DANIEL OLIVA SALES

PROJETO E IMPLEMENTAÇÃO DE JOGOS ELETRÔNICOS

Monografia de conclusão apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Ciência da Computação, para a obtenção do título de Bacharel em Ciência da Computação.

LAVRAS
MINAS GERAIS – BRASIL
2008

DANIEL OLIVA SALES

PROJETO E IMPLEMENTAÇÃO DE JOGOS ELETRÔNICOS

Monografia de conclusão apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Ciência da Computação, para a obtenção do título de Bacharel em Ciência da Computação.

Área de Concentração:
Desenvolvimento de Jogos Eletrônicos

Orientador:
Prof. Dsc. Wilian Soares Lacerda

Co-Orientador:
Prof. Msc. Cristiano Leite de Castro

LAVRAS
MINAS GERAIS – BRASIL
2008

Ficha Catalográfica preparada pela Divisão de Processos Técnicos
da Biblioteca Central da UFLA

Sales, Daniel Oliva

Projeto e Implementação de Jogos Eletrônicos /Daniel Oliva Sales. Lavras – Minas Gerais, 2008. (64)p : il.

Monografia de Graduação – Universidade Federal de Lavras. Departamento de Ciência da Computação.

1. Desenvolvimento de Jogos. 2. Computação Gráfica. 3. Inteligência Artificial. I. SALES, D.O. II. Universidade Federal de Lavras. III. Projeto e Implementação de Jogos Eletrônicos.

DANIEL OLIVA SALES

PROJETO E IMPLEMENTAÇÃO DE JOGOS ELETRÔNICOS

Monografia de conclusão apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Ciência da Computação, para a obtenção do título de Bacharel em Ciência da Computação.

APROVADA em 20/11/2008.

_____
Prof. Rudini Menezes Sampaio

_____
Prof. João Carlos Giacomin

_____
Prof. Cristiano Leite de Castro
(Co-Orientador)

_____
Prof. Wilian Soares Lacerda
(Orientador)

LAVRAS
MINAS GERAIS – BRASIL
2008

SUMÁRIO

LISTA DE FIGURAS

Projeto e Implementação de Jogos Eletrônicos

RESUMO

Jogos eletrônicos despertam cada vez mais interesse comercial e acadêmico. O desenvolvimento de jogos é um ramo multidisciplinar, já que um jogo é um software composto de vários módulos que devem funcionar perfeitamente entre si. Além disso, um jogo se caracteriza por ser uma aplicação em tempo real, o que exige a melhor interação possível entre hardware e software. O objetivo deste trabalho é descrever e aplicar os métodos utilizados na construção de um jogo para vídeo-game, desde sua concepção e arte até a fase final de programação e aprimoramentos, desenvolvendo para isso um jogo que exemplifica o processo. Este jogo foi desenvolvido em linguagem C++ e utiliza, entre outras técnicas e conceitos, módulos de Inteligência Artificial e de Computação Gráfica. Espera-se com esse estudo reunir as informações mais relevantes no processo de criação de jogos, servindo de referência inicial para trabalhos futuros.

Palavras-chave: desenvolvimento de jogos, vídeo-game, computação gráfica, inteligência artificial

Electronic Games Design and Implementation

ABSTRACT

Electronic games are increasing each time more commercial and academic interest. The game development is a multidisciplinary area, because a game is a software composed of several modules which must work perfectly together. Moreover, a game is a real-time application, which demands interaction between the hardware and software of the best possible form. The objective of this work is to describe and execute the methods used in the development of a game, since its conception and art until the final phase of programming and improvements, developing a game which is an example for the process. This game was developed in C++ language and uses, beyond other techniques and concepts, Artificial Intelligence and Computer Graphics modules. The expect with this work is to gather the most important information in the game development process, acting as a initial reference for future works.

Keywords: game design, video-game, computer graphics, artificial intelligence

1.INTRODUÇÃO

Um jogo eletrônico pode ser considerado um tipo de software especial, por conter os mais variados elementos como módulos de Computação Gráfica, Inteligência Artificial, Redes de Computadores, entre outros, que, além da necessidade de funcionar em perfeita harmonia, devem apresentar a característica fundamental de um jogo: devem responder em tempo real, exigindo que o hardware seja explorado da melhor maneira possível.

Como o principal objetivo de um jogo é entreter o usuário, diversos recursos artísticos e tecnológicos são combinados para criar a sensação de imersão. O desenvolvimento de jogos é, portanto, uma área extremamente interdisciplinar já que além de integrar várias áreas da computação, as aproxima de áreas como artes plásticas, design gráfico, música, etc. (CLUA, 2005).

A indústria de games movimentou 9,5 bilhões de dólares em 2007 (27% a mais do que em 2006), sendo 8,6 bilhões só com jogos para consoles (incluindo portáteis), enquanto o mercado para PC's totalizou 910,7 milhões de dólares. (IDG Now!, 2008) O crescimento foi impulsionado principalmente pelo aumento do público alvo. Pesquisas apontam que grupos como mulheres e pessoas com mais de 35 anos de idade, por exemplo, tem crescido mais do que a média anual.

No Brasil, já existem várias formas de incentivo à área de jogos eletrônicos, como a consolidação de uma Comissão Especial de Jogos e Entretenimento, criação da Associação Brasileira das Desenvolvedoras de Jogos Eletrônicos - ABRAGAMES (2008) e até mesmo ações governamentais, como concursos de jogos promovidos pelo Ministério da Cultura, bem como o reconhecimento dos jogos computadorizados como obra de audiovisual. Além disso, inúmeros editais acadêmicos são voltados para a área de entretenimento, o que evidencia a importância do setor conforme já havia sido apresentado por Battaiola (2000).

Dado esse contexto, o objetivo deste trabalho é descrever e aplicar as diversas técnicas e conceitos utilizados no desenvolvimento de jogos eletrônicos, desde a sua concepção até a programação, construindo um jogo que exemplifica o processo. São apresentados os passos utilizados e principais decisões de implementação como algoritmos escolhidos, modelos adotados, etc. Espera-se dessa forma tratar dos principais aspectos envolvidos no desenvolvimento de jogos, bem como servir de embasamento inicial para novos trabalhos,

deixando claro para o leitor os passos a serem seguidos para o desenvolvimento de um jogo, seja ele de qualquer gênero.

O jogo produzido deverá ser executado em um sistema embarcado, no caso o console *Sony Playstation Portable* (PSP) (SONY, 2008), o qual foi escolhido por possuir um ótimo poder de processamento, boa qualidade e diversidade dos periféricos integrados, suporte nativo do firmware a *homebrews* – softwares desenvolvidos pelos próprios usuários através de ferramentas diversas, além da simplicidade de programação e testes, já que é possível rodar o jogo criado diretamente da memória física do próprio dispositivo ou até mesmo de um computador, sem a necessidade de mídias auxiliares.

A motivação para uso de um console está justamente no fato de este ser um hardware dedicado a jogos, o que traz novos desafios na programação como a busca por otimização e adaptação do código a um sistema embarcado. Enquanto em jogos para computadores o usuário deve sempre atualizar seu hardware para ter compatibilidade com os jogos mais recentes, em consoles a preocupação é do programador em desenvolver um software que consiga o máximo de desempenho e qualidade de um hardware que não é atualizável, e que demora geralmente de quatro a cinco anos para ser substituído.

Este trabalho se apresenta portanto, para melhor apresentação do conteúdo, dividido em 7 capítulos de acordo com os módulos envolvidos e etapas do desenvolvimento.

No capítulo 2 são apresentados os passos principais no desenvolvimento de jogos, conceituação artística do jogo, bem como uma breve descrição dos módulos seguintes.

O capítulo 3 trata da Inteligência Artificial, que controla os NPC (*Non-Player Characters*), apresenta os principais métodos para escolha de uma boa jogada de acordo com o tipo de jogo, mostra como encontrar soluções ótimas e soluções aproximadas, comparando complexidades de tempo e espaço dos algoritmos.

O capítulo 4 trata de um dos mais importantes módulos, a Computação Gráfica, utilizada na construção da interface, explicando seus principais conceitos.

O capítulo 5 apresenta a classificação e o caminho metodológico seguido no desenvolvimento deste trabalho, bem como o material utilizado (hardware e software necessários para a execução).

O capítulo 6 descreve cronologicamente e de forma sucinta todos os passos do desenvolvimento do jogo exemplo.

O capítulo 7 apresenta os resultados alcançados com este trabalho, bem como a contribuição do estudo para trabalhos futuros, as conclusões do trabalho, além de propostas para trabalhos futuros.

O Anexo A exibe trechos do código fonte comentado, e o Anexo B as regras do jogo criado, complementando a explicação do desenvolvimento.

2.DESENVOLVIMENTO DE JOGOS

2.1. Processo de Elaboração de um Jogo

Como qualquer outro software, a produção de um jogo computadorizado, requer a adoção de um processo de desenvolvimento. No caso dos jogos eletrônicos, são tratados tanto os aspectos técnicos quanto os artísticos envolvidos no processo. Segundo ROLLINGS apud CLUA (2005), o processo de desenvolvimento de um jogo envolve as seguintes etapas fundamentais:

1. Confecção do *Design Bible*;
2. Produção de áudio e imagens 2D;
3. Modelagem 3D;
4. Desenvolvimento dos artefatos computacionais.
5. Integração dos aspectos artísticos com os aspectos computacionais.

2.1.1. Design Bible

Trata-se de um documento que contém todas as instruções de desenvolvimento e características desejadas do jogo a ser desenvolvido, já que é impossível desenvolver uma aplicação sem antes ter todas as suas especificações. Segundo CLUA (2005), a produção deste documento é uma etapa fundamental, tanto que o processo de desenvolvimento não começa sem que esse esteja pronto. O *Design Bible* deve conter os seguintes elementos descritos abaixo:

Roteiro

É um item fundamental para o processo de criação, já que é crucial para convencer os investidores da potencialidade do produto, mostrando seu diferencial em relação aos outros. Os roteiros de jogos são chamados de roteiros interativos, pois diferentemente dos roteiros de filmes, devem ter espaço para interferência do usuário no desencadeamento da história. (CLUA, 2005)

Game Design

Segundo CLUA (2005), entende-se por *game design* a conceituação artística do jogo. Nesta etapa são definidas as principais características dos cenários, desenhados esboços de

personagens (Figura 2.1), descritas as texturas fundamentais, mapas e fases (também chamado de *level design*).

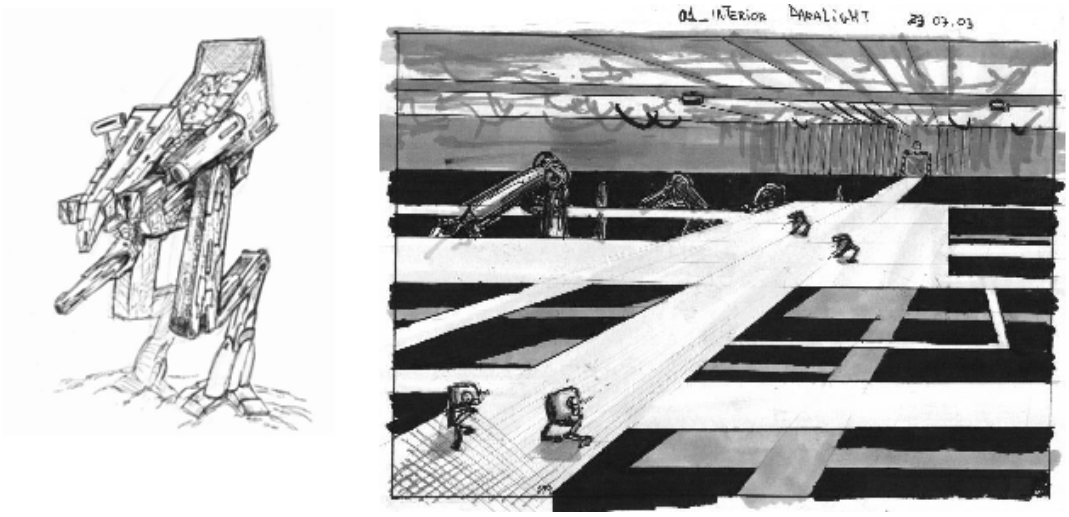


Figura 2.1 – Exemplo de conceituação artística de um personagem e de um cenário
FONTE: CLUA, 2005

Gameplay

Nesta parte do documento deve descrever-se como serão as regras do jogo, e o balanceamento das mesmas. O conteúdo desta seção é fundamental para os programadores na etapa de implementação, já que descreve as características que parte do software deve apresentar.

Interface Gráfica

Pode-se dividir a interface em *ingame* e *outgame*. A primeira é responsável pela entrada de dados do jogador para a aplicação, ou seja, botões pressionados e ações realizadas. A interface *outgame* é a forma de apresentar todos os dados do jogo, entre outras operações de suporte. Uma boa interface deve passar despercebida para o jogador, permitindo que o mesmo possa focar-se no desenrolar da história e das ações. A Figura 2.2 mostra um exemplo de interface.

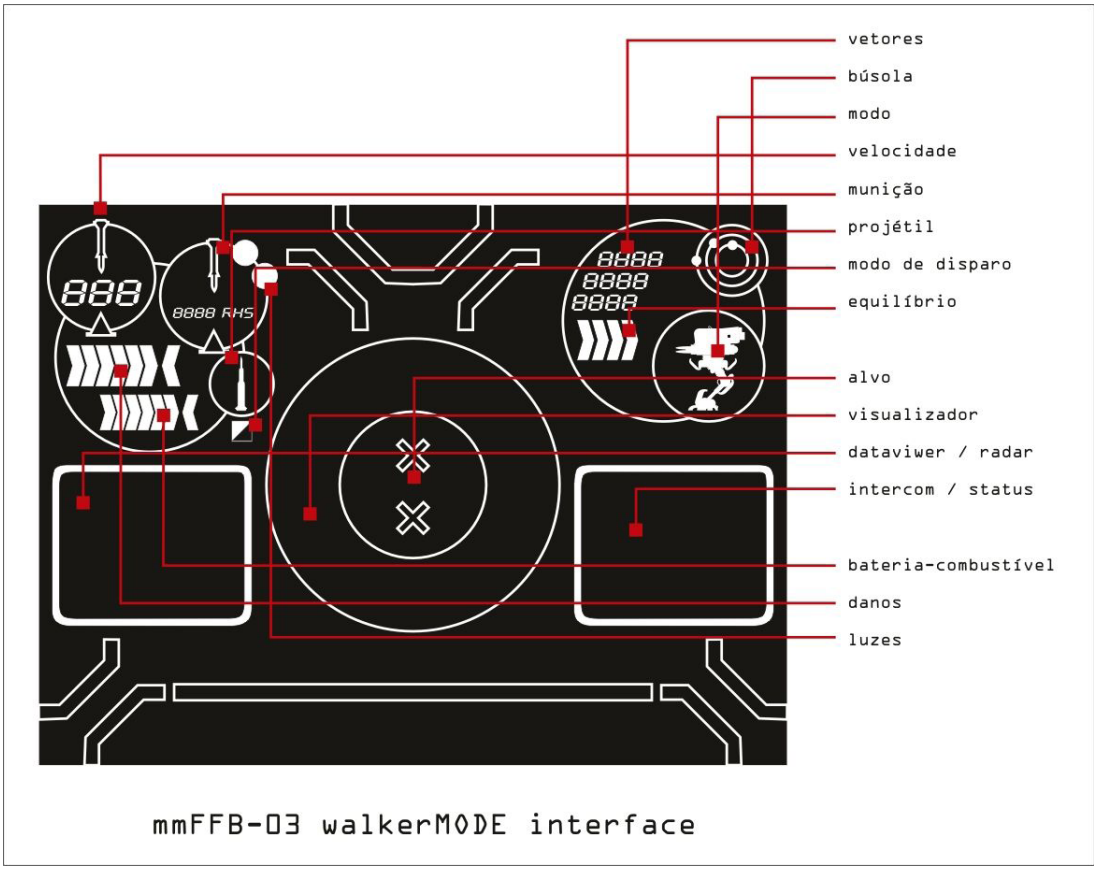


Figura 2.2 - Exemplo do design de uma interface ingame
FONTE: CLUA, 2005

Ainda segundo CLUA (2005), terminada a etapa de conceituação e elaboração, o desenvolvimento de um jogo divide-se em dois caminhos distintos: o de criação artística e o de programação, com grande interseção entre ambas. A criação artística corresponde à criação de todos vários elementos que deverão ser incluídos no jogo como modelos 3D, texturas, terrenos, sons, músicas e arquivos de configuração, enquanto a programação trata das regras do jogo e demais módulos, integrando os elementos desenvolvidos com os recursos criados.

2.1.2. Produção de Áudio e Imagens 2D

No aspecto de áudio, para incrementar a imersividade é fundamental adicionar a percepção sonora no jogo. Geralmente são usadas ferramentas específicas de edição e produção de músicas para este fim.

Quanto às imagens, é importante ressaltar que os jogos 3D não são construídos somente com modelos tridimensionais, na produção de um jogo também é necessário compor imagens bidimensionais. Em geral, tais imagens serão usadas como texturas, mas

também serão usadas para compor a interface gráfica *ingame* e *outgame*, tais como, botões, janelas, barras de energia, menus e outros componentes gráficos. Para produzir estas imagens existe uma série de ferramentas para editoração gráfica, sendo o *Adobe Photoshop* um dos softwares mais tradicionais para desempenhar tal atividade.

2.1.3. Modelagem 3D

A equipe de modelagem 3D será responsável por criar os objetos geométricos das fases. A geometria de um jogo pode ser dividida em dois tipos: modelagem estrutural e modelagem de elementos dinâmicos. (CLUA, 2005) A modelagem estrutural consiste na criação dos elementos estáticos como o cenário, e a modelagem de elementos dinâmicos consiste na criação de objetos que possuem movimento, como personagens.

Para esta etapa os principais softwares utilizados são o *3DStudio MAX*, *MAYA*, *Avid Softimage* e *Ligthwave*, pois fornecem recursos avançados tornando-os ótimos para este processo. Manipulam fácil e intuitivamente os polígonos criados, aplicam com facilidade as texturas nos polígonos, além de otimizar a modelagem dos objetos, utilizando uma quantidade reduzida de polígonos.

As ferramentas utilizadas na modelagem 3D devem possuir uma boa interface de visualização, pois para o artista, é importante que à medida que um objeto seja construído esse processo seja acompanhado em tempo real, sabendo a *priori* como o mesmo será visto no jogo. Veja Figura 2.3.

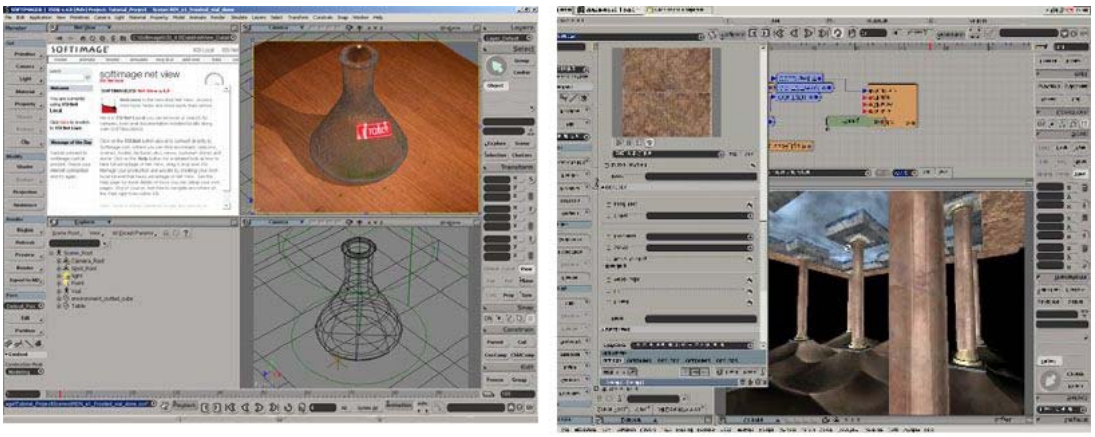


Figura 2.3 – Interface do programa Avid SoftImage
FONTE: CLUA, 2005

Neste processo, é importante que os artistas tenham capacidade de modelar objetos com quantidade reduzida de polígonos. Otimizando-se a modelagem, será possível que o

cenário possa ser mais extenso e que mais objetos possam ser inseridos no mesmo. É possível observar na Figura 2.4 que o resultado final é bastante semelhante, embora o avião da direita possua 15 vezes mais polígonos que o da esquerda. Deve-se estar atento aos limites do hardware, e quantidade de elementos que farão parte da cena para que seja utilizada uma quantidade adequada de polígonos.

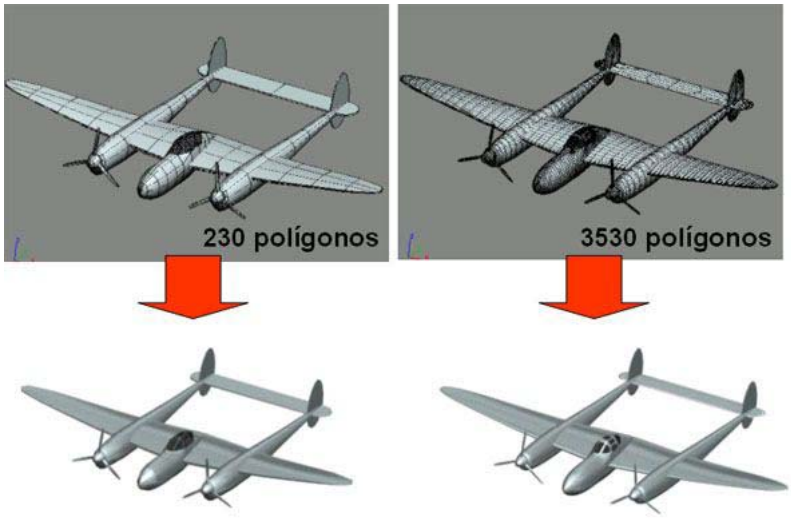


Figura 2.4 - Representação de um modelo em resoluções de polígonos diferentes
FONTE: CLUA, 2005

2.1.4. Artefatos Computacionais

Na etapa de desenvolvimento dos artefatos computacionais, são escolhidas ou criadas ferramentas para o desenvolvimento dos modelos, interação com o hardware gráfico, controle da IA, etc. Elas devem permitir que o desenvolvedor possa criar diversos jogos diferentes, reaproveitar com facilidade o código desenvolvido em projetos anteriores, abstrair a manipulação de APIs, além de possibilitar uma fácil integração entre código e modelagem 3D.(CLUA, 2005)

2.1.5. Integração

A última etapa consiste na integração dos componentes artísticos com os computacionais, ou seja, atribuir aos modelos as propriedades idealizadas, aplicar as texturas, etc. para por fim incluí-los no projeto, importando os modelos criados para o programa principal, montando assim as cenas desejadas.

2.2. Principais Componentes

Alguns dos principais aspectos a serem tratados, possuindo inclusive ferramentas exclusivas para seu tratamento são: renderização, física, som, e inteligência artificial.

A renderização é basicamente o tratamento do *pipeline* gráfico (Figura 2.5), que consiste nas transformações 3D, projeção, seleção e corte dos polígonos que serão visualizados, iluminação e texturização (CLUA, 2005).

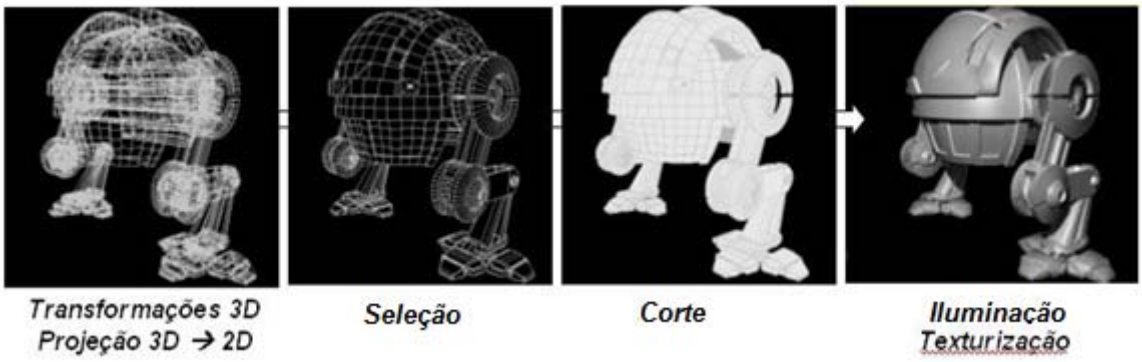


Figura 2.5 - Etapas mais importantes do pipeline gráfico
FONTE: Adaptado de CLUA, 2005

A física é o tratamento de eventos do mundo virtual que devem se assemelhar com os do mundo real como colisões, resultantes de forças, etc.

O controle do som é feito definindo-se quais os tipos e intensidades de sons que os objetos e ambiente devem apresentar durante o jogo através da manipulação dos arquivos de áudio.

A inteligência artificial descreve o comportamento de entidades não controladas pelo jogador (CLUA, 2005). Para isso são usadas máquinas de estado (Figura 2.6), busca em árvores de jogo, ou outras técnicas como, por exemplo, Redes Neurais Artificiais e Algoritmos Genéticos.

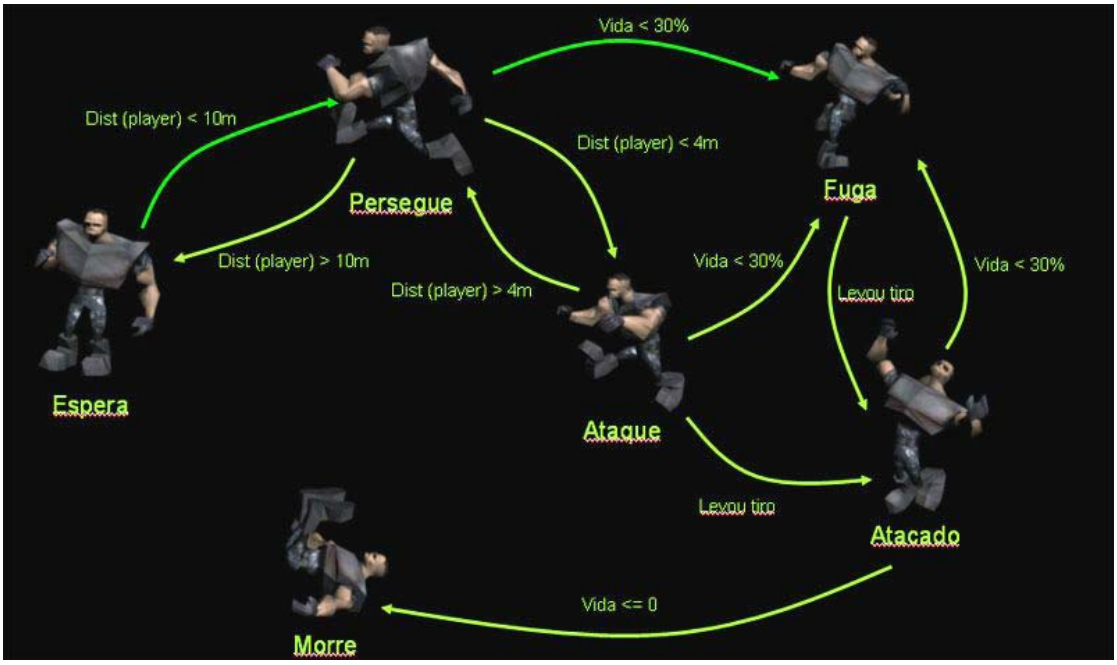


Figura 2.6 – Exemplo de máquina de estado para um jogo de ação
FONTE: CLUA, 2005

Para jogos *online* e/ou *multiplayer*, os conceitos de Redes de Computadores são utilizados para definir como o jogo irá acessar o hardware de comunicação, e a forma como os dados serão compartilhados para que seja mantida a interação em tempo real, e a sensação de imersividade desejada em jogos.

3.INTELIGÊNCIA ARTIFICIAL

A Inteligência Artificial é uma ciência recente. Abrange uma enorme variedade de subcampos, desde áreas de uso geral, como aprendizado e percepção até tarefas específicas como jogos, demonstração de teoremas matemáticos e diagnóstico de doenças. A IA sistematiza e automatiza tarefas intelectuais e, portanto, é potencialmente relevante para qualquer esfera da atividade humana. Nesse sentido, ela é verdadeiramente um campo universal. (RUSSEL, 2004)

3.1. Jogos

A teoria de jogos (matemática), visualiza qualquer ambiente multiagente como um jogo, desde que o impacto de cada agente sobre os outros seja "significativo", não importando se os agentes são cooperativos ou competitivos. Em IA, os jogos normalmente são denominados jogos determinísticos de revezamento de dois jogadores de soma zero com informações perfeitas. É o caso de um jogo de Xadrez, onde se um jogador ganha (+1), o outro jogador necessariamente perde (-1).

Os jogos, assim como no mundo real, exigem a habilidade de tomar alguma decisão, mesmo quando o cálculo da decisão ótima é inviável, e penalizam a ineficiência de forma severa. (RUSSEL, 2004)

3.2. Decisões Ótimas

Um jogo pode ser definido como uma espécie de problema de busca com os seguintes componentes:

-Estado Inicial: inclui a posição do tabuleiro e o jogador que fará a próximo movimento;

-Função Sucessor: retorna uma lista de pares (movimento, estado), cada qual indicando um movimento válido e o estado resultante;

-Teste de Término: determina quando o jogo termina;

-Função Utilidade: atribui um valor numérico aos estados terminais. (Por exemplo +1, 0 e -1 no Xadrez).

O estado inicial e os movimentos válidos para cada lado definem a árvore de jogo correspondente ao jogo. A Figura 3.1 mostra, como exemplo, parte da árvore de jogo para o jogo da velha.

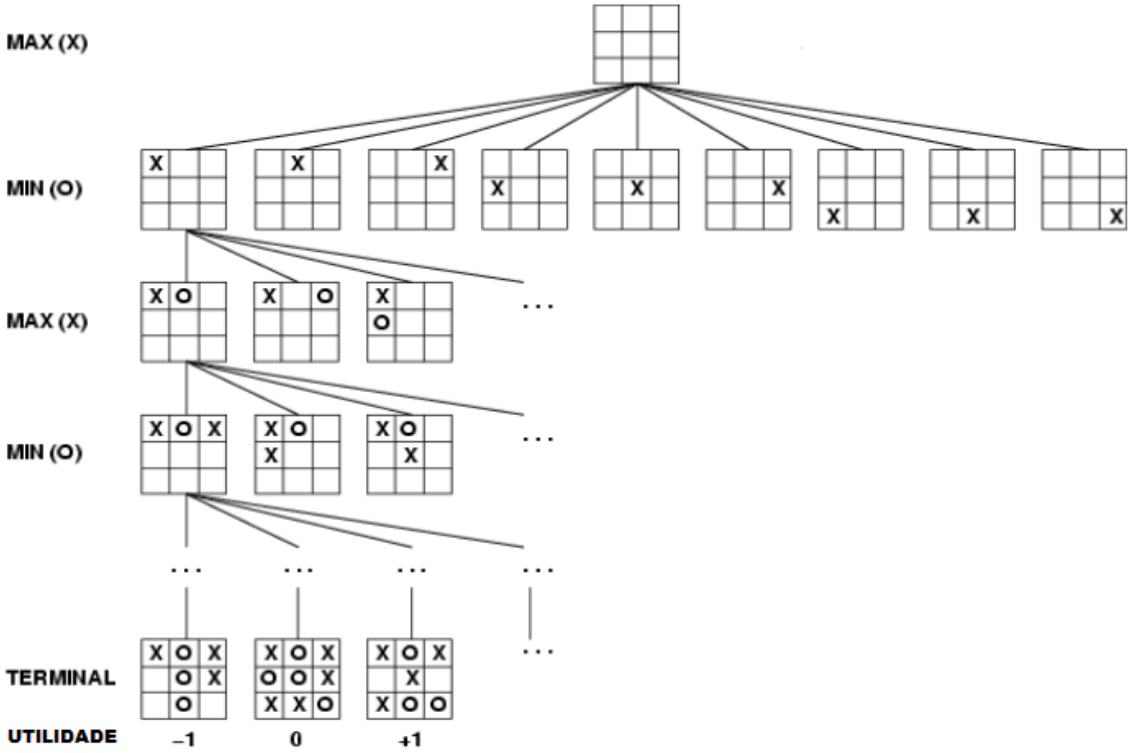


Figura 3.1 - Árvore de Jogo (parcial) para o Jogo da Velha
FONTE: RUSSEL, 2004

MAX é o jogador que inicia e o jogo, e MIN o seu adversário. A partir do estado inicial, MAX tem nove jogadas possíveis. Uma estratégia ótima leva em consideração os movimentos de MAX, e todas as respostas possíveis aos movimentos de MIN, o que faz que mesmo para um jogo simples como o jogo da velha seja inviável encontrar essa estratégia, visto que a profundidade corresponde à quantidade de jogadas possíveis.

3.2.1. O Algoritmo Minimax

O algoritmo Minimax (Figura 3.2) calcula a decisão Minimax a partir do estado corrente. Ela utiliza uma computação recursiva simples dos valores Minimax de cada estado sucessor, implementando diretamente as equações da definição :

VALOR-MINIMAX(n) =

UTILIDADE(n), se n é um estado terminal;

$\max_{s \in Sucessores(n)}$ VALOR-MINIMAX(s) se n é um nó de MAX;

$\min_{s \in Sucessores(n)}$ VALOR-MINIMAX(s) se n é um nó de MIN.

A recursão percorre todo o caminho descendente até as folhas da árvore, e depois os valores Minimax são propagados de volta pela árvore, à medida que a recursão retorna.

O algoritmo Minimax executa uma exploração completa em profundidade da árvore de jogo. Se a profundidade máxima da árvore é m e existem b movimentos válidos em cada ponto, a complexidade de tempo do algoritmo Minimax é $O(b^m)$. A complexidade de espaço é $O(bm)$ para um algoritmo que gera todos os sucessores de uma vez, ou $O(m)$ para um algoritmo que gera um sucessor de cada vez. Em jogos reais, o custo é totalmente impraticável. (RUSSEL, 2004)

```
função DECISÃO-MINIMAX(estado) retorna uma ação
    entradas: estado, estado corrente no jogo

    v ←VALOR-MAX(estado)
    retornar a ação em SUCESSORES(estado) com valor v
```

```
função VALOR-MAX(estado) retorna um valor de utilidade
    se TESTE-TERMINAL(estado) então retornar
UTILIDADE(estado)
    v ← -∞
    para a, s em SUCESSORES(estado) faça
        v ← MAX(v, VALOR-MIN(s))
retornar v
```

```
função VALOR-MIN(estado) retorna um valor de utilidade
    se TESTE-TERMINAL(estado) então retornar
UTILIDADE(estado)
    v ← ∞
    para a, s em SUCESSORES(estado) faça
        v ← MIN(v, VALOR-MAX(s))
retornar v
```

Figura 3.2 - Um algoritmo para calcular decisões Minimax
FONTE: RUSSEL, 2004

3.3. Poda Alfa-beta

O problema da busca Minimax é que o número de estados de jogo que ela tem que examinar é exponencial em relação ao número de movimentos. Infelizmente, é impossível eliminar o expoente, mas pode-se efetivamente reduzi-lo pela metade. Isso é possível através do uso do conceito de poda na árvore de busca, mais especificamente, a Poda Alfa-beta.

Quando aplicada a uma árvore Minimax padrão, retorna o mesmo movimento que Minimax retornaria, porém poda as ramificações que não terão influência possível sobre a decisão final.

O princípio geral é o seguinte: considerando um nó *n* em algum lugar da árvore, tal que o jogador tenha a escolha de movimento até esse nó. Se o jogador tiver uma escolha melhor *m* no nó pai de n ou em qualquer ponto de escolha adicional acima dele, então *n* nunca será alcançado em um jogo real.

A Poda Alfa-beta obtém seu nome a partir dos dois parâmetros a seguir:

α = o valor da melhor escolha (isto é, a de valor mais alto) encontrada até o momento ao longo do caminho para MAX

β = o valor da melhor escolha (isto é, a de valor mais baixo) encontrada até o momento ao longo do caminho para MIN

A busca alfa-beta atualiza os valores de α e β à medida que prossegue e poda as ramificações restantes em um nó tão logo se saiba que o valor do nó corrente é pior do que o valor corrente de α ou β para MAX ou MIN, respectivamente. O algoritmo completo é mostrado na Figura 3.3.

A efetividade da Poda Alfa-beta é altamente dependente da ordem em que os sucessores são examinados.

```
função BUSCA-ALFA-BETA(estado) retorna uma ação
    entradas: estado, estado corrente em jogo

    v ← VALOR-MAX(estado, -∞, +∞)
    retornar a ação em SUCESSORES(estado) com valor v
```

```
funcao VALOR-MAX(estado,α,β) retorna um valor de utilidade
    entradas: estado, estado corrente em jogo

    se TESTE-TERMINAL(estado) então retornar UTILIDADE(estado)
    v ← -∞
    para a, s em SUCESSORES(estado) faça
        v ← MAX(v, VALOR-MIN(s,α,β))
        se v ≥ β então retornar v
        α ← MAX(α,v)
    retornar v
```

```
funcao VALOR-MIN(estado,α,β) retorna um valor de utilidade
    entradas: estado, estado corrente em jogo

    se TESTE-TERMINAL(estado) então retornar UTILIDADE(estado)
    v ← +∞
    para a, s em SUCESSORES(estado) faça
        v ← MIN(v, VALOR-MAX(s,α,β))
        se v ≤ α então retornar v
        β ← MIN(β,v)
    retornar v
```

Figura 3.3 - O algoritmo de busca alfa-beta
FONTE: RUSSEL, 2004

Segundo Russel (2004), teoricamente alfa-beta precisa de $O(b^{d/2})$ nós para escolher o melhor movimento, enquanto Minimax precisa de $O(b^d)$.

Na prática, isso não pode ser realizado com perfeição, mas uma função de ordenação pode ajudar a aproximar melhor o resultado. Acrescentar esquemas dinâmicos de ordenação de movimentos, como tentar primeiro os movimentos considerados os melhores da última vez pode levar bem perto do limite teórico. (RUSSEL, 2004)

3.4. Decisões Imperfeitas em Tempo Real

O algoritmo Minimax gera o espaço de busca do jogo inteiro, enquanto o algoritmo Alfa-beta permite podar grandes partes desse espaço. Porém Alfa-beta ainda tem de fazer a busca em toda a distância até os estados terminais, pelo menos para uma parte do espaço de busca. Em geral, essa profundidade não é prática, porque os movimentos devem ser

realizados em um curto espaço de tempo. Para resolver este problema, foi proposto em 1950 por Shannon no artigo *Programming a Computer for Playing Chess* que os programas cortassem a busca mais cedo e aplicassem uma função de avaliação heurística aos dados da busca. Em outras palavras, a sugestão é alterar Minimax ou Alfa-beta de duas maneiras: a função de utilidade é substituída por uma função de avaliação de heurística AVAL, que fornece uma estimativa da utilidade de posição, e o teste de término é substituído por um teste de corte que decide quando aplicar AVAL.

3.5. Jogos que incluem um elemento de acaso

Na vida real, existem muitos eventos externos imprevisíveis, que criam situações inesperadas. Muitos jogos refletem essa imprevisibilidade incluindo um elemento aleatório, como o lançamento de dados. Em um jogo onde o lançamento de dados define a jogada seguinte, o jogador sabe quais são seus próprios movimentos válidos, mas não sabe os do adversário. Isso significa que não é possível construir uma árvore de jogo padrão, como visto anteriormente. Uma árvore de jogo em gamão, por exemplo, deve incluir nós de acaso além dos nós MAX e MIN. Esses nós são mostrados na Figura 3.4 como circunferências.

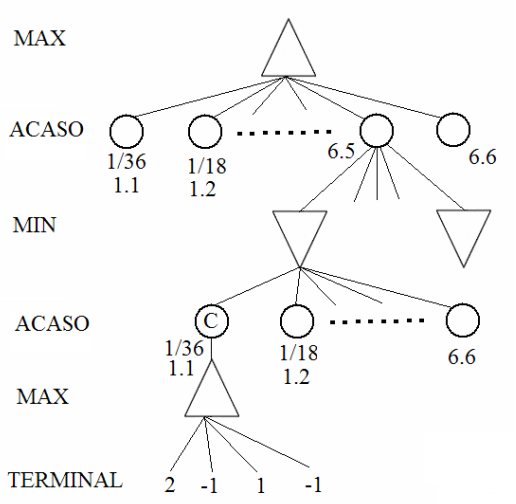


Figura 3.4 - Árvore de jogo esquemática para um jogo de gamão
FONTE: Adaptado de RUSSEL, 2004

As ramificações que levam a cada nó de acaso denotam as jogadas de dados possíveis, e cada uma é identificada com a jogada e a chance que ela ocorra.

A próxima etapa é entender como tomar decisões corretas, porém as posições resultantes não possuem valores Minimax definidos. Só é possível calcular então o valor esperado, em que a expectativa é considerada sobre todos os lançamentos de dados que poderiam ocorrer.

O valor Minimax para jogos determinísticos é então generalizado para um valor Expectminimax para jogos com nós de acaso. Nós terminais e nós de MAX e MIN (para os quais o lançamento de dados é conhecido) funcionam exatamente do mesmo modo que antes; os nós de acaso são avaliados tomando-se a média ponderada dos valores resultantes de todos os lançamentos de dados possíveis, isto é:

EXPECTMINIMAX(n) =

UTILIDADE(n), se n é um estado terminal;

$\max_{s \in Sucessores(n)}$ EXPECTMINIMAX(n) se n é um nó de MAX;

$\min_{s \in Sucessores(n)}$ VALOR-MINIMAX(n) se n é um nó de MIN

$\sum_{s \in Sucessores(n)}$ P(s) . EXPECTMINIMAX(s). se n é um nó de acaso

onde a função sucessor para um nó de acaso n simplesmente aumenta o estado de n com cada lançamento de dados possível para produzir cada sucessor s, e onde P(s) é a probabilidade de ocorrer esse lançamento de dados.

Se o programa conhecesse com antecedência todos os lançamentos de dados que ocorreriam no restante do jogo, a resolução de um jogo com dados seria muito semelhante à resolução de um jogo sem dados, o que Minimax faz no tempo de O(b^m). Como Expectminimax também está considerando todas as sequências de lançamentos de dados possíveis, ele levará o tempo de O($b^m n^m$), onde n é o número de lançamentos distintos.

Ainda que a profundidade de busca seja pequena, o custo extra comparado com o de Minimax torna pouco realista considerar a probabilidade de examinar uma distância muito grande à frente.

Alfa-beta apresenta a vantagem de ignorar desenvolvimentos futuros que não irão acontecer, se concentrando em ocorrências prováveis. Em jogos com dados, não há nenhuma seqüência provável de movimentos, porque primeiro os dados teriam que cair da maneira correta para torná-los válidos, logo uma nova estratégia é necessária para utilizar esse algoritmo. Trata-se de impor limites superiores e inferiores aos valores de utilidade, tornando possível a poda de nós de acaso.

4.COMPUTAÇÃO GRÁFICA

A Computação Gráfica é uma área da computação que se dedica ao estudo e ao desenvolvimento de técnicas e algoritmos para a geração, manipulação e análise de imagens pelo computador. Está presente em quase todas as áreas do conhecimento humano, desde o projeto de um novo modelo de automóvel até o desenvolvimento de ferramentas de entretenimento, entre as quais estão os jogos eletrônicos.

Analisando-se o histórico da Computação Gráfica é possível afirmar que a sua evolução, bem como o surgimento de novas áreas de aplicação, decorreram da evolução do próprio hardware.

Para aproveitar toda a capacidade do hardware disponível, e facilitar o desenvolvimento de aplicações gráficas, novos programas são constantemente desenvolvidos. Para os programadores, o mais usual é utilizar uma biblioteca ou um pacote gráfico, que consiste em um conjunto de rotinas básicas. Assim, é possível elaborar rapidamente programas sem a necessidade de se preocupar com detalhes particulares dos dispositivos ou com a implementação de algoritmos básicos, como por exemplo, o desenho de um segmento de reta (COHEN, 2006).

4.1. OpenGL e PSPGU

Para o console adotado, a principal ferramenta usada na programação da parte gráfica é a biblioteca PSPGU, uma biblioteca criada exclusivamente para o hardware gráfico do PSP. Há ainda a biblioteca PSPGL, uma outra adaptação da OpenGL para o PSP, porém até a data de execução deste trabalho a mesma ainda se encontra em fase de desenvolvimento e algumas funções ainda não funcionam corretamente. Mais informações sobre a PSPGL bem como atualizações e exemplos podem ser encontrados no site do projeto PSPGL (FITZHARDINGE, 2008).

A PSPGU possui a mesma forma de funcionamento da OpenGL, logo, torna-se necessário conhecer alguns conceitos, bem como principais funções e características da biblioteca OpenGL para a programação deste módulo. Além disso, é necessário conhecer as funções básicas da biblioteca GLUT, uma biblioteca auxiliar que funciona em conjunto com a OpenGL, gerenciando as janelas de visualização e cuidando do tratamento de eventos, independentemente da plataforma. A PSPGU não conta com o auxílio da GLUT

como a OpenGL, mas é importante entender o funcionamento da GLUT para que o gerenciamento das janelas de visualização e dos eventos seja implementado manualmente através das funções específicas de acesso ao hardware do PSP de forma correta.

4.1.1. Funções Básicas GLUT

Todo programa em OpenGL é estruturado em algumas funções básicas. São elas:

-Função Desenha: é utilizada sempre quando é necessário redesenhar a tela. Essa função deve limpar a tela, especificar as cores usadas para cada objeto, e criar os objetos.

-Função Teclado: assim como a Desenha, esta função é chamada em resposta a eventos, ou seja, sempre que o usuário pressionar uma tecla um trecho de código é executado.

-Função Inicializa: É chamada apenas durante a inicialização do programa, e serve para determinar o tipo e os parâmetros da projeção a ser utilizada.

-Função Main (programa principal): Chama a função de inicialização e a seguir a função que inicia o Loop com o processamento de eventos.

4.2. Definição do Espaço de Trabalho

Esta seção trata de como são abordadas em OpenGL algumas das etapas do processo de visualização para a geração de uma imagem a partir de um modelo (representação computacional de um objeto) ou conjunto de modelos.

4.2.1. Visualização Tridimensional

Quando se trabalha em três dimensões, o SRU (Sistema de Referência do Universo) passa a ser composto por três eixos ortogonais entre si (x, y e z) e pela origem (0.0, 0.0, 0.0). Uma coordenada é formada pelos valores de x, y e z, que correspondem às posições ao longo dos respectivos eixos. (COHEN, 2006)

4.2.2. Câmera Sintética

A primeira etapa do processo de visualização é a definição da cena 3D. Cada objeto que fará parte do mundo 3D é incluído e posicionado no SRU. Este posicionamento é feito através de operações de escala, rotação e translação (ver seção 4.3).

Em seguida, é incluído o observador virtual, que representa o ponto de onde a cena é vista. É um objeto como todos os outros, e portanto passa pelo mesmo processo de inclusão e posicionamento que os outros objetos. Pode ser definido fazendo analogia a uma máquina fotográfica. Possui um ponto-alvo (x, y, z), e um vetor chamado de *up* (Figura 4.1), que aponta sempre para cima, definindo a orientação da câmera.(COHEN, 2006)

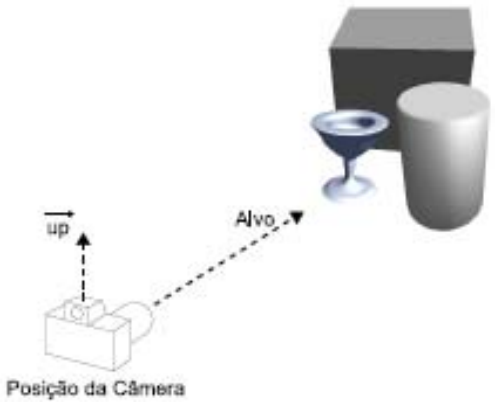


Figura 4.1 - Modelo de Câmera Sintética
FONTE: (COHEN, 2006)

4.2.3. Projeções

A tela (janela de visualização) é um plano 2D enquanto os objetos são posicionados em um mundo 3D. Para representar esses objetos na tela, existe uma etapa obrigatória: mapear suas representações 3D para imagens 2D. Esta operação de obter representações bidimensionais de objetos tridimensionais é chamada de projeção. (COHEN, 2006)

A projeção de um objeto consiste na passagem de segmentos de reta (chamados de projetantes) dos vértices até a janela de visualização. As projeções usualmente são divididas em dois tipos principais:

-Projeção Paralela Ortográfica: as projetantes são paralelas entre si, passam pelos vértices dos objetos e interseccionam o plano com um ângulo de 90º (Figura 4.2a)

-Projeção Perspectiva: as projetantes emanam de um único ponto que está a uma distância finita do plano de projeção e passam pelos vértices (Figura 4.2b).

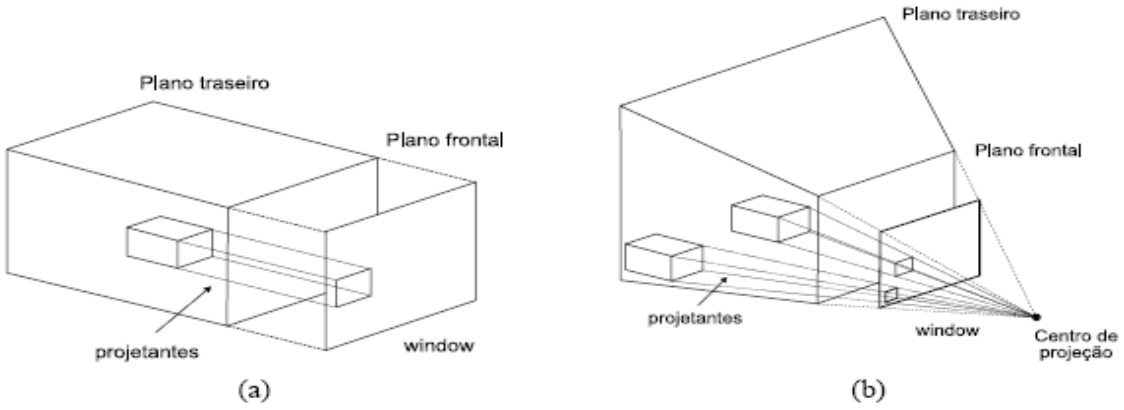


Figura 4.2 - Projeção paralela e perspectiva de paralelepípedos
FONTE: COHEN, 2006

4.3. Transformações Geométricas

Os objetos em OpenGL são definidos como um conjunto de vértices. As transformações geométricas consistem em operações matemáticas realizadas sobre estes vértices, permitindo alterar uniformemente o aspecto de um modelo já armazenado no computador. Tais alterações afetam o aspecto que o desenho vai assumir. Os três tipos fundamentais de transformações geométricas, ilustradas na Figura 4.3, são translação, rotação e escala.(COHEN, 2006)

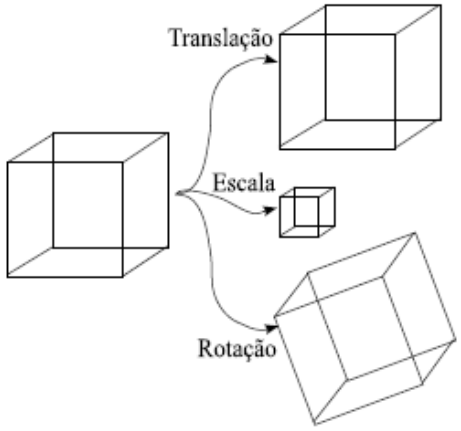


Figura 4.3 - Exemplo das transformações geométricas em um cubo
FONTE: COHEN, 2006

4.3.1. Translação

É usada para definir a posição de um objeto ou cena. Matematicamente, esta operação consiste em adicionar constantes de deslocamento a todos os vértices, ou seja, trocando o objeto ou cena de lugar, já que são dadas novas coordenadas no SRU.

4.3.2. Escala

Serve para definir a escala a ser usada para exibir o objeto ou cena. Matematicamente, esta operação consiste em multiplicar um valor de escala por todos os vértices do objeto (ou conjunto de objetos) que terá seu tamanho aumentado ou diminuído. Como se trata de uma multiplicação, para aumentar o tamanho deve ser aplicado um fator de escala maior que 1.0, em um ou nos três eixos. Para diminuir o tamanho, basta se aplicar um valor de escala entre 0.0 e 1.0. No caso de se aplicar um fator de escala negativo, o objeto terá os sinais de suas coordenadas invertidos, gerando como resultado o seu "espelhamento" no eixo que teve o fator de escala negativo.

4.3.3. Rotação

É usada para rotacionar um objeto ou cena em torno de um eixo. Matematicamente, esta operação consiste em aplicar uma composição de cálculos utilizando o seno e cosseno do ângulo de rotação a todas as coordenadas dos vértices que compõem o objeto ou cena. Quando se trabalha em 3D, também se deve definir em torno de qual eixo se procederá a rotação. É importante ressaltar que quando o valor do ângulo de rotação é positivo, a rotação é feita no sentido anti-horário.

4.3.4. Matriz de Transformação

Para combinar transformações geométricas, reduzindo a quantidade de operações matemáticas a serem aplicadas em cada vértice do modelo, estas são definidas por meio de matrizes com coordenadas homogêneas. Assim, para fazer uma combinação das transformações geométricas, multiplicam-se entre si todas as matrizes de transformação que serão aplicadas, e cada vértice é multiplicado somente pela matriz resultante, chamada de matriz de transformação corrente. (COHEN, 2006)

Tanto os comandos de visualização como as transformações geométricas aplicadas sobre os objetos são executados por meio de operações com matrizes. Portanto, no momento de aplicar as transformações, é necessário especificar qual matriz será utilizada. Por isso, há necessidade de uso das funções *LoadIdentity* e *MatrixMode*.

A primeira, *LoadIdentity*, faz com que a matriz de transformação corrente seja inicializada com a matriz identidade, indicando que nenhuma transformação foi aplicada.

A segunda, *MatrixMode,* permite identificar em qual matriz serão aplicadas as transformações. Dentre seus possíveis parâmetros pode-se destacar MODELVIEW, PROJECTION e TEXTURE, que selecionam respectivamente a matriz do modelo, da projeção ou de textura.

4.3.5. Escopo das Transformações

É possível restringir o escopo das transformações geométricas quando, por exemplo, existem vários objetos em uma cena e apenas um deles deve sofrer alguma transformação. Se for aplicada a transformação desejada e depois forem desenhados os objetos, seu efeito será aplicado em todos os objetos da cena. A OpenGL resolve este problema pela implementação de uma pilha de matrizes de transformação. Para se definir o escopo das transformações são usadas as funções *PushMatrix* e *PopMatrix*. Cada vez que *PushMatrix* é chamada, faz-se uma cópia do conteúdo da matriz corrente na memória que é armazenada no topo da pilha. Analogamente, cada vez que *PopMatrix* é chamada, a matriz armazenada no topo da pilha é retirada dali e atribuída para a matriz de transformação corrente, restaurando o estado anterior. (COHEN, 2006)

4.3.6. Transformações Hierárquicas

Muitas aplicações utilizam modelos ou cenas cujas estruturas podem ser aninhadas em forma de árvore. Um exemplo comum é um boneco, que possui o tronco como parte principal e os membros como parte de nível hierárquico inferior. Se o tronco é transladado, os braços devem acompanhar este movimento, enquanto o movimento de um braço não leva a um movimento do tronco.

Para manipular objetos hierárquicos em OpenGL é necessário acumular as transformações geométricas que vão sendo aplicadas, de maneira a combiná-las para alcançar o resultado esperado. No momento de combinar as transformações, é muito importante definir a ordem na qual os componentes de um objeto devem ser desenhados.

4.4. Realismo

Para que as imagens tenham mais realismo, é preciso considerar aspectos como iluminação e mapeamento de texturas. Sem o uso desses recursos, as imagens possuem aparência muito artificial.(COHEN, 2006)

4.4.1. Iluminação

Quando se fala de realismo em imagens geradas por CG, é necessário inicialmente compreender a natureza das fontes de luz e suas interações com os objetos, já que assim como na visão humana, a luz é fundamental para que os objetos sejam enxergados corretamente.

Existem três tipos básicos de iluminação: puntual, direcional e tipo *spot* (Figura 4.4). A luz puntual é aquela cujos raios de luz emanam uniformemente em todas as direções a partir de um único ponto no espaço, como uma lâmpada comum. Já uma fonte de luz direcional é aquela cujos raios de luz vêm sempre da mesma direção. A luz solar é o exemplo clássico de fonte de luz direcional, seus raios de luz chegam praticamente paralelos à sua superfície da Terra. O terceiro tipo mais comum de fonte de luz é a do tipo *spot*. Trata-se de uma combinação de uma luz puntual com um componente direcional: os raios de luz são emitidos na forma de um cone, apontado para uma determinada direção. O exemplo mais comum desse tipo de fonte de luz é um abajur ou uma lanterna. (COHEN, 2006)

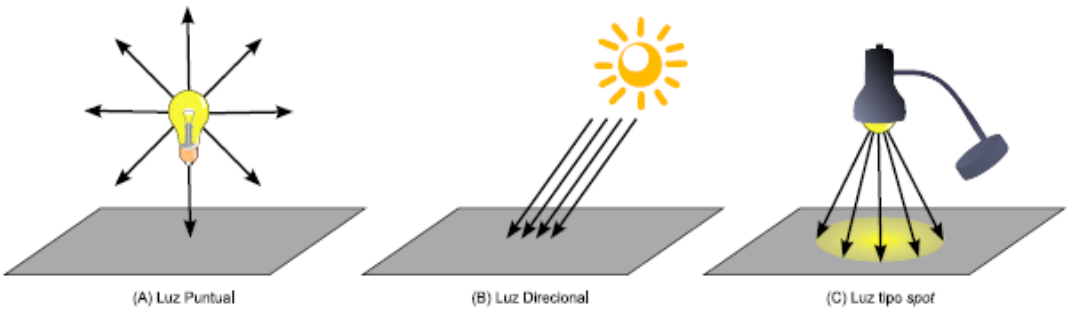


Figura 4.4 - Fontes de luz mais comuns em Computação Gráfica
FONTE: COHEN, 2006

4.4.2. Modelos de Reflexão

A interação entre a fonte de luz e a superfície dos objetos resulta na percepção de cor. Trata-se da forma como os raios que incidem sobre um objeto são refletidos. Em um computador, as cores são representadas através de um sistemas de cores, como por exemplo, o sistema RGB (vermelho, verde e azul). É importante lembrar que a fonte de luz também pode possuir cor, e não apenas ser branca.

Há três tipos básicos de reflexão: ambiente, difusa e especular. (Figura 4.5)

O modelo de reflexão mais simples é denominado reflexão ambiente (ou luz ambiente). Por definição, depende apenas da cor do objeto e gera uma iluminação constante em toda a sua superfície. Já a reflexão difusa (ou reflexão Lambertiana) ocorre na superfície da maioria dos objetos que não emitem luz. Esse tipo de reflexão depende da cor do objeto e da posição da fonte de luz: a quantidade de luz refletida percebida pelo observador não depende da sua posição. Por fim, a reflexão especular representa os objetos onde a reflexão difusa não é prefeita. Superfícies polidas, por exemplo, possuem pontos de brilho, onde a cor do brilho usualmente é a mesma da fonte de luz. À medida que a posição do observador muda em relação ao raio refletido, a intensidade do brilho diminui. (COHEN, 2006)

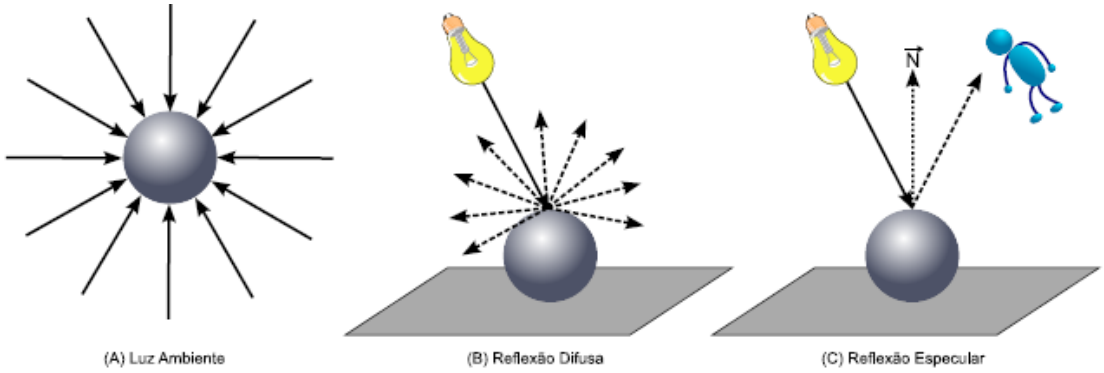


Figura 4.5 - Modelos de Reflexão da Luz
FONTE: COHEN, 2006

Para se aplicar um modelo de iluminação e de reflexão, é comum empregar um modelo de tonalização. São modelos que definem a forma como são calculadas as cores e reflexão dos objetos. Há três tipos mais comuns mostrados na Figura 4.6. O custo computacional é proporcional ao realismo obtido com o uso de cada um.

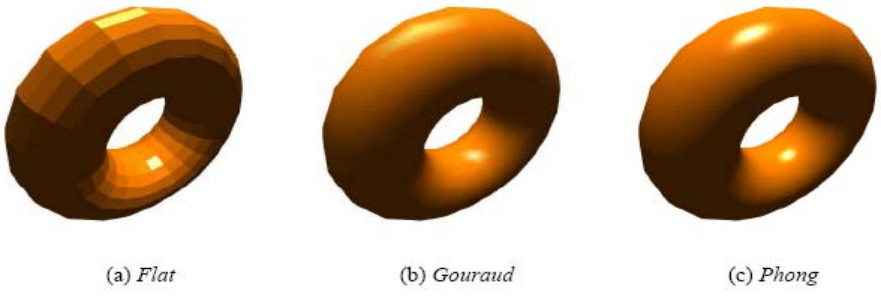
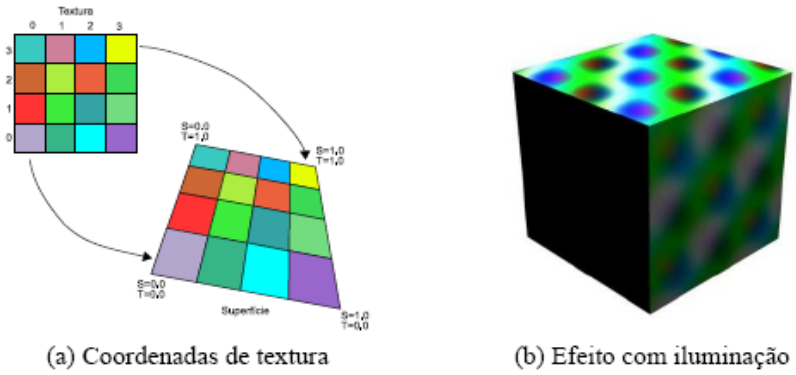


Figura 4.6 - Modelos de tonalização comumente usados em Computação Gráfica
FONTE: COHEN, 2006

4.4.3. Textura

Mesmo utilizando recursos de iluminação os objetos ainda não terão uma aparência realística. Isso ocorre porque os objetos no mundo real não possuem apenas cores, mas outros aspectos como textura e rugosidade. O processo utilizado para que um objeto seja representado com melhor riqueza de detalhes é chamado Mapeamento de Texturas. Consiste na aplicação de uma imagem sobre a superfície do objeto, através de coordenadas de textura. Estas coordenadas (usualmente chamadas de *s* e *t*) determinam como o mapeamento é realizado, conforme a Figura 4.7a apresenta. A textura funciona então, como um decalque ou um papel de parede, sendo "colada" à superfície. É importante observar que o mapeamento de textura pode (e geralmente é) combinado com o processo de iluminação, gerando um resultado ainda melhor (Figura 4.7b).(COHEN, 2006)



(a) Coordenadas de textura (b) Efeito com iluminação

Figura 4.7 - Mapeamento de Textura
FONTE: COHEN, 2006

4.5. Animação

Consiste em exibir uma seqüência de imagens em alta velocidade, ligeiramente diferentes, de maneira que o olho humano perceba apenas um movimento contínuo e não as imagens individuais. Para implementar um efeito de animação em OpenGL basta redesenhar a cena continuamente, produzindo o equivalente a uma seqüência de quadros. Pode-se causar a impressão de movimento alterando a posição, cor ou forma dos objetos. Em visualização 3D, o movimento do observador também pode causar este efeito, mesmo que os objetos estejam estáticos. Deve-se evitar um movimento descontínuo de objetos, o que ocorre quando há um grande deslocamento durante um curto espaço de tempo, por exemplo. (COHEN, 2006)

5.MATERIAL E MÉTODOS

Essa seção pretende esclarecer a classificação do trabalho, bem como os principais componentes utilizados e suas características técnicas.

5.1. Tipo de Pesquisa

Seguindo o modelo apresentado por Jung (2004), com relação ao tipo da pesquisa, esta pode ser classificada: quanto à sua natureza como aplicada, já que objetiva aplicar os conhecimentos básicos na geração de um produto; quanto aos objetivos como exploratória, já que tem por finalidade observar, registrar e analisar vários fatores no desenvolvimento de um produto; quanto aos procedimentos como experimental, já que utilizará protótipos, simulação e modelagem; e por fim classifica-se como pesquisa em laboratório, já que é possível controlar as variáveis que podem intervir no experimento.

5.2. Procedimentos

5.2.1. Fontes de Pesquisa

Dentre as fontes de pesquisa, foram utilizados livros das diversas áreas abordadas durante a execução do trabalho, além de consulta a artigos científicos relacionados ao tema e busca na internet a conteúdo informativo do hardware e tutoriais de uso das ferramentas utilizadas.

5.2.2. Ferramentas

A principal ferramenta utilizada para o desenvolvimento é o pacote PSPDev (XORLOSER, 2008) que deve funcionar em um ambiente de desenvolvimento C++, ou com outro editor de código em ambiente Windows (MICROSOFT, 2008). O projeto será compilado com a versão 4.0.2 do compilador Gcc (GNU 08) contido no pacote PSPDev, que possui além deste recurso, algumas das classes e funções necessárias para a programação do dispositivo.

As outras bibliotecas necessárias como PSPGL, libmad, etc. estão no pacote PSPDevLibInstall. Trata-se de um executável que já instala e configura todas as bibliotecas no mesmo diretório onde foi instalado o PSPDev.

Os testes foram realizados no próprio dispositivo, já que o arquivo executável gerado pode ser transferido através de uma conexão USB para a memória física do console, de onde já é diretamente executado.

5.2.3. Softwares Auxiliares

Os seguintes softwares foram utilizados:

- Adobe Photoshop CS3 – Produção e edição de imagens 2D
- Audacity – Conversão dos arquivos de áudio produzidos
- Bloodshed Dev C++ – Ambiente de desenvolvimento C++ (BLOODSHED, 2008)
- Guitar Pro 4 – Criação das músicas do jogo

5.3. Hardware Final

O console adotado é o Sony PSP Slim (Figura 5.1) (SONY, 2008). Trata-se de um ótimo sistema de entretenimento portátil, lançado em 2005 na sua versão inicial, e aperfeiçoado em 2007 quando foi lançado então na versão Slim.

Figura 5.1 - Console Sony PSP Slim

5.3.1. Especificações Técnicas

-Dimensões (LxAxC): 17 cm x 7.4 cm x 1.8 cm

-Peso: 174 g

-Processador: MIPS R4000(32-bit), 333Mhz

-Memória: 64MB RAM de memória principal e 4MB DRAM.

-Memory Bus Bandwidth: 3.2 GB por segundo

-Display: 4.3 polegadas, 480x272 pixels, 24-bit color. TFT LCD de Matriz Ativa.

-Unidade de Processamento Gráfico: 512-bit bus, 166Mhz, 2MB de VRAM onboard.

-Unidade de armazenamento: Cartões de memória flash “Memory Stick Pro Duo”, disponíveis no mercado com capacidade de 32MB a 16GB.

-Conectividade: Wi-fi IEEE 802.11b, uma porta USB 2.0 mini-B.

-Mídia principal: UMD (Universal Media Disc - discos com capacidade de 1.8GB fabricados pela Sony)

-Recursos adicionais: Sistema de auto-falantes integrado, saída para fones de ouvido, saída de vídeo (permite conectar o console a um aparelho de TV).

-Bateria: Bateria de íon Lítio, 3.6V, 1200mAh.

5.3.2. Firmware

O PSP possui um firmware atualizável que o permite além de rodar jogos, reproduzir filmes em MPEG4, aplicações em Java e Flash, músicas em MP3 e WMA, e fotos em JPEG de um disco UMD ou diretamente do Memory Stick. Além disso, possui um *browser* para navegação na Internet através de sua conexão Wi-Fi. O firmware também já está preparado para utilizar os acessórios opcionais como câmera digital, sintonizador de TV Digital e até mesmo GPS. Sua interface gráfica (Figura 5.2) apresenta de forma simples e intuitiva os recursos disponíveis ao usuário.

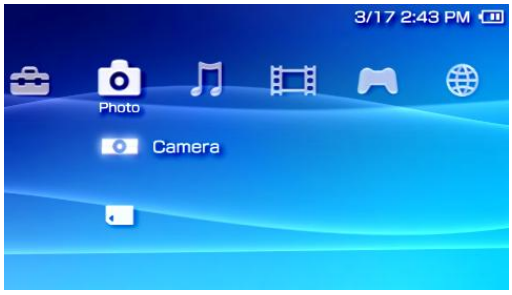


Figura 5.2 – Interface Gráfica XrossMediaBar

Até a versão 1.5, o firmware oficial da Sony suportava *homebrew* (softwares desenvolvidos pelos usuários). As versões mais novas não possuem mais esse suporte nativo, porém foram lançados *Custom Firmwares* (firmwares não-oficiais), que acompanham todas as funcionalidades dos firmwares oficiais recém-lançados, porém mantendo o suporte a *homebrew*. Para execução do jogo desenvolvido, é necessário portanto que o console esteja com a versão 1.5 oficial ou anterior, ou com uma *Custom Firmware*. Mais informações sobre *custom firmware*, bem como tutoriais para atualização de *firmware* podem ser encontrados na internet.

6.DESENVOLVIMENTO

Uma vez reunido todo o material necessário, é possível iniciar o desenvolvimento. Este capítulo apresenta em ordem cronológica todos os passos seguidos até se alcançar os objetivos deste trabalho

6.1. Concepção

6.1.1. Design Bible

Como descrito no capítulo 2, o primeiro passo no desenvolvimento de um jogo é a confecção de um *Design Bible*, documento que possui todas as informações necessárias para a execução dos passos seguintes.

O *Design Bible* do jogo desenvolvido conta com os seguintes elementos:

- Roteiro: por ser um jogo exemplo, não foi elaborado um roteiro completo, mas sim um contexto para o jogo.
- *Game Design*: foi definido o cenário principal, criados os esboços das texturas posteriormente produzidas, esboços das peças (personagens).
- *Gameplay*: o balanceamento das regras foi feito tendo como bases outros jogos já famosos. Procurou-se criar regras simples, fáceis de serem aprendidas, mas ao mesmo tempo interessantes. Além disso, foram incluídos elementos que dão ao jogo características não-determinísticas, diminuindo as possibilidades de se prever qual o próximo movimento do adversário. As regras do jogo estão descritas detalhadamente no Anexo B.
- Interface *ingame*: os controles do jogador foram definidos usando como base jogos de vídeo-game conhecidos, e procurou-se dar liberdade ao usuário para controlar além dos seus próprios personagens, a visualização do cenário. O Anexo B contém uma tabela com os botões utilizados e suas respectivas funções.
- Interface *outgame*: trata-se de um tabuleiro 8x8, visualizado tridimensionalmente, há um menu lateral com informações de ajuda e sobre o jogo. As principais regras e comandos estão listados em uma tela de ajuda que o usuário pode acessar quando desejar.

Além destas informações, foram citados alguns aspectos técnicos como forma de execução do jogo no console, e decisões de implementação que complementam a descrição do *gameplay* e servem de base para a fase de programação.

6.2. Produção de Áudio e Imagens 2D

Esta seção apresenta resumidamente os passos usados na criação das músicas do jogo, e das imagens 2D, utilizadas principalmente como texturas.

6.2.1. Áudio

A principal ferramenta para esta fase do desenvolvimento é o software Guitar Pro 4 (GUITAR-PRO, 2008). Apesar da facilidade de uso do programa, é importante que o compositor tenha bons conhecimentos em teoria e percepção musical, sabendo ler e escrever partituras.

Cada trilha do projeto corresponde a um instrumento a ser utilizado, e é editada individualmente. Inúmeras combinações de instrumentos podem ser utilizadas. O software permite ainda importar músicas já prontas em formato MIDI e editá-las.

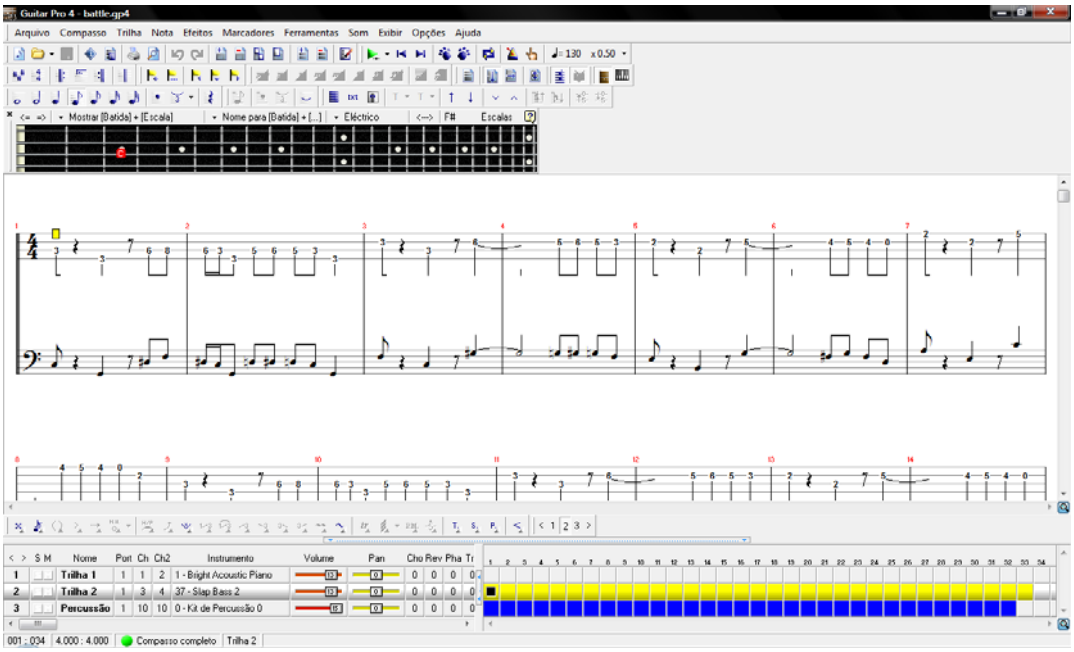


Figura 6.1 - Tela do programa Guitar Pro 4

Uma vez criada a música e salvo o projeto, é possível exportar o áudio em formato MIDI. Torna-se então necessária a utilização de um software que converta a música criada para um formato compatível com o PSP. Para este fim foi utilizado o software Audacity.

Ele também permite a edição do áudio, porém não nota por nota como o Guitar Pro, mas sim, edição da forma de onda final, já com todos os instrumentos. Nele é possível remover ruídos indesejáveis, aumentar a amplitude, incluir efeitos e cortar/incluir trechos.

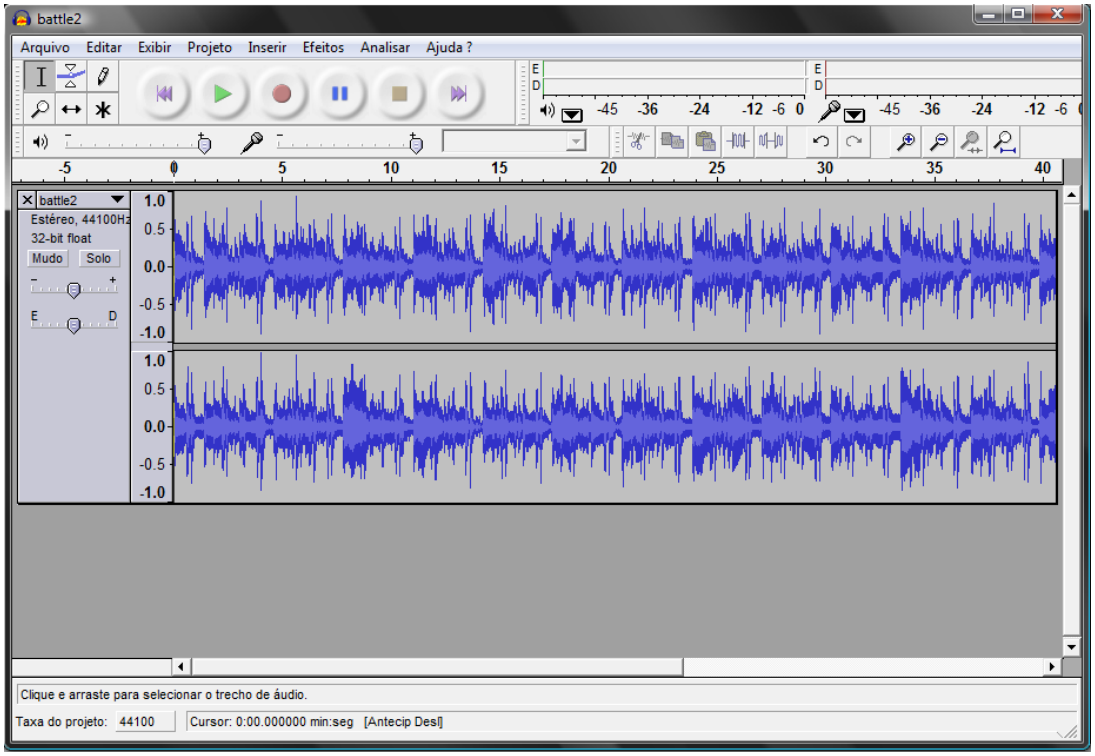


Figura 6.2 - Tela do programa Audacity

Após a edição ser concluída, o projeto é salvo, e o áudio deve ser exportado como MP3, formato a ser utilizado pelo PSP.

6.2.2. Imagens 2D

Esta etapa engloba não apenas as texturas propriamente ditas, mas também as outras imagens do jogo como menus e demais telas. Está muito próxima da etapa de texturização, e portanto é necessário conhecer antecipadamente qual o tamanho de imagem em pixels utilizado pela biblioteca gráfica e qual o formato do arquivo.

As imagens usadas como menus, apresentação e informações também são carregadas como texturas em retângulos do tamanho da janela de visualização, passando portanto pelo mesmo processo que as demais texturas.

O software utilizado nesta etapa é o Adobe Photoshop CS3.

No caso, o método escolhido para carregar texturas utiliza imagens em formato Targa (.TGA), de 32 bits, e as dimensões da figura (altura e largura) devem ser potências de dois.

O display do PSP possui 480x272 pixels. Para que as proporções se mantenham durante a execução, a imagem deve ser editada utilizando estas medidas em pixels, e só depois de finalizada ser redimensionada para as dimensões de importação, que são as potências de dois mais próximas (no caso 512x256) como mostra a Figura 6.3. Isso vale para todas as outras imagens, sempre deve-se editá-las nas proporções reais desejadas e só por fim redimensioná-las.

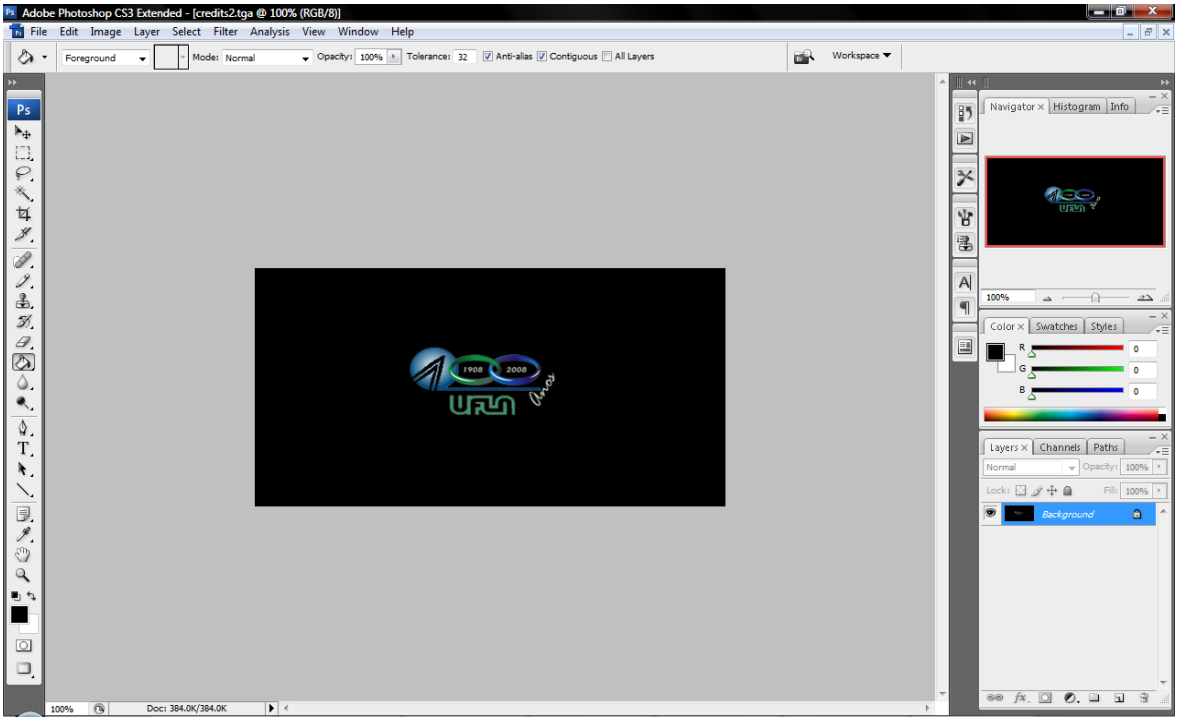


Figura 6.3 - Tela do programa Adobe Photoshop CS3

6.3. Modelagem 3D

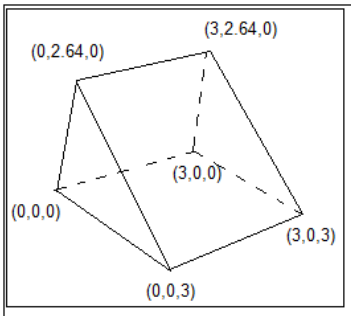
Assim como a produção de imagens 2D, a modelagem 3D está intimamente ligada com a etapa de implementação gráfica. Os modelos 3D produzidos são incluídos no jogo posteriormente. Geralmente são utilizados softwares específicos para este fim como o 3DStudio e Maya, porém como o jogo desenvolvido possui modelos muito simples, estes foram criados diretamente no código fonte do programa.

Um modelo consiste de um vetor de vértices, que possui além da informação de posição dos vértices, a cor desejada e as coordenadas para o mapeamento de texturas.

Para o PSP especificamente, há diversas formas de descrever os modelos através dos vértices, visto que há várias formas de desenho dos modelos. Mais uma vez é necessário que a forma de implementação seja bem estabelecida, já que o projeto do modelo depende completamente da forma de desenho a ser utilizada pelo hardware gráfico.

Para os modelos criados, foi escolhido o tipo TRIANGLE_STRIP, onde para desenhar um triângulo basta especificar os três pontos, e para polígonos de mais vértices usa-se uma composição de triângulos, como quadrados por exemplo, em que se especificam dois triângulos que possuem dois vértices coincidentes.

A figura abaixo mostra o modelo de peça criado e o código que especifica os vértices.




```
Vertex __attribute__((aligned(16))) pecaVermelha[3*6] =
{
    //lado esquerdo
    { -0, -0, GU_COLOR( 1.0f, 1.0f, 1.0f, 0.0f ),0.0f, 0.0f, 0.0f },
    { -0, -0, GU_COLOR( 1.0f, 1.0f, 1.0f, 0.0f ),0.0f, 2.64f, 0.0f },
    { -0, -0, GU_COLOR( 1.0f, 1.0f, 1.0f, 0.0f ),0.0f, 0.0f, 3.0f },
    //frente
    { 0, 1, GU_COLOR( 1.0f, 0.0f, 0.0f, 0.0f ),0.0f, 2.64f, 0.0f },
    { 1, 1, GU_COLOR( 1.0f, 0.0f, 0.0f, 0.0f ),3.0f, 2.64f, 0.0f },
    { 0, 0, GU_COLOR( 1.0f, 0.0f, 0.0f, 0.0f ),0.0f, 0.0f, 3.0f },

    { 0, 0, GU_COLOR( 1.0f, 0.0f, 0.0f, 0.0f ),0.0f, 0.0f, 3.0f },
    { 1, 1, GU_COLOR( 1.0f, 0.0f, 0.0f, 0.0f ),3.0f, 2.64f, 0.0f },
    { 1, 0, GU_COLOR( 1.0f, 0.0f, 0.0f, 0.0f ),3.0f, 0.0f, 3.0f },

    //lado direito
    { -0, -0, GU_COLOR( 1.0f, 1.0f, 1.0f, 0.0f ),3.0f, 0.0f, 3.0f },
    { -0, -0, GU_COLOR( 1.0f, 1.0f, 1.0f, 0.0f ),3.0f, 2.64f, 0.0f },
    { -0, -0, GU_COLOR( 1.0f, 1.0f, 1.0f, 0.0f ),3.0f, 0.0f, 0.0f },
    //tras
    { -0, -0, GU_COLOR( 1.0f, 0.2f, 0.2f, 0.0f ),3.0f, 2.64f, 0.0f },
    { -0, -0, GU_COLOR( 1.0f, 0.2f, 0.2f, 0.0f ),0.0f, 2.64f, 0.0f },
    { -0, -0, GU_COLOR( 1.0f, 0.2f, 0.2f, 0.0f ),3.0f, 0.0f, 0.0f },

    { -0, -0, GU_COLOR( 1.0f, 0.2f, 0.2f, 0.0f ),3.0f, 0.0f, 0.0f },
    { -0, -0, GU_COLOR( 1.0f, 0.2f, 0.2f, 0.0f ),0.0f, 2.64f, 0.0f },
    { -0, -0, GU_COLOR( 1.0f, 0.2f, 0.2f, 0.0f ),0.0f, 0.0f, 0.0f }
};
```

Figura 6.4 - Modelo 3D e código utilizado para sua criação

Cada linha especifica um único vértice.

Os dois primeiros parâmetros são as coordenadas de mapeamento da textura. No exemplo mostrado esta é aplicada somente na frente da peça. O valor “-0” é utilizado para determinar que neste vértice nenhuma textura será aplicada. As coordenadas para mapeamento 2D são especificadas conforme a figura abaixo.

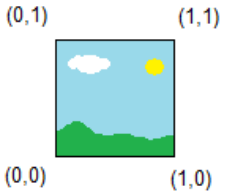


Figura 6.5 – Coordenadas de mapeamento de texturas 2D

O terceiro parâmetro é a cor do fragmento desenhado em RGBA. É necessário saber qual o modo de texturização utilizado para aplicar a cor correta. No jogo foi usado o modo MODULATE, onde a cor final do fragmento corresponde à multiplicação entre a cor original do fragmento e a cor da textura aplicada. Com isso, caso a textura possua partes brancas, estas funcionam como transparente, e caso o fragmento original seja branco, as cores predominantes são as da textura, uma propriedade bastante útil.

Os últimos três parâmetros correspondem às coordenadas do vértice no SRU.

6.4. Implementação

Na etapa de implementação, todos os módulos são integrados conforme as especificações do *Design Bible*.

6.4.1. Projeto Inicial

Deve-se levar em conta todos os detalhes especificados previamente. Nesta etapa, são criados os "esqueletos" das classes com as chamadas das principais funções e são definidos os tipos das principais variáveis envolvidas.

Para o jogo criado, foram criadas duas classes: "ia.h", que contém as funções da inteligência artificial e define as jogadas da CPU, e "tabuleiro.h", que testa as regras de movimentação e ataque das peças, e cria os parâmetros iniciais do jogo. A interface gráfica foi implementada no arquivo "main.cpp", que contém ainda a implementação do jogo propriamente dito, com as chamadas para as funções das outras classes criadas, e leitura dos botões de controle.

Como a biblioteca gráfica do PSP é baseada na OpenGL, o funcionamento do programa deve seguir o padrão de um programa OpenGL comum. A PSPGU não conta com a GLUT, logo os *callbacks* de teclado e redesenho por exemplo foram implementados com as funções específicas do hardware.

Com o início da função main inicia-se um *loop* infinito onde a função "teclado" é chamada, seguida pela função "desenha". Como nesta etapa a interface gráfica final ainda não está pronta, esta função "desenha" possui os comandos para exibir as informações do jogo em modo texto, possibilitando os testes dos outros módulos e simplificando posteriormente a inclusão da interface gráfica, sendo necessário apenas substituir os comandos de texto pelos comandos da PSPGU, e incluir as funções adicionais criadas.

A inclusão das músicas no jogo pode ser implementada a qualquer momento da etapa do desenvolvimento, porém quanto mais cedo isto acontecer mais fácil se torna a detecção e correção de possíveis erros. No modelo criado, foi usada a biblioteca libmad contida no pacote PSPDevLibInstall para este fim.

6.4.2. Módulo de Validação das Regras

Com base nas regras definidas no *Design Bible* (ver Anexo B), foi implementada uma função no arquivo "tabuleiro.h" que tem a finalidade de retornar ao programa se a jogada testada é válida ou não. Trata-se de uma série de testes que verificam se o jogador não tentou movimentar as peças do adversário, movimentar suas peças além do permitido, e nem atacar além do alcance da peça selecionada. Neste mesmo arquivo é implementada a função que avalia se as condições de fim de jogo foram atendidas ou não, e a função que realiza a jogada desejada, alterando a matriz do tabuleiro quando necessário.

Em resumo, esta classe possui implementações relacionadas ao tabuleiro e à jogada, independente do jogador.

6.4.3. Inteligência Artificial

É importante que a classe com a implementação das regras já esteja pronta nesta fase do desenvolvimento, pois não há como realizar os testes com a IA sem ela. Nesta etapa do desenvolvimento são aplicados algoritmos e técnicas para que o jogador possa interagir com a CPU.

Como citado anteriormente, uma das estratégias mais utilizadas para jogos de tabuleiro é o algoritmo Minimax. Este foi utilizado com o algoritmo de Poda Alfa-beta para obtenção de um melhor desempenho. A classe "ia.h" recebe do módulo principal o estado atual do tabuleiro e após o processamento é escolhida a jogada da CPU. É preenchido

então um vetor com os dados necessários (linha e coluna inicial e final da jogada da CPU), e estes valores são passados para a classe tabuleiro para que a jogada seja realizada.

A seção 3.4 mostra que como o uso do Minimax convencional em jogos mais complexos é impraticável, há uma forma de se obter um resultado aproximado em tempo satisfatório através de heurísticas. Essa forma de implementação foi adotada, substituindo a função de utilidade por uma função que avalia se um estado é o melhor, e ainda definindo quando utilizá-la. Em termos práticos, isso foi feito criando uma função que atribui um peso para cada estado e um limite para a profundidade da árvore de jogo. Os estados terminais passam a ser os estados com profundidade máxima. Para cada um deles é atribuído um peso, e no final da execução é escolhida a jogada que levou ao tabuleiro de maior peso.

A heurística que atribui pesos funciona da seguinte forma: é criada uma variável com valor zero. Em seguida, cada posição do tabuleiro é lida. Para cada peça do jogador atual que ainda está no tabuleiro, é somada a essa variável um determinado valor; para cada peça do adversário que ainda se encontra em jogo, é subtraído um determinado valor, forçando assim que a CPU ataque quando possível. Além disso, é somado um valor adicional quando as peças estão mais próximas do fundo do tabuleiro do adversário, o que força a CPU a movimentar suas peças para frente, e não apenas esperar que o adversário se aproxime.

Durante a execução do Minimax, o algoritmo não joga dados para simulação de ataques, ou seja, a CPU joga sem avaliar a probabilidade de um ataque ser perdido, pois como as chances da defesa são maiores, a chance de que a CPU deixasse de executar um ataque seria grande.

Para diminuir a previsibilidade das jogadas da CPU, foi incluído um teste antes da jogada, no qual há 10% de chance de que ela realize uma jogada completamente aleatória.

6.4.4. Computação Gráfica

Deve-se iniciar esta etapa com um estudo das funções do hardware gráfico através de tutoriais, exemplos e documentação.

A interface do jogo criado está incluída no arquivo “main.cpp” e se baseia em duas funções principais: “desenha” e “desenhaMenu”. A primeira é responsável por desenhar o

tabuleiro e as outras telas do jogo, enquanto a segunda é responsável pelo desenho da tela inicial e menu principal do jogo.

Os principais passos usados pela função “desenha” são:

1-Limpa a janela de visualização;

2-Define-se a projeção a ser utilizada e seus principais parâmetros;

3-Posiciona o observador virtual;

4-Desenha o tabuleiro e aplica sua textura;

5-Lê a matriz do tabuleiro e para cada peça encontrada desenha uma peça da cor correspondente e aplica a textura correspondente à sua função. Em seguida essa peça é transladada da origem (de onde é desenhada inicialmente) até a sua posição real no jogo;

6-É desenhado o cursor com base na sua posição atual. A posição inicial do cursor e posição final recebem texturas diferentes durante as jogadas;

7-É carregada a matriz identidade para que as transformações anteriores não tenham efeito a partir deste ponto, e então é desenhado um retângulo que recebe a textura de menu lateral com as informações necessárias. Este menu permanece fixo em sua posição, bem como todos os elementos que aparecem neste menu.

Já a função “desenhaMenu” executa os seguintes passos:

1-Limpa a janela de visualização;

2-Define-se a projeção a ser utilizada e seus principais parâmetros;

3-Posiciona o observador virtual;

4-Desenha um retângulo do tamanho da janela de visualização;

5-Se esta função está sendo chamada pela primeira vez, aplica seqüencialmente as texturas de apresentação com nome do autor, universidade, etc;

6-Aplica a textura com o menu principal (a opção atualmente selecionada fica em destaque).

Outra função importante para a interface é a função InitGu. Esta função é chamada no início do programa e nela são aplicados vários parâmetros que definem como será feita a

visualização dos elementos posteriormente. São definidos, por exemplo, o modo de texturização adotado, parâmetros da janela de visualização, cores utilizadas, entre outros.

Outras funções auxiliares foram criadas para atender chamadas de eventos especiais, como *Game Over*, *Pause*, etc. e possuem funcionamento semelhante à função "desenhaMenu", ou seja, é desenhado um retângulo do tamanho da janela de visualização e aplicada a textura com a imagem adequada.

Seguindo os passos do tutorial consultado (PSP-PROGRAMMING, 2008), foi criada a classe TGALoader.h responsável por carregar as imagens produzidas para o jogo.

6.4.5. Função Principal

Incluída no arquivo "main.cpp", implementa os modos de jogo possíveis e inicializa todas as variáveis. A execução consiste nos seguintes passos:

1 – É definida a frequência de operação do processador, são iniciados os *callbacks* principais e é definido o modo de operação do controle;

2 – São inicializados os dados do jogo, e o áudio;

3 – São carregados todos os arquivos de imagens, e rearranjados os dados nas matrizes através da função *swizzle*. Cada arquivo de imagem é carregado em um novo objeto da classe TGALoader;

4 – Inicializa-se a GU com seus parâmetros definidos anteriormente;

5 – Inicia o *loop* do jogo.

O *loop* do jogo é, a princípio, o menu principal. A partir dele as opções são acessadas e os diferentes modos de jogo iniciados.

As funções *tecladoMenu* e *tecladoJogo* fazem as alterações necessárias nas variáveis e executam várias funções quando solicitado. Sempre as funções teclado são seguidas de uma função *desenha* correspondente que mostra na tela eventuais alterações realizadas.

São apresentados no Anexo A trechos do código fonte do jogo comentado, permitindo uma análise mais detalhada da etapa de implementação e das funções utilizadas.

6.5. Testes e Aprimoramentos

Os testes foram realizados exaustivamente, e em todas as etapas do desenvolvimento. A cada compilação do código, um novo teste era realizado no próprio dispositivo. Após o jogo funcionando corretamente, alguns recursos foram adicionados como por exemplo um modo de jogo onde não há atuação da CPU (ambos os exércitos são controlados pelo jogador), foram incluídas novas informações para o usuário no menu lateral do jogo e vários trechos do código foram otimizados em busca de melhor desempenho.

7.RESULTADOS E CONCLUSÃO

A execução dos passos apresentados neste trabalho permitiu a implementação de um jogo para vídeo-game atendendo todos os requisitos especificados no seu projeto.

A técnica de Inteligência Artificial utilizada foi eficiente na resolução do problema proposto, gerando jogadas válidas em um baixo tempo de execução. A heurística implementada faz com que a CPU avance no tabuleiro quando possível e ataque se alguma peça adversária estiver ao alcance. Como o Minimax percorre grande parte da árvore de jogo, e o tamanho desta é proporcional à quantidade de jogadas possíveis, a IA gasta mais tempo para realizar as jogadas quando há mais peças no jogo, e quando há possibilidade de ataque. 10% das jogadas são completamente randômicas.

As técnicas de Computação Gráfica foram usadas na produção de uma interface 3D (Figuras 7.1, 7.2 e 7.3), que além de exibir as informações do jogo ainda apresenta as principais regras e objetivos do jogo quando solicitado, ajudando o usuário progredir no jogo.

Figura 7.1 - Menu Principal

Figura 7.2 - Interface do jogo

Figura 7.3 - Jogo já iniciado

A produção artística (áudio, imagens 2D, etc.) foi incluída no jogo da forma desejada, aumentando a imersividade esperada.

Foi criada uma tela de *help* (Figura 7.4) que é exibida toda vez que o jogador pressionar o botão *Select*. Esta tela possui as regas do jogo e os comandos do jogador descritos de forma sucinta, permitindo que os jogadores iniciantes se adaptem mais rapidamente ao jogo

Figura 7.4 - Tela de *Help*

Vários aspectos interessantes foram observados durante o desenvolvimento:

- É extremamente importante que o projetista conheça a fundo todas as áreas do desenvolvimento, pois deve definir todas as ferramentas e parâmetros necessários para execução do trabalho, inclusive aspectos técnicos envolvidos em cada módulo.
- As interfaces das classes devem ser muito bem definidas no projeto, pois, alterações após o início da implementação comprometem todo o desenvolvimento.
- A demanda por tempo e por uma equipe de desenvolvimento aumenta proporcionalmente à complexidade do projeto.
- Quando o projeto é dividido em equipes de desenvolvimento, é importante que todos os envolvidos no projeto conheçam as limitações do hardware final e tenham como meta um bom aproveitamento dos seus recursos, mesmo para os módulos que não utilizam as funções do hardware diretamente.

Obteve-se como principal resultado tanto o jogo desenvolvido como o próprio trabalho, o qual pode ser usado como referência inicial para o desenvolvimento de novos jogos. Sendo assim, a principal contribuição desse estudo é a sistematização do processo de criação de jogos eletrônicos através do exemplo do software desenvolvido, proporcionando ao leitor uma visão abrangente e ao mesmo tempo detalhada do processo de desenvolvimento de um jogo.

7.1. Trabalhos Futuros

Pretende-se aprofundar o estudo do desenvolvimento de jogos abordando outras áreas do desenvolvimento como a implementação de um modo *multiplayer* através da interface de rede *wireless* que o PSP possui, bem como estender a fase de aprimoramentos com a inclusão de mais cenários, modos de jogo e recursos.

Pretende-se também aprofundar o estudo da etapa de Inteligência Artificial para jogos com a implementação de outras técnicas, no caso o uso de Redes Neurais Artificiais em conjunto com o algoritmo Minimax já implementado. Os dados para treinamento da rede serão gerados manualmente, e pela execução do Minimax já implementado. O desempenho das duas implementações será comparado e analisado.

Outra possibilidade de aprofundamento está na criação de modelos 3D mais complexos, adotando um mapeamento de texturas alternativo, criado para este fim.

ANEXO A – Código Fonte Comentado

Este capítulo apresenta trechos do código fonte complementando a descrição do desenvolvimento.

A.1 Arquivo “main.cpp”

```
//Este arquivo possui o modulo principal do programa e possui as
//funcoes usadas na implementação da interface.

#include <pspkernel.h>
#include <pspctrl.h>
#include <pspdebug.h>
#include <pspaudio.h>
#include <pspaudiolib.h>
#include <psppower.h>

#include <malloc.h>           //For memalign()
#include <pspdisplay.h>

#include <pspgu.h>
#include <pspgum.h>

#include "mp3player.h"
#include <stdio.h>      // Header file for standard file i/o.
#include <stdlib.h>     // Header file for malloc/free.
#include "tabuleiro.h"
#include "ia.h"
#include "TGALoader.h"


PSP_MODULE_INFO("TCC Daniel", 0, 1, 1);
#define printf pspDebugScreenPrintf
#define sleep sceKernelDelayThread

#define BUF_WIDTH (512)
#define SCR_WIDTH (480)
#define SCR_HEIGHT (272)

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//Tipo criado para armazenar as informações dos vértices dos polígonos
typedef struct {
    float u,v;
      unsigned int color;
      float x, y, z;
} Vertex;




//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//Variaveis globais
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
```

```
SceCtrlData pad;

bool help;
bool credits = true;
bool gameExit;
int linhaAtual, colunaAtual;
int linhaInicial, colunaInicial, linhaFinal, colunaFinal;
int jogador;
int cursorMenu, opcaoMenu;
int resetCursor = 0;

void *dList;      // display List, usado pela sceGUStart
void *fbp0;       // frame buffer

//variáveis utilizadas para aramazenar as coordenadas do observador
float obsX, obsY, obsZ;
float rotX, rotY;

//objetos criados para armazenar as texturas
CTGATexture texCr, texNG, texLG, texMP, texHELP, texML, texGO, texQS;
CTGATexture texture, textureA, textureS, textureR, textureB, textureC,
textureCur, textureCurIni;
CTGATexture da1, da2, da3, da4, da5, da6, dd1, dd2, dd3, dd4, dd5, dd6;
CTGATexture j1, j2;

//-------------------------------------------------------------------------
//FUNCOES DE CALLBACK DO PSP
//-------------------------------------------------------------------------
 (...)
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//DESCRIÇÃO DOS MODELOS 3D E PLANOS
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
```

(...) Como mostrado na Figura 6.4

```
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FUNÇÃO CONVERTE POSIÇÃO
//Converte as coordenadas do tabuleiro nas coordenadas utilizadas no
plano
float convertePosicao(int valor){
    switch(valor){
        case 0: { return -19.0f; }; break;
        case 1: { return -14.0f; }; break;
        case 2: { return -9.0f; }; break;
        case 3: { return -4.0f; }; break;
        case 4: { return 1.0f; }; break;
        case 5: { return 6.0f; }; break;
        case 6: { return 11.0f; }; break;
        case 7: { return 16.0f; }; break;
    }
    return 0;
}


//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FUNÇÃO INITGU
//Define os parâmetros da interface
```

```
void InitGU( void )
{
	// Init GU
	sceGuInit();
	sceGuStart( GU_DIRECT, dList );

	// Set Buffers
	sceGuDrawBuffer( GU_PSM_8888, fbp0, BUF_WIDTH );
	sceGuDispBuffer( SCR_WIDTH, SCR_HEIGHT, (void*)0x88000, BUF_WIDTH);
	sceGuDepthBuffer( (void*)0x110000, BUF_WIDTH);

	sceGuOffset( 2048 - (SCR_WIDTH/2), 2048 - (SCR_HEIGHT/2));
	sceGuViewport( 2048, 2048, SCR_WIDTH, SCR_HEIGHT);
	sceGuDepthRange( 65535, 0);

	// setup texture
	// 32-bit image, if we swizzled the texture will return true,
otherwise false
	sceGuTexMode( GU_PSM_8888, 0, 0, texture.Swizzled() );
	sceGuTexFunc( GU_TFX_MODULATE, GU_TCC_RGB );
	sceGuTexFilter( GU_LINEAR, GU_LINEAR );        // Linear filtering
	sceGuTexScale( 1.0f, 1.0f );                    // No scaling
	sceGuTexOffset( 0.0f, 0.0f );

	// Set Render States
	sceGuScissor( 0, 0, SCR_WIDTH, SCR_HEIGHT);
	sceGuEnable( GU_SCISSOR_TEST );
	sceGuDepthFunc( GU_GEQUAL );
	sceGuEnable( GU_TEXTURE_2D );
	sceGuFrontFace( GU_CW );
	sceGuShadeModel( GU_SMOOTH );
	sceGuEnable( GU_CULL_FACE );
	sceGuEnable( GU_CLIP_PLANES );
	sceGuFinish();
	sceGuSync(0,0);
	sceDisplayWaitVblankStart();
	sceGuDisplay(GU_TRUE);  // finish
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
//FUNCOES AUXILIARES
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------

//FUNCAO DESENHA MENU
//Desenha o Menu de Opcoes
void DesenhaMenu(){
    (...) Descrita na seção 6.4.4
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
//FUNCAO DESENHA
//Lê o conteudo da matriz do tabuleiro e desenha as posicoes na tela
void Desenha(Tabuleiro tab){
    (...) Descrita na seção 6.4.4
}
```

```
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FUNCAO DESENHA GAME OVER
//Desenha a tela de gameOver
void DesenhaGameOver(){
    (...) Descrita na seção 6.4.4
}

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FUNCAO DESENHA QUIT SCREEN
//Desenha a tela que pede a confirmação de saída ao jogador
void DesenhaQuitScreen(){
    (...) Descrita na seção 6.4.4
}

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FUNCAO TECLADOJOGO
//Verifica se algum botão foi pressionado durante o jogo e realiza sua
função
void TecladoJogo(Tabuleiro &tab){
    (...) Semelhante à função TecladoMenu, porém são atribuídas ações diferentes para
cada botão pressionado.
}


//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FUNCAO TECLADOMENU
//Verifica se algum botão foi pressionado no menu e realiza sua função
void TecladoMenu(){
      sceCtrlReadBufferPositive(&pad, 1);

      if (pad.Buttons != 0){
            if (pad.Buttons & PSP_CTRL_CROSS){
                opcaoMenu = cursorMenu;
            }
            if (pad.Buttons & PSP_CTRL_UP){
                if (cursorMenu >= 2){
                    cursorMenu--;
                }
            }
            if (pad.Buttons & PSP_CTRL_DOWN){
                if (cursorMenu <= 2){
                    cursorMenu++;
                }
            }
            if (pad.Buttons & PSP_CTRL_START){
                credits = false;
            }
      }
}


//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FUNCAO INICIALIZA DADOS
//Define os parâmetros iniciais do jogo
```

```
void InicializaDados(){
    linhaAtual = 7;
    colunaAtual = 7;
    linhaInicial = -1;
    colunaInicial = -1;
    linhaFinal = -1;
    colunaFinal = -1;

    jogador = 1;
    opcaoMenu = 0;
    cursorMenu = 1;

    help = false;
    gameExit = false;

      // Inicializa as variáveis usadas para alterar a posição do
observador virtual
      rotX = 0.70f;
      rotY = -0.30f;
      obsX = 8.0f;
      obsY = 2.0f;
      obsZ = 30.0f;
}

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//FUNCAO PRINCIPAL
//Inicia as variáveis e o loop do jogo
int main() {
      scePowerSetClockFrequency(266,266,133);
      SetupCallbacks();
      sceCtrlSetSamplingCycle(0);
      sceCtrlSetSamplingMode(PSP_CTRL_MODE_ANALOG);

      pspAudioInit();

      InicializaDados();

      dList = memalign( 16, 640 );
      fbp0  = 0;

      //Carrega as Imagens do Menu
      texNG.LoadTGA("menu_new.tga");
      texNG.Swizzle();
```

(...) Mesmo procedimento para todas as outras texturas

```
    InitGU();
    fbp0 = sceGuSwapBuffers();

    while(1){

        InicializaDados();

        MP3_Init(1);
        MP3_Load("intro.mp3");
        MP3_Play();

        DesenhaMenu();
```

```
        while (opcaoMenu == 0){
            TecladoMenu();
            DesenhaMenu();
            sleep(90000);
        }

        switch(opcaoMenu){
            case 1:{   //NEW GAME
                MP3_Stop();
                MP3_FreeTune();

                //Carrega a musica do jogo
                MP3_Init(1);
                MP3_Load("test.mp3");
                MP3_Play();

                InicializaDados();

                Tabuleiro newGameTab;
                IA ia;

                Desenha(newGameTab);
                  while( gameExit == false )
                  {
                    //Executa o jogo
                    if(newGameTab.fimDeJogo() == false){
                        if (jogador == 1){
                            TecladoJogo(newGameTab);
                            Desenha(newGameTab);
                            if(resetCursor == 1){
                                   linhaInicial = -1;
                                   colunaInicial = -1;
                                   linhaFinal = -1;
                                   colunaFinal = -1;
                                   resetCursor = 0;
                            }
                            sleep(85000);
                        }else{
                            int vetor[4];
                            int randValue = rand()%10;
                            if(randValue != 1){
                                ia.realizaJogadaCPU(newGameTab, vetor);
                            }else{
                                ia.realizaJogadaCPURandom(newGameTab,
vetor); //Realiza uma jogada aleatória com chance de 10%
                            }
                            int tempLfin = linhaAtual;
                            int tempCfin = colunaAtual;

                            linhaInicial = vetor[0];
                            colunaInicial = vetor[1];
                            linhaAtual = vetor [2];
                            colunaAtual = vetor [3];

                            newGameTab.realizaJogada(vetor[0], vetor[1],
vetor[2], vetor[3], 2);
                            Desenha(newGameTab);
                            sleep(1024000);
```

```
                        linhaInicial = -1;
                        colunaInicial = -1;
                        linhaAtual = tempLfin;
                        colunaAtual = tempCfin;
                        jogador = 1;
                    }
                }else{
                    DesenhaGameOver();
                    gameExit = true;
                }
                if (MP3_EndOfStream() == 1) {
                      MP3_Init(1);
                      MP3_Load("test.mp3");
                      MP3_Play();
                }
            }
        }; break;
        case 2:{ //LOAD GAME   ***Ainda não implementado

        }; break;
        case 3:{ //MULTIPLAYER
```

(...) Semelhante ao modo new game, porém ao invés de executar uma ação com base no retorno da IA, utiliza a função teclado para realizar a jogada do jogador 2.

```
        }; break;
    }
    MP3_Stop();
    MP3_FreeTune();
}

sceGuTerm();  //Encerra a unidade grafica
free( dList );    // Libera a memoria
free( fbp0 );

sceKernelSleepThread();

return 0;
}
```

A.2 Arquivo “tabuleiro.h”

```
//Este arquivo define as regras de movimentação do jogo e cria a matriz
do tabuleiro.

#ifndef TABULEIRO_H
#define TABULEIRO_H

using namespace std;

class Tabuleiro {

          //todos os itens da classe serão publicos visando facilidade
de acesso por meio do arquivo .cpp
          public:
```

```
            //métodos
            Tabuleiro() ;
            Tabuleiro(int tab[8][8]);
            bool validaJogada(int lIni, int cIni, int lFin, int cFin, int
jogador);
            void realizaJogada(int lIni, int cIni, int lFin, int cFin, int
jogador);
            void realizaJogadaTeste(int lIni, int cIni, int lFin, int
cFin, int jogador);
            int retornaValorArmazenado(int linha, int coluna);
            int retornaValorDadoDefesa();
            int retornaValorDadoAtaque();
            bool lancaDados();
            bool buscaPeca(int matriz[8][8], int peca);
            bool fimDeJogo();



            //atributos


            int tabuleiro[8][8];
            //possui valores:
                    //  0 para posições validas livres

                    //  1 para posições com Bandeira  do jogador 1
                    //  2 para posicões com Soldado   do jogador 1
                    //  3 para posições com Arqueiro  do jogador 1
                    //  4 para posições com Cavaleiro do jogador 1
                    //  5 para posições com Rei       do jogador 1

                    //  6 para posições com Bandeira  do jogador 2
                    //  7 para posições com Soldado   do jogador 2
                    //  8 para posições com Arqueiro  do jogador 2
                    //  9 para posições com Cavaleiro do jogador 2
                    // 10 para posições com Rei       do jogador 2

            int dadoDefesa;
            int dadoAtaque;

            int qtdePecasJogador1;
            int qtdePecasJogador2;
            int gameOver;

};
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//CONSTRUTOR
Tabuleiro::Tabuleiro(){
             this->gameOver = 0;

             for (int i = 0; i <= 7; i++){
                 for (int j = 0; j <= 7; j++){
                     this->tabuleiro[i][j] = 0;
                 }
             }

             //Coloca as peças nas suas posições iniciais
             this->tabuleiro[0][2] = 9;
             this->tabuleiro[0][3] = 10;
```

```
            this->tabuleiro[0][4] = 6;
            this->tabuleiro[0][5] = 8;
            this->tabuleiro[1][2] = 7;
            this->tabuleiro[1][3] = 7;
            this->tabuleiro[1][4] = 7;
            this->tabuleiro[1][5] = 7;

            this->tabuleiro[7][2] = 3;
            this->tabuleiro[7][3] = 1;
            this->tabuleiro[7][4] = 5;
            this->tabuleiro[7][5] = 4;
            this->tabuleiro[6][2] = 2;
            this->tabuleiro[6][3] = 2;
            this->tabuleiro[6][4] = 2;
            this->tabuleiro[6][5] = 2;


            this->qtdePecasJogador1 = 8;
            this->qtdePecasJogador2 = 8;


            this->dadoDefesa = 0;
            this->dadoAtaque = 0;

            srand(time(NULL));
}

//Construtor Paramétrico
Tabuleiro::Tabuleiro(int tab[8][8]){
            this->gameOver = 0;


            for (int i = 0; i <= 7; i++){
                for (int j = 0; j <= 7; j++){
                    this->tabuleiro[i][j] = tab[i][j];
                }
            }

            this->qtdePecasJogador1 = 0;
            this->qtdePecasJogador2 = 0;

            for (int i = 0; i <= 7; i++){
                for (int j = 0; j <= 7; j++){
                    if((this->tabuleiro[i][j] < 6) && (this-
>tabuleiro[i][j] != 0)){
                        this->qtdePecasJogador1++;
                    }else{
                        this->qtdePecasJogador2++;
                    }
                }
            }


            this->dadoDefesa = 0;
            this->dadoAtaque = 0;

            srand(time(NULL));
}
```

```
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//METODOS DE ACESSO
int Tabuleiro::retornaValorArmazenado(int linha, int coluna){
    return this->tabuleiro[linha][coluna];
}

int Tabuleiro::retornaValorDadoDefesa(){
    return this->dadoDefesa;
}

int Tabuleiro::retornaValorDadoAtaque(){
    return this->dadoAtaque;
}

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FUNCAO QUE ATRIBUI VALORES ALEATORIOS PARA OS DADOS
bool Tabuleiro::lancaDados(){
    this->dadoDefesa = rand()%6 + 1;
    this->dadoAtaque = rand()%6 + 1;
    if (this->dadoDefesa >= this->dadoAtaque){
        return false;
    }
    return true;
}

//-------------------------------------------------------------------------
//FUNÇÃO VALIDAJOGADA
//Testa se a jogada é valida, analizando a posicao inicial e final dadas
pelo jogador
bool Tabuleiro::validaJogada(int lIni, int cIni,int lFin,int cFin, int
jogador){
    int conteudoIni = this->tabuleiro[lIni][cIni];
    int conteudoFin = this->tabuleiro[lFin][cFin];

    //Se posicao inicial é diferente da posicao final
    if ((lIni != lFin) || (cIni != cFin)){
        //Se posicao inicial possui uma peça do jogador atual que não
seja a bandeira
        if(((jogador == 1) && ((conteudoIni >= 2) && (conteudoIni <= 5)))
|| ((jogador == 2) && ((conteudoIni >= 7) && (conteudoIni <= 10)))){
                //Se a peca selecionada é um Cavaleiro
                if((conteudoIni == 4) || (conteudoIni == 9)){
                    //Se posicao final é no maximo 2 casas alem
                    if(((lFin <= (lIni + 2)) && (cFin <= (cIni + 2))) &&
((lFin >= (lIni - 2)) && (cFin >= (cIni - 2)))){
                        //Se posicao final é uma casa vazia
                        if (conteudoFin == 0){

                            return true;
                        }else{
                            //Se posicao final é uma peca inimiga que nao
seja bandeira
                            if(((jogador == 2) && ((conteudoFin >= 2) &&
(conteudoFin <= 5))) || ((jogador == 1) && ((conteudoFin >= 7) &&
(conteudoFin <= 10)))){
                                //Se posicao final é no maximo uma casa
alem
```

```
                                if(((lFin <= (lIni + 1)) && (cFin <=
(cIni + 1))) && ((lFin >= (lIni - 1)) && (cFin >= (cIni - 1)))){

                                    return true;
                                }
                            }
                        }
                    //Se a peca selecionada e um Arqueiro
                    }else if((conteudoIni == 3) || (conteudoIni == 8)){
                        //Se posicao final é no maximo 3 casas alem
                        if(((lFin <= (lIni + 2)) && (cFin <= (cIni + 2))) &&
((lFin >= (lIni - 2)) && (cFin >= (cIni - 2)))){
                            //Se posicao final é uma peca inimiga que nao
seja bandeira
                            if(((jogador == 2) && ((conteudoFin >= 2) &&
(conteudoFin <= 5))) || ((jogador == 1) && ((conteudoFin >= 7) &&
(conteudoFin <= 10)))){

                                return true;
                            //Se posicao final é uma casa vazia
                            }else if (conteudoFin == 0){
                                //Se posicao final é no maximo 1 casa alem
                                if(((lFin <= (lIni + 1)) && (cFin <= (cIni +
1))) && ((lFin >= (lIni - 1)) && (cFin >= (cIni - 1)))){

                                    return true;
                                }
                            }
                        }
                    //Se a peca selecionada e um Soldado
                    }else if((conteudoIni == 2) || (conteudoIni == 7)){
                        //Se posicao final é no maximo 1 casa alem
                        if(((lFin <= (lIni + 1)) && (cFin <= (cIni + 1))) &&
((lFin >= (lIni - 1)) && (cFin >= (cIni - 1)))){
                            //Se posicao final é uma peca inimiga que nao
seja bandeira
                            if(((jogador == 2) && ((conteudoFin >= 2) &&
(conteudoFin <= 5))) || ((jogador == 1) && ((conteudoFin >= 7) &&
(conteudoFin <= 10)))){

                                return true;
                            //Se posicao final é uma casa vazia
                            }else if (conteudoFin == 0){

                                return true;
                            }
                        }
                    //Se a peca selecionada e um Rei
                    }else if((conteudoIni == 5) || (conteudoIni == 10)){
                        //Se posicao final é no maximo 1 casa alem
                        if(((lFin <= (lIni + 1)) && (cFin <= (cIni + 1))) &&
((lFin >= (lIni - 1)) && (cFin >= (cIni - 1)))){
                            //Se posicao final é uma peca inimiga, inclusive
bandeira
                            if(((jogador == 2) && ((conteudoFin >= 1) &&
(conteudoFin <= 5))) || ((jogador == 1) && ((conteudoFin >= 6) &&
(conteudoFin <= 10)))){

                                return true;
```

```
                        //Se posicao final é uma casa vazia
                        }else if (conteudoFin == 0){

                            return true;
                        }
                    }
                }
            }
        }
        return false;
}
//-------------------------------------------------------------------------
//FUNCAO REALIZA JOGADA
//Realiza uma jogada em um tabuleiro
void Tabuleiro::realizaJogada(int lIni, int cIni,int lFin,int cFin, int
jogador){
    int conteudoIni = this->tabuleiro[lIni][cIni];
    int conteudoFin = this->tabuleiro[lFin][cFin];
    if (conteudoFin == 0){
        this->tabuleiro[lFin][cFin] = conteudoIni;
        this->tabuleiro[lIni][cIni] = 0;
    }else{
        if(lancaDados() == true){ //Se o jogador que ataca vencer nos
dados
            this->tabuleiro[lFin][cFin] = 0;
            if(jogador == 1){
                this->qtdePecasJogador2--;
            }else{
                this->qtdePecasJogador1--;
            }
        }
    }
}
//-------------------------------------------------------------------------
//FUNCAO REALIZAJOGADATESTE
//Realiza uma jogada de teste no tabuleiro, sem considerar se haverá
vitoria ou derrota nos dados
void Tabuleiro::realizaJogadaTeste(int lIni, int cIni,int lFin,int cFin,
int jogador){
    int conteudoIni = this->tabuleiro[lIni][cIni];
    int conteudoFin = this->tabuleiro[lFin][cFin];
    if (conteudoFin == 0){
        this->tabuleiro[lFin][cFin] = conteudoIni;
        this->tabuleiro[lIni][cIni] = 0;
    }else{
        this->tabuleiro[lFin][cFin] = 0;
    }
}
//-------------------------------------------------------------------------
//FUNCAO BUSCA PECA
//Verifica se uma peça ainda está no tabuleiro
bool Tabuleiro::buscaPeca(int matriz[8][8], int peca){
    for(int i=0; i < 8; i++){
        for(int j=0; j<8; j++){
            if(matriz[i][j] == peca) return true;
        }
    }
    return false;
}
```

```
//-------------------------------------------------------------------------
//FUNCAO FIMDEJOGO
//Verifica se as condições de fim de jogo já foram atendidas
bool Tabuleiro::fimDeJogo(){
    if ((buscaPeca(this->tabuleiro, 1) == false) || (buscaPeca(this-
>tabuleiro, 6) == false)) return true;
    if ((qtdePecasJogador1 == 1) || (qtdePecasJogador2 == 1)) return
true;
    return false;
}

#endif
```

A.3 Arquivo “ia.h”

```
//Este arquivo controla a IA. Possui uma funcao que retorna a jogada do
//computador quando solicitado.

#ifndef IA_H
#define IA_H

using namespace std;

class IA {
          //todos os itens da classe serão publicos visando facilidade
de acesso por meio do arquivo .cpp
          public:

          //métodos

          IA() ;
          void realizaJogadaCPURandom(Tabuleiro &tab, int vetor[4]);
          void realizaJogadaCPU(Tabuleiro &tab, int vetor[4]);
          int alfaBeta(Tabuleiro &tabcopia, int jogador, int altura, int
alfa, int beta);
          int calculaPeso(Tabuleiro &tab, int jogador);
          bool existeAlgumaJogada(Tabuleiro &tab, int jogador);

          int x;

};
//Construtor
IA::IA(){
    this->x = 0;
    srand(time(NULL));
}


//Realiza uma jogada aleatória
void IA::realizaJogadaCPURandom(Tabuleiro &tab, int vetor[4]){
    int linhaI = 0;
    int colunaI = 0;
    int linhaF = 0;
    int colunaF = 0;
    while(tab.validaJogada(linhaI,colunaI,linhaF,colunaF,2) == false){
        linhaI = rand()%8;
        colunaI = rand()%8;
        linhaF = rand()%8;
```

```
        colunaF = rand()%8;
    }
    vetor[0] = linhaI;
    vetor[1] = colunaI;
    vetor[2] = linhaF;
    vetor[3] = colunaF;
}

//Usa o algoritmo Minimax com Poda Alfa-beta para definir qual sera sua
jogada
void IA::realizaJogadaCPU(Tabuleiro &tab, int vetor[4]){
    int linhaI, colunaI, linhaF, colunaF;

      int alfa = -99999;

      for(int i = 0; i < 8; i++){
            for(int j = 0; j < 8; j++){
              for(int k = 0; k < 8; k++){
                  for(int l = 0; l < 8; l++){
                        //Verifica se (i, j, k, l) é uma jogada válida
para o computador, copia o tabuleiro e simula a jogada para avaliar qual
delas será a melhor utilizando o algoritmo MiniMax com poda AlfaBeta
                        if(tab.validaJogada(i, j, k, l, 2) == true){
                              int valorAtual;
                        Tabuleiro tabcopia(tab.tabuleiro);


                        tabcopia.realizaJogadaTeste(i, j, k, l, 2);
                              valorAtual = alfaBeta(tabcopia, 1, 0, alfa,
99999);


                              //Atualiza a melhor jogada até o momento
                              if(alfa < valorAtual){
                                    linhaI = i;
                                    colunaI = j;
                                linhaF = k;
                                colunaF = l;
                                    alfa = valorAtual;
                              }
                        }
                  }
              }
            }
      }

    vetor[0] = linhaI;
    vetor[1] = colunaI;
    vetor[2] = linhaF;
    vetor[3] = colunaF;
}

//Algoritmo de Poda Alfa-beta
int IA::alfaBeta(Tabuleiro &tabcopia, int jogador, int altura, int alfa,
int beta){
      altura++;

      //Teste de parada da recursão, ocorre quando se alcança o último
nível ou
      //se não existem mais jogadas possíveis
```

```
	if((altura > 3) || (!existeAlgumaJogada(tabcopia, jogador)))
		//Retorna o peso total das casas que o jogador possui, na simulação da jogada
		return calculaPeso(tabcopia, jogador);
	//Testa se a recursão fará o "Min" ou o "Max"
	if(jogador == 2){
		for(int i = 0; i < 8; i++)
			for(int j = 0; j < 8; j++)
				for(int k = 0; k < 8; k++)
					for(int l = 0; l < 8; l++)
						//Verifica se (i, j, k, l) é uma jogada válida para a pessoa, copia o tabuleiro e simula a jogada chamando a recursão novamente
						if(tabcopia.validaJogada(i, j, k, l, 1)){
							Tabuleiro tabcopia2(tabcopia.tabuleiro);

							tabcopia2.realizaJogadaTeste(i, j, k, l, 1);
						int valorAtual = alfaBeta(tabcopia2, 1, altura, alfa, beta);
							//Max
							if(valorAtual > alfa)
								alfa = valorAtual;
							if(alfa >= beta)
								return beta;
						}
		return alfa;
	} else {
		for(int i = 0; i < 8; i++)
			for(int j = 0; j < 8; j++)
				for(int k = 0; k < 8; k++)
					for(int l = 0; l < 8; l++)
						//Verifica se (i, j, k, l) é uma jogada válida para o computador, copia o tabuleiro e simula a jogada chamando a recursão novamente
						if(tabcopia.validaJogada(i, j, k, l, 2)){
							Tabuleiro tabcopia2(tabcopia.tabuleiro);

							tabcopia2.realizaJogadaTeste(i, j, k, l, 2);

							int valorAtual = alfaBeta(tabcopia2, 2, altura, alfa, beta);



							//Min
							if(valorAtual < beta)
								beta = valorAtual;
							if(alfa >= beta)
								return alfa;
						}
		return beta;
	}
}
//Função que calcula o peso total das pecas do tabuleiro para um jogador
int IA::calculaPeso(Tabuleiro &tab, int jogador){
	int peso = 0;
```

```
    for(int i = 0; i < 8; i++){
        for(int j = 0; j < 8; j++){
            if(jogador == 1){
                    switch(tab.tabuleiro[i][j]){
                        case 0: {
                            peso = peso;
                        }; break;
                        case 2: {
                            peso += (8 - i);
     //se possui um soldado, +1, +valor correspondente à linha atual
                        }; break;
                        case 7: {
                            peso -= (5 + i);
    //se oponente possui um soldado, -5, -valor correspondente à linha
atual
                        }; break;
                        case 1: {
                            peso += 5;
     //se possui sua bandeira, +5
                        }; break;
                        case 6: {
                            peso -= 10;
    //se oponente possui sua bandeira, -10
                        }; break;
                        case 3: {
                            peso += (17 - i);
    //se possui um arqueiro, +10, +valor correspondente à linha atual
                        }; break;
                        case 4: {
                            peso += (17 - i);
    //se possui um cavaleiro, +10, +valor correspondente à linha atual
                        }; break;
                        case 5: {
                            peso += (27 - i);
    //se possui um rei, +20, +valor correspondente à linha atual
                        }; break;
                        case 8: {
                            peso -= (15 + i);
   //se oponente possui seu arqueiro, -15, -valor correspondente à linha
atual
                        }; break;
                        case 9: {
                            peso -= (15 + i);
    //se oponente possui seu cavaleiro, -15, -valor correspondente à
linha atual
                        }; break;
                        case 10: {
                            peso -= (20 + i);
  //se oponente possui seu rei, -20, -valor correspondente à linha atual
                        }; break;
                    }
            }else{
                    switch(tab.tabuleiro[i][j]){
                        case 0: {
                            peso = peso;
                        }; break;
                        case 7: {
                            peso += (1 + i);
     //se possui um soldado, +1, +valor correspondente à linha atual
                        }; break;
```

```
                        case 2: {
                            peso -= (12 - i);
    //se oponente possui um soldado, -5, -valor correspondente à linha
atual
                        }; break;
                        case 6: {
                            peso += 5;
     //se possui sua bandeira, +5
                        }; break;
                        case 1: {
                            peso -= 10;
    //se oponente possui sua bandeira, -10
                        }; break;
                        case 8: {
                            peso += (10 + i);
    //se possui um arqueiro, +10, +valor correspondente à linha atual
                        }; break;
                        case 9: {
                            peso += (10 + i);
    //se possui um cavaleiro, +10, +valor correspondente à linha atual
                        }; break;
                        case 10: {
                            peso += (20 + i);
   //se possui um rei, +20, +valor correspondente à linha atual
                        }; break;
                        case 3: {
                            peso -= (22 - i);
   //se oponente possui seu arqueiro, -15, -valor correspondente à linha
atual
                        }; break;
                        case 4: {
                            peso -= (22 - i);
    //se oponente possui seu cavaleiro, -15, -valor correspondente à
linha atual
                        }; break;
                        case 5: {
                            peso -= (27 - i);
   //se oponente possui seu rei, -20, -valor correspondente à linha atual
                        }; break;
                    }
                }
            }
    }
    return peso;
}

//Função que verifica se ainda existem jogadas válidas para o jogador do
turno
bool IA::existeAlgumaJogada(Tabuleiro &tab, int jogador){
     for(int i = 0; i < 8; i++)
          for(int j = 0; j < 8; j++)
               for(int k = 0; k < 8; k++)
                 for(int l = 0; l < 8; l++)
                           if (tab.validaJogada(i, j, k, l, jogador))
                           return true;

      return false;
}

#endif
```

ANEXO B – Regras do Jogo

Regras do Jogo

O jogo é de estratégia, baseado em turnos, e cada jogador pode movimentar um personagem por turno em um cenário estruturado como um tabuleiro.

Existem cinco tipos de personagens, listados a seguir:

- Rei: Se move uma casa por vez, é o único que pode destruir a bandeira, o alcance de seu ataque é de uma casa.
- Bandeira: Não pode se mover nem atacar; só pode ser atacada por um Rei.
- Cavaleiro: Pode mover-se até duas casas por vez, porém só ataca a uma casa de distância.
- Soldado: Se move uma casa por vez e ataca a uma casa de distância.
- Arqueiro: Se move uma casa por vez, porém pode atacar um personagem que esteja até duas casas de distância.

Os alcances das peças acima são válidos em qualquer direção (vertical, horizontal ou diagonal).

O objetivo de cada jogador é destruir a bandeira inimiga ou eliminar todas as peças do adversário (exceto a bandeira), lembrando que somente o Rei pode atacar a bandeira, logo, caso o seu Rei seja eliminado, o objetivo passa a ser exclusivamente eliminar todas as peças inimigas.

Cada jogador só pode realizar uma ação por turno, ou seja, um movimento ou um ataque. Um ataque pode ser realizado quando o oponente se encontrar em uma casa na área de alcance do jogador. Neste caso, são lançados dados de ataque e defesa numerados de um a seis. O dado que obtiver o maior resultado vence, e em caso de empate, a vantagem é da defesa. Se o ataque for realizado, a peça inimiga é eliminada, e a que atacou permanece na mesma posição, já se a defesa vence, apenas termina o turno do jogador que fez o ataque.

Comandos do Jogador

Durante a execução do jogo, cada botão do console PSP exerce uma função específica, como mostra a tabela a seguir:

	Seleciona uma peça ou confirma ação do cursor
	Liga/Desliga a música de fundo
	Cancela uma seleção efetuada
	--
L	Afasta a câmera
R	Aproxima a câmera
	Movimenta o cursor
	Movimenta a câmera
START	Pausa o jogo
SELECT	Abre/Fecha o *HELP*

Tabela 1 - Comandos Básicos

REFERÊNCIAS BIBLIOGRÁFICAS

ABRAGAMES. Disponível em http://www.abragames.org. Consultado em: 12 Mai. 2008.

BATTAIOLA, ANDRÉ L. Jogos por Computador – Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação In: **XIX Jornada de Atualização em Informática**. Curitiba:SBC, Julho/2000, v. 2. pp. 83-122.

BLOODSHED. **Dev C++.** Disponível em http://www.bloodshed.net/devcpp.html. Consultado em 05 Jun. 2008.

CLUA, ESTEBAN W.G.; BITTENCOURT, JOÃO R. Desenvolvimento de Jogos 3D: Concepção, Design e Programação In: **XXV Congresso da Sociedade Brasileira de Computação**. São Leopoldo/RS: SBC, Julho/2005, pag 1313-1358.

COHEN, M; MANSSOUR, I, H. **OpenGL Uma abordagem prática e objetiva**. São Paulo, Ed. Novatec, 2006, 487p.

FITZHARDINGE, J. **PSPGL**. Disponível em http://www.goop.org/psp/gl/. Consultado em 20 Mai. 2008.

GNU Project. **GCC, The GNU Compiler Collection.** Disponível em http://gcc.gnu.org/. Consultado em 15 Mai. 2008.

GUITAR-PRO. Disponível em http://www.guitar-pro.com/en/index.php/. Consultado em 18 Mai. 2008.

IDG NOW!. Indústria de games movimenta US$ 9,5 bilhões durante 2007, diz NPD. Disponível em http://idgnow.uol.com.br/computacao_pessoal/2008/01/25/industria-de-games-movimenta-us-9-5-bilhoes-durante-2007-diz-npd/. Consultado em: 12 Mai. 2008.

JUNG, C. F. **Metodologia para pesquisa & desenvolvimento: aplicada a novas tecnologias, produtos e processos**. Rio de Janeiro/RJ: Axcel Books do Brasil Editora, 2004.

MICROSOFT. **Microsoft Windows.** Disponível em http://www.microsoft.com/windows/. Consultado em 20 Jun. 2008

PSP-PROGRAMMING. **PSPGU Tutorial.** Disponível em http://www.psp-programming.com/code/doku.php?id=c:pspgu-neheport-lesson5. Consultado em 15 Mai. 2008

RUSSELL, S; NORVIG, P. **Inteligência Artificial**. Rio de Janeiro, Ed. Campus, 2004, 1040p

SONY COMPUTER ENTERTAINMENT. **Playstation Portable.** Disponível em http://www.us.playstation.com/psp/. Consultado em 20 Jun. 2008.

WRIGHT, R, S, Jr; SWEET, M. **OpenGL SuperBible.** Waite Group Press, 2nd Ed. Indianapolis, Indiana, 2000.

XORLOSER. **PSPDev for Win32.** Disponível em http://xorloser.com/. Consultado em 05. Mai. 2008