procedimento de carga de um programa GLSL.

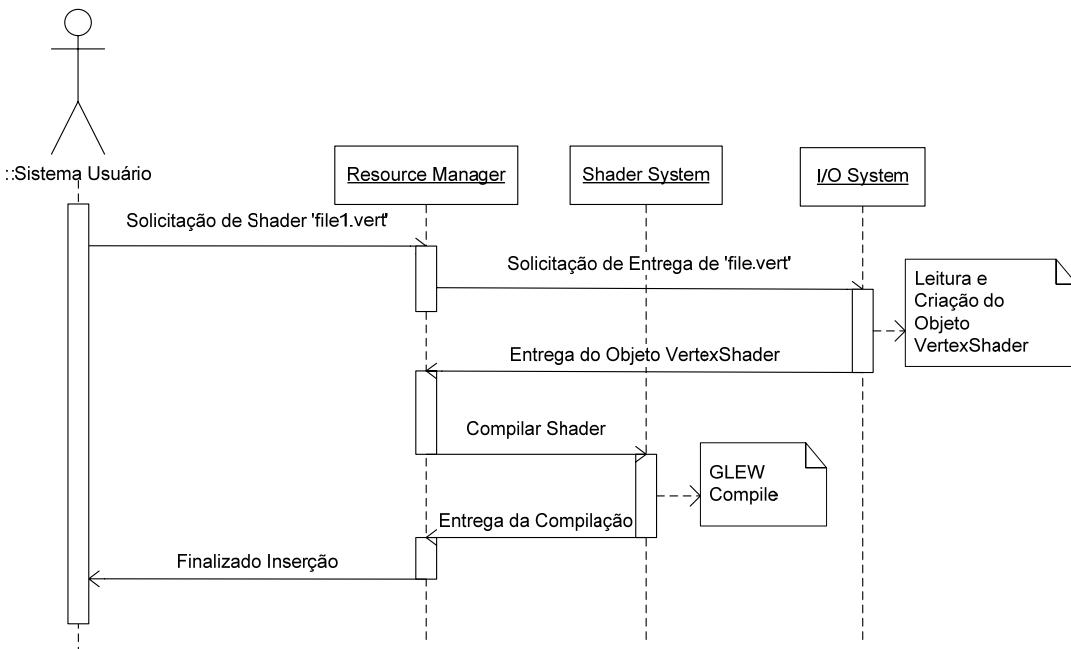


Figura 54: Diagrama de Seqüência do Compilador de Shaders

O desenvolvimento do sistema de física e de matemática teve como principal objetivo ser extremamente eficiente, desse modo, a utilização de C++ e ponteiros foram de extrema valia. Para o sistema de física foram desenvolvidos vários algoritmos resolução de colisão, tanto linear quanto angular, e além deles, também os algoritmos de solução de equações diferenciais ordinárias, os quais já foram mencionados.

Na Tabela 07 é mostrado o algoritmo de checagem de colisão linear, sem considerar efeitos de rotação, de duas esferas e o algoritmo de aplicação de impulso.

Tabela 07: Algoritmo de Verificação de Colisão de Esferas e de Impulso

```

int CheckForCollision ( SphereRigidBody *body1, SphereRigidBody *body2)
{
    Vector d;
    float r;
    int retval = 0;
    float s;
    Vector v1, v2;
    float Vm;

    r = body1->fLength/2 + body2->fLength/2;
    d = body1->vPosition - body2->vPosition;
    s = d.Magnitude() - r;

    d.Normalize();
    vCollisionNormal = d;

    v1 = body1->vVelocity;
    v2 = body2->vVelocity;
    vRelativeVelocity = v1 - v2;

    Vm = vRelativeVelocity * vCollisionNormal;
    if (fabs(s) <= ctol) && (Vm < 0.0) {
        retval = 1; // colisão
    } else if (s < -ctl) {
        retval = -1; // penetração
    } else {
        retval = 0; // sem colisão
    }

    return retval;
}

void ApplyImpulse(SphereRigidBody *body1, SphereRigidBody *body2)
{
    float j;

    j = (-1+fCr) * (vRelativeVelocity * vCollisionNormal) / 
        ( (vCollisionNormal*vCollisionNormal) *
        (1/body1->fMass + 1/body2->fMass) );

    body1->vVelocity += (j * vCollisionNormal) / body1->fMass;
    body2->vVelocity -= (j * vCollisionNormal) / body2->fMass;
}

```

A função `CheckForCollision` é utilizada para o cálculo de colisão de esferas, ela calcula as distâncias dos centros e determina se houve colisão. A função `ApplyImpulse` atua calculando o impulso linear de uma colisão de esferas, o fator `fCr` é o coeficiente de restituição. Observe que as variáveis vetoriais são tratadas com o operador padrão de soma/multiplicação, para isso utilizou-se a sobrecarga de operadores proporcionada pela linguagem C++. Além disso, para o exemplo, as variáveis `vRelativeVelocity` e `vCollisionNormal` são globais, ou pertencem a um escopo maior.

Outro algoritmo importante é o de resolução de colisão, na Tabela 08 é mostrado um exemplo do algoritmo que resolve uma colisão linear.

Tabela 08: Algoritmo de Resolução de Colisão

```
void StepSimulation(float dt, SphereRigidBody *body1, SphereRigidBody *body2)
{
    float dtime = dt;
    bool tryAgain = true;
    int check = 0;
    bool didPen = false;
    int count = 0;

    while(tryAgain && dtime > ctol)
    {
        tryAgain = false;

        UpdateBody(&body1, dtime);
        UpdateBody(&body2, dtime);

        check = CheckForCollision(body1, body2);
        if(check == PENETRATING)
        {
            dtime = dtime/2;
            tryAgain = true;
            didPen = true;
            ApplyImpulse(body1, body2);
        } else if(check == COLLISION) {
            ApplyImpulse(body1, body2);
        }
    }
}
```

A condição de parada é um valor de tolerância de penetração (**ctol**), ou seja, pode ocorrer de um objeto ficar um pouco sobre outro, isso ocorre, pois, dependendo do passo de tempo, ou intervalo de integração, especificado (**dt**), e considerando que a bola pode mover-se muito rápido, esse efeito é possível, para evitar isso, a condição de parada deve ser especificada. Além disso, os modelos geométricos possuem uma pequena folga em relação ao modelo gráfico, fazendo com que essa abordagem não cause efeitos visuais incoerentes.

Na simulação em tempo real dos fenômenos físicos geram-se várias equações diferenciais, que precisam ser resolvidas por algum método numérico. O sistema de física conta com vários algoritmos que se dispõe a fazê-lo. Dentre eles, um dos mais simples é o de Euler básico com adaptação de passo (BOURG, 2002). Essa adaptação foge do algoritmo tradicional, mas mesmo assim usa-se tal abordagem porque durante um jogo o tempo de renderização pode aumentar devido a uma cena se tornar mais complexa, causando um atraso no simulador de física. Para compensar o problema, e algumas vezes eliminar grandes erros de aproximação, usa-se o algoritmo da Tabela 09.

Tabela 09: Algoritmo de Atualização (EDO)

```

void UpdateBody(float dt)
{
    float F; // Força Total
    float A; // Aceleração
    float Vnew; // Nova velocidade
    float Snew; // Nova posição
    float dtnew; // Novo Passo de integração
    float V1, V2; // Velocidades Temporárias
    float et; // erro de aproximação

    F = (T - (C * V)); // C é o coeficiente de arrasto, T é o Arrasto
    A = F / M; // M é a massa
    V1 = V + A * dt;

    F = (T - (C * V));
    A = F / M;
    V2 = V + A * (dt/2);

    F = (T - (C * V2));
    A = F / M;
    V2 = V2 + A * (dt/2);

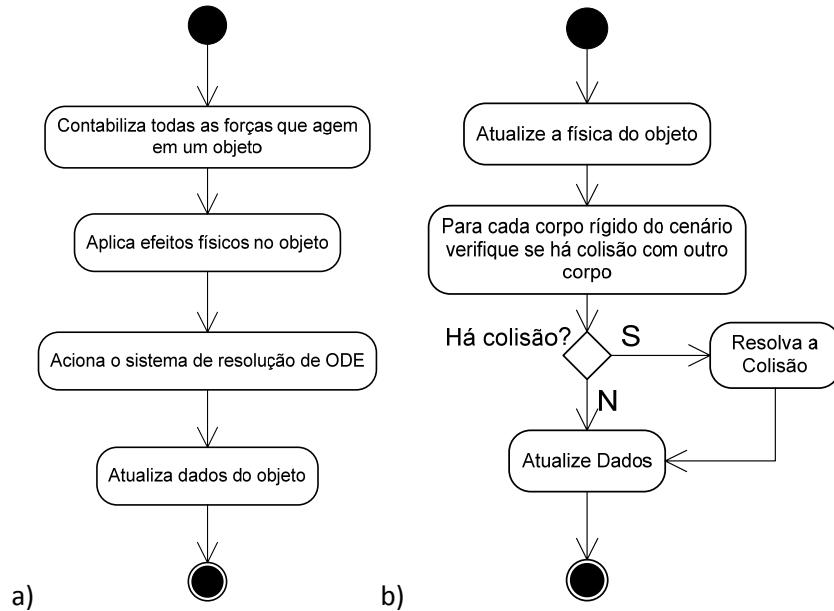
    et = absf(V1 - V2);

    dtnew = dt * SQRT(eto/et); // eto é a tolerancia de erro
                                // dtnew pode ser associado diretamente ao dt medido
    if (dtnew < dt)
    {
        F = (T - (C * V));
        A = F / M;
        Vnew = V + A * dtnew;
        Snew = S + Vnew * dtnew;
    } else {
        Vnew = V1;
        Snew = S + Vnew * dt;
    }

    V = Vnew;
    S = Snew;
}

```

Esse algoritmo é uma simplificação do processo, contudo, demonstra o funcionamento integrado das funções da física conforme mostrado na Figura 55. Algumas variáveis são globais ou de um escopo superior, sendo, portanto disponíveis para as demais funções da física.

**Figura 55:** Diagrama de Fluxo do Sistema de Física

As Figuras 55a e 55b mostram o fluxograma de execução do sistema de física, na Figura 55b, o primeiro procedimento (*Atualize a física do objeto*) consiste no fluxograma 55a, dessa forma, primeiro é feita a simulação do sistema e depois são resolvidas as colisões elemento a elemento.

Dentre os sistemas implementados, o que mais necessita de bibliotecas externas é o I/O System, pois, existe uma grande quantidade de formatos de imagem, modelos 3D, sons e outros. Por princípio, o Grifo suporta qualquer formato, contudo, não há necessidade de suportar muitos formatos, isso apenas criaria um sistema de entrada e saída extremamente grande, complexo e muito suscetível a erros. Desse modo, apenas alguns formatos foram desenvolvidos. A Tabela 10 mostra tais suportes.

Tabela 10: Formatos de dados suportados pelo motor

Tipo de Recurso	Formatos Suportados	Bibliotecas externas
Imagens e texturas	JPEG TGA BMP	Libjpeg Interna Interna
Modelos 3D	OBJ	Interna
Sons	WAV	Interna
Músicas	OGG	Libogg, Libvorbis
Scripts	LUA	Liblua
Shaders	VERT FRAG	GLEW GLEW
Materiais	MAT	Interna
Configuração	INI XML	Interna Libxml

As implementações internas foram feitas baseando-se no formato do arquivo especificado

pelo desenvolvedor do mesmo. Na Figura 56 é mostrado o formato de arquivo de som WAV, com a especificação obtida de (Wilson, 2003). Um detalhe importante desse arquivo é que se deve prestar atenção até mesmo no modo de ordenação de bits, pois, trechos do arquivo são little endian e outros big endian.

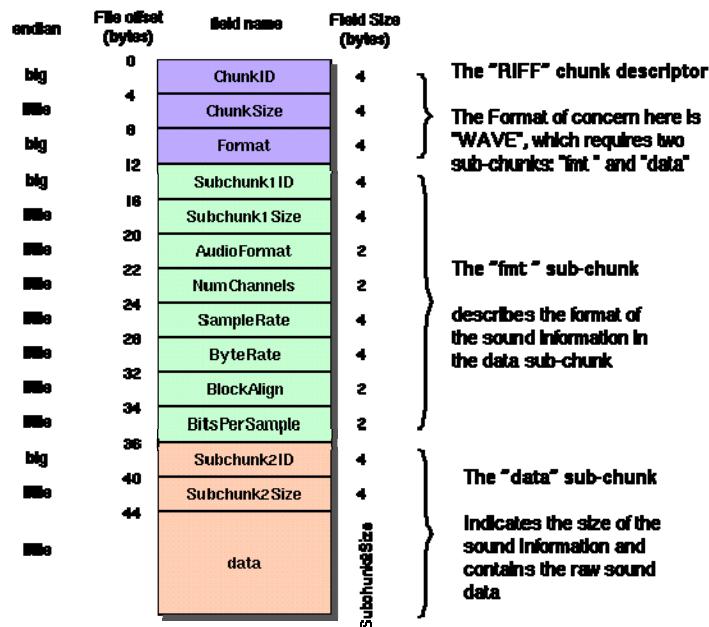


Figura 56: Formato do Arquivo WAV

Para todos os formatos suportados que utilizam bibliotecas externas foram criadas classes de abstração de dados, chamadas de Wrappers, que encapsulam o funcionamento e permite a migração das bibliotecas.

Além da abertura de arquivo, o sistema de I/O manipula os dispositivos de entrada e saída. Essa manipulação foi realizada com o auxílio da biblioteca GLUT, que facilita enormemente esse trabalho.

O sistema de som do Grifo foi desenvolvido utilizando a biblioteca OpenAL (Creative Technology, 2005), a qual suporta a maioria das funcionalidades especificadas no projeto. Contudo, a manipulação de dados como, por exemplo, a leitura e sincronização, não são realizadas pela OpenAL. Desse modo, o Sound System desempenha o papel de gerenciador e sincronizador dos recursos sonoros.

A maior dificuldade desse sistema foi a criação do subsistema de reprodução de Música. Nele, selecionou-se o padrão de música OGG como formato de armazenamento. Para que isso fosse possível, no sistema de I/O adicionou-se suporte ao padrão OGG, através das bibliotecas libogg, libvorbis e libvorbisfile.

Conforme discutido no capítulo 3, devido ao grande tamanho dos arquivos de som, a técnica de streaming foi escolhida para a reprodução de músicas. Para sincronizar a atualização das músicas no banco de músicas do sistema, foi criado uma thread de controle. Para o funcionamento multiplataforma das threads, utilizou-se a biblioteca `pthreads`, que em sistemas Unix, está presente em todas as distribuições. A Figura 57 ilustra o funcionamento do streaming.

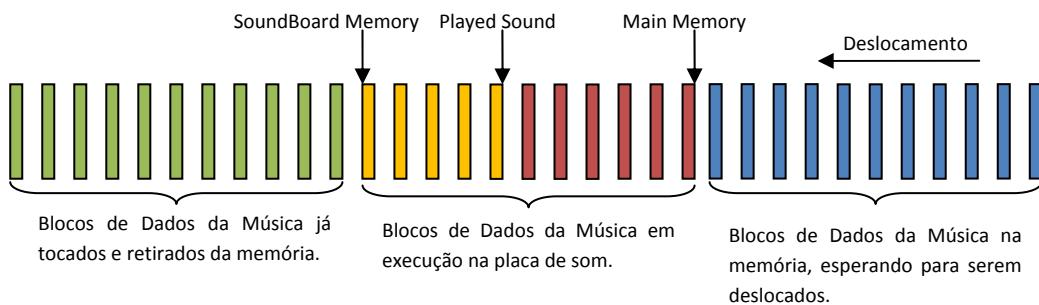


Figura 57: Funcionamento do Streaming

Observe na figura que a música é carregada por blocos, ou seja, durante a execução, trechos contíguos da música são carregados na memória principal (parte azul), e assim que um trecho anterior é finalizado na placa de som (parte amarela e vermelha), a parte carregada na memória é movida para a memória da placa de som (ou simplesmente o apontador da placa é movido) e reproduzida. Nesse processo, a parte anteriormente concluída é descarregada (parte verde) e uma nova parte é carregada.

O sistema de scripts foi desenvolvido com o uso da biblioteca `liblua`, que permite o interfaceamento da linguagem LUA (JUNG, et al., 2006), desenvolvida pela PUC-Rio no laboratório da TecGraph (Ierusalimschy, et al., 2006). A LUA vem ganhando muita popularidade pelos desenvolvedores de jogos, até mesmo grandes empresas do ramo utilizaram a linguagem em seus títulos. Essa popularidade é devido ao grande poder de expressão, à rápida execução, se comparada com as outras linguagens, à simplicidade de uso, à flexibilidade, à portabilidade e ser de código aberto.

A LUA trabalha com apenas oito tipos de dados, Number, Nil, String, Boolean, Function, Table e outros. O tipo de dados mais interessante é a Table, através dela pode-se definir vários tipos e dados. Outra característica curiosa dela é que sua indexação pode ser numérica, ou por string, isso significa que chamadas do tipo `test_table["carro1"]["x"] = 10.5` e `test_table[3.14] = "PI"` são perfeitamente válidas.

A interface com o C/C++ é feita através do uso de pilha e funções de conversão de formato. Para exemplificar esse funcionamento utilizaremos o exemplo da Tabela 11.

Tabela 11: Exemplo de chamada de variável LUA

```
-- Código LUA
-- Variaveis globais
name = "spiderman"
age = 29
...
// Código C++
char *namec = (char*) Script->getVariable("name","String");
std::string name = namec;
free(namec); // Para não criar memory leak

// Código executado por getVariable

void *getVariable( const std::string &var_name, const std::string &type)
{
    char *tempString;
    void *ret = NULL;
    ...
    lua_settop( this->State, 0); // Aponta o topo da pilha
    ...
    if( type == "String" ) {
        lua_getglobal( this->State, var_name.c_str() );
        if( isstring( this->State, 1 ) // Testa se é realmente string
            tempString = (char*)lua_tostring( this->State, 1);
        else
            tempString = NULL;
        ret = (void*) (tempString);
    } else ...
    ...
    lua_pop( this->State, 1); // Desempilha
    ...
    return ret;
}
```

Para acessar no programa C++ a variável `name`, utiliza-se a função do wrapper `getVariable`, que chama as funções relativas da liblua. Ao chamar tais funções, o sistema de script verifica o tipo de variável e realiza a conversão, *cast*, nas mesmas, desempilha os dados e retorna o valor solicitado. Deve-se ter cuidado para não gerar *memory leak*, vazamento de memória, pois tudo é alocado dinamicamente.

A função `lua_settop` aponta o início da pilha, as *funções de get* empilham os dados, e as *funções de cast* convertem o dado do escopo da LUA para o tipo nativo C++. Após isso, os itens devem ser desempilhados com a função `lua_pop`.

A LUA permite a criação de funções com saídas múltiplas, isso significa que pode haver vários valores de retorno de uma função, assim como em MatLab (HANSELMAN, et al., 2003). Da mesma forma que acontece com as variáveis, os parâmetros de entrada são empilhados no código C++ e desempilhados, automaticamente, no código LUA. Além disso, os valores de

retorno da função são empilhados na pilha. Na Tabela 12 é mostrado um exemplo desse funcionamento.

Tabela 12: Exemplo de chamada de função LUA

```
-- Código LUA
function makeVector( x, y )
    return x,y
end;
...
// Código C++
int retX, retY;
Script->CallFunction( "makeVector", 2, 2, "ii-ii", 10, 20, &retX, &retY);

// Código de CallFunction
void CallFunction( const std::string name, int nargs, int nrets,
                  const std::string types, ... )
{
    // types: (IN-OUT) i = inteiro, f = float, s = string ...
    va_list argptr;

    va_start(argptr, nargs);

    ...

    int *tempIntArgs, pIntArgs = 0;
    ...
    lua_settop( this->State, 0 ); // Aponta o topo da pilha
    lua_getglobal( this->State, name );
    ...

    for( register int i = 0; i < nargs; i++ ) {
        if( types[i] == 'i' ) {
            tempIntArgs[pIntArgs] = va_arg(argptr,int);
            lua_pushnumber( this->State, tempIntArgs[pIntArgs] );
            pIntArgs++;
        } else ...
        ...
    }

    lua_call( this->State, nargs, nrets ); // Estado, nº de entradas, nº de saídas
    ...
    *tempIntRet[i] = (int)lua_tonumber( this->State, 1 );
    lua_pop( this->State, 1 );
    ...
    va_end(argptr);
}
```

É importante notar que a função `CallFunction` é uma função de parâmetros variáveis, dessa forma, para sua implementação foi necessário a utilização do cabeçalho padrão `<stdarg.h>`. O funcionamento dessas funções é explicado em (SCHILD, 1997). O script manager deve ler todos os argumentos necessários para a chamada da função, apontar o início da pilha, empilhar a função, empilhar os argumentos e chamar a função. Após isso, os argumentos serão desempilhados e os resultados empilhados, prosseguindo, têm de desempilhar os resultados e passá-los ao usuário.

Além disso, a LUA permite ainda inserir funções implementadas em C++ dentro do código LUA e pode-se trabalhar totalmente integrado com as classes C++, para tal, deve-se usar a biblioteca luabind.

4.2.3 Scene Manager

Para o desenvolvimento do sistema de gerenciamento de cenas, foram criadas apenas as classes relevantes, descritas no capítulo 3. A única parte desse subsistema que realmente necessitou de uma melhor análise foi o GUI System.

O GUI System é responsável por prover os Widgets, que são os elementos de interface com o usuário. Para que os mesmos funcionassem de acordo com o padrão, e.g. elementos GUI do Windows, foi feito um estudo de como eles fazem a detecção de contato e como são suas máquinas de estados.

Para a detecção de contato, foi considerado que os Widgets teriam regiões de contato retangulares, dessa forma, o cálculo de contato consiste em determinar apenas a pertinência em uma região. A Figura 58 ilustra esse processo:

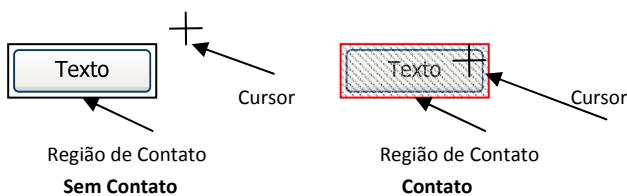


Figura 58: Teste de pertinência dos widgets

Quando o usuário interage com a tela, as coordenadas passadas são coordenadas de Tela, e não as coordenadas do sistema de renderização. Para se utilizar tais informações é necessário converter as bases do sistema. A Figura 59 mostra esse processo.

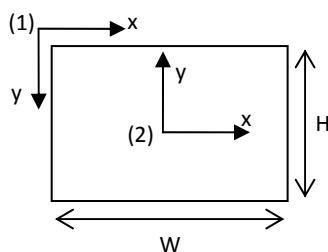


Figura 59: Transformação de sistemas de coordenadas

Na Figura, temos dois sistemas de coordenadas, (1) e (2), uma tela de visualização (2) com dimensões (no sistema (1)) W e H. Para a conversão das coordenadas de (1) para (2), pois todos os widgets estão no sistema 2, devemos aplicar as seguintes transformações, em coordenadas homogêneas:

$$\begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -\frac{W}{2} \\ 0 & -1 & \frac{H}{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 - \frac{W}{2} \\ \frac{H}{2} - y_1 \\ 1 \end{pmatrix}$$

Equação 24: Transformação de sistemas de coordenadas

Para que o widget interaja com o usuário, ele deve ser programado para reagir de acordo com as necessidades do usuário. Isso é feito com auxílio das máquinas de estados, que fornecem meios de representarmos e desenvolvermos reações dadas às entradas, ou seja, as interações do usuário, e às reações anteriores.

Após uma análise do funcionamento dos botões, desenvolveu-se a máquina de estados da Figura 60. Essa máquina mostra o funcionamento do botão e as ações que se desencadeiam de cada interação (no caso os callbacks e redesenho), o símbolo (!) significa o operador de negação lógica.

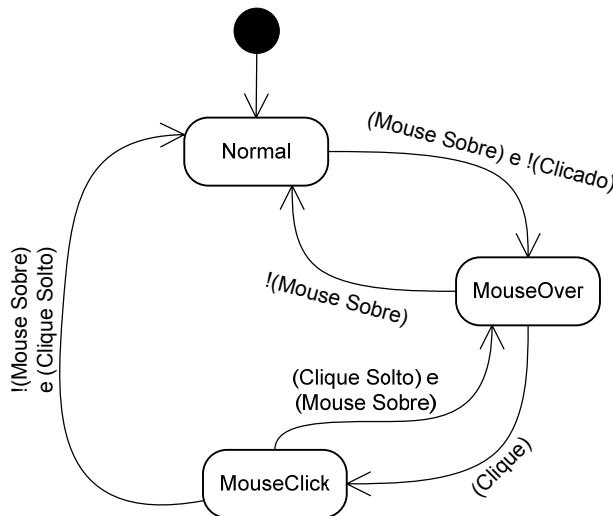


Figura 60: Máquina de Estados do Botão

Uma análise similar foi feita para os demais widgets, contudo, um widget que merece atenção especial é o *EditBox*, a caixa de texto. Além de sua relevância, o mesmo funciona de forma diferente dos demais, pois, possui o teclado como fornecedor de interações, e necessita de um tratamento especial para que seja desenhado corretamente. Na Figura 61 temos a máquina de estados do mesmo.

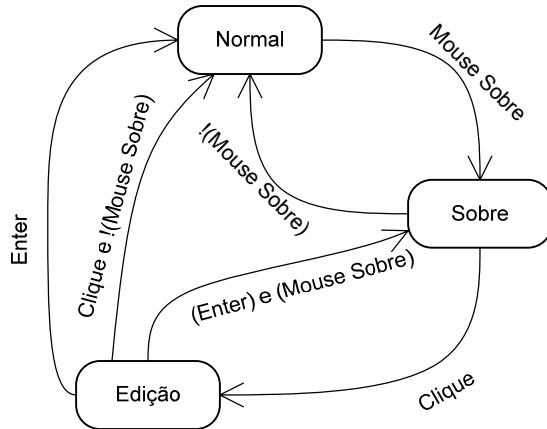


Figura 61: Máquina de Estados da Caixa de Texto

Quando o usuário coloca o EditBox em estado de edição, esse necessita que a cada tecla digitada, essa seja desenhada sobre o EditBox, mas sem ultrapassar os limites do mesmo. Para evitar que o desenho de uma tecla ultrapasse os limites, o EditBox possui um vetor dinâmico de caracteres que mantém o que é digitado nele e, a cada tecla digitada é recalculado o novo tamanho do desenho da string, para isso usa-se as funções da GLUT, e é desenhado até o caractere que não ultrapasse a dimensão do EditBox.

Outro efeito relevante é o de *shifting* que deve ser realizado quando o cursor, outro elemento que deve ser desenhado, encontra-se no limite da caixa de texto e o usuário pressiona a tecla → ou ←. As teclas BackSpace e Delete devem ser tratadas também, pois elas exercem a função de remoção de caracteres. O estado de edição pode ser representado por um grafo direcionado completo, ou melhor, um digrafo completo com pelo menos cinco estados. Essa abordagem gera um número muito grande de transições ($n = n(n - 1) = 5 * 4 = 20$ transições), logo, é preferível criar mais um estado de espera para centralizar o processamento, gerando o grafo abaixo com 10 transições, simplificando consideravelmente a programação conforme Figura 62.

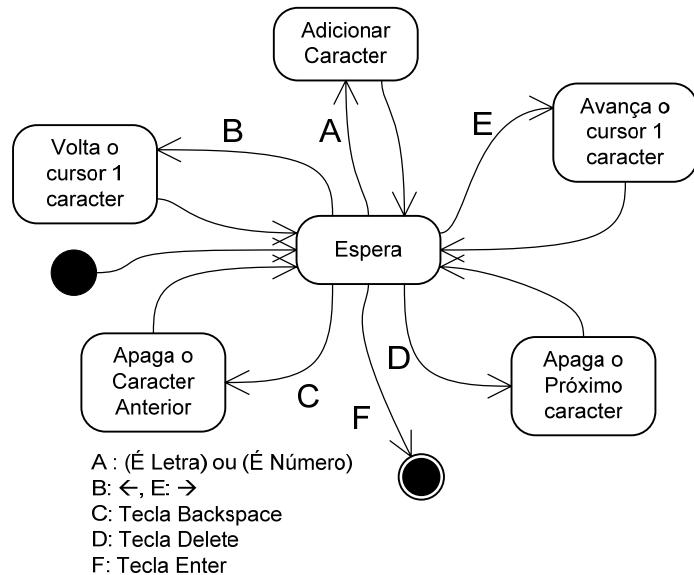


Figura 62: Máquina de Estados do Modo Edição do EditBox

Para o reconhecimento de seleção com o mouse de objetos 3D, foi utilizada a função de seleção 3D disponível na GLUT.

Terminado a explanação sobre o desenvolvimento do motor, passa-se a apresentar o desenvolvimento de um jogo utilizando o Grifo como base.

CAPÍTULO V

DESENVOLVIMENTO DO JOGO ROBOTICS SOCCER

Nesse capítulo será discutido o processo de criação de um jogo utilizando o Grifo. Esse processo (RIBEIRO, 2005) inicia-se com a elaboração do projeto do jogo, que contêm o objetivo e a história (storyboard), as regras (SALEM, et al., 2004), as personagens, as telas e os mecanismo de funcionamento e controle, os quais impactam na jogabilidade¹⁵.

Concluído o projeto, é necessária a produção técnico-artística dos elementos definidos para o jogo. Nesse processo são projetados e desenvolvidos os modelos 3D das personagens e suas respectivas animações, os modelos 3D dos objetos e os cenários 3D. Além disso, são produzidas as texturas dos elementos de jogo, as telas de interface, que devem ser cuidadosamente planejadas, seguindo os procedimentos de interface com o usuário (PRESSMAN, 2005).

O processo completo de projeto de jogos é um assunto extremamente amplo e multidisciplinar. Nesse texto não serão abordadas todas as etapas e nem todos os aspectos do desenvolvimento de jogos. É importante salientar que os capítulos anteriores descrevem o processo de criação de um software, um motor de jogos, ficando assim, mais inseridos com a área de computação. Já esse capítulo descreve-se o projeto de um jogo e, portanto, várias áreas se relacionam ao processo.

5.1 Contexto do jogo

O *Robotics Soccer*, ou *RoS*, é um jogo de futebol 3D, em que as personagens são robôs. Eles são bípedes, possuem braços e tentam se parecer com humanos usando máscaras. Mas a habilidade futebolística deles é muito duvidosa, de forma que, devido às restrições de movimento, eles não conseguem exibir o “futebol arte”.

¹⁵ Jogabilidade é a característica que um jogo possui para ser fácil e intuitivo de se jogar.

O jogador irá controlar um dos times de uma partida entre grandes clubes de robô, o Flabot e o BotFogo. Na arquibancada estarão milhares de robôs que acompanharão a partida, o desafio é conseguir vencer o time que é comandado por um supercomputador servidor, e assegurar a hegemonia dos humanos. Contudo, é permitido jogar em dupla, com cada jogador controlando um time.

O estilo do jogo é do tipo de simulação esportiva, no caso o futebol, sendo o público alvo a grande massa de jogadores de futebol de computador, que vão desde pessoas de 14 até 45 anos de idade (ou mais).

5.1.1 StoryBoard

Storyboards são organizações gráficas, como por exemplo, uma série de ilustrações, ou uma seqüência de imagens. O propósito deles é fazer uma pré-visualização da seqüência de acontecimentos, seja de histórias em quadrinhos, filmes ou outras mídias (no caso jogos). O livro *The Story of Walt Disney* (Miller, 2005), mostra a origem dos storyboards, os quais foram criados em 1933 no curta metragem da Disney, *Three Little Pigs* (Os três porquinhos). A Figura 63 mostra um storyboard criado para o jogo Lohan, desenvolvido pelo autor desse texto.

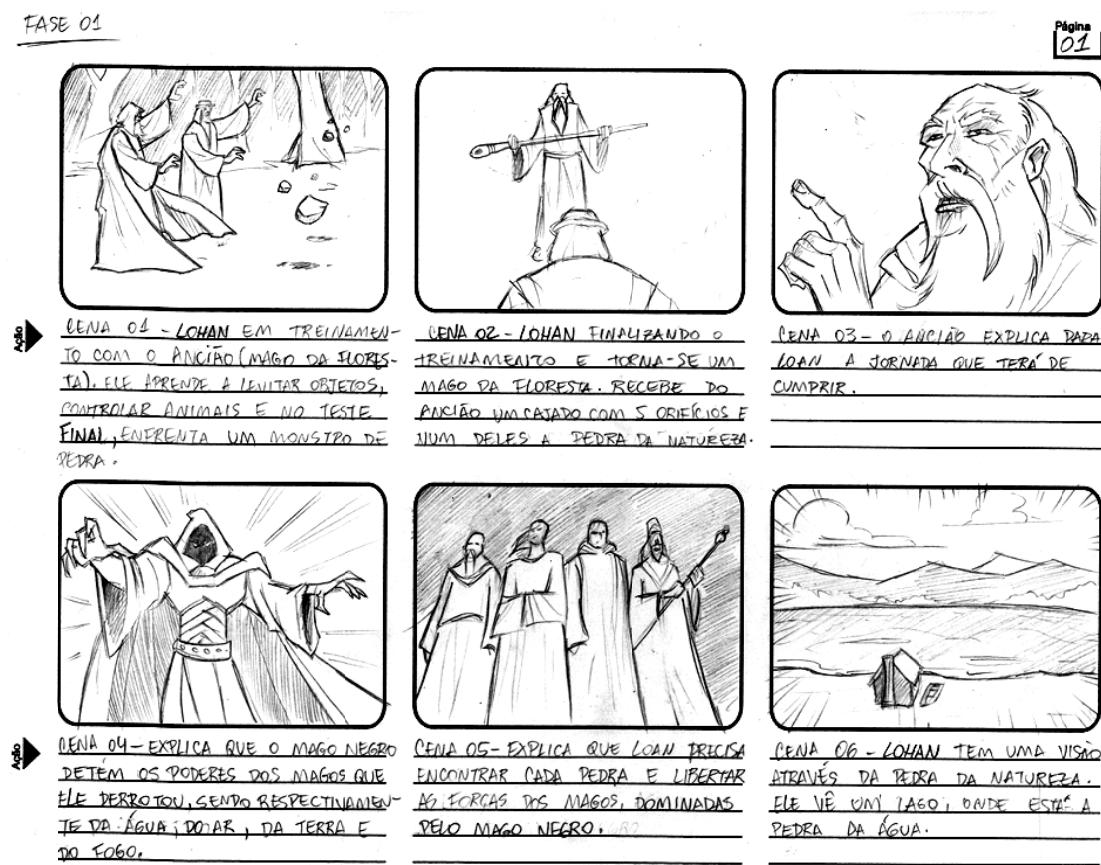


Figura 63: StoryBoard do Jogo Lohan

Para o RoS foi criado um storyboard, o qual apenas descreve a seqüência de telas do jogo. Na Figura 64 temos uma das páginas do storyboard do jogo, apesar da simplicidade, ele auxilia imensamente na organização da forma de ação do jogo.

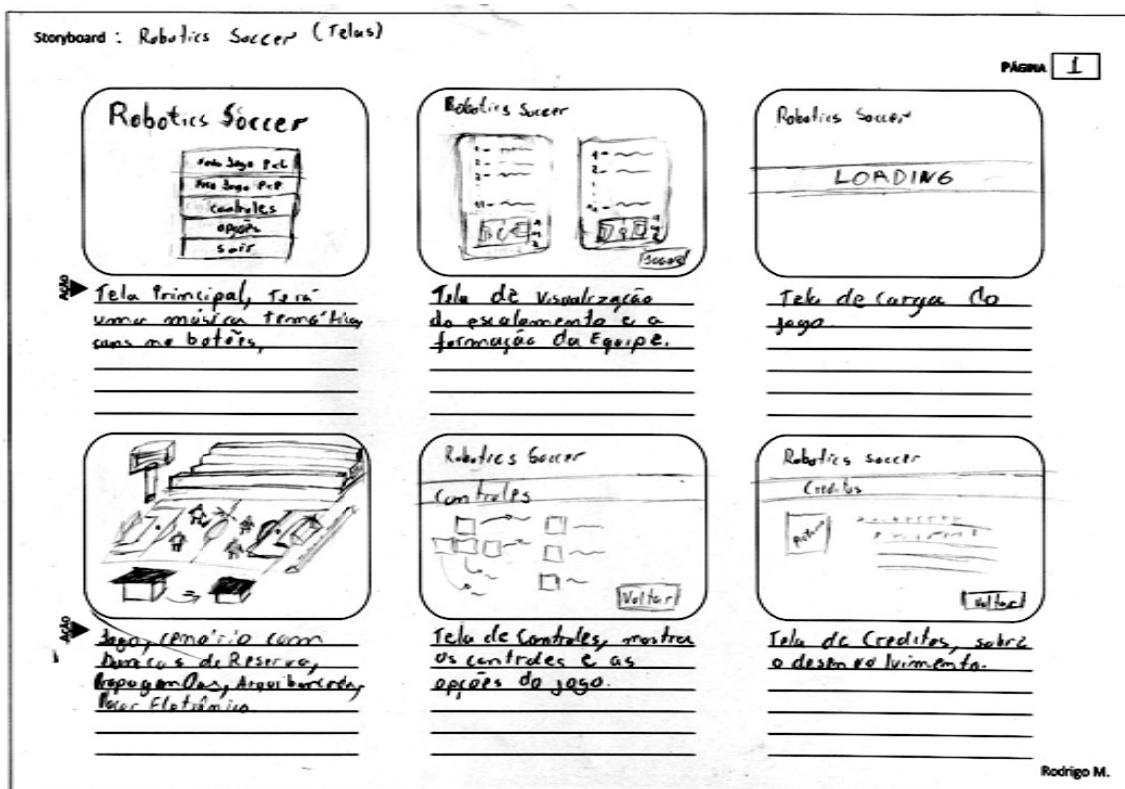


Figura 64: StoryBoard do RoS

5.2 Regras do jogo

Todo jogo possui regras (RIBEIRO, 2005) que criam o desafio e o estímulo para se entreter com o jogo. No caso do RoS, muitas regras como, as regras e regulamentos de futebol (Fédération Internationale de Football Association - FIFA, 2007) e as leis da Física, já estavam implicitamente definidas. Contudo, algumas regras devem ser adicionadas para melhorar a jogabilidade, ou aumentar/diminuir a dificuldade.

No RoS, foi necessária a elaboração de algumas regras de funcionamento e considerações de simulação. É sabido que em jogos de futebol, quando o jogador encontra-se mais próximo da linha de meta adversária do que a bola e o penúltimo adversário, é dito que o mesmo se encontra na situação de impedimento, sendo assim, o ataque (se for o caso) é impedido e o árbitro concede um tiro livre indireto à equipe adversária, no lugar onde ocorreu a infração (impedimento). Essa situação não é tratada pelo RoS.

Outra consideração importante é feita quando se analisa as infrações de falta, pois nesse caso o livro de regras do futebol é subjetivo, o que complica a aplicação dessa regra no jogo. Para comportar tal regra, foi realizada uma modificação para que a mesma fosse tratada como uma regra de decisão. A regra criada diz que, quando ocorrer uma colisão entre jogadores de equipes opostas, calcula-se o módulo da diferença de velocidade entre os envolvidos, calculada pela Equação 25, e identifica-se utilizando o vetor de frente a forma de colisão sendo, em seguida, aplicado o critério da Tabela 13. A função `random(X)` gera um número aleatório de 0 a X.

$$\underline{d}(v_1, v_2) = ||v_1 - v_2|| \cdot \left(0,8 + 0,2 \frac{random(100)}{100} \right)$$

Equação 25: Cálculo do Fator de Colisão

Tabela 13: Tabela de decisão do tipo de falta

	Colisão Frontal	Colisão Traseira	Colisão Lateral
Sem Falta	$\underline{d} < 1.2 \text{ m/s}$	$\underline{d} < 0.5 \text{ m/s}$	$\underline{d} < 1.2 \text{ m/s}$
Falta	$1.2 \leq \underline{d} < 6 \text{ m/s}$	$0.5 \leq \underline{d} < 3 \text{ m/s}$	$1.2 \leq \underline{d} < 6 \text{ m/s}$
Cartão Amarelo	$6 < \underline{d} < 11 \text{ m/s}$	$3 < \underline{d} < 8 \text{ m/s}$	$6 < \underline{d} < 12 \text{ m/s}$
Cartão Vermelho	$\underline{d} > 11 \text{ m/s}$	$\underline{d} > 8 \text{ m/s}$	$\underline{d} > 12 \text{ m/s}$

Sabendo que o valor máximo do módulo da velocidade dos jogadores robôs é de 8 m/s , ou 28,8 km/h, o módulo da diferença é de, no máximo, 16 m/s. Dessa forma, montou-se o critério da tabela 13.

Além disso, a cobrança de lateral é feita com os pés, assim como no futebol de salão. O tempo de jogo foi marcado para que se tenha uma equivalência de 10 minutos por partida, 5 minutos por período de jogo, não há tempo extra e vence quem realizar o maior número de gols durante o tempo regulamentar. Por fim, outras regras foram adaptadas para o melhor e mais simples funcionamento do jogo.

5.3 Mecanismos do jogo

Com a história e as regras definidas, é necessário projetar e desenvolver os mecanismos para o funcionamento do jogo, a forma de interação com o usuário e os recursos disponíveis para o mesmo.

O ponto fundamental dessa análise é a definição dos dispositivos de entrada do jogo (controles) e os comandos de jogo. Para tal, o RoS permite a utilização do Teclado e do Joystick durante o jogo e Mouse durante os menus sendo, também possível mudar a posição de uma

câmera com o mouse. No caso do uso de joystick, deve-se instalá-lo no sistema e calibrá-lo corretamente.

Escolhidos os controles, deve-se definir a tabela de teclas. Nela são mostrados os comandos do jogo e as teclas associadas aos mesmos. O RoS possui uma tabela padrão, e foi, foi criado um arquivo XML (Anexo 1), no qual tais teclas podem ser mudadas. Além disso, um programa de gráfico de configuração foi criado e nele é permitido a alteração das teclas (Anexo 2). A Figura 65 mostra a tabela de teclas.



Figura 65: Lista de Comandos e Teclas do Ros

Para os jogadores desavisados, no menu principal do jogo há uma cena que apresenta as teclas padrão.

O jogo possui seis câmeras, uma da visão superior do campo, uma da visão lateral do jogo, uma da visão colocada dentro da bola, uma da visão do robô (1^a Pessoa), uma da visão de 3^a Pessoa e uma câmera livre, que o usuário pode movê-la e colocá-la em qualquer lugar, sendo essa muito utilizada na análise do cenário.

Além disso, outras opções são disponíveis para o usuário, como utilizar luzes, efeitos como motion blur, shader, fog e a exibição de um mapa do campo.

5.4 Projeto Artístico

Prosseguindo o desenvolvimento, é preciso modelar as personagens. Para isso foi feito um croqui de cada personagem e nessa fase os detalhes ergonômicos e físicos, tais como, tamanho da personagem, são avaliados e após a conclusão do esboço, é utilizado um software de Modelagem 3D, no caso foi utilizado foi o 3D Studio Max 8, sendo que a primeira versão das personagens foram feitas diretamente no código, para modelar a personagem em 3D. Com esse modelo desenvolvido, o mesmo é exportado para o formato obj e carregado no motor.

A Figura 66a mostra o esboço da personagem jogador, que serviu como modelo para outras, e na Figura 66b é mostrado o mesmo modelo, contudo esse já modelado no 3D Studio Max 8.

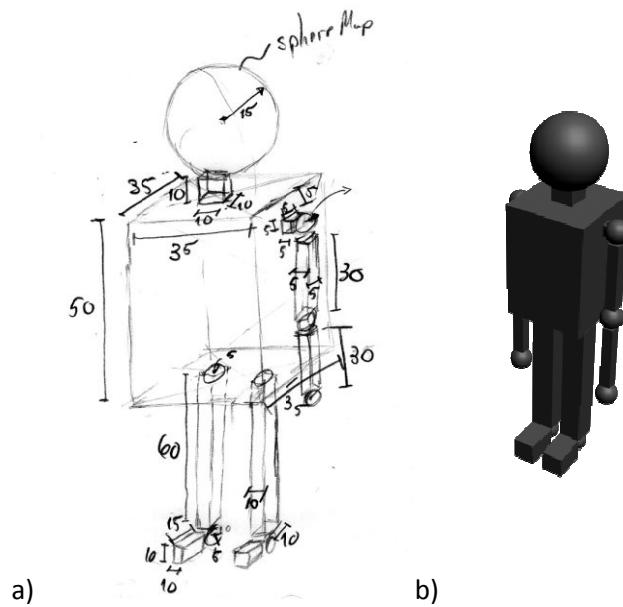


Figura 66: a) Esboço da personagem jogador b) Modelo do Jogador

A Figura 67 mostra a interface e o modelo da personagem em processo de construção e é possível perceber que a modelagem utiliza quatro visões do objeto, a visão frontal, a visão lateral esquerda, a visão superior e a visão perspectiva.

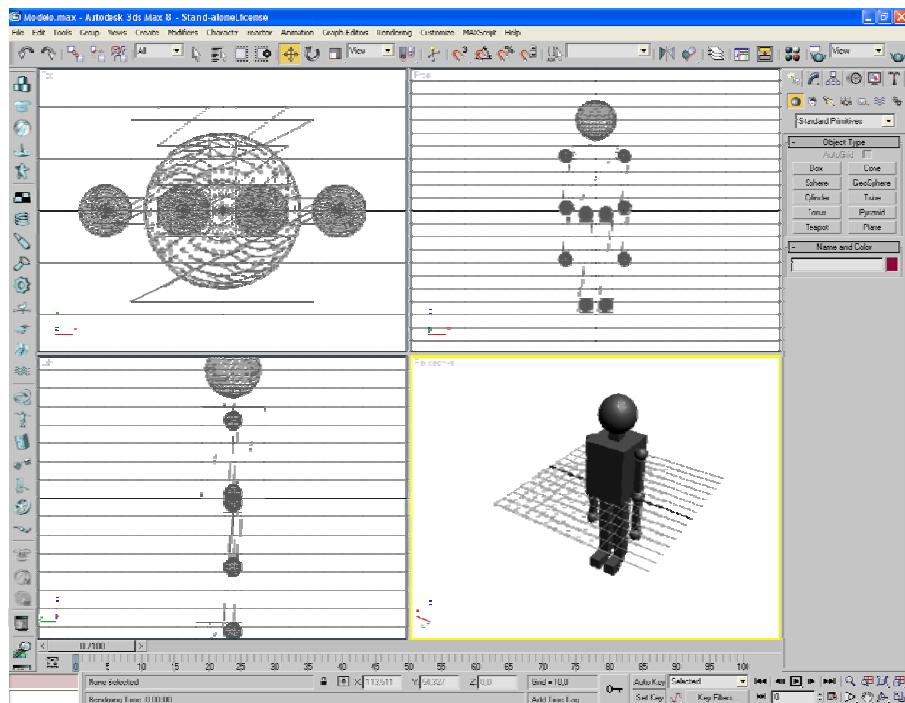


Figura 67: Modelagem da Personagem

Tendo a modelagem da personagem pronta, deve-se pensar na forma de animação. Como o Grifo não utiliza animação, pensou-se em dividir o modelo em três partes, cabeça e tronco, braços, antebraços e duas pernas, esse tipo de modelagem é conhecida com modelagem hierárquica. Dessa forma, definiu-se que o braço poderia rotacionar apenas em torno do eixo x. Para girar, dever-se-ia modificar o ângulo de rotação do braço. A Figura 68 mostra o grafo de hierarquia do modelo e a Figura 69 apresenta os ângulos e a forma de rotacionar o braço. As pernas possuem esquema similar.

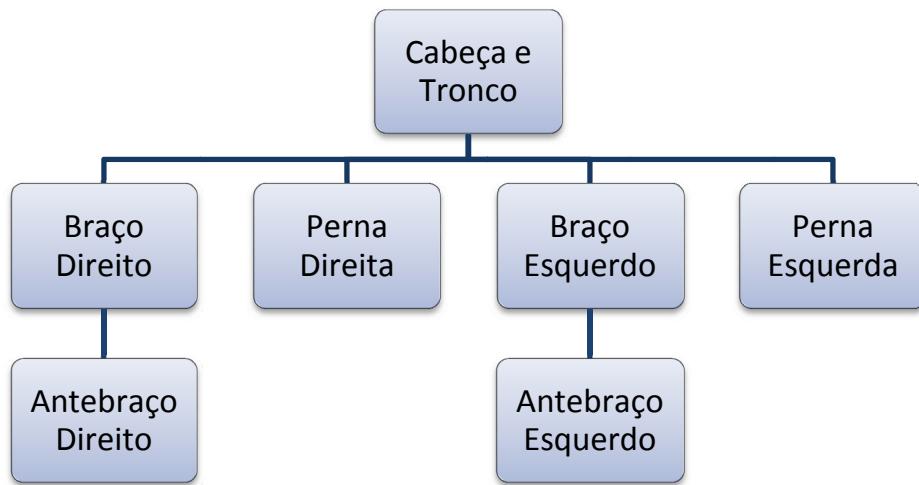


Figura 68: Grafo do modelo hierárquico da personagem

O grafo da Figura 68 mostra que para o desenho da personagem existem partes intermediárias. Dessa forma, uma parte pode ser submetida a uma rotação antes de ser desenhada. Tendo isso, para a animação utilizou-se a função — para criar o efeito oscilatório do braço balançando. É claro que a função só é ativada quando a personagem se move e quando ela começa a parar, é especificado em código que o ao passar pela posição de equilíbrio (braço abaixado), o movimento seja cessado (animação). Um detalhe importante é que a função varia a velocidade da oscilação dependendo da velocidade da personagem, sendo o parâmetro k é apenas uma constante de controle.

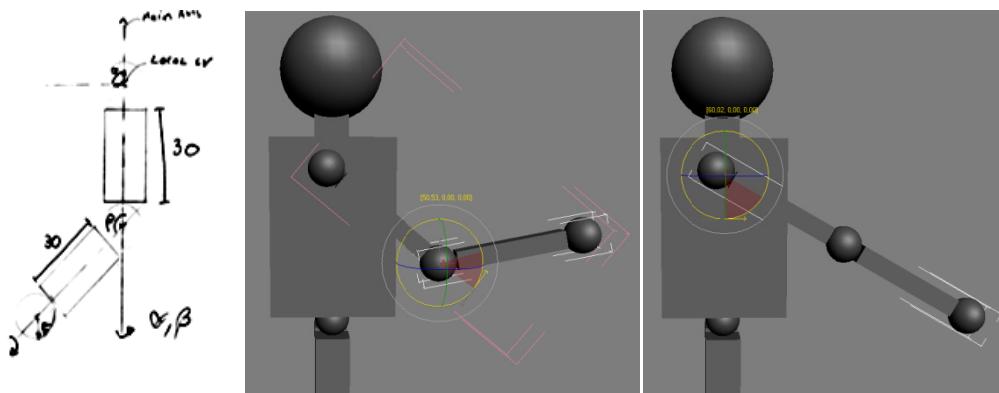


Figura 69: a) Esboço dos ângulos de rotação α, β b) Ângulo α c) Ângulo β

Os modelos dos bandeirinhas e árbitro são os mesmos dos jogadores, mas com a textura diferente.

Após a modelagem da personagem, iniciou-se a etapa de modelagem do cenário, nela foram consideradas as dimensões e linhas de campo especificadas pela FIFA. As Figuras 70 e 71 mostram as linhas necessárias e as dimensões, respectivamente.

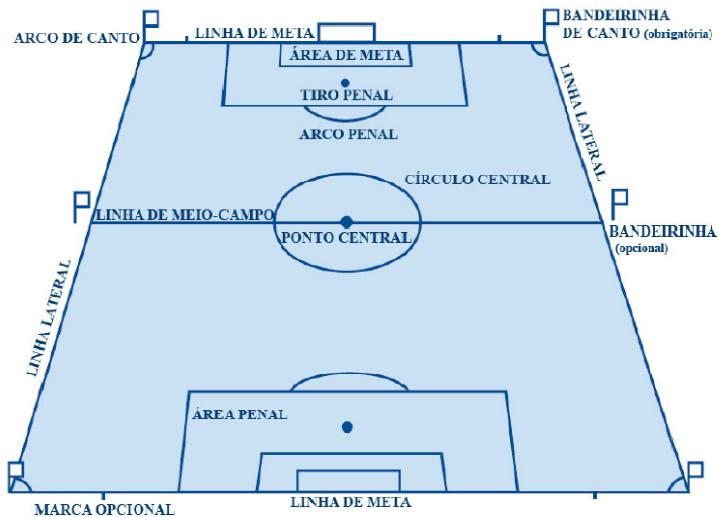


Figura 70: Linhas do Campo

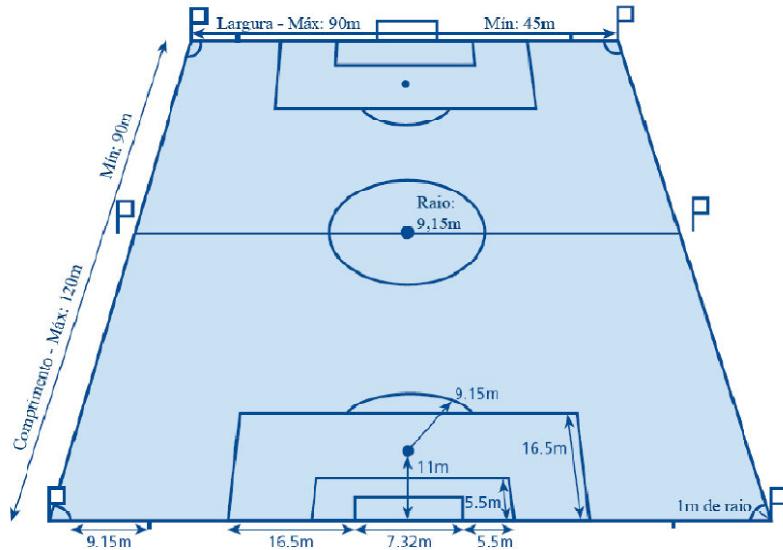


Figura 71: Dimensões Oficiais do Campo

Além do campo, foi indispensável a criação de outros elementos da cena, como placar, arquibancada, área técnica, propaganda e outros, para contextualizar o jogo. Isso necessitou de outro esboço e mais modelagem. Foram também posicionadas as luzes do jogo (refletores). A Figura 72a mostra o esboço da cena, a Figura 73a mostra a criação do modelo, a 73b mostra o posicionamento das luzes e a 72b mostra o cenário.

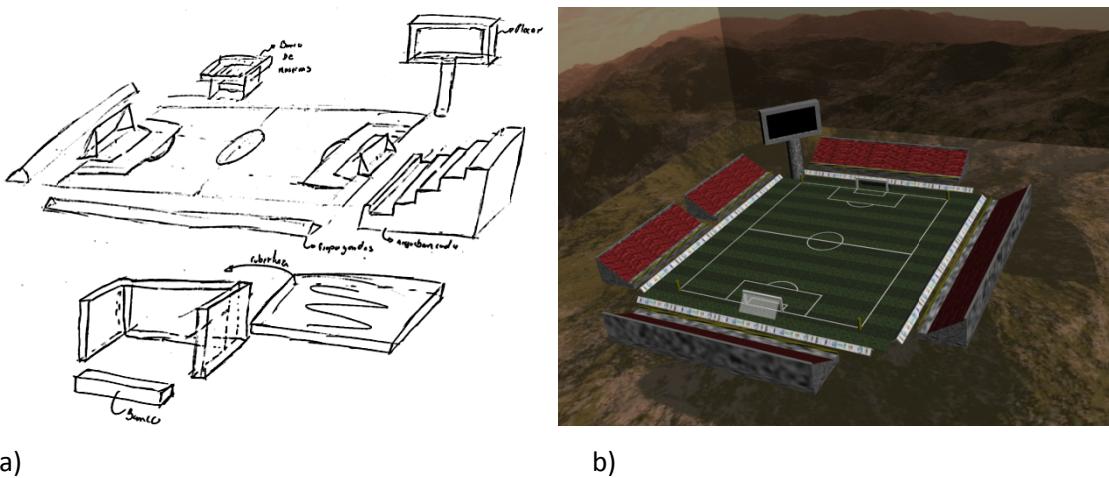
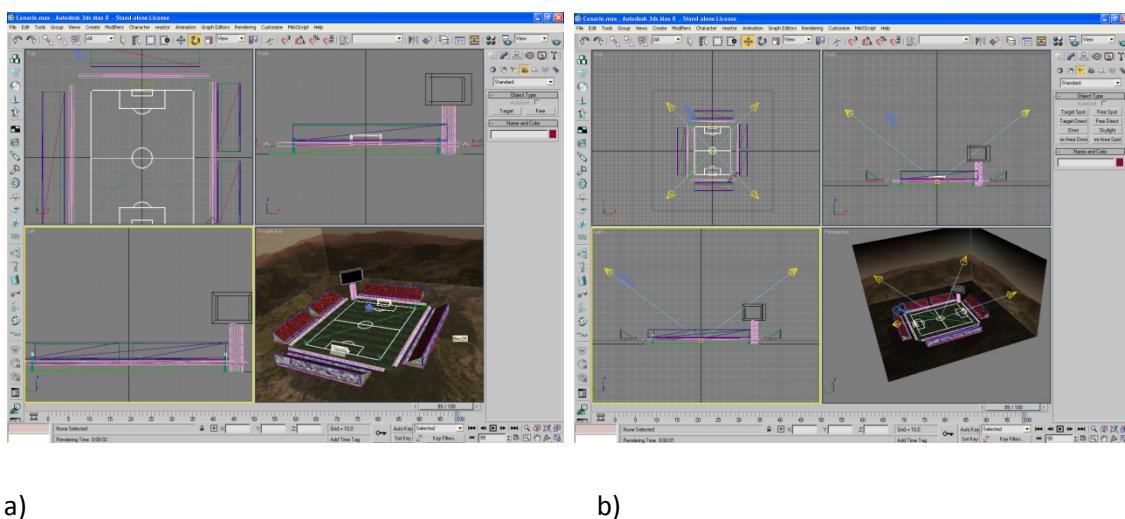


Figura 72: a) Esboço do cenário b) Modelo do cenário

A produção de texturas é outro ponto importante. Para o RoS várias texturas foram produzidas. O processo de mapeamento foi, de inicio, feito manualmente, conforme Figura 16a, depois tal processo foi refeito no Ambiente de Modelagem através das funções de UVW Map do 3D Studio Max.

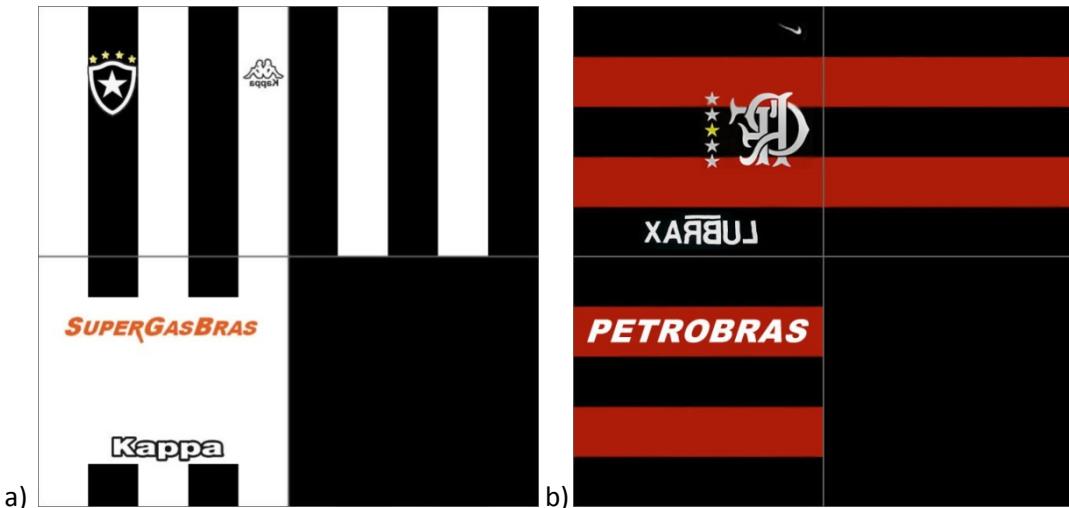


a)

b)

Figura 73: a) Modelo em construção do cenário b) Modelo em construção do cenário com luzes

As Figuras 74a e 74b são as texturas das roupas das personagens do time. Observa-se que estas figuras 2D conseguem mapear um objeto 3D, no caso, a parte frontal da roupa está na parte superior esquerda e é invertida pelo tipo de mapeamento que realizado, as partes laterais são repetidas e estão na parte superior direita, as parte superiores e inferiores da personagem são texturizadas pela parte inferior direita e, por fim, a parte traseira da roupa é mapeada na região inferior esquerda da imagem.

**Figura 74:** Texturas da camisa do BotFogo e do FlaBot

As Figuras 75a e 75b mostram a textura de um dos rostos possíveis para as personagens e a textura da bola. Observe que ambas foram produzidas como se houvesse um escalamamento da cabeça de um indivíduo, isso ocorre, pois, em ambos os casos, o mapeamento é sobre uma esfera.



Figura 75: Textura do resto e da bola

Para finalizar a parte de modelagem, é necessário especificar o mapa da cena, no caso, sendo apenas o posicionamento dos jogadores. No futebol existem várias opções, no RoS, por padrão, o posicionamento é o 4-4-2, quatro jogadores na defesa, quatro no meio de campo e dois no ataque. A Figura 76 mostra um exemplo de posicionamento, mapa da cena.

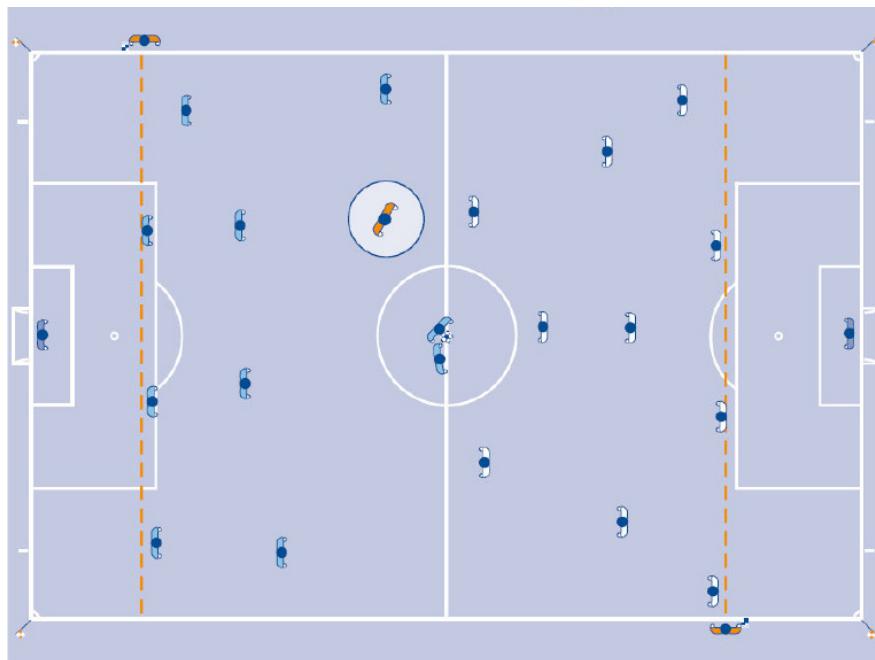


Figura 76: Posicionamento dos jogadores

Tendo o cenário concluído, define-se o layout das cenas (todos esses processos podem ser feitos em paralelo). Nessa parte analisa-se o que cada cena precisa de passar de informação, e como elas se comunicam, ou seja, as trocas de cena. Daí, cria-se a máquina de estados das cenas, apresentada na Figura 77, o layout e as imagens dos cenários. A Tabela 14 mostra todas as cenas do RoS, excluindo a cena do jogo, nota-se que as cenas são as mesmas apresentadas no storyboard.

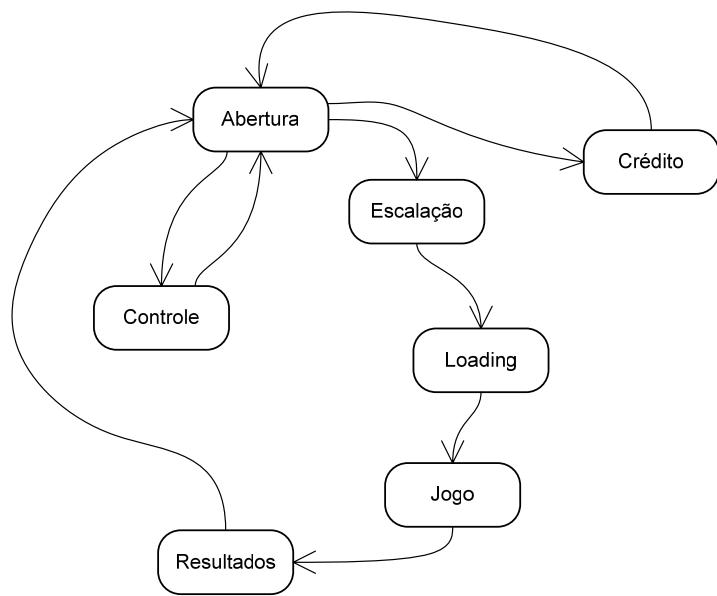
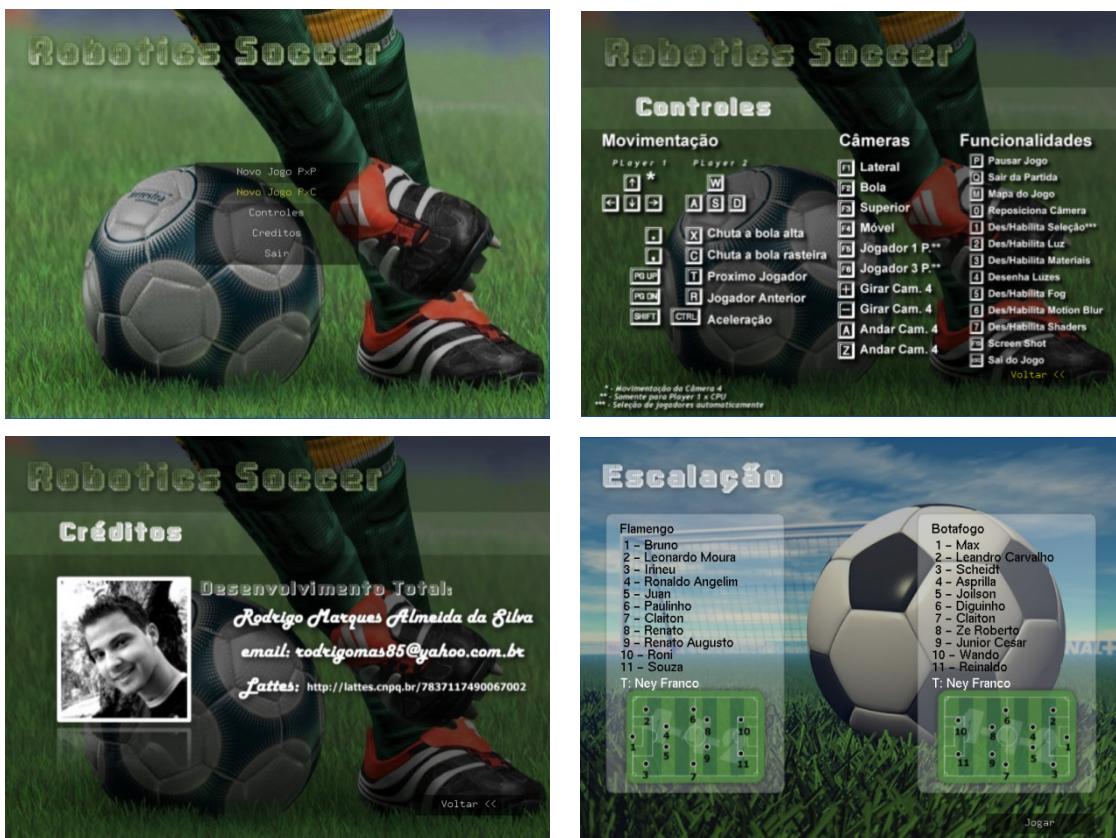


Figura 77: Máquina de Estados das Cenas

Tabela 14: Cenas do jogo





Outra necessidade na parte artística é escolher, e muitas vezes produzir, efeitos sonoros, como barulho de colisão, gritos e outros, além de criar, ou selecionar, as músicas, ou também trilhas, do jogo.

5.5 Projeto Técnico

Para o funcionamento do jogo, foram criadas classes específicas para cada elemento do mesmo. Essas classes foram herdadas do Grifo para poder utilizar seus recursos. O diagrama da Figura 78 mostra algumas dessas heranças.

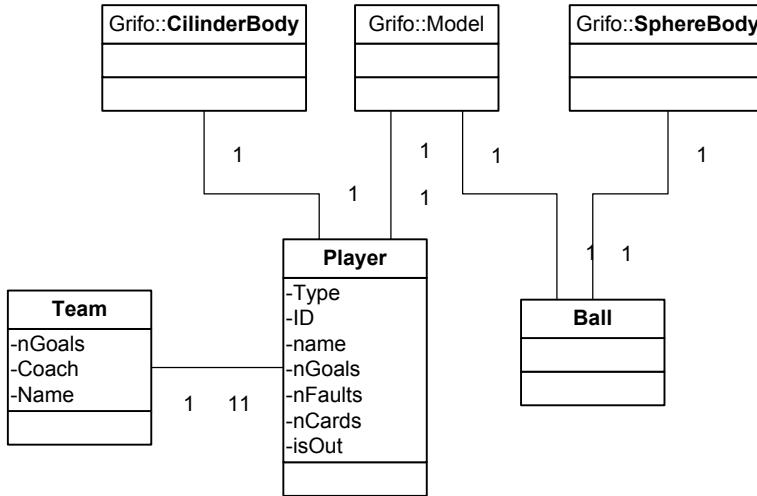


Figura 78: Diagrama de Classes do RoS

Os agrupamentos de classes são feitos para que cada uma delas, e.g. Player, tenha todos os dados integrados, ou seja, Modelo 3D e Física. É importante que se observe que cada classe já seleciona o tipo de modelo físico empregado na simulação.

Na instanciação da classe, a parte que pertence ao Modelo 3D é inserida na árvore do sistema de visualização, a parte de física é inserida no sistema de física. As características comuns são agrupadas, que no caso do C++ isso pode ser feito com o uso da herança múltipla.

Toda a parte de teste de regras do futebol foi desenvolvida especificamente para esse jogo. Esse sistema usa os dados do Grifo, como a posição, velocidade, teste de colisão, para gerar e testar a efetivação de uma regra do jogo.

Para o funcionamento da inteligência dos robôs, foram projetadas máquinas de estados para cada tipo de jogador. Nesse projeto, foram consideradas quatro classes de jogadores: zagueiro, goleiro, atacante e meio-campo. Cada classe possui uma máquina de estados específica que guia a inteligência do jogador. A implementação da mesma foi realizada com o auxílio do Script System do Resource Manager, sendo descrita em linguagem LUA.

A Tabela 15 descreve as transições de estados dos zagueiros, a variável distância é a distância entre o jogador e a bola. Novamente, a função `random(X)` gera um número aleatório de 0 a X.

Tabela 15: Transição da FSM dos Zagueiros

Estado Inicial	Função de Transição	Estado Final
RONDA	(distância < 60) e (random(100) < 90)	DEFESA
DEFESA	(distância < 1)	TOCANDO
TOCANDO	(distância > 3)	DEFESA
DEFESA	(distância > 65) e (random(100) < 80)	RONDA

A Tabela 16 mostra o que deve ser feito em cada estado. As descrições são feitas em linguagem comum e devem ser traduzidas para LUA.

Tabela 16: Ações dos estados dos zagueiros

ESTADO	Ação
RONDA	Seguir caminho programado
DEFESA	Ir a direção da bola
TOCANDO	Escolher colega com menor número de adversários próximos e tocar.

Apesar desse esquema simples de máquinas de estado, sendo o sistema de IA feito em script, apenas mudando o script, pode-se alterar completamente o funcionamento dos robôs. Com isso, a elaboração de avançadas táticas de jogo é possível. Para que isso funcionasse, criou-se um padrão no script LUA.

O script da Tabela 17 é o responsável pela inteligência da equipe do FlaBot. As variáveis `Enemies`, `Player`, `Ball` e `golz` são utilizadas para armazenar informações como posição e

velocidade dos jogadores do time adversário, jogadores do time FlaBot, da bola e do gol adversário, respectivamente.

Tabela 17: Funcionamento do script de IA

```
-- flamengo.lua
Enemies = []
Players = []
Ball = []
golz = 0
function think(id,team)
    --io.write("[LUA] Flamengo:" .. id .. "\n")
    -- Tática de jogo
    -- Atualiza posição

    return Func, Players[id]["x"], Players[id]["y"],Extra
end

function update_pos( id, x, z )
    Players[id]["x"] = x
    Players[id]["z"] = z
    --io.write("[LUA] Flamengo:" .. id .. " " .. x .. " " .. z .. "\n")
end

function update_vel( id, x, z )
    Players[id]["vx"] = x
    Players[id]["vz"] = z
    --io.write("[LUA] Flamengo:" .. id .. " " .. x .. " " .. z .. "\n")
end

function update_ball_vel( x, y, z )
    Ball["vx"] = x
    Ball["vy"] = y
    Ball["vz"] = z
    --io.write(" [LUA] Flamengo:" .. x .. " " .. y .. " " .. z .. "\n")
end

function update_ball_pos( x, y, z )
    Ball["x"] = x
    Ball["y"] = y
    Ball["z"] = z
    --io.write(" [LUA] Flamengo:" .. x .. " " .. y .. " " .. z .. "\n")
end

function update_goal_pos( z )
    golz = z
    --io.write(" [LUA] Flamengo:" .. z .. "\n")
end

function update_enemy_pos( id, x, z )
    Enemies [id]["vx"] = x
    Enemies [id]["vz"] = z
    --io.write(" [LUA] Flamengo:" .. id .. " " .. x .. " " .. z .. "\n")
end
```

As funções `update_pos`, `update_vel`, `update_ball_pos` e `update_ball_vel` são responsáveis por atualizar, respectivamente, os dados da posição e velocidade dos jogadores do time e da

bola. A função `update_enemy_pos` atualiza os dados da posição dos jogadores do time oposto, os dados de velocidade não são revelados, isso é feito para dificultar a tática, visto que, em situações mais realísticas essa informação não é de fácil obtenção. Essas funções são chamadas após a resolução de colisões, de forma a passarem a última informação de simulação de física.

A função `think` é de suma importância para a Inteligência do Jogo, ela que indica o que deve ser feito com os jogadores, qual atitude tomar. Conforme apresentado nos capítulos anteriores, devido à problemas no funcionamento da biblioteca luabind, não foi possível chamar funções em objetos C++ dentro de scripts LUA. Dessa forma, esse processo foi solucionado usando um campo no vetor de dados de retorno da função. Essa função deve retornar dados no formato: [código da função a chamar, x, z, Extra], que são interpretados em C++ pela Tabela 18.

Tabela 18: Tabela de funções de IA do RoS

Código da Func.	Função Chamada	X	Z	Extra
0	Nenhuma	-	-	-
1	gotoPos	Posição X	Posição Z	Velocidade
2	kickForce	Força X	Força Y	Força Z
3	kickToVel	Posição X	Posição Z	Velocidade
4	passTo	Posição X	Posição Z	-

A função `gotoPos` faz com que o jogador vá até a posição especificada (X,0,Z) com a velocidade (Extra), limitada a 8 m/s. A função `kickForce` aplica uma força (X, Z, Extra) na bola, ou seja, é um chute usando a força. Outra possibilidade é dar um chute usando a função `kickToVel`. Essa função implementa um lançamento, ou seja, a função chuta a bola, lançando, até a posição (X,0,Z) , chegando nesse ponto com a velocidade em y de módulo (Extra), veja a Figura 79. Por fim, a função `passTo` implementa um chute rasteiro, partindo da posição do jogador que a chamou até a posição (X,0,Z). Essas funções criadas fazem as devidas considerações de atrito e arrasto aerodinâmico.

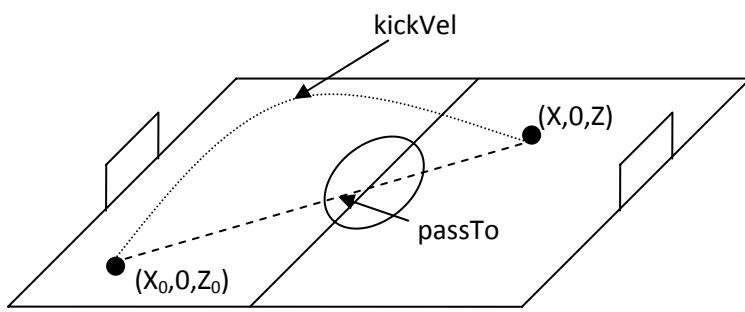


Figura 79: Tipos de chutes

Para testar o sistema de shaders, foram criados dois programas de shader, um vertex shader e outro pixel shader. O objetivo desses shaders é criar o efeito de contorno de uma luz sobre uma superfície. O código é simples, contudo, não foi considerada a matriz de textura, logo, ao aplicar o shader, não é possível visualizar as texturas. O efeito obtido é ilustrado na Figura 80.

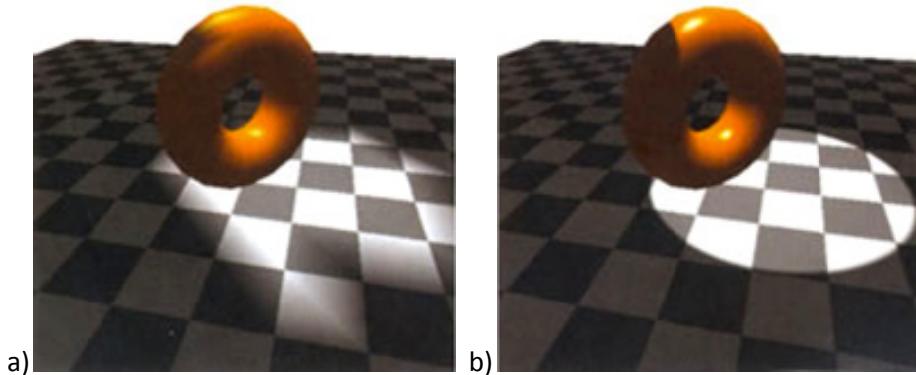


Figura 80: a) Efeito com o Vertex Shader b) Efeito com o Pixel Shader

Observe que o shader de vértice não cria um efeito muito bom, contudo o shader de pixel consegue simular muito bem o efeito desejado, entretanto não é considerada a sombra do torus¹⁶. Além dos shaders, os efeitos de Motion Blur e Fog foram utilizados (Figura 81).

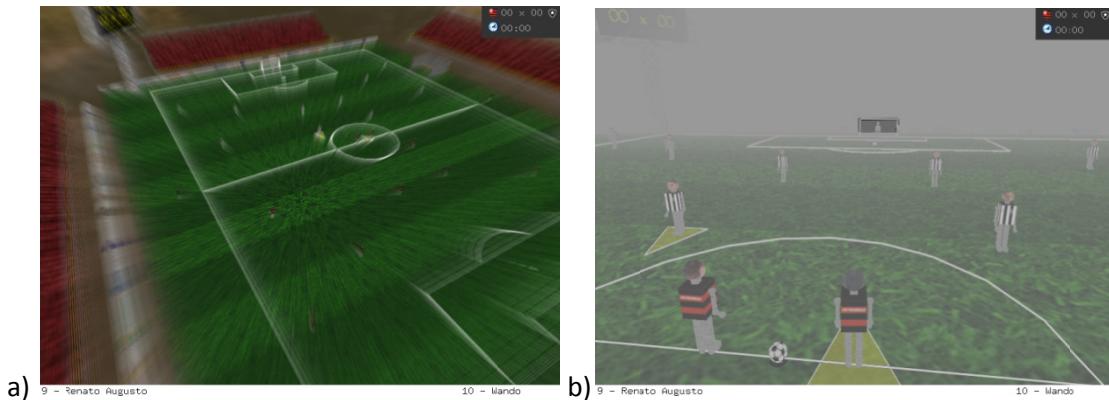


Figura 81: Efeitos de Motion Blur e Fog

Para melhorar a jogabilidade do RoS, foi implementado uma pequena simulação de sombra da bola. Para tal efeito, sem o uso de shaders, foram utilizados os conceitos de física ótica e geometria.

Outra funcionalidade incorporada foi a criação de um mapa do jogo, mostrado na Figura 82, ou seja, uma pequena representação 2D do jogo. Para isso, fez-se o mapeamento da cena 3D para

¹⁶ Torus (pl. tori) é um superfície de revolução gerada pela rotação de um círculo em um eixo coplanar à ele, mas que não o toca.

o espaço 2D do mapa. Esse processo é similar ao descrito no capítulo 4, consistindo basicamente de conversão de bases vetoriais.



Figura 82: Mapa do jogo

Por fim, muitas funcionalidades do Grifo estão presentes no RoS, a descrição detalhada de cada item delongaria desnecessariamente o capítulo.

5.6 Resultados

A utilização do motor Grifo auxiliou imensamente a produção do jogo RoS. Apesar da simplicidade do jogo, ele foi capaz de mostrar muitas das funcionalidades do motor.

O jogo foi testado por um grupo de 10 usuários. Apesar do procedimento informal dos testes, todos aqueles, avaliaram positivamente o jogo, no sentido que o mesmo atende aos requisitos mínimos de um jogo desse tipo. É claro que, em vista dos jogos comerciais de futebol, os resultados obtidos com o RoS são muito restritos. Esses são produzidos por equipes que chegam a mais de 200 profissionais, das mais diversas áreas. As alterações sugeridas pelos testadores foram melhorias gráficas e algumas melhorias de jogabilidade.

A seqüência de figuras a seguir mostra os modelos do jogo e o jogo em funcionamento, isso demonstra o trabalho funcionando na prática. Cabe ressaltar que o produtor do jogo não possui formação artística, logo, muito poderia ser melhorado apenas com um melhor trabalho artístico das cenas.

Tabela 19: Tabela de Imagens dos Modelos do RoS

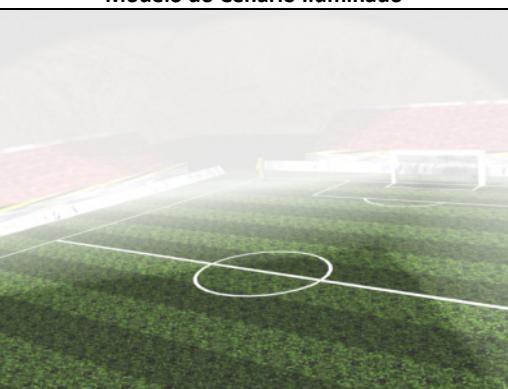
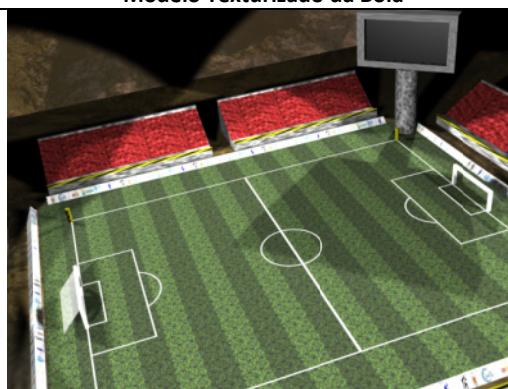
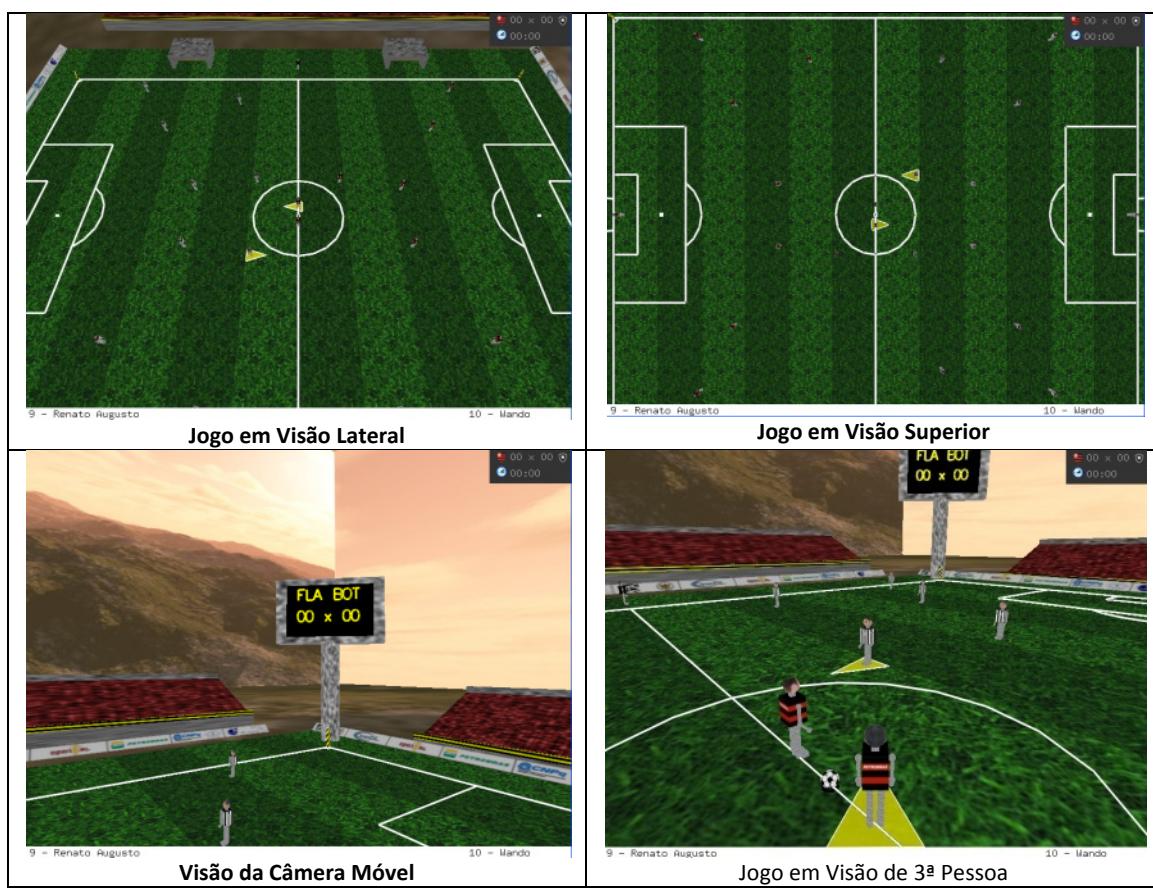
	
Modelo do Cenário Iluminado	Modelo Texturizado da Bola
	
Modelo do Cenário com Nêvoa (Fog)	Modelo do Cenário na Visão Lateral
	
Modelo do Cenário na Visão Superior	Modelo do Cenário com Nêvoa na Visão Superior
	
Modelo com Iluminação da Personagem	Modelo Texturizado da Personagem

Tabela 20: Imagens do Jogo em Funcionamento

CAPÍTULO VI

CONCLUSÕES E TRABALHOS FUTUROS

Esta monografia apresentou um pouco da história e do mercado de jogos o qual teve o objetivo de mostrar a importância desse segmento. Além disso, foi discutido o projeto do motor de jogos Grifo, detalhando a maioria das técnicas de computação gráfica, simulação de tempo real, processo de software e outras áreas. Por fim, foi descrito o desenvolvimento e implementação do projeto do jogo RoS.

Durante o desenvolvimento deste texto pôde-se perceber a grande diversidade de técnicas e áreas envolvidas diretamente e indiretamente no projeto de um jogo, daí a grande complexidade dos jogos. É fato que muita das áreas foram superficialmente discutidas, contudo, o objetivo teve o enfoque na apresentação dos conceitos, e não no uso de uma tecnologia específica.

Um ponto que necessita de uma melhor análise é o teste, pois, partindo do princípio de que em geral todo software possui problemas e a única forma de reduzi-los é com o uso de uma boa engenharia de software, apoiada com uma grande massa de testes que, assim, validam o funcionamento do sistema como um todo.

Com isso, conclui-se que o segmento de jogos eletrônicos é, sim, uma grande área de pesquisa e que necessita das mais variadas e complexas técnicas de computação, eletrônica e outras áreas das ciências. No tocante a tal fato, esse texto apresenta um resumo de todo o processo de desenvolvimento de tais aplicações, não se atendo a somente a produção do software (jogo) final, mas a todos os requisitos de sua base de desenvolvimento, desde as estruturas elementares de computação gráfica até aos complexos sistemas de simulação de física.

Os resultados obtidos neste trabalho permitem, ainda, a confecção de um motor de qualquer gênero de jogos, contribuindo assim, com o desenvolvimento da tecnologia de jogos brasileira.

Por fim, o objetivo do projeto foi alcançado, não só com a produção de um motor de jogos, mas também como uma reunião de vários conceitos adquiridos durante o curso de Engenharia de Computação. Esse objetivo é considerado pelo autor, ainda, maior que a construção desses sistemas, pois esse é um projeto de vários anos e que só foi possível com a colaboração de vários professores e de grande esforço individual.

Trabalhos Futuros

O tema abordado nesta monografia é bastante amplo e oferece várias áreas de pesquisa. Dentre os trabalhos futuros relacionados ao que foi aqui apresentado, destacam-se:

- Estudar métodos de animação de personagens, como, por exemplo, técnicas de quadro chave, *keyframing*, com interpolação, animação por script (WATT, 2000).
- Estudar técnicas de animação facial de personagens, uma área que ainda há muito que pesquisar.
- Incluir sistema de partículas e Billboards¹⁷, para explorar efeitos de explosões e outros.
- Estudar e projetar os sistemas de comunicação para o suporte de jogos em rede. Essa área é de vital importância, pois com a mesma seria possível a criação de um jogo de futebol 3D em rede, por exemplo. Existem muitas tecnologias e estratégias para o desenvolvimento desse sistema.
- Estudar métodos de simulação física multiprocessada, como por exemplo, o uso integrado da GPGPU e o CPU do computador, um trabalho similar pode ser encontrado em (Clua, et al., 2007). Isso melhoraria significativamente o desempenho da simulação física, além de melhor utilizar os recursos do computador.
- Finalização dos subsistemas não implementados, informados ao longo do texto.
- Estudo de técnicas para comunicação do sistema, através de redes sem fio, com sistemas remotos como robôs reais (e.g. Lego Mindstorm).

O desenvolvimento desse trabalho permitirá o estabelecimento da área de jogos como área de interesse específica e estratégica de qualquer curso de Engenharia de Computação em qualquer instituição de nível superior pública ou privada.

¹⁷ Billboards são imagens bidimensionais que são renderizadas como planos em cenas 3D, com o detalhe de sempre ficarem de frente à câmera virtual.

ANEXOS

Anexo 1 - Arquivos de Configuração

Para configurar os diversos sistemas do Grifo foi criado um arquivo XML com os parâmetros de configuração. Nesse arquivo é possível inserir luzes no jogo, mudar suas posições, orientações e habilitar efeitos especiais.

Além disso, é possível especificar as constantes físicas de simulação, podendo, dessa forma, variar a gravidade, viscosidade do meio, o atrito e outros.

O arquivo de configuração contém, ainda, todos os comandos de ação do jogo, desse modo, é possível mudar as teclas do mesmo, além de tocar o dispositivo de jogo. A Figura 83 apresenta o arquivo. As configurações de linguagem também podem ser alteradas.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <robotics version="0.2.1" languagefile="default.lng">
3      <teams file1="data/Flamengo.xml" file2="data/botafogo.xml" arbitro="Pedro" laterall1="Jose" laterall2="Joao"/>
4      <physics method="euler" pentol="0.2" wind="true" magnus="true" drag="true" maxspeed="30.0">
5          <gravity>9.7</gravity>
6          <sndspeed>343.3</sndspeed>
7          <dopplerfactor>1.0</dopplerfactor>
8          <friction floorY="0.65" floorXZ="0.97" player="0.3"/>
9          <restitution playerBall="0.99" ballFloor="0.5"/>
10         <wind dir="0.1,0.1,0.1" random="false"/>
11         <air viscosity="0.0000183" density="1.244"/>
12     </physics>
13     <graphics enablemotion="false" motionframes="5" motionfactor="0.9" fadesteps="20" fadetimer="50" useshader="false" vextexshader="",
14         <fog enable="false" color="0.6,0.6,0.6" density="0.1" start="-80.0" end="80.0" mode="0"/>
15         <lights enable="false" ambientcolor="0.9,0.9,0.9,0.9" draw=false nlights="5">
16             <light type="spot" ambient="0.9,0.9,0.9,0.9" diffuse="0.6,0.6,0.6" specular="0.2,0.2,0.2" position="0.0,20.0,0.0" direction="0.0,
17                 <light type="spot" ambient="0.9,0.9,0.9,0.9" diffuse="0.6,0.6,0.6" specular="0.2,0.2,0.2" position="-60.0,20.0,-70.0" direction="",
18                 <light type="spot" ambient="0.9,0.9,0.9,0.9" diffuse="0.6,0.6,0.6" specular="0.2,0.2,0.2" position="-60.0,20.0,70.0" direction="6,
19                 <light type="spot" ambient="0.9,0.9,0.9,0.9" diffuse="0.6,0.6,0.6" specular="0.2,0.2,0.2" position="60.0,20.0,-70.0" direction="",
20                 <light type="spot" ambient="0.9,0.9,0.9,0.9" diffuse="0.6,0.6,0.6" specular="0.2,0.2,0.2" position="60.0,20.0,70.0" direction="6
21             </lights>
22     </graphics>
23     <controls usejoystick="true" joypolling="100" joyx="100,-100" joyy="100,-100" joyz="100,-100">
24         <player3 up="AXIS-Y+" down="AXIS-Y-" left="AXIS-X+" right="AXIS-X+" kick1="BTN1" kick2="BTN2"/>
25         <player1 up="UP" down="DOWN" left="LEFT" right="RIGHT" kick1="x" kick2="c"/>
26         <player2 up="W" down="S" left="A" right="D" kick1="V" kick2="B"/>
27         <player4 up="AXIS-Y+" down="AXIS-Y-" left="AXIS-X+" right="AXIS-X+" kick1="BTN1" kick2="BTN2"/>
28     </controls>
29 </robotics>
```

Figura 83: Arquivo de Configuração

Outro arquivo de configuração disponível no RoS é o arquivo de time, que especifica todos os dados de time. Nele é possível selecionar o brasão do time, a bandeira, os nomes e posições dos jogadores, a camisa do time, o script de inteligência e até mesmo o nome do técnico. O objetivo desse arquivo é permitir que sejam criados outros times, tornando o jogo mais interessante. A Figura 84 mostra o arquivo de configuração do time do Flamengo.

```
1 <?xml version='1.0' encoding='iso-8859-1'?>
2 <robotics version='0.2.1'>
3   <team name='Flamengo' sigla='FLA' nplayers='11' >
4     <images basepath='textures/team/' imbrasso='flamengo-brasao.jpg' imcamisa='flamengo-body.jpg'></images>
5     <coach name='Mey Franco'></coach>
6     <players is='scripts/flamengo.lua' stupid='true'>
7       <player name='Bruno' p0x='0' p0z='-58' pfx='0' pfz='58' type='goleiro' number='1'></player>
8       <player name='Leonardo Moura' p0x='40' p0z='-40' pfx='0' pfz='58' type='zaga' number='2'></player>
9       <player name='Irineu' p0x='25' p0z='-32' pfx='0' pfz='58' type='zaga' number='3'></player>
10      <player name='Ronaldo Angelim' p0x='25' p0z='-32' pfx='0' pfz='58' type='zaga' number='4'></player>
11      <player name='Juan' p0x='40' p0z='-40' pfx='0' pfz='58' type='zaga' number='5'></player>
12      <player name='Paulinho' p0x='35' p0z='-15' pfx='0' pfz='58' type='meio' number='6'></player>
13      <player name='Claiton' p0x='7' p0z='-22' pfx='0' pfz='58' type='meio' number='7'></player>
14      <player name='Renato' p0x='7' p0z='-22' pfx='0' pfz='58' type='meio' number='8'></player>
15      <player name='Renato Augusto' p0x='35' p0z='-15' pfx='0' pfz='58' type='ataque' number='9'></player>
16      <player name='Roni' p0x='9' p0z='-9' pfx='0' pfz='58' type='ataque' number='10'></player>
17      <player name='Souza' p0x='9' p0z='-9' pfx='0' pfz='58' type='ataque' number='11'></player>
18    </players>
19  </team>
20</robotics>
```

Figura 84: Arquivo de configuração do time do flamengo

O RoS disponibiliza ainda os arquivos de tradução para que seja possível traduzir o jogo para outras línguas, os arquivos de script LUA, os arquivos de script GLSL (shaders).

Anexo 2 - Programa de Carga e Instalador



Figura 85: Tela principal do programa de carga

A modificação de algumas funcionalidades do RoS são muito corriqueiras e, por isso, modificá-las alterando os arquivos de configuração é um tanto antiquado. Para facilitar essa configuração, foi criado um pequeno programa de carga para o jogo. Esse programa foi desenvolvido com a biblioteca de widgets wxWidgets e compilado para o Windows e Linux.

Dentre as funcionalidades desse programa, temos a possibilidade de seleção do modo de jogo (Figura 85), essa função permite que o jogo rode em Full Screen (Tela cheia) em vários modos de exibição (800x600:32), ou funcione em uma janela do sistema operacional.

Além disso, podemos configurar dados como o arquivo de definição dos times, permitindo variar os times do jogo, mudar os controles, tanto as teclas quanto o dispositivo, além de habilitar funcionalidades gráficas, como efeitos de motion blur e shaders (Figura 86).



Figura 86: Telas de configuração

Além do programa de carga, foi criado um programa instalador do jogo. Esse programa foi desenvolvido com o software Inno Setup 5. O objetivo dele é facilitar a distribuição e implantação do jogo.

GLOSSÁRIO

A

Abstract Factory é um padrão de projeto de software. Este padrão permite a criação de famílias de objetos relacionados ou dependentes, através de uma única interface e sem que a classe concreta seja especificada.

API, de Application Programming Interface (ou Interface de Programação de Aplicativos) é um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades por programas aplicativos - isto é: programas que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços.

B

Billboards são imagens bidimensionais que são renderizadas como planos em cenas 3D, com o detalhe de sempre ficarem de frente à câmera virtual.

BrainStorming (ou "tempestade de idéias") é procedimento utilizado para encontrar solução para um problema através de uma série de idéias.

C

Callbacks são funções pré-definidas pelo programador para reagir à sinais ou eventos emitidos por algum outro sistema.

C for Graphics (CG) é uma linguagem de programação da NVIDIA derivada do ANSI C para suporte ao design gráfico, aproveitando os recursos do hardware NVIDIA. Usada principalmente para o desenvolvimento de algoritmos para pixel shaders e vertex shaders que são instruções específicas para os GPUs das placas de video.

CPU (*Central Processing Unit* em inglês, ou Unidade Central de Processamento), é a parte de um computador que interpreta e leva as instruções contidas no software.

F

Fragments ou Fragmentos são os possíveis pixels a serem mostrados na tela, sua exibição depende do tratamento feito pelo Pixel Shader.

Framework ou arcabouço é uma estrutura de suporte definida em que um outro projeto de software pode ser organizado e desenvolvido. Um framework pode incluir programas de suporte, bibliotecas de código, linguagens de script e outros softwares para ajudar a desenvolver e juntar diferentes componentes de um projeto de software.

G _____

Games é um sinônimo de jogos eletrônicos.

GLEW (OpenGL Extension Wrangler Library) é uma biblioteca de código livre e multiplataforma dedicada ao carregamento e compilação de shaders na linguagem GLSL.

GLSL (OpenGL Shading Language), ou também GLslang, é uma linguagem de alto nível para programação de shaders fortemente baseada na linguagem C.

GLU (OpenGL Utility Library) é uma biblioteca de extensão das funcionalidades gráficas do OpenGL.

GLUT (OpenGL Utility Toolkit) é uma biblioteca de funções para programas OpenGL, fornecendo suporte a I/O, controle de janelas e outros.

GPGPU (General-purpose computing on graphics processing units), ou também referido como GP²U é uma unidade de processamento gráfico que pode executar código específico, assim como uma CPU.

GUI (no Brasil, interface gráfica do usuário; abreviadamente, a sigla GUI, do inglês Graphical User Interface) é um mecanismo de interação homem-computador. Com um mouse ou teclado o usuário é capaz de selecionar esses símbolos e manipulá-los de forma a obter algum resultado prático. Esses signos são designados de widgets e são agrupados em kits.

H _____

HLSL (High Level Shader Language) é uma linguagem utilizada pelo Microsoft DirectX para programar vertex e pixel shaders.

I _____

I/O é um sigla para Input/Output, em português E/S ou Entrada/Saída. Este termo é utilizado quase que exclusivamente no ramo da computação, indicando entrada de dados por meio de algum código ou programa, para algum outro programa ou hardware, bem como a sua saída ou retorno de dados, como resultado de alguma operação de algum programa.

INI File (Initialization File) é um arquivo que armazena configurações de uma aplicação.

J _____

Jogabilidade é a característica que um jogo possui para ser fácil e intuitivo de se jogar.

L _____

LoD (Level of Detail) são técnicas que envolvem a redução da complexidade de cenas e objetos 3D a fim de reduzir o consumo de processamento.

Lua é uma linguagem de programação imperativa, procedural, pequena e leve, projetada para expandir aplicações em geral. Ela é um a linguagem script.

M _____

Memory leak, ou vazamento de memória, é um fenômeno que ocorre em sistemas computacionais quando uma porção de memória, alocada para uma determinada operação, não é liberada quando não é mais necessária. A ocorrência de vazamentos de memória é quase sempre relacionada a erros de programação e pode levar a falhas no sistema se a memória for completamente consumida.

Microsoft DirectX é uma coleção de APIs que tratam de tarefas relacionadas a programação de jogos para o sistema operacional Microsoft Windows, ou seja, é quem padroniza a comunicação entre software e hardware. O DirectX foi inicialmente distribuído pelos criadores de jogos junto com seus produtos, mas depois foi incluído no Windows.

Motor de jogo (ou game engine) é um programa de computador e/ou um conjunto de bibliotecas para simplificar o desenvolvimento de jogos ou outras simulações em tempo real, para videogames e computadores.

N

NES (Nintendo Entertainment System) é um videogame lançado pela Nintendo.

O

ODE (Ordinary Differential Equation) é a relação entre uma função $f(t)$ dependendo de variável t e das derivadas da função.

OGG é um formato livre de encapsulamento de multimédia orientado a stream. Ele pode ser lido e escrito numa mesma etapa, sem precisar armazenar todo ou grande parte do fluxo de dados antes. Essa característica é um requisito natural para streaming e processamento em pipelines.

OpenAL (Open Audio Library) é uma especificação definindo uma API multiplataforma de áudio. Ela é projetada para a reprodução eficiente em multicanais e com posicionamento 3D de audio.

OpenGL (Open Graphics Library) é uma especificação definindo uma API multiplataforma e multilínguagem para a escrita de aplicações capazes de produzir gráficos computacionais 3D (bem como gráficos computacionais 2D).

P

Pixel (Picture e Element, ou seja, elemento de imagem) é o menor elemento num dispositivo de exibição ao qual é possível atribuir-se uma cor. De uma forma mais simples, um pixel é o menor ponto que forma uma imagem digital.

Pixel Shader é shader que manipula pixels por meio de efeitos aplicados a cada um deles na tela.

Plotar é desenhar (uma imagem, especialmente um gráfico) baseando-se em informação fornecida como uma série de coordenadas.

R

Rasterizar é a operação de converter uma imagem vetorial em uma imagem bitmap.

Renderização é o processo pelo qual se podem obter imagens digitais. Este processo aplica-se essencialmente em programas de modelagem e animação (3ds Max, Maya etc.), como forma de visualizar a imagem final do projecto bidimensional ou tridimensional.

S

Shader na área da computação gráfica significa um conjunto de instruções de software, que serão utilizadas nem uma GPGPU para a produção de efeitos de renderização.

SNES (Super Nintendo Entertainment System) é um videogame lançado pela Nintendo.

Storyboard é um protótipo de uma seqüência de cenas cinematográficas muito utilizado na publicidade, animação e em cinema em geral.

Stream é um fluxo de dados. Quando um arquivo é carregado para ser editado, esta carga ocorre num fluxo, ou seja, linha a linha até o carregamento total do arquivo, como água a correr num cano ou bytes sendo lidos por um programa.

T _____

Toolkit é um conjunto de widgets, elementos básicos de uma GUI. Normalmente são implementados como uma biblioteca de rotinas ou uma plataforma para aplicativos que auxiliam numa tarefa.

V _____

Vertex shader é um shader capaz de trabalhar na estrutura de vértices do modelo 3D.

W _____

Widget é um termo sem tradução que designa componentes de interface gráfica com o usuário (GUI). Qualquer item de uma interface gráfica é chamada de widget, por exemplo: janelas, botões, menus e itens de menus, ícones, barras de rolagem, etc.

X _____

XML (eXtensible Markup Language) é uma recomendação da W3C para gerar linguagens de marcação para necessidades especiais. Ela é capaz de descrever diversos tipos de dados. Seu propósito principal é a facilidade de compartilhamento de informações através da Internet.

REFERÊNCIAS BIBLIOGRÁFICAS

- AbraGames.** 2004. Plano Diretor da Promoção da Indústria de Desenvolvimento de Jogos Eletrônicos no Brasil. *ABRAGAMES - Associação Brasileira das Desenvolvedoras de Jogos Eletrônicos*. [Online] 2004. [Citado em: 2007 de Dezembro de 01.]
http://www.abragames.org/docs/pd_diretrizesbasicas.pdf.
- Aguiar, C.E. e Rubini, G.** 2004. A aerodinâmica da bola de futebol. *Revista Brasileira de Ensino de Física*. 2004, Vol. 26, 1.
- AZEVEDO, Eduardo.** 2005. *Desenvolvimento de Jogos 3D*. Rio de Janeiro : Campus, 2005.
- BOOCH, Grady.** 1994. *Object-Oriented Analysis and Design with Applications*. s.l. : Addison-Wesley, 1994.
- BOURG, David M.** 2002. *Physics for Game Developers*. Sebastopol : O'REILY& Associates Inc, 2002.
- BUCKLAND, Mat.** 2005. *Programming Game AI by Example*. Plano : Wordware Publishing, Inc., 2005. 1-55622-078-2.
- BURDEN, L. Richard e FAIRES, J. Douglas.** 2003. *Análise Numérica*. São Paulo : Thomson Learning, 2003.
- Clua, Esteban W. G., et al.** 2007. The GPU Used as a Math Co-Processor in Real Time Applications. *Proceedings of VI Brazilian Symposium on Computer Games and Digital Entertainment*. 7-9 de Novembro de 2007, pp. 37-43.
- COHEN, Marcelo e MANSOUR, Isabel Harb.** 2006. *OpenGL: Uma Abordagem Prática e Objetiva*. São Paulo : Novatec, 2006.
- CONCI, Aura e AZEVEDO, Eduardo.** 2003. *Computação Gráfica: Teoria e Prática*. Rio de Janeiro : Campus, 2003.
- CORMEN, Thomas H., et al.** 2002. *Algoritmos: Teoria e Prática*. Rio de Janeiro : Campus, Elsevier, 2002.
- Creative Technology.** 2005. *OpenAL Programmer's Guide*. 2005.

- Entertainment Software Association. 2007.** *ESA - Entertainment Software Association.* [Online] 2007. [Citado em: 2007 de Dezembro de 01.]
<http://www.theesa.com/archives/files/ESA-EF%202007.pdf>.
- Fédération Internationale de Football Association - FIFA. 2007.** Regras do Jogo 2007/2008.
Confederação Brasileira de Futebol - CBF. [Online] 2007. [Citado em: 2007 de Setembro de 05.]
<http://www2.uol.com.br/cbf/regras/livroderegras.pdf>.
- FEIJÓ, Bruno, PAGLIOSA, Aristarco e CLUA, Esteban Walter Gonzalez. 2006.** Visualização, Simulação e Games. [A. do livro] K. BREITMAN e R. ANIDO. *Atualizações em Informática.* Rio de Janeiro : Editora PUC-Rio, 2006, pp. 127-186.
- FOLEY, James D., et al. 1997.** *Computer Graphics: Principles and Practice.* s.l. : Addison Wesley, 1997.
- HALLIDAY, David, RESNICK, Robert e KRANE, Kenneth S. 2004.** *Física 1.* 5ª Edição. Rio de Janeiro : LTC, 2004. Vol. 1.
- HALLIDAY, David, RESNIK, Robert e KRANE, Kenneth S. 2004.** *Física 2.* 5ª Edição. Rio de Janeiro : LTC, 2004. Vol. II.
- Hamilton, William Rowan. 1853.** *Lectures on Quaternions.* s.l. : Royal Irish Academy, 1853.
- HANSELMAN, Duane e LITTLEFIELD, Bruce. 2003.** *MatLab 6 - Curso Completo.* São Paulo : Prentice Hall, 2003. 85-87918-56-7.
- HEARN, Donald e BAKER, M. 2004.** *Computer Graphics with OpenGL.* 3ª Edição. Upper Saddle River, NJ : Pearson Prentice-Hall, 2004.
- Ierusalimschy, R., Figueiredo, L. H. de e Celes, W. 2006.** *Lua 5.1 Reference Manual.* s.l. : Lua.org, 2006. 85-903798-3-3.
- JUNG, Kurt e BROWN, Aaron. 2006.** *Beginning Lua Programming.* s.l. : Wiley Publishing, Inc., 2006.
- KUCHANA, Partha. 2004.** *Software Architecture Design Patterns in Java.* New York : Auerbach/CRC Press, 2004.
- LUNA, Frank D. 2003.** *Introduction to 3D Game Programming with DirectX 9.0.* s.l. : Wordware Publishing, Inc., 2003.
- Miller, Diane Disney. 2005.** *The Story of Walt Disney.* s.l. : Disney Editions, 2005. 978-0786855629.
- PALMER, Grant. 2005.** *Physics for Game Programmers.* New York : Apress, 2005.
- PRESSMAN, ROGER S. 2005.** *Software Engineering - A Practitioner's Approach.* s.l. : The McGraw-Hill Companies, Inc., 2005. 85-86804-57-6.
- RIBEIRO, Fábio. 2005.** *Design de Jogos: O Design sob a ótica da interatividade e do desenvolvimento de projeto.* Florianópolis : s.n., 2005.

- ROST, R. J. 2004.** *OpenGL(R) Shading language*. s.l. : Addison-Wesley, 2004.
- ROYCE, Winston. 1970.** Managing the Development of Large Software Systems: Concepts and Techniques. *Technical Papers of Western Electronic Show and Convention*. Agosto de 1970.
- SALEM, K. e ZIMMERMAN, E. 2004.** *Rules of play – Game Design fundamentals*. Cambridge, Massachusetts : The MIT Press, 2004.
- SCHILD, Herbert. 1997.** *C - Completo e Total*. 3ª Edição. São Paulo : Person Education, 1997. 85-346-0595-5.
- SHREINER, Dave, et al. 2005.** *The OpenGL Programming Guide - OpenGL Programming Guide*. s.l. : Addison-Wesley Professional, 2005.
- SISSOM, Leighton E. e PITTS, R. Donald. 1979.** *Fenômenos de Transporte*. Rio de Janeiro : LTC, 1979.
- STROUSTRUP, Bjarne. 1999.** *The C++ Programming Language*. s.l. : Addison Wesley Longman, Inc., 1999.
- WATT, Alan. 2000.** *3D Computer Graphics*, 3ª Edição. s.l. : Addison Wesley, 2000.
- Wikipédia, a encyclopédia livre.** Biblioteca (computação). *Wikipédia*. [Online] [Citado em: 16 de 09 de 2007.] [http://pt.wikipedia.org/wiki/Biblioteca_\(computa%C3%A7%C3%A3o\)](http://pt.wikipedia.org/wiki/Biblioteca_(computa%C3%A7%C3%A3o)).
- Wilson, Scott. 2003.** WAVE PCM soundfile format. *Music 422: Perceptual Audio Coding*. [Online] 2003. [Citado em: 2007 de Setembro de 15.] <http://ccrma.stanford.edu/courses/422/projects/WaveFormat/>.