GIULIANO CARVALHO DE OLIVEIRA

GUSTAVO MATSUMOTO ROSENDO DOS SANTOS

RAFAEL TADEU AYRES

ROBERTO SOARES CALDAS

Desenvolvimento de um jogo de estratégia em tempo real utilizando como metodologia o Processo Unificado

Orientadores: Romero Tori
João Luiz Bernardes Jr.

São Paulo

2005

GIULIANO CARVALHO DE OLIVEIRA

GUSTAVO MATSUMOTO ROSENDO DOS SANTOS

RAFAEL TADEU AYRES

ROBERTO SOARES CALDAS

Desenvolvimento de um jogo de estratégia em tempo real utilizando como metodologia o Processo Unificado

Projeto de Formatura apresentado à disciplina PCS 2050 – Laboratório de Projeto de Formatura II, da Escola Politécnica da Universidade de São Paulo.

Orientadores: Romero Tori
João Luiz Bernardes Jr.

São Paulo

2005

FOLHA DE APROVAÇÃO

Giuliano Carvalho de Oliveira
Gustavo Matsumoto Rosendo dos Santos
Rafael Tadeu Ayres
Roberto Soares Caldas

Projeto de Formatura apresentado à disciplina PCS 2050 – Laboratório de Projeto de Formatura II, da Escola Politécnica da Universidade de São Paulo.

Aprovado em:

Banca Examinadora

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

DEDICATÓRIA

Aos amigos,

por me apoiarem nesta jornada.

Aos companheiros Giuliano, Gustavo e Roberto,

pelo valoroso trabalho em equipe e comprometimento.

Aos irmãos Fabio, Marcus e Guilherme,

pelo exemplo, compreensão e afeto.

Aos meus pais Edno e Meire,

pela educação baseada na honra e na ética.

Rafael Tadeu Ayres

DEDICATÓRIA

Aos meus amigos que aceitaram fazer este projeto comigo, realizando um sonho, e principalmente à Clarissa, meus pais, Adarica e Sérgio, e tios Roberto e Oscar, por todo amor e suporte.

Roberto Soares Caldas

DEDICATÓRIA

Aos meus pais, por me ensinarem o valor do esforço. Aos meus irmãos, por me ensinarem o valor do respeito e da lealdade. Aos meus amigos Gustavo, Rafael e Roberto, pela perseverança e espírito de equipe.

Giuliano Carvalho de Oliveira

DEDICATÓRIA

Dedico este projeto de formatura aos meus pais, Hélio e Rita, por seu imenso amor, carinho, compreensão e apoio durante toda a minha vida e nestes anos de faculdade. E ao meu tio, Silvio, por sua inestimável amizade e incentivo, de valor imensurável ao longo destes anos.

Gustavo Matsumoto Rosendo dos Santos

AGRADECIMENTOS

Aos nossos orientadores, Prof. João Luiz Bernardes Jr. e Prof. Dr. Romero Tori, pela dedicação, orientação e auxílio dados durante o desenvolvimento deste trabalho.

Ao Prof. Dr. Jorge Risco Becerra, pela coordenação e acompanhamento dos projetos de formatura e pelas lições aprendidas em sala de aula.

Aos professores do PCS, que nos transmitiram os conhecimentos necessários para nossa sólida formação como Engenheiros de Computação, por sua dedicação e disponibilidade.

À Escola Politécnica da Universidade de São Paulo, por tudo o que nos proporcionou aprender e por todo o amadurecimento pessoal e profissional adquirido neste período de graduação.

À Deus, por sua benção durante todos esses anos.

RESUMO

O mercado de entretenimento tem despertado grande interesse, pois apesar da pouca divulgação, é o que mais cresce atualmente. Segundo pesquisas de instituições renomadas, o faturamento do segmento de jogos eletrônicos já ultrapassou o do cinema, chegando a atingir dezenas de bilhões de dólares em 2004, e com previsão de crescimento para os próximos anos. Este trabalho propõe a criação de uma versão demonstrativa de um jogo para computador do tipo "estratégia em tempo real", utilizando o Processo Unificado como metodologia de Engenharia de Software. Foi escolhida a engine gráfica gratuita OGRE (Object Oriented Graphics Rendering Engine) como solução para implementação dos recursos gráficos, pois possui grande aceitação e suporte além de possuir os requisitos necessários para implementação do projeto. Basicamente, a arquitetura consiste em um módulo principal que controla os elementos do jogo, tais como os jogadores e unidades (personagens controlados pelos jogadores), um módulo de interface com o mouse e teclado, um módulo de pathfinding que gera rotas para as unidades, um módulo de inteligência artificial para controlar o jogador computador e, por fim, um módulo de overlay, que gera menus para interface com o jogador. O Processo Unificado divide o projeto de software em fases, que podem ser divididas em uma ou mais iterações. O resultado de cada iteração é a geração de um mini-projeto, que é representado pelos seus artefatos. Os módulos acima são produzidos durante as iterações, em conjunto com todos os artefatos decorrentes da utilização da UML. Na única iteração da primeira fase, a fase de Concepção e Elaboração, foram produzidos como artefatos a descrição dos requerimentos, a descrição dos casos de uso, os requisitos não funcionais, esboço da interface e o diagrama de classes. Na primeira iteração da fase de Construção foi implementada a arquitetura básica que suporta os módulos e o módulo de interface com o mouse e teclado. Na segunda iteração foram implementados os módulos de pathfinding, iniciados os módulos de inteligência artificial e overlay, além do aprimoramento da documentação. Na terceira iteração da fase de Construção também aconteceu o aprimoramento da documentação, os módulos de inteligência artificial e overlay foram finalizados e algumas funcionalidades dos jogos também foram implementadas. Não foi possível implementar todos os casos de uso criados para a versão demonstração do jogo, porém o objetivo foi atingido: o Processo Unificado foi empregado e, portanto, tem-se uma metodologia que permite dar continuidade ao projeto, criando uma versão demonstração do jogo, evoluindo para um jogo completo e expansível.

Palavras chave: Processo Unificado, Jogos, estratégia em tempo real, computação gráfica.

ABSTRACT

The digital entertainment industry has attracted a lot of attention because, despite its modest divulgation, it's the most growing market nowadays. According to researches made by renowned institutes, the electronic games segment earnings exceeded the motion picture industry earnings, reaching in 2004 dozens of billion dollars, with growing expected for the next years. The present work proposes the creation (development) of a demonstrative version of a Real Time Strategy Computer Game, using the Unified Process as the Software Engineering methodology. The open-source graphics engine OGRE (Object Oriented Graphics Rendering Engine) was chosen as the solution for the implementation of the graphics resources by the fact of being well supported and accepted as well as having all the necessary requisites for the implementation of the project. Basically, the game architecture consists of a main module that controls the elements of the game, such as the players and units (objects that are controlled by the players), an interface module with the mouse and keyboard, a pathfinding module that generates paths for the units, an artificial intelligence module for controlling the computer player and, finally, an overlay module, which generates menus for the interface with the player. The Unified Process divides the software project into phases, which can be divided into one or more iterations. The result of each iteration is the generation of an artifact, that is, a mini-project. The modules above are artifacts yielded during the iterations, altogether with the products originated by the use of the UML. After the only iteration of the first phase, the Conception and Elaboration phase, the artifacts produced were the description of the requirements, the description of the use cases, the non-functional requisites, the outline of the user-interface and the class diagram. In the first iteration of the Construction phase, the implemented artifacts were the basic architecture that provides support for the modules and the interface module with the mouse and keyboard. In the second iteration, the Pathfinding module was implemented, the overlay and artificial intelligence modules were initiated as well as the improvement of the documentation. In the third iteration of the Construction phase the documentation was also refined, the artificial intelligence module and the overlay module were completed and some functions were developed too. It was impossible to implement all the use cases proposed for the demonstration version of the game, but the objective was reached: the Unified Process was used and, therefore, we have a methodology that permits the continuity of the project, creating a demonstration version of the game, evolving to a complete and expansive game

Keywords: Unified Process, Games, Real-time Strategy, Computer Graphics.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e SIGLAS

API	– Application Programming Interface
BFS	– Best First Search
DTI	– Department of Trade and Industry
E3	– Electronic Entertainment Expo
FSM	– Finite State Machine
FuSM	– Fuzzy State Machine
GA	– Genetic Algorithms
GUI	– Graphical User Interface
IA	–Inteligência Artificial
NPC	– Non-player character
OGRE	– Object-Oriented Graphics Rendering Engine
RTS	– Real Time Strategy
TBS	– Turn-based Strategy
TIGA	–The independent games developers association
UP	– Unified Process
XML	– Extensible Markup Language
FPS	– Frames per Second

Sumário

1 Introdução

1.1 Objetivo

Este projeto tem por objetivo desenvolver uma versão demonstrativa de um jogo de estratégia em tempo real (RTS), aplicando os conceitos do Processo Unificado (UP) como metodologia de Engenharia de Software, de tal modo que esteja voltada para este aplicativo em particular. Para isso, procura-se customizar a metodologia proposta pelo UP a fim de utilizar as ferramentas mais adequadas em detrimento de outras que não adicionariam valor concreto à especificação do projeto.

Em outras palavras, o que se pretende gerar aqui é um jogo de estratégia em tempo real, utilizando uma metodologia formal para o seu desenvolvimento.

O projeto não tem como escopo a elaboração de uma engine gráfica ou de simulação física. Para tanto, foi adotada como atividade do projeto o estudo acerca das principais engines gráficas e de simulações físicas existentes no mercado, para posterior incorporação ao projeto.

Quanto ao modo de se jogar, o escopo do projeto não engloba funcionalidades do tipo multiplayer, se restringindo à opção single-player.

Escolheu-se este tema para o projeto de formatura porque envolve uma série de assuntos abordados pela Engenharia de Computação, ou seja, tem uma característica multidisciplinar, na medida em que necessita de conhecimentos das

áreas de Inteligência Artificial, Computação Gráfica, Programação e Engenharia de Software, além da parte mais artistica de Game Design. Além disso, a área de entretenimento tem crescido muito não só no Brasil, mas no mundo, mostrando-se bastante promissora.

O jogo em si, descrito em maior profundidade no documento de Design (anexo A), contém o mapa de um território onde estarão as construções e unidades com as quais o jogador irá interagir visualmente e através do mouse e do teclado. O objetivo principal é obter o controle de todo o território derrotando seus inimigos. O jogador poderá comprar e vender produtos, contratar unidades, atacar territórios inimigos, etc. As unidades terão atributos físicos (Vida) e não-físicos (Habilidade).

1.2 Contexto

Esta seção visa tratar da questão dos jogos no mundo e no Brasil, apontando as principais políticas adotadas no mundo em relação ao mercado de jogos, bem como as principais dificuldades encontradas pelos países que se dedicam a esse tipo de atividade.

1.2.1 Mercado de jogos no Brasil e no mundo

Segundo estudo publicado recentemente pela PriceWaterhouse Coopers

apud ABRAGAMES(2004), o mercado de entretenimento é o que mais cresce no mundo.

O segmento de jogos eletrônicos já ultrapassou o faturamento do cinema, e tem previsão de crescimento de 20,1% ao ano pelos próximos 5 anos. Segundo a empresa de consultoria Informamedia apud ABRAGAMES(2004), esse segmento faturou em 2004 cerca de 50 bilhões de dólares.

O percentual de desenvolvedores de jogos espalhados pelo mundo reflete a força de suas economias, como é exibido no gráfico abaixo.

Figura 1 – Mercado de jogos no mundo

Pelo gráfico, tem-se uma parcela definida como "resto do mundo", e que se desenvolveu no mercado de jogos de duas formas distintas.

A primeira forma se deu para os países que encontraram uma situação de baixo custo de desenvolvimento dos seus jogos, devido às condições cambiais e salariais favoráveis. Nesta categoria estão as empresas do leste europeu.

Apesar do relativo sucesso na área, os países desta categoria possuem um mercado corroído pela pirataria, não possuem tradição em desenvolvimento para consoles de videogame (apenas produzem para PCs), e basicamente exportam seus produtos. Esses países acabam culturalmente prejudicados, já que não conseguem desenvolver seus jogos visando o mercado interno, fazendo com que praticamente todos os jogos não possuam temática local, nem diversidade.

A segunda forma ocorreu em países que empregaram políticas públicas de incentivo eficientes, ampliando o mercado interno e apoiando os desenvolvedores de jogos. Tal categoria é formada por países como Coréia do Sul e Austrália.

Como exemplo do sucesso vivido pelos países que se desenvolveram no mercado de jogos por meio desta forma tem-se, no caso da Coréia do Sul, a geração de 50.000 empregos no setor, num mercado gerador de 3,3 bilhões de dólares para o país (ABRAGAMES, 2004). No caso da Austrália, além de desenvolverem jogos para PC, eles desenvolvem para consoles, sendo que os consoles representam a maior parte do faturamento do mercado mundial.

Além dos países integrantes do "resto do mundo", há os tradicionais líderes do mercado, como o Reino Unido e os Estados Unidos.

O Reino Unido, apesar de possuir uma posição bem consolidada no mercado de jogos, conta com diversas medidas orientadas para o desenvolvimento contínuo da área. O *Department of Trade and Industry* (DTI) planejou, juntamente

com uma associação de desenvolvedores de jogos local (TIGA) uma série de ações para promover a sua indústria de jogos.O DTI, juntamente com a UK Trade Investment, patrocina a E3, maior feira de exposição de entretenimento eletrônico do mundo.

Os Estados Unidos, líder do mercado de jogos, anunciou um conjunto de medidas de apoio financeiro e incentivo às companhias locais.

Segundo a ABRAGAMES, uma associação brasileira de desenvolvedores independentes de jogos eletrônicos, até Setembro de 2004, havia no Brasil 40 empresas dedicadas ao desenvolvimento de jogos, sendo que 10 dessas empresas surgiram no último ano. Nesse mesmo período, havia 25 jogos em desenvolvimento e 35 jogos nacionais (apenas para PC) lançados nos 2 anos anteriores.

Atualmente, as empresas nacionais desenvolvedoras de jogos atuam em um mercado árido, sem regras e incentivos, enfrentando grandes dificuldades de popularizar os seus jogos, tanto dentro quanto fora do país.

De forma a lidar com tais adversidades, em Abril de 2004, surgiu a ABRAGAMES, dedicada a promover a cooperação e o espírito de comunidade entre os associados, desenvolver uma infra-estrutura compartilhada, promover a indústria local, e atuar junto ao governo.

Segundo ABRAGAMES (2004), as empresas nacionais atuantes no mercado de jogos possuem os seguintes modelos de negócio: advergames, jogos para celular, jogos sérios, jogos para PC e jogos voltados para o mercado externo.

O palavra inglesa advergames deriva dos termos "*advertising* " e "*games*", e esse tipo de negócio consiste em utilizar os jogos eletrônicos como ferramenta de

marketing e relacionamento com o público alvo. Utilizado por grandes anunciantes, os jogos aumentam a visibilidade e o tempo de visitação do site e promovem a marca da empresa.

Esse mercado tornou-se um dos mais cobiçados no país, mas o pouco uso dos anunciantes, justamente pelo fato de ser uma cultura ainda poucamente impregnada, impede o crescimento deste ramo.

Os jogos para celular, distribuídos pelas operadoras de telefonia (compartilhando as receitas com os desenvolvedores), são um mercado pouco rentável no Brasil. Isso se deve à falta de investimento em marketing para tornar popular o consumo desses jogos. Como vantagem, este mercado pode traduzir os seus jogos para outras línguas e enviá-los para o mercado internacional.

Nova no Brasil, a modalidade de jogos sérios consiste na produção de jogos a serem utilizados como ferramenta educativa, de treinamento ou de seleção de pessoas. Como é um mercado novo e ainda possui poucas empresas atuantes, poucos resultados foram obtidos.

A área de jogos para PC enfrenta a pirataria, que é uma grande fonte de prejuízo, existindo poucas produções, e os poucos jogos lançados não conseguem capitalizar as empresas para investidas maiores.

Considerada por muitos como a única solução atual para as produtoras brasileiras, a produção de jogos sob encomenda para o mercado externo pode contribuir para a balança comercial, mas os resultados culturais são poucos, e os resultados econômicos poderiam ser muito maiores se houvesse espaço para a propriedade intelectual própria.

O mercado internacional possui grande espaço de atuação para as empresas brasileiras, mas a concorrência é muito acirrada. Uma das poucas vantagens que as empresas nacionais levam são os baixos custos de produção em relação a alguns países do primeiro mundo. Contudo, existem outros países concorrentes que também possuem baixos custos de produção e maior tradição em desenvolvimento que o Brasil.

1.2.2 Jogos de Estratégia em Tempo Real (RTS)

Nesta seção, descreve-se os principais conceitos que envolvem os jogos RTS, bem como sua história e principais características.

1.2.2.1 Definição

Tipicamente, jogos de estratégia são de tabuleiro ou eletrônicos, onde as habilidades de tomada de decisão dos jogadores possuem grande importância no desenrolar do jogo. Muitos jogos incluem este elemento em maior ou menor grau (WIKIPEDIA, 2005).

Jogos de estratégia contrastam com jogos de sorte, pois os primeiros se baseiam principalmente na habilidade do jogador, enquanto os segundos, no puro acaso.

Um jogo de estratégia em tempo real (real-time strategy ou RTS) consiste

num tipo de jogo de estratégia onde a ação do jogo é contínua, e desta forma os jogadores têm decidir e agir num ambiente de constante mudança de estados (WIKIPEDIA, 2005).

Na realidade, os jogos RTS são uma evolução dos jogos baseados em turnos (turn-based strategy ou TBS). Estes últimos têm como característica a divisão do fluxo de ações do jogo em partes bem-definidas, chamadas turnos e *rounds*. Cada turno é reservado para a ação de um único jogador. Neste momento, o jogador coloca em prática a sua decisão, oriunda de uma análise prévia ocorrida quando ainda não era o seu turno. Quando todos os jogadores fizerem uso de seu turno, um *round* é completado.

Tanto os jogos eletrônicos TBS quanto RTS possuem elementos básicos que os constituem. Como elementos constituintes têm-se: um cenário, uma história e um conjunto de elementos para se vencer o jogo.

O cenário é o ambiente onde a história do jogo se desenrola. Todos os jogadores participantes iniciam o jogo com uma parte do cenário reservada para si.

À medida que o jogo evolui, algumas partes que antes pertenciam a certo jogador podem ser tomadas por um outro adversário.

Alguns jogos eletrônicos de estratégia acrescentam ao cenário elementos vivos que não pertencem a nenhum jogador, mas que podem influir no desenrolar do jogo.

São os chamados NPCs (ou non-player characters). Podem ser vistos nos jogos como habitantes, animais, além de outras formas, dependendo do contexto do jogo.

A história, presente em todos os jogos de estratégia, serve para definir o contexto do jogo, os papéis e os recursos de cada jogador.

Em certos jogos de estratégia, a história é dividida nas chamadas campanhas, que são uma série de pequenas missões, onde os jogadores jogam entre si num cenário com objetivos definidos, geralmente dentro do contexto da história de fundo. Freqüentemente cada missão possui um estilo diferente de se jogar.

Por fim, têm-se como fator constituinte dos jogos eletrônicos de estratégia, o conjunto de elementos básicos para se vencer o jogo. Como elementos básicos, têm-se: unidades, construções, e recursos.

A unidade é o personagem controlado pelo jogador. Em grande parte dos casos humanóide, ela possui uma série de atributos, alguns específicos de cada jogo, outros comuns a maioria dos jogos.

Como atributos comuns à maioria dos jogos, têm-se vida e função. A vida de uma unidade pode ser representada por um número positivo, e que sofre decremento em seu valor quando a unidade é atacada. Quando a vida atinge o valor zero, a unidade morre. A função da unidade indica o papel desempenhado pela mesma no contexto da história. Como exemplos, existem unidades dedicadas somente a atuar nos combates, enquanto outras se dedicam a produzir novos recursos.

Figura 2 - Exemplo de uma unidade no jogo Age of Empires III.

Uma construção consiste numa estrutura física, como um prédio ou uma casa, e possui diversas funções, tais como: proteger unidades e estocar recursos. Em certos jogos, serve também para criar unidades, ou então para produzir novos recursos.

Figura 3 - Exemplo de uma construção no jogo Age of Empires III.

Os recursos são elementos utilizados pelo jogador para expandir o seu domínio no jogo. Como exemplo, cita-se o jogo *Age of Empires*, onde a madeira é um recurso que serve para construir novas construções, e o ouro é necessário para a aquisição de novas unidades ou de outros recursos.

1.2.2.2 Uma breve história

Segundo Wikipedia (2005), O primeiro jogo RTS se chama "Stonkers", e foi criado em 1983 pelos *designers* D. H. Lawson e John Gibson e pelo artista gráfico

Paul Lindale. Contudo, o jogo que realmente definiu o estilo RTS foi o "Battle Master", em 1990.

O precursor dos jogos de estratégia modernos foi o "The Ancient Art of War", criado em 1984 por Dave Murry e Barry Murry, da Evryware.

1.2.2.3 As características dos jogos RTS

Por causa de seu natural ritmo acelerado, e em muitos casos pela sua rápida curva de aprendizado, os jogos RTS superaram a popularidade dos jogos TBS. Contudo, os adeptos desta última modalidade criticaram os jogos RTS por apresentarem uma tendência desenvolver as chamadas "festas de clique", em que os jogadores mais rápidos no uso do mouse geralmente ganhavam, já que dariam ordem às suas unidades para atacar mais rapidamente.

De modo a superar esta crítica, a maioria dos jogos RTS atuais possuem características que reduzem a importância de um trabalho de mouse rápido, possibilitando ao jogador dar enfoque maior na parte de estratégia. Como exemplo, muitos jogos dão às unidades do jogador atributos como força e fraqueza, desencorajando-os a derrotar o adversário por meio de ataques continuos simplesmente por estar em maioria de exército.

De uma forma geral, a maioria dos jogos RTS segue o seguinte padrão:

- Construa sua base e forças;
- Adquira mais recursos;

- Ataque o inimigo, tentando desprovê-lo de recursos e destruindo a sua infra-estrurura.

A maioria dos jogos também proporciona campanhas *single-player*. Uma característica que alguns jogos RTS proporcionam é a capacidade de o jogador construir bases e exércitos. Segundo Wikipedia (2005), os jogos que apresentam esta característica se dividem em duas vertentes: os jogos de micro-gerenciamento, e os de macro-gerenciamento.

Os jogos de micro-gerenciamento permitem ao jogador construir exército e base, mas eles limitam o tamanho limite do exército (muitas vezes severamente). O intuito desta medida é a criação de uma atmosfera mais tática, prevenindo que um dos lados simplesmente acumule unidades e as lance sobre o adversário até que ele entre em colapso.

Os jogos de macro-gerenciamento possuem um enfoque maior na produção econômica (em contraposição à criação somente de exércitos) e manobras estratégicas de larga escala.

1.3 Metodologia

A metodologia utilizada no projeto será o processo unificado, cuja teoria será descrita no capítulo 2 e a descrição da implementação, no capítulo 3. Vale ressaltar que o processo unificado será adaptado às particularidades do projeto,

com o intuito de se evitar que se adotem procedimentos que não contribuam de maneira relevante ao projeto, e que de alguma forma possam torná-lo mais extenso do que o necessário.

1.4 Organização do trabalho

O capítulo 1 descreve o objetivo e o escopo do projeto, bem como trata do contexto em que o projeto está inserido, com a descrição a respeito do mercado de jogos no Brasil e no mundo, e as teorias e particularidades que envolvem um jogo de estratégia em tempo real.

O capítulo 2 descreve os conceitos teóricos que envolvem o projeto, como o processo unificado, o algoritmo de *pathfinding*, técnicas de inteligência artificial e conceitos de *game design*.

O capítulo 3 descreve como, a partir dos conceitos teóricos, o projeto foi desenvolvido.

2 Fundamentos Teóricos

Nesta seção serão discutidos os fundamentos teóricos utilizados para o desenvolvimento do jogo de estratégia em tempo real, dentre eles o Processo Unificado, que foi a metodologia escolhida para o desenvolvimento deste projeto.

Outros temas abordados são: computação gráfica, *pathfinding*, inteligência artificial e *game design*. Mais adiante serão apresentadas as soluções adotadas e de que forma os conceitos aqui discutidos foram utilizados no projeto.

2.1 Metodologia de projeto

A utilização de uma metodologia clara de projeto é um dos objetivos de nosso projeto, pois, sendo um jogo de computador um projeto complexo, que exige a integração de diversos aspectos do conhecimento, era necessário uma correta coordenação dos esforços da equipe.

"Um processo de desenvolvimento de software é uma definição do conjunto completo de atividades necessárias para se transformar os requisitos do usuário em um conjunto consistente de artefatos que representam o produto de software" (Jacobson et al, 1999 p. 24).

A utilização de um processo bem definido e comum a todos dentro de uma equipe de projeto e também entre equipes diversas traz uma série de vantagens. Uma vez que todos estejam habituados aos métodos e documentos utilizados no projeto, cada um é capaz de entender o que tem que fazer e também é capaz de entender o que os demais colaboradores estão fazendo nas diversas etapas do projeto.

Outra vantagem apresentada por Jacobson et al. (1999) é que supervisores, gerentes e qualquer pessoa que não consiga entender código é capaz de entender o que está sendo feito, graças aos diagramas de arquitetura e demais artefatos gerados. Também é possível, graças aos mesmos artefatos, que recursos sejam

transferidos entre projetos e possam facilmente se integrar com as atividades que estão sendo desenvolvidas, uma vez que a descrição completa do produto a ser gerada pode ser entendida pela análise dos diagramas e demais artefatos.

Por fim, ainda temos como vantagem da utilização de um processo o fato de que ele torna as atividades de desenvolvimento padronizada e repetível, facilitando estimativas de custo, prazos e riscos, além de reduzir custos de treinamento na metodologia, pois ela já é de conhecimento da equipe, bastando treinar (através de cursos rápidos ou mesmo de outro colega) apenas aqueles que forem novos na equipe.

Assim sendo decidimos utilizar como metodologia de desenvolvimento de nosso projeto, a fim de orientar as atividades da equipe e controlar os progressos do projeto, o Processo Unificado visto que este era o de maior familiaridade entre os participantes do projeto.

2.1.1 Processo unificado (Unified Process – UP)

Como dito anteriormente, um processo é um modelo, um conjunto de atividades necessárias ao desenvolvimento de um projeto. Um projeto é uma instancia do processo e como tal representa o subconjunto das atividades e documentos previstos no processo original necessárias à realização do projeto em questão.

Essa especialização é necessária porque nenhum processo em particular pode ser utilizado de forma genérica. Cada projeto apresenta diferentes restrições,

tais com prazo, custo e qualidade exigidos e dessa forma o processo deve ser adaptado para que se ajuste aos diferentes contextos em que pode ser empregado.

Nesse sentido, o Processo Unificado foi desenvolvido, segundo seus idealizadores, para ser um processo genérico, uma *framework* que pode ser especializada para uma grande quantidade de sistemas de software, diferentes áreas de aplicação, tipos de organização e tamanhos de projeto. Jacobson et al.(1999) sugerem, entretanto, que dentro de uma mesma organização exista alguma consistência entre os processos utilizados para que não seja necessário um grande custo de treinamento quando da movimentação de recursos entre os diversos projetos da organização.

O Processo Unificado foi concebido inicialmente para atender às necessidades de grandes projetos, mas pode ser facilmente ajustado a projetos de menor complexidade pela simplificação de algumas atividades ou até pela completa supressão das mesmas, dependendo das necessidades do projeto.

Tendo o UML (Unified Modeling Language) como linguagem para a documentação do projeto, o Processo Unificado é baseado em componente, o que significa que o sistema desenvolvido é construído a partir de componentes de software conectados através de interfaces bem definidas. Dessa forma, é necessário que haja uma documentação consistente, clara e padronizada de todos os componentes, uma linguagem única que permita a todos os participantes do projeto compreender os diversos componentes através de tais documentos. Daí deriva a importância do UML para o Processo Unificado.

A característica principal que difere o UP dos demais processos são as três palavras chave com as quais os autores definem o processo:

- Guiado por casos de uso;
- Centrado em arquitetura;
- Iterativo e incremental.

2.1.1.1 Processo Guiado por Casos de Uso

Quando um sistema é desenvolvido, ele sempre tem como objetivo ser útil aos seus usuários. Assim sendo, é necessário que o sistema possua um conjunto de funcionalidades claras e que estas efetivamente tragam benefícios a seus usuários.

Nesse sentido, a utilização de casos de uso no desenvolvimento de sistemas representa um avanço em relação à tradicional especificação funcional. A especificação funcional do sistema nos leva a perguntar "O que o sistema deve fazer?", enquanto que os casos de uso nos levam a pensar isso, mas para cada usuário do sistema (ator), seja ele uma pessoa ou um sistema externo. Tal mudança de enfoque é importante, pois permite que especifiquemos funcionalidades que realmente agregam valor ao sistema, no sentido de que são funcionalidades que suprem as necessidades específicas de cada um dos usuários do sistema.

Outra vantagem fundamental encontrada na utilização de casos de uso é o fato de ela guiar o restante do desenvolvimento do projeto. É a partir dos casos de uso que os desenvolvedores irão extrair as classes que compõem o sistema e que

são capazes de implementar a funcionalidade descrita no caso de uso. É também a partir dos casos de uso que poderão ser extraídas as interfaces homem-máquina do sistema, pois ao conhecermos a funcionalidade e a forma de interação do usuário com a mesma, somos capazes de conceber a interface que melhor se ajusta a essa necessidade.

Além de auxiliar na análise e projeto do sistema (conforme exemplificado acima), os casos de uso também são de muita importância para os testes. Os responsáveis pelos testes deverão prever atividades que garantam o correto funcionamento do sistema, no sentido de certificar que todos os casos de uso previstos funcionam conforme especificado.

Por fim, a correta especificação dos casos de uso permite que as atividades seja particionadas, possibilitando um desenvolvimento iterativo e incremental do sistema. Desenvolvendo-se logo nas primeiras iterações um conjunto de casos de uso que seja arquiteturalmente relevante, conseguimos criar uma base de arquitetura estável, o que facilitará o desenvolvimento das demais funcionalidades do sistema.

2.1.1.2 Processo centrado em arquitetura

A arquitetura de um sistema de software desempenha um papel semelhante ao da arquitetura de um prédio. Antes de se iniciar a construção de um prédio, o mesmo é especificado em função de suas diversas características, sendo olhado por diferentes pontos de vista diferentes: encanamentos, distribuição de eletricidade, estrutura, etc. Da mesma forma, um sistema de software é olhado por diversos

ângulos que nos fornecem informações específicas sobre determinadas características do sistema, deixando de lado detalhes desnecessários.

A arquitetura contempla os mais significativos aspectos estáticos e dinâmicos do sistema. Tais aspectos são determinados conforme as necessidades do sistema a ser desenvolvido, tal como descrito nos casos de uso, mas também são influenciados por fatores como plataforma (sistema operacional, arquitetura de hardware, etc.) e aspectos não funcionais.

Uma vez que a arquitetura é influenciada tanto pelas requisições do sistema quanto por atributos externos, é necessário que haja um balanço entre arquitetura e casos de uso. Os casos de uso devem ser desenvolvidos para se ajustarem às possibilidades que a arquitetura fornece e, por outro lado, a arquitetura deve fornecer aos casos de uso a base necessária para que eles possam realizar todas as funcionalidades necessárias aos usuários do sistema. Assim sendo, é necessário que casos de uso e arquitetura sejam desenvolvidos paralelamente.

A utilização de uma metodologia centrada na arquitetura é fundamental pois ela nos permite obter um maior entendimento do sistema. Uma boa arquitetura permite aos desenvolvedores, gerentes, clientes e demais interessados entender o que está sendo feito com detalhes o bastante para facilitar sua participação.

Outro aspecto importante da arquitetura é que ela auxilia na organização do desenvolvimento. Em projetos de grande porte (principalmente aqueles cujas equipes estão geograficamente dispersas), a comunicação entre os diversos desenvolvedores para a compreensão e coordenação de esforços é uma atividade demorada e custosa. Nesse sentido, uma boa arquitetura é aquela que permite a

divisão do sistema em subsistemas com interfaces claramente definidas entre si, capazes de tornar claro para ambas as pontas o que elas têm que saber sobre o que as outras equipes estão fazendo. Dessa forma, ao se atribuir um subsistema a cada equipe, reduzimos a necessidade de comunicação entre as equipes para a coordenação de esforços.

Por fim, o desenvolvimento centrado em arquitetura permite a reutilização de componentes, o que reduz consideravelmente o custo e o tempo de desenvolvimento. Cabe ao arquiteto do sistema definir tais componentes, motivo pelo qual o desenvolvimento deve ser centrado em arquitetura.

Todos os aspectos citados acima são de fundamental importância para futuras evoluções do sistema, uma vez que o mesmo encontra-se bem definido e dividido em partes de fácil compreensão. Sistemas com arquiteturas mal definidas tornam difícil a manutenção e a evolução dos mesmos.

2.1.1.3 Processo Iterativo e Incremental

O desenvolvimento de um software comercial é um projeto grande que pode se estender por meses ou anos, dependendo da complexidade do sistema. Assim sendo, é útil dividir-se o desenvolvimento em mini-projetos que resultam em incrementos no sistema. Tais mini-projetos são as **iterações** do Processo Unificado. Cada iteração deve ser definida e desenvolvida de forma planejada e por isso podem ser consideradas mini-projetos.

Uma iteração não é obrigatoriamente aditiva ao produto. Especialmente nas fases iniciais do projeto, uma iteração costuma ter por objetivo substituir uma visão

superficial do sistema por outra mais detalhada. Tipicamente, nas fases mais avançadas do projeto as iterações tem um aspecto aditivo.

Jacobson et al.(1999) afirmam que a definição de o que vai ser executado em cada uma das iterações deve ser baseada em dois fatores. Primeiro, a iteração trata de um grupo de casos de uso que juntos representem uma extensão da usabilidade do sistema desenvolvido até então, ou seja, devem ser selecionados casos de uso que estejam relacionados a uma mesma funcionalidade do sistema. Segundo, a iteração trata dos riscos mais importantes.

Cada iteração se desenvolve sobre os artefatos gerados na iteração anterior. Sendo um mini-projeto, em cada iteração os desenvolvedores identificam e especificam os casos de uso relevantes, criam o design usando uma arquitetura escolhida como guia, implementam o design em componentes e verificam se os componentes satisfazem os casos de uso (testam os componentes).

Se ao final da iteração for constatado que ela atingiu os objetivos previstos, o desenvolvimento prossegue com uma nova iteração. Caso contrário, deve-se revisar o que foi feito e tentar-se uma nova abordagem.

Para garantir a redução dos custos no desenvolvimento, devem-se selecionar apenas as iterações necessárias para se alcançar os objetivos do projeto e devem-se encadear tais iterações em uma ordem lógica. Um projeto bem sucedido prosseguirá ao longo de um curso reto com somente pequenos desvios do curso inicialmente planejado. O uso de iterações permite a identificação prévia de problemas, reduzindo-se a quantidade de problemas não previstos.

Existem muitos benefícios no desenvolvimento iterativo.

Primeiramente, o uso de iterações reduz o risco ao custo de uma única iteração. Se for necessária a repetição de uma iteração, será perdido apenas o custo de uma iteração e não o valor do projeto inteiro.

Outro risco que o desenvolvimento iterativo reduz é o de não ser possível disponibilizar o produto ao mercado no tempo previsto. Ao se identificar previamente riscos ao desenvolvimento do projeto, o tempo gasto resolvendo-os ocorre antes no planejamento, num momento em que os prazos ainda são mais flexíveis. Na abordagem tradicional, tais problemas costumam surgir pela primeira vez é num momento no qual o tempo necessário para resolvê-los é maior que o tempo restante no planejamento, forçando um atraso na entrega do produto.

O uso de iterações aumenta a velocidade do projeto como um todo, pois o trabalho dos desenvolvedores se dará de forma mais eficiente numa agenda de curto prazo e com resultados mais claros e do que numa sempre mutável agenda de longo prazo.

2.1.1.4 Fases do Processo Unificado

Um projeto, ao ser desenvolvido através do Processo Unificado, possui uma série de *workflows*, que são os conjuntos de atividades necessárias ao desenvolvimento do projeto. Cada *workflow* se desenrola ao longo das diversas fases do projeto, sendo que cada fase pode ser realizada em uma ou mais iterações. A figura abaixo ilustra esse conceito.

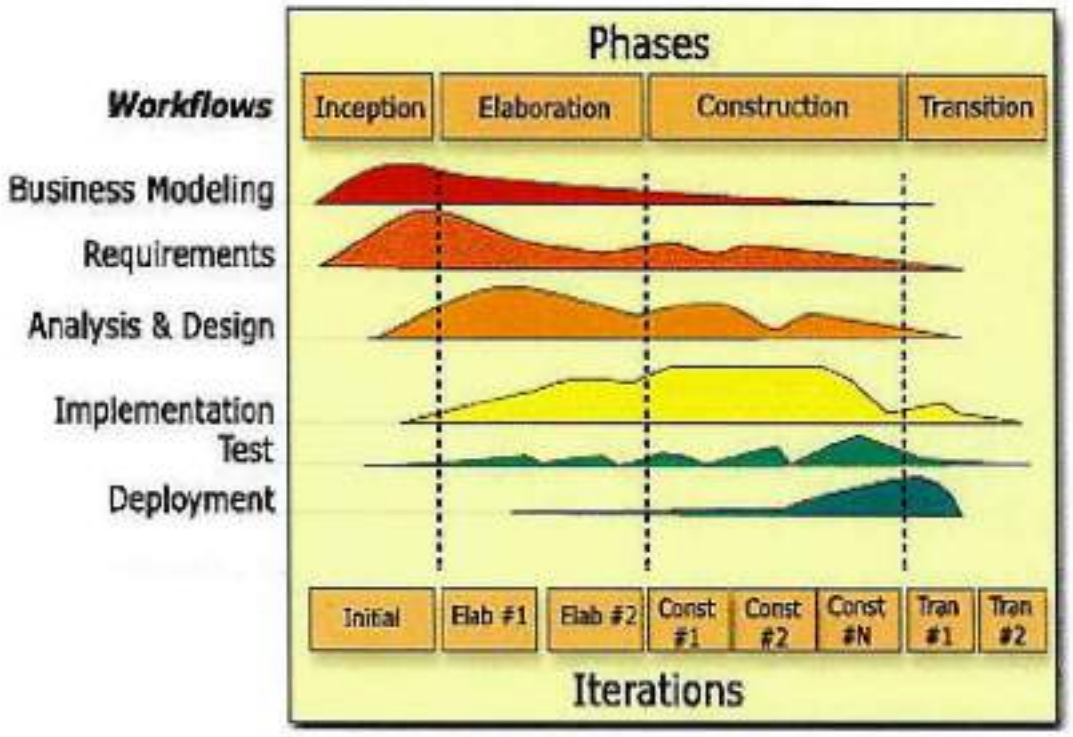

Figura 4 - Diagrama geral do UP

O Processo Unificado divide o ciclo de desenvolvimento em quatro fases: **concepção, elaboração, construção e transição.** Cada uma dessas fases tem um propósito específico e é delimitada por um *milestone*. Um *milestone* é um ponto no tempo em que decisões criticas são tomadas e objetivos principais devem ser alcançados.

A fase de concepção tem por objetivo delimitar o escopo do projeto, através da identificação de todas as entidades externas com as quais o sistema irá interagir (atores) e da natureza de tais interações. Isso envolve identificar todos os casos de uso e descrever os mais significativos. Também é definido nesse momento o business case, que inclui a estimativa de custos, a análise de risco e o planejamento macro do projeto, mostrando os principais milestones.

Ao final desta fase, esperas-se uma visão geral do projeto, mostrando os requisitos, funcionalidades chave e principais restrições do projeto. Espera-se que nesse momento, tenha-se uma primeira aproximação do modelo de casos de uso (cerca de 20% completo).

O objetivo da fase de elaboração é analisar o domínio do problema, estabelecer a arquitetura fundamental, desenvolver o plano do projeto e eliminar os principais elementos de risco do projeto. Para tanto, deve-se ter uma visão ampla do projeto, pois as decisões de arquitetura devem ser feitas com base em um entendimento de todo o sistema: seu escopo, principais funcionalidades e requisitos não funcionais, como desempenho.

A fase de elaboração é a mais critica, pois nesse momento decide-se entre prosseguir ou não para as fases de construção e transição. Isso significa passar de uma operação flexível e de baixo risco para uma de alto risco e grande inércia.

Espera-se ao final da fase de elaboração que cerca de 80% do modelo de casos de uso esteja completo, o que significa que todos os casos de uso e atores foram identificados e que a grande maioria deles já foi desenvolvida. Espera-se também um modelo da arquitetura do sistema, bem como um protótipo de tal arquitetura, que tem por objetivo identificar e eliminar riscos tecnológicos do projeto, além de demonstrar a estabilidade da arquitetura projetada.

Nessa fase, é feita também uma revisão das analises de custo e risco realizadas na fase anterior.

A fase de construção é o momento em que o software toma uma forma concreta, passando da visão de diagramas UML para uma visão de implementação.

Nesse momento, todos os componentes remanescentes são desenvolvidos e integrados ao produto e todas as funcionalidades são testadas. A construção é como um processo de manufatura, em que o foco de gerenciamento passa a ser o controle das operações, a redução dos custos, a otimização dos prazos e o controle da qualidade. Ao final desta fase tem-se a primeira versão do software, que poderá ser implantada no cliente.

Por fim, a fase de transição é a fase onde o produto se desloca do desenvolvimento para o usuário final. Embora pareça que a tarefa das equipes de desenvolvimento terminou, isto não é verdade, pois no mundo do software a primeira versão de um produto não significa que o sistema inteiro funcionará (ao menos não sempre). São necessários ajustes finos no sistema que só conseguem ser detectados quando é feito o teste junto ao usuário, submetendo o sistema a situações reais de funcionamento. As atividades da fase incluem: treinamento dos usuários e dos mantedores, teste beta do sistema para validá-lo de acordo com as expectativas dos usuários finais.

As fases do Processo Unificado podem ser divididas em uma ou mais iterações. Iterações são mini-projetos nos quais se desenrolam os diversos *workflows* previstos no Processo Unificado. A intensidade de um *workflow* depende da fase e da iteração que está se desenrolando. Um *workflow* é uma seqüência de atividades que produz resultados de valor observável.

Os principais *workflows* do Processo Unificado são:

- Modelagem de Negócio
- Captura de Requisitos

- Analise e Projeto
- Implementação
- Teste
- Distribuição

Na modelagem de negócio são documentados os processos de negocio utilizando-se casos de uso de negócio. Isso assegura um entendimento comum entre os participantes de quais processos de negocio precisam ser apoiados na organização. Os casos de uso de negocio são analisados para entender-se como o negocio pode apoiar os processos de negocio. Com isso, espera-se garantir um mapeamento direto entre o negócio e os modelos de software, garantindo que este ultimo agregue valor efetivo ao negócio.

O objetivo da captura de requisitos é descrever o que o sistema deve fazer e permitir aos desenvolvedores e clientes entrar em acordo sobre essa descrição. Os atores são identificados, representando os usuários e quaisquer outros sistemas que interajam com o sistema que está sendo desenvolvido. São identificados também os casos de uso, representando o comportamento do sistema. Porque os casos de uso são desenvolvidos de acordo com as necessidades dos atores, é mais provável que o sistema seja relevante para os usuários.

Cada caso de uso é descrito em detalhes. A descrição de casos de uso mostra como o sistema interage passo a passo com os atores e o que o sistema faz. A análise e projeto tem por objetivo mostrar como o sistema será realizado na fase de implementação. São desenvolvidos um modelo de projeto e um modelo de

análise. O modelo de projeto serve como uma abstração do código fonte, isto é, o modelo de projeto funciona como uma planta de como o código fonte será estruturado e escrito.

O modelo de projeto consiste no projeto de classes estruturadas dentro do projeto de pacotes e projeto de subsistemas com interfaces bem definidas, representando o que serão os componentes na implementação. Ele também contém descrições de como os objetos desse modelo colaboram para executar os casos de uso.

Como o Processo Unificado é um processo centrado em arquitetura, todas as atividades de projeto tem esse mesmo enfoque. A arquitetura é representada por uma série de vistas, que capturam as principais decisões de design. Em essência, as vistas de arquitetura são uma abstração ou simplificação do projeto como um todo, onde importantes características são tornadas mais visíveis pela supressão de detalhes. A arquitetura é um importante meio não apenas para o desenvolvimento de um bom modelo de projeto, mas também para aumentar a qualidade de qualquer modelo feito durante o desenvolvimento do sistema.

O *workflow* de implementação tem por objetivo converter os modelos desenvolvidos nos *workflows* anteriores em código compilável. É nessa hora que os diversos componentes descritos nos modelos gerado anteriormente são integrados. Uma vez que se tenha o código compilável pronto, pode-se iniciar os testes, que visam verificar a interação entre os objetos, a correta integração entre os componentes de software, verificar se todos os requisitos foram devidamente

implementados e identificar e garantir que defeitos sejam corrigidos antes do distribuição do software.

Por fim, a distribuição engloba atividades como fechamento do pacote de software, instalação do software no cliente, planejamento e condução dos testes beta, migração de dados existentes em sistemas legados e testes de aceitação.

É importante notar que, apesar dos *workflows* descritos lembrarem as fases seqüenciais do processo de desenvolvimento em cascata, o Processo Unificado é iterativo, de forma que tais *workflows* são constantemente revisitados nas diversas iterações do desenvolvimento do projeto.

2.2 Computação Gráfica

A área de computação gráfica engloba o processo de criar imagens por meio de computador. Envolve técnicas de desenho e pintura em duas dimensões, bem como o uso de técnicas que empregam recursos mais sofisticados de renderização em três dimensões. Segundo Foley et al. (1996), a área de computação gráfica também engloba, além do processo de criação, o armazenamento e a manipulação de modelos de imagens de objetos.

Para auxiliar a criação de imagens tridimensionais em computadores pessoais, os sistemas operacionais fornecem as chamadas APIs (*application programming interface*), que são um conjunto de funções utilizadas por uma aplicação para gerar um programa (WALSH, 2003). Uma API abstrai uma grande quantidade de funções, como multiprocessamento e proteção de memória, bem

como fornece interfaces para conceitos de nível mais alto, como menus e caixas de diálogo.

As APIs estão disponíveis nos principais sistemas operacionais existentes no mercado, como o Windows, o Linux e o Macintosh.

O Windows, por exemplo, possui um conjunto vasto de APIs. Pode-se fazer de tudo, desde rodar vídeos até carregar páginas web.

Sob o ponto de vista de gráficos tridimensionais, existem as APIs de gráfico 3D, presentes nos sistemas operacionais citados acima. Dentre as mais populares, estão a OpenGL e a Direct3D (subconjunto do DirectX).

Com base nas APIs de gráfico 3D, pode-se construir aplicações gráficas tridimensionais. Uma das formas mais conhecidas de se modelar e controlar uma cena 3D é através do uso de grafos de cena.

Um grafo de cena é uma estrutura de dados em árvore que armazena, organiza e renderiza informação de uma cena tridimensional (objetos 3D, materiais, luzes e comportamentos).

Como exemplo de aplicações 3D que fazem uso de grafos de cena, pode-se citar as aplicações feitas com a API Java 3D.

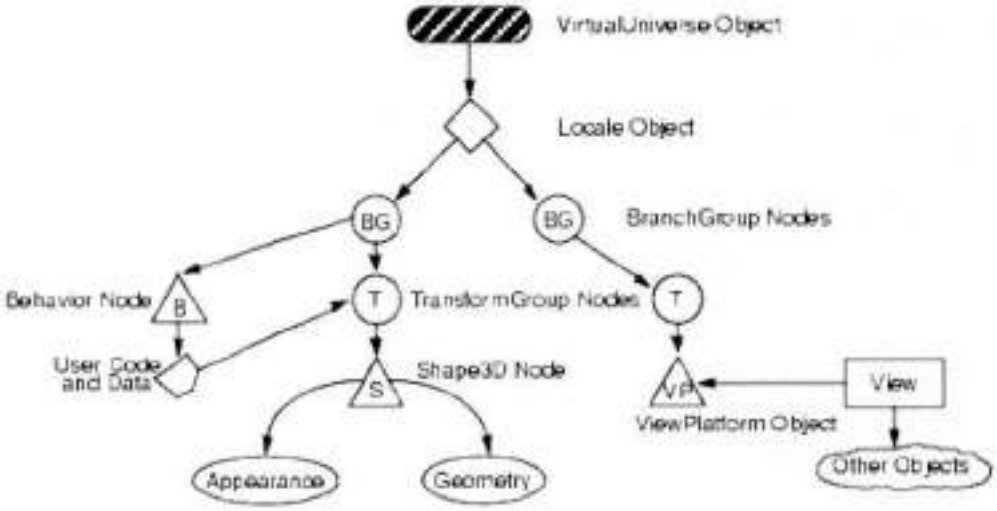

Figura 5 - Exemplo de grafo de cena

2.3 *Pathfinding*

Pathfinding é um conceito normalmente associado com a procura pelo caminho de menor custo, mas existem outras possibilidades de interpretação, e que podem ser consideradas de maneira diferente (REESE et al, 1999). Dentre as várias possibilidades, inclue-se o encontro de:

- qualquer caminho;
- caminho que realiza a máxima cobertura de uma área (EMMANUEL et al. apud REESE et al., 1999);
- caminho que propicia à unidade em movimento a menor exposição possivel aos eventuais adversários (HOFF et al. apud REESE, 1999).

No projeto em questão, aborda-se o *pathfinding* sob o ponto de vista de procura pelo caminho de menor custo.

Para a determinação do caminho, o desenvolvedor deve levar em consideração quatro aspectos básicos: dinâmica do ambiente, geometria do ambiente, nível de exposição do terreno, e uma miscelânea de fatores. Segue abaixo a descrição de cada um dos aspectos

2.3.1 Dinâmica do ambiente

Algumas partes do ambiente que compõe o jogo podem mudar ao longo do tempo. Desta forma, lista-se abaixo os principais tipos de mudança de ambiente, de acordo com a origem da mudança:

- Ambiente estático: A navegação se torna mais fácil porque o ambiente é previsível. Existe um amplo número de soluções para este caso (LATOMBE apud REESE et al., 1999);
- Unidades móveis e obstáculos estáticos: A presença de outras unidades móveis precisa ser tratada dinamicamente, já que seus futuros caminhos não são conhecidos. Felizmente, é possível desconsiderar tais unidades enquanto se traça um caminho, e tratar delas assim que as mesmas cruzarem o caminho;
- Obstáculos móveis: Se os obstáculos podem ser realocados, não se pode pré-calcular o caminho inteiro. É caro sob o ponto de vista computacional replanejar o caminho toda vez em que uma discrepância é detectada entre a configuração do ambiente antes e depois da realocação de um obstáculo. Para estes casos, esquemas mais dinâmicos são requeridos, como o D* (STENTZ apud REESE et

al., 1999) e estrutura de dados cinética (BASCH et al. apud REESE et al., 1999).

- Obstáculos que aparecem e desaparecem: Se os obstáculos podem, além de se mover, também aparecer e desaparecer, o *pathfinding* se torna muito mais complicado.
- Manipulação: Se a unidade possui a capacidade de alterar o ambiente durante a sua movimentação, o planejamento do menor caminho se torna muito mais complicado. Isso pode acontecer pela movimentação (BEN-SHAHAR et al. apud REESE et al., 1999), destruição ou mesmo pela construção de novas partes do ambiente.

2.3.2 Geometria do ambiente

A aparência física do ambiente exerce uma grande influência sobre o *pathfinding*. Quanto às formas, unidades móveis e obstáculos que possuem formas geométricas simples, como cubos e esferas, são muito mais fáceis de tratar se comparadas com entidades de forma complexa.

Sob o ponto de vista de particionamento, é comum o uso de algoritmos baseados em grafos para se traçar um caminho. O Grafo é encontrado se dividindo o ambiente em partições, o que pode ser feito de diversas formas. O tamanho e a conectividade do grafo resultante influi na performance dos algoritmos de busca. Assim, o método de particionamento do ambiente é muito importante.

Em se tratando de concavidade, algumas técnicas de navegação assumem que a unidade móvel deve circunavegar os obstáculos, ao invés de contorná-las de forma brusca, como em labirintos não-curvilíneos. O conhecimento do tipo de terreno é fundamental para a escolha da técnica de navegação mais adequada.

Por fim, a penetrabilidade do terreno pode ser mais subdividida do que a penetrabilidade binária (em que um trecho é penetrável ou impenetrável). Algumas partes podem ser difíceis, mas não impossíveis de se penetrar.

2.3.3 Nível de exposição do terreno

A quantidade de informação disponível pode ser limitada. Pode haver partes desconhecidas do terreno, ou partes anteriormente visitadas cuja configuração foi alterada. Quanto mais informação disponível, mais precisamente o caminho pode ser traçado.

Quanto à exploração do terreno, a exploração de terreno desconhecido pode aumentar a quantidade de informação disponível, mas a decisão de como explorar e quais partes explorar pode ser custosa.

2.3.4 Miscelânea de fatores

Nem todas as técnicas de *pathfinding* são capazes de operar adequadamente com uma quantidade limitada de memória. Algumas técnicas poder

ser adaptadas para funcionar em tais condições, mas normalmente requerem um compromisso entre tempo e espaço de armazenamento;

O nível de acurácia espacial do caminho é importante. Menos acurácia leva à uma determinação de caminho mais veloz, além de requerer menos memória. Contudo, menos acurácia pode produzir movimentos ineficientes ou não-naturais;

Fatores como gravidade, inércia e velocidade limitada podem exercer influência sobre o tipo de método usado para a determinação do caminho;

Unidades móveis podem ter de interagir (como por exemplo cooperar ou competir) para alcançar seus objetivos. Um exemplo é dois jogadores tentando passar por um corredor estreito. Eles devem coordenar os seus movimentos de modo a prevenir o bloqueio mútuo do caminho.

Baseando-se nos quatro fatores acima, e levando-se em consideração as características do projeto em desenvolvimento, realizou-se um estudo do algoritmo mais adequado de *pathfinding*.

Como caracteristicas do projeto levadas em consideração, tem-se:

- Unidades móveis e obstáculos estáticos;
- Unidades estáticas, aqui encaradas como obstáculos estáticos, possuindo formas simples (paralelepípedos) e unidades dinâmicas com formas relativamente simples (humanóides);
- Terreno sem concavidade;
- Terreno de penetrabilidade binária;
- Terreno com informação ilimitada (plenamente conhecido);
- Memória Limitada;

- Acurácia Média – Compromisso entre rapidez do algoritmo de *pathfinding* e naturalidade do movimento das unidades;
- Sem gravidade;
- Sem inércia;
- Velocidade Limitada;
- Baixa interação entre as unidades móveis;

A imposição das características acima deve-se ao escopo determinado para o projeto. Ao mesmo tempo em que fornecem as condições necessárias para o desenvolvimento de um projeto abrangente sob o ponto de vista de computacional (algoritmo de *pathfinding*, tratamento de colisão, inteligência artificial, design e animação), suporta os fatores condicionantes, como os recursos tecnológicos disponíveis, e o tempo limitado para implementação do projeto, associado à inexperiência na área de desenvolvimento de jogos por parte dos envolvidos.

2.3.5 Algoritmos de Pathfinding

Os algoritmos de *pathfinding* provenientes de trabalhos científicos trabalham com grafos no sentido puramente matemático – um conjunto de vértices com arestas conectando-os. Um terreno demarcado pode ser considerado um grafo, com cada demarcação sendo um vértice, e as arestas sendo desenhadas entre os vértices adjacentes.

O algoritmo de Dijkstra funciona através da visita aos vértices do grafo começando pelo ponto de partida do objeto. Ele então examina repetidamente o vértice mais próximo ainda não examinado, adicionando este vértice ao conjunto de vértices a serem examinados. Ele então expande do ponto de partida até que se chegue ao destino.

O algoritmo de Dijkstra assegura o encontro do caminho mais curto entre um ponto de partida e um destino desde que nenhuma das arestas possua custo negativo.

O algoritmo *Best-First-Search* (BFS) funciona de modo similar, exceto pelo fato de possuir certa estimativa (chamada heurística) de quão longe do destino um vértice está.

Ao invés de selecionar o vértice mais próximo do ponto de partida, o algoritmo seleciona o vértice mais próximo do alvo. O algoritmo BFS não assegura o encontro do caminho mais curto. Todavia, ele é executado mais rapidamente do que o algoritmo de Dijkstra porque ele usa uma função heurística para traçar o seu caminho até o destino de forma rápida. Por exemplo, se o destino está ao sul da posição inicial, BFS tenderá a se concentrar em caminhos que levam à direção sul.

O problema é que o BFS se mostra um algoritmo "afoito", e tenta se mover rumo ao destino mesmo que não esteja percorrendo o caminho certo. Desde que considera somente o custo para chegar ao destino e ignora o custo do caminho, ele continua o trajeto mesmo que o caminho traçado tenha se tornado longo.

Analisando-se os dois algoritmos acima, conclui-se que o ideal seria uma combinação entre ambos.

Segundo Patel (2005), o algoritmo A* foi desenvolvido em 1968 para combinar aproximações de heurística do algoritmo BFS com as aproximações formais do algoritmo de Dijkstra.

O A* é a mais popular escolha para *pathfinding*, isto porque ele é extremamente flexível e pode ser usado em uma ampla variedade de contextos.

O A* é como outros algoritmos de busca por grafo na medida em que ele tem o potencial de procurar uma ampla área do mapa. É como o algoritmo de Dijkstra na medida em que pode ser usado para encontrar o menor caminho. É como o algoritmo BFS já que pode utilizar uma heurística para se guiar. No caso simples, é tão rápido quanto o BFS.

O segredo para o seu sucesso é que ele combina as informações que o Dijkstra utiliza (dando preferência aos vértices que estão próximos ao ponto de partida) e a informação utilizada pelo BFS (dando preferência aos vértices que estão próximos ao destino).

Na terminologia usada quando se fala do algoritmo A*, g(n) representa o custo do caminho do ponto de partida para qualquer vértice n, e h(n) representa o custo do vértice n até o destino estimado pela heurística. O A* balanceia os dois custos enquanto se move do ponto de partida até o destino.

2.3.6 Heurísticas

A função heurística h(n) informa ao A* uma estimativa do mínimo custo calculado a partir de qualquer vértice n até o destino, e pode ser usada para controlar o comportamento do A*:

Em um extremo, se h(n) é igual à zero, então somente g(n) influencia, e A* se transforma no algoritmo de Dijkstra, garantindo que se encontre o menor caminho;

Se h(n) é sempre menor do que (ou igual a) o custo de mover-se do vértice n até o destino, então A* assegura que será encontrado o menor caminho. Quanto menor for o valor de h(n), mais o número de nós do A* se expande, tornando-o mais lento;

Se h(n) for exatamente igual ao custo de mover-se do vértice n até o destino, então o A* seguirá somente o melhor caminho e nunca se expandirá, tornando-se muito rápido. Embora não seja possível alcançar tal intento em todos os casos, pode-se torná-lo exato em alguns casos especiais;

Se o h(n) for algumas vezes maior do que o custo de mover-se a partir do vértice n até o destino, então o A* não garante o encontro do menor caminho, mas pode ser executado rapidamente;

Em outro extremo, se h(n) é muito elevado em relação ao g(n), então somente h(n) exerce influência, e o A* se torna o BFS.

Em um jogo, esta flexibilidade do A* pode ser muito proveitosa. Por

exemplo, pode-se considerar em algumas situações um bom caminho, e não um caminho perfeito.

2.3.7 Velocidade versus acurácia

A habilidade do A* variar seu comportamento baseado na heurística e em funções de custo pode ser muito útil em um jogo. O compromisso entre velocidade e acurácia pode ser explorado para fazer o jogo mais rápido. Para a maioria dos jogos, não é realmente necessário encontrar-se o melhor caminho entre dois pontos. Algumas vezes, é preciso apenas algo próximo.

A escolha entre velocidade e acurácia não precisa ser estática. Você pode escolher dinamicamente baseado na velocidade da CPU, a fração de tempo utilizada para se executar o algoritmo, o número de unidades do mapa, a importância da unidade, o tamanho do grupo, o nível de dificuldade, ou qualquer outro fator.

Além disso, a escolha entre velocidade e acurácia não necessita ser global. Pode-se escolher algumas coisas dinamicamente baseando-se na importância de se ter acurácia em certa região do mapa. Por exemplo, talvez não seja tão importante ter de encontrar o caminho mais curto em uma área segura do mapa, mas quando se passa por uma área inimiga, segurança e velocidade são fundamentais.

2.3.8 Escala

O algoritmo A* calcula f(n) = g(n) + h(n). Para usar os parâmetros g e h, ambos precisam estar na mesma escala. Se g(n) é medido em horas e h(n) é medido em metros, então o A* considerará g ou h muito grande ou muito pequeno, e assim se terá ou caminhos ruins ou o A* será executado mais lentamente do que deveria.

Em uma grade, existem funções heurísticas amplamente reconhecidas para se usar.

A heurística padrão é a distância Manhattan. Neste método, considera-se a distância do vértice n até o destino como sendo:

$$h(n) = D * (abs (n.x - destino.x) + abs (n.y - destino.y))$$

Onde:

- D – distância entre duas marcações adjacentes;
- n.x e n.y – coordenadas x e y do vértice n;
- destino.x e destino.y – coordenadas x e y do destino.

Neste modo, a estimativa do caminho futuro (h(n)) considera que as unidades se movimentam pelo mapa vertical e horizontalmente, não podendo explorar as diagonais.

Se no mapa é permitida movimentação diagonal, e se é desejado que a heurística considere tal fato, é necessária uma heurística diferente.

Na distância Manhattan, uma movimentação de 4 marcações para o norte seguida de 4 para o leste constituiria no valor 8 * D. Se essa movimentação pudesse ser simplificada com 4 marcações percorridas diagonalmente, então a heurística daria 4 * D, assumindo-se que, ao se percorrer duas marcações adjacentes, movimentos na vertical, horizontal e diagonal custam todos D.

Para esse caso, a fórmula ficaria:

$$h(n) = D * max (abs (n.x - destino.x), abs (n.y - destino.y))$$

Se o custo de movimentação na diagonal não for D, mas algo do tipo D2 = sqrt(2) * D, a heurística acima não é ideal. Neste caso, deve-se ter algo mais sofisticado como:

$$h\_diagonal(n) = min(abs (n.x - destino.x), abs (n.y - destino.y))$$
$$h\_vertic\_horiz(n) = (abs (n.x - destino.x) + abs (n.y - destino.y))$$
$$h(n) = D2 * h\_diagonal(n) + D * (h\_vertic\_horiz(n) - 2 * h\_diagonal(n))$$

Se as unidades podem se mover em qualquer ângulo, então deve-se usar a distância Euclideana:

```
h(n) = D * sqrt ( ( n.x – destino.x) ^ 2 + ( n.y – destino.y ) ^ 2 )
```

Contudo, se for este o caso, então se pode ter problemas ao se utilizar o A* diretamente porque a função de custo g não será igual à função heurística h. Como a distância Euclideana é menor do que a distância Manhattan ou a distância Diagonal, serão obtidos os caminhos mais curtos, mas o A* levará mais tempo para ser executado.

Das heurísticas apresentadas acima, escolheu-se a distância Manhattan, pois a mesma exige menor processamento, ao mesmo tempo em que atende às necessidades de *pathfinding* exigidas pelo jogo.

2.4 *Inteligência Artificial*

O campo de Inteligência Artificial tem sido estudado ativamente a partir da década de 50. Naquela época, muitas técnicas foram desenvolvidas e utilizadas em uma ampla série de aplicações comerciais e escolares.

Os jogos modernos possuem sofisticados recursos gráficos, assim como as suas tecnologias de animação. À medida que a animação gráfica se transforma em um problema resolvido, cada vez mais se direcionam as atenções para a área de inteligência artificial.

2.4.1 A Inteligência Artificial na área de jogos

Desde o surgimento do mercado de jogos para computador, a inteligência artificial (IA) tem sido um componente padrão nos jogos, especialmente para os pertencentes ao estilo *single-player*. Embora recentemente já exista suporte para atividades *multi-player* (especialmente via Internet), os recursos de inteligência artificial ainda se mostram necessários para a maioria dos jogos.

Segundo Woodcock (1998), atualmente, cada vez mais desenvolvedores de jogos estão se envolvendo na implementação da inteligência artificial logo nos estágios iniciais do projeto, e mais recursos estão sendo reservados para o processamento da IA. A tabela 1 mostra o resultado de uma pesquisa feita na

Conferência de Desenvolvedores de Jogos para Computador, realizada nos anos de 1997 e 1998. A pesquisa consistia de duas perguntas realizadas aos desenvolvedores de jogos presentes:

O seu time inclui programadores dedicados a inteligência artificial em tempo integral?		
	1997 CGDC	1998 CGDC
SIM	50/207 (~24%)	87/190 (~46%)

Qual a porcentagem de processamento da CPU a inteligência artificial de seu jogo consome?		
	1997 CGDC	1998 CGDC
Jogos em Tempo Real	1-2%	5-10%
Jogos Baseados em Turnos	5-25%	5-50%

Tabela I - Recursos destinados a IA aos jogos.

O número de projetos com desenvolvedores dedicados a IA aumentou de 1997 (24%) para 1998 (46%). O número médio de ciclos de *clock* da CPU dedicados ao processamento de IA aumentou também de 1997 para 1998.

Á medida que aumenta a importância de uma eficiente IA nos jogos, desenvolvedores têm se dirigido ao setor acadêmico e defendido pesquisas para se encontrar métodos mais complexos e mais eficientes do que as tentativas realizadas freqüentemente no passado. Segue abaixo as principais tecnologias de IA empregadas recentemente em jogos, de acordo com Woodcock (1998):

2.4.2 IA baseada em regras

IA baseadas em regras, caracterizadas pelo largo emprego de máquinas de estado finito (FSM) e seus parentes próximos, as máquinas de estado *fuzzy* (FuSM), continuam sendo a principal escolha de tecnologia para desenvolvimento de jogos. Todo jogo no mercado utiliza IA baseadas em regras em certo grau, e com uma boa razão para isso: estes métodos funcionam.

IA baseada em regras continuam a base da inteligência do oponente na maioria dos jogos por três simples razões:

1 – Aproximações de IA baseadas em regras são familiares, tornando seus princípios confortavelmente aplicáveis aos paradigmas de programação;

2 – Implementações baseadas em regras são geralmente previsíveis e, portanto, fáceis de se testar (embora isso leve a uma das maiores reclamações dos jogadores quando se trata de IA – a previsibilidade da IA dos jogos);

3 – A maioria dos desenvolvedores carece de treinamento ou conhecimento das tecnologias de IA mais complexas e, portanto, não costumam fazer uso das mesmas quando as *deadlines* do projeto estão próximas;

2.4.3 Máquinas de Estado Finito

As máquinas de estado finito (FSM) são provavelmente a mais comum tecnologia de IA em uso. A FSM é uma hierarquia lógica de regras e condições que podem ser aplicadas a um número fixo de estados possíveis. A entrada e a saída em cada estado são sempre as mesmas, e não há escolha da seqüência de estados em que a máquina é visitada.

2.4.4 Máquinas de Estado *Fuzzy*

As máquinas de estado fuzzy (FuSM) são muito parecidas com as FSMs, com uma pequena diferença: em lugar de um conjunto de entradas mapear uma saída específica, elas mapeiam uma série de possíveis respostas. Estas possíveis respostas recebem um peso baseado no grupo ao qual elas pertencem.

Desenvolvedores então utilizam uma variedade de métodos para selecionar uma dada resposta – os pesos podem ser usados como uma simples probabilidade. O resultado é uma máquina que pode gerar uma variedade de respostas plausíveis, a partir de um dado estímulo.

2.4.5 Algoritmos Genéticos

Algoritmo genético (GA) é uma aproximação de aprendizado de máquina cujo comportamento deriva dos processos de evolução natural. Isto é feito criando-se em uma máquina uma população de indivíduos representados por cromossomos. Os indivíduos da população entram em um processo de evolução em que eles são testados a partir de critérios de adaptabilidade. Aqueles que falham são descartados, enquanto aqueles que atingem pontuações mais altas são conservados. Mutação é freqüentemente permitida para prevenir um estado de constância. O resultado é uma população de indivíduos que gradualmente se adaptam às condições dos ambientes digitais, ao longo do tempo.

2.4.6 Redes Neurais

Redes neurais são uma tentativa de se resolver problemas através da imitação dos trabalhos em um cérebro. Pesquisadores iniciaram com a tentativa de imitar o aprendizado dos animais utilizando uma coleção de neurônios idealizados, e aplicando estímulos neles de modo a mudarem seus comportamentos.

Essa área evoluiu muito nos últimos anos, principalmente devido à descoberta de vários algoritmos de aprendizado, permitindo a implementação da idéia de redes neurais com sucesso. Todavia, existem ainda muitas descobertas a

serem feitas neste campo para que se torne possível simular o cérebro humano eficientemente.

2.5 *Game Design*

Designers de jogos despendem uma grande quantidade de tempo questionando o quê os jogadores estão procurando em um jogo de computador. Investigam os elementos que podem ser incorporados aos seus jogos não inclusos anteriormente e que agradariam aos jogadores.

Segundo Rouse III(2003), para responder ao questionamento acima, pode-se focar em dois pontos básicos: porquê os jogadores jogam e o quê eles esperam.

2.5.1 Por que os jogadores jogam?

Conforme Rouse III(2003), a primeira pergunta a ser considerada é: por que os jogadores jogam? Qual o aspecto único nos jogos de computadores em relação a outras ferramentas de entretenimento?

Entendendo os aspectos atrativos em jogos e que outros meios não fornecem, os designers ficam possibilitados de enfatizar tais diferenças, conseguindo assim diferenciar sua arte das demais.

2.5.1.1 Jogadores querem um desafio

Muitos jogadores apreciam certos jogos desde que os mesmos lhe propiciem desafios. Isto constitui um dos motivadores básicos para jogadores.

Jogos podem entreter seus usuários através do tempo, de uma forma diferente a cada vez que são jogados, ao mesmo tempo em que estimulam a mente de seus jogadores de uma forma diferente de um livro, filme, ou outra forma de arte.

Jogos forçam as pessoas a pensarem ativamente, a tentarem diferentes soluções para os problemas, e a entenderem um dado mecanismo do jogo.

Quando um indivíduo encara um desafio e o supera, ele aprendeu algo. Assim, jogos desafiadores podem ser fontes de aprendizado. Nos melhores jogos, os jogadores aprenderão lições que poderão ser aplicadas a outros aspectos de suas vidas, mesmo que eles não se dêem conta disso.

Isto significa que esses jogadores podem aplicar métodos de resolução de problemas em seus trabalhos, habilidades espaciais aprimoradas para organizar suas mobílias, ou até mesmo desenvolver grande empatia por meio de RPGs (role-playing games).

2.5.1.2 Jogadores querem se socializar

Segundo Rouse III (2003), *designers* de jogos para computadores precisam se lembrar de que as raízes dos jogos, e uma parte importante de seu apelo, estão em sua natureza social.

Para a maioria das pessoas, a razão básica pela qual elas jogam é para ter uma experiência social com suas famílias e amigos. É verdade que diversas pessoas apreciam jogos de cartas solitários, mas existem muito mais jogadores *multi-player* do que *single-player*.

2.5.1.3 Jogadores querem uma experiência anti-social dinâmica

Embora essa afirmação possa se mostrar contraditória com o item anterior, deve-se salientar que as mesmas não ocorrem simultaneamente: uma parcela de jogadores está à procura de uma experiência social, e uma diferente parte está em busca de algo dinâmico que eles possam apreciar sozinhos. Algumas vezes os amigos não estão disponíveis, ou um jogador está cansado de seus amigos, ou simplesmente está cansado de falar com seus amigos no momento.

Jogos possuem aspectos diferentes de outras atividades individuais, como ler um livro ou assistir a um vídeo, na medida em que eles fornecem aos seus usuários algo com que possam interagir, reagindo como um humano o faria, ou ao

menos de forma semelhante. Contudo, os jogadores estão sempre no controle, e podem começar e parar o jogo a qualquer momento. Desta forma, pessoas podem se dirigir aos jogos de computador para viver uma experiência dinâmica e interativa, mas ainda sim anti-social.

2.5.1.4 Jogadores querem vangloriar-se

Particularmente em jogos do tipo *multi-player*, jogadores participam para ganhar respeito. Mesmo em jogos *single-player*, os jogadores falarão com seus amigos sobre como eles terminaram um jogo ou sobre como são habilidosos em outro.

Assim, jogadores vêem nos jogos uma oportunidade para se destacar sobre todos os seus amigos em um jogo de computador.

Mesmo não revelando a ninguém, jogadores podem sentir uma imensa satisfação quando vencem um jogo particular. Quando vitoriosos em um jogo de grande desafio, eles se dão conta de que são capazes de fazer algo bem, provavelmente melhor do que a maioria das pessoas, o que os faz sentirem-se bem consigo mesmos.

2.5.1.5 Jogadores querem uma experiência emocional

Assim como em outras formas de entretenimento, jogadores podem estar à procura de algum retorno emocional quando se dirigem a um jogo de computador. Isto pode ser tão simples quanto uma descarga de adrenalina e de tensão em jogos rápidos, ou algo mais complexo, como o sentimento de perda, ou fatalidade.

Segundo Rouse III (2003), as pessoas querem sentir algo quando interagem com uma arte, e isso não precisa necessariamente ser um sentimento alegre, positivo. Talvez os efeitos de catarse que as pessoas obtêm de tais proveniências façam as mesmas se tornarem experiências valorosas.

Conforme Rouse III(2003), escala emocional é algo poucamente explorado pelos desenvolvedores de jogos, com muitos desenvolvedores receosos em fazer um jogo muito triste.

Assim, *designers* de jogos precisam ser sábios para se concentrar em expandir a experiência emocional em jogos através da alegria e realização para um território emocional mais desconhecido.

2.5.1.6 Jogadores querem fantasiar

Se considerarmos novelas, filmes, ou histórias em quadrinhos, veremos que muitas pessoas experimentam estes trabalhos para escapar de suas vidas reais

para um mundo completamente diferente, formado de personagens interessantes, atividades estimulantes, e de viagem a lugares exóticos.

Alguns críticos ridicularizam tais formas de arte, e certamente muitos livros, filmes e quadrinhos que lidam com um panorama mais realístico obtêm um excelente efeito. Todavia, ainda assim persiste o fato de que muitas pessoas querem ser transportadas para um mundo mais glamourizado do que os seus próprios.

Jogos de computador, então, mostram potencial para ser mais do que uma forma imersiva de escapismo. Em jogos, os jogadores possuem a chance de realmente ser alguém mais interessante. Enquanto em livros e filmes a audiência pode apenas assistir aos caracteres levarem uma vida excitante, em um jogo de computador bem produzido um jogador terá a chance de ele mesmo viver suas vidas. Na maioria dos jogos, os usuários não necessitam se preocupar em comer, dormir ou ir ao banheiro. Assim, um jogo pode criar uma vida fantasiosa desprovida de detalhes tediosos e, mais importante, o nível de imersão fantasiosa pode ser superior ao das outras formas de arte devido à natureza interativa dos jogos.

Outra parte do elemento fantasioso dos jogos de computadores é a capacidade de fornecer ao jogador a possibilidade de se engajar em comportamentos socialmente inaceitáveis em um ambiente seguro. Muitos jogos permitem aos seus usuários ser criminosos ou assassinos. Segundo Rouse III (2003), desta forma, jogos de computador fornecem um bom meio para os jogadores explorarem lados de suas personalidades que se mantém submersos em suas vidas diárias.

2.5.2 O que os jogadores esperam?

Uma vez que o jogador decidiu jogar um determinado jogo devido a um fato motivador ou outro, ele terá expectativas em relação ao jogo em si. Entre o jogo não dar *bugs* e parecer visualmente agradável, os jogadores possuem uma série de expectativas, e se as mesmas não são atendidas, o jogador logo se frustra e encontra outro jogo para se entreter. É trabalho do *designer* de jogo assegurar que o jogo corresponda a tais expectativas.

2.5.2.1 Jogadores esperam um mundo consistente

Assim que os jogadores começam a jogar, eles começam a entender quais ações lhe são permitidas de realizar, e quais resultados estas ações produzem.

Poucas coisas são mais frustrantes do que quando o jogador antecipa certo resultado de uma ação praticada e o jogo, por alguma razão imperceptível, produz um resultado diferente. A situação se mostra pior ainda quando as conseqüências das ações do jogador se mostram tão imprevisíveis que o jogador não pode nem mesmo estabelecer algum tipo de expectativa. Não ter expectativa do que acontecerá se uma manobrada é tentada somente frustrará e confundirá o jogador, que irá logo procurar um jogo diferente, mais consistente.

2.5.2.2 Jogadores esperam entender os limites do mundo do jogo

Ao jogar, um jogador quer entender quais ações são possíveis e quais não. Ele não precisa saber imediatamente quais ações são necessárias para uma determinada situação, mas ele deve entender quais ações são possíveis de serem realizadas e quais estão fora do escopo do mundo do jogo.

2.5.2.3 Jogadores esperam que soluções razoáveis funcionem

Uma vez que o jogador passou algum tempo jogando, ele começa a entender os limites do mundo do jogo. Ele resolveu diversos problemas, e viu o conjunto de soluções que lhe trarão retorno. Mais tarde durante o jogo, ele se deparará com um novo problema, e considerará a tentativa que ele considera uma solução razoável. Se ele tentar esta solução e a mesma falhar sem alguma boa razão, ele se frustrará, e se considerará enganado pelo jogo.

É tarefa do *designer* de jogo antecipar o que o jogador tentará fazer no mundo do jogo, e então assegurar que alguma coisa razoável aconteça quando o jogador tenta aquela ação.

2.5.2.4 Jogadores esperam direção

Bons jogos são caracterizados por permitir os seus jogadores fazerem o que eles desejarem, até certo ponto. Jogadores querem criar suas próprias estórias de sucesso, seus próprios métodos para ganhar um jogo, algo que seja unicamente deles. Mas ao mesmo tempo, precisam ter alguma idéia do que é esperado por parte deles de modo a serem bem-sucedidos no jogo. Não ter direção é muito como na vida real, e jogadores já possuem uma vida real. Muitos indivíduos estão provavelmente jogando para escapar de suas vidas reais, para fantasiar. Segundo Rouse III(2003), eles normalmente não jogam para simular vida real em seus computadores.

Jogadores querem ter alguma idéia de qual é o seu objetivo e receber alguma sugestão de como eles podem alcançar os seus objetivos. Com um objetivo, mas sem uma idéia de como alcançá-lo, jogadores inevitavelmente ziguezagueam, tentando tudo aquilo que eles conseguem imaginar, e se frustram quando suas tentativas não os conduzem nenhum pouco perto do seu objetivo.

Pelo outro lado, sem uma idéia do objetivo, jogadores são liberados para vaguear despropositadamente, talvez se deleitando com o cenário, se maravilhando com o mundo imersivo do jogo. Um exemplo deste tipo de jogo é o *Simcity*, um simulador de cidade, onde o jogador pode construir uma cidade simplesmente pela emoção de criar. Contudo, o truque consiste no seguinte fato: desde que o jogo simula realidade (construindo e liderando uma cidade), e o jogador sabe o que é considerado sucesso na realidade (uma cidade povoada repleta de estádios,

bibliotecas apalaçadas, e cidades felizes), ele tenderá a impor suas próprias regras para obter sucesso no jogo.

Eles se dedicarão a elaborar uma idéia de cidade perfeita, e tornar os cidadãos felizes e sua economia flutuante.

Assim, embora *Simcity* não tenha explicitamente um objetivo, a natureza do jogo e sua base na realidade encoraja os seus jogadores a disporem de seus próprios objetivos.

2.5.2.5 Jogadores esperam concluir suas tarefas de maneira incremental

Dado aos jogadores o entendimento de qual é o seu objetivo no mundo do jogo, ele gostam de saber se estão na trilha certa rumo ao cumprimento de seu objetivo principal. Segundo Rouse III (2003), a melhor maneira de se fazer isso é fornecer sub-tarefas ao longo do jogo, mas com uma recompensa proporcional. Sem fornecer este tipo de *feedback*, e se os passos necessários para se atingir um objetivo são particularmente longos, um jogador pode estar na trilha certa, mas não se dar conta disso.

Quando não há reforço positivo para se manter o jogador na trilha certa, ele estará sujeito a tentar algo diferente. E quando ele não consegue descobrir a solução para certo obstáculo, ele se decepcionará, e parará de jogar.

2.5.2.6 Jogadores esperam imergir no jogo

Uma vez que o jogador está concentrado no jogo, está em certo nível do mesmo, possui bom entendimento dos controles do jogo, está empolgado, ele não quer ter sua experiência abolida de forma abrupta. Certamente o jogo não deve apresentar *bugs*, e esta é a experiência mais desagradável possível. Dentro deste contexto, o jogador não quer pensar a respeito da interface de desenvolvimento do jogo. Se a interface não for elaborada para ser transparente e se ajustar ao restante da arte do jogo, a imersão do jogador será prejudicada.

Um aspecto importante sobre imersão de jogador é o seu personagem controlado. Se o personagem não é alguém de quem o jogador gosta, a imersão do jogador é rompida. A cada vez que um personagem diz algo que o jogador jamais diria se tivesse a chance, o jogador é lembrado de que ele está apenas num jogo, e de que ele não está no controle de seu personagem o quanto gostaria.

De modo a um jogador se imergir completamente num jogo, ele precisa ver a si mesmo em seu representante no mundo do jogo.

2.5.2.7 Jogadores esperam falhar

Jogadores tendem a depreciar jogos que eles conseguem terminar logo na primeira tentativa. Deve-se lembrar que as pessoas são levadas a jogar porque elas querem um desafio. E desafio implica necessariamente que os jogadores não sejam bem-sucedidos de início, e que muitas tentativas sejam praticadas até eles finalmente obterem sucesso. Uma vitória muito fácil é considerada uma vitória vazia.

É importante entender que os jogadores querem falhar por suas próprias escolhas, não devido às particularidades do jogo. Quando um jogador falha, ele deve identificar o que deveria ser feito ao invés da alternativa escolhida por ele, e assim reconhecer instantaneamente o porquê de sua tentativa falhar. Se o jogador sente que o jogo o derrotou por meio de algum truque, ele se desiludirá com o mesmo.

Jogadores precisam culpar a si mesmos por falhar, mas ao mesmo tempo o jogo deve ser desafiador o suficiente para que eles não obtenham sucesso na primeira vez.

Para Rouse III (2003), é uma boa idéia permitir ao jogador vencer um pouco no início do jogo. Isto lhe produzirá interesse, fazendo-o pensar que o jogo não é tão difícil assim. Os jogadores podem até mesmo desenvolver um sentimento de superioridade em relação ao jogo. A partir daí a dificuldade deve aumentar ao ponto de eles falharem. A esta altura, o jogador já estará envolvido com o jogo, terá investido o seu tempo com o mesmo, e quer continuar jogando para derrotar o obstáculo que o derrotou. Se um jogador é derrotado muito cedo por um jogo, ele

pode decidir que o mesmo é muito difícil para si, ou não entender quais tipos de recompensas conquistará se continuar jogando.

2.5.2.8 Jogadores esperam uma chance justa

Jogadores não querem se deparar com um obstáculo onde sua única chance de superá-lo é por meio de tentativa e erro, onde o erro resulte na morte de seu personagem e o fim do jogo. Um jogador deve ser capaz de encontrar um meio de superar o obstáculo por tentativa e erro, mas deve haver uma forma de o jogador descobrir a solução logo na primeira tentativa. Assim, estendendo esta regra para o jogo inteiro, mesmo sem ter nunca jogado certo jogo anteriormente, o jogador deve ser capaz de progredir através do jogo inteiro assumindo-se que ele é extremamente observador e habilidoso.

À medida que os jogadores continuam morrendo a cada tentativa realizada, eles notarão a inexistência de um modo real de evitar tais mortes. Ficarão então frustrados, e não perderão mais tempo com o jogo.

2.5.2.9 Jogadores esperam que não haja repetições

Uma vez que o jogador concluiu certo objetivo no jogo, ele não quer ter de concluí-lo novamente. Se um problema não for muito divertido de se resolver, e o

jogador precisar resolvê-lo durante todo o jogo, ele se decepcionará com a limitada criatividade do *designer* em lhe proporcionar novos desafios.

O baixo desejo dos jogadores repetirem suas ações é a razão de serem criados os mecanismos de salvamento do jogo.

2.5.2.10 Jogadores esperam não ter falsas esperanças

Não deve haver momento no jogo em que o jogador seja incapaz de ganhar de alguma forma, independentemente de quão improvável isso seja. Muitos jogos de aventuras antigos costumavam quebrar esta regra. Freqüentemente nestes jogos, se o jogador falhava em uma ação específica num dado momento, ou então falhava na captura de certo item localizado num ponto inicial do jogo, o jogador se tornava incapaz de terminar o jogo. O problema consiste no fato de o jogador não perceber este fato após muitas horas desperdiçadas de jogo. O jogo está essencialmente acabado, mas o jogador ainda continua jogando. Não há nada mais decepcionante do que jogar algo que não possa ser vencido.

2.5.2.11 Jogadores esperam fazer, não somente assistir

Por um determinado tempo, a indústria ficou extremamente empolgada com o surgimento dos filmes interativos. Durante este período, as cenas de vídeo dos

jogos ficaram cada vez mais longas. Rapidamente atores de filmes famosos começaram a estrelar tais vídeos.

Os jogos tornaram-se cada vez menos interativos. Então, para surpresa da indústria, os jogadores não gostaram destes tipos de jogos. Muitas companhias faliram, e todos dedicaram seu tempo perguntando-se o que deu errado.

Obviamente os jogadores sabiam, e os *designers* de jogos logo descobriram o que estava faltando. O problema era que os jogadores queriam fazer, e não assistir. E ainda hoje eles permanecem com essa concepção.

Deve-se lembrar que a razão pela qual as pessoas se dirigem aos jogos é que elas querem algo diferente do que um filme, livro, programa de rádio, ou os quadrinhos podem oferecer.

3 Materiais e Métodos

Conforme mencionado no capítulo 1, o projeto em questão adotou como metodologia de desenvolvimento o Processo Unificado, que é caracterizado por dividir o projeto em fases, que por sua vez sofrem uma ou mais iterações, onde cada iteração é constituída por um conjunto de etapas.

Cada uma das iterações pode ser considerada um mini-projeto, e tem como produto gerado um ou mais artefatos, e que servem para descrever o sistema sob uma ótica específica.

O trabalho de customização da metodologia para atender aos fins do projeto se mostrou necessário e suas motivações sãos explicitadas no capítulo 1. O que segue neste capítulo é a descrição de como se deu esta adaptação (quais fases e respectivas etapas foram adotadas e quais os artefatos foram gerados).

Primeiramente, para realizar as adaptações necessárias, levou-se em consideração que o projeto possui como característica o baixo risco e a pequena escala.

A partir da afirmação de Smith (2002) de que as fases de concepção e elaboração podem ser severamente reduzidas para projetos com as características acima citadas, as duas fases foram tratadas em uma única iteração. Como fases escolhidas dentro desta única iteração, escolheram-se as seguintes fases: captura de requerimentos, análise e projeto.

Outra simplificação realizada para o projeto foi a de não se adotar a fase de transição. Isso se deu por esta fase envolver questões que fogem do escopo estipulado para o projeto, como a distribuição de uma versão beta do sistema para usuários finais, e o levantamento de sugestões de melhorias no sistema feitas por esses mesmos usuários.

A fase de construção foi realizada em três iterações, descritas abaixo:

Primeira iteração – Nesta iteração foram executadas as etapas de implementação e de testes. A implementação baseou-se na arquitetura e demais artefatos gerados ao final da fase anterior, e os testes foram executados para verificar se o mini-projeto gerado ao final desta iteração atende os requisitos funcionais e não funcionais especificados também na fase anterior.

Como elementos implementados nesta iteração, procurou-se tratar da inclusão dos recursos visuais mais básicos do jogo, como o terreno e as unidades controladas pelo jogador. Nesta iteração, o código correspondente às ações das unidades permitiu somente a movimentação das mesmas ao longo do terreno, sem se preocupar com a questão de colisão entre as unidades. Nesta iteração foram estudados também os mecanismos de ajustes de câmera e animação das unidades.

Segunda iteração – Nesta iteração e na iteração seguinte, foram executadas as seguintes etapas: captura de requerimentos, análise, projeto, implementação e testes. Nesta segunda iteração pertencente à fase de construção, implementaram-se as funcionalidades previstas para as unidades do jogo, como ataque a uma unidade inimiga, e busca pelo caminho de menor custo (incluindo detecção e desvio de obstáculos).

Também foram implementados nesta fase os recursos básicos dos menus do jogo, como criação e salvamento de jogo. Por fim, nesta fase foram implementados os elementos básicos da inteligência artificial pertencentes ao *player* representado pelo computador, e que será o único adversário do *player* humano. Como exemplo de elementos básicos de inteligência artificial criados nesta etapa, pode-se citar a formação de grupos de ataque, em que duas ou mais unidades do computador se juntam para atacar as unidades do jogador.

Um ponto importante a ser considerado nesta iteração é que as etapas de captura de requerimentos, análise e projeto foram executadas para incorporar ao projeto de forma segura os módulos de inteligência artificial e de menus. A análise dos artefatos gerados em cada iteração, e que será mostrará logo abaixo, tornará essa percepção mais clara.

Terceira iteração – nesta última iteração foram implementadas as construções do jogo, bem como finalizados os códigos referentes às funcionalidades do player humano (contratação de unidades e venda de produtos), os menus do jogo e a inteligência artificial do adversário do player humano. Ao final desta iteração, o player representado pelo computador, além de formar grupos de ataque baseado em regras explicadas no item 3.7, também pode efetuar a contratação de unidades e a venda de produtos, seguindo a lógica definida para o jogo na seção Anexo A.

A figura abaixo ilustra o resultado da customização das fases e iterações do projeto:

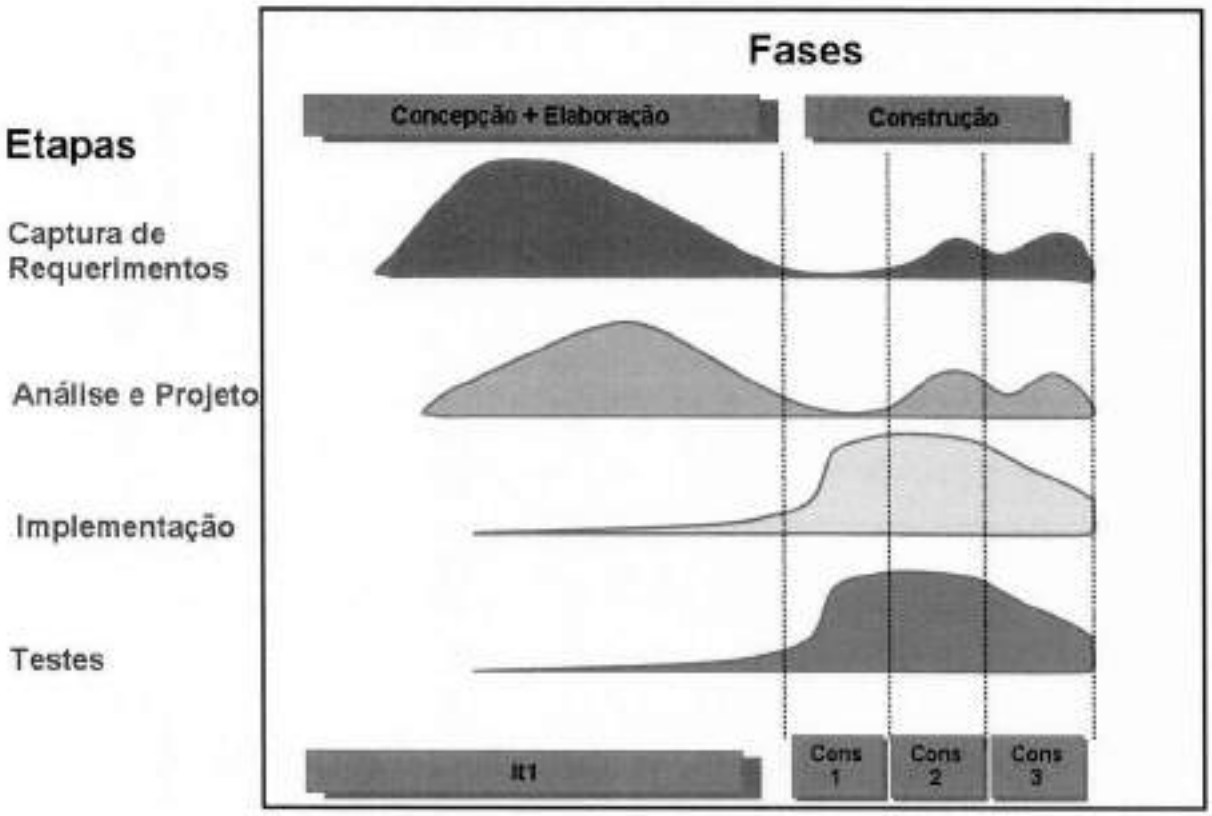

Figura 6 - Diagrama representando o UP customizado para o projeto

3.1 Fase de Concepção e elaboração

Conforme mostra a figura 4, as fases de concepção e elaboração foram unidas numa só iteração (representada na figura pelo termo It1). Pela figura, pode-se ver que as etapas de Captura de Requerimentos, análise e projeto (as etapas de análise e projeto estão representadas conjuntamente, por serem extremamente ligadas) foram adotadas para a execução desta iteração. Conforme dito anteriormente, cada iteração do UP corresponde a um mini-projeto, onde são gerados produtos de valor para as fases seguintes.

Para o caso desta única iteração, foram gerados os seguintes artefatos: descrição dos requerimentos, descrição de casos de uso (que descrevem os requisitos funcionais do sistema), requisitos não-funcionais, esboço da interface, e diagrama de classes.

Cada um dos artefatos gerados será descrito a seguir.

3.1.1 Descrição dos requerimentos

Este artefato descreve textualmente os requerimentos do software, servindo como esboço da especificação do sistema

O artefato de descrição dos requerimentos encontra-se no Apêndice A.

3.1.2 Descrição dos casos de uso

Os casos de uso são uma forma de captura dos requisitos funcionais do sistema, e segundo Jacobson et. al (1998) , descrevem um conjunto de seqüências e de ações que resultam num resultado observável e de valor para um determinado ator. Aqui entende-se ator como sendo um usuário do caso de uso, e não pode ser visto apenas como o usuário final do sistema. Pode, por exemplo, ser um subsistema ou uma instância de classe.

Este artefato assume grande importância para o projeto, e está localizado no Apêndice B.

3.1.3 Requisitos não-funcionais

Os casos de uso, embora sejam de extrema importância para o projeto, descrevem somente os requisitos funcionais do sistema.

Para a descrição dos requisitos não-funcionais é, portanto, necessária a adoção de mais um artefato, que é a lista de requisitos não funcionais.

Temos abaixo a lista de requisitos não-funcionais do sistema a ser produzido:

Manutenibilidade - O sistema deve apresentar elevada manutenibilidade, já que é intenção dos desenvolvedores transformarem futuramente o sistema em uma versão comercial. Atualmente, no mercado de jogos, os softwares precisam ser modificados (gerando-se versões com novos recursos) em intervalos de tempo cada vez mais curtos para não se tornarem obsoletos, e por isso uma elevada manutenibilidade se mostra necessária para que o software se torne competitivo no mercado.

Desempenho - O sistema deve apresentar alto desempenho no que diz respeito ao tempo de resposta e à taxa de renderização de quadros, na medida em que se caracteriza por ser em tempo real. Assim, a cada comando executado pelo usuário,

o sistema não pode demorar mais do que 1 segundo para responder à solicitação do usuário com alguma ação compatível e não deve apresentar taxas de renderização de quadro inferiores a 15 quadros por segundo.

Figura 7 – Artefato: Requisitos não-funcionais do Sistema

3.1.4 Esboço da Interface

Este artefato representa o que se pretende tornar visível ao usuário final do sistema.

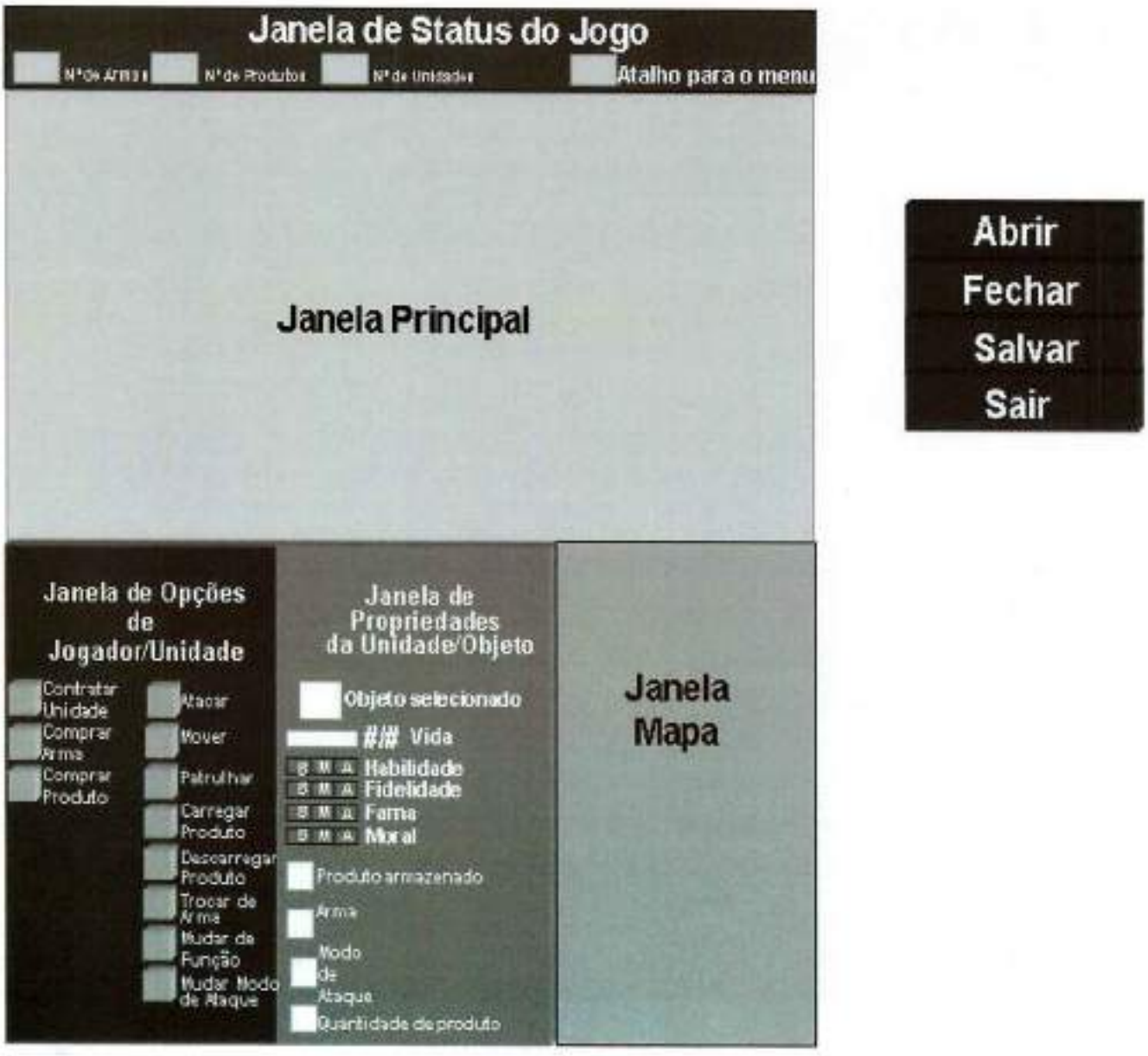

Figura 8 – Artefato: Esboço da interface gráfica do jogo

Um ponto importante a ser considerado é que a figura acima se encontra fora de escala, servindo apenas como um elemento ilustrativo.

Pela figura, vê-se 6 janelas básicas: Janela de Status do Jogo, Janela Principal, Janela de Opções de Jogador/Unidade, Janela de Propriedades da Unidade/Objeto, Janela Mapa e, por fim uma janela de menu (Abrir, Fechar, Salvar e Sair).

A janela de Status do jogo contém informações básicas para o jogador tomar suas decisões, como o número de armas e unidades, e a quantidade de produtos que estão sob sua posse. Além disso, dispõe em seu canto direito de um botão de atalho para a janela de menu, que será explicada adiante.

A Janela Principal é onde o jogador poderá clicar com botão esquerdo do mouse para selecionar as suas unidades e construções, e clicar com o botão direito para ordenar um ataque a determinado ponto. Neste caso, só atacarão as unidades que dispõem da função ataque, conforme explicado no Anexo A. É nesta janela também que o jogador poderá selecionar diversos objetos simultaneamente, para isso segurando o botão esquerdo do mouse e arrastando-o até se formar um retângulo visível na tela, representando a área de seleção. Serão selecionados os objetos dispostos dentro da área de seleção gerada.

Na janela principal, o jogador poderá visualizar também as unidades pertencentes ao computador e os NPC's que percorrem o cenário.

A Janela de Opções de Jogador/Unidade fornece todas as funcionalidades de que o jogador dispõe. Os três botões agrupados à esquerda correspondem às decisões macroscópicas que o jogador pode tomar. Quando se diz funções macroscópicas, entende-se pelos comandos que o jogador executa sem clicar sobre uma determinada unidade ou construção sua, mas que possuem grande importância para o seu sucesso no jogo.

Ao lado destes três botões de função macroscópica, esta janela possui oito botões, que ficam ativados quando uma unidade ou construção do jogador está selecionada. Estes oito botões representam as funções disponíveis para cada unidade e construção do jogador. Por exemplo, uma unidade do tipo personagem distribuidor (cujas características são explicadas no anexo A e cujas ações são descritas no anexo B por meio dos casos de uso), possui os seguintes botões ativados: Mover, Patrulhar, Carregar Produto, Descarregar Produto e Mudar de Função.

A janela de Propriedades da Unidade/Objeto contém informações a respeito do objeto do cenário selecionado no momento. Pode também conter informações a respeito de um conjunto de objetos selecionados ao mesmo tempo. Neste último caso, a janela exibirá apenas os nomes dos objetos selecionados.

A janela Mapa contém o mapa do terreno, e servirá como base para o jogador enxergar a disposição de suas próprias unidades e das unidades inimigas identificadas.

Por fim, tem-se a janela de menu, que é ativada sempre que o usuário clica com o botão esquerdo do mouse sobre o botão de atalho para o menu, botão este localizado no canto direito da janela de Status do jogo.

A janela de menu fornece os recursos de entrada e saída do jogo, contendo 4 funções principais: Abrir, Fechar, Salvar e Sair. Para desativar esta janela, basta clicar novamente sobre o botão de atalho para o menu.

3.1.5 Diagrama de Classes

Como artefato gerado ao fim da fase de concepção e elaboração, temos o diagrama de classes representado abaixo:

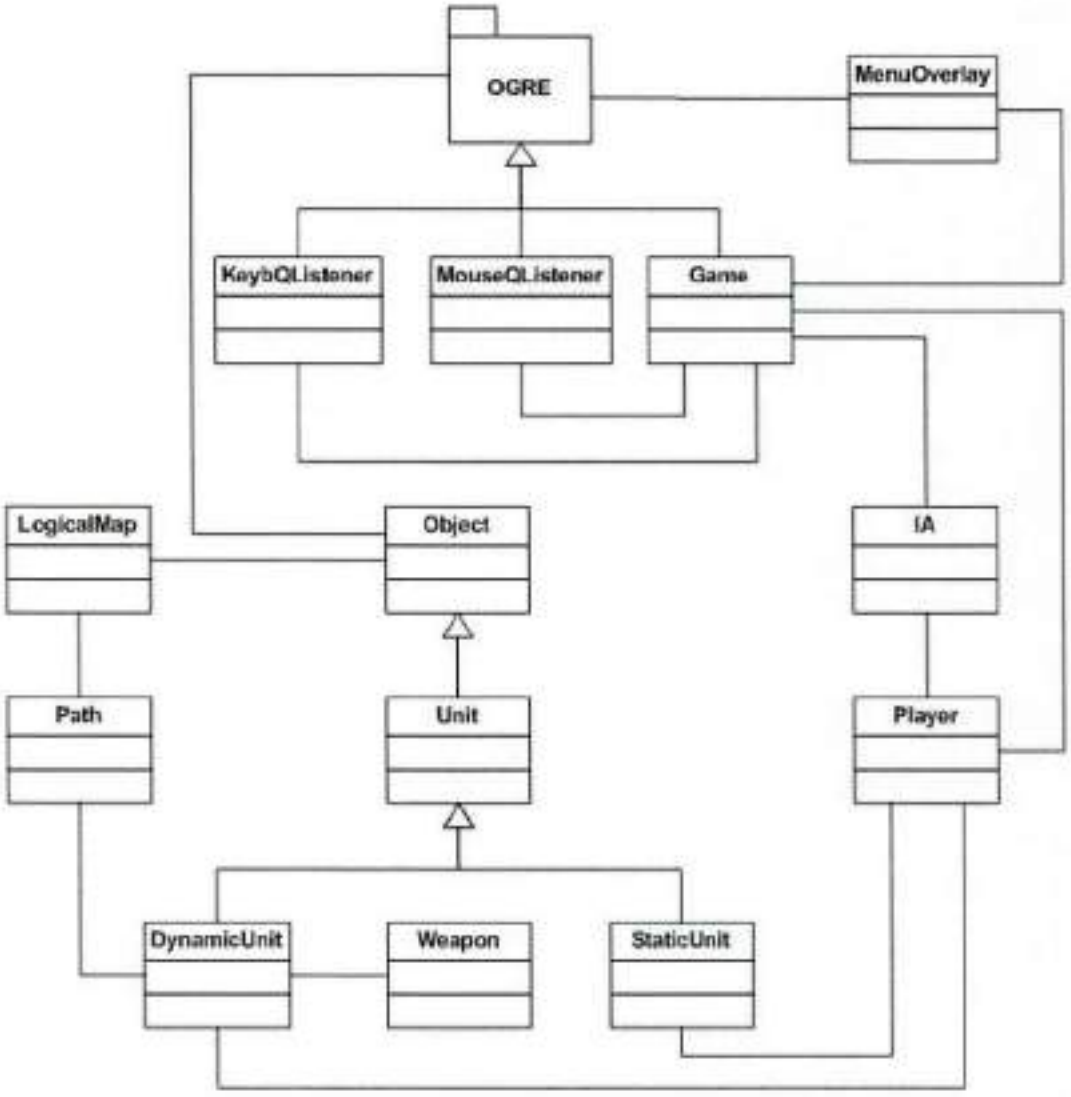

Figura 9 – Artefato: Diagrama de classes do projeto.

O diagrama de classes acima demonstra as classes do projeto e como elas se relacionam. Será dada aqui uma explicação geral do funcionamento de cada classe e suas relações. A partir da seção **Erro! A origem da referência não foi encontrada.**, é feito um detalhamento maior acerca das funcionalidades e implementação de cada classe.

Algumas classes utilizam o pacote OGRE, implementado pela engine Ogre: as classes MouseQueryListener, KeyboardQueryListener e Game são, cada uma, herança de uma classe do pacote Ogre. Conforme explicado posteriormente, listeners captam eventos dos hardwares, e estas classes foram implementadas para utilizar os eventos da maneira mais adequada.

A classe MenuOverlay utiliza a interface gráfica do Ogre, que provê suporte ao XML para a criação de layouts gráficos.

A classe Game é a principal classe do projeto. É nela que está contido o loop principal, através do qual ocorrem todas as atualizações do jogo. Além disso, a classe Game está ligada ao pacote Ogre, tendo acesso a informações gráficas do jogo, tais como o grafo de cena, dados da janela, etc.

A classe Game implementa o jogo, e todo jogo possui pelo menos um jogador: no projeto, os jogadores são representados pela classe Player. Um jogador possui unidades, que são representadas pelas classes StaticUnit e DynamicUnit, herdadas da classe Unit.

A classe Object implementa um objeto do jogo constituído por duas partes: um objeto gráfico, que é anexado ao grafo de cena do Ogre para ser renderizado, e um objeto lógico, que possui todas as informações necessárias para que o objeto seja

manipulado pelo jogo. Portanto, a classe Unit herda da classe Object e implementa uma unidade do jogo. A classe StaticUnit herda da classe Unit e implementa as unidades estáticas: as construções, enquanto a classe DynamicUnit herda da classe Unit para implementar as unidades dinâmicas.

As unidades dinâmicas se movimentam ao longo do território, e para isso utilizam o algoritmo de pathfiding, que traça rotas. O algoritmo de pathfinding por sua vez, utiliza informações das localizações dos objetos do jogo que estão armazenadas em um mapa lógico implementado pela classe LogicalMap.

Algumas unidades dinâmicas possuem armamentos, estes armamentos são representados pela classe Weapon.

Por fim, a classe IA é a classe que implementa a inteligência artificial do jogador computador. Ela está ligada à classe game para obtenção de dados de jogo para tomada de decisões e à classe Player para obtenção de dados dos jogadores e para manipular as unidades dinâmicas do jogador computador.

3.2 *Fase de Construção*

Conforme descrito anteriormente, a fase de construção foi dividida em 3 iterações.

Ao se analisar a figura 4, pode-se perceber esta divisão. Pela figura, nota-se também que a primeira iteração desta fase (identificada por cons1) dá ênfase essencialmente às etapas de Implementação e de testes. Isso se deve ao fato de ser uma iteração cujo intuito é a codificação imediata dos diagramas e esboços criados na fase anterior. Os testes foram realizados para se verificar como o sistema está atendendo aos requisitos não funcionais. Para a execução destes testes, foram utilizados os seguintes recursos: cronometragem e janela de estatísticas da engine gráfica OGRE.

O mecanismo de cronometragem foi adotado para se medir o tempo de resposta do sistema para cada ação realizada pelo jogador, seja por meio de clique ou arrastamento do mouse, seja pelo pressionamento de teclas do teclado.

A janela de estatísticas do OGRE fornece informações úteis a respeito do processamento gráfico em si. Uma das informações utilizadas para verificação da taxa de renderização é o campo *Average* FPS (Frames por segundo médio), identificado na figura abaixo.

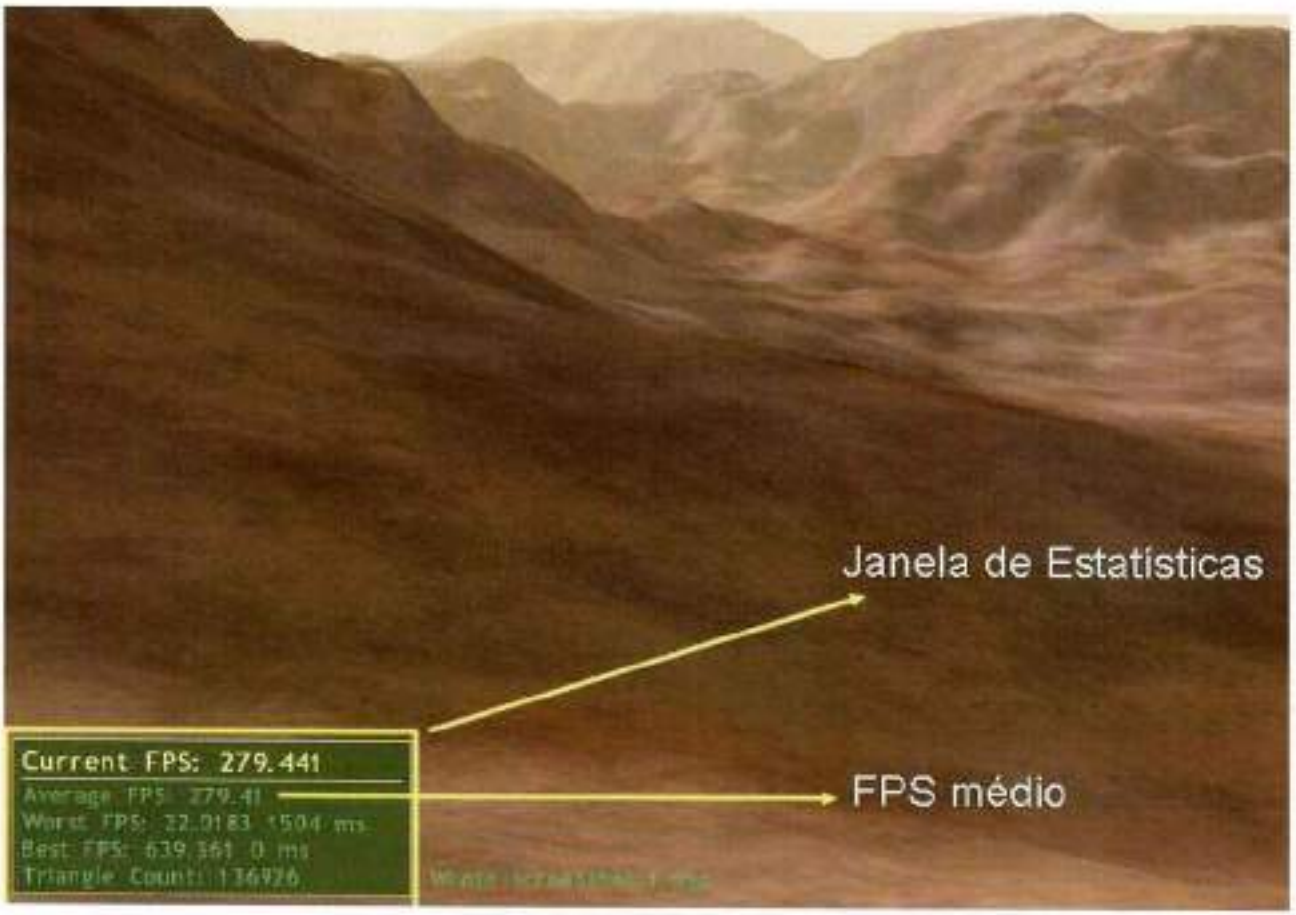

Figura 10 - Janela de Estatísticas do OGRE

Ao final desta iteração foi gerada uma primeira versão codificada do sistema, e cuja aparência gráfica é mostrada na figura abaixo:

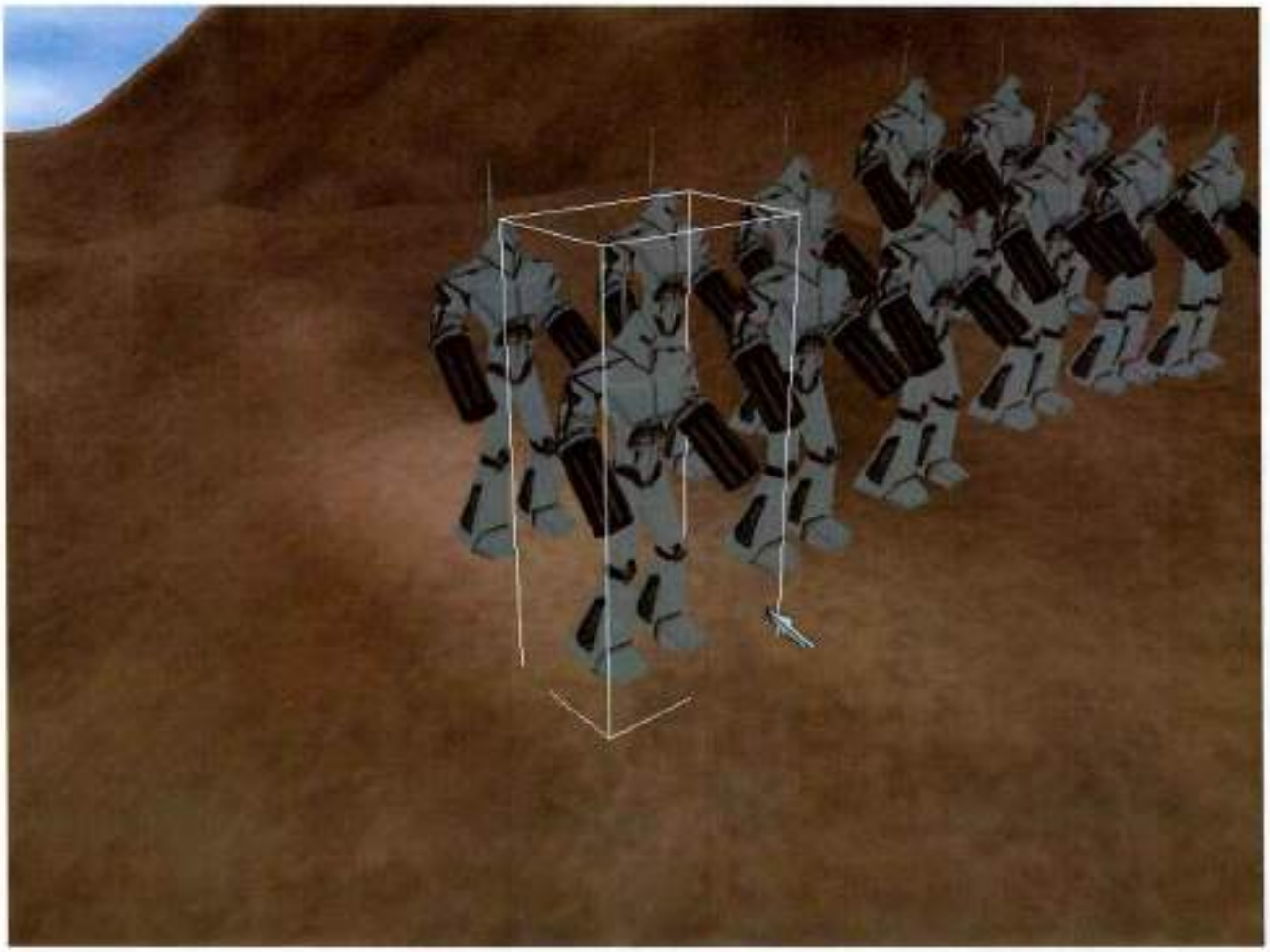

Figura 11 – Resultado da primeira iteração da fase de construção

As funcionalidades implementadas ao final desta iteração foram descritas no inicio do capítulo 3.

A segunda iteração da fase de construção (denotada na figura 4 por cons2) aproveitou o código gerado na iteração anterior, e incrementou as funcionalidades geradas pela mesma. As funcionalidades implementadas por esta iteração foram descritas no início do capítulo 3, e a aparência gráfica do jogo ao final desta iteração é mostrado na figura abaixo:

Figura 12- Resultado da segunda iteração da fase de construção

Analisando a figura 4, pode-se ver que nesta iteração também foram executadas as etapas de captura de requerimentos, de análise e de projeto. Isso ocorre porque nesta fase foi iniciada a criação dos módulos MenuOverlay e IA, ambos mostrados no diagrama de classes da figura 7. Para a implementação do módulo de IA, foi necessária a definição dos requisitos da inteligência artificial do jogo, bem como a criação dos casos de uso relacionados com este módulo, e que não tinham sido criados anteriormente. A criação dos casos de uso da Inteligência Artificial deu-se nesta iteração porque foi visto como um espelho dos casos de uso

cujo ator é o jogador humano. Assim, procurou-se definir ao longo do projeto quais eram as funcionalidades viáveis de serem implementadas, que depois seriam disponibilizadas para o jogador e, em seguida, disponibilizadas ao player representado pelo computador.

Assim, esta iteração apresenta como artefato adicional os casos de uso da inteligência artificial, que são descritos no anexo B.

A terceira e última iteração (identificada na figura 4 por cons3) aparece como uma evolução da iteração anterior. As funcionalidades do jogo como um todo são concluídas nesta iteração, e o artefato gerado nesta fase é o produto final.

A descrição das funcionalidades implementadas nesta iteração foram descritas no início do capítulo 3.

Figura 13 - Versão final do projeto

3.3 Listeners

Um 'listener' é uma interface desenvolvida para ser chamada sempre que um

evento particular ocorre. A engine Ogre dispõe de diversos Listeners, sendo que para o projeto em questão foram utilizados 4 tipos de Listeners: FrameListener, KeyListener, MouseListener e MouseMotionListener.

3.3.1 FrameListener

Esta classe define um listener relacionado com eventos de *frame*, isto é, sempre que um evento de renderização for detectado. No projeto em questão, a classe Game é uma herança da FrameListener.

Uma característica importante da classe FrameListener é a presença dos métodos frameStarted e frameEnded, que são usados automaticamente pelo objeto Root para executar o loop principal de renderização. Mais detalhes sobre o loop principal de renderização estão descritos na seção Loop Principal. A classe Root é o ponto de partida para a aplicação cliente, e uma instância sua deve ser sempre criada antes de ser chamada qualquer operação da engine Ogre.

O método frameStarted é chamado quando um frame está prestes a ser renderizado, e retorna verdadeiro para começar a renderização, ou falso para

abortar a renderização e sair do loop de renderização. O método frameEnded é chamado assim que um frame acabou de ser renderizado, e retorna verdadeiro para continuar com o processo de renderização de frames, ou falso para sair do loop de renderização.

3.3.2 KeyListener

Este listener está relacionado com os eventos de teclado, e é herdado a partir de uma classe abstrata do Ogre chamada EventListener.

Esta classe possui diversos métodos para captura de ações do teclado, mas para o projeto são utilizados basicamente dois métodos: keyPressed e keyReleased.

O método keyPressed é acionado pela classe sempre que uma tecla for pressionada, e o método keyReleased é acionado sempre que uma tecla for liberada.

Um ponto importante tratado pelo grupo foi a questão da taxa de leitura do teclado, pois se o usuário mantiver uma determinada tecla pressionada, a mesma pode ser acionada diversas vezes, o que pode se tornar indesejável dependendo da tecla pressionada.

Para resolver este problema, fez-se uso de um mecanismo de temporização, explicado na seção Mecanismo de Temporização.

3.3.3 MouseListener

Este listener está relacionado com eventos discretos de mouse (isto desconsidera o movimento de mouse, que não é considerado discreto), e é herdado a partir da classe EventListener.

A classe MouseListener possui diversos métodos, mas para o projeto são utilizados basicamente mousePressed, e mouseReleased.

O método mousePressed é acionado sempre que uma tecla de mouse é pressionada. No projeto, este método faz a verificação de qual tecla do mouse foi pressionada, e armazena essa informação e as coordenadas do ponteiro do mouse em variáveis auxiliares, que são utilizadas por outros métodos do projeto. O método mouseReleased é acionado sempre que uma tecla de mouse é liberada. Seu comportamento é semelhante ao método mousePressed, exceto pelo fato de ser verificada qual a tecla estava sendo pressionada antes de ser liberada.

3.3.4 MouseMotionListener

Este listener está relacionado com eventos de movimento do mouse e, da mesma forma que as classes KeyListener e MouseListener, é herdado a partir da classe EventListener.

A classe MouseMotionListener possui diversos métodos, mas o projeto utiliza os métodos mouseMoved e mouseDragged.

O método mouseMoved é acionado sempre que o mouse é movimentado. No projeto, este método armazena as coordenadas relativas do ponteiro do mouse em variáveis auxiliares, passando-as como parâmetro para a função injectMouseMove, que desenha o ponteiro do mouse na tela.

O método mouseDragged é acionado sempre que o mouse é arrastado enquanto uma tecla sua é mantida pressionada. No projeto, este método verifica qual a tecla que está sendo pressionada enquanto o mouse é arrastado, e armazena esta informação e as coordenadas do ponteiro do mouse em variáveis auxiliares, que são usadas por outros métodos do projeto.

3.4 O Loop Principal

O projeto em questão possui um *loop* principal, *ou game loop*, que é executado repetidamente enquanto não for acionado algum evento que provoque a saída do programa (tecla ESC pressionada, erro fatal, etc.). Dentro do *game loop* está o código do projeto que será executado de maneira cíclica.

Para a execução deste *loop*, o grupo utilizou o próprio *loop* de renderização presente na engine Ogre.

O *loop* principal de renderização do Ogre consiste basicamente de 3 etapas:

- O Objeto Root chama o método frameStarted do FrameListener criado;
- O Objeto Root renderiza um frame;
- O Objeto Root chama o método frameEnded do FrameListener criado.

Este *loop* é então executado até que o FrameListener retorne falso a partir dos métodos frameStarted ou frameEnded.

Pela descrição acima, pode-se concluir que qualquer trecho de código colocado dentro dos métodos frameStarted ou frameEnded será executado repetidamente, já que os mesmos são acionados ciclicamente pelo objeto Root.

Para o projeto, o código a ser executado de maneira cíclica foi inserido dentro do método frameStarted.

Este código inserido dentro do método frameStarted (sendo portanto executado periodicamente) faz uso de um mecanismo de temporização, dos

listeners de mouse e de teclado (descritos na seção Listeners), bem como faz a atualização das unidades e faz a inteligência artificial do jogo.

3.4.1 Mecanismo de temporização

Como descrito na seção Loop Principal, o projeto utiliza como *game loop* o próprio *loop* de renderização da engine Ogre. Desta forma, todas as ações colocadas dentro do método frameStarted serão executadas com a mesma freqüência que a taxa de renderização do Ogre. Contudo, tal restrição foi evitada pelo grupo, de acordo com os motivos descritos na seção 2.4 (Game Design).

Para se fazer a dissociação entre a taxa de renderização do Ogre e a freqüência de execução das ações do jogo, empregou-se um mecanismo de temporização.

O Mecanismo de Temporização consiste de uma variável incremental, de uma taxa de incremento, e de uma variável de referência.

A variável incremental é do tipo Real, e é inicializada com valor 0. A cada vez que o método frameStarted é chamado, esta variável sofre um incremento de valor igual à taxa de incremento. Para o projeto, foi utilizado como taxa de incremento o intervalo de tempo decorrido entre a última renderização completa de frame e o instante atual.

A variável de referência representa o valor máximo que a variável incremental pode atingir após sucessivos incrementos. Após a variável incremental atingir o valor da variável de referência, a variável incremental é zerada.

Assim, antes de ser chamado um método a ser executado dentro do frameStarted, faz-se a comparação entre a variável incremental e a variável de referência. Se a variável incremental for menor do que a variável de referência, o método não é executado. Caso contrário, o método é executado e em seguida a variável incremental é zerada.

Como exemplo de uso do mecanismo de temporização, temos a regulação da taxa de leitura do teclado:

- A variável incremental é inicializada com o valor lógico 0;

- A cada chamada do método frameStarted , a variável é incrementada com o valor da taxa de incremento;

- Se um usuário pressiona certa tecla, verifica-se se a variável incremental é menor ou igual à variável de referência. Se sim, o evento da tecla é ignorado. Se não (o valor da variável incremental é maior que a variável de referência), o evento da tecla é tratado e em seguida a variável incremental é zerada.

De acordo com o exposto acima, se um usuário pressiona uma determinada tecla, o código faz a verificação de qual tecla foi pressionada e qual o valor da variável incremental. O controle de incremento e zeragem da variável incremental permite então que a taxa de leitura do teclado seja regulada conforme desejado.

3.4.2 Listeners do Mouse e do Teclado

O *game loop* do projeto em questão faz uso dos listeners do mouse e do teclado.

Primeiramente, verifica-se o valor da variável incremental. Se a mesma for menor do que a variável de referência, os métodos de mouse e teclado são

ignorados. Caso, contrário, os mesmos são executados de acordo com o evento ocorrido, e em seguida a variável incremental é zerada.

Como eventos de mouse tratados pelo *game loop*, temos:

- Mouse arrastado com a tecla direita pressionada: giro de câmera;
- Tecla esquerda pressionada: seleção simples de unidades (explicado na seção Funcionalidades do Mouse);
- Tecla direita pressionada: ataque de unidades (explicado na seção Ataque de Unidades);
- Tecla esquerda liberada: seleção de unidades por área (explicado na seção Funcionalidades do Mouse).

3.5 Funcionalidades do Mouse

O jogo em questão apresenta um conjunto de funcionalidades relacionadas com os eventos do mouse. As funcionalidades são: seleção de unidades, mudança de status da unidade, overlay.

3.5.1 Seleção de unidades

Existem dois modos de seleção de unidades: seleção simples e seleção por área.

Para os dois modos faz-se uso de uma classe especial, a RaySceneQuery. Esta classe possui a capacidade de traçar raios através da cena, e que são utilizados para listar objetos presentes na mesma. O procedimento se dá pela

seguinte maneira: um raio é traçado a partir da posição da câmera até a posição em que o mouse se encontra. Feito isso, a classe RaySceneQuery armazena uma lista de objetos interceptados pelo raio, e que serão analisados pelo código para se fazer a seleção das unidades.

Na seleção simples de unidades, primeiramente é feita a captura das coordenadas do ponteiro do mouse quando o botão esquerdo do mesmo é pressionado, e o seu armazenamento em variáveis auxiliares.

Em seguida, é lançado o raio a partir da posição da câmera até a posição registrada nas variáveis auxiliares. Feito isso, é criado um laço que percorre a lista de todas as unidades presentes no jogo, e a cada uma é atribuído o status de não-selecionada.

Após a execução do laço, é percorrida a lista de objetos interceptados pelo raio, fazendo-se a seguinte verificação: se o objeto em questão é uma unidade do jogador, é atribuído a esta o status de selecionada. Após isso, verifica-se o próximo objeto da lista de objetos interceptados pelo raio.

Na seleção de unidades por área, a diferença se dá pelo seguinte fato: são registradas em variáveis auxiliares, além das coordenadas do ponteiro do mouse quando o botão esquerdo do mesmo é pressionado, também as coordenadas do ponteiro do mouse quando o botão esquerdo do mesmo é liberado. Após, é traçado o raio a partir da posição da câmera até a posição onde o botão esquerdo do mouse foi liberado.

Como na seleção simples de unidades, é percorrida a lista de objetos interceptados pelo raio, dando-se o mesmo tratamento. A diferença consiste na

realização de uma etapa adicional: é traçado um retângulo cuja diagonal é formada pela ligação entre o ponto em que o botão esquerdo do mouse foi pressionado e o ponto em que o botão esquerdo do mouse foi liberado. Após se traçar este retângulo, é feita a verificação da posição de cada unidade do jogador, e atribuindo-se o status de selecionada àquelas situadas dentro da região estabelecida pelo retângulo traçado.

3.5.2 Overlay

Os overlays são mecanismos que possibilitam mostrar elementos 2D ou 3D numa camada sobreposta àquela do conteúdo principal das cenas renderizadas.

Eles têm o objetivo de criar efeitos como menus (com opções de jogo, por exemplo: salvar, sair, etc.), displays informativos com características pertinentes ao jogo, painéis, dentre outros.

A implementação dos overlays ou menus de interação com o usuário é de extrema importância para a usabilidade e jogabilidade do sistema, uma vez que se trata de um jogo de estratégia em tempo real, no qual o mundo simulado envolve uma certa complexidade e há uma gama mais vasta de comandos ou ações que o usuário pode realizar no decorrer do jogo.

Usando a engine escolhida (OGRE), é possível implementar os overlays de duas maneiras: 1- através do próprio código-fonte do jogo, utilizando as classes e métodos apropriados para a GUI; 2- através da utilização de scripts baseados em

XML que definem a estrutura do overlay e, posteriormente, são manipulados no código.

No projeto desenvolvido, decidimos utilizar a segunda opção pelo fato de ela apresentar maior flexibilidade, possibilitando a alteração rápida da estrutura dos overlays sem necessidade de recompilar o código. Além disso, apresenta maior padronização por ser baseada em XML.

Assim, o script provê o layout para o overlay, definindo os tipos de objetos gráficos utilizados, seu tamanho, posição, etc. além de estabelecer uma estrutura hierárquica entre os elementos do overlay. Por exemplo, pode-se definir um layout em que haja uma janela principal, dentro da qual serão colocados elementos como listas de seleção, input boxes, botões e assim por diante. Abaixo, segue um exemplo de script:

```
<?xml version="1.0" ?>
<GUILayout>
<Window Type='DefaultGUISheet' Name='root'>
    <Window Type='DefaultGUISheet' Name="OgreGuiDemo">
        <Property Name='Size' Value='w:1 h:1' />

            <Window                                    Type='TaharezLook/Button'
Name='OgreGuiDemo/TabCtrl/Page1/NewButton'>
                <Property Name="Position" Value="x:0.1 y:0.25" />
                <Property Name='Size' Value="w:0.8 h:0.07" />
                <Property Name='Text' Value='New Game' />
            </Window>
    </Window>
```

```
</Window>
</GUILayout>
```

Figura 14 - Exemplo de script para overlay.

Feito isto, implementou-se uma classe especialmente para lidar com as ações relacionadas ao menu e interliga-las às funcionalidades do jogo. Essa classe, chamada *MenuOverlay*, encapsula os métodos para criação, manipulação e destruição dos menus.

A classe principal (*Game*) possui uma instância da *MenuOverlay* e, durante a fase de setup do jogo, é chamado o método **createMenu()** do objeto do menu pertencente à *Game*. Então, esta instância/objeto se encarrega de carregar o script citado acima e, a partir daí, renderizar o menu na tela. Também neste momento é feito o *setup* dos eventos que estão associados aos elementos pertencentes ao menu.

Os **eventos** são disparados através da interação do usuário com o menu, ou seja, para cada tipo de elemento há um conjunto de ações que o usuário pode realizar que disparam determinados eventos. Por exemplo, o clique de um botão da interface, a digitação de alguma palavra num "*textbox*" ou seleção de uma opção numa lista.

Esse tipo de interface gráfica é chamada de **event-based GUI** (Interface baseada em eventos). Assim, criou-se o menu quando é feito o setup do jogo, os eventos relacionados aos elementos a ele pertencentes são ligados à funções específicas. Por exemplo, quando o botão "Save" do menu é clicado, a classe que lê eventos do mouse detecta este clique e a função *saveRequested()* da classe

MenuOverlay é executada e sinaliza à classe principal *Game* que é preciso salvar o jogo. Esta, por sua vez, chama a função *saveGame()*, que pertence ao escopo de funções da classe principal.

Desse modo, a classe *MenuOverlay* se comunica com as classes chamadas **Listeners** (que são responsáveis pela captura de eventos de entrada e saída do mouse e do teclado) e com a classe principal. Os listeners, ao obterem entradas/saídas relacionadas ao sistema de menu (cujos eventos já foram cadastrados previamente), dispara as chamadas às funções cadastradas para esses eventos. Essas funções, por sua vez, sinalizam para a classe principal que toma as devidas ações de acordo com a sinalização.

3.6 Unidades

Neste tópico são descritos os aspectos relacionados com as unidades dinâmicas do jogo, como o processo de atualização de suas atividades, bem como o processo de criação de unidades, disponível tanto ao *player* humano quanto ao computador.

3.6.1 Atualização de unidades

O Main Loop é responsável por chamar a atualização de cada unidade do jogo, porém quem realiza a atualização é a própria unidade, isto é, os métodos de atualização dentro da classe DynamicUnits. Tudo que o Main Loop deve fazer é

chamar o método de atualização da unidade, passando o valor do intervalo de tempo decorrido entre a última atualização e a atual.

A atualização da unidade pode ser dividida em basicamente quatro atividades:

- Atualização de status;
- Movimentação;
- Ataque;
- Trabalho.

Essas atividades são efetuadas em conjunto, pois são interdependentes. O método de atualização inicia verificando qual o status da unidade: se ela está viva, se ela está se movimentando ou trabalhando, se ela tem a intenção de atacar e em caso positivo, se ela está perto de seu alvo. Após isso uma ação é executada em função do status:

Caso a unidade esteja somente parada, o sistema simplesmente atualiza a animação da unidade, como será explicado posteriormente. Porém se a unidade estiver parada, mas com o status de "trabalhando", um temporizador é iniciado com um valor pré-determinado e quando ele dispara o jogador recebe uma quantidade de dinheiro, também pré-determinada. Vale lembrar que apenas a unidade do tipo distribuidor tem a funcionalidade de trabalhar e apenas a unidade do tipo soldado tem a funcionalidade de atacar. No entanto, este algoritmo de atualização não faz a verificação do tipo, essa verificação é feita no momento em que é dado o comando de atacar ou trabalhar a uma unidade.

Caso a unidade esteja se movimentando e não tenha a intenção de atacar, uma ação de movimentação é tomada, isto é, o algoritmo irá atualizar a posição da

unidade de acordo com o tempo decorrido, a velocidade de movimentação da unidade e o sistema de Pathfind. Em seguida é feita uma verificação da posição da unidade em relação à posição objetivo da unidade e caso elas forem iguais à unidade tem seu status de movimentação alterado para "parado".

Caso a unidade esteja se movimentando e tenha intenção de atacar uma ação igual a anterior é tomada, exceto por um detalhe: a verificação da posição da unidade em relação ao objetivo, isto é, sua unidade alvo, deverá ser feita de forma diferente. As unidades são inicializadas com um objeto da classe Weapon que possui as características do armamento utilizado pela arma, como quantidade de dano, alcance e cadência. Portanto caso a distância entre a unidade e o seu alvo seja menor ou igual ao alcance do armamento utilizado pela unidade o algoritmo mudará o status de movimentação para "parado" e o status de próximo do alvo para verdadeiro.

Caso a unidade tenha a intenção de atacar e estiver próxima de seu alvo, um temporizador é configurado de acordo com a cadência do armamento, isto é, a freqüência de disparo da arma. Quando esse temporizador disparar, um ataque é feito e o temporizador é novamente configurado com o valor da cadência do armamento.

O ataque é uma funcionalidade de interação entre as unidades: para atacar, a unidade atacante verifica o status da unidade alvo. Caso a unidade alvo esteja viva, o atacante gera um dano na unidade alvo chamando um método da unidade alvo. O dano é o decremento no atributo de vida da unidade, é um número

randômico que varia de zero ao dano máximo – característica do armamento utilizado pelo atacante.

A unidade que receber um dano deverá subtraí-lo de seu atributo de vida e se este for menor que zero deverá modificar seu status para "morto", sem movimentação e sem intenção de atacar. Caso contrário, a unidade tentará atacar a unidade que enviou o dano, isto é, terá seu status de movimentação e intenção de atacar para verdadeiro, além de atribuir a seu atributo "alvo" o seu atacante.

Finalmente, é verificado se a unidade está viva. Na verdade esta é a primeira verificação feita pelo algoritmo, já que caso a unidade não esteja viva nenhum dos passos anteriores deverá acontecer, porém este passo é explicado por último para uma melhor seqüência lógica.

No momento em que o status da unidade é alterado para "morto", um temporizador é inicializado. Portanto, se a unidade estiver morta o algoritmo atualizará o temporizador, e no momento do disparo fará com que a unidade seja removida do jogo. Esse tempo é necessário para que as outras unidades que porventura estejam atacando a unidade possam perceber que a unidade "morreu", isto é, ler o status da unidade alvo no momento do ataque. Se esse temporizador não existisse as unidades tentariam acessar um objeto que não existiria e um erro iria ocorrer.

Ao mudar o status de uma unidade, a animação utilizada em sua renderização também é modificada. O algoritmo verifica o conjunto de status da unidade e atribui uma animação de acordo com esse conjunto: animação da unidade andando, atacando, trabalhando, parada ou morrendo.

No final do método de atualização de unidade, o algoritmo faz a chamada do método do Ogre que faz atualização de animação, o addTime, onde é passado o valor do intervalo de tempo decorrido entre a última atualização e a atual.
Além dessas ações, algumas funcionalidades foram inseridas no algoritmo:

- Uma unidade atacante sempre perseguirá a unidade alvo caso esta não esteja ou venha a sair do alcance do armamento da unidade atacante;
- A unidade sempre olhará para o alvo quando estiver atacando-o, isto é, o algoritmo sempre atualizará a direção da unidade para a posição de seu alvo;

Uma unidade somente revidará um ataque caso já não esteja atacando outra unidade ou não esteja se movimentando.

3.6.2 Contratação de Unidades

A contratação é um recurso essencial para o jogo: sem a contratação não se pode aumentar o número de unidades de um jogador. O jogador e a inteligência artificial utilizam da contratação para aumentar o seu domínio no jogo. É uma função básica dos jogos de estratégia, sem a criação de unidades não é possível montar uma estratégia e o jogo seria finalizado logo no começo.
De acordo com o documento de game design, a contratação deveria fazer com que uma unidade passasse do jogador gaia (a "mãe-natureza", isto é, o jogador controlado pelo computador que não é inimigo de ninguém e tem apenas a função de transitar pelo mapa do jogo), para o jogador que a contratou. Assim, quando uma unidade fosse eliminada, seria criada outra unidade para o jogador gaia, para que o

número total de unidades do jogo fosse constante. Porém, isso não foi implementado devido à falta de tempo no cronograma do projeto.

Portanto, a contratação implementada foi a seguinte: ao contratar uma unidade – através do menu para o jogador humano ou através de chamada de função para o jogador computador – o algoritmo faz a verificação do dinheiro que o jogador possui. Caso o dinheiro do jogador seja menor que o preço pré-determinado da unidade, a ação é cancelada. Caso contrário, é descontado o preço da unidade do dinheiro, é criada uma nova unidade no jogo do tipo escolhido pelo jogador, e esta é alocada para o jogador.

Perde-se o controle do número de objetos que podem ser criados no jogo além da falta de correspondência com a documentação original, porém a implementação ficou muito mais simples.

3.7 Pathfinding

Para a solução do problema de busca de caminhos durante a movimentação das unidades, foi utilizado no projeto o algoritmo A*, descrito anteriormente.

Inicialmente, antes da implementação do algoritmo, era necessário que o mapa fosse descrito numa estrutura lógica capaz de ser utilizada pelo algoritmo na busca dos caminhos. Assim sendo, dividimos o mapa físico em pequenas áreas quadradas (cujo tamanho é parametrizável). Cada uma dessas pequenas áreas representa uma célula no mapa lógico e pode ser definida como *walkable* (livre) ou *unwalkable* (ocupada) e é com base nessas informações que o algoritmo define o caminho a ser

percorrido pela unidade. Tal mapa lógico foi implementado na classe LogicalMap, que é um **singleton** (GAMMA, 1995).

A característica principal que permite a um singleton possuir uma e apenas uma instância da classe em todo o programa é a existência de um atributo estático que é ponteiro para objetos do mesmo tipo da classe que está sendo criada (no exemplo abaixo, a classe *Singleton* possui o atributo **static Singleton *single** que é um ponteiro para um objeto estático do tipo *Sigleton)*. A utilização de um atributo estático é de fundamental importância, pois um atributo estático de uma classe possui o mesmo valor para todos os objetos instanciados dessa classe.

Outra caracteristica importante é a existência do método **static Singleton* getInstance()** . O método getInstance() verifica se o atributo *single já foi inicializado. Caso não tenha sido inicializado, cria um objeto Singleton e atribui ao atributo *single, retornando-o ao chamador do método. Caso o atributo já tenha sido inicializado, retorna o valor atual do atributo. O fato de o método ser estático permite que o método seja chamado sem a necessidade de instanciação do objeto, podendo ser chamado diretamente a partir da classe.

Para se utilizar um singleton implementado como o modelo acima, cria-se um ponteiro para um objeto do tipo Singleton mas, ao invés de se criar uma nova instancia do objeto para se atribuir ao ponteiro, utiliza-se o método getInstance() para se obter a instancia já existente do objeto. Uma vez obtido o objeto já instanciado, pode-se utilizar o ponteiro para acessar os métodos do objeto.

A classe LogicalMap é responsável por todas as atividades relativas ao mapa lógico, dentre as quais se encontra a busca de caminho, através do método

FindPath(), que implementa o algoritmo A*. O método recebe como entrada a posição inicial da unidade e seu destino final e, através da verificação do mapa lógico, define o menor caminho livre (walkable) que liga os dois pontos. Uma vez determinado o caminho, o método retorna um objeto Path, que armazena o caminho encontrado e provê métodos para movimentação através desse caminho.

A classe Path armazena todos os pontos intermediários do caminho que foi definido pelo método FindPath() e a quantidade total de pontos desse caminho, além de um contador que indica a posição da unidade dentro desse caminho lógico.

Assim, no inicio da movimentação e cada vez que a unidade alcança um desses pontos em sua movimentação, ela consulta o objeto Path para obter o próximo ponto do caminho e se movimentar até ele. Quando o objeto Path recebe a consulta (através do método ReadPath()) ele retorna o próximo ponto do caminho no mapa lógico e atualiza a contagem da posição da unidade, atribuindo o valor *false* para o flag pathFound quando esse contador chegar ao total de pontos intermediários do caminho. Como o método retorna a posição lógica do próximo ponto do caminho, é trabalho da unidade determinar a posição física desse próximo ponto e realizar a movimentação até ele.

Para caminhar através do caminho determinado pelo método FindPath() a unidade realiza, em seu ciclo de atualização, uma seqüência de operações para garantir o resultado da movimentação.

No inicio da movimentação, a unidade chama o método ReadPath (do objeto Path) para determinar o primeiro ponto intermediário do caminho. A partir daí, a cada ciclo de movimentação, a unidade verifica se alcançou o ponto determinado

anteriormente e continua andando em direção a ele (em linha reta) caso ainda não o tenha alcançado.

Uma vez que o ponto intermediário já tenha sido alcançado, a unidade verifica se o flag pathFound é *true* ou *false*. Se o valor do flag for *false* significa que foi atingido o ultimo ponto do caminho, então a unidade encerra sua movimentação. Caso o valor do flag seja *true*, significa que ainda existem outros pontos do caminho a serem percorridos, então a unidade obtém o próximo ponto lógico (através do método ReadPath()), converte essa posição lógica em uma posição física (o centro do quadrado físico determinado pela célula lógica) e caminha em direção a esse ponto.

A movimentação de uma unidade entre dois pontos intermediários do caminho se dá em passos menores que a distancia entre os centros das células lógicas. Assim sendo é possível que, entre uma atualização e outra, a distancia percorrida não faça a unidade mudar sua posição lógica, apesar de mudar sua posição física. Devido a essa condição, a unidade deve sempre verificar se o passo físico que ela vai realizar a faz mudar de posição lógica. Caso o passo não seja suficiente, ela apenas realiza a atualização de sua posição física.

Caso o passo físico a ser realizado pela unidade também represente uma mudança de posição lógica da unidade, ela deve verificar se a posição lógica em questão está livre ou não. Essa verificação é necessária para que se evite colisões com outras unidades que estejam se movimentando pelo terreno, fazendo o papel de uma *bounding box* simplificada. Se a posição estiver ocupada, a unidade re-calcula seu caminho, de forma a desviar desse novo obstáculo. Caso o contrário, a

unidade realiza o passo e atualiza sua posição, marcando sua antiga posição lógica como livre (*walkable*) e sua posição lógica atual como ocupada (*unwalkable*).

3.8 IA

Esta seção descreve como foi implementada a inteligência artificial no projeto. A inteligência artificial é um sistema que simula o jogador computador, isto é, o jogador controlado pelo computador inimigo do jogador humano.

É importante destacar que o sistema de inteligência artificial aborda somente esta definição descrita anteriormente, pois seria possível considerar Pathfinding como parte da inteligência artificial. Também seria possível considerar como parte desse sistema, a inteligência artificial individual da unidade dinâmica como, por exemplo, ao contra-atacar quando é atacada, porém isso não foi feito para manter a estrutura orientada a objeto.

Portanto, a inteligência artificial da unidade dinâmica está dentro da classe da unidade dinâmica, e o Pathfind também tem uma classe própria. Igualmente, o sistema de inteligência artificial descrito aqui está implementado em uma classe específica. Existe um método chamado pelo loop principal para atualização dessa classe a uma taxa pré-determinada por um temporizador.

O sistema de inteligência artificial pode ser considerado um agente racional. Um agente é qualquer entidade que percebe seu ambiente através de sensores e age sobre o ambiente através de atuadores. O ambiente pode ser físico ou virtual, como é o caso do projeto. A escolha do mapeamento do ambiente que leva o

agente a tomar uma ação, assim como as ações tomadas, definem o grau de sucesso na realização de uma tarefa.

Como será visto posteriormente, o ambiente de jogo é percebido pelo agente através dos dados de jogo. O comportamento do sistema pode ser facilmente aperfeiçoado adicionando ou trocando os dados utilizados, porém este não é o foco do projeto. Do mesmo modo que o projeto não deu foco à parte artística do jogo, não foi dado foco ao aperfeiçoamento do comportamento da inteligência artificial: foi construído um sistema que pode ser alimentado ou facilmente aperfeiçoado para se ter um jogador computador com comportamento mais humano, porém isso envolve um trabalho de design que não faz parte do escopo do projeto, cujo objetivo principal é a engenharia de software.

O agente utilizado no sistema de inteligência artificial é um agente reativo. É um agente que procura ter regras de condição-ação inteligíveis, modulares e eficientes. No conceito de agente reativo não existe uma tabela que mapeia todos os cenários possíveis do ambiente de jogo e para cada cenário tenha uma ação (este tipo de agente existe e é chamado de agente tabela). O agente reativo percebe uma série de condições, que como já foi descrito, são retiradas dos dados de jogo e toma uma ação.

Foi feita uma pesquisa para descobrir como é implementada a inteligência artificial nos jogos, porém pouca informação foi encontrada. Pode-se dizer que geralmente, nos jogos de estratégia em tempo real são utilizados os conceitos de agente reativo e agente tabela. O agente tabela é utilizado para tomar pequenas

ações para aperfeiçoar a inteligência ou para fazer pequenas correções nas ações demandadas pelo agente reativo.

A implementação do agente no projeto foi de concepção exclusiva do grupo de projeto, o grupo não conseguiu encontrar informações explicitas sobre a implementação do agente para o caso do jogo de estratégia em tempo real e, portanto, criou o seu próprio sistema.

Para implementar o agente descrito, foi necessária uma estrutura básica fundamentada em máquinas de estado e num gerenciador de recursos para essas máquinas.

O gerenciador de recursos é responsável por distribuir as unidades dinâmicas do jogador pelas máquinas de estado, e cada uma das três máquinas existentes é responsável por controlar uma das três funções básicas das unidades dinâmicas: atacar o inimigo, defender e trabalhar.

Para distribuir as unidades pelas máquinas de estado, o gerenciador utiliza a idéia da percepção de ambiente do agente reativo: a inteligência artificial tem acesso aos seus dados e aos dados do jogador humano. Pode-se dizer que a inteligência artificial trapaceia no jogo, pois ela tem acesso aos dados do jogador, porém o jogador não tem acesso a todos os dados dela. Contudo essa é uma técnica utilizada por alguns sistemas de inteligência artificial em jogos.

Esses dados são utilizados para calcular o que foi chamado de "necessidade". Existem três tipos de necessidade: a necessidade de ataque, a necessidade de defesa e a necessidade de lucro, que são utilizadas para determinar

se a inteligência artificial deverá alocar uma unidade para atacar, defender ou trabalhar.

Cada uma das unidades dinâmicas deve pertencer a uma e somente uma máquina de estado. Então, a cada chamada para atualização da inteligência artificial, o gerenciador de recursos utiliza os dados do jogo em fórmulas que calculam qual máquina de estado deverá receber uma unidade dinâmica. Por exemplo, no projeto foi definido que a inteligência artificial deverá atacar quando detectar que o jogador está enfraquecido. Portanto, a inteligência artificial utiliza dados como a quantidade de unidades e dinheiro que o jogador possui para determinar o que chamamos de "necessidade de ataque".

No início do jogo, todas as unidades dinâmicas do tipo soldados são colocadas na máquina de estados de defesa e todas do tipo distribuidor são colocadas na máquina de trabalho. Após calcular cada uma das necessidades, a inteligência artificial verifica o maior valor dentre os resultados das necessidades e toma uma ação. A ação a ser tomada depende da necessidade "maior", isto é, a inteligência artificial tomará ações específicas para cada necessidade:

- Necessidade de Ataque: caso a inteligência artificial determine que fazer um ataque é mais prioritário, o algoritmo tentará retirar uma unidade da máquina de estados de defesa e alocar na máquina de estados de ataque. Caso não exista nenhuma unidade na máquina de estados de defesa para ser retirada, o algoritmo tentará fazer uma contratação (criação) de unidade e tentará alocá-la na máquina de

estados de ataque. Caso não consiga, por não ter dinheiro, o algoritmo não executa nenhuma ação.

- Necessidade de Defesa: similarmente à necessidade de ataque, a inteligência artificial tentará tirar uma unidade da máquina de estados de ataque para alocar na máquina de estados de defesa, caso não consiga tentará fazer uma contratação e caso ainda não tenha sucesso não executará nenhuma ação.
- Necessidade de Lucro: ao determinar que deve ganhar mais dinheiro, a inteligência artificial tentará contratar uma unidade dinâmica distribuidora para alocá-la na máquina de estados de trabalho.

O algoritmo de inteligência artificial pode ser customizado em várias partes para atender requisitos do designer. No gerenciador de recursos, a fórmula da necessidade tem o seguinte formato:

$$\Sigma(V_i \cdot m_i) \cdot K + C$$

Os valores V são os dados utilizados na fórmula (dinheiro, número de soldados, etc.) transformados para unidade do dinheiro. Todos esses dados podem ser transformados para essa unidade, pois têm um preço no jogo, e conseqüentemente pode-se utilizar da somatória como a união de todos os fatores que influenciam na decisão. As variáveis m são modificadores de V, já que um dado V pode ter um impacto negativo na decisão e, portanto, deve ser um valor negativo.

A variável m também pode ser diferente de 1 ou -1 para acentuar a importância de seu V na fórmula. K é uma constante multiplicadora da somatória que tem o propósito de diminuir ou aumentar o intervalo de resultados da fórmula em relação aos resultados das outras fórmulas. O propósito da constante C é semelhante, ela é somada para deslocar o intervalo de resultados da fórmula em relação aos outros resultados, deixando-o mais negativo ou positivo.

Para customizar as fórmulas deve-se simplesmente alterar os valores das constantes e utilizar os dados que julgar necessário.

Ainda no gerenciador de recursos, as ações também poderiam ser programadas de forma diferente e evoluídas para se ter uma inteligência artificial mais poderosa. Por exemplo, poderia ser utilizado o conceito de níveis de necessidade para se tomar uma ação. Assim, se a necessidade de ataque resultasse num certo valor o algoritmo tomaria uma ação, caso resultasse em um outro valor mais alto, tomaria outra ação, um ataque mais forte. Uma alternativa mais simples seria inverter a ordem das ações que o algoritmo executa atualmente, descritas acima. Assim, o computador preferiria gastar seu dinheiro ao invés de desalocar a sua defesa ao fazer um ataque. Modificar as fórmulas utilizadas no gerenciador de recursos permite a criação de jogadores com personalidades diferentes: ofensivas, defensivas, construtivas, etc.

Após a atualização do gerenciador de recursos, a inteligência artificial deverá atualizar as máquinas de estado. Tecnicamente, a máquina de trabalho é um container de unidades dinâmicas do tipo distribuidor, e as máquinas de ataque e defesa são containeres de containeres de unidades dinâmicas do tipo soldado, isto

é, uma máquina de ataque ou defesa é um container que armazena pequenos grupos de soldados.

Na máquina de estado de trabalho, qualquer unidade distribuidora alocada passa para o estado de "posicionando". De acordo com o game design, a inteligência artificial deveria mandar a unidade se posicionar num lugar estratégico para o trabalho, porém o implementado foi que a unidade deve-se posicionar-se onde está, isto é, ela passa diretamente para o próximo estado da máquina, que é o estado "trabalhando". Neste estado a unidade passará a gerar dinheiro para o jogador computador.

Na máquina de estados de ataque, uma unidade soldado alocada é inserida num grupo, que tem tamanho máximo pré-determinado, e recebe uma ordem para posicionar-se ao lado da primeira unidade do grupo – se ela não for a primeira unidade do grupo. Ao atingir o tamanho máximo é criado um novo grupo e assim por diante. O objetivo de posicionar as unidades ao lado uma das outras é criar pequenos grupos, grupos de ataque. Enquanto as unidades estão sendo posicionadas e o grupo não estiver completo o grupo está no estado "posicionando".

A partir daí o grupo passa para o estado "atacando" e cada unidade do grupo recebe a ordem para atacar a unidade mais próxima do grupo. Exterminado o alvo, o grupo segue para outro ataque à unidade mais próxima e assim por diante. O grupo nunca volta ao estado "posicionando" e ao ir para o estado "atacando" não é adicionado mais unidades a ele, mesmo que alguma morra.

O conceito de container de grupos também é utilizado na máquina de defesa. Porém, nesta máquina, um grupo no estado "posicionando" tem suas

unidades posicionadas ao lado de uma unidade distribuidora, para protegê-la. Outro grupo deve ir para outra unidade distribuidora, e assim por diante. Quando todas as unidades do grupo estiverem ao lado da unidade distribuidora, o grupo passa para o estado "aguardando", onde é utilizado um temporizador para determinar quando o grupo deve voltar para o estado "posicionando". O objetivo dessa volta é atualizar a posição do grupo para acompanhar uma eventual mudança de posição do distribuidor.

A inteligência artificial também pode ser altamente customizada quanto as máquinas de estado. Como exemplo, pode-se mudar a estratégia de ataque, procurando atacar unidades distribuidoras (que por definição são desarmadas) que estejam longe de unidades soldado ou mudar o tamanho máximo do grupo de ataque para adequar-se ao tamanho de um grupo que será atacado numa proporção favorável ao jogador computador. Os grupos de defesa poderiam ser mais ativos e tentar receptar um ataque vindo do inimigo antes dele chegar nas unidades distribuidoras, detectando o movimento de grupos.

Quanto à implementação do código de inteligência artificial, a manipulação das unidades entre as máquinas foi o maior desafio. Tivemos que implementar containeres e controles para transferência de unidades de uma máquina para outra além de controles com outras funções, como a retirada de unidades mortas das máquinas.

As customizações citadas são simples em suas definições e poderiam deixar a inteligência artificial bem mais real, porém custosas – em termos de tempo – para serem implementadas, principalmente na aquisição de dados que não são

atributos de classes, como a determinação de grupos de unidades através das distâncias entre as unidades e a detecção da movimentação de grupos de unidades.

3.9 Colisão

No projeto, o ambiente tridimensional que constitui o jogo é composto por um terreno levemente ondulado, sobre o qual se encontram algumas construções, bem como personagens móveis controlados pelo jogador (unidades do jogador) e pelo computador (unidades do computador e NPC's).

Para o ambiente mencionado acima, podemos dividir as colisões em três tipos básicos: colisão entre os personagens móveis e o terreno, colisões entre personagens móveis, e colisão entre os personagens móveis e as construções.

Cada um dos tipos recebeu um tratamento apropriado, sendo todos descritos logo abaixo:

3.9.1 Colisão entre os personagens móveis e o terreno

O terreno do jogo apresenta ondulações que, mesmo apresentando inclinações relativamente suaves, mostram-se como obstáculos para um personagem que deseja movimentar-se de um ponto para outro do terreno. Assim, torna-se necessário um tratamento para impedir que o personagem, em sua movimentação, transpasse o terreno.

A forma escolhida para realizar este tratamento é descrita logo abaixo.

À cada iteração do loop principal, são identificados os personagens que estão em movimento. Feita essa identificação, lança-se mão do recurso de raios presente no Ogre: para cada personagem em movimento, lança-se um raio vertical de cima para baixo, de modo que o mesmo siga o eixo principal do personagem e chegue até a superfície do terreno. Por meio deste procedimento, descobre-se a altura do terreno onde o personagem deveria estar. Feito isso, desloca-se o personagem até a altura encontrada.

Por meio desta técnica, o personagem em movimento sempre se encontrará sobre a superfície do terreno.

3.9.2 Colisões entre unidades

Quando dois personagens em movimento estão muito próximos, é preciso um tratamento que consiga impedir a interpenetração de ambos.

No projeto, aproveitou-se a própria técnica de *pathfinding* para realizar este tratamento. A técnica de *pathfinding* será descrita na seção 2.3.

De maneira semelhante à colisão entre personagens, foi aproveitada a própria técnica de *pathfinding* para tratar deste tipo de colisão.

4 Resultados

Este capítulo apresenta os resultados obtidos pelo projeto, de acordo com alguns testes realizados. As discussões mais aprofundadas sobre tais resultados são abordadas no capítulo 5 de conclusões.

A parte de Overlays, destinada à geração de menus do jogo, no que toca a questão de desempenho, mostrou-se pouco influente com relação à performance gráfica. Sua inclusão no projeto não afetou significativamente a taxa de renderização (frames/segundo).

De acordo com o que foi documentado nos casos de uso referentes ao overlay, foram implementadas as funcionalidades contempladas nos casos de uso "Salvar Jogo" e "Fechar Jogo", sendo que aqueles que tratavam de compra de arma e produtos e "Novo Jogo" foram retirados do escopo por restrições de tempo e por serem menos prioritários. Além disso, foram incluídas algumas funcionalidades adicionais ao menu principal, como os botões de ataque e movimentação, pelo fato de agregarem valor à jogabilidade.

A inteligência artificial apresentou os resultados esperados: as unidades são manipuladas de acordo com os dados que representam o ambiente em que elas se encontram. Porém, as ações tomadas pelo jogador computador são grosseiras, pois as equações que refletem a percepção do ambiente não estão calibradas

adequadamente. Pode-se notar uma grande mudança no comportamento do jogador computador ao alterar alguns fatores de uma dada fórmula.

Quanto aos requisitos de desempenho do jogo, foram realizados testes para a verificação da taxa de renderização e da contagem de triângulos sob certas circunstâncias, descritas logo abaixo.

Cenário 1	
Processador	AMD Sempron (TM) 2400+ 1.67 GHz
Memória	512MB de RAM
Placa de Vídeo	GeForce - 128MB

Tabela II - Cenário 1 de teste

O jogo foi compilado no Visual Studio .NET 2003 em modo *debug*, com todos os módulos funcionando juntos (Inteligência Artificial, Pathfinding, Picking e Overlay) e foram dispostas 12 unidades dinâmicas ao longo do cenário. Para o cenário 1 de teste, com a câmera inicialmente centrada em 5 unidades, a média de FPS apresentou um valor de 45, e a quantidade de triângulos renderizados assumiu valores por volta de 5.020.

Para finalizar o teste no cenário 1, foram dispostas as 12 unidades de modo a se tornarem todas visíveis. Neste caso, teve-se uma média de 18 FPS, e uma quantidade de triângulos renderizados na faixa de 6.900.

Cenário 2	
Processador	Athlon 2.5 GHz
Memória	512 MB de RAM
Placa de Vídeo	Radeon 9600 Pro - 128MB

Tabela III - Cenário 2 de teste

Ao realizar-se o mesmo teste, agora para o cenário 2, com 5 unidades na tela, o resultado foi uma média de FPS de 77 para uma quantidade de triângulos renderizados por volta de 5.000.

No segundo teste realizado para este cenário, foram dispostas 12 unidades visíveis pela câmera, chegando a uma quantidade de triângulos de cerca de 7.000. Neste contexto, a taxa de FPS alcançada teve uma média de 47.

Pelos resultados observados acima, vê-se que o último teste no cenário 1 aponta uma taxa de quadros por segundo situada abaixo da média estipulada como requisito de desempenho. Este efeito é contornado de duas formas: ao se mudar o modo de compilação do Visual Studio para *release*, e/ou reduzindo a quantidade de triângulos que constituem as unidades ao se utilizar modelos tridimensionais mais simplificados. A primeira forma se explica pelo fato de a compilação em modo *debug* utilizar informações de depuração, sem otimização. A otimização não é feita neste modo por tornar a depuração do código mais complicada, uma vez que torna mais complexa a relação entre o código fonte e as instruções geradas.

Já no modo *release*, o código intermediário se torna menor e mais rápido de ser transformado devido a algumas otimizações que o compilador faz.

Um exemplo das melhorias de desempenho na taxa de renderização ao se modificar para o modo *release*, é que em testes realizados pelos autores deste projeto, para uma mesma versão de código mais antiga, a compilação em modo *release* tornou a taxa de quadros por segundo cerca de dez vezes superior à que foi obtida com o mesmo código em modo *debug*. Para a versão atual de

implementação do projeto, não possível realizar os testes em modo *release* por restrições de tempo.

Ao comparar-se os resultados entre os dois cenários, pode-se observar uma significativa diferença de desempenho em renderização no que se refere a taxa de FPS. Isso leva a concluir que para a compilação em modo *debug*, a questão do processamento e a placa de vídeo do computador têm considerável influência no requisito de desempenho do jogo, pois o computador do cenário 2 apresenta processamento razoavelmente superior ao do cenário 1.

Com relação aos casos de uso, são listados abaixo os casos de uso que não foram implementados até a última versão produzida ao final da última iteração da fase de construção.

1.4 - Entrada de unidades personagens em unidades construções

2.1 - Seleção de NOvo jogo no menu principal

2.5 - Compra de produtos

2.6 - Compra de arma

2.7 - Compra de construção

3.1 - Carregamento de unidade

3.2 - Descarregamento de unidade

3.5 - Mudança de função de uma Unidade Personagem

3.6 - Armamento com arma de uma Unidade Personagem

3.7 - Fornecimento de cobertura feita por uma Unidade Personagem

3.8 - Patrulha

3.9 - Carregamento da UPD com produto fornecido pela UPG

4.1 - Carregamento da unidade do tipo personagem com produto

4.2 - Ataque de uma unidade dinâmica do jogador em modo ofensivo a uma unidade dinâmica inimiga

4.5 - Ganho de habilidade de uma unidade dinâmica

4.7 - Venda de produto para cliente em veiculo

4.8 - Fuga de NPC de um tiroteio

4.9 - Inserção de morador

4.10 - Transferência de posse de construção

Um ponto importante a ser considerado ao se analisar os casos de uso acima, é que existe uma forte interdependência entre certos casos de uso. Assim, um caso de uso que não seja implementado em sua plenitude compromete a conclusão dos casos de uso que dele dependem. Um exemplo deste fato se dá com as funcionalidades relacionadas com produtos, que comprometem os casos de uso 3.1, 3.2, 3.9, 4.1 e 4.7.

Tais caso de uso não foram implementados devido às limitações de tempo encontradas durante a execução do projeto. Contudo, como o projeto caracterizou-se por ser constituido por módulos bem-definidos e facilmente expansíveis, em novas iterações os casos de uso listados acima certamente poderiam ser incorporados ao projeto.

As funcionalidades relacionadas com produtos, conforme explicado anteriormente, não foram totalmente implementadas, ficando-se restritas a venda de produtos. Neste caso, a venda de produtos foi simulada por meio de incrementos no valor de dinheiro na posse do jogador, quando uma ou mais unidades do jogador estiverem trabalhando.

Da mesma maneira, todas as outras funcionalidades que não foram totalmente implementadas, tiveram uma solução de escopo reduzido, mas que ainda sim permitiram a jogabilidade minimamente desejada.

5 Conclusões

O capitulo que se segue descreve as principais conclusões obtidas após a realização do projeto. Serão discutidas as vantagens e desvantagens da metodologia empregada, das tecnologias e ferramentas empregadas.

O Processo Unificado mostrou-se uma metodologia bastante adequada para o desenvolvimento de jogos de estratégia. Como toda metodologia, impõe aos desenvolvedores a necessidade de seguirem um planejamento, orientando as ações de modo a se ter o mínimo de retrabalhos. Além disso, o planejamento contempla procedimentos cujo objetivo é o máximo reaproveitamento de código, como documentação e encapsulamento de elementos em comum.

Para a realização de documentação, o Processo Unificado utiliza a UML, linguagem cuja semântica bem-definida permite que a comunicação entre os envolvidos num projeto aconteça de forma não-ambígua.

Outro ponto positivo do Processo Unificado é a sua adaptabilidade. Com a existência de uma enorme variação no grau de complexidade entre os diversos *games* atuais, uma metodologia que se ajuste às condições de cada projeto permite aos seus desenvolvedores a redução de desperdícios com a eliminação de atividades desnecessárias.

Além das vantagens descritas acima, a utilização da metodologia permite que, em empresas com equipes razoavelmente constantes, o custo e tempo de

treinamento sejam gastos apenas uma vez, mas que suas vantagens sejam aproveitadas por vários projetos no futuro.

Embora a adoção do Processo Unificado no desenvolvimento de um jogo RTS forneça as vantagens apresentadas acima, tem-se, como principal desvantagem encontrada, o tempo despendido para o estudo e assimilação da metodologia. Porém, tal tempo investido nas etapas iniciais do projeto foi recuperado nas fases seguintes pois, conforme dito anteriormente, a metodologia permitiu que se reduzisse os retrabalhos e orientou os passos dos autores na realização do projeto.

Quanto às tecnologias utilizadas, a *engine* gráfica OGRE mostrou uma elevada relação custo-benefício. Sendo uma ferramenta *open-source*, não acarreta aos seus usuários nenhum custo financeiro. Sua ampla documentação (incluindo tutoriais e códigos-exemplo) e a ativa participação da comunidade de usuários permitem a redução de tempo na resolução de problemas de implementação. A engine OGRE permite a criação de gráficos de altíssima qualidade com uma razoável simplicidade na codificação.

Embora a *engine* OGRE disponha aos seus usuários os benefícios citados acima, fornece apenas uma solução gráfica para realização de um jogo. Para a implementação de outros recursos, como som, rede, inteligência artificial, colisão e física, é necessária a integração da OGRE com outras funcionalidades codificadas e, dependendo do caso, com outras bibliotecas.

Um exemplo dessa necessidade de integração foi a resolução do problema de pathfinding. Após pesquisarmos as opções disponíveis, encontramos a engine

OpenSteer, que segundo os desenvolvedores da OGRE é de fácil integração e possui diversos recursos para pathfinding e colisão. Apesar disso, tivemos grande dificuldade em utilizar tal engine e fomos obrigados a desenvolver nosso próprio mecanismo de pathfinding e colisão que pudéssemos integrar à OGRE.

Quanto às técnicas de geração de menus do jogo, a utilização do padrão XML para codificação do layout dos menus apresentou bons resultados quanto à flexibilidade, à medida que possibilitou implementar diversos tipos de layout de maneira modular, além de permitir sua rápida alteração sem a necessidade de re-compilar todo o jogo.

Sobre o algoritmo de busca de caminho realizado para a movimentação das unidades móveis, a utilização do algoritmo A* mostrou-se uma alternativa viável e eficaz, visto que esse algoritmo obtém o menor caminho com uma expansão pequena de nós do grafo, o que representa uma redução do tempo necessário para o processamento do algoritmo. Outra vantagem que encontramos é o fato de tal algoritmo ser de fácil implementação, o que é muito importante, uma vez que não era parte de nosso trabalho fazer uma pesquisa extensa sobre tais algoritmos, dado o escopo escolhido para o projeto.

Uma desvantagem que encontramos no A* é o fato de que, dado que o ambiente era dinâmico e as unidades móveis poderiam estar em qualquer lugar do mapa a qualquer instante, o caminho completo deveria ser re-calculado sempre que houvesse ameaça de colisão entre duas unidades moveis e só então elas poderiam prosseguir em seu caminho. Uma solução possível para esse problema seria a utilização do algoritmo D* que é uma evolução do A*, permitindo a reavaliação dos

custos em tempo real, quando o terreno for mutável (por exemplo a movimentação de unidades). Devido à sua complexidade, decidimos pela utilização do A*

Uma outra desvantagem do A* é o fato de que tal algoritmo pressupõe o calculo do caminho completo da origem para o destino o que, no caso de um jogo de estratégia nem sempre é possível, mas algumas vezes o usuário gostaria que fosse alcançado um resultado próximo do ideal, caso este não fosse possível.

Com relação à IA do jogo, como foi explicado no capítulo Materiais e Métodos, o comportamento dos agentes pode ser aprimorado sem grandes dificuldades. As fórmulas de percepção do ambiente poderiam ser calibradas para uma melhor tomada de decisão, porém foi decidido que isto não deveria ser feito por ser um trabalho que não acrescentaria tanto valor ao projeto, considerando o seu objetivo.

A utilização de Máquinas de Estados Finitos para a implementação da inteligência artificial e também do comportamento das unidades dinâmicas permitiu uma fácil previsão das ações das unidades. A vantagem é que essa previsibilidade de ações não deixa o jogo em si mais previsível, já que ela é mascarada pelo gerenciador de decisões do módulo de inteligência artificial. A utilização de maquinas de estados finitos encadeadas para a tomada de decisões aumenta a ilusão de "não previsibilidade" por parte do jogador, o que é muito desejável, pois os jogadores em geral apreciam um jogo que possua uma inteligência artificial desafiadora. A previsibilidade gerada pelas matuinas de estados finitos apenas facilita a implementação, já que dada uma ordem de ação a uma unidade ou grupo de unidades, é fácil de verificar se a ação é executada corretamente.

Uma desvantagem da utilização de Máquinas de Estados Finitos em inteligência artificial é a impossibilidade de aprendizado por parte do agente, isto é, o computador sempre responderá da mesma maneira a uma dada estratégia do jogador humano. Os jogos de estratégia atuais também funcionam dessa maneira, portanto se fosse decidido fazer um módulo de inteligência artificial com aprendizado, como um módulo utilizando redes neurais, o grupo faria algo distinto do comumente utilizado, o que representa um risco para o projeto devido a necessidade de grande quantidade de tempo para estudo de um agente mais.

Referências [1]

ABRAGAMES (Org.), **Plano Diretor da Promoção da Indústria de Desenvolvimento de Jogos Eletrônicos no Brasil – Diretrizes Básicas**. Disponível em <www.abragames.org>. Acesso em: 05 dez. 2005.

BERNARDES JR., J. L. - "**Desenvolvimento de um ambiente para visualização tridimensional da dinâmica de risers**" - 2004. 202 f. Dissertação (Mestrado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2004.

CRAWFORD, CHRIS - "**The Art of Computer Game Design**" - 1982. Mcgraw-Hill Osborne Media.

FOLEY, J. et al. **Computer Graphics**: Principles and Practice. 2nd. ed. Addison-Wesley, 1996. 1175 p.

GAMMA, E. **Design Patterns**: elements of reusable object-oriented software. Addison- Wesley, 1995. 395 p.

IBM. **Best Practices for Sofware Development Teams,** Rational Software, 2003. (White Paper).

JACOBSON, I.; Booch, G. & Rumbaugh, J. **The Unified Software Development Process**. Addison-Wesley, 1999. 463 p.

PATEL, A.J (Comp.), **Amit's Thoughts on Path-Finding and A-Star**. Disponível em <http://theory.stanford.edu/~amitp/GameProgramming>. Acesso em: 05 dezt. 2005.

[1] De acordo com: ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. NBR 6023: informação e documentação: referências: elaboração. Rio de Janeiro, 2002.

REESE, B. et al., **Finding a Pathfinder**. In: Spring Symposium on Artificial Intelligence and Computer Games, 1999.

ROUSE III, R. **Game Design: Theory & Practice**.
Walsh, P. Advanced 3D Game Programming Using DirectX 9.0. Texas: Worldware Publishing, Inc, 2003. 706 p.

WIKIPEDIA(Comp.),**Real-time strategy**. Disponível em <http://en.wikipedia.org/wiki/Real-time_strategy>. Acesso em: 05 dez. 2005.

WIKIPEDIA (Comp.), **Strategy game**. Disponível em <http://en.wikipedia.org/wiki/Strategy_game> . Acesso em: 05 dez. 2005.

WOODCOCK, S. **Game AI: The State of the Industry**. Nov. 1998. Disponível em <http://www.gamasutra.com/features/19981120/gameai_01.htm>. Acesso em: 05 dez. 2005.

Apêndice A – Documento de descrição de requerimentos

Este documento descreve um jogo de estratégia em tempo real (RTS - Real Time Strategy), que procura ter funcionalidades diferentes das encontradas nos jogos de estratégia atuais.

O objetivo do design diferente é proporcionar uma experiência em criação de jogos, o design do jogo é a primeira e uma das mais importantes fases da criação, portanto decidimos criar um design de jogo de estratégia.

O jogo não tem um tema definido, o design foi concebido pensando em funcionalidades de jogos de estratégia existentes e modificando-as. Não é o objetivo fazer um trabalho artístico, por isso nosso jogo não apresenta um tema, porém não poderíamos pular essa fase tão importante que é o design, já que nosso projeto de formatura consiste em estudar uma metodologia para criação de jogos.

Apesar do jogo não possuir um tema, o design foi feito de modo a permitir que um tema seja encaixado no jogo. Nós pretendemos encaixar um tema durante a fase de implementação, essa não é a melhor prática, já que o tema influencia muito no design do jogo, porém já discutimos sobre o assunto e encaixar o tema neste jogo será fácil.

A diferença fundamental do jogo para os outros jogos de estratégia é que os outros jogos dão ênfase às batalhas, isto é, a capacidade de destruição das unidades, enquanto o nosso jogo dá ênfase aos atributos humanos das unidades,

onde para vencer o jogo, o jogador deverá primeiramente conquistar o respeito de suas unidades antes de conquistar os inimigos.

Devemos lembrar também que o documento de design não é um documento para leigos em jogos, ele é escrito para as pessoas que vão implementar o jogo e portanto não podemos garantir que uma pessoa que não conhece jogos de estratégia entenderá este documento.

Este documento não é final e neste projeto de formatura não escreveremos outras versões. Exercitamos a criação de design e possuímos aqui um conteúdo maior do que implementaremos, e esse é o motivo pelo qual não continuaremos com a escrita de novas versões. Este documento, por não ser final, é um subconjunto do documento que descreveria totalmente o jogo, mas como um dos objetivos do projeto é a construção da demo do jogo, e não o jogo completo, uma parte deste documento já é suficiente. As partes deste documento que serão implementadas estão descritas nos casos de uso.

1 Introdução

Como foi dito anteriormente, este documento descreve um jogo de estratégia em tempo real, sem suporte a rede, que procura ter funcionalidades diferentes das encontradas nos jogos de estratégia atuais.

O jogo possui um mapa onde deverão aparecer construções e unidades (i.e, pessoas, veículos, animais, etc.). O espaçamento entre as construções deve ser suficiente grande para possibilitar maior jogabilidade, permitindo que o jogador movimente suas unidades com maior liberdade.

O jogador deverá, resumidamente, obter dinheiro através da venda de produtos previamente comprados para poder dominar todo território e destruir seus inimigos, comprando armas e controlando um pequeno exército (de 5 a 20 unidades). Para isso deverá comprar construções e contratar unidades. O objetivo principal é ter o controle de território, o jogador poderá medir o seu poder através do tamanho do seu território, a dificuldade do jogo é conseguir manter o território ocupado.

Esse ciclo compra de produtos -> venda de produtos -> compra de armas -> conquista de território provoca uma sensação de poder e acúmulo de riqueza, que é a sensação que queremos provocar aos jogadores.

Algumas das novas funcionalidades provêm dessas divisões do ciclo: para cada função exercida há uma "profissão" associada à unidade controlada pelo jogador, e essa profissão poderá mudar quando o jogador quiser (Ex.: dê uma arma

a um negociador e ele será um soldado), porém o jogador deverá sofrer as conseqüências dessas mudanças (i.e., nem todo negociador está apto a virar soldado). Apesar disso, essa mudança deverá ocorrer sempre, já que não se deve contratar "desconhecidos" para exercer funções importantes.

Essas e outras funcionalidades foram modeladas para tornar o jogo diferente dos demais e assim cumprir os nossos objetivos e para isso foram criados novos atributos associados ao jogador e suas unidades, como por exemplo, Moral, Fidelidade e Conhecimento.

Esses atributos criam um caráter de unicidade entre as unidades, onde cada unidade é conhecida do jogador, umas mais que as outras, mas não se baseando nas habilidades com armas ou nos atributos físicos da unidade, e sim no relacionamento entre o jogador e a unidade, como na vida real. O pensamento que gostaríamos de provocar não é o "vou enviar essa unidade à missão pois ela é melhor em tal arma" e sim "vou enviar essa unidade à missão pois ela é mais confiável".

A evolução do jogo se dá pela evolução dos produtos e evolução das armas.

Todos as variáveis envolvidas em cálculos nesse jogo são dadas como sugestão e deverão ser modificadas para equilibrar o jogo, durante a fase de testes.

O jogo não é divido em fases. O problema de jogos extensos sem fases é que se pode perder o controle da dificuldade, fazendo com que o jogador fique jogando um jogo que ele na verdade já perdeu. Esse problema é resolvido adicionando objetivos que garantem o sucesso no jogo caso sejam cumpridos, mas

não são essenciais, isto é, o não cumprimento não implica o insucesso, deixando de comprometer a liberdade.

Esses objetivos não serão descritos nesta versão do documento.

2 Mecânica do Jogo

A mecânica do jogo será descrita baseando-se no ciclo compra de produtos -> venda de produtos -> compra de armas -> conquista de território apresentado na introdução, além de outros tópicos.

Segue uma lista dos assuntos encontrados em cada tópico. Essa lista é para uma simples organização do documento e consulta, podendo ser ignorada na primeira leitura.

Estrutura do Jogo:

- Área de Atuação;
- Círculo de Atuação;
- Revelando a Área de Atuação do oponente;
- Introdução aos atributos Fidelidade e Fama;
- Movimentos automáticos.

Unidades e Construções:

- Reabastecimento automático;

- Associação de função de unidades;
- Descrições de cada unidade;
- Compra/definição de construções;
- Descrições de cada construção.

Atributos:

- Vida;
- Fidelidade;
- Testes feitos com Fidelidade;
- Conhecimento;
- Fama;
- Exibição de nomes das unidades;
- Relevância;
- Intensidade das cores da Área de Atuação;
- Moral;
- Tabelas de Aumento/Decréscimo na Moral.

Compra de Produtos:

- Como comprar e entregar o produto em um Estoque;
- Relação segurança/economia.

Venda de Produtos:

- Associação Gerente/Posto;
- Usando Gerente como Soldado;
- Associação Gerente/Distribuidor;
- Reabastecimento do Posto;
- Associação Distribuidor/Ponto;

Compra de Armas:

- Comparação com Compra de Produtos;
- Venda de armas;
- Perda de arma;
- Associação de arma a uma unidade;
- Troca de armas;
- Armas de uso único.

Conquista de território:

- Batalhas;
- Armas ficam escondidas;
- Unidades que fogem;
- Tortura;
- Informações dadas em tortura;
- Conquista de construções.

2.1 Área de Atuação

O jogador começa com uma pequena casa no em algum lugar do território, com uma grande distância de seus oponentes.

Essa casa inicial do jogador deve ser sempre escondida de seus oponentes, e a sua tomada representa a morte do jogador. Outros tipos de construção serão descritos nos tópicos seguintes.

Esta casa e a área ao redor dela é a Área de Atuação inicial do jogador, e essa área inicial deve ser representada por um círculo colorido que cobre a tela, chamado de Círculo de Atuação. A Área de Atuação é dinâmica, mudando ao conquistar territórios e de acordo com outros fatores. Deve ter vários graus de intensidade da cor, dependendo do grau de importância do local para o jogador. Os oponentes também devem ter Áreas de Atuação com cores diferentes da do jogador.

Toda unidade ou construção possui um Círculo de Atuação representando que a área ocupada por aquela unidade ou construção possui uma relação com o jogador (ou oponente), de certa forma ele domina a área e tem um poder de atuação sobre ela. O Círculo de Atuação tem raio determinado e estático, pré-definido para cada unidade ou construção.

O mapa deve ser totalmente visível ao jogador, diferentemente do que é normalmente encontrado nos jogos de estratégia atuais, porém, a Área de Atuação do oponente será invisível ao jogador e este terá descobri-la.

Há duas maneiras de se descobrir a Área de Atuação do oponente, na primeira, o jogador move as suas unidades pelo mapa, e a Área de Atuação vai se revelando conforme a unidade passa por ela, como a "escuridão" encontrada nos jogos de estratégia, e conforme dois atributo das unidades e construções chamados Fidelidade e Fama. Esses atributos serão explicados posteriormente, porém pode-se adiantar que a Fidelidade de uma unidade representa, nesse caso, se ela contará ao jogador o que viu ao passar por um lugar, o quanto ela se arriscaria pelo jogador ao ver um inimigo e reagir como ordenado, e a Fama representa o quanto a unidade ou construção do oponente é conhecida pelo jogador, isto é, se a unidade do jogador não reconhecer o inimigo não saberá que aquela área é uma Área de Atuação do oponente.

A outra maneira de descobrir a Área de Atuação do oponente é através das unidades do oponente. Tendo posse uma unidade do oponente o jogador terá a opção de obter informações do oponente. A descrição de como tomar posse de uma unidade do oponente será feita no tópico Atributos, porém para acessar essas informações, o jogador deve selecionar a unidade tomada e clicar em um botão do seu menu. Isso significa que o jogador tentará obter informações através de tortura e a unidade pode fornecê-las ou não, dependendo do atributo Fidelidade que a unidade tem pelo oponente e outros fatores.

Essas descrições de como descobrir a Área de Atuação funcionam para os dois lados, isto é, os oponentes controlados pelo computador agem da mesma maneira em relação à Área do Jogador.

2.2 Movimento Automático

O jogo possui uma funcionalidade de movimento automático de unidades em certas situações, como por exemplo, quando uma unidade distribuidora de produtos precisa se reabastecer, essa unidade faz o caminho automaticamente, sem nenhuma ação do jogador.

Porém, os movimentos automáticos gerados pelo computador poderão ser interrompidos pelo jogador, e completados após a finalização do comando dado pelo jogador. Isto é, se uma unidade está percorrendo um caminho de A a C e o jogador seleciona a unidade e a manda para o ponto B, a unidade deverá interromper o seu caminho, passar pelo ponto B e continuar pelo ponto C. Para interromper a unidade deve-se clicar em um botão do menu da unidade.

Essa funcionalidade permite que uma unidade faça não o menor caminho, gerado pelo computador, mas sim o caminho mais seguro, indicado pelo jogador.

2.3 Unidades e Construções

Neste tópico são apresentadas as unidades controladas pelo jogador e computador e as construções que estes podem possuir.

Detalhes de como são utilizados serão descritos em outros tópicos.

Este jogo apresenta a funcionalidade de reabastecimento automático de produtos que utiliza a funcionalidade de movimentação automática descrita anteriormente. Detalhes de quando o reabastecimento automático acontece serão descritos nos tópicos seguintes.

Neste jogo, as unidades não são criadas, e sim contratadas e associadas a uma profissão. Para contratar uma nova unidade o jogador deverá acessar uma tela onde poderá contratar uma unidade baseando-se nos atributos daquela unidade. Para associar uma profissão - ou função - à unidade o jogador deverá clicar na unidade e escolher a função para essa unidade em um menu da tela. Ao selecionar uma função que requer armamento, caso a unidade ainda não esteja armada, o jogo deverá exibir um menu com as armas disponíveis ao jogador. Para maiores detalhes sobre armamento de unidades, vide o tópico Compra de Armas.

Algumas unidades possuem a capacidade de carregar produtos. A quantidade de produtos é determinada pela função exercida pela unidade.

2.3.1 Unidade do tipo Personagem Soldado (UPS ou Soldado)

É a unidade armada principal. Serve para proteger o território e invadir novos territórios.

2.3.2 Unidade do tipo Personagem Gerente (UPG ou Gerente)

É a unidade que constitui as negociações dentro de um Posto (as construções serão apresentadas no próximo tópico). Pode ser armada, porém não é o seu foco o combate.

2.3.3 Unidade do tipo Personagem Distribuidor (UPD ou Distribuidor)

É a unidade não armada que faz a venda fora do Posto. Utiliza um Posto para se reabastecer de produtos e volta ao seu Ponto para vendê-las.

2.3.4 Unidade do tipo Personagem Cliente (UPC ou Cliente)

É a unidade não armada e controlada pelo computador que compra produtos do Gerente e principalmente dos Distribuidores.

2.3.5 Unidade do tipo Personagem Morador (UPM ou Morador)

É a unidade não armada que serve para, simplesmente ficar navegando pelo mapa, como as pessoas em uma rua. Pode sofrer ataques.

2.3.6 Unidade do tipo Personagem Saqueador (UPS ou Saqueador)

Unidade armada controlada pelo computador que realiza combate com o jogador e com oponentes do jogador.

2.3.7 Unidade do tipo Veículo (UV ou Veículo)

Os Veículos são utilizados para realizar um conjunto de unidades e transportá-las de modo mais rápido. Os Veículos podem gerar dano por atropelamento ou podem assumir a capacidade de gerar dano das unidades armadas que estão dentro do Veículo. Os Veículos também podem armazenar cargas de produtos e/ou armas, servindo como um Estoque. Apesar disso as unidades não irão automaticamente ao Veículo para se reabastecer ou pegar armas. A idéia é que os Veículos sejam utilizados para o transporte, mas não queremos tirar a liberdade do jogador para criar novas soluções para o seu jogo, isto é, não queremos que o jogador utilize o Veículo como Estoque mas se o jogador quiser ele pode usar.

Quando um Veículo chega a um nível de dano ele pára de funcionar. Quando um Veículo toma todo o dano ele explode, matando todos que estão dentro e destruindo toda carga que carrega. Além disso, ele gera um dano radial aos que estão perto.

As construções, diferentemente de outros jogos de estratégia, não são construídas, e sim compradas ou estabelecidas, dependendo do tipo de construção.

Para comprar uma construção deve-se acessar uma construção já existente no mapa e comprá-la. A função que essa construção terá será determinada pelas unidades que ocuparão o local.

2.3.8 Unidade do tipo Construção Sede (UCS ou Sede)

Como explicado anteriormente, é a base do jogador, a Área de Atuação do jogador (e oponentes) é inicialmente definido pelo local dessa Sede. A tomada da Sede representa a morte do jogador e no caso do oponente a morte deste. Maiores detalhes vide o tópico Conquista de Território.

2.3.9 Unidade do tipo Construção Estoque (UCE ou Estoque)

O Estoque é o lugar onde os produtos e/ou armas deverão ficar e de onde as unidades que fazem a distribuição irão tirá-las. Qualquer construção onde fica uma carga de produtos ou armas é considerada um Estoque.

2.3.10Unidade do tipo Construção Posto de Negociação (UCP ou Posto)

É o local onde o Gerente realiza negociações com os Distribuidores e com Clientes, este último com menos freqüência.

2.3.11 Unidade do tipo Construção Ponto de Distribuição (UCD ou Ponto)

É um posto nas ruas onde se vende produtos. Não é um local físico, o Ponto é definido pelo local de ação de um Distribuidor: ao mandar um Distribuidor trabalhar clicando no botão do menu o jogador deverá então definir o local do Ponto. Portanto, o jogador não precisa comprar um Ponto, o Ponto não tem custo para ser estabelecida.

Esse local deverá ser representado por uma bandeira, com um Círculo de Atuação próprio.

O Ponto tem uma distância máxima do Posto para ser definido, já que um Distribuidor precisa estar próximo ao Posto para se reabastecer. Isso força o jogador a criar novos Postos.

2.4 Atributos

Algumas das novas funcionalidades deste jogo existem por conta dos novos atributos criados para tornar o jogo diferente dos demais e mais próximo da realidade. Neste tópico estão listados todos os atributos, das unidades, construções e jogador.

2.4.1 Vida

É a quantidade de vida, ou energia, que uma unidade tem. A unidade morre quando a Vida é zerada.

A Vida é recuperada, com o tempo, quando a unidade está dentro da Área de Atuação do jogador, obviamente o Círculo de Atuação gerado pela unidade não conta.

2.4.2 Habilidade

A Habilidade mede o nível de habilidade que a unidade tem no manuseio de armamento. Essa medida não leva em consideração o tipo de armamento utilizado pela unidade, para simplificar o jogo é como se utilizar um fuzil fosse a mesma coisa que utilizar uma granada. O objetivo aqui é tornar o jogo diferente de jogos como *X-Com*, onde o jogador deve se preocupar em escolher a arma mais propícia a unidade, e sim fazer com que o jogador se preocupe com as ações que criam o equilíbrio entre os atributos (descritos em seguida), além de vender produtos e conquistar territórios.

A Habilidade aumenta de forma gradual a cada combate sofrido pela unidade.

A Habilidade deve ser representada ao jogador através de categoria (Ex.: Muito Bom, Muito Ruim) apesar de internamente ser representada por um número, pois assim é mais real, afinal de contas ninguém diz que tal pessoa é um atirador nota 6.

As unidades não armadas também devem ter o atributo Habilidade (com exceção do Veículo), já que são todas tratadas como pessoas que têm profissões associadas e dinâmicas e não unidades com funções pré-definidas e estáticas.

2.4.3 Fidelidade

A Fidelidade indica o quanto a unidade é fiel ao jogador. Uma unidade com um nível de Fidelidade baixo pode causar os seguintes problemas para o jogador:

- Roubo: de tempos em tempos o jogo deverá fazer um teste de Fidelidade versus quantidade carregada para cada unidade carregando produtos para verificar se a unidade irá roubar a carga e desaparecer do jogo. É preciso ter um nível muito baixo de Fidelidade para que isso aconteça. Toda vez que houver um roubo o jogador deverá receber uma mensagem de aviso constando o nome da unidade (sim, cada unidade terá um nome, como descrito posteriormente).
- Fugir da batalha / se entregar: ao enviar uma unidade para batalha, o jogo deverá fazer um teste de Fidelidade versus relação entre número de unidades do oponente e numero de unidades do jogador na batalha versus Fama para verificar se a unidade irá fugir ou se render. Esse teste é realizado

a cada momento da batalha, para cada unidade. É preciso ter um nível médio-baixo de Fidelidade para não falhar no teste, a relação entre unidades deve contar bastante no teste. A relação do atributo Fama nesse teste será explicada logo abaixo em sua descrição.

- Entregar informações ao oponente: uma unidade capturada pelo oponente pode entregar informações relevantes sobre o jogador ao oponente, como introduzido no tópico Área de Atuação, através de tortura. A tortura é uma funcionalidade descrita no tópico Conquista de Território, mas pode-se adiantar que no caso de tortura, um teste de Fidelidade é realizado para verificar se a unidade entregará informações ou não. É preciso ter um nível alto de Fidelidade para não falhar no teste.
- Entregar informações ao jogador: a unidade do jogador que se movimentar por um território deverá reportar os territórios que são Área de Atuação do oponente. Porém isso pode não acontecer, o jogo deve fazer um teste de Fidelidade da unidade e Relevância do território, caso falhe a unidade não reportará o que viu. A Relevância é um atributo das construções e será apresentado posteriormente. Também deve ser feito um teste de Fidelidade versus Fama da unidade inimiga, toda vez que a unidade do jogador encontra uma unidade inimiga. É preciso ter um nível baixo de Fidelidade para falhar nestes testes.
- Traição: de tempos em tempos o jogo deverá fazer um teste de Fidelidade versus Conhecimento para cada unidade, para que ele traia o jogador. Caso falhe, a unidade deverá desaparecer do jogo e tentar oferecer informações,

como acontece no caso de tortura, em troca de dinheiro. Toda vez que houver uma traição o jogador deverá receber uma mensagem de aviso constando o nome da unidade.

O controle das ações pelo nível de Fidelidade seria muito fácil se estes fossem apresentados ao jogador como forma de número, portanto para dificultar, apesar desse atributo ser representado internamente por um número, ele deverá ser apresentado ao jogador como uma categoria que engloba uma faixa de números (Ex.: Fidelidade Baixa, Fidelidade Média, Fidelidade Alta). Assim o jogador não poderá distinguir unidades com a mesma categoria de Fidelidade porém diferentes, tendo mais chances de errar.

A Fidelidade é um atributo calculador através do atributo Moral, do jogador, e do tempo em que a unidade foi contratada pelo jogador. Quanto maior o tempo maior a Fidelidade. A Moral será descrita posteriormente, porém pode-se dizer que quando alguma ação do jogador muda o nível de Moral, uma porcentagem randômica da diferença deve ser repassada à Fidelidade da unidade. A randomicidade acontece por que uma ação do jogador deve ter efeitos diferentes em pessoas diferentes.

2.4.4 Conhecimento

O Conhecimento é um atributo ligado diretamente a Área de Atuação do jogador. Uma unidade ganha Conhecimento ao se movimentar pela Área de Atuação, e o Conhecimento aumenta de acordo com a Relevância do local.

O Conhecimento deve ser controlado pelo jogador baseando-se na Fidelidade. Unidades com Fidelidade baixa não devem conhecer locais relevantes, para evitar algumas das coisas ruins descritas no atributo Fidelidade. Alguns exemplos: um Distribuidor pouco fiel não deve trabalhar para um Posto central, importante. Deve trabalhar em um Posto próxima a Área de Atuação do inimigo, algum lugar onde já é praticamente um caso perdido ou um local conhecido pelo inimigo. A Casa do jogador deve ser guardada apenas pelos Soldados mais fiéis. O Estoque deve ser conhecido pelo menor número de pessoas possível.

No tópico de Conquista de território será descrito como o Conhecimento é importante no caso de uma tortura. Uma unidade que sabe pouco só pode falar pouco. No tópico de Inteligência Artificial será descrito como as informações dadas aos inimigos se relacionam com o Conhecimento e também como acontece o ganho de Conhecimento.

2.4.5 Fama

A Fama é um atributo que apresenta vantagens e desvantagens. É analisada sob o ponto de vista jogador-unidade inimiga. Um Soldado ou Gerente que entrar em batalha com qualquer outra unidade terá o seu nível de Fama aumentado. Isso significa que alguém fica conhecido quando mata outra pessoa.

Um Distribuidor também tem seu nível de Fama aumentado quando vender produtos.

Um Gerente que tem um Posto com maior circulação de produtos (i.e, mais vendas) deverá ter seu nível de Fama aumentado.

A vantagem da Fama é o "medo" proporcionado ao oponente no momento da batalha, este medo é simulado através do teste "Fugir da batalha / se entregar" explicado anteriormente na descrição do atributo Fidelidade. Perceba que para um Distribuidor a Fama não acarreta vantagem, já que ele não pode atacar, a não ser que seja promovido.

A Fama tem uma forma de representação diferente de todos os atributos: ela é representada pelo nome da unidade escrito logo acima da unidade, como visto no jogo *Ultima Online*, porém permanente. Isso significa que o jogo deve ter uma lista grande de nomes que são associados às unidades. Essa representação feita com o nome cria um vínculo entre o jogador e a unidade, onde a unidade deixa de ser uma simples unidade aparentemente igual as outras e passa a ser conhecida

pelo jogador. O jogador não terá que ficar procurando aquele Soldado preferido pelo mapa, ele verá o nome dele na tela.

O nome deve aparecer para todas as unidades do jogador, porém só em alguns momentos os nomes das unidades do oponente serão exibidos:
No nível mais baixo de Fama o nome da unidade terão seus nomes escritos com um tamanho de fonte bem pequeno.

A cada nível maior de Fama, o nome da unidade deve aparecer com uma Fonte maior, significando que é uma unidade mais importante, e se a unidade for um Gerente, ela deve ter seu nome escrito em negrito, além de ser grande, pois o Gerente é a unidade mais importante do jogo, já que é o representante de um Posto.

As unidades do oponente, independente do nível de Fama, deverão ter seus nomes aparecendo apenas quando adentrarem a Área de Atuação do jogador ou quando forem reconhecidas por uma unidade do jogador ao encontrá-la (dependendo ainda do teste de Fidelidade versus Fama descrito no atributo Fidelidade). Esse reconhecimento da unidade pelo inimigo facilitado pela Fama é a desvantagem que este atributo traz.

Outra desvantagem da Fama é que quanto mais Fama uma unidade tiver, mais Moral ganhará o inimigo que matar essa unidade. Isso será definido na descrição do atributo Moral.

2.4.6 Relevância

A Área de Atuação é muito importante e não deve ser revelada aos oponentes. Ao conhecer a Área de Atuação o jogador poderá saber onde estão concentradas as forças do oponente, bem como seus Estoques e Postos.

A importância de um local é dada pela sua Relevância, atributo dado a cada tipo de construção que é alterado com o tempo de acordo com os seguintes fatores: para um Posto e uma Ponto é calculado através do dinheiro gerado naquele local por unidade de tempo, para um Estoque e também para um Posto é calculado através da quantidade de dinheiro armazenada em forma de produtos e armas. A Sede é uma construção com Relevância estática e alta.

Como descrito no tópico Área de Atuação, a Área de Atuação é representada pelos Círculos de Atuação que cada construção e unidade tem, com graus de intensidade de cor diferentes. Esses graus de intensidade são a representação da Relevância, quanto maior a Relevância mais intensa a cor, não é possível obter a Relevância de um local por outra característica que não seja a intensidade da cor de seu Círculo de Atuação.

Uma Área que é intersecção de dois ou mais Círculos de Atuação deve ter uma Relevância maior, os graus de intensidade de cor devem ser somados.

De uma certa forma a Relevância é o atributo Fama para uma construção, pois como foi descrito no atributo Fidelidade, uma unidade que passar por uma construção deverá fazer um teste de Fidelidade e Relevância para verificar se ela

revelará ou não a Área de Atuação vista. Isso significa que um local de Relevância alta é um local onde acontecem várias negociações ou que armazena uma grande quantidade de produtos valiosos e provavelmente chama a atenção das pessoas.

2.4.7 Moral

A Moral pode ser considerada o atributo mais importante do jogo, por uma série de fatores. É um medidor do poder, influência do jogador sobre suas unidades e sobre a comunidade.

A Moral é um medidor global, resultado das ações tomadas pelo jogador e determina como suas unidades vão responder ao jogo após essas ações, isto é, algumas ações fazem com que a Moral caia e outra fazem com que a Moral suba e isso mudará a Fidelidade do jogador.

A Moral tem representação interna numérica, porém, deve ser mostrada ao usuário como um nível na tela, algo como um termômetro ou qualquer coisa do tipo.

Quando o jogador sofre um aumento em sua Moral seu inimigo sofre um decréscimo de Moral e o tamanho desse decréscimo é igual a uma fração do aumento do jogador. Da mesma maneira, se o jogador sofrer uma ação descrita na tabela, sofrerá um decréscimo de Moral e seu inimigo um aumento.

2.5 *Compra de Produtos*

Para realizar a venda de produtos o jogador deverá primeiramente comprá-las através de intermediadores.

O jogo deve apresentar ao jogador um menu, acessível a qualquer instante, com uma lista de produtos que o jogador possa comprar. Quanto maior a quantidade, menor o custo da compra.

A compra deverá chegar em um veículo que passa a ser controlado pelo jogador, após um tempo randômico de aproximadamente 1 minuto da confirmação da compra. O jogador deve se preparar para essa chegada, pois quanto maior a quantidade de produtos comprada mais atenção ela chamará e conseqüentemente maior a probabilidade do veículo ser interceptado pelos Saqueadores durante o percurso, fazendo com que o jogador perca a compra. Detalhes sobre a ação dos Saqueadores na compra serão descritos na seção de Inteligência Artificial e detalhes sobre combate serão descritos no tópico de Conquista de Território.

O jogador deverá levar o veículo para uma construção Estoque ou Posto, após isso o controle do veículo passa a ser do computador, que o tira do jogo. O jogador terá a oportunidade de tomar o estoque (a carga, não a construção "Estoque") do oponente e também poderá ter o seu estoque tomado. Maiores detalhes descritos no tópico de Conquista de Território.

Essa opção da quantidade comprada pelo jogador oferece uma relação segurança/economia que deve ser balanceada pelo jogador como ele preferir: comprar uma carga grande é mais inseguro por dois motivos: o jogador poderá perder a sua carga no transporte durante a compra e poderá perder a carga quando tiver o Estoque tomado. Uma compra menor oferece mais segurança, porém é mais cara e deve ser feita com mais freqüência, além da possibilidade de um certo produto desejado estar indisponível no momento.

2.6 Venda de Produtos

A venda de produtos acontece com as associações Gerente/Posto e Distribuidor

/Ponto. Além disso, também existe uma associação Gerente/Distribuidor.

Gerente/Posto e Gerente/Distribuidor

Para associar um Gerente a um Posto basta selecionar o Gerente e clicar com o botão direito em um Posto. O Gerente deverá ir para o Posto e começar suas negociações.

O Gerente é uma unidade que pode ser armada e, portanto, algumas vezes será utilizada fora de seu Posto. Ao selecionar um Gerente e move-lo para fora do Posto o Gerente perderá a sua função de negociador, servindo apenas como se fosse um Soldado. Porém, ele não perde a profissão ou representação Gerente, ele só está fora do local de trabalho. Para que o Gerente possa voltar ao Posto o

jogador pode optar por associa-lo novamente aquele Posto ou simplesmente clicar em um botão do menu tipo "retorne à atividade".

Um Gerente pode ser associado a um outro Posto; basta realizar o processo de associação a um Posto para desassociá-lo da antiga e associa-lo ao novo.
Uma Posto comporta apenas um Gerente. O jogo não irá permitir a associação de um Gerente a um Posto que já possua um Gerente.

Estando em um Posto, o Gerente realizará suas negociações com os Clientes que entrarem no Posto, fornecendo produtos. Essas negociações acontecem com uma freqüência definida na seção de Inteligência Artificial.

Além das vendas no Posto, o Gerente também é responsável pelo reabastecimento do Distribuidor. O jogo deve mostrar essa relação, quando um Distribuidor faz o reabastecimento ele deverá "falar" com o Gerente. Não há reabastecimento se não houver Distribuidor, isto é o Posto não funciona sem um Gerente.

Quando os produtos do Posto acabar o Gerente deverá automaticamente ir ao Estoque mais próximo para o reabastecimento.

2.7 Distribuidor/Ponto

Para fazer com que o Distribuidor venda produtos, o jogador deverá selecioná-lo e clicar no botão do menu que dá a ordem para começar a trabalhar e então deverá definir um Ponto de Distribuição, clicando em alguma parte do mapa que não contenha construções.

Estando no Ponto o Distribuidor realizará suas negociações com os Clientes que aparecerem, em uma freqüência definida na seção de Inteligência Artificial.

Toda vez que a mercadoria do Distribuidor acabar ele deverá voltar ao Posto mais próximo para o reabastecimento, porém a bandeira representativa do Ponto deve permanecer lá, como também o Círculo de Atuação do Ponto.

Um Ponto é perdido quando o Distribuidor associado a esse Ponto morre, ou quando outro local é determinado para ser o Ponto daquele Distribuidor.

2.8 Compra de Armas

A compra de armas é feita de uma maneira muito semelhante à compra de produtos: deve-se acessar um menu, fazer as compras - que chegarão em um veículo - e guiar o veículo até um Estoque. Como na compra de produtos, a quantidade de armas compradas influencia no preço da compra.

Para diminuir a complexidade do jogo as armas tem munição infinita, isto é, poderá ser utilizada até ser perdida, com exceção de alguns tipos de armas de uso único, isto é, que só podem ser utilizadas uma vez (Ex.: granada de mão).
A perda da arma acontece na morte da unidade que a utiliza.

A associação de uma arma a uma unidade acontece de três maneiras: na associação de uma função a essa unidade, na troca de arma ou na adição de uma arma de uso único.

Na associação de função, como descrito na seção Unidades e Construções, caso a unidade necessite de uma arma, o jogo deverá exibir um menu com as armas que o jogador possui para fazer a escolha da arma para essa unidade.

O jogador poderá trocar a arma da unidade se quiser, clicando na unidade e selecionando opção em menu.

Através do mesmo menu o jogador poderá adicionar armas de uso único, de acordo com a quantidade máxima definida na tabela a seguir:

Ao ganhar uma arma ou trocar de arma, a unidade deverá automaticamente ir ao Estoque mais próximo que possua a arma que ela necessite, pegar a arma e deixar a eventual arma que possuir.

2.9 Conquista de território

Neste tópico está a descrição de como ocorrem as batalhas no jogo e as conquistas de território decorrentes dessas batalhas.

Todas essas ações descritas que podem ser acionadas pelo jogador também podem ser acionadas pelo computador, seu oponente.

Como descrito anteriormente no atributo Fama, uma unidade do oponente pode não ser reconhecida, e então não há como diferenciar uma unidade do oponente de uma unidade do jogo, um simples morador andando pela rua, se o jogador não souber que a unidade é do oponente. Porém se uma unidade não reconhecida aparecesse pelas ruas andando com um rifle o jogador saberia que é

uma unidade armada. Por isso, a solução simples é que a unidade deve aparecer desarmada até o momento de um ataque, ao chegar próximo da unidade a ser atacada. É uma solução simples pois não modela a realidade, ninguém consegue esconder um rifle no bolso. A unidade deverá "esconder" a arma assim que o computador verificar que não há unidades inimigas dentro de um raio da unidade.

Em situações explicadas anteriormente uma unidade poderá fugir do jogador, simplesmente sumindo de sua vista. O jogo deverá simular essa situação levando a unidade a algum local onde o jogador não consiga vê-la e então retirar a unidade do jogo, como por exemplo, uma unidade vira a esquina e simplesmente desaparece atrás do muro. O jogador perde o controle dessa unidade, sendo que a única ação que possa ser tomada é mandar matar essa unidade, porém será difícil, pois ele deverá perceber que a unidade irá fugir e o tempo até ela sumir é muito pequeno. Como matar uma unidade do próprio jogador será explicado posteriormente.

Quando uma unidade inimiga se entregar em uma batalha ela deverá parar de atacar e todos deverão parar de atacar essa unidade. Ainda assim a unidade poderá ser eliminada, basta o jogador ou oponente ordenar um ataque a unidade após a rendição.

A unidade rendida seguirá a unidade inimiga por onde ela for, caso o jogador queira liberar a unidade, deverá fazê-lo através do menu da unidade rendida, clicando em um botão. Assim, a unidade volta a ser controlada pelo jogador original, mas sem armas.

O objetivo de se conseguir uma unidade inimiga rendida é a tortura. Como explicado anteriormente, ao torturar uma unidade inimiga o jogador poderá receber informações importantes sobre o inimigo.

Toda vez que o jogador tentar obter informações do inimigo através de tortura o jogo deverá subtrair uma fração do índice máximo de Vida que a unidade pode ter e, caso ainda esteja viva, fazer um teste de Fidelidade. Caso falhe, a unidade deverá dar informações de acordo com o seu grau de Conhecimento. Como foi dito anteriormente, é preciso ter um nível alto de Fidelidade para não falhar no teste. O botão do menu para tortura da unidade deve ser desabilitado caso ela forneça alguma informação. Toda vez que uma unidade entregar informações ao oponente, o jogador deverá receber uma mensagem de aviso constando o nome da unidade.

Para realizar um ataque deve-se selecionar a unidade armada e clicar com o botão direito do mouse em uma unidade do jogo que não seja a própria unidade do jogador. Isso fará com que a unidade se mova até a outra e a ataque.

Um ataque automático acontecerá toda vez que uma unidade inimiga identificada seja visível à unidade do jogador.

As unidades terão um menu próprio de modo de combate: o jogador poderá escolher, para cada unidade, se a unidade atacará sempre que puder, que é a opção padrão descrita anteriormente, ou se agirá na defensiva, atacando apenas se for atacada.

Este menu dá ao jogador a opção de criar dois tipos de exército: um que fica parado, defendendo algum ponto importante, como um Estoque por exemplo, de algum possível ataque inimigo e outro que sai para atacar o inimigo.

Às vezes o jogador vai precisar matar alguma unidade pertencente a ele mesmo, para isso ele deve atacar a unidade da mesma forma que um ataque normal, porém deve deixar pressionada a tecla CTRL ao clicar com o botão direito na unidade a ser atacada.

2.10 Conquista de Construções

A duas maneiras de se conquistar território no jogo, uma é pacífica, através da compra/associação de construções. A outra é por tomada de construções do inimigo, e será descrita neste tópico.

Diferentemente do que é normalmente encontrado nos jogos de estratégia, as construções não podem ser destruídas. Não é muito real alguém destruir totalmente uma casa usando uma pistola e, além disso, esse modo de batalha forçará uma luta de unidade contra unidade, pessoa contra pessoa, e não uma batalha tipo castelo contra unidade como visto em outros jogos.

Para tomar uma construção inimiga, com exceção do Ponto de Distribuição que não pode ser tomado, deve-se deixar uma unidade dentro do Círculo de Atuação gerado pela construção inimiga por 1 minuto, sem sofrer batalhas. A construção então passa a ser do jogador, tendo um Círculo de Atuação da cor do jogador e possibilitando que ele utilize a construção.

2.11 Evolução

A evolução é o que permite que o jogo não fique repetitivo, ela faz com que o jogador queira continuar jogando para conseguir coisas novas. A evolução do jogo são novidades que aparecem no decorrer do jogo, e não a evolução que o jogador sofre ao conseguir mais dinheiro, território, etc.

A evolução acontece em dois aspectos do jogo: nos produtos e nas armas, porém esse é um dos principais tópicos que podem ser aperfeiçoados em outras versões do documento para criação de um jogo melhor.

No início do jogo, o jogador consegue comprar apenas alguns tipos de produtos e armas. Ao longo do jogo, novos produtos e armas surgem, estes com maior freqüência.

2.12 Inteligência Artificial

Se a seção de Mecânica do Jogo descreve o que o jogador pode fazer no jogo e as estruturas do jogo a seção de Inteligência Artificial descreve como o computador deve responder as ações tomadas pelo jogador, além de descrever como o computador deve agir quando o jogador não estiver fazendo nada.

Na seção de Mecânica do Jogo, já foram abordados alguns tópicos de como o computador deve responder a ações do jogador e, portanto, não serão descritas novamente.

Uma maneira simples de descrever o comportamento dos oponentes é dizendo que eles agem como se fossem um jogador, como se o jogador estivesse jogando um jogo multiplayer, com oponentes reais.

Todos os testes descritos anteriormente devem ser aplicados aos oponentes também, se o teste envolver um outro jogador (o jogador ou outro oponente), o computador deverá realizar um sorteio.

Além disso, o computador também deve cuidar de outros aspectos do jogo, como o comportamento dos Saqueadores, que não é considerada um oponente jogado pelo computador, o comportamento dos Clientes, o comportamento dos Moradores, que não são controladas pelos oponentes e outras.

A seção está dividida por tópicos de descrição do comportamento de cada um desses tipos de unidades citados acima, e também por outros tópicos.

2.13 Comportamento dos Saqueadores

Os Saqueadores são uma característica do jogo que foi criada para deixá-lo mais equilibrado, atacando aqueles que são mais fortes. Eles não podem ser derrotados, totalmente eliminados. Eles tem atuação esporádica.

Os Saqueadores podem ser comparados aos lobos que navegam pelo mapa no jogo *Age of Empires* e atacam quando o jogador se aproxima, porém os Saqueadores não aparecem sempre no mapa, eles chegam ao mapa, como se estivessem em um outro lugar que não é representado no mapa, para realizar ataques contra o jogador ou contra o oponente controlado pelo computador.

O computador deve realizar um ataque randômico dos Saqueadores de tempos em tempos, aproximadamente de 10 em 10 minutos com 60% de chances de ocorrer. O ataque deve ser sorteado entre os jogadores, isto é, entre o jogador e seus oponentes. O período entre um ataque e outro deve diminuir de acordo com o número de oponentes, 10 minutos deve ser o tempo para um tipo de jogo sem nenhum oponente.

Os Saqueadores sempre utilizam Veículos, vai até as proximidades de uma Posto e realiza um ataque.

Os Saqueadores devem atacar os jogadores em *40% * (quantidade comprada / quantidade de compra considerada grande)* das vezes que eles realizam compras de produtos ou armas. Como descrito anteriormente, isso deve acontecer para criar um equilíbrio segurança / economia, onde as compras grandes chamam mais atenção de Saqueadores, porém são mais baratas.

Esse é o comportamento dos Saqueadores definido para esta versão do documento. O comportamento deve ser redefinido em outras versões, aprimorado. Os ataques de Saqueadores devem depender de outros fatores ou de acordo com a popularidade dos jogadores ou ganho de dinheiro dos jogadores, ou durante uma batalha ou de acordo com a proximidade de um Posto em um local periférico, longe de qualquer Área de Atuação "antiga" (isto forçaria que o jogador protegesse seus novos Postos).

Além disso, deve ser descrita uma funcionalidade de negociação com os Saqueadores, onde o jogador pode fazer pagamentos para obter proteção e coisas do tipo.

2.14 Comportamento dos Moradores

Os Moradores são unidades que devem ficar caminhando pelo mapa, dando um aspecto de lugar habitado no jogo. Devem fugir de tiroteios, toda vez que um Morador entrar num raio de distância de uma batalha ele deve fugir sem mudar de direção, a não ser, obviamente, que seja obrigado a virar, até alcançar um raio de distância maior.

Os Moradores podem sofrer ataques do jogador, porém o jogador terá que dar a ordem para matá-la, isto é, uma unidade nunca deverá atacar um Morador simplesmente por ele adentrar uma área, como a unidade faz com inimigos conhecidos.

Um oponente controlado pelo computador nunca ataca os Moradores, não há motivo para isso.

Como explicado agora, os Moradores não devem sofrer ataques, a não ser que o jogador especifique que queira atacá-los, porém os Moradores ainda podem sofrer ataques randômicos em uma batalha, resultado de balas perdidas. Isso será descrito nessa seção, no tópico de Batalhas.

O número de Moradores em um jogo deve ser constante, toda vez que um Morador morre deve surgir outro Morador em algum lugar do mapa, vindo de outro local que não aparece no mapa, como acontece com os Saqueadores. Deve haver um intervalo entre a morte do Morador e a chegada do outro, de 1 minuto.

Deverão existir Moradores em Veículos, com o objetivo de confundir o jogador: como será descrito, Clientes virão em Veículos comprar produtos, e o jogador poderia simplesmente rastrear esses Veículos para descobrir a localização de Pontos e Postos dos oponentes. Esses Moradores poderão parar seus carros e sair andando randomicamente e eventualmente pegar outros Veículos.

2.15 Comportamento dos Clientes

Os Clientes são em sua maioria Moradores que mudam de função. O computador deve levar, dependendo de condições descritas no tópico de Compra de Produtos, um Morador "transformado" em Cliente a um Posto ou Ponto, fazer uma negociação de compra de produtos e depois "transformar" o Cliente em Morador novamente e deixar com que ele navegue pelo mapa.

Essa "transformação" nada mais é que uma transferência de papéis, onde uma unidade que fica navegando pelo mapa randomicamente pára e vai até um Gerente ou Distribuidor fazer compras. Não há nenhuma mudança gráfica.

Existem também Clientes que vêm de lugares distantes, podem aparecer em Veículos, e devem surgir no mapa como os Saqueadores, isto é, vindo de um local que não aparece no mapa. Esses clientes devem se aproximar de um Gerente ou Distribuidor, realizar a compra e sair do mapa.

Clientes em Veículos que fazem compras em Pontos não precisam sair do carro para realizar uma compra.

2.16 Venda de Produtos

A negociação de compra de produtos deve ocorrer graficamente da seguinte maneira: a unidade compradora deverá se aproximar da unidade vendedora e ficar uma de rosto virado para outra, como se conversassem, por 5 segundos. A unidade compradora deverá então ir embora e enquanto isso um símbolo "$" deverá aparecer durante 2 segundos, como visto no jogo *Tycoon Transports*, representando que o vendedor ganhou dinheiro. Neste momento também deve ser incrementada a quantidade de dinheiro que o jogador (ou oponente) tem. Esse símbolo indicando a entrada de dinheiro só deve aparecer para as negociações do jogador, as negociações dos oponentes não devem ser obvias ao jogador, respeitando o conceito de que o jogador não conhece tudo que acontece no mapa.

As negociações devem ocorrer somente quando as unidades vendedoras estão em condições de negociar, isto é, não deverão ocorrer negociações próximas as batalhas ou quando uma unidade vendedora não possuir produtos para vender.

O período entre uma venda de produto e outra deverá ser de 10 segundos * número de jogadores (isto é, jogador + oponentes, e não suas unidades). O computador deverá sortear quem fará a venda da seguinte maneira:

Primeiro sorteia-se se a venda será feita por uma Gerente ou um Distribuidor de acordo com a seguinte fórmula: $9 * ND / NT$ de chance de ser um Distribuidor e NG / NT de chance de ser um Gerente, sendo ND o número total de Distribuidores no jogo (jogador mais oponentes), NG o número total de Gerentes no jogo e NT igual a $ND + NG$. Isto quer dizer que para uma quantidade igual de Distribuidores e Gerentes 90% das vendas serão feitas por Distribuidores.

Esse sorteio fará com que o jogador tenha Pontos de Distribuição, mesmo sabendo que as vendas feitas pelo Gerente são mais lucrativas. Caso o jogador tente construir apenas Postos, ele vai perder a maior parte dos clientes para os Distribuidores dos oponentes.

Sabendo o tipo de unidade que fará a venda o computador deverá então realizar um novo sorteio para determinar qual unidade fará a venda. Esse sorteio deverá ser igualitário e proporcional ao número de unidades vendedoras, do tipo escolhido, de cada jogador. Isto quer dizer que se o jogador tiver 9 vezes mais Gerentes que a soma dos Gerentes de seus oponentes, ele terá uma chance de 90% de ser sorteado em uma venda feita por Gerente.

Encerramos aqui a descrição dada pelo documento de design. Os poucos detalhes que não foram dados e serão implementados serão discutidos pelo grupo na criação dos casos de uso.

Apêndice B – Descrição dos casos de uso

Os casos de uso apresentados estão divididos de acordo com a estrutura encontrada no projeto. Nesta versão do documento não estão todos os casos de uso e nós procuramos primeiramente criar os casos de uso onde existe maior interação por parte do usuário. Os casos de uso apresentados estão distribuídos nos seguintes tópicos:

1. **Casos de Uso correspondentes à Interface:** Basicamente, são os casos de uso de manipulação (seleção, movimentação, etc.) das Unidades e Objetos do jogo.
2. **Casos de Uso correspondentes aos Menus:** São os casos de uso para realização de funções básicas do sistema (salvar, carregar, sair) e outros que implementam funcionalidades do jogo. Menus fazem parte da interface, porém os casos de uso aqui representam idéias bem distintas das dos casos de uso de Interface e por isso merecem essa divisão.
3. **Casos de Uso correspondentes às Unidades:** Descrevem as ações que o jogador pode disparar com as Unidades e Objetos do jogo. Os casos de uso correspondentes à Interface são a base para estes casos de uso.
4. **Casos de Uso Automáticos correspondentes às Unidades:** Descrevem as ações que o computador pode disparar com as Unidades e Objetos do jogo.

A nomenclatura e explicação dos elementos do jogo são dadas no Documento de Design, porém, para criar uma estruturação clara das Unidades e Objetos do jogo, apresenta-se um glossário:

- *Unidade do tipo Personagem Gerente:* UPG
- *Unidade do tipo Personagem Distribuidor*: UPD
- *Unidade do tipo Personagem Soldado*: UPS
- *Unidade do tipo Personagem*: UP = UPG + UPD + UPS
- *Unidade do tipo Veículo*: UV
- *Unidade do tipo Dinâ mica*: UD = UP + UV
- *Unidade do tipo Construção Sede*: UCS
- *Unidade do tipo Construção Posto de Negociação*: UCP
- *Unidade do tipo Construção Ponto de Distribuição*: UCD
- *Unidade do tipo Construção*: UC = UCP + UCS + UCD
- *Non-personal Caracter*: NPC = Morador + Cliente
- *Carga*: Arma ou Produto (isto é, uma carga pode significar tanto uma arma quanto um produto).
- *Objeto*: Qualquer Unidade e itens que possam ser selecionados (Cargas, árvores, pedras, etc.).

Padronização: Quando se diz simplesmente "seleciona", significa "seleciona com o botão esquerdo do mouse".

1. Casos de Uso correspondentes à Interface

Caso de uso 1.1: Seleção de um Objeto.
Descrição: Este caso de uso descreve a seleção de um único Objeto.
Atores: Usuário, Objeto.
Pré-condição:
Seqüência de eventos:

1. Usuário seleciona o Objeto clicando sobre ele com o botão esquerdo do mouse.

Pós-condição: Objeto selecionado, com exibição de seus respectivos menus e uma indicação de que o Objeto foi selecionado (círculo em volta da unidade, no chão).
Extensões:
Inclusão:

Caso de uso 1.2: Seleção de Unidades Dinâmicas pela seleção de área.
Descrição: Este caso de uso descreve a seleção de uma ou mais Unidades Personagem e Unidades Veículo.
Atores: Usuário, Unidades Dinâmicas.
Pré-condição:
Seqüência de eventos:

1. Usuário seleciona uma área utilizando o botão esquerdo do mouse (como na seleção de vários ícones no Windows).
2. Sistema seleciona todas as Unidades contidas na área selecionada pelo usuário.

Pós-condição: Unidades selecionadas, com exibição de menus comuns a todas as Unidades e uma indicação de que a Unidade foi selecionada (círculo em volta da unidade, no chão), para cada Unidade.
Extensões:
Inclusão:

Caso de uso 1.3: Movimentação de Unidades Dinâmicas.
Descrição: Este caso de uso descreve a movimentação de uma ou mais Unidades Dinâmicas pelo mapa.
Atores: Usuário, Unidades Dinâmicas.
Pré-condição: Unidade(s) selecionada(s).
Seqüência de eventos:

1. Usuário clica com o botão direito no mouse sobre um local destino no mapa.
2. Sistema faz com que as Unidades selecionadas se movam até a proximidade do local destino. Caso seja uma única unidade ela deverá parar exatamente no ponto destino.

Pós-condição: Unidades selecionadas no local destino.
Extensões:
1. Usuário clica com o botão direito do mouse sobre uma UD: Sistema efetua caso de uso de ataque.
2. Usuário clica com o botão direito do mouse sobre uma UC: Sistema efetua caso de uso de Entrada de Unidades Personagem em Unidades Construções.

Inclusão: Caso de uso de Ataque de uma UD a outra UD ou NPC.

Caso de uso 1.4: Entrada de Unidades Personagem em Unidades Construções
Descrição: Este caso de uso descreve a entrada de uma ou mais Unidades Dinâmicas pelo mapa.
Atores: Usuário, Unidades Dinâmicas.
Pré-condição: Unidade(s) selecionada(s).
Seqüência de eventos:
1. Usuário clica com o botão direito no mouse sobre uma Unidade Construção destino.
2. Sistema faz com que as Unidades selecionadas se movam até a Unidade Construção destino.
3. Sistema faz com que as Unidades adentrem a Unidade Construção destino.

Pós-condição: Unidades selecionadas dentro do local destino.
Extensões:
1. Unidade Construção cheia (passo 2): Não havendo nenhum lugar na Unidade Construção a entrada (passo3) deve ser cancelada, porém a movimentação até as proximidades da Unidade Construção deverá ser concluída (passo 2).

Inclusão: Caso de Uso Movimentação de Unidade Dinâmica (Passo 2).

2. Casos de Uso correspondentes aos Menus:

Caso de uso 2.1: Seleção de Novo Jogo no Menu Principal.
Descrição: Este caso de uso descreve a seleção de um novo jogo.
Atores: Usuário.
Pré-condição:
Seqüência de eventos:
1. Usuário seleciona opção Novo Jogo através do Menu Principal.
2. Sistema abre janela com mapas disponíveis no diretório de mapas.
3. Usuário seleciona mapa e confirma.
5. Usuário descarta o jogo atual.
6. Sistema abre o mapa escolhido, com um novo "jogo".

Pós-condição: Usuário em novo jogo com novo mapa.
Extensões:

1. Já existe um jogo em execução: Sistema pergunta se Usuário deseja salvar o jogo atual. Em caso positivo, executa o caso de uso salvar jogo. Caso contrário, vai para passo 2 (passo 1).

Inclusão: Salvar Jogo.

Caso de uso 2.2: Carregar Jogo.
Descrição: Este caso de uso descreve como carregar um jogo salvo.
Atores: Usuário.
Pré-condição:
Seqüência de eventos:

1. Usuário seleciona opção Carregar Jogo através do Menu Principal.
2. Sistema abre janela com jogos disponíveis do diretório de jogos salvos.
3. Usuário seleciona jogo e confirma.
4. Sistema carrega o jogo salvo.

Pós-condição: Usuário no jogo salvo.
Extensões:

1. Já existe um jogo em execução: Sistema pergunta se Usuário deseja salvar o jogo atual. Em caso positivo, executa o caso de uso salvar jogo. Caso contrário, vai para passo 2 (passo 1).

Inclusão: Salvar Jogo.

Caso de uso 2.3: Salvar Jogo.
Descrição: Este caso de uso descreve como salvar um jogo.
Atores: Usuário.
Pré-condição: Usuário estar com um jogo aberto.
Seqüência de eventos:

1. Usuário seleciona opção Salvar Jogo através do Menu Principal.
2. Sistema abre janela com jogos disponíveis no diretório de jogos salvos e caixa de texto para entrada de um nome.
3. Usuário seleciona jogo salvo ou entra com um novo nome para o jogo.
4. Caso tenha selecionado um jogo salvo o sistema pergunta se Usuário deseja substituir jogo existente.
5. Sistema salva jogo, guardando todas as informações do jogo, inclusive a posição de cada Unidade, de tal forma a ter exatamente o mesmo jogo quando carregá-lo.
6. Sistema volta ao jogo.

Pós-condição: Jogo salvo.
Extensões:

1. Usuário não deseja substituir jogo existente (passo 4): Sistema não salva o jogo e retorna a tela de salvar jogo.

Inclusão:

Caso de uso 2.4: Fechar Jogo.
Descrição: Este caso de uso descreve como sair de um jogo.
Atores: Usuário.
Pré-condição: Usuário estar com um jogo aberto.
Seqüência de eventos:

1. Usuário seleciona opção Fechar Jogo através do Menu Principal.
2. Sistema pergunta se Usuário deseja salvar o jogo atual.
3. Usuário descarta o jogo atual.
4. O sistema carrega tela inicial do jogo.

Pós-condição: Jogo fechado.
Extensões:

1. Usuário deseja salvar o jogo atual (passo 3): Sistema abre a janela de Salvar Jogo (Caso de Uso "Salvar Jogo") e depois continua com o passo 4.

Inclusão: Salvar Jogo.

Caso de uso 2.5: Compra de produtos.
Descrição: Este caso de uso descreve o processo de compra e chegada de produto.
Pré-condição: O usuário seleciona a opção de compra de produto no Menu do Jogador.
Seqüência de eventos:

2. Usuário seleciona o(s) tipo(s) e as quantidades de cada produto a ser comprado(s).
3. Usuário confirma a compra.
4. Sistema debita o valor da compra.
5. Sistema, após um intervalo de tempo dentro de uma faixa pré-estabelecida (50s a 70s), cria uma UV carregada com a carga de produto especificada no ponto de entrada do mapa.
6. Sistema manda um aviso ao usuário de que o produto chegou.
7. Sistema ativa contagem regressiva para descarregamento de produto (60s), exibindo na tela a contagem.
8. Usuário seleciona a UV e a guia até o ponto desejado.
9. Usuário clica em botão de descarregamento.
10. Sistema descarrega a carga e toma controle do veículo.
11. Sistema leva o veiculo para fora do cenário.

Pós-condição: Produto descarregado no cenário.
Extensões:

1. Usuário não tem dinheiro suficiente para compra (passo 2): Sistema informa usuário de não pode comprar a quantidade especificada.
2. Usuário não descarrega a carga em tempo suficiente: Sistema assume controle do veículo e o guia para fora do cenário.
3. UV não se encontra na Área de Atuação de uma UC e clica no botão de descarregamento: a carga é descarregada no chão (passo 9).

Inclusão:

Caso de uso 2.6: Compra de arma.
Descrição: Este caso de uso descreve o processo de compra e chegada de arma.
Pré-condição: O usuário seleciona a opção de compra de arma no Menu do Jogador.
Seqüência de eventos:

1. Usuário seleciona o(s) tipo(s) e as quantidades de cada arma a ser comprada(s).
2. Usuário confirma a compra.
3. Sistema debita o valor da compra.
4. Sistema, após um intervalo de tempo dentro de uma faixa pré-estabelecida (50s a 70s), cria uma UV carregada com a carga de arma especificada no ponto de entrada do mapa.
5. Sistema manda um aviso ao usuário de que a arma chegou.
6. Sistema ativa contagem regressiva para descarregamento de arma (60s), exibindo na tela a contagem.
7. Usuário seleciona a UV e a guia até o ponto desejado.
8. Usuário clica em botão de descarregamento.
9. Sistema descarrega a carga e toma controle do veículo.
10. Sistema leva o veículo para fora do cenário.

Pós-condição: Produto descarregado no cenário.
Extensões:

1. Usuário não tem dinheiro suficiente para compra (passo 2): Sistema informa usuário de não pode comprar a quantidade especificada.
2. Usuário não descarrega a carga em tempo suficiente: Sistema assume controle do veículo e o guia para fora do cenário.
3. UV não se encontra na Área de Atuação de uma UC e clica no botão de descarregamento: a carga é descarregada no chão (passo 9).

Inclusão:

Caso de uso 2.7: Compra de construção.
Descrição: Este caso de uso descreve o processo de compra de uma construção.
Pré-condição: O usuário seleciona a opção de compra de construção no Menu do Jogador.
Seqüência de eventos:

1. Usuário seleciona a construção no mapa que deseja comprar.
2. Sistema exibe tela pedindo confirmação de compra de construção.
3. Usuário confirma a compra da construção.
4. Sistema debita o valor da compra.

Pós-condição: Construção apresentando a Área de atuação com a cor do jogador..
Extensões:

1. Usuário seleciona construção que não pode ser comprada: Sistema informa usuário de que não pode comprar a construção selecionada.

2. Usuário não tem dinheiro suficiente para compra (passo 3): Sistema informa usuário de que não pode comprar a construção selecionada.

Inclusão:

3. Casos de Uso correspondentes às Unidades

Caso de uso 3.1: Carregamento de Unidade.
Descrição: Este caso de uso descreve o processo de carregamento da Unidade (pode ser UP, UC ou UV) com arma ou produto.
Atores: Usuário, Unidade e carga.
Pré-condição:
Seqüência de eventos:

1. Para a Unidade em questão, o usuário seleciona a opção de carregamento de carga.
2. Usuário seleciona com botão direito do mouse o objeto no cenário que representa a UC onde se efetuará o carregamento.
3. Se a unidade é dinâmica, se desloca pelo cenário em direção à carga selecionada.
4. Se a unidade é estática, sistema verifica se a carga está dentro da área de atuação da UC.
5. Unidade coleta a carga.

Pós-condição: Ao se selecionar a Unidade que carregou a carga, a carga é exibida no menu de propriedades da Unidade juntamente com a descrição de todas as suas características (tipo e quantidade) atualizadas.
Extensões:

1. Usuário seleciona com o botão direito do mouse um ponto ou objeto do cenário que não representa uma carga: Sistema cancela o processo de carregamento da Unidade com carga.
2. Usuário seleciona com o botão esquerdo do mouse um ponto ou objeto do cenário para carregar: Sistema cancela o processo de carregamento da Unidade com carga, e realiza a seleção do objeto clicado.
3. Usuário seleciona UPS para a realização de carregamento de carga e depois seleciona com o botão direito do mouse um produto para carregamento: Sistema cancela o processo de carregamento de carga.
4. Usuário seleciona UC para carregamento de carga e depois seleciona com o botão direito do mouse qualquer coisa que não seja carga,e que esteja dentro da área de atuação da UC: Sistema cancela o processo de carregamento da UC com carga.
5. Usuário seleciona UC para carregamento de carga e depois seleciona com o botão direito do mouse uma carga, mas que não está dentro da área de atuação da UC: Sistema cancela o processo de carregamento da UC com carga.

Inclusão:

Caso de uso 3.2: Descarregamento de carga feito pela Unidade.
Descrição: Este caso de uso descreve o processo de descarregamento de carga armazenada pela Unidade (que pode ser do tipo UP, UC ou UV).
Atores: Usuário, Unidade, carga.
Pré-condição: Para a Unidade em questão, o usuário seleciona a carga que será descarregada.
Seqüência de eventos:

1. Sistema deposita a carga no cenário, tirando-a da Unidade.

Pós-condição: Ao se selecionar a Unidade que descarregou a carga, a carga é exibida no menu de propriedades da Unidade juntamente com a descrição de todas as suas características (tipo e quantidade) atualizadas.
Extensões:
Inclusão:

Caso de uso 3.3: Ataque de uma UD a outra UD ou NPC.
Descrição: Este caso de uso descreve o processo de ataque da Unidade dinâmica a uma outra UD ou NPC.
Atores: Usuário, UD, NPC.
Pré-condição: Existe uma UD armada selecionada, que não seja UPD.
Seqüência de eventos:

1. Usuário seleciona a opção de ataque no menu.
2. Sistema entra no modo de ataque.
3. Usuário seleciona com o botão esquerdo do mouse o objeto no cenário que representa o alvo a ser atacado.
4. UD se desloca pelo cenário em direção ao objeto selecionado.
5. UD imediatamente ataca o objeto com a arma padrão.

Pós-condição: Ataque realizado pela UD.
Extensões:

1. Usuário seleciona um ponto do cenário que não seja um objeto atacável: Sistema não realiza o passo 5 (Passo 3).
2. Usuário seleciona com o botão direito do mouse um objeto para ser atacado: Sistema cancela o processo de ataque (Passo 3).
3. Usuário seleciona uma Unidade aliada para ser atacada sem manter pressionada a tecla CTRL: Sistema não realiza o passo 5 (Passo 3).

Inclusão: Caso de Uso Movimentação de Unidade Dinâmica (Passo 4).

Caso de uso 3.4: Ataque de uma UD a uma UD inimiga ou NPC diretamente, sem botão do menu.
Descrição: Este caso de uso descreve o processo de ataque da Unidade dinâmica a uma UD inimiga ou NPC, sem utilizar o menu.
Atores: Usuário, UD do jogador, UD inimiga, NPC.
Pré-condição: Unidade que irá atacar selecionada.

Seqüência de eventos:

1. Usuário seleciona com o botão direito do mouse o objeto no cenário que representa o alvo a ser atacado.
2. UD se desloca pelo cenário em direção ao objeto selecionado.
3. UD imediatamente ataca o objeto com a arma padrão.

Pós-condição: Ataque realizado pela UD contra a UD inimiga ou NPC.
Extensões:

1. Usuário seleciona com o botão direito do mouse um ponto do cenário que não seja um objeto atacável: Sistema não realiza o passo 3 (Passo 1).
2. Usuário seleciona com o botão direito uma Unidade aliada para ser atacada sem manter pressionada a tecla CTRL: Sistema não realiza o passo 3 (Passo 1).

Inclusão: Caso de Uso Movimentação de Unidade Dinâmica (Passo 2).

Caso de uso 3.5: Mudança de função de uma UP.
Descrição: Este caso de uso descreve o processo de mudança de função feita por uma UP.
Atores: Usuário e UP.
Pré-condição: Para a UP em questão, o usuário seleciona a opção de mudança de função.
Seqüência de eventos:

1. Usuário seleciona a nova função que deseja para a UP selecionada.

Pós-condição: Ao se selecionar a UP que mudou de função, esta UP apresenta em sua janela de propriedades o status indicando a nova função, bem como as opções que são disponibilizadas para tal função.
Extensões:
Inclusão:

Caso de uso 3.6: Armamento com arma de uma UP (opção disponível somente para uma UPG ou UPS).
Descrição: Este caso de uso descreve o processo de armamento de uma UP (uma UPG ou UPS).
Atores: Usuário e UP.
Pré-condição: Para a UP em questão, o usuário seleciona a opção de armamento.
Seqüência de eventos:

1. Sistema exibe menu com uma tabela relacionando construções, armas e quantidades (é importante enfatizar que as armas que são mostradas nesse menu são apenas as localizadas dentro de UCP's).
2. Usuário seleciona a arma que deseja.
3. Sistema retira a arma da UCP selecionada à UP.
4. Sistema exibe menu com as UCP's estocáveis para destino da arma antiga caso a UP possua alguma.
5. Usuário seleciona a UCP destino.
6. Sistema retira a arma antiga da UP e a deposita na UCP selecionada.

Pós-condição: Ao se selecionar a UP que mudou de arma, esta UP apresenta em sua janela de propriedades o status indicando a nova arma.
Extensões:
1. UP não possui arma. Sistema não exibe menu para deposito da arma antiga.

Inclusão:

Caso de uso 3.7: Fornecimento de cobertura feita por uma UP (opção disponível somente para uma UPG ou UPS) a uma outra UP.
Descrição: Este caso de uso descreve o processo de fornecimento de cobertura a ser realizado por uma UP a uma outra UP aliada.
Atores: Usuário e UP.
Pré-condição: Para a UP em questão, o usuário seleciona a opção de realização de cobertura.
Seqüência de eventos:
1. Usuário seleciona com o botão direito do mouse a UP que para quem deseja dar cobertura.
2. UP em questão se desloca em direção à UP que deve receber cobertura.
3. UP ataca pelo modo padrão os inimigos que circundam a UP que deve receber cobertura.

Pós-condição: UP em questão acompanhando e circundando a UP que deve receber cobertura.
Extensões:
1. Usuário seleciona um ponto do cenário que não reconheça objeto: Sistema cancela o processo de fornecimento de cobertura a ser feita por uma UPG à outra UPG.
2. Usuário seleciona uma unidade inimiga: Sistema cancela o processo de fornecimento de cobertura a ser feita por uma UPG à outra UPG.
3. Usuário seleciona com o botão esquerdo do mouse um objeto para dar cobertura: Sistema cancela o processo de fornecimento de cobertura a ser realizado por uma UP a uma outra UP aliada, e realiza a seleção do objeto clicado.

Inclusão:

Caso de uso 3.8: Patrulha.
Descrição: Este caso de uso descreve a funcionalidade de Patrulha feito por uma UD (opção disponível somente para uma UPS, UPG ou UV), onde a UD fica percorrendo o caminho A -> B -> A, definido pelo Usuário, indefinidamente.
Atores: Usuário, UP's com capacidade de realizar ataques.
Pré-condição: Usuário seleciona opção de Patrulha através de botão no menu das Unidades.
Seqüência de eventos:
1. Usuário clica com o botão direito no mouse sobre um local destino no mapa ou uma Unidade Construção destino.

2. Sistema faz com que as Unidades selecionadas se movam até a proximidade do local destino ou da Unidade Construção destino e voltem para origem, repetindo esta ação indefinidamente.

Pós-condição: Unidades selecionadas em modo de Patrulha.

Extensões:

1. Usuário seleciona com o botão esquerdo do mouse um local destino ou UC destino onde fará patrulha: Sistema cancela o processo Patrulha a ser realizado por uma UD, e realiza a seleção do objeto clicado quando o ponto clicado representa um objeto. Quando o ponto clicado não representa nenhum objeto, sistema simplesmente cancela o processo de realização de patrulha.

Inclusão:

Caso de uso 3.9: Carregamento da UPD com produto fornecido pela UPG.

Descrição: Este caso de uso descreve o processo de carregamento da Unidade do tipo personagem distribuidor (UPD) com produto, sendo a carga fornecida ou por uma unidade do tipo personagem gerente (UPG).

Atores: Usuário, UPD, UPG, produto.

Pré-condição: Para a UPD em questão, o usuário seleciona a opção de carregamento de carga.

Seqüência de eventos:

1. UPD se desloca em direção a UCP mais próxima e que esteja dentro do raio de ação permitido para o estabelecimento de uma UCD.
2. UPD coleta o produto da UPG, com o máximo que pode carregar.
3. UPD volta a sua UCD e retorna a sua atividade.

Pós-condição: Ao se selecionar a UPD que carregou a carga, a carga é exibida no menu de propriedades da UPD juntamente com a descrição de todas as suas características (tipo e quantidade) atualizadas.

Extensões:

4. Para a UPD em questão, não há nenhuma UCP com gerente dentro e que esteja dentro do raio de ação permitido para o estabelecimento de uma UCD: UPD volta a sua UCD e retorna à sua atividade, continuando com a mesma carga pessoal de produto anterior. Caso a carga pessoal de produto da UPD tenha acabado, a UPD aciona o caso de uso de carregamento automático com produto.
2. UPG não consegue carregar UPD completamente: UPD volta a sua UCD e retorna a sua atividade.

Inclusão:

Caso de uso 3.10: Estabelecimento de uma UCD para uma UPD.

Descrição: Este caso de uso descreve o estabelecimento de uma UCD para uma UPD, que fará a venda de produtos neste local.

Atores: Usuário, UPD, UCD.

Pré-condição: Para a UPD em questão, o usuário seleciona a opção de estabelecimento de UCD.

Seqüência de eventos:

1. Usuário clica com botão esquerdo do mouse em local onde deseja estabelecer uma UCD.
2. Caso UPD já possua uma UCD associada, sistema desassocia (apaga) a UCD existente.
3. UPD se desloca em direção ao novo local.
4. Sistema estabelece (cria) a nova UCD.
5. UPD inicia suas atividades de distribuição de produtos.

Pós-condição: UPD iniciando suas atividades na nova UCD.

Extensões:

1. Usuário clica em local inválido (objeto) (passo 1): Sistema cancela o processo de estabelecimento de nova UCD, e seleciona o objeto.

Inclusão:

4. Casos de Uso Automáticos correspondentes às Unidades

Caso de uso 4.1: Carregamento da Unidade do tipo personagem com produto.

Descrição: Este caso de uso descreve o processo de carregamento da UP com produto.

Atores: UP, UC e produto.

Pré-condição: Para a UP em questão, a carga pessoal de produto se esgota e UP não está em modo inativo.

Seqüência de eventos:

1. Se a UP for uma UPG ou UPS e estiver dentro de uma UC, UP se recarrega com a carga de produto dentro da UC.
2. Se a UP for uma UPG ou UPS e estiver fora de uma UC, UP se desloca em direção à UC mais próxima e que tenha carga e entra nela, realizando 1.
3. UP volta para a atividade anterior (se a atividade estava sendo exercida fora da UC, UP sai da UC e retorna ao ponto de atividade. Se não estava exercendo nenhuma atividade, volta para o ponto onde estava ociosa).
4. Se a UP for uma UPD e estiver fora de uma UC, UPD se desloca em direção a UCP mais próxima que tenha gerente dentro e que esteja dentro do raio de ação permitido para o estabelecimento de uma UCD.
5. UPD coleta o produto da UPG, com o máximo que pode carregar.
6. UPD volta a sua UCD e retorna a sua atividade.

Pós-condição: Ao se selecionar a Unidade que carregou a carga, a carga é exibida no menu de propriedades da Unidade juntamente com a descrição de todas as suas características (tipo e quantidade) atualizadas.

Extensões:

1. Durante o passo 1, a UC não pode carregar completamente a UP que está dentro dela (só pode ser UPG ou UPS): a UP em questão sai da UC e realiza o passo 2.
3. UPG não consegue carregar UPD completamente: UPD volta a sua UCD e retorna a sua atividade.

4. Para a UPD em questão, não há nenhuma UCP com gerente dentro e que esteja dentro do raio de ação permitido para o estabelecimento de uma UCD: UPD fica em modo inativo.

Inclusão:

Caso de uso 4.2: Ataque de uma UD do jogador(só não pode ser uma UPD) em modo ofensivo a uma UD inimiga.
Descrição: Este caso de uso descreve o processo de ataque da Unidade dinâmica do jogador(só não podendo ser uma UPD, já que esta não possui a função ataque nem a opção modo de ataque disponíveis) em modo ofensivo a uma UD inimiga. A Unidade dinâmica utilizará uma arma padrão.
Atores: Unidade dinâmica do jogador, UD inimiga.
Pré-condição: Para a UD do jogador em questão, a UD está no modo de ataque ofensivo e uma UD inimiga invade o circulo de atuação da UD do jogador.
Seqüência de eventos:

1. UD do jogador ataca a UD inimiga com a arma padrão.

Pós-condição: UD utilizando arma contra a UD inimiga ou NPC.
Extensões:

1. UD do jogador está desarmada: Sistema cancela o processo de ataque de uma UD do jogador(só não pode ser uma UPD) em modo ofensivo a uma UD inimiga.

Inclusão:

Caso de uso 4.3: Ataque de uma UD do jogador(só não pode ser uma UPD) em modo defensivo a uma UD inimiga.
Descrição: Este caso de uso descreve o processo de ataque da Unidade dinâmica do jogador(só não podendo ser uma UPD, já que esta não possui a função ataque nem a opção modo de ataque disponíveis) em modo defensivo a uma UD inimiga. A Unidade dinâmica utilizará uma arma padrão.
Atores: Unidade dinâmica do jogador, UD inimiga.
Pré-condição: Para a UD do jogador em questão, a UD está no modo de ataque defensivo e uma UD inimiga invade o circulo de atuação da UD do jogador.
Seqüência de eventos:

1. Se a UD do jogador for atacada pela UD inimiga, UD do jogador ataca a UD inimiga com a arma padrão.

Pós-condição: UD utilizando arma contra a UD inimiga ou NPC.
Extensões:

1. UD do jogador está desarmada: Sistema cancela o processo de ataque de uma UD do jogador(só não pode ser uma UPD) em modo defensivo a uma UD inimiga.

Inclusão:

Caso de uso 4.4: Perda de vida de uma Unidade Dinâmica.
Descrição: Este caso de uso descreve o processo de perda de vida (energia porcentual) de uma unidade dinâmica.
Atores: Unidade dinâmica do jogador, UD inimiga.
Pré-condição: Para a UD do jogador em questão, a UD recebe um ataque de uma UD inimiga.
Seqüência de eventos:

1. Sistema subtrai da energia porcentual da UD do jogador atacada um valor proporcional à potência da arma da UD inimiga.

Pós-condição: UD do jogador atacada com energia porcentual reduzida.
Extensões:

1. UD do jogador esgota sua energia porcentual (morte): Sistema destrói a UD do jogador atacada, e subtrai a arma e a carga armazenada pela UD do total existente para o jogador.

Inclusão:

Caso de uso 4.5: Ganho de Habilidade de uma Unidade Dinâmica.
Descrição: Este caso de uso descreve o processo de ganho de habilidade (mínimo 0 e máximo 10) de uma unidade dinâmica.
Atores: Unidade dinâmica do jogador e unidade dinâmica inimiga.
Pré-condição: Para a UD do jogador em questão, a UD não possui valor de 10 de habilidade como visível ao sistema (sendo muito bom como visível ao usuário) e sobrevive à uma batalha contra uma UD inimiga.
Seqüência de eventos:

1. Sistema adiciona um valor fixo ao valor de habilidade da UD do jogador que sobreviveu à uma batalha contra uma UD inimiga.

Pós-condição: UD do jogador que sobreviveu à uma batalha contra uma UD inimiga têm seu valor de habilidade aumentado.
Extensões:
Inclusão:

Caso de uso 4.6: Venda de produto para cliente transeunte.
Descrição: Este caso de uso descreve o processo de venda de produto para um cliente
transeunte.
Atores: NPC, UPD(ou UPG) e UCD(ou UCP).
Pré-condição: Sistema seleciona a UPD (ou UPG) de um jogador (ou oponente) para fazer a venda de produto. A seleção da UPD(ou UPG) deve respeitar os critérios que possibilitam a venda de produtos (não deverão ocorrer negociações próximas às batalhas ou quando uma unidade vendedora não possuir produtos para vender.)
Seqüência de eventos:

1. Sistema seleciona a UCD (ou UCP) associada à UPD(ou UPG) previamente selecionada para vender produto.
2. Sistema seleciona o NPC mais próximo à UCD (ou UCP) selecionada em (1).

3. Sistema desloca o NPC até a UCD(ou UCP) selecionada em (1).
4. Sistema faz a NPC ficar junto à UCD(ou UCP) selecionada em (1) durante 5 segundos.
5. Se a UCD(ou UCP) pertence ao jogador, sistema exibe o símbolo "$" em cima da UCD(ou UCP) durante 2 segundos.
6. Sistema volta a movimentar o NPC randomicamente.

Pós-condição: Sistema incrementa a quantidade de dinheiro do jogador(ou oponente).
Extensões:
Inclusão

Caso de uso 4.7: Venda de produto para cliente em veículo.
Descrição: Este caso de uso descreve o processo de venda de produto para um cliente
em veículo.
Atores: UV, UPD(ou UPG) e UCD(ou UCP).
Pré-condição: Sistema seleciona a UPD (ou UPG) de um jogador (ou oponente) para fazer a venda de produto. A seleção da UPD(ou UPG) deve respeitar os critérios que possibilitam a venda de produtos (não deverão ocorrer negociações próximas às batalhas ou quando uma unidade vendedora não possuir produtos para vender.)
Seqüência de eventos:

1. Sistema seleciona a UCD (ou UCP) associada à UPD(ou UPG) previamente selecionada para vender produto.
2. Sistema seleciona a UV (que possua NPC dentro) mais próximo à UCD (ou UCP) selecionada em (1).
3. Sistema desloca a UV até a UCD(ou UCP) selecionada em (1).
4. Sistema faz a UV ficar junto à UCD(ou UCP) selecionada em (1) durante 5 segundos.
5. Se a UCD(ou UCP) pertence ao jogador, sistema exibe o símbolo "$" em cima da UCD(ou UCP) durante 2 segundos.
6. Sistema desloca a UV para fora do mapa.

Pós-condição: Sistema incrementa a quantidade de dinheiro do jogador(ou oponente).
Extensões:
Inclusão

Caso de uso 4.8: Fuga de NPC de um tiroteio.
Descrição: Este caso de uso descreve o processo de fuga realizado por um NPC quando este se vê em meio a um tiroteio.
Atores: NPC.
Pré-condição: Sistema detecta redução de Vida de um NPC.
Seqüência de eventos:

1. Sistema seleciona o NPC que teve a redução de Vida detectada.
2. Sistema desloca o NPC na mesma direção para a qual ele estava caminhando até que o NPC esteja fora do raio de batalha (raio de batalha está ainda a se determinar).

Pós-condição: NPC continua movimento randômico pelo mapa.
Extensões:
Inclusão

Caso de uso 4.9: Inserção de morador.
Descrição: Este caso de uso descreve o processo inserção de um NPC.
Atores: NPC.
Pré-condição: Sistema detecta destruição de um NPC.
Seqüência de eventos:

1. Sistema aguarda 1 minuto, e depois cria um novo NPC.
2. Sistema insere o novo NPC em qualquer posição fora da área do mapa.

Pós-condição: Novo NPC se movimentando randomicamente pelo mapa.
Extensões:
Inclusão

Caso de uso 4.10: Transferência de posse de construção.
Descrição: Este caso de uso descreve o processo transferência de posse de uma UCP(ou UCS) de um jogador para um oponente e vice-versa.
Atores: UP do jogador e UP do oponente.
Pré-condição: Sistema detecta a invasão de uma UP adversária à uma UCP(ou UCS).
Seqüência de eventos:

1. Sistema aguarda 1 minuto.
2. Se não tiver ocorrido batalha, sistema disassocia a UCP(ou UCS) da UP que a possuía.
3. Sistema associa a UCP à UP adversária que conseguiu ocupá-la por mais de 1 minuto sem batalhas.

Pós-condição: UCP com Área de atuação da mesma cor que a UP que se apossou dela.
Extensões:

1. A UC em questão é uma UCS:Sistema dá fim de jogo. Se a UCS invadida pertence ao jogador, ele vence o jogo, caso contrário, ele perde o jogo.
2. Ocorre batalha antes de 1 minuto de invasão: Sistema encerra o caso de uso.

1 Casos de uso de IA

Além dos casos de uso citados acima, foram adicionados a partir da segunda iteração da fase de construção os casos de uso de inteligência artificial, que são explicados a seguir.

Caso de uso 5.1: Distribuição de recursos por parte da inteligência artificial.
Descrição: Este caso de uso descreve o processo de distribuição de recursos por parte do Gerenciador de Recursos da inteligência artificial.
Atores: Jogador computador e unidades dinâmicas do jogador computador.
Pré-condição:
Seqüência de eventos:

1. Sistema recolhe informações de jogo e de todos os jogadores e, através de fórmulas pré-determinadas, verifica qual a necessidade maior do jogador computador: atacar, defender ou contratar unidades para trabalhar.
2. Caso a necessidade maior seja atacar, o sistema deverá retirar uma unidade de um grupo de defesa e colocá-la num grupo de ataque (um container que possui unidades soldado . Se não, o sistema deverá tentar contratar uma unidade e coloca-la num grupo de ataque. Caso a necessidade maior seja defender, o sistema deverá retirar uma unidade de um grupo de ataque e coloca-la num grupo de defesa. Se não, o sistema deverá tentar contratar uma unidade e coloca-la num grupo de defesa. Caso a necessidade maior seja ganhar dinheiro, o sistema deverá tentar contratar uma unidade e coloca-la num grupo de trabalho.
3. Quando todas as unidades de um grupo estiverem posicionadas, o sistema deverá dar uma ordem de ataque para todas unidades do grupo atacarem a unidade inimiga mais próxima.
4. Quando a unidade alvo for destruída, o grupo deverá atacar a unidade mais próxima e assim por diante.

Pós-condição: Recursos distribuídos pelos containers de defesa, ataque e trabalho.
Extensões:
Inclusão:

Caso de uso 5.2: Ação de ataque por parte da inteligência artificial.
Descrição: Este caso de uso descreve o processo de ataque feito pelo jogador computador.
Atores: Jogador computador e unidades dinâmicas do jogador computador.
Pré-condição: Unidades dinâmicas de ataque fornecidas pelo Gerenciador de Recursos da IA em algum tipo de container.

Seqüência de eventos:

1. Sistema cria um grupo de unidades com tamanho máximo pré-determinado, as unidades restantes são colocados em outro grupo do mesmo tamanho e assim por diante.
2. Sistema deverá mover cada unidade de todos os grupos para a posição ao lado da primeira unidade de seu grupo.
3. Quando todas as unidades de um grupo estiverem posicionadas, o sistema deverá dar uma ordem de ataque para todas unidades do grupo atacarem a unidade inimiga mais próxima.
4. Quando a unidade alvo for destruída, o grupo deverá atacar a unidade mais próxima e assim por diante.

Pós-condição: Grupos de unidades atacando o inimigo.
Extensões:
Inclusão:

Caso de uso 5.3: Ação de trabalho por parte da inteligência artificial.
Descrição: Este caso de uso descreve o processo de colocar unidades para trabalhar, feito pelo jogador computador.
Atores: Jogador computador e unidades dinâmicas do jogador computador.
Pré-condição: Unidades dinâmicas distribuidoras fornecidas pelo Gerenciador de Recursos da IA em algum tipo de container.
Seqüência de eventos:

1. Para cada unidade do container, o sistema dá uma ordem de trabalho.

Pós-condição: Unidades distribuidoras trabalhando.
Extensões:
Inclusão:

Caso de uso 5.4: Ação de defesa por parte da inteligência artificial.
Descrição: Este caso de uso descreve o processo de defesa feito pelo jogador computador.
Atores: Jogador computador e unidades dinâmicas do jogador computador.
Pré-condição: Unidades dinâmicas de defesa fornecidas pelo Gerenciador de Recursos da IA em algum tipo de container.
Seqüência de eventos:

1. Sistema cria um grupo de unidades com tamanho máximo pré-determinado, as unidades restantes são colocados em outro grupo do mesmo tamanho e assim por diante.
2. Sistema deverá mover cada unidade de todos os grupos para a posição ao lado de uma unidade distribuidora. O grupo não deverá se mover para uma unidade que já possua outro grupo de defesa ao lado.
3. Quando todas as unidades de um grupo estiverem posicionadas, o sistema deverá refazer o passo 2 a uma taxa pré-determinada (baixa) para atualizar as posições caso a unidade distribuidora se mova.

4. Quando alguma unidade de defesa puder atacar uma unidade inimiga (estiver no raio de alcance do armamento da unidade), o sistema deverá dar uma ordem de ataque ao grupo todo para atacar essa unidade inimiga.

Pós-condição: Grupos de unidades ao lado da unidade distribuidora, defendendo-a.

Extensões:

Inclusão:

Apêndice C – Conteúdos do CD anexo

O CD que se encontra em anexo apresenta os seguintes diretórios:

Documentação – contém a cópia da monografia, bem como os artefatos gerados.

RTSProject – contém o código fonte do projeto, modelos, texturas, e scripts.