MARIO MASSAKUNI KUBO

**FMMG: UM FRAMEWORK PARA JOGOS
MULTIPLAYER MÓVEIS**

**Tese apresentada à Escola Politécnica
da Universidade de São Paulo para
obtenção do Título de Doutor em
Engenharia.**

São Paulo

2006

MARIO MASSAKUNI KUBO

**FMMG: UM FRAMEWORK PARA JOGOS
MULTIPLAYER MÓVEIS**

**Tese apresentada à Escola Politécnica
da Universidade de São Paulo para
obtenção do Título de Doutor em
Engenharia.**

**Área de Concentração:
Engenharia de Computação e Sistemas
Digitais**

**Orientador:
Prof. Dr. Romero Tori**

São Paulo

2006

FICHA CATALOGRÁFICA

Kubo, Mario Massakuni

FMMG: um *framework* para jogos *multiplayer* móveis / M.M. Kubo. -- São Paulo, 2006.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Jogos 2.Jogos eletrônicos 3.*Frameworks* I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

*“Só posso levantar as mãos pro céu,
agradecer e ser fiel ao destino que Deus me deu.
Se não tenho tudo que preciso, com o que tenho,
vivo, de mansinho, lá vou eu.
Se a coisa não sai do jeito que eu quero,
também não me desespero, o negócio é deixar rolar.
E aos trancos e barrancos, lá vou eu,
e sou feliz e agradeço por tudo que Deus me deu.
E deixa a vida me levar (vida leva eu)”.*

(Zeca Pagodinho)

*Dedico este trabalho
às minhas obatian (in memorian) e meus odition (in memorian)
e aos meus pais, Natsue e Tatsumi,
pelo amor e incentivo dispensados ao longo de
toda a minha vida.*

Agradecimentos

*A Deus, por ter me dado a oportunidade de chegar até aqui,
e por ter me dado forças para continuar firme, mesmo em tempos de dificuldades.*

*Agradeço especialmente ao Prof. Dr. Romero Tori pela dedicação, ajuda,
incentivo, paciência para comigo e cooperação como
orientador deste trabalho, e principalmente pela sua valiosa amizade.*

*À Mariza Ushijima Leone, pelo apoio fornecido através da
secretária do laboratório Interlab.*

*A todos os professores e funcionários da Engenharia Elétrica da Escola
Politécnica da USP, departamento de Engenharia da Computação e
Sistemas Digitais pela criação de um ambiente amigável
e agradável de trabalho, e pelas disciplinas, cursos,
workshops e seminários que me ajudaram na minha
formação profissional durante esses anos.*

*Agradeço especialmente ao meu amigo de vários anos
Renato de Moraes Dezidério pelas valiosas contribuições,
apoio para a realização do trabalho e pela amizade verdadeira de carinho.*

*Aos amigos Fernando (Gom), Eduardo (Dú), Guilherme (Guila),
Gustavo (Beterraba), Elenilson, Alexandre (Mori), Vera, Danielle (Dani),
Elby (Japoneis), Gustavo (Leite), Clara, Laura e Mauricio (Borga) e outros,
tantos que gostaria que me perdoassem se não aparecem nesta lista,
mas que estarão sempre nas minhas lembranças
pela amizade e apoio durante o trabalho.*

*Aos amigos Luiz Carlos (Jiraya), Ana Paula, Marcos (Xuxa),
Edson (Mura), Sergio (Cat), Sandro, Bianchi (Bibi), Elisangela (Eli),
Kássia, Salvador (Pre), Giovane (Gi), Marcio, Ana Claudia (Claudinha),
Henrique, Ildeberto (Beto), José Gustavo (Zé) e Wesley (Lelo)
pela amizade, o convívio de vários anos e
principalmente durante toda essa etapa de doutorado.*

*A Jaci Kazue Horiute e Luiza Zanati Bárias pelos constantes
apoio recebidos através de fé, amizade, amor e carinho.*

Aos meus companheiros e amigos de Brasília, São Paulo, São Carlos e Lins, pelos momentos de alegria, reflexão, estudo, descontração e divertimento.

Aos meus pais, pelo incentivo, ajuda e amor recebidos durante toda a minha vida, e pelo ensinamento do respeito às outras pessoas, que é a base da realização deste e de outros trabalhos da minha vida.

Aos meus irmãos pelo carinho e amizade eterna Seiko, Massato, Yashiyo, Miyuri e Mitie, meus sobrinhos Miyuki, Ayumi, Yukari, Kenji e Yoshiaki, e por fim a minha cunhada e meus cunhados.

Por fim, agradeço a todos aqueles que direta ou indiretamente contribuíram para este trabalho, pois sozinho jamais seria possível a sua realização.

A todos vocês, o mais sincero obrigado.
Kubo, Mario Massakuni

SUMÁRIO

Resumo	I
Abstract	II
Capítulo 1 - Introdução	1
1.1 Motivação	1
1.2 Objetivos	4
1.3 Organização da Tese	4
Capítulo 2 - Jogos	6
2.1 Introdução	6
2.2 Classificação dos Jogos	7
2.2.1 Tipos de Mídia	7
2.2.2 Interface	7
2.2.3 Objetivo	8
2.2.4 Número de Jogadores	9
2.3 Game Engines	10
2.4 Conclusão do Capítulo	12
Capítulo 3 - Jogos Multiplayer	13
3.1 Introdução	13
3.2 Requisitos	14
3.3 Componentes dos Jogos Multiplayer	16
3.4 Características Técnicas	17
3.4.1 Largura de Banda	17
3.4.2 Protocolos de Comunicação	19
3.4.3 Modelos de Comunicação	21
3.4.4 Estruturas Básicas de Comunicação	24
3.4.5 Formas de Distribuição e Armazenamento dos Dados	25
3.4.6 Formação de Grupos	27
3.4.7 Técnicas de Redução de Tráfego de Comunicação	27
3.4.8 Atraso	28
3.5 Conclusão do Capítulo	30
Capítulo 4 - Framework	31
4.1 Introdução	31
4.2 Definições	32

4.3 Classificação	32
4.4 Vantagens e Desvantagens da Utilização de Frameworks	34
4.5 Documentação de Frameworks	35
4.6 Metodologias de Desenvolvimento de Frameworks	36
4.7 Características Técnicas de Frameworks para Jogos	38
4.7.1 Requisitos	38
4.7.2 Arquitetura	39
4.8 Trabalhos Relacionados	39
4.8.1 Amphibian	39
4.8.2 wGEM	41
4.8.3 FORGE V8	42
4.8.4 FORGE 16V	43
4.8.5 FreeMMG	44
4.8.6 FunGen	45
4.8.7 Guff	46
4.9 Conclusão do Capítulo	47
Capítulo 5 - Framework FMMG	49
5.1 Introdução	49
5.2 Metodologia	50
5.3 Requisitos	51
5.4 Arquitetura	52
5.4.1 Controller	54
5.4.2 Model	56
5.4.3 View	57
5.4.4 Funcionamento do Framework FMMG	58
5.5 Estruturação do Framework	62
5.5.1 Módulos de Gerenciamento	64
5.5.2 Módulo Básico	66
5.5.3 Módulo de Rede	70
5.6 Conclusão do Capítulo	76
Capítulo 6 - Aplicabilidade do Framework FMMG: Estudo de Caso	78
6.1 Infraestrutura Utilizada	78
6.1.1 BREW	78
6.1.2 Plataforma Utilizada	80

6.2 Estudo de Caso: Caça ao Tesouro	81
6.2.1 Enredo do Jogo	81
6.2.2 Requisitos	82
6.2.3 Visualização	82
6.2.4 Configuração de Gráficos	83
6.2.5 Estrutura de Estado do Jogo	84
6.3 Arquitetura	85
6.4 Comunicação de Rede	87
6.5 Conclusão do Capítulo	101
Capítulo 7 - Conclusão	102
7.1 Considerações Iniciais	102
7.2 Contribuições	105
7.3 Perspectivas e Trabalhos Futuros	105
7.3 Publicação de Artigos em Eventos Nacionais e Internacionais	106
Referências Bibliográficas	107

Índice de Figuras

Figura 5.1 - Infraestrutura MVC	53
Figura 5.2 - Classe <i>Controller</i>	55
Figura 5.3 - Classe <i>Model</i>	57
Figura 5.4 - Classe <i>View</i>	58
Figura 5.5 - Estrutura do Padrão MVC pelo <i>Framework</i>	59
Figura 5.6 - Classe <i>DataObject</i>	59
Figura 5.7 - Método <i>changeState</i>	60
Figura 5.8 - Método <i>run</i>	60
Figura 5.9 - Exemplo da Utilização do Padrão MVC em um Modelo de Jogo	61
Figura 5.10 - Estrutura de Comunicação entre os Módulos	64
Figura 5.11 - Classes do Módulo de Gerenciamento	65
Figura 5.12 - Classes do Módulo de Processamento	66
Figura 5.13 - Classes do Módulo de Formato	68
Figura 5.14 - Classes do Módulo de Apresentação	69
Figura 5.15 - Estrutura do Módulo Cliente	72
Figura 5.16 - Diagrama de Classe do Módulo Cliente	74
Figura 5.17 - Estrutura do Módulo Servidor	75
Figura 5.18 - Diagrama de Classe do Módulo Servidor	76
Figura 6.1 - Caça ao Tesouro	81
Figura 6.2 - Caçadores	82
Figura 6.3 - Tesouro	83
Figura 6.4 – Cenário do Jogo	83
Figura 6.5 - Menu Principal do Jogo	84
Figura 6.7 - Exemplo da Utilização do Padrão MVC e Estrutura de Estados no Jogo Caça ao Tesouro	86
Figura 6.8 - Exemplo da Utilização do Padrão MVC e Estrutura de Estados do <i>GamePlayState</i> do Jogo Caça ao Tesouro	87
Figura 6.9 - Criar Túnel TCP ou UDP	88
Figura 6.10 - Cliente Solicita Conexão	88
Figura 6.11 - Envia o Objeto <i>socket</i>	88
Figura 6.12 - Recebe o Objeto <i>socket</i>	89
Figura 6.13 – Diagrama Macro de Caso Caça ao Tesouro	89
Figura 6.16 - Conexão com o Servidor do Jogo	90

Figura 6.15 – Comunicação com o Servidor do Jogo	91
Figura 6.16 - Função <i>open</i>	93
Figura 6.17 - Função <i>write</i>	94
Figura 6.18 - Função <i>read</i>	95
Figura 6.19 - Função <i>close</i>	96
Figura 6.20 - Função <i>login</i>	97
Figura 6.21 - Função <i>execute</i>	99
Figura 6.22 - Função <i>logout</i>	100

Abreviaturas

1xRRT - *1 x Radio Transmission Technology*

2.5 G - duas gerações e meia

2^{1/2} D - duas dimensões e meia

2D - duas dimensões

3D - três dimensões

3G - terceira geração

ACKs - protocolos “orientados ao transmissor”

API - *Application Programming Interface*

BDS - *BREW Distribution System*

BREW - *Binary Runtime Environment for Wireless*

BREW SDK - *BREW Software Development Kit*

CAVE - *Cave Automatic Virtual Environment*

CDC - *Connected Device Configuration*

CDMA - *Code Division Multiple Access*

CLDC - *Connected, Limited Device Configuration*

CSD - *Circuit Switched Data*

FMMG - *Framework for Mobile Multiplayer Games*

Gbps - Giga bits por segundo

GHz - *Giga Hertz*

GPRS - *General Packet Radio Service*

GSM - *Global System for Mobile*

GUI - *Graphical User Interface*

HMD - *Head Mounted Display*

HTTP - *HyperText Transfer Protocol*

HTTPS - *HyperText Transfer Protocol Secure*

Hz - *Hertz*

IA - *Inteligência Artificial*

IEEE - *Institute of Electrical and Electronics Engineers*

INTERLAB - *Laboratório de Tecnologias Interativas*

IP - *Internet Protocol*

J2ME - *Java 2 Micro Edition*

Kbps - *Kilobits por segundo*

KVM - *Kilobyte Virtual Machine*

LAN - *Local Area Network*

Mbps - *Mega Bits por Segundo*

MIDP - *Mobile Information Device Profile*

MMMOG - *Mobile Massive Multiplayer Online Games*

MMOG - *Massive Multiplayer Online Games*

MMUG - *Massively Multiplayer Ubiquitous Game*

ms - *milisegundos*

MVC - *Model View Controller*

NACKs - *protocolos “orientados ao receptor”*

ORB - *Object Request Broker*

OSI - *Open System Interconnection*

PC - *Personal Computer*

RMP - *Reliable Multicast Protocol*

RPG - *Role Playing Game*

SDL - *Simple DirectMedia Layer*

SMS - *Short Message Service*

TCP - *Transmission Control Protocol*

TCP/IP - *Transmission Control Protocol/Internet Protocol*

TDMA - *Time Division Multiple Access*

TibiaME - *Tibia Micro Edition*

TV - Televisor

UDP - *User Datagram Protocol*

USP - Universidade de São Paulo

WAN - *Wide Area Network*

wGEM - *Wireless Game Engine for Mobile Devices*

WLAN - LANs sem fio

XML - *eXtensible Markup Language*

Resumo

A área de jogos para dispositivos móveis vem despertando o interesse cada vez maior de diversas pessoas, seja na área acadêmica ou de empresas comerciais, principalmente quando o desafio é desenvolver jogos multiusuário. No entanto, apesar do crescimento no número de *frameworks e engines*, e da ampliação das pesquisas nessa área, não existe ainda uma arquitetura padrão para o desenvolvimento de jogos para dispositivos móveis, devido principalmente à complexidade do problema, à grande diversidade de tipos de dispositivos móveis disponíveis no mercado e aos requisitos que diferem para cada tipo de jogo.

O trabalho de pesquisa e desenvolvimento consistiu em, a partir de um estudo sobre o estado da arte e as necessidades de usuários e desenvolvedores, se desenvolver um *framework* para Jogos *Multiplayer* Móveis, denominado FMMG (*Framework Mobile Multiplayer Games*). Como resultado a Tese apresenta uma descrição detalhada da arquitetura do *framework* FMMG e dos módulos propostos, mostrando a integração entre eles, bem como as principais inovações que foram produzidas. A arquitetura do FMMG se baseia no padrão MVC (*Model View Controller*) e em uma estrutura de estado do jogo de acordo com os módulos preestabelecidos de gerenciamento, básico e rede. O *framework* resultante desta Tese visa dessa forma disponibilizar um conjunto de componentes que auxiliem na implementação de Jogos *Multiplayer* Móveis.

Por fim, é apresentado um estudo de caso para a validação do *framework* desenvolvido, o qual demonstra o funcionamento dos principais recursos oferecidos.

Abstract

The branch of mobile device games has drawn increasing interest of both universities and industries, specially when it comes to the development of multiplayer games. However, despite the growing number of frameworks and engines, as well of researches in this field, no standard architecture for the development of mobile device games has been established. Such situation can be explained by the complexity of the problem, the vast diversity of mobile devices currently in the market, and the different requirements of each type of game.

The research and the elaboration of this project consisted of the development of a framework for Mobile Multiplayer Games called FMMG, based on both a study of the state of art and the needs of users and developers. As a result, the thesis presents a detailed description of both the FMMG framework architecture and of the modules proposed, showing the integration between them, as well as the main innovations produced during the development of this project. The FMMG architecture is based on the MVC (Model View Controller) standard and on a state-of-game structure in accordance with the pre-established management modules: basic and network. The goal of the resulting framework presented hereby is thus to offer a set of components that assists the implementation of Mobile Multiplayer Games.

Finally a case is presented to validate the framework under consideration, demonstrating the functioning of the main features available.

Capítulo 1 - Introdução

1.1 Motivação

A área de jogos está em franca expansão, movimentando anualmente bilhões de dólares no mundo todo. Um grande fator para esse sucesso, além do enorme interesse do público, é a diversidade de equipamentos com características próprias que vêm sendo criados continuamente ao longo desse tempo, de PC's a videogames, com destaque especial aos dispositivos móveis.

Apesar do grande avanço observado na área de jogos, há ainda aspectos importantes que desafiam os pesquisadores. Esses aspectos estão principalmente ligados à heterogeneidade de plataformas de desenvolvimento, linguagens de programação, *hardware*, formatos do banco de dados e programas aplicativos; o formato e o controle em que os dados devem ser transferidos entre máquinas diferentes; gerenciamento de uma aplicação e seu sistema, considerando-se a diversidade de ambientes; ergonomia; reutilização de *software* por meio de *framework*, arquiteturas e plataformas, as limitações de desempenho das linhas de comunicação, memória dos dispositivos; a heterogeneidade de sistemas operacionais, protocolos e equipamentos; desenvolvimento e análise de desempenho de protocolos de comunicação de dados em redes heterogêneas; forma dos serviços de suporte para aplicações (jogos, educação, trabalho cooperativo, engenharia de *software*, realidade virtual, multimídia, negócios, entretenimento e etc.), entre outras.

A área de jogos, e principalmente as aplicações de jogos para dispositivos móveis são extremamente estimulantes pelos desafios tecnológicos que apresentam e pelo potencial de aplicabilidade. O fato de esta ser uma área emergente e multidisciplinar, e de apresentar várias visões com requisitos diferentes, faz com que não existam ainda muitos sistemas disponíveis que tenham respostas para tantas questões em aberto. Isto torna os jogos multiusuário para dispositivos móveis, denominada Jogos *Multiplayer* Móveis, uma extensa área de pesquisa em: *framework*, *engine*, arquiteturas, redes, sistemas de computação distribuídos, entre outros. Além disso, ainda há espaço de pesquisa em outras áreas coincidentes a jogos não *multiplayer*, como dispositivos de

entrada/saída mais precisos, interface homem-computador, computação gráfica, entre outras.

O desenvolvimento de Jogos *Multiplayer* Móveis é uma tarefa ainda mais desafiadora do que desenvolver jogos para PC's ou *consoles*. Além da dificuldade de desenvolver o jogo em si, o desenvolvedor deve se ater a vários problemas inerentes a estes dispositivos, como limitação de memória, recursos e processamento, tamanho da tela e mecanismo de entrada não apropriada (Brooks, 1987) (Bethke, 2003).

No entanto, existem vários elementos em comum que são necessários ao desenvolvimento de todos os tipos de jogos, como: detecção de colisão, animação, renderização da tela, entradas do usuário, modelagem e animação dos objetos, inteligência artificial, comunicação de rede, algoritmos, entre outros. A partir da identificação desses elementos comuns, é possível unificá-los em um componente de *software*, que seja genérico o suficiente para permitir sua utilização pelos mais variados tipos de jogos, mas ao mesmo tempo seja extremamente otimizado para garantir um bom desempenho, qualidade de *software* e diminuição do tempo de desenvolvimento. Esse é o papel fortemente aplicado aos conceitos de *framework* da engenharia de *software*.

Motivou o desenvolvimento desta Tese o fato de que vários dos aspectos citados anteriormente, ainda não estão bem conceituados, no que se refere à área de Jogos *Multiplayer* Móveis, combinado com a ausência de uma estrutura de componentes efetivamente consolidada para desenvolvimento do jogo e com os problemas inerentes das aplicações multiusuária.

Neste contexto, a Tese propõe o desenvolvimento de um *framework* para Jogos *Multiplayer* Móveis, denominado *Framework for Mobile Multiplayer Games* (FMMG). O uso do *framework* FMMG proporciona a comunicação distribuída entre os usuários em um mesmo ambiente de jogo por meio de dispositivos móveis, com os seguintes objetivos:

- Facilitar o reuso de *software*;
- Facilitar a portabilidade dos Jogos *Multiplayer* Móveis;

- Prover um conjunto de componentes;
- Fornecer mecanismos transparentes de comunicação do modelo cliente/servidor;
- Desenvolver mecanismos de comunicação para melhorar a interação entre os diversos usuários de dispositivos móveis.

A avaliação do *framework* FMMG, para ambiente multiusuário, foi realizada por meio do desenvolvimento de um jogo especialmente especificado, visando-se validar aspectos críticos do *framework* proposto.

Diante da diversidade de dispositivos móveis existentes, vale ressaltar que o desenvolvimento do *framework* e dos protótipos de jogo acima citados será realizado apenas sobre plataformas de telefones celulares de terceira geração (3G).

A Tese destaca-se por abordar problemas de infraestrutura de desenvolvimento de componentes relacionados a jogos em dispositivos móveis, principalmente na definição de um *framework* e pela integração com as demais áreas de conhecimento. A originalidade do *framework* FMMG é a utilização do padrão MVC (*Model View Controller*) e o mecanismo de estrutura de estados do jogo definido no Capítulo 5.

A Tese enquadra-se em uma área de pesquisa bem recente, denominada jogos computacionais, que vem despertando muito interesse da comunidade científica e de empresas, por revolucionar substancialmente as aplicações de entretenimento digital. No Brasil, a área de desenvolvimento de jogos ainda está em processo de evolução, mas é bastante promissora do ponto de vista de empresas, universidades e em outras áreas do conhecimento.

No Laboratório de Tecnologias Interativas da Escola Politécnica da USP (INTERLAB), onde esta Tese foi desenvolvida, há uma linha de pesquisa em entretenimento digital na qual estão sendo realizados diversos trabalhos relacionados a desenvolvimento de jogos, como os apresentados em (Tori & Nakamura, 2002), (Favaro et al., 2003), (Bianchini, 2005) e (EnJine, 2006).

1.2 Objetivos

Os objetivos desta Tese são:

- Propor e especificar um *framework* para Jogos *Multiplayer* Móveis, denominado FMMG (*Framework for Mobile Multiplayer Games*), bem como sua arquitetura de implementação;
- Oferecer um *framework* para facilitar a criação de jogos para dispositivos móveis e proteger o desenvolvedor das características específicas das plataformas e da implementação das rotinas básicas da maioria dos jogos;
- Contribuir com a consolidação da pesquisa e com a geração de novas competências científicas e tecnológicas nas áreas abrangidas pelo projeto.

1.3 Organização da Tese

A Tese está organizada em sete capítulos, descritos resumidamente a seguir:

O Capítulo 2 apresenta o estado da arte relacionado ao desenvolvimento de jogos, suas definições e classificações enfocando questões sobre *engines*.

O Capítulo 3 apresenta os conceitos, requisitos, componentes dos jogos *multiplayer* e descreve as suas características técnicas, em relação à largura de banda, protocolos, modelos, estruturas básicas de comunicação, formas de distribuição e armazenamento dos dados, formação de grupos, técnicas de redução de tráfego de mensagens, atraso e processos.

O Capítulo 4 discute conceitos relacionados aos *frameworks* que são relevantes ao Projeto.

O Capítulo 5 apresenta a proposta do *framework* FMMG, descrevendo os seus requisitos, sua arquitetura e os detalhes dos componentes.

O Capítulo 6 retrata a implementação do estudo de caso, apresentando as ferramentas, códigos, os alguns desafios na implementação, além da avaliação do protótipo, baseado no *framework* FMMG.

O Capítulo 7 apresenta a síntese e as conclusões obtidas, bem como contribuições geradas pelo trabalho, e também enfatiza alguns tópicos importantes para futuros projetos de pesquisa.

Capítulo 2 - Jogos

Atualmente vivemos na chamada era digital. Os jogos de computadores ganharam muito espaço no dia a dia, criando uma grande indústria e o seu próprio mercado.

Este capítulo descreve o estado da arte e os principais conceitos de jogos de computadores, os tipos de mídias e classificações existentes.

2.1 Introdução

Segundo Battaiola (Battaiola, 2000), “um jogo de computador pode ser definido como um sistema interativo que permite ao usuário experimentar, uma situação de conflito. Quase sempre, como plano de fundo dessa situação, existe um enredo, que define do que se trata o jogo, as regras que o jogador deve seguir e os objetivos que ele deve se esforçar para atingir”.

Um jogo de computador pode ser definido também como um sistema composto de três partes básicas: enredo, interface interativa e *engine (framework)*. O sucesso de um jogo está associado à combinação perfeita destes componentes (LaMothe et al., 1994) (Battaiola et al., 2001).

O enredo define o tema, a trama e os objetivos do jogo que através de uma série de passos o jogador deve se esforçar para atingir. A definição da trama não envolve apenas a criatividade e a pesquisa sobre o assunto, mas também a interação com pedagogos, psicólogos e especialistas no assunto.

A interface interativa controla a comunicação entre o *engine* e o usuário, apresentando um novo estado do jogo. O desenvolvimento da interface envolve aspectos artísticos, cognitivos e técnicos.

O *engine* do jogo é o seu sistema de controle, o mecanismo que controla a reação do jogo em função de uma ação do usuário. A implementação do *engine* envolve diversos aspectos computacionais, tais como, a escolha apropriada da linguagem de programação em função de sua facilidade de uso, objetivo, desempenho e portabilidade, o desenvolvimento de algoritmos específicos, o tipo de interface com o usuário, etc.

2.2 Classificação dos Jogos

2.2.1 Tipos de Mídia

Os jogos de computadores vêm se tornando cada vez mais populares e estão presentes em um grande número de plataformas e distribuídos por diversas mídias. Quanto ao tipo de mídia utilizada podem ser classificados como (Battaiola et al., 2001):

- *Consoles* - Equipamento digital dotado de um conector para o aparelho de TV e um encaixe para algum tipo de unidade de armazenamento contendo programas e/ou dados de um jogo em particular;
- PC - Aplicativos para serem jogados em computador. São geralmente vendidos em grandes redes varejistas e lojas especializadas, ou obtidos *online*, com a possibilidade de serem jogados pela *internet*;
- Aparelho celular - Jogos que podem ser executados em telefones celulares com capacidade para processar imagens em 2D, 2 1/2 D ou 3D;
- *Handhelds* e *consoles* portáteis - Essas plataformas possuem telas de maior tamanho e melhor qualidade que celulares, possibilitando o acesso a jogos de melhor resolução gráfica;
- TV Digital - Jogos que podem ser executados em aparelhos de TV digital como forma de conteúdo interativo;
- Multiplataforma - Outra possibilidade é que o jogo seja destinado a mais de uma plataforma. Assim, poderiam atingir diferentes nichos tecnológicos, tais como computadores pessoais, celulares, *handhelds*, *internet*, videogames ou qualquer outro dispositivo.

2.2.2 Interface

A interface define uma forma particular de atuação do usuário no universo do jogo. Quanto ao tipo de interface os jogos podem ser classificados como:

- Primeira pessoa - Quando a visão que se tem da cena é através da visão de uma

das personagens do jogo e esta visão é controlada diretamente pelo jogador, ou seja, a cena exibida representa o que os olhos do jogador vêem;

- Terceira pessoa - Em que a visão da cena é externa a qualquer uma das personagens existentes, ou seja, o jogador se vê na cena.

2.2.3 *Objetivo*

A classificação de jogos de computadores pode variar bastante, em função do objetivo e das características que se deseja ressaltar. A tendência hoje é de se criar jogos que combinem características de diferentes tipos clássicos de jogos, visando-se oferecer produtos inovadores. Uma das classificações mais tradicionais é a seguinte (SOFTEX, 2005) (Battaiola, 2000):

- Histórias Interativas - RPG (*Role Playing Game*) é o tipo de jogo que consiste em uma ambientação contextual e tem por objetivo criar no computador um ambiente capaz de reproduzir um ambiente real;
- Estratégia - São jogos que tratam de raciocínio e podem ser jogados em tempo real; geralmente apresentam um enredo que envolve guerra;
- Simulação - São jogos que possuem níveis de complexidade variando de média a alta, em razão da grande quantidade de comandos necessários para jogar. Geralmente, respeitam, ou tentam respeitar, de forma fiel, o tratamento das leis da física;
- Ação - Os jogos de ação têm como características principais a facilidade de operação, em virtude das poucas teclas necessárias, objetivos bem definidos, histórias simples e exigências de bons reflexos;
- Aventuras (*adventures*) - Esse tipo de jogo, muito popular entre os micros da geração 8 *bits*, tinha como peculiaridade a interatividade da sua interface. Inicialmente o jogador digitava os textos para que as ações pudessem ser interpretadas. Geralmente, o jogo não possuía gráficos, apenas textos. Em evoluções desse tipo de jogo a interface já era composta pelas palavras, que poderiam ser selecionadas com o mouse e executadas com um clique num objeto

na tela, apresentando, assim, gráficos estacionários ou com pouca animação. Esse tipo de jogo se caracteriza pela exigência de raciocínio lógico e pode ser classificado, também, como uma história interativa;

- **Infantis** - Jogos infantis se caracterizam por sua interface simples e seu caráter educacional, esse tipo de jogo pode apresentar características de quaisquer dos jogos acima citados;
- **Passatempo** - São programas, de níveis simples a complexos, que têm como objetivo essencial de atingir uma pontuação alta. Esses jogos têm, usualmente, interfaces 2D;
- **Esporte** - São programas que simulam esportes, como os jogos de futebol, futebol americano, vôlei, basquete, boxe, etc. Como o usuário comanda times inteiros, os jogos têm interfaces 3D que, em geral, fazem uso extensivo de *sprites* 3D. Em termos de programação, os *engines* desses jogos têm os mesmos problemas dos simuladores porque a ação é normalmente em tempo real;
- **Educação/Treinamento** - Jogos educacionais podem envolver as características de qualquer um dos jogos anteriores. O que diferencia os jogos de educação ou treinamento dos jogos somente para diversão é que eles levam em conta critérios didáticos e pedagógicos associados aos conceitos que visam difundir;
- **Advergames** - Jogos interativos disponíveis gratuitamente em sites institucionais de empresas, visando promover marcas, produtos e atrair mercado. Como os jogos educacionais, estes também podem envolver as características de qualquer um dos jogos anteriores. As tecnologias utilizadas nos jogos são acessíveis à maioria dos usuários e utilizam técnicas de marketing *online*.

2.2.4 Número de Jogadores

Os jogos são classificados quanto ao número de jogadores em: jogos *monoplayer* ou *multiplayer*. Os jogos *monoplayer* são aqueles em que apenas um jogador participa do jogo, enquanto os jogos *multiplayer* envolvem dois ou mais jogadores.

Os jogos *multiplayer* podem ser também caracterizados como Jogos *Online*

Massivamente Multiusuário (MMOG - *Massive Multiplayer Online Games*), nos quais milhares de jogadores interagem concorrentemente em um mundo virtual persistente. Estes jogos ocorrem continuamente, sem interrupções, onde partes dos jogadores possam se ausentar, enquanto outros permanecem jogando. Exemplos de MMOG compreendem Ultima Online (Origin, 2005), EverQuest (Sony, 2005) e Lineage (NCSoft, 2005).

A união entre MMOG e jogos móveis cria um novo cenário, onde milhares de usuários interagem através de dispositivos móveis, criando os chamados Jogos Online Massivamente Multiusuário para Dispositivos Móveis (MMMOG - *Mobile Massive Multiplayer Online Games*). Por diversas razões, esta integração ainda não aconteceu totalmente. A grande maioria dos jogos disponíveis hoje para aparelhos celulares no mercado mundial consiste em aplicações de usuário único e de execução *offline*, remetendo aos jogos que surgiram no mercado de PCs na década de 1980. Entretanto, ao analisar a evolução natural dos serviços sem fio e os mercados mais avançados, como Ásia e Europa, percebe-se que há um consenso em torno da necessidade e do potencial comercial de se produzir jogos MMOG. Esta plataforma herda os mesmos problemas de um MMOG tradicional, como necessidade de garantia de robustez, escalabilidade, comunicação em tempo real, segurança e tolerância à falhas. Neste sentido, as primeiras experiências já estão sendo realizadas, através de jogos como TibiaME (Tibia Micro Edition, 2005) e Samurai Romanesque (Dwango, 2005).

Atualmente, surgem os Jogos Massivos Ubíquos (MMUG - *Massively Multiplayer Ubiquitous Game*), os quais abrangem um conjunto de cenários onde jogadores interagem entre si e com o mundo virtual através de diferentes dispositivos, de formas adaptadas aos seus contextos de execução, criando uma idéia do usuário nunca estar completamente desconectado do jogo. Os desafios tecnológicos e científicos para realização de um MMUG são grandes e terão impacto em temas como interatividade, comunidades virtuais, ubiquidade e convergência digital, que serão estratégicos para os próximos anos.

2.3 Game Engines

Os *engines* podem ter as seguintes definições:

- Um *engine* define uma coleção de módulos de código de simulação que não especificam diretamente o comportamento ou o ambiente do jogo (Lewis & Jacobson, 2002);
- *Engine* é um conjunto de componentes de *softwares*, integráveis e reutilizáveis, que fornecem serviços a serem utilizados em um jogo (Bernades et al., 2004).

Segundo Bernades et. al. (2004), o *engine* atua como uma camada intermediária entre os elementos que caracterizam um jogo em particular (conteúdo do jogo e regras) e a plataforma em que o jogo é executado.

Pode-se classificar um *engine* de acordo com os recursos e facilidades que oferece para o desenvolvimento. A seguir estão alguns critérios que são importantes para engines de jogos (Battaiola et al., 2001):

- Facilitar o desenvolvimento de um tipo de jogo específico;
- Possibilitar o cálculo de portais da cena e de mapas de iluminação;
- Permitir que o programador possa escrever uma série de funções relacionadas a ações e eventos do jogo numa linguagem de mais alto nível;
- Beneficiar-se dos recursos disponíveis no *hardware* gráfico, por meio de uma camada de *Application Programming Interface* (API);
- Facilitar o desenvolvimento de um jogo para vários jogadores;
- Possibilitar que se possam construir, com facilidade, bibliotecas de objetos, funções e mapas, de forma a poder reutilizá-las com facilidade ao programar no mesmo jogo ou em outros jogos;
- Disponibilizar ferramentas para interagir com outros programas de modelagem;
- Facilitar que projetistas possam elaborar jogos, sem estarem preocupados com programação.

O desenvolvimento de um *engine* é um campo em aberto e a falta de padronização

prejudica a portabilidade e otimização de recursos a serem utilizados nos jogos (Corley, 1998). Esforços têm sido realizados no sentido de se definir arquiteturas padrões, genéricas e flexíveis, bem como de se associar aos *game engines* técnicas de engenharia de *software*, de forma a garantir níveis satisfatórios de desempenho, robustez, modularidade, reusabilidade de código, etc. (Madeira, 2001).

Atualmente, o que encontramos são *game engines* construídos sob medida para um determinado tipo de jogo. O objetivo principal do *engine* é proporcionar a diminuição do tempo e custo, aumentando a qualidade e a produtividade de desenvolvimento de jogos.

2.4 Conclusão do Capítulo

Neste capítulo, foram apresentados os conceitos e definições encontrados na literatura relacionados a jogos. As classificações de jogos podem ter variações de acordo com o tipo de jogo, nível de entretenimento e arquitetura. Desta forma, relatamos as classificações de jogos de acordo com tipos de mídia, interface, objetivo e número de jogadores.

Foram também apresentados conceitos sobre *engine* de jogos que tem como premissa a definição e a utilização de componentes de *software*, reutilizável, padronizados, integrados e portáteis.

Capítulo 3 - Jogos Multiplayer

Este capítulo apresenta adaptações dos conceitos fundamentais dos ambientes virtuais distribuídos para os jogos *multiplayer*, compondo assim, as bases teóricas para o desenvolvimento da Tese.

As seções deste capítulo definem os conceitos, requisitos e componentes dos jogos *multiplayer* e descrevem suas características técnicas em relação à largura de banda, protocolos, modelos, estruturas básicas de comunicação, formas de distribuição e armazenamento dos dados, formação de grupos, técnicas de redução de tráfego de mensagens, atraso e processos independente de plataforma para jogos computacionais baseados em PC's ou dispositivos móveis.

3.1 Introdução

As aplicações de jogos podem ser vistas sob um aspecto bastante amplo, variando de um único jogador, usando um único dispositivo, até muitos jogadores, usando um sistema multiusuário, denominado jogos *multiplayer*.

Segundo Battaiola et al. (2001), em um jogo *multiplayer*, cada jogador possui a sua própria visão de estado do jogo, sendo que esta deve ser consistente, o mais semelhante possível para todos os jogadores.

Os jogos *multiplayer* envolvem estudos e recursos ligados com percepção, *hardware*, *software*, interface do usuário, fatores humanos e aplicações. Para a elaboração de um jogo *multiplayer* é necessário ter algum domínio sobre: dispositivos, convencionais ou não, de entrada e saída, computadores ou processadores de alto desempenho, processamento gráfico, sistemas paralelos e multiusuário, modelagem geométrica, simulação em tempo real, interação, detecção de colisão, avaliação, impacto social, projeto de interfaces, e aplicações simples e distribuídas.

Os jogos *multiplayer* devem promover a integração dos recursos computacionais com o jogador, de acordo com os requisitos da aplicação em que os jogadores são capazes de interagir, visualizar e manipular os objetos, por meio dos controles dos dispositivos.

Existem várias definições para jogos *multiplayer* devido sua natureza multidisciplinar. É possível definir jogos *multiplayer* como:

- Jogos em tempo real, altamente interativos que suportam múltiplos jogadores simultâneos (Battaiola et al., 2001) (Hagsand, 1996);
- Jogos que envolvam a execução de programas com vários dispositivos de diferentes tipos, conectados por uma rede de comunicação. Os jogos *multiplayer* são caracterizados como um ambiente interativo multiusuário, em que usuários dispersos geograficamente têm como objetivo a integração dos recursos computacionais em tempo real (Benford et al., 1994) (Kubo, 2000).

3.2 Requisitos

É possível relacionar os requisitos dos jogos *multiplayer* de acordo com os requisitos dos Ambientes Virtuais Distribuídos (Van Dam, 1993) (Deriggi, 1998):

- Taxas rápidas de atualização do jogo;
- Latência máxima (atraso);
- Tratamento de múltiplos dispositivos de Entrada/Saída;
- Simulação de um grande número de objetos com comportamento simples, médio e complexo;
- Representação realista do jogo;
- Distribuição dos dados a longa distância;
- Suporte aos recursos computacionais e de redes heterogêneas;
- Escalabilidade.

Segundo Sementille (Sementille, 1999), os Ambientes Virtuais Distribuídos podem ser evidenciados por cinco características comuns, sendo que tais características podem ser aplicadas aos jogos *multiplayer*. São elas:

- Sensação de Compartilhamento de Espaço: todos os jogadores de um jogo *multiplayer* têm a ilusão de estarem localizados no mesmo lugar. Este espaço compartilhado deve apresentar as mesmas características a todos os jogadores;
- Sensação de Presença: quando entra em um local compartilhado, cada participante torna-se um “jogador virtual”, denominado de *avatar*. O *avatar* é uma representação gráfica, podendo ou não possuir as seguintes características: um modelo de estrutura corporal, um modelo de movimento e um modelo físico. Uma vez dentro de um jogo *multiplayer*, cada jogador pode visualizar outros *avatares* localizados no mesmo espaço. Similarmente, quando um jogador deixa o jogo *multiplayer*, os outros jogadores vêem seu *avatar* partir. Nem todos os *avatares* precisam ser controlados por jogadores. Alguns *avatares* podem ser entidades sintéticas controladas por modelos de simulação dirigidos por eventos;
- Sensação de Tempo Compartilhado (sincronização): os jogadores podem ser capazes de ver os efeitos dos comportamentos uns dos outros em tempo real;
- Comunicação entre os jogadores: embora a visualização seja a base para um jogo *multiplayer* efetivo, muitos jogos *multiplayer* também se empenham em permitir que algum tipo de comunicação ocorra entre os jogadores. Esta comunicação pode acontecer por meio de movimentos, animação, texto e áudio. Esta comunicação adiciona uma sensação maior de realismo a qualquer jogo simulado;
- Forma de Compartilhamento: os elementos mencionados anteriormente efetivamente oferecem um sistema de videoconferência de alta qualidade. O poder real de um jogo *multiplayer* deriva da habilidade dos jogadores em interagir uns com os outros e com o próprio jogo. Em uma simulação de combate, por exemplo, os jogadores precisam atirar e/ou colidir uns nos outros. Além disso, ser capazes de escolher, mover, e manipular objetos que existem no jogo e até entregá-los a outros jogadores. Estas interações precisam ser modeladas o mais próximo da realidade.

A importância dos jogos *multiplayer* reside na integração dos gráficos com a habilidade de possuir diversos jogadores compartilhando informação e manipulando objetos no jogo. A presença de diversos jogadores independentes diferencia os jogos *multiplayer* (multiusuário) dos sistemas de jogos tradicionais (monousuário). Os jogos *multiplayer* são mais apropriados para jogos que demandam a criação de telepresença, ou seja, a ilusão que todos jogadores estão presentes no mesmo jogo, mesmo que situados fisicamente em lugares remotos.

3.3 Componentes dos Jogos Multiplayer

Os jogos *multiplayer* consistem de quatro componentes básicos: *displays* gráficos, dispositivos de comunicação e de controle, sistema de processamento e rede de comunicação. Estes componentes trabalham juntos para fornecer a sensação de envolvimento em diferentes localidades, conforme relacionados a seguir:

- *Displays* Gráficos: oferecem ao jogador a visão da estrutura gráfica do jogo. Exemplos dos *displays* gráficos são: monitores, visores de dispositivos móveis, óculos (HMD - *Head Mounted Display*) e Cavernas Virtuais (CAVE - *Cave Automatic Virtual Environment*);
- Dispositivos de Comunicação e Controle: os jogadores necessitam manipular os objetos e se comunicar com outros jogadores no jogo. Estas tarefas são executadas por meio do uso de diversos dispositivos de entrada. Os dispositivos de entrada mais comuns são controles, teclado, mouse e atualmente também os meios de entrada dos dispositivos móveis. Porém, apesar de comuns, controle, mouse e teclado nem sempre são muito eficientes. Para tarefas de manipulação mais precisas, uma luva (*dataglove*) pode ser usada. A comunicação textual, por sua vez, distrai o jogador do completo envolvimento que alguns jogos *multiplayer* buscam atingir. Nos jogos *multiplayer* mais sofisticados, os jogadores comunicam-se verbalmente através de interação multimodal. No futuro pode se tornar comum o uso de câmeras para reconhecimento de gestos ou para efeitos de realidade aumentada, sensores de posição e interfaces tangíveis (Siscoutto, 2003);

- Sistema de Processamento: os jogos *multiplayer* demandam uma considerável capacidade de processamento. A unidade de processamento é responsável por receber os eventos dos dispositivos de entrada do jogador e computar o efeito destas entradas sobre a posição do jogador e de outros objetos no jogo. O processador deve determinar como e quando notificar os outros jogadores das mudanças iniciadas pelo jogador local, e vice-versa. Finalmente, o processador deve renderizar o *display* gráfico e manter a atualização do ponto de vista do jogador no jogo. Portanto, um dos desafios para os projetistas de jogos *multiplayer* é alocar o tempo do processador para as tarefas requeridas para manter a ilusão de presença do jogador de forma transparente;
- Rede de Comunicação: os jogadores de um jogo *multiplayer* necessitam da rede de comunicação para trocarem informações, como por exemplo, a mudança de localização dos jogadores ou a movimentação de algum objeto. A rede também é responsável pela sincronização do estado compartilhado no jogo *multiplayer*, tais como tempo e visibilidade. Deve também suportar a comunicação por texto e áudio entre os jogadores.

3.4 Características Técnicas

A comunicação em rede dos jogos *multiplayer* possui diversas características que devem ser consideradas para entender os requisitos relatados anteriormente. As principais características são: largura de banda, protocolos de comunicação, modelos de comunicação, estruturas básicas de comunicação, formas de distribuição e armazenamento de dados, formação de grupos, técnicas de redução de tráfego de mensagens e atraso.

3.4.1 Largura de Banda

Largura de banda consiste na capacidade total de transferência de dados na rede (Macedonia & Zyda, 1997). Nos jogos *multiplayer*, a quantidade de dados transmitidos pela rede define o tamanho e a riqueza da aplicação. A vazão (*throughput* – dados reais transferidos) em uma rede *Local Area Network* (LAN) do padrão *Ethernet* é cerca de 100 Mbps, tornando limitado o número de usuários. Na rede *Wide Area Network* (WAN) a vazão é cerca de 1.5 Mbps, limitando ainda mais o número de usuários.

Atualmente, existem novas tecnologias *Ethernet* que são capazes de suportar até 1 Gbps e existem também as tecnologias que utilizam os cabos de fibra ótica que são capazes de transferir informações de 10 Gbps ou mais.

Nas redes de comunicação sem fio, ainda não se atingiu as características de velocidade e largura de banda de uma rede com fio. As redes sem fio trabalham a uma velocidade de 1 Mbps, no padrão IEEE 802.11 e no *Bluetooth*, até o máximo de 54 Mbps, no padrão IEEE 802.11g (Gast, 2002).

O *Bluetooth* é uma especificação aberta para uma tecnologia que permite comunicações de dados e voz em uma área de curto alcance, oferecendo um padrão que pode ser seguido por dispositivos em qualquer lugar do mundo. O *Bluetooth* opera na faixa não licenciada de 2.4 GHz do espectro de frequência. Por não ser licenciada, outras tecnologias também podem operar nesta faixa, como IEEE 802.11 para redes locais. Sendo assim, a comunicação sem fio através de *Bluetooth* desenvolveu-se para permitir tráfego de dados no canal entre dispositivos de baixo consumo de energia, minimizando os efeitos de interferência (Miller & Bisdikuan, 2002).

O grupo IEEE 802.11 se preocupa em definir protocolos e interfaces para LANs sem fio (WLAN). A intenção em se estruturar uma WLAN é se ter a capacidade de utilização dos mesmos serviços oferecidos por uma LAN, com a vantagem de não serem necessários fios para o tráfego de informações. Para tanto decidiu-se implementar uma compatibilidade entre *Ethernet* e IEEE 802.11. Portanto, o envio de um pacote IP (*Internet Protocol*) tanto em uma rede sem fio ou em uma *Ethernet* é feito da mesma forma (Gast, 2002). O primeiro padrão IEEE 802.11 funcionava a taxas de 1 ou 2 Mbps, e logo surgiram reclamações quanto a taxas tão baixas (Tanenbaum, 2003). Rapidamente, desenvolveram-se dois novos padrões, o IEEE 802.11a que adotou uma faixa de frequência mais larga e funciona a uma velocidade de 54 Mbps; e o IEEE 802.11b que utiliza a mesma faixa de frequência, mas cuja camada física adota outra técnica de modulação para atingir até 11 Mbps. Além disso, foi desenvolvido o padrão IEEE 802.11g que utiliza a técnica de modulação do IEEE 802.11a e a frequência do IEEE 802.11b.

Podem-se destacar também as redes de comunicação sem fio GPRS (*General Packet*

Radio Service) e 1xRRT (*1 x Radio Transmission Technology*).

O GPRS é um serviço de transmissão de informações em uma rede de comunicação sem fio que não utiliza os recursos de áudio. O GPRS suplementa as tecnologias atuais de CSD (*Circuit Switched Data*) e SMS (*Short Message Service*). A taxa de transferência varia em torno de até 171.2 Kbps. O GPRS é um serviço de dados que permite o envio e o recebimento de informações através de pacotes de uma rede sem fio que suporte IP e X.25 como, por exemplo, a rede de celulares móveis que utiliza a comunicação via GSM (*Global System for Mobile*). A tecnologia GPRS é freqüentemente chamada de tecnologia 2.5G porque representa a primeira etapa que uma operadora GSM empreende quando inicia a transição à terceira geração (3G) dos sistemas de comunicações móveis.

O 1xRRT é uma extensão do padrão CDMA (*Code Division Multiple Access*) que transmitem dados e áudio a 144 Kbps com conexão permanente a banda larga, tornando possível a transferência de mídias como filme, música e fotografias e também fazer *download* de aplicativos.

Os dados transmitidos nas aplicações de sistema multimídia são aqueles que consomem uma enorme largura de banda para suportar imagens, vídeo, áudio e troca de dados em tempo real entre múltiplos usuários. Num jogo *multiplayer*, a demanda de tráfego na rede tende a ser menor, pelo fato das imagens serem geradas localmente ou no servidor - a rede transmite informações de posicionamento de objetos. No entanto, como o número de usuários de jogos *multiplayer* tende a ser grande (milhares), uma largura de banda alta continua sendo necessária.

Para atender tais necessidades, as tecnologias atuais buscam aumentar a largura de banda com o desenvolvimento de novos protocolos, dispositivos de alta velocidade, servidores e máquinas com alto desempenho computacional e capacidade de processamento gráfico, comunicação direta entre dispositivos (sem passar por roteadores, *gateways*, etc.), técnicas de *links* particulares, como *Dead-Reckoning*, *Interest Management* e *Message Compression and Agregation*, entre outras.

3.4.2 Protocolos de Comunicação

Outro fator importante na comunicação de rede é o protocolo de comunicação, que

define o modo de comunicação entre processos, implicando na confiabilidade e velocidade de comunicação da rede. A confiabilidade da rede é uma medida que reflete a quantidade de dados que são perdidos pela rede durante o trajeto do *nó* origem para o *nó* destino. O dado perdido pode ser classificado como: dado interrompido ou dado corrompido. O dado interrompido é quando o dado não chega por completo ao *nó* destino. O dado corrompido é quando o conteúdo dos pacotes de dados é alterado durante a transmissão e o que chega ao *nó* destino é basicamente inútil. Muitos pesquisadores consideram a interrupção e a corrupção de dado essencialmente equivalente, isto porque, em ambos os casos, o dado não atinge efetivamente a aplicação destino. A confiabilidade da rede frequentemente força um compromisso entre a largura de banda e a latência (Singhal & Zyda, 1999) (Sementille, 1999).

O protocolo de rede chamado *Internet Protocol* (IP) tornou-se o elemento principal da *Internet*. Tal protocolo pertence à camada de rede do modelo OSI (*Open System Interconnection*) e seu objetivo é fornecer o melhor caminho de transportar datagramas da origem para o seu destino, independente das máquinas estarem na mesma rede ou em outras redes intermediárias (Tanenbaum, 2003). Nas aplicações de jogos, são utilizados mecanismos do protocolo IP para a distribuição dos dados através da comunicação *multicasting*. Tal comunicação está relacionada à classe de endereço D desse protocolo, que identificam um grupo de *nós*.

Os protocolos TCP (*Transmission Control Protocol*) e o UDP (*User Datagram Protocol*) (Tanenbaum, 2003) (Locke, 1995) são outros dois protocolos utilizados acima da camada de rede IP e pertencem à camada de transporte. O TCP garante que a mensagem chegue ao seu destino enquanto o UDP não tem essa garantia. A escolha da utilização do TCP ou UDP depende do objetivo da aplicação. Para aplicações que têm a necessidade de entrega de informações confiáveis, como, por exemplo, na teleoperação de um robô em uma sala de cirurgia, o protocolo TCP é o mais indicado. Entretanto, o protocolo UDP é suficiente para aquelas aplicações que não são tão rígidas na entrega de informações, tais como as transmissões de áudio em tempo real.

Uma das técnicas utilizadas para assegurar a confiabilidade dos jogos *multiplayer* é a utilização do tratamento de reconhecimento ACKs e NACKs (Tanenbaum, 1995) (Macedonia et al., 1995). ACKs são também conhecidos por protocolos “orientados ao

transmissor” e os NACKs são conhecidos por “orientados ao receptor”. No método ACK, o transmissor emite uma mensagem pela rede e espera que o destinatário envie um ACK. Quando o transmissor não recebe a mensagem do destinatário e o processo de *time-out* é executado, o transmissor emite novamente a mensagem para o receptor, até que o mesmo receba a mensagem do receptor. O NACK é um método em que o receptor envia uma mensagem para o transmissor, somente quando ele não recebe a mensagem. Consequentemente, o ACK possui uma quantidade maior de mensagens trafegando na rede, enquanto o NACK possui uma quantidade menor de mensagens. Portanto, para o desenvolvimento dos jogos *multiplayer*, os NACKs constituem-se em um método mais otimizado. Como por exemplo, o RMP (*Reliable Multicast Protocol*) (Whetten & Kaplan, 1994), o qual é baseado em um protocolo *token-ring* situado no topo do IP *multicast*. Este método usa método sequencial e NACKs. O problema com este método é o potencial de implosão de NACKs, na qual um grupo de receptores enviam simultaneamente NACKs, causando congestionamento e, conseqüentemente, provocando perda de pacotes com a introdução de mais NACKs. Por isso, sistemas confiáveis normalmente não operam muito bem em tempo real (Sementille, 1999).

3.4.3 Modelos de Comunicação

Os modelos de comunicação em um jogo *multiplayer* são basicamente divididos em: Modelo Cliente/Servidor, Peer to Peer e Híbrido (Kirner & Pinho, 1996) (Borges, 1996) (Kubo, 2000) (Meiguins, 2003).

Modelo Cliente/Servidor: O paradigma cliente/servidor em um jogo *multiplayer* implica na comunicação direta somente do cliente com o servidor ou vice-versa, não havendo a troca direta de mensagens entre os clientes. O servidor, por sua vez, é responsável pela transmissão das informações necessárias a outros clientes.

As vantagens do modelo cliente/servidor são:

- As informações deverão sempre passar pelo servidor. O servidor pode executar um processamento adicional antes das informações recebidas serem retransmitidas;
- O servidor pode ser responsável pelo filtro ou seleção das mensagens enviadas

pelos clientes, gerando economia no tráfego de mensagem;

- Gerenciamento centralizado no servidor, o que acarreta simplicidade no controle das atividades entre os clientes;
- Os clientes apenas comunicam com o servidor, que resulta em apenas uma conexão por máquina, com protocolos de comunicação mais simples;
- O servidor armazena um diretório com os endereços dos clientes e o cliente tem apenas o endereço do servidor;
- Diminui o processamento do cliente, desde que o processamento seja realizado previamente no servidor;
- Implementação do cliente passa a ser mais simplificada;
- Controle das versões do jogo é simplificado, já que a cópia principal é mantida no servidor, podendo ser acessado por todos os clientes.

As desvantagens do modelo cliente/servidor são:

- Podem ocorrer atrasos na transmissão das informações, uma vez que toda informação deve ser enviada primeiramente ao servidor e depois ser replicada aos clientes;
- Escalabilidade do sistema depende da capacidade do servidor em processar e retransmitir as informações;
- Se o servidor falhar, todo o sistema fica sem operação;
- Sobrecarga ou gargalo sobre o servidor.

Modelo Peer to Peer: O paradigma Peer to Peer implica na comunicação direta entre os *nós*, sem intermédio de um ou mais servidores.

As vantagens do modelo Peer to Peer são:

- A falha em um cliente afeta apenas a participação do jogador ligado àquele

cliente, sem afetar a operação do jogo como um todo;

- Diminui o problema de gargalo distribuindo o gerenciamento das atividades entre os clientes;
- Pode suportar um número crescente de clientes (escalabilidade).

As desvantagens do modelo Peer to Peer são:

- A divisão do gerenciamento das atividades entre os clientes, acarreta uma cooperação através de protocolos de comunicação, o que leva a uma maior complexidade de *software*, custos altos de desenvolvimento e menor desempenho em cada cliente;
- Cada cliente deve conhecer os endereços dos outros clientes para que possam se comunicar diretamente;
- Maior complexidade no gerenciamento de entrada/saída de novos clientes;
- O controle de versões de jogo é mais difícil de ser mantido nesta abordagem distribuída.

Modelo Híbrido: O paradigma Híbrido consiste na combinação do modelo Cliente/Servidor e Peer to Peer.

As vantagens do modelo híbrido são:

- Como os *nós* intermediários tratam da movimentação dos objetos, não há a necessidade de comunicação entre servidores de ambiente;
- As ligações entre os *nós* intermediários e os servidores de ambiente podem ser feitas através de *links* de alta velocidade;
- Suporta bem a dispersão geográfica, independente da localização do servidor de ambiente, o cliente pode ser conectado a um *nó* intermediário próximo de sua localização;

- A entrada de novos clientes, controlada no respectivo *nó* intermediário, não precisa ser anunciada aos atuais clientes deste *nó*;
- Os clientes se conectam a um *nó* intermediário, precisando manter apenas uma conexão, o que implica em protocolos mais simples de comunicação e gerenciamento;
- Mantém os benefícios da centralização da informação em um servidor de ambiente centralizado, aumentando a segurança e facilitando a manutenção de consistência do estado do jogo, além de facilitar também a atualização de versões de *software*;
- Assim como no modelo cliente/servidor, máquinas mais poderosas, e portanto mais caras, são mantidas num menor número possível, como servidores de ambiente.

As desvantagens do modelo híbrido são:

- Latência maior que a do modelo cliente/servidor com a existência de dois *nós* entre o cliente e o servidor e entre servidores de ambiente;
- Maior complexidade nos protocolos entre os servidores de ambiente e os *nós* intermediários.

3.4.4 Estruturas Básicas de Comunicação

As estruturas básicas de comunicação são: *unicast*, *broadcast* e *multicast* (Macedonia & Zyda, 1997) (Rodello, 2003).

Na estrutura *unicast*, a comunicação dos dados é conhecida como “um-para-um”, sendo realizada somente entre dois *nós*. *Broadcast* (difusão) é o mecanismo de comunicação dos dados conhecido como “um-para-vários”, em que um *nó* envia um dado para todos os outros *nós* da rede. *Multicast* é o mecanismo de comunicação dos dados conhecido como “um-para-vários” ou “vários-para-vários” em uma única operação.

A importância do *multicast* é decorrente da restrição do número de jogadores em um mesmo escopo de recepção de mensagens, fazendo um mapeamento entre os grupos de

jogadores para usar de forma eficiente a largura de banda, reduzir a latência e aumentar a vazão sobre as redes (Hagsand, 1996).

3.4.5 Formas de Distribuição e Armazenamento dos Dados

O armazenamento dos dados é um fator importante em um jogo *multiplayer*, porque implica na transferência e consistência das informações dos estados de um jogo. Decidir onde as informações serão armazenadas afeta principalmente as escalas de dados do jogo, os requisitos de comunicação e a confiabilidade no sistema em geral (Macedonia & Zyda, 1997) (Rodello, 2003). Pelos mecanismos atuais, a escolha da forma de distribuição e armazenamento dos dados ainda está restrita ao objetivo da aplicação.

Segundo Macedonia e Zyda (Macedonia & Zyda, 1997), o armazenamento dos dados em um jogo pode ser restrito a: banco de dados replicados de mundos homogêneos, banco de dados centralizado e compartilhado, banco de dados multiusuário e compartilhado com alterações ponto a ponto, e banco de dados multiusuário e compartilhado Cliente/Servidor.

As principais características de cada modelo são:

Banco de Dados Centralizado e Compartilhado

- Os dados estão centralizados em uma única máquina, que atua como servidor;
- Há uma melhor consistência do jogo *multiplayer*;
- Os usuários mantêm somente os dados pertinentes a cena em memória;
- Apenas um usuário pode modificar o banco de dados em um determinado momento;
- Limitação no número de usuários;
- Dificuldade em manter a consistência do banco de dados dos objetos em tempo real;
- O servidor torna-se o gargalo da aplicação.

Banco de Dados Replicados de Mundos Homogêneos

- Cada *nó* conectado à aplicação possui uma réplica completa dos dados do ambiente (o terreno, modelo geométrico, texturas e comportamento de todos representantes no jogo);
- Utilização de mecanismos de garantia da consistência como “*heartbeat*” (o *nó* envia periodicamente mensagens informando o seu estado);
- As vantagens desta abordagem é que as mensagens são relativamente pequenas;
- As desvantagens estão diretamente relacionadas com a inflexibilidade e o aumento das informações do jogo. Deste modo, todas as informações do banco de dados devem estar em todos os *nós*;
- Sistema inconsistente à medida que o número de participantes aumenta no mundo, por causa do prejuízo do estado e das mensagens de eventos.

Banco de Dados Multiusuário e Compartilhado com Alterações Ponto a Ponto

- Cada *nó* possui apenas uma parte do jogo em sua base de dados;
- Existe um maior número de mensagens, mas usa-se a técnica de *multicast* para diminuir o tráfego;
- Apropriado para construir jogos multiusuário de larga escala;
- A desvantagem desta abordagem é a dificuldade de escalabilidade por causa do custo da comunicação associada com a confiabilidade e a consistência de dados.

Banco de Dados Multiusuário e Compartilhado Cliente/Servidor

- O modelo baseia-se no modelo Cliente/Servidor;
- A base de dados é particionada em seus clientes;

- Um agente chamado *broker* é responsável pelo conhecimento do servidor para manter o mundo atualizado;
- Um cliente pode requisitar um dado de forma transparente.

3.4.6 Formação de Grupos

Os grupos dos jogos *multiplayer* podem ser formados a partir da área de interesse, que é basicamente formada por três classes: funcional, temporal e espacial (Macedonia et al., 1995). A classe funcional consiste na comunicação dos *nós* que possuem a mesma função dentro dos jogos *multiplayer*. A classe temporal consiste na comunicação dos *nós*, onde não há a necessidade de atualizações em tempo real de todos os estados do jogo. A classe espacial consiste na comunicação de *nós* que possuem a representação de uma área física de interesse limitada do jogo.

O jogo *multiplayer* é particionado em vários conjuntos de subambientes, em que cada subambiente é um grupo *multicast*, tendo um gerenciador da área de interesse. Os grupos *multicast* permitem a comunicação dos *nós* sobre a rede com apenas uma ação, diminuindo conseqüentemente o tráfego na rede. Cada *nó* pode fazer parte de mais de um endereço *multicast* em um mesmo jogo.

Os grupos *multicast* podem ser formados por meio do particionamento estático de um jogo e o endereço *multicast* é alocado para um espaço físico pré-definido, principalmente para espaços confinados, como por exemplo, uma quadra de esporte. Os grupos *multicast* também podem ser alocados por espaços físicos dinâmicos, principalmente quando tratam de espaços abertos, como por exemplo, um campo de batalha.

3.4.7 Técnicas de Redução de Tráfego de Comunicação

Existem várias técnicas para redução de tráfego de comunicação entre os *hosts*, com o objetivo de reduzir a largura de banda utilizada. Dentre elas é possível destacar as técnicas de *Dead-Reckoning*, *Interest Management* e *Message Compression and Agregation* (Azevedo, 2005).

A técnica de *Dead-Reckoning* é um algoritmo que faz a previsão de uma trajetória,

baseada nas posições anteriores, velocidade e direção. A técnica de *Dead-Reckoning* proporciona a redução do tráfego na rede e conseqüentemente a diminuição do atraso, sendo aplicado em cada *nó* que contém as informações do jogo. A técnica de *Dead-Reckoning* surgiu com a navegação marítima e se refere à prática de extrapolar a posição e direção atual do objeto em questão, baseado num conjunto de cálculos, a fim de deduzir posições futuras (Gosseiler et al., 1994) (Fishwick, 1995) (Stytz, 1996).

A técnica de *Interest Management* inclui abordagens que permitem aos *hosts* expressarem interesse em apenas um subconjunto de informações que sejam relevantes, o que reduz o número de mensagens transmitidas no jogo.

A técnica de *Message Compression and Agregation* possibilita a compressão de mensagens, economizando-se recursos computacionais, e a agregação de mensagens, que reduz a freqüência de transmissão a partir da união das informações de diversas mensagens no jogo.

3.4.8 Atraso

Atraso (*Lag*) é o tempo decorrido entre a atuação do jogador no sistema e a visualização do resultado no jogo, ou seja, é o tempo entre o jogador realizar uma interação no dispositivo de entrada (por exemplo, luva) do sistema e o resultado dessa ação a ser observado no dispositivo de saída (por exemplo, capacete) da aplicação (Wloka, 1995).

As maiores frustrações dos jogadores são, em geral, causadas principalmente pelos atrasos nas respostas de interação do jogador com o sistema. Em um jogo, esse problema fica ainda mais acentuado, principalmente em jogos imersivos.

Para um jogador ter a sensação de que ele realmente está envolvido num jogo, é necessário minimizar ao máximo o valor do atraso para que esse ambiente aproxime-se do ambiente real. Atrasos a partir de valores em torno de 100 ms começam a degradar o desempenho humano e, se o atraso exceder 300 ms, os jogadores desassociam os seus movimentos do resultado sintetizado no dispositivo de saída da aplicação, acabando com o efeito imersivo. Em vista desses dados, para obter-se o efeito de envolvimento é necessário aumentar a interatividade das aplicações de jogos, tornando o atraso inferior a 300 ms (Latham, 1994) (Pantel & Wolf, 2002).

Por meio do monitoramento e teste do sistema, pode-se estabelecer a complexidade do ambiente que poderá executar em uma máquina, sem perder o efeito de tempo real. Assim, pode-se definir a plataforma mínima para execução de um determinado sistema.

Em geral, o atraso é um dos principais fatores limitantes para aplicações de jogos e, por esse motivo, são discutidas as principais fontes de atraso, segundo (Wloka, 1995). As formas existentes de atrasos em um jogo são: atraso de dispositivos de entrada do jogador, atraso do processamento, atraso de sintetização, atraso de sincronização e atraso induzido pela taxa de quadros por segundo.

O atraso de dispositivos de entrada do jogador consiste no tempo de comunicação de dados desses dispositivos, envolvendo o atraso decorrente da geração de sinal e da comunicação. Dependendo do tipo de dispositivo e do modo de operação, o atraso varia de 10 a 120 ms, com vazão entre 30 e 50 quadros por segundo (Wloka, 1995).

O atraso de processamento é altamente dependente da aplicação, dessa forma, altamente variável. Quanto mais complexo for o jogo, maior será o atraso de processamento. O atraso, nesse caso, varia entre 1 e 500 ms (é claro que algumas aplicações de jogos podem ter um atraso superior a 500 ms, porém as aplicações não são caracterizadas imersivas) (Wloka, 1995).

O atraso de sintetização é caracterizado pelo tempo decorrido desde o envio de dados para o *hardware* de sintetização até que esse mesmo dado seja mostrado no monitor. A taxa de atualização do monitor (tipicamente entre 60 - 72 Hz) causa um atraso de sintetização que pode variar de 0 a 17 ms, tipicamente. O atraso de sintetização é altamente dependente da complexidade da cena a ser mostrada, e assim esse atraso é variável durante a execução de uma aplicação.

O atraso de sincronização envolve as aplicações paralelas de jogos que processam vários estágios de entrada do jogador: processamento de dispositivos de entrada do usuário, processamento dependente da aplicação e sintetização. Atraso de sincronização é definido como o tempo total em que uma amostra (dados) espera entre os estágios de entrada sem a existência de processamento. Por exemplo, um dispositivo de entrada envia uma nova amostra para a porta serial logo depois da aplicação ler esta porta.

Assim, a aplicação ficará ocupada, processando as informações antigas. Essa aplicação levará certo tempo até ler novamente a porta serial. Esse tempo, em que os novos dados de entrada ficam sem serem processados, é chamado de atraso de sincronização. No melhor caso, o atraso de sincronização é zero: quando cada estágio escreve suas saídas exatamente antes da leitura do próximo estágio. No pior caso, a saída em cada estágio ocorre justamente depois da leitura do próximo estágio.

O atraso induzido pela taxa de quadros ocorre quando os dados da tela não são atualizados e os dados de processamento do jogo encontra-se em processo de atualização.

A soma de todos esses atrasos é conhecida como atraso de fim-a-fim, devendo manter a valores em torno de 100 ms a 300 ms.

3.5 Conclusão do Capítulo

Neste capítulo, foram apresentados os conceitos relacionados a jogos *multiplayer* e os componentes que os compõem. Tais componentes são oriundos dos conceitos de ambientes virtuais distribuídos.

Foram também apresentados os requisitos e características que os jogos *multiplayer* devem implementar para uma melhor qualidade no seu funcionamento. Os requisitos e as características estão relacionados às questões que envolvem o meio de comunicação, distribuição e armazenamento de dados, formação de grupos de jogadores, meios de redução do tráfego de rede e atrasos que podem ocorrer nestas aplicações de jogos.

O objetivo deste capítulo foi mostrar os aspectos relevantes para o desenvolvimento de jogos *multiplayer*.

Capítulo 4 - Framework

Este capítulo apresenta um resumo do estado da arte do conceito de *framework* na área de engenharia de *software*, sobre o qual foi fundamentado o *framework* proposto nesta Tese e apresentado no capítulo 5.

Nas seções deste capítulo são apresentados os conceitos fundamentais necessários para o desenvolvimento de um *framework*, definições, classificação, vantagens e desvantagens de sua utilização e formas de reuso, a importância da documentação e as metodologias existentes. Por fim, são relacionados os requisitos e a arquitetura para *framework* de jogos e são discutidos alguns *frameworks* existentes.

4.1 Introdução

Um *framework* é uma estrutura genérica que pode ser adaptada a aplicações específicas, servindo como um modelo para o desenvolvimento de aplicações. Em outras palavras, seguindo as práticas mais comuns em engenharia de *software*, um *framework* pode ser descrito como uma estrutura abstrata orientada a objetos ou procedural, adaptável às diversas necessidades de aplicações. As aplicações construídas a partir de um mesmo *framework* apresentam a mesma estrutura, diferenciando-se em seu comportamento, que é dependente do objetivo da aplicação. Isto torna as aplicações desenvolvidas a partir do *framework*, fáceis de serem mantidas e com uma maior consistência para os desenvolvedores de sistemas.

Além disso, um *framework* pode fornecer um alto grau de reutilização (Firesmith, 1993), pois além de fornecer reutilização de código, que é oferecido por classes e objetos, também oferece reutilização de projeto, liberando o desenvolvedor de se preocupar com aspectos comuns daquele domínio de aplicação. A reutilização de projeto permite uma reutilização em larga escala, que é uma condição necessária para o aumento de produtividade no desenvolvimento de *software*. Além disso, a reutilização de projeto é mais importante que a reutilização de código, porque o projeto é mais difícil de criar e recriar (Ralph & Foote, 1988).

Como resultado da crescente utilização da engenharia de *software* no desenvolvimento

de jogos para PC's ou dispositivos móveis, a indústria começou a isolar os recursos utilizados na maioria dos jogos em *frameworks* de desenvolvimento (também chamados de “*engines*” - *game engines*). A idéia principal é permitir que os recursos comuns a quase todos os jogos sejam reutilizados a cada novo jogo desenvolvido.

4.2 Definições

É possível definir *framework* de diversas maneiras, tais como:

Framework é o projeto de um conjunto de objetos que colaboram para por em prática um conjunto de responsabilidades. O *framework* é um modo de reutilizar projetos de alto nível (Orfaly, 1995).

Framework é um conjunto de classes relacionadas que podem ser estendidas para implementar uma aplicação do mesmo domínio (Pires, 1999).

Framework é um conjunto de classes relacionadas, as quais permitem a reutilização em um projeto com classes específicas do *software* (Gamma et. al., 1995).

Framework é um projeto reutilizável, de todo ou parte de um sistema, representado por um conjunto de classes abstratas e por um modelo de como interagem as instâncias (Johnson, 1996).

4.3 Classificação

Um *framework* pode ser classificado de acordo com a visibilidade da sua estrutura interna (*black box* e *white box*) (Johnson e Foote, 1988):

- O tipo *black box* disponibiliza somente sua interface. O modo como é implementada a funcionalidade interna não é visível ao usuário. Esse encapsulamento facilita seu uso, pois o usuário não necessita conhecer o funcionamento interno, mas pode reduzir a sua flexibilidade e extensibilidade por não permitir ao mesmo adaptar-se à novas necessidades;
- O tipo *white box* tem a sua estrutura interna visível ao usuário. Essa visualização

de código permite estudar sua funcionalidade e fazer alterações, implementadas externamente à classe do *framework*. O uso de um *framework white box* requer conhecimento da funcionalidade interna.

Rogers (Rogers, 1997) classifica os *frameworks* em duas categorias de acordo com sua aplicabilidade (*frameworks* horizontais e *frameworks* verticais):

- *Framework* horizontal - É mais abrangente, pode ser usado por vários tipos de aplicações. *ToolKits* gráficas é um exemplo de *framework* horizontal, pois pode ser utilizado para construir interfaces de usuários para um grande número de aplicações (Krasner & Pope, 1988);
- *Framework* vertical - é mais específico para um domínio particular. Ele não é usualmente utilizado em domínios diferentes. Como exemplo de *framework* vertical, destaca-se o *framework Hotdraw* (Pressman, 1995), baseado em herança, para a criação de editores gráficos para figuras bidimensionais. *Hotdraw* pode ser utilizado para construir editores de gráficos especializados, tais como esquemas de *hardware* ou diagramas de projeto de *software*.

Firesmith (Firesmith, 1993) sugere uma classificação que se relaciona ao domínio da aplicação a ser desenvolvida utilizando o *framework*:

- *Frameworks* para Interfaces com Usuário - Fornecem classes de objetos para a construção de interfaces gráficas, permitindo que as aplicações desenvolvidas a partir dele possuam o mesmo *look and feel*;
- *Frameworks* para Ambientes - Fornecem independência a partir da portabilidade entre múltiplos sistemas operacionais e plataformas de *hardware*.

Os *frameworks* também podem ser classificados quanto à forma de desenvolvimento, neste caso tem-se:

- *Frameworks* Procedurais - São os *frameworks* que não são baseados em conceitos provenientes do paradigma orientado a objetos. A necessidade deste tipo de *framework* provém da quantidade e importância do código já produzido

desta forma (Orfaly, 1995);

- *Frameworks* Orientados a Objetos - São construídos utilizando-se o paradigma orientado a objetos. As próprias definições apresentadas neste trabalho referem-se diretamente a conceitos orientados a objetos, evidenciando-se assim uma tendência na construção e adoção deste tipo de *framework*.

Os *frameworks* são classificados em três grupos, segundo Fayad e Schmidt (Fayad & Schmidt, 1997): *frameworks* de infraestrutura do sistema, *frameworks* de integração de *middleware* e *frameworks* de aplicação empresarial.

- Os *frameworks* de infraestrutura do sistema - Simplificam o desenvolvimento da infraestrutura de sistemas portáteis e eficientes, como por exemplo, os sistemas operacionais, sistemas de comunicação, interfaces com o usuário e ferramentas de processamento de linguagem. Em geral são usados internamente em uma organização de *software* e não são vendidos a clientes diretamente;
- Os *frameworks* de integração de “*middleware*” - São usados, em geral, para integrar aplicações e componentes distribuídos. Eles são projetados para melhorar a habilidade de desenvolvedores em modularizar, reutilizar e estender sua infraestrutura de *software*. Exemplos dessa classe de *framework* são o ORB (*Object Request Broker*), “*middleware*” orientado a mensagens e bases de dados transacionais;
- Os *frameworks* de aplicação empresarial - Estão voltados a domínios de aplicação mais amplos e são fundamentais para atividades de negócios das empresas, como, por exemplo, sistemas de telecomunicações, aviação, manufatura e engenharia financeira.

4.4 Vantagens e Desvantagens da Utilização de Frameworks

A utilização de *frameworks* fornece uma série de vantagens à atividade de desenvolvimento de *software*:

- Desenvolvimento de novas aplicações torna-se mais rápido e mais barato, uma

vez que utiliza componentes pré-fabricados e pré-testados. O desenvolvedor não precisa descobrir novas classes e projetar interfaces. Basicamente, ele somente necessita reescrever o comportamento de métodos específicos de determinadas classes. Além disso, a estrutura do programa já está especificada e o fluxo de execução definido;

- Os *frameworks* permitem a reutilização de código e projeto. Normalmente, o reuso de código é obtido por meio de polimorfismo e/ou herança. O reuso de projeto é obtido a partir do próprio *framework*, uma vez que ele foi desenvolvido como uma solução genérica para uma determinada categoria de problemas;
- A manutenção é reduzida. Os *frameworks* fornecem a maior parte do código necessário às aplicações, tornando menores os custos de manutenção. Devido à herança, quando um erro em um *framework* é corrigido, ou uma característica adicionada, os benefícios são imediatamente estendidos às novas classes (Orfaly, 1995);
- Possibilidade de desenvolvimento de aplicações mais poderosas e complexas. A criação de *frameworks* baseados em outros *frameworks* permite o desenvolvimento de aplicações cada vez mais complexas.

As desvantagens na utilização de *frameworks* no desenvolvimento de *software* são (Maldonado et al., 2005) (Sobral, 2002):

- Desenvolvimento da primeira aplicação tende a ser muito longo;
- Inicialmente, o código toma muito tempo para ser desenvolvido devido a um maior esforço para construção e aprendizado;
- Necessidade de documentação de manutenção e apoio.

4.5 Documentação de Frameworks

A tarefa de documentação de uma aplicação convencional é considerada importante na área de engenharia de *software*, tornando-se ainda mais importante quando a aplicação a ser desenvolvida é um *framework*. Afinal, se os outros desenvolvedores não entenderem

o *framework*, eles não o utilizarão (Sobral, 2002). Mais do que isso, a complexidade de se documentar um *framework* é maior que a de documentar uma aplicação convencional, devido à natureza genérica dos *frameworks*.

A documentação do *framework* tem propósito de comunicar informações relativas ao projeto do *framework* e outras informações relacionadas ao seu desenvolvimento e utilização. Estas informações são importantes porque tornam claro para o usuário as restrições e regras que precisam ser seguidas para que o *framework* possa ser utilizado.

4.6 Metodologias de Desenvolvimento de Frameworks

As principais metodologias propostas para o desenvolvimento de *frameworks* encontradas na literatura são: o projeto dirigido por exemplos de Johnson (Johnson, 1993), o projeto orientado a pontos adaptáveis de Pree (Pree, 1994) e metodologia Taligent encontrado no trabalho de Silva (Silva, 2000).

Segundo Johnson (Johnson, 1993), na metodologia de projeto dirigido por exemplos (*Example-Driven Design*), o desenvolvimento de um *framework* é feito a partir de um processo de aprendizado sobre um determinado domínio. Este aprendizado ocorre com o estudo de aplicações previamente desenvolvidas ou a partir do desenvolvimento de novas aplicações. A construção de um *framework*, um projeto abstrato, deve ser obtida a partir da generalização de aplicações concretas, que são as aplicações desenvolvidas anteriormente. Os aspectos comuns das aplicações se transformam em classes abstratas, enquanto que os detalhes específicos de cada aplicação se transformam em classes concretas, especializações das abstratas.

Abordagem de metodologia de projeto orientado a pontos adaptáveis (*Hot Spot Driven Design*), desenvolvida por Pree (Pree, 1994), consiste na identificação dos pontos fixos e dos pontos adaptáveis do domínio de uma aplicação. Os pontos fixos correspondem às partes comuns das aplicações correlatas, enquanto que os pontos adaptáveis são as partes que podem ser estendidas para cada aplicação específica, dando ao *framework* a capacidade de ser flexível e se moldar às diferentes aplicações.

A metodologia Taligent (Silva, 2000), proposta pela empresa de mesmo nome, difere das anteriores pelo conjunto de princípios que norteiam o desenvolvimento de

frameworks. Primeiramente, a visão de desenvolver um *framework* que cubra as características e necessidades de um domínio é substituída pela visão de produzir um conjunto de *frameworks* estruturalmente menores e mais simples, que usados em conjunto, darão origem às aplicações. Pequenos *frameworks* são mais flexíveis e podem ser reutilizados mais frequentemente. Assim, a ênfase passa a ser o desenvolvimento de *frameworks* pequenos e direcionados a aspectos específicos do domínio. Um outro aspecto é que uma das linhas mestras do desenvolvimento é tornar o uso do *framework* o mais simples possível, minimização da quantidade de código que o usuário deve produzir, por meio da disponibilidade de classes concretas, minimização do número de classes que devem ser criadas e minimização do número de métodos que devem ser sobrepostos.

O objetivo de qualquer metodologia de desenvolvimento de *frameworks* é desenvolvê-lo de tal forma que este seja (Sobral, 2002):

- Completo - Um *framework* deve fornecer as características desejadas pelos clientes e uma implementação padrão sempre que possível. Além disso, deve fornecer classes concretas que derivem de classes abstratas do *framework*, assim como implementações *default* dos métodos destas classes, para que os clientes possam entender o *framework*. Isto permite que eles possam concentrar-se nos aspectos específicos de suas aplicações;
- Flexível - Um *framework* deve permitir que suas abstrações sejam usadas em diferentes contextos;
- Extensível - Os clientes podem facilmente modificar e adicionar funcionalidades. O *framework* deve fornecer mecanismos aos clientes para que eles possam customizar o comportamento das aplicações por meio da criação de novas subclasses, ou através da combinação de instâncias de objetos;
- Compreensível - As interações entre o *framework* e o código do cliente devem ser claras e o *framework* bem documentado. Devem-se seguir padronizações durante o projeto e a implementação visando um rápido aprendizado e também fornecer aplicações de referência desenvolvidas a partir do *framework*.

4.7 Características Técnicas de Frameworks para Jogos

Jogos de computadores estão em uma categoria de sistemas bastante complexos, e o propósito de *frameworks* é a simplificação de sistemas complexos, o que torna o uso de *frameworks* para o desenvolvimento de jogos uma abordagem bastante adequada. O fundamental do projeto de um *framework* é permitir que futuros desenvolvimentos de jogos sejam realizados com menos dificuldades.

4.7.1 Requisitos

Um *framework* é a parte principal no desenvolvimento de jogos de computadores e dispositivos móveis, tendo como principal objetivo executar uma grande parte das tarefas de baixo nível necessárias na maioria dos jogos. Alguns dos principais requisitos de um *framework* para jogos de computadores são (Pessoa, 2001):

- Arquitetura modular, para que possa ser reutilizado em cada novo jogo;
- Bom nível de abstração, ao prover objetos com funcionalidades já implementadas;
- Definição de objetos básicos, como os objetos e os mapas do jogo;
- Detecção e gerenciamento de eventos gerados por teclado, mouse, *joysticks*, etc;
- Algoritmos para desenho dos objetos do jogo, podendo ser 2D, 3D, etc.;
- Funcionalidades úteis a jogos 3D, como iluminação e transformações geométricas, etc.;
- Execução dos sons do jogo em resposta a eventos ocorridos;
- Implementação de algoritmos de inteligência artificial para os objetos inteligentes dos jogos;
- Implementação de algoritmos para troca remota de dados, gerenciamento de sessões, sincronização das informações compartilhadas do jogo, etc;
- Implementação de algoritmos para modelagem física de objetos.

Em consequência dessa grande variedade de requisitos dos *frameworks*, muitos procuram limitar-se a determinados estilos de jogos.

4.7.2 Arquitetura

A arquitetura de um *framework* corresponde à definição dos módulos e de como é realizada a interação entre eles. Os módulos são responsáveis pelo gerenciamento dos eventos gerados pelo usuário até aqueles causados por interação entre os objetos do jogo.

A qualidade da arquitetura é um dos aspectos mais importantes de um *framework* (Pessoa et al., 2002). A arquitetura torna-se crítica pelo fato de que um *framework* de jogos, em geral, contém um conjunto de classes abstratas que requer uma adaptação a cada jogo a ser desenvolvido. Assim, é importante que a arquitetura do *framework* seja simples o suficiente para que tal tarefa seja possível. Infelizmente, não há consenso sobre arquitetura de *frameworks* nem muitas publicações sobre esse tema, o que implica num grande esforço de análise e generalização de diversas soluções para compilar características comuns (Madeira et al., 2000).

Uma vez definida a arquitetura do *framework*, os seus diversos módulos são então implementados. A implementação dos módulos, portanto, envolve a criação de algoritmos específicos para diversas tarefas existentes. Geralmente durante a fase de implementação há sempre uma longa etapa de otimização. As otimizações realizadas no *framework* são de extrema importância, pois o seu desempenho tem impacto direto em todos os jogos construídos sobre o *framework*.

4.8 Trabalhos Relacionados

Nesta seção é apresentado um resumo das características básicas de alguns dos principais *frameworks* encontrados na literatura.

4.8.1 Amphibian

O *framework* Amphibian (Menezes, 2003) (Bittencourt, 2004) é projetado para o desenvolvimento de *game engines* executáveis em diferentes plataformas do Sistema Operacional Microsoft Windows, utilizando lógica de aplicação e reutilização de vários

componentes. Desta forma, busca-se otimizar o processo de desenvolvimento de jogos computadorizados.

O objetivo do Amphibian é fornecer um conjunto de classes para o desenvolvimento de *engines* multimídia multiplataforma, onde as aplicações oriundas destes *engines* serão orientadas à lógica da aplicação, ou seja, a lógica de determinado domínio poderá ser portada para diferentes plataformas e visualizada de diferentes formas, assim garantindo maior reusabilidade.

O projetista do Amphibian procurou definir uma *game engine* com quatro módulos principais - Interação, Comunicação, Controle e Visualização.

O componente de Interação relaciona-se com os periféricos de entrada, tais como teclado, mouse, *joystick*, caneta utilizada por *handhelds* e teclas de telefones celulares. O componente de Comunicação permite a criação de jogos em rede, utilizando protocolo *Transmission Control Protocol/Internet Protocol* (TCP/IP). O componente de Controle é responsável por manter a lógica do jogo propriamente dita e manipular os objetos lógicos. O componente de Visualização cria a imagem que representa o estado atual do jogo sob a perspectiva de determinada entidade controlada pelo jogador.

A arquitetura do Amphibian permite utilizar qualquer Máquina Virtual Java, portanto qualquer sistema operacional que ofereça uma máquina virtual, pode visualizar o mesmo jogo.

As principais funcionalidades oferecidas pelo Amphibian são:

- Mecanismos de visualização 2D ou 3D;
- Música e sons;
- Gerenciamento de eventos gerados por diferentes dispositivos de entrada;
- Processamento lógico;
- Persistência do estado atual do jogo;
- Criação de jogos em rede usando uma arquitetura cliente/servidor.

4.8.2 wGEM

O *framework* wGEM (*Wireless Game Engine for Mobile Devices*) (Pessoa et al., 2002) (Pessoa, 2001) é um *framework* de desenvolvimento de jogos para dispositivos móveis utilizando a tecnologia J2ME/MIDP (*Mobile Information Device Profile*).

O *framework* wGEM é resultado de uma generalização e adaptação de características encontradas em vários *frameworks* de desenvolvimento de jogos para PCs às limitações dos dispositivos móveis e da tecnologia J2ME e consiste basicamente de uma API para desenvolvimentos de jogos para dispositivos móveis.

O *framework* foi implementado sobre o *profile* MIDP que abrange celulares e *paggers*. Este estende a API básica do CLDC (*Connected, Limited Device Configuration*) definindo um conjunto de classes específicas para celulares e *paggers*, responsáveis, por exemplo, por manipular a entrada de dados e a tela do dispositivo, e definir o formato em que as aplicações devem ser desenvolvidas a fim de serem executadas pelos dispositivos.

Os módulos do *framework* wGEM são:

- Objeto do jogo - Representação do objeto que contém as informações necessárias para o seu gerenciamento (como posição e representação gráfica) e também as funcionalidades necessárias para a execução do jogo (como detecção de colisão);
- Mapa do jogo - O *framework* provê a utilização de mapas para facilitar o desenvolvimento de jogos com cenários complexos. Recursos como *scrolling* e mapas baseados em *tiles* são oferecidos;
- Gerenciamento de Objetos - É responsável por inicializar, armazenar e controlar o acesso aos objetos do jogo que ele gerencia;
- Gerenciamento de Entrada - Identifica eventos do teclado do dispositivo e encaminha o ocorrido para o gerenciador do jogo;
- Gerenciamento Gráfico - Realiza a apresentação gráfica do ciclo do jogo,

coordenando o processo de desenho do mapa e dos objetos do jogo;

- Gerenciamento do jogo - Implementa o ciclo do jogo, utilizando os módulos de gerenciamento de objetos, do mapa do jogo, de entrada e gráfico;
- Gerenciamento de Rede - Implementa as funcionalidades de *download* e *upload* de arquivos utilizando o protocolo HTTP suportado pelo *profile* MIDP;
- Gerenciamento de Aplicação - Implementa as funcionalidades exigidas pela KVM às aplicações do *profile* MIDP para que estas possam ser gerenciadas;
- Integração com Editor de Cenários - Oferece as funcionalidades necessárias para importação de mapas produzidos por meio da utilização de algum editor de cenários.

4.8.3 **FORGE V8**

O *framework* FORGE V8 (Madeira, 2001) é um *engine* para jogos e aplicações multimídia, oferecendo suporte às diversas características para estas aplicações como: interface gráfica, sonorização, inteligência e modelagem física de personagens, e multiusuário em rede, além de ferramentas tais como editores de cenários.

O FORGE V8 possibilita a construção de jogos em 2D, 2 1/2D e 3D, utilizando uma mesma arquitetura de baixo nível para renderização em 3D, tanto para ambientes abertos quanto para ambientes fechados.

As características do FORGE V8 são apresentadas por Madeira (Madeira, 2001) em sete categorias: interface gráfica, sonorização, inteligência e modelagem física de personagens, controle de objetos, multiusuário em rede, editores de cenários e genéricos.

O FORGE V8 tem um módulo principal, a camada de aplicação que é responsável pelo gerenciamento das aplicações desenvolvidas com base neste *framework*.

A camada de aplicação é a camada de mais alto nível do *framework*, responsável pela interligação entre este e os projetos de aplicações e ferramentas. O ideal no desenvolvimento de um *framework* reutilizável é que as aplicações nunca tenham seus componentes específicos inseridos neste *framework*. Portanto, esta camada contém o

módulo de aplicação para atingir essa separação. Este módulo integra e controla a inicialização, configuração, execução e finalização de todas as outras camadas do *framework* (camada de sistema e camada de gerenciadores).

A camada de sistema é a camada de mais baixo nível, responsável por realizar toda comunicação com o *hardware*. Essa camada gera uma biblioteca de sistema que tem o objetivo de executar todas as operações dependentes de plataforma para as aplicações, sendo a única que necessita de extensões caso se deseje portar o *engine* para diferentes plataformas.

A camada de gerenciadores é uma camada de nível intermediário, responsável pelo controle da execução da aplicação, tratando os processos pela lógica do jogo. Ela contém diversos módulos que provêm funcionalidades específicas e gerais, do mesmo modo que a camada de sistema. Os módulos de funcionalidade específica são: gerenciador gráfico, gerenciador sonoro, gerenciador de objetos, gerenciador de eventos, gerenciador de inteligência artificial, gerenciador de modelagem física e gerenciador de multiusuário. Os módulos de funcionalidades gerais são: *console* e suporte para outros módulos.

4.8.4 FORGE 16V

O FORGE 16V (Rocha, 2003) foi criado levando em consideração os problemas ocorridos com o FORGE V8. A abordagem utilizada foi a de desenvolver um *engine* com menos funcionalidades, mas que sejam bem implementadas. Isto possibilita um aprendizado mais gradativo em relação ao projeto anterior. O domínio dos jogos que podem ser desenvolvidos utilizando o FORGE 16V foi restringido para os jogos bidimensionais e isométricos. Mesmo com esta redução, existe um grande número de jogos que podem ser desenvolvidos.

O FORGE 16V foi desenvolvido usando apenas o DirectX como base. Esta escolha restringe a plataforma do jogo para o Sistema Operacional Windows.

A arquitetura utilizada pelo FORGE 16V foi a modular. Isto permite que futuramente novas funcionalidades sejam implementadas, seguindo a filosofia do aprendizado gradativo.

Este possui os seguintes módulos principais, descritos abaixo:

- Gerenciador Principal: Este é o módulo principal do *framework*. Através dele é possível ter acesso a todos os outros módulos. Uma das funções mais importantes deste módulo é controlar o fluxo de dados entre os módulos e o comportamento padrão do *framework*. É ele que contém o *loop* principal do jogo;
- Módulo Gráfico: Este é um dos módulos mais importantes do *framework*. A função deste módulo é ajudar na construção de uma cena gráfica a cada iteração do *loop* principal;
- Módulo de Entrada: O módulo de entrada é instanciado automaticamente pelo módulo principal. Ele deve verificar quais os dispositivos de entradas estão disponíveis no computador. Além disso, ele deve disponibilizar um mecanismo de consulta do estado atual de tais dispositivos;
- Módulo de Som: Basicamente ele carrega arquivos de sons que podem ser tocados em ambientes diferentes. Alguns ambientes podem tocar mais de um som ao mesmo tempo;
- Módulo de Log: Toda aplicação deve prover um melhor meio de *debug*. A idéia é que o cliente repasse a mensagem de *log* para o dispositivo interessado em recebê-las (rede, *console* gráfico, arquivo, etc).

4.8.5 FreeMMG

O FreeMMG (Cecin et al., 2003) é um *framework* (ou *engine*) para MMG (*Massively Multiplayer Games*). Os MMG são plataformas de simulação interativa em tempo real que simulam ambientes virtuais. Atualmente MMG comercialmente disponíveis são voltados ao entretenimento, contudo, outras aplicações podem ser encontradas, como ferramenta para estudos econômicos, plataforma educativa, etc.

O FreeMMG visa sanar o problema do gargalo do MMG, que é o custo do servidor. Para este fim, são propostas mudanças tanto no algoritmo de simulação utilizado (menos centralizado, mais cooperativo) quanto no modelo de distribuição empregado

(menos cliente/servidor, mais ponto a ponto e distribuído).

O componente central do FreeMMG é uma replicação aproximada do ponto a ponto.

Primeiramente, o estado global do jogo é dividido em segmentos. Por exemplo, um jogo de guerra com um terreno 2D pode ser dividido em segmentos quadrados. Então, cada segmento é ajustado para funcionar com uma replicação independente de simulação do ponto a ponto. Os objetos do jogo podem mover-se de um segmento para outro por um processo de transferência: primeiramente, o objeto é removido do segmento da origem e emitido ao segmento do destino, onde será programado para a inserção em uma volta futura.

Servidor é responsável por persistir todos os registros do jogador e autenticar todos os jogadores antes que esteja permitido se juntar ao jogo. O servidor é também responsável para criar as informações iniciais do mundo da simulação e dividi-los em segmentos.

O servidor determina a área atual de interesse de um jogador. Na simulação distribuída, a área de interesse de um jogador é a parte do mundo que o jogador pode potencialmente interagir. No FreeMMG, a área de interesse de um jogador é um conjunto de segmentos do mundo. Como exemplo, o servidor pode determinar que um jogador esteja interessado em todos os segmentos, que têm pelo menos um objeto do jogo que pertence ao jogador.

4.8.6 FunGen

O *framework* FunGEn (*Functional Game Engine*) (Furtado & Santos, 2002), é um *engine* para jogos bidimensionais desenvolvido através da HOpenGL, na linguagem de programação funcional *Haskell*, destinado ao desenvolvimento de jogos nesta mesma linguagem.

Entre algumas das funcionalidades e facilidades oferecidas pelo FunGEn a um programador de jogos, inclui-se o gerenciamento completo dos objetos do jogo (inicialização, atualização, localização, remoção, agrupamento e renderização). A renderização é um dos pontos a ser destacado neste tópico: um objeto do jogo pode estar associado a uma simples primitiva gráfica (como círculos, quadrados ou triângulos) ou ser representado por *sprites* animadas, cujas figuras (quadros) são carregadas de

arquivos gráficos.

A detecção de colisão entre objetos é outra funcionalidade oferecida pelo *engine* ao programador que, desta forma não precisa se deter em cálculos envolvendo posição, tamanho ou velocidade dos objetos para checar se eles se chocarão ou não.

O gerenciamento do mapa de fundo é mais um tópico que visa a simplificação do trabalho do programador. O mapa pode ser especificado por cores, texturas simples ou até construções mais complexas, como um mapa de *tiles* em que informações específicas podem estar associadas a cada *tile* (como a dificuldade de transposição de um terreno, por exemplo). Algumas funcionalidades em torno do mapa de fundo, como auto-rolamento (*scrolling*), também são oferecidas ao programador.

O controle das entradas fornecidas pelo jogador, provenientes do mouse ou teclado, é realizado de modo bastante intuitivo pelo FunGEn: cada tecla (ou botão do mouse) é associada a um evento e a uma ação. O evento indica um dos três estados possíveis para a tecla (ou botão): pressionada, liberada ou mantida continuamente pressionada. A ação, por sua vez, é definida pelo programador e indica o que deve acontecer no jogo em resposta à tecla e evento correspondentes.

O primeiro benefício observado na programação de jogos utilizando o FunGEn é uma redução significativa do número de linhas de código.

Por ser baseado numa biblioteca gráfica independente de plataforma (OpenGL), o próprio FunGEn também é independente: o mesmo código-fonte poderia ser compilado nas plataformas Windows, UNIX ou MacOS, por exemplo. Além disso, é interessante ressaltar que jogos implementados através do FunGEn herdam da biblioteca OpenGL algumas importantes características como robustez e estabilidade.

4.8.7 Guff

O *framework* Guff (*games-uff*) (Valente, 2005) foi desenvolvido para a aplicação de alguns dos conceitos sobre o desenvolvimento de jogos.

O *framework* Guff possui duas partes principais: a camada de aplicação e um *toolkit*. A camada de aplicação determina as interfaces e a arquitetura dos programas que usam o

framework. O *toolkit* representa um conjunto de classes auxiliares que podem ser usadas para desenvolver os jogos.

Em linhas gerais, o *framework* Guff foi projetado para atender aos seguintes requisitos: oferecer uma interface que modele uma possível arquitetura genérica de um jogo; proporcionar um conjunto de ferramentas para auxiliar no desenvolvimento dos programas; funcionar em Windows e Linux; gestão automática de todos os recursos; facilidade de uso; reuso de bibliotecas já existentes; possibilitar que a configuração de parâmetros da aplicação seja determinada fora do código fonte.

Para que o *framework* possa funcionar em Windows e Linux, é necessário criar uma camada que abstraia os detalhes específicos do sistema operacional. Esse problema é resolvido com o uso da biblioteca *Simple DirectMedia Layer* (SDL). A SDL é uma biblioteca multimídia que oferece acesso de baixo nível a dispositivos de entrada, áudio e sistema de janelas, entre outros. Seu código é aberto e existem implementações para vários sistemas operacionais.

Além da portabilidade, outro objetivo do uso de bibliotecas é evitar a re-implementação de funcionalidades já disponíveis. De fato, é comum na indústria de jogos que o reuso de código não seja valorizado devidamente, o que dificulta o desenvolvimento.

A gestão de recursos é uma das tarefas mais corriqueiras que devem ser realizadas em uma aplicação. Entretanto, a realização dessa tarefa é de responsabilidade do desenvolvedor, tradicionalmente. Caso essa tarefa não seja realizada corretamente, vários problemas podem ser manifestados, como perda de recursos, *deadlocks*, “bugs aleatórios” (um defeito difícil de reproduzir), entre outros.

O *framework* Guff faz extenso uso da gestão automática de recursos, isto é, um objeto que é capaz gerir seus recursos automaticamente.

O *toolkit* do *framework* Guff oferece soluções para diversos módulos como visualização, matemática, áudio e dispositivos de entrada.

4.9 Conclusão do Capítulo

Em questão ao que foi discutido anteriormente, *framework* são estruturas genéricas e

facilmente reutilizáveis em várias aplicações, proporcionando um alto grau de desenvolvimento e qualidade. A utilização de *framework* em jogos tem como objetivo atingir uma forma completa, reutilizável, flexível, simplificada no desenvolvimento e de fácil compreensão em seus propósitos.

Foram também apresentados os *frameworks e engines* relacionados a jogos encontrados na literatura, são eles: Amphibian, wGEM, FORGE V8, FORGE 16V, FreeMMG, FunGen e o Guff. Na seção 7.1 é apresentada uma tabela de comparação dos *frameworks e engines* apresentados em relação ao *framework* FMMG.

Capítulo 5 - Framework FMMG

Este capítulo apresenta o *framework* FMMG (*Framework for Mobile Multiplayer Games*) proposto nesta Tese, baseando-se nos conceitos e projetos descritos nos capítulos anteriores.

A intenção é apresentar em detalhes cada um dos módulos e a estrutura do *framework* proposto, e como eles se integram de forma a constituir uma arquitetura a ser utilizada no desenvolvimento de Jogos *Multiplayer* Móveis.

Nas seções deste capítulo são apresentados os requisitos, a arquitetura, os componentes e a estrutura de classes do *framework*.

5.1 Introdução

Um *framework* é um conjunto de componentes de *software* organizado e projetado para a construção de aplicações de um determinado domínio específico. É importante notar que um *framework* de jogos não é um conjunto de funções, ou uma API, relativas a um determinado objetivo. Ele também não é um produto pronto. O *framework* de jogos deve ser integrado junto com a implementação do *game design* (Azevedo, 2005) (Pessoa, 2001), para formar o jogo completo. Então, pode-se perceber que, o objetivo do *framework* é fazer com que os desenvolvedores de jogos, possam implementar apenas o *game design* do seu jogo.

Desenvolver jogos para dispositivos móveis é uma tarefa complexa. Além da dificuldade de desenvolver o jogo em si, o desenvolvedor deve se ater a vários problemas inerentes a estes dispositivos, como limitação de memória, recursos, processamento, tamanho da tela e entrada não apropriada (Brooks, 1987) (Bethke, 2003). Desta forma, o uso de *frameworks* de jogos vem para capturar todo o conhecimento e experiência do desenvolvedor em uma determinada plataforma, podendo prover um conjunto de componentes projetados e otimizados para os dispositivos. Além disto, a utilização de *frameworks* visa a reusabilidade de código, elementos inerentes ao jogo e características da plataforma.

Neste contexto, o *framework* FMMG proposto, procura ser o mais genérico possível de forma a ser base para qualquer tipo de aplicação de Jogos *Multiplayer* Móveis, facilitando a portabilidade de programas entre plataformas de desenvolvimento para dispositivos móveis que utilizem a linguagem de programação orientada a objetos.

O *framework* FMMG visa disponibilizar um conjunto de classes que auxiliam o desenvolvedor na implementação de Jogos *Multiplayer* Móveis. O *framework* FMMG disponibiliza um ambiente consistente de funcionamento, em que o desenvolvedor deve se preocupar exclusivamente com o comportamento que deseja para o seu jogo.

As outras funcionalidades ficam a cargo do *framework* FMMG resolver, como por exemplo, as possíveis incompatibilidades no funcionamento do ambiente operacional (cada aparelho, mesmo que suportando a mesma plataforma, pode ter funcionalidades e funcionamento diferentes e cabe ao *framework* fazer as correções necessárias).

De posse dos recursos disponibilizados pelo *framework*, o desenvolvedor está livre para implementar seu jogo da melhor forma possível, independentemente do seu nível de conhecimento do ambiente em que o *framework* está sendo utilizado.

Nas próximas seções apresenta-se a metodologia utilizada, os requisitos, a arquitetura e os componentes que compõem o *framework* FMMG.

5.2 Metodologia

Uma metodologia de desenvolvimento de *frameworks* consiste em procedimentos que abrangem a captura das características de um domínio, construção e validação da estrutura de um *framework*. As metodologias se caracterizam por estabelecer o processo de desenvolvimento de *frameworks* em linhas gerais, sem se ater à definição de técnicas de modelagem ou detalhar o processo de desenvolvimento, conforme apresentado na seção 4.6.

A metodologia utilizada para o *framework* FMMG é baseada na metodologia de projeto dirigido por pontos de flexibilização (*Hot Spot Driven Design*) que enfatiza a busca de *hot spots* na estrutura de classes de um domínio. *Hot spots* são as partes da estrutura de classes de um *framework* que devem ser mantidas flexíveis para garantir que o

framework cubra as diferentes aplicações do domínio tratado. O desenvolvimento de *frameworks* auxiliado pela metodologia *Hot Spot Driven Design* segue a seguinte seqüência de procedimentos:

- Identificação de classes - O desenvolvedor de *frameworks*, a partir das informações obtidas na análise do domínio, define uma estrutura de classes inicial para o *framework*;
- Identificação de *hot spots* - É preciso identificar que aspectos diferem de aplicação para aplicação e definir o grau de flexibilidade que deve ser mantido em cada caso;
- Projeto do *framework* - Modificação da estrutura de classes definida no início do processo, de modo a comportar a flexibilidade requerida;
- Refinamento da estrutura do *framework* - Refinamento da estrutura de classes do *framework* a partir das informações obtidas na análise do domínio. Nessa etapa, se o *framework* não for flexível o suficiente para generalizar as aplicações do domínio, deve-se voltar à etapa de definição do *hot spots*.

5.3 Requisitos

Em linhas gerais, os requisitos do *framework* FMMG são apresentados a seguir:

- Arquitetura modular, para que possa ser reutilizado em cada novo jogo;
- Bom nível de abstração, ao prover objetos com funcionalidades já implementadas;
- Definição de objetos básicos;
- Detecção e gerenciamento de eventos gerados por teclado, mouse, *joysticks*, etc;
- Funcionalidades úteis a jogos 2D, 3D, como iluminação e transformações geométricas, etc.;
- Execução dos sons e músicas do jogo em resposta a eventos ocorridos;

- Implementação de algoritmos para troca remota de dados, sincronização das informações compartilhadas do jogo, etc;
- Exibição de imagens, utilizando mecanismos de visualização 2D ou 3D;
- Criação de jogos em rede usando uma arquitetura cliente/servidor;
- Oferecer um conjunto de classes para auxiliar no desenvolvimento dos jogos;
- Permitir o funcionamento em vários tipos de dispositivos móveis;
- Facilidade de uso e reuso de bibliotecas já existentes.

5.4 Arquitetura

A arquitetura do *framework* FMMG foi definida de acordo com os requisitos identificados e foi decidido que seria um *framework* de arquitetura aberta, ou seja, os desenvolvedores têm acesso ao código fonte e podem fazer modificações.

O padrão *Model View Controller* (MVC) (Krasner & Pope, 1988) foi adaptado e utilizado para o *framework* FMMG. O padrão MVC fornece uma maneira de dividir as funcionalidades envolvidas na manutenção e apresentação dos dados de uma aplicação. O padrão MVC não é novo e foi originalmente desenvolvido para mapear as tarefas tradicionais de entrada, processamento e saída para o modelo de interação com o usuário.

No padrão MVC, o *Model* representa os dados da aplicação e as regras do negócio que manipulam o acesso e a modificação dos dados. O *Model* mantém o estado persistente do negócio e fornece ao *Controller* a capacidade de acessar as funcionalidades da aplicação encapsuladas pelo próprio *Model*.

A *View* renderiza o conteúdo de uma parte particular do *Model* e encaminha para o *Controller* as ações do usuário. Além disso, a *View* acessa também os dados do *Model* via *Controller* e define como esses dados devem ser apresentados.

As vantagens do padrão MVC são:

- Como o padrão MVC gerencia múltiplos visualizadores usando o mesmo *Model* é fácil manter e atualizar sistemas;
- Torna a aplicação escalável;
- Desenvolvimento em paralelo para o *Model*, *View* e *Controller*, pois as responsabilidades são divididas e independentes.

As desvantagens do padrão MVC são:

- Requer uma quantidade maior de tempo para analisar e modelar o sistema;
- Não é aconselhável para pequenas aplicações.

O resumo do padrão MVC utilizado como base para utilização no *framework* FMMG é apresentado na Figura 5.1:

Figura 5.1 - Infraestrutura MVC

- *Controller* - É a implementação do componente controlador do *framework* FMMG, que é responsável por controlar as ações de manipulação dos objetos lógicos. Estas classes também utilizam alguns componentes genéricos de suporte oferecidos pelo *framework* FMMG. Abaixo segue um resumo da descrição do papel do *Controller* para o *framework* FMMG:
 - ✓ Define e gerencia o comportamento da aplicação;
 - ✓ Mapeia as ações do usuário em alterações da interface *Model*;
 - ✓ Seleciona e controla a *View* ativa.

- *Model* - São as classes responsáveis pela lógica do jogo. Estas classes utilizam somente algumas classes genéricas de suporte oferecidas pelo *framework* FMMG. Abaixo segue um resumo da descrição do papel do *Model* para o *framework* FMMG:
 - ✓ Encapsula o estado da aplicação;
 - ✓ Ponto de acesso às funcionalidades;
 - ✓ Notifica as mudanças à *View*;
 - ✓ Responde as consultas da *View*.
- *View* - São as classes que definem os objetos entrada e saída. Entre elas estão a forma de visualização de cada objeto lógico, o componente referente à cena responsável pelo desenho destes componentes em uma superfície gráfica e a definição de mecanismos de entrada para interagir com aplicação. Abaixo segue um resumo da descrição do papel do *View* para o *framework* FMMG:
 - ✓ Renderiza e trata informações de apresentação da aplicação;
 - ✓ É a camada de interface com o usuário;
 - ✓ É usada para receber a entrada de dados e apresentar o resultado;
 - ✓ Recebe pedido de atualizações do *Model*.

5.4.1 Controller

Controller é uma classe abstrata que deve ser implementada pelo desenvolvedor do jogo e informada ao *framework*. Através desta classe o *framework* pode informar ao jogo os eventos ocorridos nos dispositivos móveis. A classe *Controller* possui todos os métodos que o *framework* utiliza para se comunicar com o jogo, conforme mostra a Figura 5.2.

Figura 5.2 - Classe *Controller*

Os métodos possuem as seguintes funções:

- *start()* - Método executado somente uma vez, quando o *jogo* é iniciado. Este método deve ser utilizado para alocar todos os recursos básicos que o jogo precisará em seu funcionamento;
- *stop()* - Método executado somente uma vez, quando o jogo é finalizado. Este método deve ser utilizado para desalocar todos os recursos que foram disponibilizados ao jogo pelo método *start()*;
- *suspend()* - Método executado toda vez que o dispositivo móvel necessitar realizar algumas de suas funções. Exemplo: receber uma chamada de voz, uma *shot message*, avisos de bateria fraca, etc. Este método deve ser utilizado para colocar o jogo em estado de bloqueado, onde os recursos utilizados pelo jogo devem ser liberados para que o sistema operacional do dispositivo móvel possa executar suas outras funções;
- *resume()* - Método executado toda vez que o sistema operacional finalizar uma de suas funções que tenham causado a interrupção do jogo (*suspend()*). Este método deve ser utilizado para realocar todos os recursos que o jogo necessita e colocá-lo em estado de execução novamente;
- *run()* - Método executado a cada fração de tempo (dependendo da quantidade de *frames* por segundo que foi configurado no *framework*). Este método é a porta de entrada para a execução do jogo. Ele deve ser implementado de tal forma que

seja executado, e retorne o controle ao *framework* (não pode haver laços infinitos). Este tipo de implementação deve ser respeitado, pois o ambiente operacional utilizado para a implementação do *framework*, deve suportar multitarefa cooperativa e um processo que não retorne o controle ao ambiente poderia prejudicar o funcionamento do dispositivo móvel, não permitindo nem que as teclas pressionadas sejam reconhecidas. O método *run()* é o laço principal do jogo, porém, este laço é realizado/controlado pelo *framework*;

- *event()* - Método executado toda vez que ocorrer um evento no ambiente operacional e que possa vir a ser de interesse do jogo;
- *changeState()* - Método executado para alterar o estado macro do jogo.

A implementação da classe *Controller* é responsável pela manutenção do estado em que se encontra o jogo. É a classe que define qual ação e visão a ser executada, denominada de estados macro do jogo.

5.4.2 Model

Model é a classe abstrata que deve ser implementada para representar a ação de um estado do jogo. Toda a inteligência que deve ser implementada para determinado estado do jogo é encapsulada na implementação da *Model*, conforme mostra a Figura 5.3.

Basicamente, esta classe *Model* possui os mesmos métodos da classe *Controller*, com as mesmas finalidades. Pode ser denominada de estados micro do jogo, pois sua responsabilidade se refere única e exclusivamente à implementação da inteligência de um macro estado do jogo. Um macro estado do jogo pode ser composto de diversos micros estados do jogo e estes micros estados são de responsabilidade da implementação da classe *Model*.

Figura 5.3 - Classe *Model*

Os métodos possuem as seguintes funções:

- *start()* - Método utilizado para iniciar as ações do estado do jogo;
- *stop()* - Método utilizado para desalocar todas as ações que foram disponibilizados ao jogo pelo método *start()*;
- *suspend()* - Método executado toda vez que o dispositivo móvel necessitar realizar a suspensão de suas funções na ação do estado do jogo;
- *resume()* - Método executado toda vez que o sistema operacional finalizar uma de suas funções que tenham causado a interrupção do jogo (*suspend()*) na ação do estado do jogo;
- *run()* - Método executado a cada fração de tempo. Este método é a porta de entrada para a execução da classe.

5.4.3 View

A classe abstrata *View* deve ser implementada para representar graficamente um determinado estado do jogo. A classe possui basicamente os mesmos métodos que a classe *Controller*, com exceção do método *run()* que nesta classe é substituído pelo método *draw()*. Quando chamado, o método *draw()* é responsável pela atualização da apresentação, conforme mostra a Figura 5.4.

Figura 5.4 - Classe *View*

Os métodos possuem as seguintes funções:

- *start()* - Método utilizado para iniciar a *View* do jogo;
- *Stop()* - Método utilizado para desalocar os recursos da *View* que foram disponibilizados ao jogo pelo método *start()*;
- *suspend()* - Método executado toda vez que o dispositivo móvel necessitar realizar a suspensão de suas funções na *View* do estado do jogo;
- *resume()* - Método executado toda vez que o sistema operacional finalizar uma de suas funções que tenham causado a interrupção do jogo (*suspend()*) na *View* do estado do jogo;
- *draw()* - Método executado a cada fração de tempo. Este método apresenta e renderiza as informações de um jogo.

5.4.4 Funcionamento do Framework FMMG

A infraestrutura proposta para utilização do padrão MVC no *framework* FMMG é baseada na utilização do *Controller*, *Model* e *View* descrita nas seções anteriores, e nas estruturas *State* e classe *DataObject*. A estrutura *State* determina a ocorrência atual de um jogo e depende da entrada ou da ação realizada pelo jogador, denominada estados macro do jogo. A classe *DataObject* é responsável pela estrutura de dados do estado a ser manipulado no jogo.

A Figura 5.5 mostra a estrutura do padrão MVC utilizado pelo *framework*.

Figura 5.5 - Estrutura do Padrão MVC pelo *Framework*

A comunicação entre a classe *Model* e a classe *View* se dá através da implementação de uma classe *DataObject*, conforme a Figura 5.6. Esta classe *DataObject* possui todas as informações necessárias para a execução de métodos da classe *Model* e da classe *View*.

Figura 5.6 - Classe *DataObject*

A classe *Controller* controla os estados macros do jogo pelo método *changeState()* e são instanciados para representá-lo a partir dos seguintes estados:

- *StateDataObject*;
- *StateModel*;
- *StateView*.

O código do *changeState()* é apresentada na Figura 5.7.

```

...
Controller::changeState( int state )
{
    switch ( state )
    {
        case state_1:
            stateDataObject    = new State_DataObject( ) ;
            stateAction         = new State_Action(stateDataObject ) ;
            stateView           = new StateEstado_View(stateDataObject ) ;
        }
    }
}
...

```

Figura 5.7 - Método *changeState*

A cada iteração do *framework* FMMG, o método *run()* da classe *Controller* é chamado. Baseado em seu controle de estados macro, as classes representantes do macro estado em questão são acionadas, como mostra a Figura 5.8.

```

...
Controller::run( )
{
    stateAction->run( ) ;
    stateView->draw( ) ;

    if ( stateDataObject->EndOfState( ) )
    {
        changeState( nextState( ) ) ;
    }
}
...

```

Figura 5.8 - Método *run*

Esta infraestrutura do *framework* FMMG dá a liberdade para desenvolver o jogo da forma que for necessária e ao mesmo tempo oferece a possibilidade de reusar a implementação de um estado em outro jogo, pois a implementação de um estado é dada por meio da implementação do conjunto das três classes em questão (*DataObject*, *Model* e *View*). Isso é uma vantagem no desenvolvimento, já que as partes dos jogos (exemplo: *SplashScreen*, *Menu*, *Rank*, *About* e *Help*) podem ser reusadas e com isso o desenvolvedor deve se concentrar no desenvolvimento do estado *GamePlay* do jogo, conforme mostra a Figura 5.9. Tem se também como vantagem o aumento da velocidade de desenvolvimento, podendo distribuir os estados existentes do jogo entre os desenvolvedores. Sendo necessário somente integrar as partes para que o jogo seja completado, tornando assim uma estrutura fracamente acoplada.

A infraestrutura MVC utilizada para os jogos baseada em estados pode conter outros

estados, à medida da necessidade do desenvolvedor de jogos.

Um jogo pode utilizar os seguintes estados a partir do padrão MVC, conforme mostra o exemplo da Figura 5.9:

- *SplashScreenState* - Trata da abertura inicial de um jogo;
- *MenuState* - Trata da estrutura do menu do jogo;
- *GamePlayState* - Trata da execução do jogo;
- *RankState* - Trata da apresentação de uma lista do *ranking* de um determinado jogo;
- *HelpState* - Trata da apresentação de ajuda do jogo;
- *AboutState* - Trata das informações sobre o jogo.

Figura 5.9 - Exemplo da Utilização do Padrão MVC em um Modelo de Jogo

As classes *Model* e *View* são responsáveis por chamar outras estruturas de classes que fazem parte dos módulos dos componentes do *framework* FMMG. Estes módulos possuem as funcionalidades para o desenvolvimento de um jogo, conforme descrito na

seção 5.3.

Com a estrutura da Figura 5.9, é possível criar jogos interessantes e bastante complexos de forma fácil, simples e rápida, devido ao fato que a complexidade do tratamento dos eventos enviados pelo ambiente de desenvolvimento é encapsulada pelo *framework*, que envia ao *Controller* os eventos de forma mais amigável, por meio da chamada de seus métodos.

Uma possível desvantagem pode ocorrer em jogos que exijam tempo real, pois a estrutura pode comprometer o desempenho por delegar a execução de muitas das tarefas simultaneamente.

O *framework* FMMG disponibiliza ao desenvolvedor um ambiente consistente de funcionamento, em que deve se preocupar exclusivamente com o comportamento que deseja para o jogo, deixando para o *framework* resolver possíveis incompatibilidades no funcionamento do ambiente operacional.

5.5 Estruturação do Framework

De acordo com os diversos *frameworks* e *engine* de jogos analisados (Pessoa, 2001) (Pessoa et al., 2002) (Madeira, 2001) (Rocha, 2003) (Cecin et al., 2003) (Furtado & Santos, 2002) (Bernades et al., 2004) (Menezes, 2003) (Bittencourt, 2004) foi possível identificar quais os módulos de componentes que inicialmente seriam necessários.

Abaixo, segue a descrição da arquitetura integrada de *framework* ou *engines* de jogos que foram propostas por Pessoa (Pessoa, 2001) e Bernades (Bernades et al., 2004) para PC's:

- Gerenciador de Objetos - Responsável pela interação dos objetos uns com os outros e com o cenário. É possível ter vários gerenciadores de objetos, um para cada grupo de objetos similares do jogo;
- Gerenciador do Mundo - Gerencia o estado atual do jogo, interagindo com os outros gerenciadores para informar o que deve ser feito;
- Gerenciador Principal - É a ponte de troca de informações e provê um ponto de

acesso único às funcionalidades do jogo;

- Renderização - Componentes capazes de gerar imagens do jogo com um determinado padrão de qualidade, com velocidade suficiente para manter a interatividade do jogo;
- Comunicação em Rede - Componentes responsáveis pela comunicação dos jogadores de um jogo;
- Som e Música - Componentes que emitem som ou música a partir de arquivos de formatos aceitos dentro de uma plataforma de desenvolvimento de um jogo;
- Inteligência Artificial - Componentes que implementam inteligência dos personagens de um jogo controlados pelo jogo;
- Colisão - Componentes que tratam da detecção de colisão dos objetos pertencentes a um jogo;
- Simulação Física - Componentes independentes que se caracterizam pela capacidade de simular em tempo real conjuntos de objetos com vínculos e graus de liberdade configuráveis;
- Entrada do Jogador - Componentes que tratam da captura e transmissão de comandos realizados através de dispositivos de entrada;
- Multimídia - Componentes responsáveis pela execução de vídeos e outros tipos de mídias.

Nota-se que a arquitetura de componentes descrita anteriormente é para a implementação de um *engine* ou *framework* de jogos para PC's, sendo portanto, uma arquitetura que necessita de bastante recursos em termos de memória e processamento. Sendo assim, essa arquitetura não é viável para ser implementada em plataforma de dispositivos móveis.

Diante dos *frameworks* e *engines* estudadas neste trabalho, foi possível identificar quais os principais módulos a serem incorporados no *framework* FMMG. Os módulos que

compõem a arquitetura do *framework* para dispositivos móveis são: módulos de gerenciamento, módulo básico e módulo de rede. De posse dos recursos dos módulos disponibilizados pelo *framework* FMMG, o desenvolvedor está livre para implementar seu jogo da melhor forma possível, independentemente do seu nível de conhecimento do ambiente em que o *framework* está sendo utilizado.

As classes dos módulos de gerenciamento, módulo básico e módulo de rede são capazes de interagir umas com as outras, conforme a necessidade do jogo como apresentado na Figura 5.10.

Figura 5.10 - Estrutura de Comunicação entre os Módulos

5.5.1 Módulos de Gerenciamento

As classes relacionadas são pertencentes ao módulo de gerenciamento do *framework* que apóiam os demais módulos do *framework* FMMG. O *framework* visa disponibilizar um conjunto de classes de gerenciamento e controle que auxiliam o desenvolvedor na implementação de um jogo, conforme a Figura 5.11. São elas:

- *LogManager* - Classe utilitária que auxilia o processo de depuração/*debug* de um jogo;
- *ResourceManager* - Responsável pelo gerenciamento dos recursos (imagens, *string*, etc.) que podem ser utilizados por um jogo;
- *SoundManager* - Responsável pelo gerenciamento das músicas e efeitos sonoros

de um jogo;

- *FileManager* - Responsável pelo gerenciamento dos arquivos utilizados por um jogo;
- *FontManager* - Responsável pelo gerenciamento das fontes disponíveis para um jogo.

Figura 5.11 - Classes do Módulo de Gerenciamento

5.5.2 Módulo Básico

As classes relacionadas são pertencentes ao módulo básico do *framework* que apóiam os demais módulos do *framework* FMMG. O *framework* visa disponibilizar um conjunto de classes que auxiliam o desenvolvedor na implementação de um jogo. Estas classes encapsulam APIs disponíveis no ambiente operacional do *framework*. As classes são divididas em 3 módulos: Processamento, Formato e Apresentação.

A Figura 5.12 apresenta as classes do módulo de Processamento:

Figura 5.12 - Classes do Módulo de Processamento

- *CircleQueue* - Classe que implementa a classe *Queue* de maneira a se obter uma fila circular;
- *Context* - Classe abstrata que representa o contexto do ambiente em que o *framework* está implementado;
- *Device* - Classe que disponibiliza informações e acesso aos dispositivos do dispositivo móvel;
- *File* - Classe que representa um arquivo no sistema de arquivos do dispositivo móvel;
- *List* - Classe que representa uma lista encadeada;
- *Object* - Classe base para todos os objetos do *framework*. Todas as classes

devem estender de *Object*;

- *Pool* - Classe que representa um *pool* de *Objects*. Utilizado para armazenar temporariamente os objetos;
- *Poolingtask* - Classe que implementa uma *TimerTask* e é utilizada para chamar o método *run()* do *Controller*, respeitando as informações de *frames* por segundos especificados para o *framework*. Esta classe representa o laço de funcionamento contínuo do jogo;
- *Queue* - Classe abstrata que representa as funcionalidades que uma fila deve possuir;
- *Session* - Classe que é disponibilizada pelo *framework* representando a sessão do jogo, onde podem ser armazenadas informações que poderão ser acessadas de qualquer ponto do jogo;
- *Stack* - Classe que representa uma pilha onde informações podem ser armazenadas;
- *System* - Classe que disponibiliza as funcionalidades do *framework* ao jogo. É por meio dela que o desenvolvedor tem acesso aos gerenciadores de fonte, rede, arquivos, *log* e a funções de verificação de teclas;
- *Timer* - Classe utilizada quando se deseja que uma tarefa seja agendada para execução em determinado horário, sem o controle do *framework*;
- *Timertask* - Classe abstrata que deve ser implementada para ser utilizada com a classe *Timer*. Uma *TimerTask* deve ser implementada para ser agendada uma execução.

A Figura 5.13 apresenta as classes do módulo de Formato:

Figura 5.13 - Classes do Módulo de Formato

- *Circle* - Classe responsável por criar círculos;
- *Font* - Classe que representa determinada fonte (letras);
- *Label* - Representação textual de uma informação no jogo;
- *Rectangle* - Classe responsável por criar retângulo;
- *Shape* - Classe base que representa um objeto visual do jogo;
- *String* - Classe que representa um texto;
- *Tile* - Responsável pelo gerenciamento de um *Tile* dentro de uma imagem. Representa um *frame* de imagem dentro de um arquivo de imagem que contém vários *frames* (utilizada na composição de mapas e em animações);
- *TileContainer* - Classe utilizada para armazenar (*container*) diversos *tiles* que compõem uma imagem maior (ou tela). Utilizada para agrupar imagens e facilitar o trabalho do desenvolvedor;
- *TiledImage* - Classe utilizada para construir uma imagem que possui um padrão de repetição.

A Figura 5.14 apresenta as classes do módulo de Apresentação:

Figura 5.14 - Classes do Módulo de Apresentação

- *Bitmap* - Implementação da classe *Image* para um arquivo *bitmap* do *Windows*;
- *Camera* - Responsável por percorrer, vertical e horizontalmente, um cenário e registrar ou capturar, toda cena em seu campo de visão;
- *Canvas* - Responsável pelas ações sobre a tela física do dispositivo móvel;
- *Film* - Uma lista de cenas que fazem parte e um instante de gravação de uma câmera;
- *Image* - Classe básica que representa uma imagem;
- *Layer* - Estrutura de dados que representa uma camada dentro de uma cena. É utilizada para que se possa avaliar a ordem dos objetos (diferenciar o que está no plano anterior e posterior);
- *Projector* - Classe responsável por projetar na tela o filme obtido/gerado pela câmera;
- *Scene* - Estrutura de dados que representa uma cena do jogo. Consideramos cena

uma área equivalente ao tamanho da tela. Uma cena é constituída de diversas camadas;

- *Scenery* - Estrutura de dados que representa o cenário do jogo. Um cenário é constituído de diversas cenas (*Scene*);
- *Window* - Classe *utilizada* para construir uma janela na tela do dispositivo móvel.

5.5.3 Módulo de Rede

Baseado no capítulo 3, o módulo de comunicação de rede é uma solução baseada no modelo cliente/servidor e modelo de armazenamento banco de dados replicados de mundos homogêneos para jogos. As vantagens e desvantagens desse modelo foram apresentadas na seção 3.4.

Os componentes do módulo de rede visam facilitar o desenvolvimento de Jogos *Multiplayer* Móveis. O módulo utiliza o paradigma objeto/propriedade/evento. Cada mudança ocorrida em uma propriedade é reconhecida como um evento. Para permitir que múltiplos jogadores executem e participem do mesmo jogo, cada jogador precisa estar apto a receber certas atualizações realizadas por outros jogadores. Para prover essa situação, a propriedade deve ser compartilhada tanto pelo jogador que a modifica quanto pelos outros jogadores que recebem.

Internamente, os eventos causam atualizações nas propriedades compartilhadas e são enviadas pela rede aos jogadores. Pode-se ainda, especificar ações a serem executadas em resposta a um evento, limitando-se a transferência de dados apenas às propriedades compartilhadas, diminuindo a utilização da largura de banda.

O servidor deve fornecer as seguintes capacidades:

- Um simples ponto de sincronização para consistência dos dados;
- Pré-processamento das informações do jogo;
- Filtragem de mensagens e técnicas para a redução da utilização da largura de banda no servidor.

Uma vez que o servidor é um programa, o mesmo pode executar simultaneamente com o cliente, tornando o gerenciamento do jogo transparente, ou seja, o jogo que está sendo executado não será interrompido quando um novo cliente conectar-se ao servidor.

Em um modelo cliente/servidor, desde que todas as mensagens sejam roteadas por um processo central, a escalabilidade do sistema é dependente de dois fatores:

- A habilidade de ajustar o poder da máquina que hospeda o processamento central;
- A habilidade de compartilhar a carga de trabalho.

Em resumo, pode-se dizer que o módulo de comunicação de rede gerencia a conexão, a notificação de *login/logout*, compartilhamento de dados e processamento de dados do Jogo.

Cliente

Partindo da idéia que conexão à rede é um recurso limitado em um celular (em média pode-se ter 3 conexões a rede por aparelho celular), desenvolveu-se a seguinte solução para atender maior número de conexão.

No modelo de rede do FMMG desenvolveu-se uma estrutura, cliente/servidor, que é capaz de criar um túnel TCP ou UDP, por meio de uma única conexão à rede. Uma vez criado este túnel, o jogo pode se conectar com um número ilimitado de servidores de jogos, sendo o limite a capacidade de memória.

O funcionamento do túnel é conforme as descrições a seguir:

- Por meio da classe *NetManager* o programador cria um túnel TCP ou UDP com um determinado servidor remoto;
- Uma vez estabelecido o túnel, o cliente pode solicitar conexões com os servidores de jogos por meio dos métodos *open* e *bind* da classe *NetManager*;
- Quando uma conexão é criada, um objeto *socket* é retornado (*StreamSocket* para TCP ou *DatagramSocket* para UDP);

- Por meio do objeto *socket* a comunicação é realizada.

A arquitetura para a comunicação de rede para o módulo cliente é apresentada na Figura 5.15 e os seus componentes são descritos abaixo:

Figura 5.15 - Estrutura do Módulo Cliente

- *NetManager* - é o responsável por manter a comunicação com o servidor remoto. A classe *NetManager* tem as seguintes funcionalidades:
 - ✓ Inicialmente a classe *NetManager* gerencia o túnel. Quando o cliente quer se conectar a um servidor de jogo, é chamado os métodos *open* ou *bind* do *NetManager*;
 - ✓ Chama a classe *SocketFactory* para criar o *socket* correspondente (*open* = *StreamSocket*, *bind* = *DatagramSocket*);
 - ✓ De posse do *socket*, a classe *NetManager* registra o *socket* na classe *ConnectionManager* que realiza o trabalho de se conectar ao servidor do jogo;
 - ✓ Todos os dados que são trafegados pela rede são encapsulados e desencapsulados pela classe *NetManager* (métodos *multiplex* ou *demultiplex*) em um protocolo que o *NetManager* e o servidor remoto conhecem.

- *ConnectionManager* - é o responsável por manter a comunicação com o servidor de jogos (não é a mesma coisa que servidor remoto, pois um servidor remoto pode ter vários jogos e vários servidores de jogos).
- ✓ Quando o cliente quer enviar dados ao servidor de jogo, o mesmo escreve no *socket* (método *write*), que por sua vez encaminha os dados para o *ConnectionManager* que encapsulará os dados em um protocolo que o Servidor de Jogo entende;
- ✓ Quando o *NetManager* recebe dados do servidor remoto, primeiro desencapsula os dados enviados pelo servidor remoto e os encaminha para o *ConnectionManager* para que o mesmo possa desencapsular as informações vindas do servidor de jogo.
- ✓ *ConnectionManager* então descobre a qual *socket* aqueles dados pertencem, então os disponibiliza no *buffer* do *socket*. Quando o programa executar um método *read* no *socket*, as informações estarão no buffer e serão disponibilizadas ao programa;
- ✓ *ConnectionManager* é o responsável por permitir a conexão de vários *socket* a vários servidores de jogos. Este esquema é útil em aplicações de *Instant Message* e em jogos ponto a ponto.
- *SocketFactory* - é o responsável por criar o *socket* solicitado (*StreamSocket* ou *DatagramSocket*);
- *StreamSocket* - esta classe representa uma conexão TCP com o servidor de jogos e é por meio da classe que é possível escrever e ler dados no servidor de jogos;
- *DatagramSocket* - esta classe representa uma conexão UDP com o servidor de jogos e é por meio da classe que é possível escrever e ler dados no servidor de jogos.

A Figura 5.16 apresenta o diagrama de classe do módulo cliente do *framework* FMMG.

Figura 5.16 - Diagrama de Classe do Módulo Cliente

Servidor

Partindo da idéia que conexão à rede é um recurso limitado nos aparelhos celulares, desenvolveu-se a seguinte estrutura de comunicação para o funcionamento do túnel, de acordo com as descrições a seguir:

- A criação de uma estrutura de servidor remoto que é capaz de conectar diversos servidores de jogos ao mesmo, por meio de túneis;
- Quando um cliente se conecta ao servidor, ele está apto a se conectar a qualquer servidor de jogo disponível ao mesmo;
- Para se conectar a um servidor, deve ser enviado um comando solicitando conexão ao servidor de jogo (*login* <usuario>/<servidor de jogo>);
- Uma vez estabelecida a conexão ao servidor de jogo, todos os dados enviados pelo túnel serão encaminhados diretamente ao servidor de jogo.

A arquitetura proposta para a comunicação de rede para o módulo servidor é apresentada na Figura 5.17 e os componentes são descritos:

Figura 5.17 - Estrutura do Módulo Servidor

- *Server* - É o responsável por indicar o servidor remoto e ficar esperando as conexões remotas;
- *ConnectionManager* - É o responsável por identificar para qual servidor de jogo a conexão está sendo direcionada;
- *PluginManager* - É o responsável por adicionar/remover (gerenciar) servidores de jogos do servidor remoto;

Os eventos gerados pelo jogador na interface do jogo são capturados pelo módulo cliente e transferidos para o módulo servidor representado pela classe *ConnectionManager* que transfere o evento gerado para todos os participantes.

A classe *ConnectionManager* gerencia, além da troca de mensagens, a entrada e saída dos jogadores do sistema. Toda e qualquer ação realizada pelo jogador que passa pelo servidor que reflete a mesma para todos os jogadores.

A Figura 5.18 apresenta o diagrama de classes do módulo servidor do *framework* FMMG.

Figura 5.18 - Diagrama de Classe do Módulo Servidor

5.6 Conclusão do Capítulo

O *framework* FMMG foi projetado para ser utilizado na arquitetura de Jogos *Multiplayer* Móveis. É possível desenvolver jogos utilizando-se da sua reusabilidade e os benefícios da plataforma de desenvolvimento orientada a objetos. A utilização do padrão MVC e a estrutura de estados determinam a ação realizada pelo jogador durante o jogo. Além disso, o capítulo apresenta o mecanismo de comunicação cliente/servidor projetado no *framework* FMMG.

As vantagens do *framework* FMMG são:

- Modularidade do jogo;
- Não precisar se preocupar com o ambiente externo ao do jogo;
- Diversas funcionalidades implementadas no *framework*;
- Com uma estrutura de estados, cada estado terá associado a uma ação e uma *View*, facilitando ainda mais o trabalho do desenvolvedor;
- Facilidade em manter uma equipe no desenvolvimento do jogo, devido a modularidade, deste modo cada desenvolvedor faz a sua parte do jogo;

- Utilização de padrões consagrados de desenvolvimento, como o padrão MVC;
- Foco no jogo, não perde tempo com particularidades dos dispositivos móveis;
- Permite a implementação de Jogos *Multiplayer* Móveis;
- Facilidade de manutenção e de reuso;
- Facilidade para portar para dispositivos móveis com mais recursos.

As desvantagens do *framework* FMMG são apresentadas a seguir:

- Dificuldade para se manter compatibilidade com versões anteriores, já que *frameworks* se tornam mais maduros com o passar do tempo e as aplicações devem evoluir em paralelo;
- Depende da implementação do jogo, a flexibilidade e a generalização do *framework* podem trabalhar contra sua eficiência;
- Para o desenvolvedor iniciante e com os padrões utilizados, há uma curva de aprendizado alta (perda dos vícios da programação não modular/monolítica).

Capítulo 6 - Aplicabilidade do Framework FMMG: Estudo de Caso

Este capítulo apresenta o estudo de caso desenvolvido a partir do *framework* FMMG (*Framework for Mobile Multiplayer Games*) proposto nesta Tese descrito no capítulo 5.

A intenção é apresentar, em detalhes, cada um dos módulos, a estrutura do *framework* FMMG, sua validação, e como eles se integram de forma a constituir uma arquitetura a ser utilizada no desenvolvimento de Jogos *Multiplayer* Móveis.

Nas seções deste capítulo é apresentado a infraestrutura utilizada, o jogo Caça ao Tesouro em relação à arquitetura, os componentes e a estrutura de classes do *framework* FMMG.

6.1 Infraestrutura Utilizada

Esta seção apresenta plataforma BREW (*Binary Runtime Environment for Wireless*) (Qualcomm, 2006) como a principal tecnologia de desenvolvimento de jogos para aparelho celular e a infraestrutura de *software* e *hardware* utilizada no desenvolvimento do estudo de caso.

6.1.1 BREW

A Qualcomm (Qualcomm, 2006) desenvolveu a plataforma BREW, Ambiente de Desenvolvimento Binário para Dispositivos Móveis, que oferece uma solução completa e aberta para aplicações em dispositivos móveis. Neste ambiente, as aplicações são desenvolvidas nas linguagens de programação C/C++, compiladas e executam em um componente separado do processador principal do dispositivo, o que possibilita um desempenho melhor em relação a J2ME.

A plataforma BREW oferece uma API para acesso à várias funcionalidades dos dispositivos móveis, incluindo tela, rede, teclado, som, etc. O uso desta API ocorre no acesso às interfaces, que possuem um conjunto de funções associadas às funcionalidades. Estas interfaces possuem um identificador único e que implementa algum padrão de projeto, para gerenciar a alocação e desalocação de memória.

A plataforma BREW possui um razoável suporte para o desenvolvimento de jogos. É possível desenhar primitivas gráficas e imagens, mas não é permitido acessar os *pixels* da imagem nem realizar transformações como rotação e escala. Também tem suporte a som, criação de temporizadores, conectividade e persistência de dados.

A plataforma BREW não é simplesmente uma API ou um ambiente para execução, a proposta da Qualcomm foi estabelecer um completo processo para o desenvolvimento de aplicações para aparelhos celulares que possibilitasse criar um canal entre o desenvolvedor e as operadoras de telefonia, dando garantias mínimas de qualidade dos aplicativos construídos e do pagamento pela comercialização dos mesmos.

Baseado nesta proposta, os seguintes componentes podem ser listados para o ambiente BREW:

- API - Conjunto de funções que dão acesso aos recursos nativos do aparelho;
- Ferramentas de Compilação/Emulação - Aplicativos que rodam em máquinas com sistema operacional Windows e permitem o desenvolvimento das aplicações e a emulação destas num ambiente com vários recursos para depuração;
- Ferramentas de Teste - Aplicativos que simulam alguns testes realizados no procedimento *True BREW Test*;
- *True BREW Test* - Conjunto de procedimento de testes realizados por laboratórios certificados pela Qualcomm para garantir a qualidade da aplicação e dar segurança para as operadoras que os aplicativos disponibilizados a seus clientes não provocarão nenhum dano a seus aparelhos;
- *BREW Developer Extranet* - *Extranet* utilizada como canal de comunicação dos diversos envolvidos no processo (Qualcomm, desenvolvedores, fabricantes de aparelhos e operadoras).

A plataforma BREW como um ambiente de execução, permite que sejam desenvolvidas aplicações nas linguagens de programação C/C++ que são compiladas e executadas

como código nativo no aparelho celular. A API de BREW é rica em serviços associados ao dispositivo, permitindo acessar quase todas as suas funcionalidades desde tela e teclas até conexões via *socket*, agenda telefônica e mensagens SMS. Alguns serviços da API já foram desenvolvidos voltados para jogos como a implementação da interface de imagens, que permite a criação de animações utilizando uma só imagem, como a classe *Sprite* de J2ME MIDP 2.0.

É possível, ainda, desenvolver aplicações em outras linguagens que não sejam as linguagens de programação C/C++, bastando que estas rodem sobre uma máquina virtual. Desta forma, já existem iniciativas de Máquinas Virtuais Java e outros ambientes de execução (Python e XML) que rodem sobre BREW e permitam que desenvolvedores não familiarizados com as linguagens de programação C/C++ possam desenvolver suas aplicações para celulares neste ambiente.

Para o suporte de rede, BREW oferece a possibilidade de abrir conexões com *sockets* permitindo a criação de jogos baseados em arquiteturas cliente/servidor ou ponto a ponto. A plataforma BREW permite o acesso a *internet* através de conexões TCP ou UDP sobre IP, através de uma interface de *sockets* modelada como *Sockets*. Garantindo assim uma abertura muito maior dos jogos que podem ser desenvolvidos para esta plataforma.

Comparando com o acesso à rede oferecido por J2ME MIDP 2.0, BREW possui maior flexibilidade por permitir a criação de *sockets* e o controle deles diretamente. MIDP 2.0 especifica diversas formas de conexão, incluindo *sockets*, mas obriga os fabricantes a implementarem os protocolos HTTP e HTTPS.

6.1.2 Plataforma Utilizada

A plataforma de utilizada para o desenvolvimento do estudo de caso é composta das seguintes configurações:

Software: Microsoft Visual Studio .NET - Visual C++ .NET, BREW SDK versão 2.0.1 e BREW Emulator 2.0.

Hardware: Aparelhos Celulares Nokia 3152 e Nokia 6155i.

6.2 Estudo de Caso: Caça ao Tesouro

6.2.1 Enredo do Jogo

O Caça ao Tesouro é um jogo de estratégia que tem como objetivo o resgate de um tesouro escondido pelos Astecas em um labirinto. Os Astecas, em sua passagem pela Terra, deixaram um tesouro de valor incalculável escondido em um labirinto. Os caçadores de tesouros vindos de diversas partes do planeta serão reunidos em um labirinto e terão que lutar entre si para conseguir pegar o tesouro.

O Caça ao Tesouro é um jogo *multiplayer*, em que todos os jogadores possuem os mesmos objetivos. Os jogadores são representados pelos caçadores e serão colocados dentro do labirinto para lutar entre si e pelo tesouro. Deste jogo somente um sairá vitorioso e terá em suas mãos o tesouro mais valioso do planeta.

Cada caçador terá em seu poder uma arma capaz de imobilizar seus oponentes, por alguns segundos, o que poderá ser utilizado para facilitar seu acesso ao tesouro.

Caça ao Tesouro é apresentado na Figura 6.1.

Figura 6.1 - Caça ao Tesouro

6.2.2 Requisitos

Os requisitos do jogo Caça ao Tesouro possui as seguintes configurações:

- Níveis - O jogo terá somente um nível;
- Fases - O jogo terá somente uma fase, onde todos lutam contra todos;
- Controles do Jogador - O jogador pode se mover para esquerda, para direita, para cima e para baixo. As teclas utilizadas para realizar a movimentação são: 4 ou seta para esquerda, para se deslocar para a esquerda, 6 ou seta para a direita, para se deslocar para a direita, 2 ou seta para cima, para se deslocar para cima, 8 ou seta para baixo, para se deslocar para baixo, e 0 para disparar;
- Inimigos - Cada caçador que entrar no jogo será um inimigo. O caçador poderá utilizar sua arma de imobilização, e quando um inimigo estiver imobilizado o caçador poderá passar pelo seu inimigo sem ser afetado, caso contrário o caçador e o inimigo perdem o jogo;
- Ganhar - Ganha o jogo o caçador que pegar o tesouro;
- Perder - Somente um jogador pega o tesouro, e isso define quem perde o jogo.

6.2.3 Visualização

A seguir são apresentados os objetos que compõem o jogo Caça ao tesouro:

- Caçadores - O jogo inclui diversos caçadores, conforme a Figura 6.2;

Figura 6.2 - Caçadores

- Tesouro - O jogo inclui um tesouro, conforme a Figura 6.3;

Figura 6.3 - Tesouro

- Inimigo - O jogo inclui inimigos, que são os demais caçadores de tesouro do jogo. Eles serão representados pela mesma Figura 6.2 do caçador de tesouros, e o que os diferirá do caçador, que representa o jogador, é que eles aparecerão com um contorno em seu desenho;
- Cenário - O jogo incluirá um cenário, conforme a Figura 6.4.

Figura 6.4 – Cenário do Jogo

6.2.4 Configuração de Gráficos

A resolução do jogo é adaptável a cada aparelho de celular permitindo que a partida não mude de um aparelho celular para outro.

A profundidade de cores do jogo é de 8 *bits* (256 cores) para se adequar à capacidade dos aparelhos celulares (com relação ao tamanho dos arquivos de imagem).

O caçador de tesouros, inimigos e cenário que representam os objetos gráficos que farão parte do jogo e são representados por imagens 2D.

O cenário e o *background* do jogo têm um cenário baseado em um labirinto e usará técnica de *scrolling* vertical e horizontal para representar sua extensão.

6.2.5 Estrutura de Estado do Jogo

A estrutura de estados do jogo é composta dos seguintes itens:

- Menu Principal - Este menu terá opções para iniciar um novo jogo, parar de jogar, configurar o jogo e ver as pontuações mais altas, conforme a Figura 6.5. Este menu não ficará disponível durante a partida. O menu principal pode ser navegado usando as teclas direcionais padrão, as teclas 4, 6, 2, 8 e 0, a tecla de seleção ou a tecla 5 para selecionar itens;

Figura 6.5 - Menu Principal do Jogo

- Jogando - Este é o modo de jogo principal, veja na seção 6.2 requisitos do jogo para obter mais detalhes;
- Venceu o jogo - Todos os inimigos foram superados e o caçador de tesouros encontrou o tesouro;
- Fim do jogo - Um de seus inimigos encontrou o tesouro. É apresentando uma

tela indicando o final do jogo.

6.3 Arquitetura

Baseado no padrão MVC, a Figura 6.6 apresenta as classes do jogo Caça ao Tesouro. As classes *HunterView*, *HunterCtl*, *HunterAct* e *HunterDo* são classes próprias do jogo Caça ao Tesouro e herdam características das classes *View* e *Model* e se associam com as demais classes dos módulos de gerenciamento, básico e rede. As funcionalidades das classes do Caça ao Tesouro estão descritas a seguir:

Figura 6.6 - Diagrama de Classe do Jogo Caça Tesouro

- *HunterView* - Classe que define os objetos de entrada e saída do jogo;
- *HunterCtl* – Classe que controla a execução do jogo. A partir desta classe que são disparadas as funções das classes para a execução e exibição do jogo;
- *HunterAct* - Classe controla e executa os estados do jogo. Os possíveis estados são:
 - ✓ *STATE_WAITTING_PLAYERS* - Indica que o jogador está conectado ao servidor, porém está aguardando por outros jogadores;

- ✓ *STATE_GAME_PLAYING* - Indica que o jogo está em pleno funcionamento e o jogador deve buscar o tesouro;
- ✓ *STATE_GAME_WIN* - A partida terminou e o jogador foi o vencedor;
- ✓ *STATE_GAME_OVER* - A partida terminou e o jogador perdeu.
- *HunterDo* - Classe responsável pela execução e retorno das informações do jogo que foram recebidas/enviadas do servidor.

O jogo Caça ao Tesouro utiliza os seguintes estados, a partir do padrão MVC e estrutura de estados, denominado macro estado do jogo, conforme mostra o exemplo da Figura 6.7:

- *SplashScreenState* - Trata da abertura inicial de um jogo;
- *MenuState* - Trata da estrutura do menu do jogo;
- *GamePlayState* - Trata da execução do jogo;
- *RankState* - Trata da apresentação de uma lista do ranking de um determinado jogo.

Figura 6.7 - Exemplo da Utilização do Padrão MVC e Estrutura de Estados no Jogo Caça ao Tesouro

O estado *GamePlayState* também utiliza os seguintes estados a partir do padrão MVC e estrutura de estados, denominado micro estado do jogo, conforme mostra o exemplo da Figura 6.8:

Figura 6.8 - Exemplo da Utilização do Padrão MVC e Estrutura de Estados do *GamePlayState* do Jogo Caça ao Tesouro

As estruturas de estados representados nas Figuras 6.7 e 6.8 se integram com as classes dos módulos de gerenciamento, módulo básico e módulo de rede.

6.4 Comunicação de Rede

O módulo de comunicação de rede é baseado no modelo cliente/servidor do *framework* FMMG, conforme descrito na seção 5.5.

Para o jogo Caça ao Tesouro, o funcionamento do túnel de transmissão de dados pela rede é conforme as descrições a seguir:

- O cliente cria um túnel TCP ou UDP com um determinado servidor remoto por meio da classe *NetManager*, conforme a Figura 6.9;


```
void criarTunel( )
{
    NetManager* net = new NetManager( "www.fmmg.org.br", 5748 ) ;
    ...
}
```

Figura 6.9 - Criar Túnel TCP ou UDP

- Uma vez estabelecido o túnel, o cliente pode solicitar conexões com os servidores de jogos por meio do método *open* da classe *NetManager*, conforme a Figura 6.10;
- Quando uma conexão é criada, um objeto *socket* é retornado (*StreamSocket* para TCP ou *DatagramSocket* para UDP), conforme a Figura 6.10;

```
void conectarServidorDoJogo( NetManager* net )
{
    Socket* socket = net->open( "caca_tesouro" ) ; // cria conexão TCP com o servidor de jogo
                                                    "caca_tesouro"
    ...
}
```

Figura 6.10 - Cliente Solicita Conexão

- Por meio do objeto *socket* a comunicação é realizada, conforme a Figura 6.11 e a Figura 6.12.

```
void enviarComandoDoJogo( Socket* socket )
{
    const int    bufferSize = 512 ;
    char*        buffer     = new char[ bufferSize ] ;

    // enviar comando
    stdlib::strcpy( buffer, "move left" ) ;
    socket->write( buffer, stdlib::strlen( buffer ) ) ;
    ...
}
```

Figura 6.11 - Envia o Objeto *socket*

```

void receberComandoDoJogo( Socket* socket )
{
    const int    bufferSize = 512 ;
    char*        buffer     = new char[ bufferSize ] ;

    // receber resposta
    int read = socket->read( buffer, bufferSize ) ;
    buffer[ read ] = '\0' ;
    ...
}

```

Figura 6.12 - Recebe o Objeto *socket*

As estruturas de comunicação entre os componentes dependem da função a ser realizada pelo o cliente e o servidor do jogo Caça ao Tesouro.

A Figura 6.13 apresenta o diagrama macro do jogo Caça ao Tesouro, que é composta pelas estruturas *NetManager*, *Socket*, *Model*, *View*, *Controller* e *Framework FMMG*. Conforme descrito no capítulo 5.

Figura 6.13 – Diagrama Macro de Caso Caça ao Tesouro

A Figura 6.14 mostra o diagrama de sequência que modela a conexão com o servidor.

Figura 6.16 - Conexão com o Servidor do Jogo

A sequência de passos que ocorre no diagrama de sequência de conexão com o servidor é:

1. *event* - O ambiente operacional *BREW* envia um evento ao *Framework*;
2. *process* – O *Framework* processa o evento;
3. *run* - O *Framework* solicita ao *Controller* que execute, pois um evento ocorreu no jogo;
4. *new* - O *Controller* cria um *NetManager* que gerenciará um túnel TCP ou UDP;
5. *connect* - O *NetManager* conecta-se ao servidor remoto, criando um túnel TCP ou UDP;
6. **netManager* - O objeto do tipo *NetManager* é retornado ao *Controller*;
7. *run* - O *Controller* solicita a *Model* que execute;
8. *open* - A *Model* solicita ao *NetManager* que uma conexão com o servidor de

jogo seja criada;

9. *new* - O *NetManager* cria um novo *Socket* (TCP ou UDP) que será utilizado para encapsular a conexão com o servidor do jogo;
10. *open* – O *socket* criado solicita ao servidor remoto uma conexão com o servidor do jogo (*login*);
11. **socket* - O *socket* é retornado à *Model*, que o utilizará para a comunicação com o servidor do jogo.

A Figura 6.15 mostra o diagrama de seqüência que modela a comunicação com o servidor.

Figura 6.15 – Comunicação com o Servidor do Jogo

A seqüência de passos que ocorre no diagrama de seqüência de comunicação com o servidor é:

1. *event* - O ambiente operacional *BREW* envia um evento ao *Framework*;
2. *process* - O *Framework* processa o evento;
3. *run* - O *Framework* solicita ao *Controller* que execute, pois um evento ocorreu no jogo;
4. *run* - O *Controller* solicita a *Model* que execute;
5. *process* - A *Model* processa o evento atualiza informações sobre seu estado;
6. *write* - A *Model* grava suas informações atualizadas no *socket*;
7. *write* - O *Socket* envia as informações ao servidor do jogo;
8. *read* - A *Model* lê a resposta do servidor através do *socket*;
9. *read* - O *Socket* lê a resposta enviada pelo servidor do jogo;
10. **data* - Os dados da resposta são disponibilizados à *Model*;
11. *update* - A *Model* atualiza seu estados com as informações enviadas pelo servidor do jogo;
12. *draw* - O *Controller* solicita a *View* que a tela, representando o estado atual, seja desenhada.

A Figura 6.16 apresenta o diagrama de seqüência da função *open* executado do cliente do jogo Caça ao Tesouro.

Figura 6.16 - Função *open*

A sequência de passos que ocorre no diagrama de sequência da função *open* é:

1. *open* - A aplicação cliente (jogo) solicita acesso ao servidor de jogos;
2. *createStreamSocket* - O *NetManager* solicita à *SocketFactory* a criação de um *socket* TCP;
3. *new* - A *SocketFactory* cria um novo *socket* TCP;
4. **socket* - Um objeto do tipo *StreamSocket* é retornado a *Factory*;
5. **socket* - O *socket* é retornado ao *NetManager*;
6. *registerSocket* - O *NetManager* registra a *socket* no *ConnectionManager*;
7. *multiplex* - O *ConnectionManager* encapsula um pedido de conexão ao servidor de jogos;
8. *write* - O *ConnectionManager* solicita ao *NetManager* que seus dados sejam enviados ao servidor remoto (pedido de conexão ao servidor do jogo);

9. *multiplex* - O *NetManager* encapsula os dados que serão enviados ao servidor remoto;
10. *write* - O *NetManager* envia os dados, através do túnel TCP previamente criado, ao servidor remoto;
11. **socket* - O *socket* é retornado a aplicação cliente (jogo);

A Figura 6.17 apresenta o diagrama de seqüência da função *write* executado do cliente do jogo Caça ao Tesouro.

Figura 6.17 - Função *write*

A seqüência de passos que ocorre no diagrama de seqüência da função *write* é:

1. *write* - A aplicação cliente (jogo) solicita o envio de dados ao servidor de jogos;
2. *write* - O *StreamSocket* (*socket* TCP) solicita ao *ConnectionManager* o envio dos dados ao servidor de jogos;
3. *multiplex* - O *ConnectionManager* encapsula os dados a serem enviados ao servidor de jogos;
4. *write* - O *ConnectionManager* solicita ao *NetManager* que seus dados sejam enviados ao servidor remoto (envio de dados ao servidor do jogo);

5. *multiplex* - O *NetManager* encapsula os dados que serão enviados ao servidor remoto;
6. *write* - O *NetManager* envia os dados, através do túnel TCP previamente criado, ao servidor remoto.

A Figura 6.18 apresenta o diagrama de seqüência da função *read* executado do cliente do jogo Caça ao Tesouro.

Figura 6.18 - Função *read*

A seqüência de passos que ocorre no diagrama de seqüência da função *read* é:

1. *read* - A aplicação cliente (jogo) solicita o recebimento de dados do servidor de jogos;
2. *read* - O *StreamSocket* (*socket* TCP) solicita ao *ConnectionManager* o recebimento dos dados enviados pelo servidor de jogos;
3. *read* - O *ConnectionManager* solicita ao *NetManager* o recebimento dos dados enviados pelo servidor de jogos;

4. *read* - O *NetManager* solicita o recebimento dos dados, através do túnel TCP previamente criado, enviados pelo servidor remoto;
5. **data* - Os dados são recebidos pelo *NetManager*;
6. *demultiplex* - Os dados são desencapsulados pelo *NetManager*;
7. **data* - Os dados são passados ao *ConnectionManager*;
8. *demultiplex* - Os dados são desencapsulados pelo *ConnectionManager*;
9. **data* - Os dados são passados ao *StreamSocket*;
10. **data* - Os dados são disponibilizados a aplicação cliente (jogo).

A Figura 6.19 apresenta o diagrama de seqüência da função *close* executado do cliente do jogo Caça ao Tesouro.

Figura 6.19 - Função *close*

A seqüência de passos que ocorre no diagrama de seqüência da função *close* é:

1. *close* - A aplicação cliente (jogo) solicita o fechamento da conexão com o servidor de jogos;
2. *unRegisterSocket* - O *StreamSocket* (*socket* TCP) informa ao

- ConnectionManager* que a conexão não será mais utilizada;
3. *multiplex* - O *ConnectionManager* encapsula um pedido de fechamento da conexão ao servidor de jogos;
 4. *write* - O *ConnectionManager* solicita ao *NetManager* que seus dados sejam enviados ao servidor remoto (pedido de fechamento da conexão ao servidor do jogo);
 5. *multiplex* - O *NetManager* encapsula os dados que serão enviados ao servidor remoto;
 6. *write* - O *NetManager* envia os dados, através do túnel TCP previamente criado, ao servidor remoto.

A Figura 6.20 apresenta o diagrama de seqüência da função *login* executado do servidor do jogo Caça ao Tesouro.

Figura 6.20 - Função *login*

A seqüência de passos que ocorre no diagrama de seqüência da função *login* é:

1. *login* - O servidor remoto recebe uma solicitação de conexão (*login*), através do túnel TCP ou UDP previamente estabelecido, ao servidor do jogo;
2. *demultiplex* - O servidor remoto desencapsula os dados da solicitação de conexão;
3. *login* - O servidor repassa a solicitação de *login* ao *ConnectionManager*;
4. *demultiplex* - O *ConnectionManager* desencapsula a solicitação de *login*;
5. *get* - O *ConnectionManager* solicita ao *PluginManager* (Servidor de Jogos) uma referência ao servidor do jogo em questão (Caça ao Tesouro);
6. *new* - O *PluginManager* criar uma instância do jogo Caça ao Tesouro que atenderá àquela conexão;
7. **caca_tesouro* - O *PluginManager* devolve a referência do servidor do jogo;
8. *sessionAttach* - O *ConnectionManager* adiciona à sessão da conexão o servidor do jogo que deverá responder às solicitações;
9. *multiplex* - O *ConnectionManager* encapsula uma resposta à solicitação de conexão (*login*);
10. *ok* - A resposta é encaminhada ao servidor remoto;
11. *multiplex* - O servidor remoto encapsula os dados a serem enviados pelo túnel TCP ou UDP;
12. *ok* - A resposta é enviada através do túnel.

A Figura 6.21 apresenta o diagrama de seqüência da função *execute* executado do servidor do jogo Caça ao Tesouro.

Figura 6.21 - Função *execute*

A sequência de passos que ocorre no diagrama de sequência da função *execute* é:

1. *execute* - O servidor remoto recebe uma solicitação de execução de um comando no servidor do jogo;
2. *demultiplex* - O servidor remoto desencapsula os dados da solicitação;
3. *execute* - O servidor repassa a solicitação ao *ConnectionManager*;
4. *demultiplex* - O *ConnectionManager* desencapsula a solicitação;
5. *execute* - O *ConnectionManager* encaminha a solicitação ao servidor do jogo (Caça ao Tesouro);
6. **result* - Uma resposta ao comando executado é devolvida ao *ConnectionManager*;

7. *notifyAll* - O *ConnectionManager* notifica todos os jogadores participantes da mesma sessão de jogo;
8. *multiplex* - O *ConnectionManager* encapsula a resposta enviada pelo servidor do jogo;
9. **result* - A resposta é encaminhada ao servidor remoto;
10. *multiplex* - O servidor remoto encapsula os dados a serem enviados pelo túnel TCP ou UDP;
11. **result* - A resposta é enviada através do túnel.

A Figura 6.22 apresenta o diagrama de seqüência da função *logout* executado do servidor do jogo Caça ao Tesouro.

Figura 6.22 - Função *logout*

A seqüência de passos que ocorre no diagrama de seqüência da função *logout* é:

1. *logout* - O servidor remoto recebe uma solicitação de desconexão (*logout*), através do túnel TCP ou UDP previamente estabelecido, ao servidor do jogo;
2. *demultiplex* - O servidor remoto desencapsula os dados da solicitação de desconexão;
3. *logout* - O servidor repassa a solicitação de *logout* ao *ConnectionManager*;
4. *demultiplex* - O *ConnectionManager* desencapsula a solicitação de *logout*;
5. *sessionDeAttach* - O *ConnectionManager* remove da sessão da conexão o servidor do jogo que deverá responder às solicitações;
6. *release* - O *ConnectionManager* solicita ao *PluginManager* (Servidor de Jogos) que a referência ao servidor do jogo em questão (Caça ao Tesouro) seja liberada;
7. *release* - O *PluginManager* libera a instância do jogo Caça ao Tesouro que atendia àquela conexão;
8. *multiplex* - O *ConnectionManager* encapsula uma resposta à solicitação de desconexão (*logout*);
9. *ok* - A resposta é encaminhada ao servidor remoto;
10. *multiplex* - O servidor remoto encapsula os dados a serem enviados pelo túnel TCP ou UDP;
11. *ok* - A resposta é enviada através do túnel.

6.5 Conclusão do Capítulo

Este capítulo apresentou as principais características relacionadas aos aspectos de implementação e funcionamento do *framework* FMMG, por meio do estudo de caso do jogo Caça ao Tesouro.

Inicialmente é apresentada a infraestrutura utilizada na implementação, posteriormente foi detalhado o jogo Caça ao Tesouro, e por fim é mostrada a utilização do *framework* FMMG para sua validação no desenvolvimento de Jogos *Multiplayer* Móveis.

Capítulo 7 - Conclusão

7.1 Considerações Iniciais

Neste trabalho foram apresentadas, inicialmente as principais vertentes dos conceitos e definições encontrados na literatura relacionados a jogos, *engine*, jogos *multiplayer*, *framework* e trabalhos relacionados. Em seguida, foi apresentado o *framework* FMMG com seus requisitos, arquitetura e funcionalidades, posteriormente foi demonstrada a utilização do *framework* em uma implementação de um estudo de caso.

Algumas limitações tecnológicas, que foram aqui discutidas, impedem o sucesso e o alcance das expectativas que vêm sendo colocadas sobre a tecnologia de dispositivos móveis. Estas limitações incluem desde limitação de memória, capacidade de processamento, tamanho da tela e mecanismo de entrada não apropriado e melhor resolução de imagem. Mais ainda, a diminuição do atraso do tempo entre uma ação do jogador e a visualização do reflexo desta no jogo, reduzida para poucos milissegundos.

Dentro do contexto acima, a Tese aqui apresentada teve como resultado principal o *framework* FMMG que torna os jogos produzidos a partir do mesmo orientados à lógica de um determinado domínio, assim garantindo maior reusabilidade. Com isso, o desenvolvedor do jogo deverá preocupar-se com a mecânica do jogo e com os elementos de interação, tais como: gráficos, sons e música de cada plataforma que o jogo irá executar. Pois se espera com o *framework* FMMG tornar o processo de projeto e programação de jogos uma questão de escolha de componentes e integração dos mesmos.

Em resumo, a arquitetura do *framework* FMMG permite:

- Desenvolver jogos utilizando diferentes formas de manipulação;
- Utilizar a estrutura geral do *framework* FMMG, a partir do padrão MVC e estrutura de estados;
- Separar a lógica do jogo de sua forma de visualização e de seus dispositivos de interação.

As principais funcionalidades oferecidas pelo *framework* FMMG são:

- Apresentação de imagens do jogo, utilizando recursos de visualização;
- Fornecer um conjunto de classes básicas para facilitar o desenvolvimento de jogos;
- Controle de eventos gerados por diferentes dispositivos de entrada utilizados pelos jogadores;
- Criação de jogos em rede usando uma arquitetura cliente/servidor.

É importante notar a diversidade de cada um dos componentes dos módulos, pois podem existir inúmeros tipos de estruturas de comunicação, periféricos de entrada, dispositivos de áudio e de vídeo, entre outros. Por esta razão, o *framework* FMMG propõe-se ser genérico para permitir a integração destes componentes e criar um jogo independente de suas implementações específicas. Procurou-se adotar um processo de modelagem de *software* orientado às boas práticas de programação orientadas a objetos.

Pelo fato do FMMG ser um *framework* a ser livremente distribuído, existe a facilidade de seu constante aperfeiçoamento e o surgimento de novas extensões que poderão ser reutilizadas em diferentes aplicações de jogos. O estudo de caso, mesmo sendo um jogo bastante simples, serve para ilustrar a flexibilidade no desenvolvimento de aplicações usando o *framework* proposto.

Entre outros motivos de utilização do *framework* FMMG, encontram-se:

- Extenso uso do *framework* torna possível implementar os requisitos de forma mais rápida focando na lógica do jogo, minimizando os problemas de desenvolvimento, aumentando a confiabilidade e robustez;
- A utilização de estrutura de estados facilita a especificação e o gerenciamento das várias etapas de um jogo.

Espera-se com essa Tese uma contribuição substancial para o avanço da pesquisa na computação, e uma transferência rápida desses benefícios para a sociedade, com

reflexos altamente positivos em áreas como: educação, treinamento, pesquisa, negócios e em muitas outras atividades do cotidiano das pessoas.

Seguem na Tabela 1, as características do *framework* FMMG em relação aos *frameworks* e *engines* encontrados na literatura.

Frameworks	Classificações		
	Visibilidade	Aplicabilidade	Domínio da Aplicação
FMMG	White Box	Framework Vertical	Framework para Ambientes
Amphibian	Black Box	Framework Vertical	Framework para Interfaces com o Usuário Framework para Ambientes
Forge V8	Black Box	Framework Horizontal	Framework para Interfaces com o Usuário Framework para Ambientes
Forge 16V	Black Box	Framework Vertical	Framework para Interfaces com o Usuário Framework para Ambientes
FreeMMG	Black Box	Framework Vertical	Framework para Ambientes
Fungen	Black Box	Framework Vertical	Framework para Ambientes
Guff	Black Box	Framework Vertical	Framework para Ambientes
wGEM	Black Box	Framework Vertical	Framework para Interfaces com o Usuário Framework para Ambientes

Frameworks	Classificações			
	Forma de Desenvolvimento	Grupos	Portabilidade	Plataforma
FMMG	Frameworks Orientados a Objetos	Frameworks de Infra-estrutura do Sistema	Dispositivos Móveis	Multiusuário
Amphibian	Frameworks Orientados a Objetos	Frameworks de Infra-estrutura do Sistema	PC Dispositivos Móveis	Multiusuário
Forge V8	Frameworks Orientados a Objetos	Frameworks de Infra-estrutura do Sistema	Várias	Monousuário
Forge 16V	Frameworks Procedurais	Frameworks de Infra-estrutura do Sistema	PC	Monousuário
FreeMMG	Frameworks Orientados a Objetos	Frameworks de Infra-estrutura do Sistema	Dispositivos Móveis	Multiusuário
Fungen	Frameworks Procedurais	Frameworks de Infra-estrutura do Sistema	PC	Monousuário
Guff	Frameworks Procedurais	Frameworks de Infra-estrutura do Sistema	PC	Monousuário
wGEM	Frameworks Orientados a Objetos	Frameworks de Infra-estrutura do Sistema	Dispositivos Móveis	Monousuário

Tabela 1 – Características dos *Frameworks* e *engines*

7.2 Contribuições

As contribuições relacionadas ao trabalho proposto são resumidas nos seguintes itens:

- A contribuição principal refere-se ao estudo e a elaboração de um *framework* FMMG para Jogos *Multiplayer* Móveis baseado no padrão de desenvolvimento MVC e a estrutura de estados;
- O próprio estudo de caso implementado consiste em uma contribuição, por serem totalmente funcional, sendo possível utilizá-lo efetivamente como aplicação de entretenimento.

7.3 Perspectivas e Trabalhos Futuros

A seguir são listadas as pesquisas que poderão dar continuidade a este trabalho. Estas pesquisas referem-se a aperfeiçoamentos do nível de utilização do *framework* FMMG:

- Avaliar outras formas de distribuição e armazenamento dos dados nos modelos de suporte de comunicação;
- Analisar e validar o desempenho de rede das aplicações desenvolvidas a partir do *framework* FMMG;
- Desenvolver um *engine* para integrar com o *framework* ou integrá-lo ao enJine do INTERLAB;
- A proposta mais trivial de melhoria para o *framework* FMMG é a incorporação de outros módulos (como Inteligência Artificial) e refinamento daqueles já existentes;
- A proposta de incorporar um módulo de *plugins*. O objetivo desse tipo de módulo é permitir que novas funcionalidades sejam acrescentadas ao *framework* de maneira independente e simplificado;
- Realizar comparações do *framework* FMMG em ambientes de PC's e Dispositivos Móveis;

- Aplicabilidade do *framework* FMMG para jogos de educação e demais áreas;
- Complementar os modelos de suporte de comunicação para atender as aplicações de jogos de larga escala (milhares de usuários pertencentes ao jogo), denominados Jogos *Online* Massivamente Multiusuário para Dispositivos Móveis (MMMOG) e Jogos Massivos Ubíquos (MMUG).

7.3 Publicação de Artigos em Eventos Nacionais e Internacionais

No decorrer do desenvolvimento deste trabalho, foram publicados alguns artigos em eventos nacionais e internacionais, que são relacionados abaixo:

- M. M. Kubo, R. Tori, “FMMG: A Framework for Mobile Multiplayer Games”, 8th International Computer Games Conference on the theme: AI, Mobile, Educational and Serious Games, IEEE, Louisville, Kentucky, USA, 2006;
- M. M. Kubo, B. S. Meiguins, M. B. Garcia, L. A. Oliveira, R. Tori, “Aplicações de Ambientes Virtuais Colaborativos”, Livro Realidade Virtual - Conceitos e Tendências. 1 ed. São Paulo, 2004;
- L. Rodrigues, M. M. Kubo, I. A. Rodello, A. C. Sementille, R. Tori, J. Remo, “Ambientes Virtuais Distribuídos”, Livro Realidade Virtual - Conceitos e Tendências. 1 ed. São Paulo, 2004;
- M. M. Kubo, R. Tori, C. Kirner, “Interaction in Educational Collaborative Virtual Environments”, Revista de Engenharia de Computação e Sistemas Digitais, 2003;
- M. M. Kubo, R. Tori, C. Kirner, “Interaction in Educational Collaborative Virtual Environments”, Cyberpsychology and Behavior Journal - VR in South America, 2002;
- I. A. Rodello, M. M. Kubo, C. Kirner, R. Tori, “Análise dos Principais Fatores para o Desenvolvimento de Sistemas Distribuídos de Realidade Virtual”, IV Simpósio Brasileiro de Realidade Virtual – SRV’01, Florianópolis, 2001.

Referências Bibliográficas

Azevedo, E. - *“Desenvolvimento de Jogos 3D e Aplicações em Realidade Virtual”*, Editora Elsevier, 2005.

Battaiola, A. L. - *“Jogos por Computador - Histórico Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação”*, Anais da XIX Jornada de Atualização em Informática (JAI) - V. 2, pp. 83-122, SBC, 2000.

Battaiola, A. L., Domingues, R. G., Feijo, B., Swareman, D., Clua, E. W. G., Kosovitz, L. E., Dreux, M., Pessoa, C. A., Ramalho, G. - *“Desenvolvimento de Jogos em Computadores e Celulares”*, RITA, V. 3, N. 2, 2001.

Benford, J. B., Fahlén, L. E., Mariani, J., Rodden, T. - *“Supporting Cooperative Work in Virtual Environments”*, The Computer Journal, V. 37, N. 8, pp. 653-668, 1994.

Bernades, J. L., Bianchini, R., Cuzziot, M., Jacober, E., Nakamura, R., Tori, Rmoro - *“Jogos Eletrônicos e Realidade Virtual”*, Livro do Pré-Simpósio, VII Symposium on Virtual Reality, pp. 159-176, São Paulo, 2004.

Bethke, E. - *“Game Development and Production”*, Wordware Publishing, 2003.

Bianchini, R. C. - *“Uma Arquitetura BDI para Comportamentos Interativos de Agentes em Jogos Computacionais”*, Tese de Doutorado, USP, São Paulo, 2005.

Bittencourt, J. R. - *“Um Framework para Criação de Jogos Computadorizados Multiplataforma”*, Dissertação de Mestrado, PUCRS, Porto Alegre, 2004.

Borges, R. A. - *“Especificação e Análise de um Sistema Multiusuários de Realidade Virtual”*, Tese de Doutorado, USP, São Paulo, 1996.

Brooks, F. P. - *“No Silver Bullet: Essence and Accidents of Software Engineering”*, Computer Magazine, 20(4), pp. 10-19, 1987.

Cecin, F. R., Barbosa, J. L. V. , Geyer, C. F. R. - *“FreeMMG: Uma Proposta Baseada em Peer-to-Peer para Simulação Interativa Competitiva em Tempo Real e Distribuída”*

em Grande Escala”, Primeiro Workshop do Grupo de Processamento Paralelo e Distribuído, UFRGS, Porto Alegre, 2003.

Corley, S. - “*Architecting a 3D Animation Engine*”, Game Developer, Abril, 1998.

Deriggi Jr., F. V. - “*Suporte de Comunicação para Sistemas Multiusuários de Realidade Virtual*”, Dissertação de Mestrado, UFSCar, São Carlos, 1998.

Dwango - “*Samurai Romanesque*”, 2005, Disponível em: <www.dwango.co.jp>.

EnJine - “*Engine para Jogos 3D Multiusuários em Java*”, 2006, Disponível em <<http://incubadora.fapesp.br/projects/enjine/>>.

Favaro, D., Yoshino, F. L. D., Santos, F. M., Moretti, R. H. - “*F2DR - Jogo 3D para PDA e Sartphone*”, Projeto de Graduação, Escola Politécnica de São Paulo - USP, São Paulo, 2003.

Fayad, M.E., Schmidt, D.C. - “*Object-Oriented Application Frameworks*”, Communications of the ACM, V. 40, N. 10, p. 32-38, 1997.

Firesmith, D. G. - “*Frameworks: the Golden Path to Object Nirvana*”, Journal of Object-Oriented Programming, p. 5-8, 1993.

Fishwick, P. A. - “*Simulation Model Design and Execution: Building Digital Worlds*”, Prentice-Hall, Englewood Cliffs, 1995.

Furtado, A. W. B., Santos, A. L. M. - “*FunGEn - A Game Engine for Haskell*”, Workshop Brasileiro de Jogos e Entretenimento Digital, WJogos 2002, Ceará, 2002.

Gamma, E., et. al. - “*Design Patterns - Elements of Reusable Object-Oriented Software*”, Addison-Wesley, USA, 1995.

Gast, M. S. - “*8021.11 Wireless networks - The Definitive Guide*”, O’Reilly, 2002.

Gossweiler, R., Laferriere, R. J., Keller, M. L., Pausch, R. - “*An Introductory Tutorial for Developing Multi-User Virtual Environments*”, Presence, V. 3, N. 4, pp. 255-264, 1994.

Hagsand, O. - *"Interactive Multiuser Vessels in the DIVE System"*, IEEE Multimedia, Spring, V. 3, N. 1, pp. 30-39, 1996.

Johnson R., Foote, B. - *"Designing Reusable Classes"*, Journal of Object-Oriented Programming, p. 22-35, 1988.

Johnson, R. - *"How to Design Frameworks"*, Tutorial Notes of Conference on Object-Oriented Programming, System and Languages - OOPSLA'93, 1993.

Johnson, R., - *"How to Develop Frameworks"*, Tutorial Notes of ECOOP 96, Austria, 1996.

Kirner, C., Pinho, M. - *"Introdução à Realidade Virtual"*, Minicurso JAI/SBC, Recife, PE, 1996.

Krasner, E. K., Pope, S.T. - *"A Cookbook for using the Model-View- Controller User Interface Paradigm in Smalltalk-80"*, Journal of Object Oriented Programming, p. 26-49, 1988.

Kubo, M. - *"Suporte de Comunicação para Sistemas Interoperáveis de Realidade Virtual"*, Dissertação de Mestrado, UFSCar, São Carlos, 2000.

LaMothe, A., Ratcliff, J., Seminatore, M., Tyler, D. - *"Tricks of the Game Programming Gurus"*, Sams Publishing, 1994.

Latham, R. W. - *"If VR Is So Great Why Are VR Entertainment Systems So Poor?"*, Computer Graphics, V. 28, N. 2, pp. 113-114, 1994.

Lewis, M., Jacobson, J. - *"Game Engines in Scientific Research"*, Communications of ACM, Vol. 45 nº 1, 2002.

Locke, J. - *"An Introduction to the Internet Networking Environment and SIMNET/DIS"*, The Physical Design of the Internet, 1995.

Macedonia, M. R., Zyda, M. J. - *"A Taxonomy for Networked Virtual Environments"*, IEEE Multimedia, V. 4, N. 1, pp. 48-56, 1997.

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Brutzman, D. P., Barham, P. T. - *"Exploiting Reality With Multicast Groups"*, IEEE Computer Graphics and

Applications, V. 15, N. 9, pp. 38-45, 1995.

Madeira, C. A. G. - *“FORGE V8: Um Framework para o Desenvolvimento de Jogos de Computador e Aplicações Multimídia”*, Dissertação de Mestrado, UPFE, 2001.

Madeira, C., Araújo, A., Macedo, H., Andrade, R., Cavalcanti, D., Ferraz, F., Ramalho, G. - *“NetMaze: um Jogo de Ação Multimídia Distribuído”*, Simpósio Brasileiro de Sistemas Multimídia e Hiperídia, pp. 129-139, 2000.

Maldonado, J. C., Braga, R. T. V., Germano, F. S. R., Masiero, P. C. - *“Padrões e Frameworks de Software”*, Notas Didáticas, 2005.

Meiguins, B. S. - *“Arquitetura para Suporte à Comunicação de Aplicações de Realidade Virtual Colaborativas”*, Tese de Doutorado, UFPA, Belém, 2003.

Menezes, J. R. B. - *“Um framework para Criação de Jogos Computadorizados Multiplataforma”*, Dissertação de Mestrado, PUC-RS, Porto Alegre, 2003.

Miller, B. A., Bisdikuan, C. - *“Bluetooth Revealed: The insider’s guide to an open specification for global wireless communications”*, 2nd edition, Prentice Hall PTR, 2002.

Nakamura, R., Tori, R., Bernardes, J. L., Bianchini, R., Jacober, E. - *“A Practical Study on the Usage of a Commercial Game Engine for the Development of Educational Games”*, Workshop Brasileiro de Jogos e Entretenimento Digital - WJogos 2003, Salvador, 2003..

NCSOFT - *“Lineage”*, 2005, Disponível em: <<http://www.lineage.com/>>.

Orfaly, E. - *“The Essential Distributed Objects Survival Guide”*, Object Frameworks: An Overview, pp. 221-238, John Wiley & Sons, 1995.

Origin - *“Ultima Online”*, 2005, Disponível em: <<http://www.owo.com/>>.

Pantel, L., Wolf, L. - *“On the Impact of Delay on Real-Time Multiplayer Games”*, 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video, ACM Press, 2002.

Pessoa, A.C., Ramalha, G.L., Battaiola, A.L. - *“wGEM: Um Framework de Desenvolvimento de Jogos para Dispositivos Móveis”*, XXIX Seminário Integrado de Software e Hardware, Sociedade Brasileira de Computação, 2002.

Pessoa, C. A. C. - *“wGEM: Um Framework de Desenvolvimento de Jogos para Dispositivos Móveis”*, Dissertação de Mestrado, UFPE, 2001.

Pires, F. - *“OLAP MOLAP ROLAP e Frameworks Orientados a Objetos”*, Informativo técnico N. 52, Revista de Informação e Tecnologia, Unicamp, 1999.

Pree, W. - *“Design Patterns for Object-Oriented Software Development”*, Addison Wesley Publishing Company, 1994.

Pressman, R. S. - *“Engenharia de Software”*, Makron Books, São Paulo, 1995.

Qualcomm - *“Qualcomm BREW”*, 2006, Disponível em: <http://brew.qualcomm.com/>.

Ralph, E. J., Foote, B. - *“Designing Reusable Classes”*, Journal of Object-Oriented Programming, p. 22-35, 1988.

Rocha, E. J. T. S. - *“FORGE 16V: Um Framework para Desenvolvimento de Jogos Isométricos”*, Dissertação de Mestrado, Centro de Informática, Universidade Federal de Pernambuco, PE, 2003.

Rodello, I. A. - *“Um Ambiente Virtual Distribuído para Análise Molecular”*, Tese de Doutorado, USP, São Carlos, 2003.

Rogers, G. F. - *“Framework - Based Software Development in C++”*, Printice Hall, New Jersey, 1997.

Sementille, A. C. - *“A Utilização da Arquitetura CORBA na Construção de Ambientes Virtuais Distribuídos”*, Tese de Doutorado, USP, São Carlos, 1999.

Silva, R. P. - *“Suporte ao Desenvolvimento e uso de Frameworks e Componentes”*. Dissertação de Mestrado, UFRGS, Porto Alegre, 2000.

Singhal, S. K., Zyda, M. - *“Networked Virtual Environment - Design and Implementation”*, Addison Wesley, 1999.

Siscoutto, R. A. - *“Proposta de Arquitetura para Teleconferência Baseada na Integração de Video Avatar Estereoscópico em Ambiente Virtual Tridimensional”*, Tese de Doutorado, USP, São Paulo, 2003.

Sobral, D. - *“Framework para Comércio Eletrônico, Via Internet Móvel, Mediado por Agente de Software”*, Dissertação de Mestrado, UFSCar, São Carlos, 2002.

SOFTEX - *“Tecnologias de Visualização na Indústria de Jogos Digitais: Potencial Econômico e Tecnológico para a Indústria Brasileira de Software / Softex”*, Campinas, Softex, 2005, Disponível em: <<http://www.softex.br/>>.

Sony - *“EverQuest”*, 2005, Disponível em: <<http://www.everquest.com/>>.

Stytz, M. R. - *“Distributed Virtual Environments”*, IEEE Computer Graphics & Applications, pp. 19-31, 1996.

Sun - *“Java 2 Platform, Micro Edition (J2ME)”*, 2006, Disponível em <<http://java.sun.com/j2me/>>.

Tanenbaum, A. S. - *“Distributed Operating Systems”*, Prentice-Hall, 1995.

Tanenbaum, A. S. - *“Redes de Computadores”*, 4ª Edição, Campus, 2003.

Tibia Micro Edition - *“TibiaME”*, 2005, Disponível em: <<http://www.tibiame.com/>>.

Tori, T., Nakamura, R - *“Desenvolvimento de Jogos para o Aprendizado de Java 3D: Um estudo de Caso”*, Workshop Brasileiro de Jogos e Entretenimento Digital, WJogos 2002, Ceará, 2002.

Trinta, F., Ferraz, C., Ramalho, G. - *“Incorporando Mobilidade em Jogos Massivamente Multi-Usuários”*. Anais do WebMedia, Ribeirão Preto, 2005.

Trinta, F., Ferraz, C., Ramalho, G. - *“Serviços de Middleware para Jogos Massivos Ubíquos”*, Workshop Brasileiro de Jogos e Entretenimento Digital, WJogos 2005, São

Paulo, 2005.

Valente, L - "*Guff: Um Sistema para Desenvolvimento de Jogos*", Dissertação de Mestrado, Universidade Federal Fluminense, Rio de Janeiro, 2005.

Van Dam, A. - "*VR as a Forcing Function: Software Implications of new Paradigm*", IEEE'93 Symposium on Research Frontiers in Virtual Reality, San Jose, CA, 1993.

Whetten, B., Kaplan, S. - "*A High Performance Totally Ordered Multicast Protocol*", SIGCOMM'94, 1994.

Wloka, M. M. - "*Lag in Multiprocessor Virtual Reality*", Presence, V. 4, N.1, pp. 50-63, Winter, 1995.