Departamento de Ciência da Computação

Instituto de Matemática e Estatística

Universidade de São Paulo

Desenvolvimento de um simulador de jogo de estratégia em tempo real

Autores:
Fabiano Aono
Leandro Aono

Orientador:
Prof. Dr. Paulo André Vechiatto de Miranda

2 de fevereiro de 2014

Sumário

I Parte técnica **1**

Introdução **3**

1 Preliminares **5**
1.1 Grafos 5
1.2 Estruturas de dados 6
1.2.1 Fila 6
1.2.2 Fila de prioridade 6

2 Jogos de estratégia em tempo real **7**
2.1 História e cenário competitivo 7
2.2 Representação do mundo 8
2.3 Recursos 9
2.4 Unidades 9
2.5 Edifícios 10
2.6 Névoa de guerra 11
2.7 Árvore tecnológica 11
2.8 Gerenciamento 13

3 Implementação **15**
3.1 Ferramentas utilizadas 15
3.2 Mapa 15
3.3 Interface do simulador 18
3.4 Unidades e edifícios 19
3.5 Sistema de recursos 19
3.6 Máquina de estado finita 19
3.7 Laço principal 21

4 Conceitos e técnicas **23**
4.1 Movimentação 23
4.1.1 Movimentação individual 23
4.1.2 Mapeamento de zona 28
4.2 Colisão 29

4.2.1 Detecção 29
4.2.2 Resposta 35
4.3 Inteligência Artificial 37
4.3.1 Análise do mapa 37
4.3.2 Sistema baseado em regras 39

II Parte subjetiva **41**

5 Fabiano **43**
5.1 Desafios e frustrações 43
5.2 Disciplinas relevantes 43
5.3 Próximos passos 44

6 Leandro **45**
6.1 Desafios e frustrações 45
6.2 Disciplinas relevantes 45
6.3 Próximos passos 46

7 Agradecimentos **47**

Lista de Figuras

2.1 Linha do tempo de jogos importantes do gênero 8
2.2 Tipos de grade 9
2.3 Classificação das entidades controladas pelo jogador 10
2.4 Névoa de guerra 11
2.5 Árvore tecnológica baseada em edifícios 12
2.6 Exemplos de caminhos tecnológicos 12

3.1 Distâncias entre vizinhos na grade quadrada e hexagonal 16
3.2 Representações do mapa 17
3.3 Movimentações válidas a partir de um *tile* 17
3.4 Representação do mapa em um grafo 18
3.5 Interface do simulador 18
3.6 Máquina de estados do simulador 20
3.7 Máquina de estados para a modelagem da IA 20

4.1 Comparação entre os algoritmos de busca 28
4.2 Mapeamento de zona 29
4.3 Dois modos de detecção de colisão 30
4.4 Distância entre os centros das unidades 30
4.5 Representação de um estado do mapa particionado em quadrantes 32
4.6 Quadtree - Sequência 1 33
4.7 Quadtree - Sequência 2 33
4.8 Quadtree - Sequência 3 34
4.9 Quadtree - Sequência 4 34
4.10 Resposta a colisão - Solução 1 - Unidade empurra a outra 35
4.11 Resposta a colisão - Solução 2 - Unidade dá a volta 36
4.12 Pontos de estrangulamentos do mapa 39

Parte I

Parte técnica

Introdução

Estratégia em tempo real (RTS: *Real-time Strategy*) é um subgênero de jogos eletrônicos de estratégia, em que o andamento do jogo ocorre em tempo real. Em um RTS, os jogadores controlam unidades e edifícios com o objetivo de eliminar as forças de seus oponentes. Outras tarefas características do gênero são: produção de novas unidades, construção de edifícios, coleta de recursos e pesquisa de tecnologias.

Esse gênero, diferentemente de um jogo de estratégia baseado em turnos, requer uma rapidez maior na tomada de decisões, deste modo, um jogador bem-sucedido deve possuir um alto nível de aprendizagem para se adaptar a um ambiente extremamente dinâmico.

A Inteligência Artificial (IA) dos jogos comerciais acaba não sendo tão bem desenvolvida por diversos motivos, entre eles, a alta complexidade de um RTS e a dificuldade em se conduzir experimentos de IA nestes jogos [23]. A alta complexidade está associada a natureza de um RTS, em que centenas de ações são executadas simultaneamente pelos jogadores, em um pequeno intervalo de tempo. Outro fator que aumenta a complexidade é que unidades e edifícios são criados e removidos durante uma partida.

Os jogos comerciais também não oferecem um bom suporte a realização de experimentos de IA [23], em razão disso, algumas soluções foram criadas como alternativa. Uma delas foi a BWAPI [2], que é uma aplicação de código aberto, utilizada no desenvolvimento de IAs do jogo Starcraft: Brood War (Blizzard Entertainment, 1998). Diversas competições de IA são realizadas anualmente, desde 2010, utilizando a aplicação [3].

Outra alternativa para a realização de experimentos de IA de um RTS é a criação de um simulador próprio. Esta solução fornece mais liberdade, uma vez que a IA não ficaria amarrada as particularidades de um jogo, sendo possível adaptar o simulador conforme as necessidades. O objetivo deste trabalho é o desenvolvimento de um simulador de RTS para o estudo e aplicações práticas de técnicas de IA.

Organização

A parte técnica da monografia está organizada como descrito abaixo. No capítulo 1 são apresentadas as definições e notações utilizados no trabalho. No capítulo 2 é exposta uma visão geral dos jogos de estratégia em tempo real, que inclui a história e alguns conceitos particulares do gênero. No capítulo 3 são detalhados os aspectos técnicos da implementação do simulador e no capítulo 4 os conceitos e técnicas estudados para a sua implementação.

Capítulo 1

Preliminares

Neste capítulo apresentaremos as definições e notações referentes à teoria dos grafos e estruturas de dados utilizadas na parte técnica.

1.1 Grafos

Esta seção foi baseada no livro *Introduction to Algorithms*, de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein [24].

Um **grafo direcionado** G é um par (V, E), onde V é um conjunto finito e E é uma relação binária de V. O conjunto V é chamado **conjunto de vértices** de G e seus elementos são chamados **vértices**. O conjunto E é chamado **conjunto de arestas** de G e seus elementos são chamados **arestas**. Sejam u, $v \in V$ e $u \neq v$, uma aresta de G é um par ordenado $(u, v) \in E$. A partir deste ponto chamaremos os grafos direcionados simplesmente de grafos.

Dado $G = (V, E)$ um grafo, denotamos por V(G) o conjunto de vértices de G e por E(G) o conjunto de arestas de G.

Se $(u, v) \in E$, então dizemos que v é **adjacente** a u. Dizemos também que v é **vizinho** de u.

Denotamos por $ADJ(v)$ o conjunto de vértices adjacentes ao vértice v.

Um **caminho** em G é uma sequência de vértices $P = (v_0, v_1, \ldots, v_k)$, onde $(v_{i-1}, v_i) \in E$ para $i = 1, 2, \ldots, k$. Um caminho é dito **simples** se os vértices do caminho são distintos.

O **comprimento** de um caminho P é a quantidade de arestas do caminho.

Se existe um caminho do vértice u ao v, então dizemos que v é **alcançável** a partir de u.

Um grafo $G = (V, E)$ **com custos** nas arestas possui uma função custo $c : E \rightarrow \mathbb{R}$.

Seja (u, v) uma aresta de G, dizemos que $c(u, v)$ é o **custo da aresta** (u, v). A aresta (v, u) é chamada de **anti-paralela** a (u, v).

Seja $P = (v_0, v_1, \ldots, v_k)$ um caminho no grafo G, o **custo do caminho** P é a soma dos custos de suas arestas, denotado por $c(P)$:

$$c(P) = \sum_{i=1}^{k} c(v_{i-1}, v_i)$$

Denotamos por $CAMINHOS(u,v)$ o conjunto de todos os caminhos possíveis do vértice u ao vértice v. Definimos o **custo do caminho mínimo** de u até v como:

$$\delta(u,v) = \begin{cases} min\{c(P) : P \in CAMINHOS(u,v)\} & \text{se } CAMINHOS(u,v) \neq \emptyset \\ \infty & \text{caso contrário} \end{cases}$$

Um **caminho mínimo** do vértice u ao vértice v é qualquer caminho P, em que $c(P) = \delta(u,v)$.

Um **corte** definido pelo conjunto de vértices S de um grafo $G = (V,E)$ é o conjunto de arestas que liga um vértice em S a algum vértice em $V - S$. Ele é denotado por $C(S)$ ou também por $(S, V - S)$:

$$C(S) = \{(u,v) \in E : u \in S, v \in V - S\}$$

1.2 Estruturas de dados

1.2.1 Fila

Fila é uma estrutura de dados que mantém um conjunto de elementos e que permite dois tipos de operação: inserção e remoção. Os elementos são removidos por ordem de chegada, deste modo, o elemento a ser removido é o que está a mais tempo na fila. Esta estratégia é conhecida como *First In First Out* (FIFO).

1.2.2 Fila de prioridade

Fila de prioridade é uma fila em que os elementos são associados a um dado adicional, chamado de prioridade. Ela estende as operações da fila comum da seguinte maneira:

- Os elementos são inseridos juntamente com uma prioridade;
- O elemento de maior prioridade é removido. Se dois elementos possuem a mesma prioridade, o elemento a mais tempo na fila é removido primeiro.

Capítulo 2

Jogos de estratégia em tempo real

Neste capítulo serão apresentados um breve resumo da história dos jogos de estratégia em tempo real e sua importância no cenário competitivo. Em seguida, serão apresentados alguns conceitos e terminologias deste gênero de jogo.

2.1 História e cenário competitivo

A história dos RTS não pode ser escrita sem falarmos do jogo Dune II: The Building of a Dynasty (Westwood Studios, 1992), que é considerado um dos pais do gênero. Ele revolucionou os jogos de estratégia [7], introduzindo elementos que são utilizados nos jogos atuais e ajudando a definir o gênero RTS. Alguns destes novos elementos foram: coleta de recursos para o treinamento de unidades, construção de edifícios, dependências entre construções (árvore tecnológica), utilização de diferentes facções com unidades únicas para cada uma delas e controle de unidades através do mouse [15].

Os jogos das franquias Warcraft e Starcraft (Blizzard Entertainment) tiveram grande contribuição na popularização do gênero RTS, vendendo milhões de cópias em todo o mundo. Em 2010, o jogo StarCraft II: Wings of Liberty vendeu 1.5 milhão de cópias em 48 horas após seu lançamento, tornando-se o jogo de estratégia que vendeu mais unidades rapidamente de todos os tempos [1].

Outros jogos importantes que merecem destaque são: Command & Conquer: Red Alert (Westwood Studios, 1996), Age of Empires (Ensemble Studios, 1997) e Total Annihilation (Cavedog Entertainment, 1997). No cenário nacional, podemos mencionar o jogo brasileiro Outlive (Continuum Entertainment, 2001), produzido por uma equipe de apenas seis pessoas. Ele introduziu algumas inovações em relação aos jogos RTS do período, como o custo de manutenção de unidades, que posteriormente também foi implementado pelo jogo Warcraft III: Reign of Chaos (Blizzard Entertainment, 2002) [13].

Na figura 2.1 são exibidos alguns jogos do gênero, em ordem cronológica:

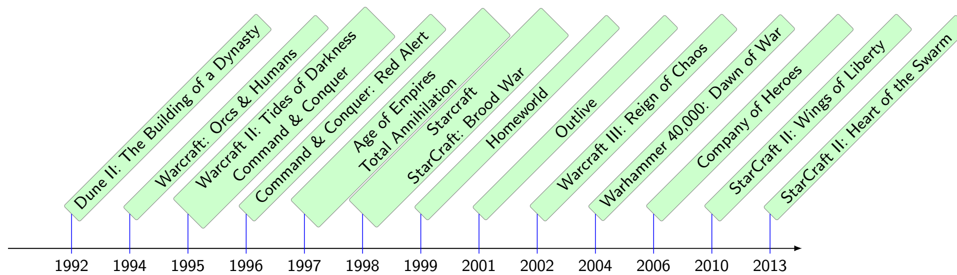


Figura 2.1: Linha do tempo de jogos importantes do gênero [16][1]

O termo *eSports (Electronic Sports)* é utilizado para jogos eletrônicos disputados em um cenário competitivo. Um dos gêneros mais importantes neste contexto é o RTS, juntamente com os gêneros de tiro em primeira pessoa (FPS: *First Person Shooter*), de luta e o *Multiplayer Online Battle Arena* (MOBA) [17].

O mercado dos *eSports* vem em uma grande crescente, movimentando milhões de dólares em premiações e patrocínios. Até outubro de 2013, foram distribuídos mais de U$15.000.000,00 em premiações, somente em competições de RTS. Neste quesito, a primeira colocação geral é ocupada por um RTS, o jogo Starcraft II distribuiu quantias que ultrapassaram U$8.500.000,00 [4].

O estabelecimento do mercado dos *eSports*, como conhecido atualmente, se deve principalmente a Coréia do Sul e ao jogo Starcraft: Brood War [6]. No país houve grande envolvimento do governo local, que em 2000, criou uma organização que cuida do gerenciamento dos *eSports*, denominada KeSPA (Korean e-Sports Association). Na época, o jogo que impulsionou tal mercado foi o Starcraft: Brood War, que vendeu no país 4,5 dos 9,5 milhões de cópias comercializadas mundialmente [14]. Devido a sua grande popularidade, as competições do jogo passaram a ser televisionadas em canais dedicados e os jogadores que mais se destacavam eram tratados como celebridades [12].

Este cenário altamente competitivo evidencia, com mais clareza, a importância do desenvolvimento de IAs de RTS, não somente para fins acadêmicos, mas também para fins comerciais (futuros jogos).

2.2 Representação do mundo

O ambiente no qual os jogadores disputam a partida é conhecido como o **mapa** do jogo. O mapa é formado por uma área limitada que pode ser representado de diversas maneiras. A mais simples delas é a representação em grade (*grid*), que se baseia na utilização de polígonos regulares de mesmo tamanho. Os polígonos mais conhecidos para esta representação são triângulos, quadrados e hexágonos [11]. A figura 2.2 ilustra grades formadas por estes polígonos.

[1]Fonte: elaborado pelo autor

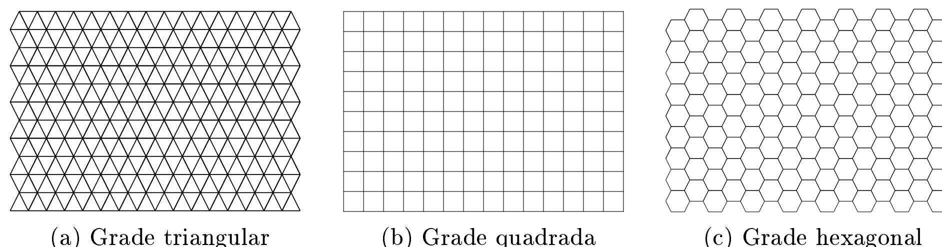


Figura 2.2: Tipos de grade[1]

2.3 Recursos

Os recursos são moedas virtuais do jogo que são associadas a cada jogador. A quantidade que o jogador possui é normalmente exibida em um contador, na tela de jogo. A maioria dos RTS possuem diversos tipos de recurso, proporcionando aos jogadores uma maior variedade de estratégias e, por consequência, resulta em um jogo mais interessante [10].

Durante uma partida, os jogadores podem executar ações que alteram suas quantidades de recursos, tanto positivamente, quanto negativamente. O acréscimo de recursos é chamado de **ganho de recursos** e o decréscimo de **gasto de recursos**. A taxa de ganho de recursos é definida como a **economia** do jogador.

O ganho de recursos pode ocorrer de duas maneiras diferentes [10]:

- **Coleta:** os recursos são coletados ativamente por unidades. Estes recursos coletáveis são distribuídos ao longo do mapa do jogo;
- **Produção:** os recursos são produzidos passivamente por edifícios.

O gasto de recursos pode ser feito de diversas maneiras, sendo os mais comuns, na produção de novas unidades, construção de novos edifícios e pesquisa de novas tecnologias.

2.4 Unidades

As **unidades** são entidades móveis controladas pelo jogador. Elas são produzidas em determinados edifícios do jogo. Cada tipo de unidade possui características particulares, como custo de produção, pontos de vida, pontos de ataque, tempo de produção e velocidade de movimentação.

Alguns RTS impõem um limite máximo de unidades que um jogador pode possuir em um determinado instante da partida. Nestes jogos o acréscimo do limite é feito normalmente através da construção de edifícios específicos.

As unidades podem ser classificados em dois grupos:

[1]Fonte: elaborado pelo autor

- **Unidades trabalhadoras:** são responsáveis pela coleta de recursos e construção de edifícios. Elas possuem uma capacidade reduzida de batalha em relação as unidades de combate;
- **Unidades de combate:** são responsáveis pelas ações relacionadas a batalha, sendo as mais comuns, as de ataque, defesa e reconhecimento. Geralmente um tipo de unidade de combate possui fraquezas que podem ser exploradas por outras, de modo que, não exista um tipo que se sobreponha a todas as outras.

2.5 Edifícios

Os **edifícios** são entidades imóveis controladas pelo jogador. Assim como as unidades, cada tipo de edifício possui características particulares.

Eles podem ser classificados nos seguintes grupos:

- **Suporte:** edifícios de suporte ao crescimento econômico e militar do jogador;
- **Defesa:** edifícios especializados na defesa, podendo atacar unidades inimigas;
- **Militar:** edifícios onde são produzidas unidades de combate;
- **Pesquisa:** edifícios onde são pesquisadas novas tecnologias.

A figura abaixo apresenta uma classificação das entidades controladas pelo jogador em um RTS.

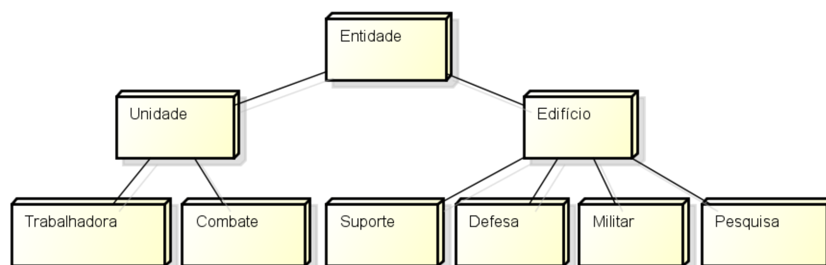


Figura 2.3: Classificação das entidades controladas pelo jogador[1]

[1] Fonte: elaborado pelo autor

2.6 Névoa de guerra

Névoa de guerra é uma mecânica que limita a visão do jogador a determinadas áreas do mapa. As áreas visíveis ao jogador são restritas pelo alcance da visão de suas unidades e edifícios. Ela adiciona um fator de incerteza a jogabilidade e aumenta o valor da informação. Quanto mais dados sobre o oponente um jogador possuir, mais bem embasadas serão suas decisões, que em tese, serão melhores que as decisões tomadas às cegas. A figura 2.4 ilustra esta mecânica.

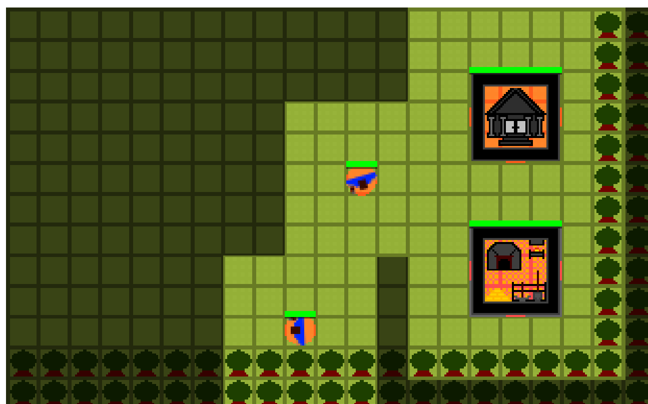


Figura 2.4: Névoa de guerra. A área visível pelo jogador é limitada por uma distância fixa de distância de suas entidades.[1]

2.7 Árvore tecnológica

A árvore tecnológica[2] é um conceito que especifica os pré-requisitos necessários para que o jogador tenha acesso a uma determinada tecnologia. As tecnologias podem ser entendidas como: novas unidades, novos edifícios ou quaisquer melhorias. Alguns exemplos comuns de melhorias em unidades são: aumento dos pontos de ataque, aumento dos pontos de defesa, aumento da velocidade e liberação de habilidades especiais.

[1]Fonte: elaborado pelo autor

[2]Jargão utilizado em RTS. Não se trata do conceito de árvore vista em teoria dos grafos.

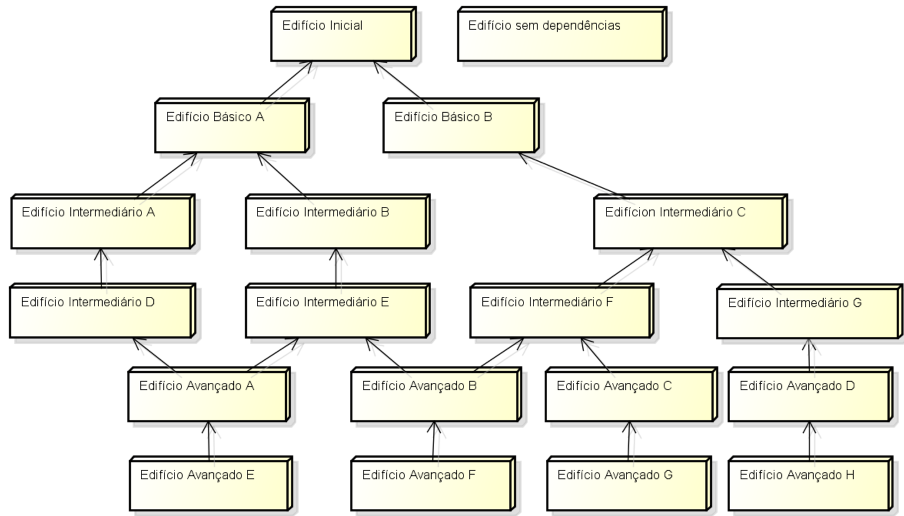
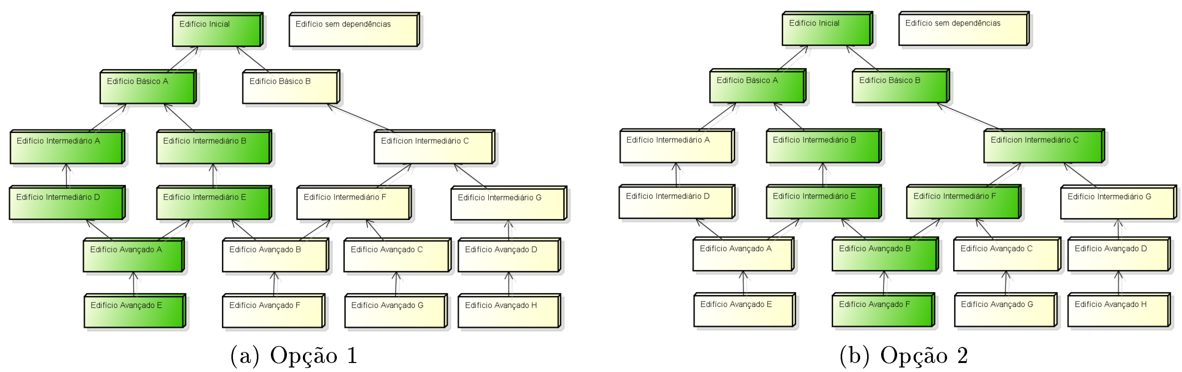


Figura 2.5: Árvore tecnológica baseada em edifícios. A seta representa uma relação de dependência entre os edifícios.[1]

A adição deste conceito proporciona um senso de evolução ao jogador, uma vez que novas tecnologias vão sendo disponibilizadas no decorrer da partida. Ela também cria um novo elemento estratégico, visto que o jogador deve planejar o caminho tecnológico que ele percorrerá, como exemplificado na figura 2.6.



(a) Opção 1 (b) Opção 2

Figura 2.6: Exemplos de caminhos tecnológicos. Os caminhos estão destacados em verde.[2]

[1]Fonte: elaborado pelo autor
[2]Fonte: elaborado pelo autor

2.8 Gerenciamento

Em um RTS, as ações dos jogadores estão associadas basicamente a dois tipos de gerenciamento:

- **Microgerenciamento (micro):** são ações relacionadas ao controle de suas unidades;
- **Macrogerenciamento (macro):** são ações relacionadas ao gasto de recursos.

Os jogadores devem desempenhar bem ambas as tarefas de gerenciamento, para que eles sejam bem sucedidos em um RTS. Uma forma conhecida de se mensurar o desempenho de um jogador é fazendo a contagem da quantidade de ações que ele consegue executar em um intervalo de um minuto. Esta unidade de medida é chamada de ações por minuto (APM).

O APM de um ser humano é limitado por sua agilidade, ou seja, o quão rápido ele consegue enviar comandos através do teclado e mouse. Deste modo, é compreensível que o jogador distribua seu esforço entre os dois tipos de gerenciamento. Neste aspecto, uma IA possui grande vantagem sobre um humano, visto que o APM da IA é limitado apenas pelo poder de processamento da máquina. Os melhores jogadores atingem cerca de 300 APM, enquanto que uma IA é capaz de atingir mais de 2.000 APM [5].

Capítulo 3

Implementação

Neste capítulo serão listadas as ferramentas utilizadas no desenvolvimento e descritos detalhes da implementação do simulador. Estas informações serão utilizadas como base para um melhor entendimento dos capítulos seguintes.

3.1 Ferramentas utilizadas

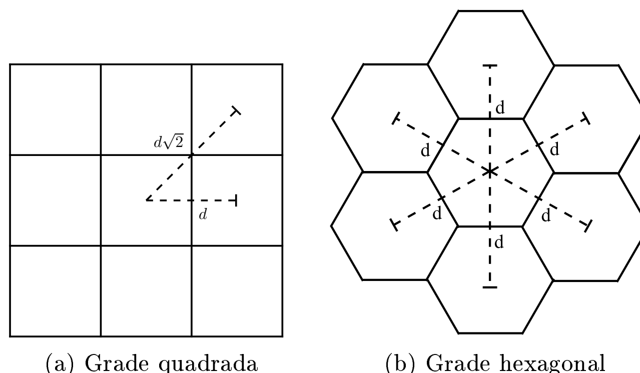
C++ é uma linguagem compilada orientada a objetos situando-se num nível intermediário de programação, por possuir características de linguagens de baixo e alto nível de abstração.

Allegro é uma biblioteca de código aberto que implementa algumas rotinas básicas de baixo nível, normalmente requeridas no desenvolvimento de jogos em C e C++. As rotinas tratam da criação de janelas, captura de entrada, reprodução de sons, entre outros. Ela é uma biblioteca multiplataforma que oferece suporte a Linux, Windows e MacOS X.

CodeBlocks é um ambiente integrado de desenvolvimento (IDE: *Integrated Development Environment*), de código aberto, desenvolvido em C++ que oferece suporte à múltiplos compiladores e plataformas. Ela utiliza uma arquitetura baseada em plugins, deste modo, ela é uma IDE de característica extensível.

3.2 Mapa

Uma das primeiras decisões de implementação foi a escolha da representação a ser utilizada para o mapa. Como a grade triangular não é tão utilizada em jogos, ela foi descartada. A escolha ficou entre duas candidatas: a grade quadrada e a hexagonal. A principal vantagem da grade hexagonal é que as células vizinhas são equidistantes. Esta característica não é encontrada na grade quadrada, já que neste tipo de grade, as distâncias para os vizinhos variam. A figura 3.1 ilustra esta propriedade.



(a) Grade quadrada (b) Grade hexagonal

Figura 3.1: Distâncias entre vizinhos na grade quadrada e hexagonal[1]

Por simplicidade, foi decidido pelo uso da grade quadrada, já que ela possui um sistema de coordenadas simples, que é facilmente mapeável em uma matriz. Cada posição desta matriz faz referência a um tipo de terreno, que por sua vez é associado a uma imagem chamada de *tile*.

Tiles são pequenas imagens, que ao serem colocadas lado a lado com outros *tiles*, formam uma imagem maior [18], neste caso, o mapa de jogo. O tamanho dos tiles foi fixado em 16x16 pixels.

A classe que armazena as propriedades dos *tiles* foi estendida para que ela guarde também, outras propriedades apresentadas a seguir:

- **Caractere:** o caractere que representa o *tile*. Utilizado na leitura do mapa a partir de um arquivo texto;
- **Andável:** identifica se as unidades podem caminhar neste *tile*;
- **Tipo:** o identificador do tipo de *tile*. Utilizado para determinar se é possível construir determinados edifícios neste *tile*;
- **Imagem:** a imagem utilizada para a representação do tile.

A tabela abaixo apresenta os *tiles* existentes no simulador, juntamente com suas propriedades:

Tile	Caractere	Andável	Tipo	Imagem
Árvore	t	não	0	
Grama	.	sim	1	
Ouro	g	sim	2	

O processo de transformação das representações do mapa, que é iniciado na leitura do mapa de um arquivo texto, até sua renderização, é ilustrado na figura 3.2.

[1]Fonte: elaborado pelo autor

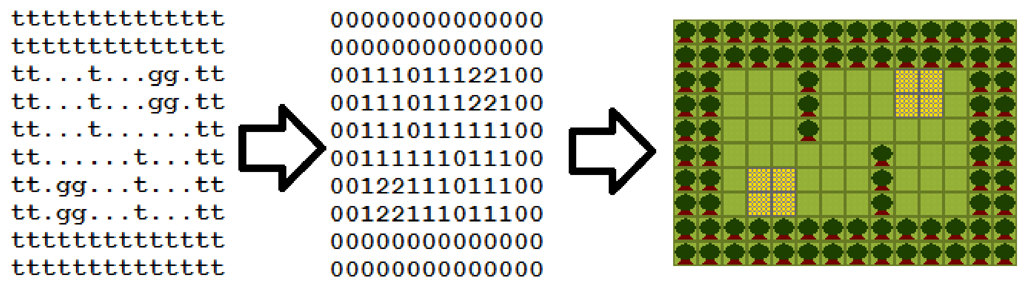


Figura 3.2: Representações do mapa. O mapa inicialmente é armazenado em um arquivo texto utilizando os caracteres. Em seguida, os tipos são armazenados em uma matriz, para que seja renderizado no final do processo.[1]

O simulador apresenta uma vista de cima para baixo do mapa. A terceira figura de 3.2 ilustra um trecho de mapa composto por três tipos de *tiles*, utilizando esta perspectiva.

A movimentação das unidades nessa representação pode ser realizada em oito direções diferentes, correspondentes aos oito *tiles* vizinhos (dois na horizontal, dois na vertical e quatro nas diagonais), desde que, eles sejam *tiles* andáveis.

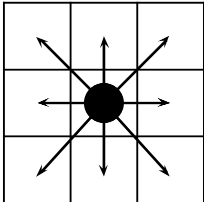


Figura 3.3: Movimentações válidas a partir de um *tile* [2]

O mapa também foi representado por um grafo com custos nas arestas. A utilização deste grafo é importante para resoluções de problemas relacionados a busca de caminho que serão apresentados posteriormente.

Cada *tile* do mapa é representado por um vértice do grafo, o qual possui arestas que ligam aos vértices que representam os *tiles* vizinhos. O custo de cada aresta é definido pela distância euclidiana entre os centros de cada *tile*.

A figura 3.4, demonstra um trecho de mapa 3x3, com seus respectivos vértices, arestas e pesos.

[1]Fonte: elaborado pelo autor
[2]Fonte: elaborado pelo autor

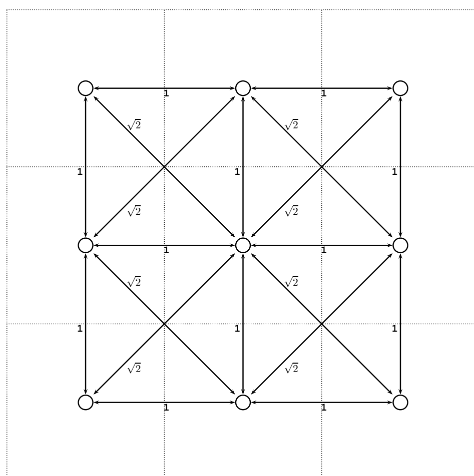


Figura 3.4: Representação do mapa em um grafo [1]

3.3 Interface do simulador

A interface do simulador é composta por três áreas, como ilustrado na figura 3.5.

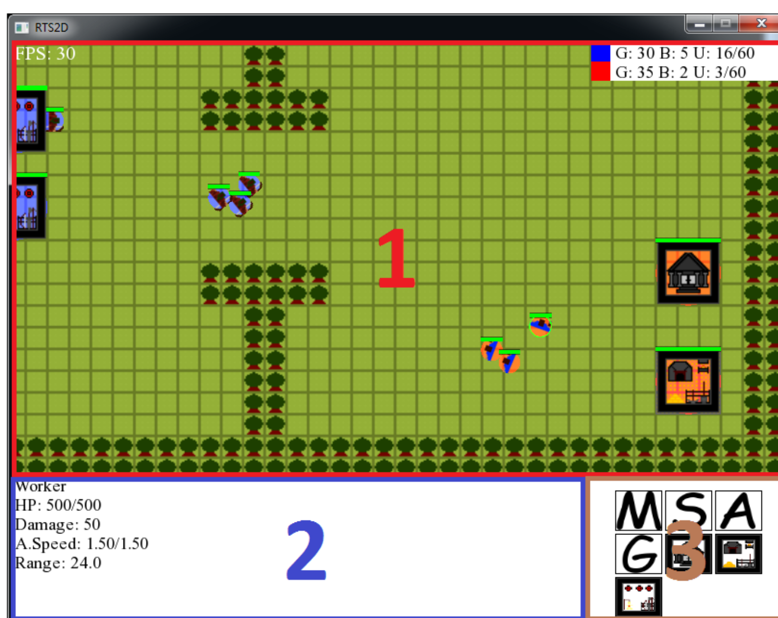


Figura 3.5: Interface do simulador. 1.Mapa 2.Informações sobre as entidades selecionadas 3.Comandos disponíveis para as entidades selecionadas [2]

[1]Fonte: elaborado pelo autor

3.4 Unidades e edifícios

No simulador foi implementada uma única facção, com três tipos de unidades e três tipos de edifícios.

As unidades implementadas são:

- **Trabalhador (Worker):** utilizada na construção de edifícios e na coleta de recursos;
- **Guerreiro (Warrior):** unidade básica de combate corpo a corpo;
- **Arqueiro (Archer):** unidade básica de combate a distância.

Os edifícios implementados são:

- **Edifício principal (Hall):** edifício onde são treinados os trabalhadores;
- **Mina de ouro (Gold Mine):** edifício que serve para a coleta de recursos. Ele pode ser construído somente sobre *tiles* do tipo ouro. Mais detalhes relativos a coleta de recursos são apresentados na seção 3.5;
- **Quartel (Barracks):** edifício onde são treinados os **guerreiros** e **arqueiros**.

3.5 Sistema de recursos

Apenas um tipo de recurso foi implementado, o ouro. Alguns *tiles* do tipo ouro são distribuídos ao longo do mapa e eles podem ser coletados em duas etapas:

1. O edifício **mina de ouro** deve ser construído em posições ocupadas por *tiles* do tipo ouro.
2. **Trabalhadores** devem ser enviadas a **mina de ouro** construída, para que o ganho de recursos comece a ser contabilizado. A taxa de ouro ganha pelo jogador é linearmente proporcional a quantidade de trabalhadores que estejam minerando.

3.6 Máquina de estado finita

Máquina de estado finita (FSM: *Finite State Machine*) é um modelo computacional utilizado na representação de programas de computador. Uma FSM é definida por uma lista de estados e por transições que ligam estes estados. Estas transições são ativadas por condições definidas no modelo.

No simulador foi implementada uma FSM para o tratamento do laço principal do programa. A cada iteração no laço, são verificadas as condições das transições válidas para o estado atual. Os estados existentes são:

- **PauseState:** estado em que a partida no simulador está parada.
- **PlayingState:** estado em que a partida no simulador está em andamento.

[2]Fonte: elaborado pelo autor

- **ResultState:** estado em que a partida no simulador foi finalizada e um jogador é declarado vencedor. Um jogador é considerado vencedor quando seus oponentes não possuírem nenhum edifício em jogo.

O diagrama que representa a FSM do laço principal é representado abaixo:

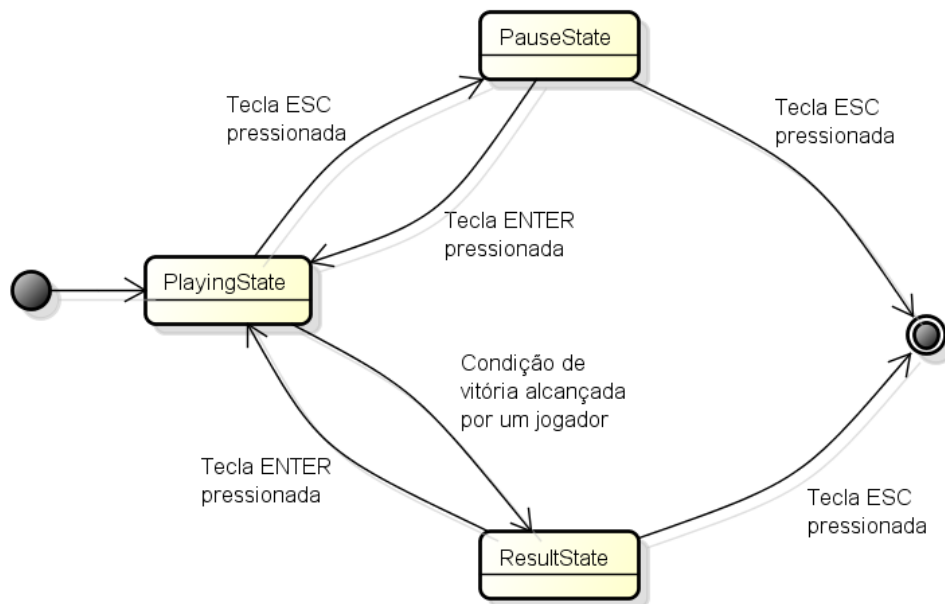
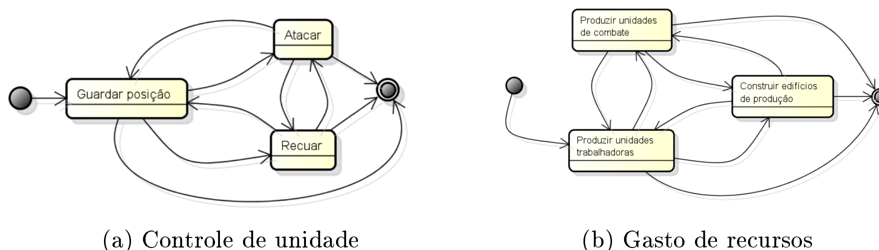


Figura 3.6: Máquina de estados do simulador [1]

Uma IA do simulador pode se basear em uma FSM, tanto em níveis micro (controle de unidades) quanto em níveis macro (gerenciamento da economia). A figura a 3.7 ilustra dois esboços de FSMs de componentes de uma IA.



(a) Controle de unidade

(b) Gasto de recursos

Figura 3.7: Máquina de estados para a modelagem da IA[2]

[1]Fonte: elaborado pelo autor
[2]Fonte: elaborado pelo autor

3.7 Laço principal

Quando o estado atual do simulador é **PlayingState**, cada iteração no laço principal do simulador incrementa a simulação da partida em um passo. O laço é apresentado abaixo:

```
while(sm->isRunning()) {
    sm->readInput();
    sm->update();
    sm->render();
}
```

Nele são executadas as seguintes rotinas:

1. *readInput()*: são capturadas as entradas do usuário;
2. *update()*: a partida é atualizada em um passo, ele consiste nas tarefas a seguir:
   (a) Atualização da movimentação das unidades;
   (b) Tratamento das colisão;
   (c) Atualização dos ataques;
   (d) Remoção das entidades com pontos de vida zerados;
   (e) Atualização do treinamento das unidades nos edifícios;
   (f) Atualização dos edifícios em construção;
   (g) Atualização dos recursos coletados.
3. *render()*: são renderizados o mapa, as unidades, os edifícios e a interface do usuário.

Capítulo 4

Conceitos e técnicas

Neste capítulo serão apresentadas os principais conceitos e técnicas estudados durante o desenvolvimento do simulador de RTS. Os principais problemas encontrados tratam da movimentação e do tratamento da colisão das unidades. Será apresentada também uma IA criada no simulador juntamente com as técnicas utilizadas neste processo.

4.1 Movimentação

A movimentação das unidades é um dos processos críticos de um RTS, já que ela influi diretamente na jogabilidade. Além disso, como este processo ocorre o tempo todo em uma partida, é imprescindível que ela seja funcional e eficiente. As unidades devem percorrer caminhos satisfatórios e as buscas destes caminhos não podem consumir muito processamento da máquina.

A movimentação individual de unidades é um problema de resolução simples, em que devemos encontrar um caminho satisfatório entre a origem e o destino. Na movimentação em grupo, outros fatores devem ser levados em consideração, como por exemplo, se desejamos que a formação inicial seja mantida durante a movimentação. No simulador implementado, adotamos uma solução simplificada, onde cada unidade é tratada de forma individual.

4.1.1 Movimentação individual

Nesta seção serão apresentadas as técnicas relacionadas a movimentação individual de unidades. Para a resolução destes problemas foram estudados diversos algoritmos em grafos, que serão apresentados abaixo.

Sejam $G = (V, E)$ um grafo com custos nas arestas e a função de custo $c : E \rightarrow \mathbb{R}$ tal que $c(u, v) > 0, \forall u, v \in V$. A busca de caminho consiste em encontrar os vértices do grafo pelos quais uma unidade no vértice s deve passar para chegar à um vértice t.

Em [27] são listados quatro critérios que os algoritmos de busca de caminho devem atender:

1. Se o caminho existe, o algoritmo deve devolvê-lo;

2. Se múltiplos caminhos existem, o algoritmo deve devolver o melhor caminho, segundo algum critério apropriado;
3. Se nenhum caminho existe, a execução do algoritmo deve terminar e devolver tal informação;
4. O algoritmo deve ser eficiente.

Algoritmo de busca em largura

O primeiro algoritmo estudado foi o de busca em largura (BFS: *Breadth-first search*). Ele explora os vértices ordenados pelo comprimento do caminho à origem e utiliza uma estrutura de dados de fila para o controle dos vértices a serem explorados.

Algoritmo 1 Algoritmo de busca em largura

Entrada: G, s, t
1: inicializa o vetor de pais *pai* com NULL
2: $pai[s] \leftarrow s$
3: insira s na fila Q
4: **enquanto** Q não está vazia **faça**
5:     remova *atual* de Q
6:     **para todos** vértices $adj \in ADJ(atual)$ **faça**
7:         **se** $pai[adj] =$ NULL **então**
8:             $pai[adj] \leftarrow atual$
9:             **se** $adj = t$ **então**
10:                 **devolva** $MontaCaminho(pai, s, t)$
11:             insira *adj* em Q
12: **devolva** vazio

O algoritmo devolve o caminho, se ele existir. Na execução do algoritmo, cada vértice possui armazenado a informação do vértice predecessor. O caminho encontrado pode ser obtido, percorrendo-se a lista de predecessores a partir do vértice t, como apresentado no algoritmo 2. Este algoritmo também é utilizado pelos algoritmos de Dijkstra e A* apresentados a seguir.

Algoritmo 2 Algoritmo de extração do caminho (MontaCaminho)

Entrada: pai, s, t
Saída: P
1: $atual \leftarrow t$
2: **enquanto** $atual \neq s$ **faça**
3:     insira *atual* na lista P
4:     $atual \leftarrow pai[atual]$
5: **devolva** Inverte(P)

O algoritmo de busca em largura encontra o caminho de menor comprimento. No entanto este critério não leva em consideração os pesos das arestas, o que requer outros algoritmos mais sofisticados com melhores critérios que serão apresentados a seguir.

Algoritmo de Dijkstra

O algoritmo de Dijkstra é uma variação do algoritmo de busca em largura. Originalmente ele foi projetado para resolver o problema de caminho de custo mínimo em grafos.

O algoritmo utiliza uma fila de prioridades, para determinar o próximo vértice a ser explorado. Os vértices com maior prioridade são os que possuem menor custo calculado em relação ao vértice origem. O valor do custo calculado será representado pelo vetor $dist$.

Algoritmo 3 Algoritmo de Dijkstra

Entrada: G, s, t

1: inicializa o vetor de custos à s $dist$ com ∞
2: inicializa o vetor de pais pai com NULL
3: $pai[s] \leftarrow s$
4: $dist[s] \leftarrow 0$
5: insira s na fila de prioridades PQ
6: **enquanto** PQ não está vazia **faça**
7:     remova $atual$ de PQ
8:     **se** $atual = t$ **então**
9:         **devolva** $MontaCaminho(pai, s, t)$
10:     **para todos** vértices $adj \in ADJ(atual)$ **faça**
11:         $custo \leftarrow$ c($atual$, adj)
12:         **se** $pai[adj]$!= NULL **então**
13:             $pai[adj] \leftarrow atual$
14:             $dist[adj] \leftarrow dist[atual] + custo$
15:             adicione adj em PQ
16:         **senão**
17:             **se** $custo + dist[atual] < dist[adj]$ **então**
18:                 $dist[adj] \leftarrow custo + dist[atual]$
19:                 $pai[adj] \leftarrow atual$
20: **devolva** vazio

Algoritmo A*

O A* (A-estrela) é um algoritmo de busca de caminho muito semelhante ao algoritmo de Dijkstra. A diferença é que ele utiliza o valor de custo calculado acrescido à uma função de estimativa na fila de prioridades, representado pelo vetor $custoTotal$.

Seja $v \in V$ um vértice alcançável a partir de s. O custoTotal de v é definido por:

$$custoTotal[v] = dist[v] + h(v, t)$$

onde $h(v, t)$ é uma função de estimativa de custo entre os vértices v e t. Esta função também é conhecida como função heurística.

A função heurística utilizada é dada pela distância euclideana entre os tiles representados pelos vértices v e t:

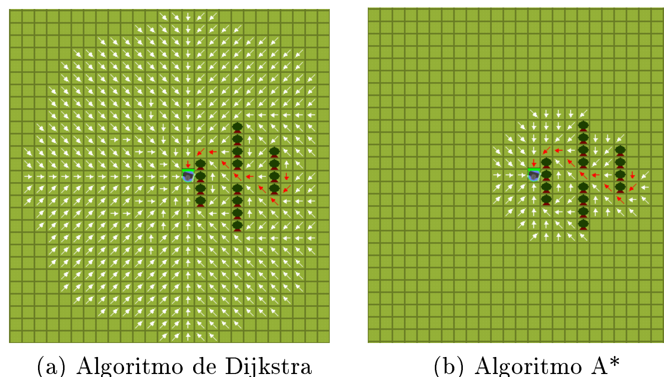
$$h(v, t) = \sqrt{(v_x - t_x)^2 + (v_y - t_y)^2}$$

Algoritmo 4 Algoritmo A*

Entrada: G, s, t

1: inicializa o vetor de custos à s $dist$ com ∞
2: inicializa o vetor de pais pai com NULL
3: $pai[s] \leftarrow s$
4: $dist[s] \leftarrow 0$
5: insira s na fila de prioridades PQ
6: **enquanto** PQ não está vazia **faça**
7:     remova $atual$ de PQ
8:     **se** $atual = t$ **então**
9:         **devolva** $MontaCaminho(pai, s, t)$
10:     **para todos** vértices $adj \in ADJ(atual)$ **faça**
11:         $custo \leftarrow$ c($atual$, adj)
12:         **se** $pai[adj]$!= NULL **então**
13:             $pai[adj] \leftarrow atual$
14:             $dist[adj] \leftarrow dist[atual] + custo$
15:             $custoTotal[adj] \leftarrow dist[adj] + h(adj, t)$
16:             adicione adj em PQ
17:         **senão**
18:             **se** $custo + dist[atual] < dist[adj]$ **então**
19:                 $dist[adj] \leftarrow custo + dist[atual]$
20:                 $custoTotal[adj] \leftarrow dist[adj] + h(adj, t)$
21:                 $pai[adj] \leftarrow atual$
22: **devolva** vazio

Devido a utilização da função heurística, o algoritmo A* explora uma quantidade de nós menor em relação ao algoritmo de Dijkstra. A figura 4.1 ilustra uma comparação entre os dois algoritmos, realizada no simulador.



(a) Algoritmo de Dijkstra (b) Algoritmo A*

Figura 4.1: Comparação entre os algoritmos de busca. Os caminhos encontrados pelos algoritmos estão destacados em vermelho, já os nós explorados estão em branco.[1]

O algoritmo A* foi o selecionado no tratamento da movimentação individual das unidades do simulador, pois satisfaz os critérios estabelecidos por [27].

4.1.2 Mapeamento de zona

Um dos problemas dos algoritmos de busca de caminho, ocorre no caso de não existir nenhum caminho entre o vértice origem e o vértice destino. Os algoritmos apresentados acabam por visitar todos os vértices alcançáveis, até que eles detectem que não há caminhos, ou seja, grande parte do processamento utilizado seria desperdiçado. Esta perda é diretamente proporcional ao tamanho do mapa. Uma das maneiras de se evitar que essa busca seja feita é utilizando uma técnica chamada mapeamento de zona [27].

O mapeamento de zona consiste na rotulação de vértices do grafo que são adjacentes dois a dois com valores iguais.

A técnica consiste na rotulação de cada vértice do grafo com um valor inteiro, chamado de **valor de zona**.

Seja v um vértice do grafo, denotamos por $ZV(v)$ o valor de zona de v. O algoritmo percorre todos os vértices do grafo que não estejam rotulados e realiza as seguintes operações.

Caso o vértice represente um *tile* não andável, ele é rotulado com um valor especial 0. Caso contrário, um novo rótulo é criado (o valor é incrementado), e todos os vértices alcançáveis a partir dele são rotulados com o mesmo valor.

Concluído o mapeamento, os algoritmos de busca serão chamados somente se o vértice origem s e vértice destino t possuam o mesmo valor de zona, ou $ZV(s) = ZV(t)$.

A figura 4.2 ilustra os valores de zona dos vértices de um mapa, calculados pelo simulador.

[1]Fonte: elaborado pelo autor

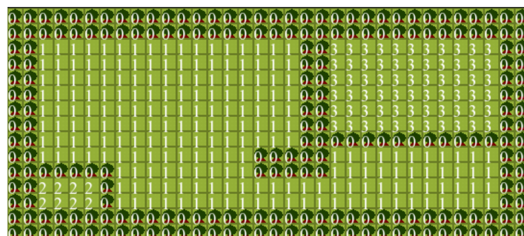


Figura 4.2: Mapeamento de zona [1]

4.2 Colisão

Em jogos eletrônicos, a colisão é um evento em que dois objetos ocupam o mesmo espaço em um determinado instante. Em alguns jogos a colisão é um evento aceitável e que pode ser ignorado. Não é o caso de um RTS, em que é interessante que a colisão seja tratada, já que o posicionamento das unidades é considerado um fator estratégico.

Como o tratamento da colisão é um processo realizado a cada passo do jogo, ele deve ser eficiente. O tratamento é realizado em duas etapas: Detecção e Resposta.

4.2.1 Detecção

A detecção da colisão pode ser feita de dois modos, *a priori* ou *a posteriori* [19].

Na detecção *a priori*, as colisões são detectadas antes que elas ocorram. O algoritmo calcula as posições de cada objeto para um determinado intervalo de tempo futuro, utilizando-se das informações disponíveis, como suas trajetórias e velocidades. Caso uma colisão seja identificada no intervalo de tempo, é realizado o devido tratamento para que a colisão deixe de ocorrer. Por exemplo, as unidades em iminência de colisão poderiam ter suas trajetórias e velocidades alteradas.

Na detecção *a posteriori*, as colisões são detectadas depois que elas ocorrem. A cada passo do jogo, o algoritmo verifica os objetos que estão colidindo e gera uma lista destes objetos. Em seguida, os objetos obtidos são devidamente tratados na etapa de resposta a colisão. A detecção *a posteriori* foi implementada no simulador por ser mais simples, apesar de ser menos confiável, já que algumas colisões podem não ser detectadas em certas situações.

A detecção de colisão entre dois objetos é realizada verificando se eles se intersectam, ou seja, se pelo menos um pixel de um objeto se sobrepõe aos de outro objeto. A verificação pixel por pixel é muito complexa e custosa, podendo prejudicar o desempenho do jogo.

Uma forma de simplificar a detecção de colisões é utilizar formas geométricas conhecidas como uma aproximação dos corpos dos objetos. Deste modo, a verificação passa a ser feita entre figuras geométricas, que pode ser realizado rapidamente através de fórmulas conhecidas. A figura mais simples para esta aproximação e que foi escolhida na implementação do simulador, foi o círculo.

A figura 4.3 ilustra os dois modos de detecções de colisão: pixel por pixel e a aproximada por círculos.

(a) Pixel por pixel (b) Aproximado por círculos

Figura 4.3: Dois modos de detecção de colisão. A colisão ocorrida está em destaque na figura.[1]

A detecção de colisão entre dois círculos é feita considerando a distância entre seus centros e a soma de seus raios.

Sejam u_1 e u_2 duas unidades do jogo tais que:

- o centro de u_1 esteja na coordenada (x_1,y_1) e possui raio r_1
- o centro de u_2 esteja na coordenada (x_2,y_2) e possui raio r_2

Podemos calcular a distância euclidiana entre os centros pela fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Podemos dizer que u_1 e u_2 estão colidindo se $d < r_1 + r_2$.

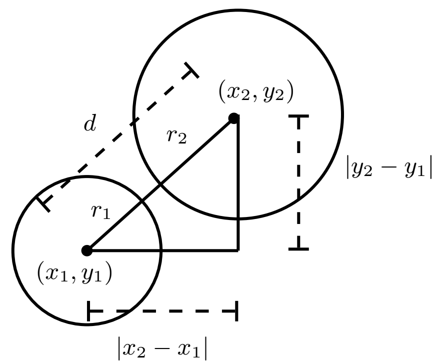


Figura 4.4: Distância entre os centros das unidades [2]

A maneira mais simples de detectar todas as colisões que estão ocorrendo é realizando comparações dois a dois entre as unidades do jogo. A complexidade algorítmica dessa

[1]Fonte: elaborado pelo autor
[2]Fonte: elaborado pelo autor

forma de detecção é quadrática em relação ao número de unidades dentro do jogo, o que pode ocasionar uma queda considerável de desempenho, tornando o jogo desagradável.

Um modo de diminuir a quantidade de comparações é limitando o espaço de busca para unidades que estejam próximas. Isto pode ser feito através do particionamento do mapa, utilizando uma estrutura de dados chamada Quadtree.

Quadtree

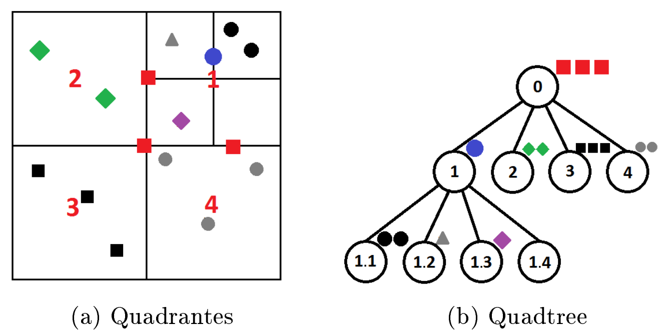
Quadtree é uma estrutura de dados que herda as características topológicas de uma árvore. Cada nó interno da Quadtree possui exatamente quatro filhos.

A propriedade abaixo vale para a Quadtree:

Propriedade 1: Cada nó possui uma lista de unidades pertencentes estritamente ao quadrante que ele representa.

Inicialmente, cria-se um nó raiz, correspondente ao quadrante do mapa inteiro. Após um certo número de inserções de unidades à aquele quadrante, divide-se uniformemente o quadrante pai em quatro quadrantes filhos. Após isso, as unidades são distribuídas entre os filhos respeitando a propriedade (1), caso a unidade esteja na intersecção de dois quadrantes, essa unidade é mantida na lista do pai. O número máximo de divisões feitas é definida pelo desenvolvedor.

A figura 4.5 ilustra um exemplo de particionamento juntamente com a árvore que representa esta configuração.



(a) Quadrantes (b) Quadtree

Figura 4.5: Representação de um estado do mapa particionado em quadrantes[1]

Para detecção de colisão, compara-se apenas unidades que estão no conjunto de todas as listas dos nós abaixo:

1. Nó atual da unidade;
2. Nós pais até a raiz da árvore;
3. Nós filhos;

No exemplo apresentado na figura 4.5, caso a unidade a qual queremos verificar as colisões pertença ao nó **1**, devemos fazer a comparação com as unidades dos nós **1** (nó atual), **0** (nó pai até a raiz), **1.1** ,**1.2**, **1.3** e **1.4** (nós filhos). Caso fosse uma unidade do nó **1.1**, as comparações seriam feitas com as unidades dos nós **1.1** (nó atual), **1** e **0** (nós pais até a raiz).

A sequência de figuras abaixo ilustra os quadrantes criados, na medida em que as unidades são inseridas no mapa do simulador. No exemplo foi estabelecido o limite máximo de três unidades por quadrante, antes que ele seja subdividido.

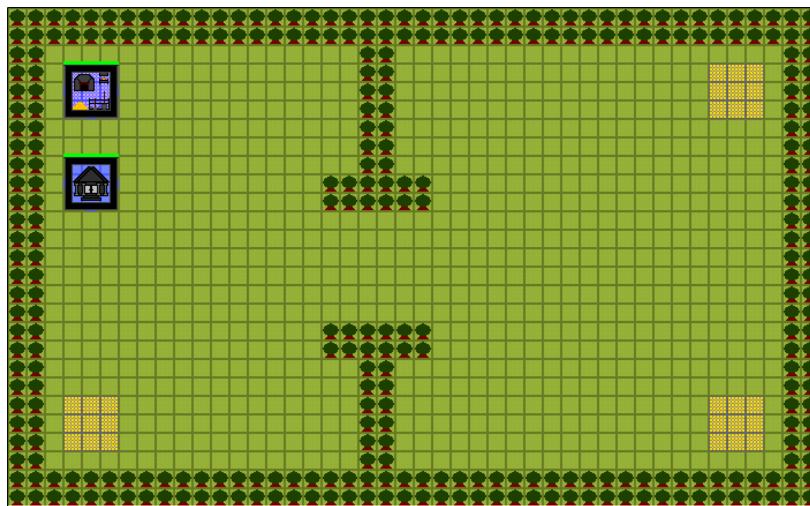


Figura 4.6: Quadtree - Sequência 1 - Nenhuma unidade no mapa, o mapa inteiro é o quadrante pai. [1]

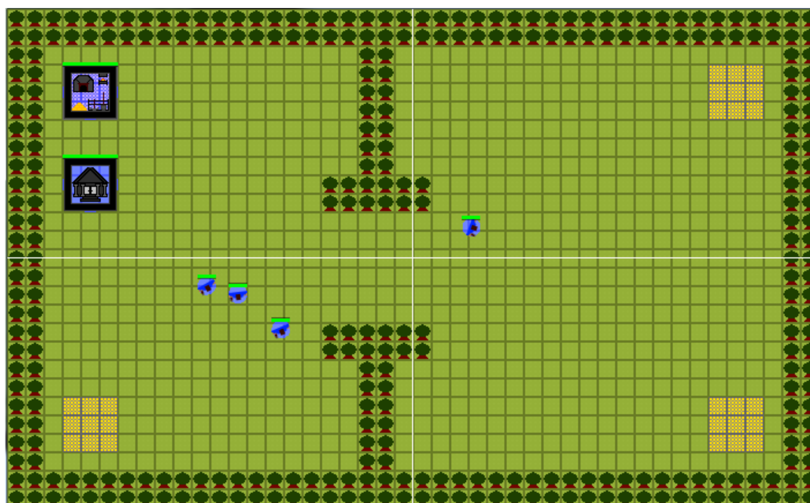


Figura 4.7: Quadtree - Sequência 2 - Com quatro unidades criadas, o quadrante pai é particionado. [2]

[1]Fonte: elaborado pelo autor
[2]Fonte: elaborado pelo autor

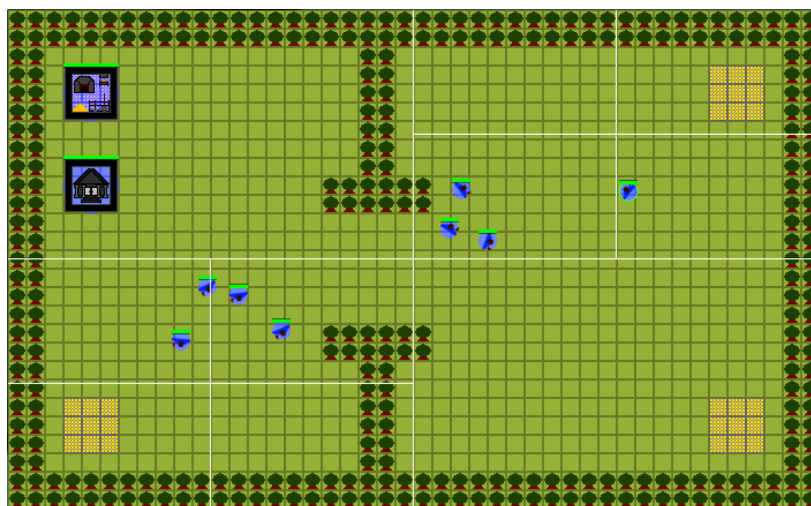


Figura 4.8: Quadtree - Sequência 3 - Com quatro unidades inseridas em dois dos quadrantes, eles são particionados.[1]

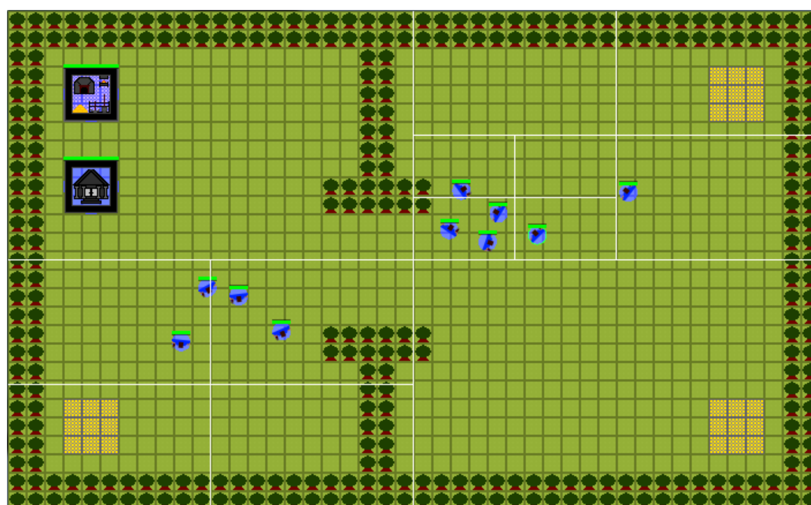


Figura 4.9: Quadtree - Sequência 4 - E assim por diante, com mais unidades inseridas, mais subpartições são criadas. [2]

[1]Fonte: elaborado pelo autor
[2]Fonte: elaborado pelo autor

4.2.2 Resposta

A resposta a colisão é o processo em que as colisões detectadas no passo anterior são resolvidas. Foram implementados dois tipos diferentes de resposta, apresentados abaixo:

Solução 1 - Unidade empurra a outra

Quando a colisão ocorre com uma unidade aliada que não esteja atacando, foi implementada uma solução que remete as leis da física. A unidade em movimento empurra a unidade com quem ela colidiu. A figura abaixo ilustra a sequência de passos referente a solução 1:

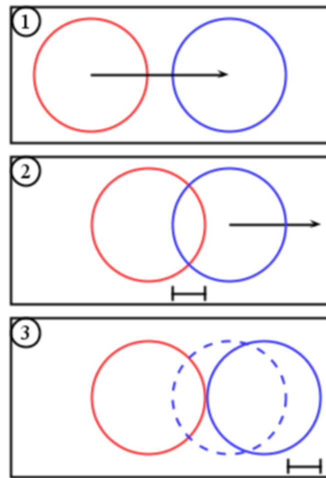


Figura 4.10: Resposta a colisão - Caso 1 - Unidade empurra a outra [1]

Chamaremos de unidade **A** a unidade em vermelho e de **B** a unidade em azul. Neste caso, **A** e **B** são aliados e **B** não está atacando. Os passos da sequência são:

1. **A** está se movimentando em direção a **B**;
2. **A** colide com a **B**;
3. **B** é deslocada em uma distância, de modo que a colisão deixe de ocorrer.

[1]Fonte: elaborado pelo autor

Solução 2 - Unidade dá a volta

Quando a colisão ocorre com uma unidade aliada que esteja atacando ou com uma unidade inimiga, foi implementada uma solução em que ela dá a volta em torno da unidade. Os motivos da solução anterior não ter sido implementada para estes dois casos foram:

- **Colisão com aliada que esteja atacando:** empurrar sua unidade seria prejudicial ao próprio jogador, pois ele poderia acabar movendo suas unidades já posicionadas para locais desfavoráveis;
- **Colisão com inimiga:** o simulador ofereceria uma mecânica passível de abuso por parte dos jogadores.

A figura abaixo ilustra a sequência de passos referente a solução 2:

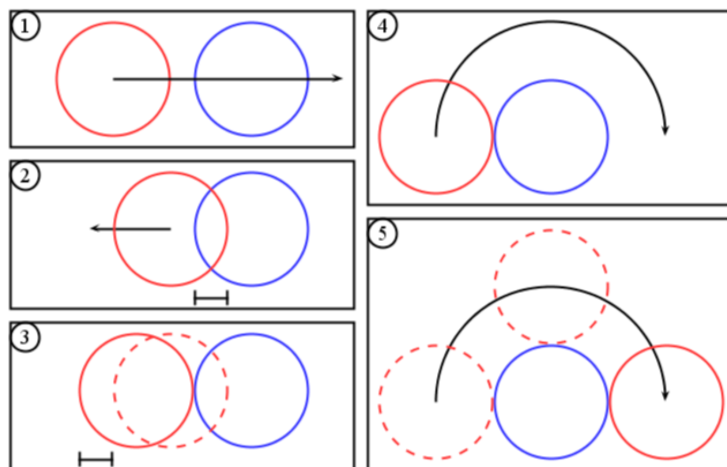


Figura 4.11: Resposta a colisão - Caso 2 - Unidade dá a volta[1]

Para este caso, vale uma das duas condições:

- **A** e **B** são inimigos;
- **A** e **B** são aliados e **B** está atacando.

Os passos da sequência são:

1. **A** está se movimentando em direção a **B**;
2. **A** colide com a **B**;
3. **A** é deslocada em uma distância contrária ao movimento, de modo que a colisão deixe de ocorrer;
4. **A** inicia o movimento em torno de **B**;
5. **A** dá a volta ao redor de **B**.

[1]Fonte: elaborado pelo autor

4.3 Inteligência Artificial

Nesta seção serão apresentadas as técnicas utilizadas na implementação da Inteligência Artificial do simulador.

4.3.1 Análise do mapa

A IA de um RTS pode utilizar diversas informações do mapa do jogo, dentre as quais, podemos destacar a utilização dos pontos de estrangulamento do mapa como um dos elementos estratégicos. Pontos de estrangulamento são áreas do mapa pelas quais as unidades passam ao se locomoverem entre suas diversas regiões. Normalmente são áreas mais estreitas e de alto valor estratégico por serem facilmente defensáveis.

A partir do estudo do **problema do fluxo máximo** em grafos, verificamos que os algoritmos que resolvem este problema, podem ser utilizados na identificação dos pontos de estrangulamento, se aplicados no mapa do simulador. A seguir será introduzido o problema do fluxo máximo e apresentados os métodos utilizados para a identificação dos pontos de estrangulamento.

Problema do fluxo máximo

Imagine que temos uma rede de canos de diferentes espessuras, com diferentes capacidades de transporte de um material. Nesta rede temos dois vértices em especial, a **fonte** em que esse material é produzido e o **sorvedouro** em que o material é consumido. O problema consiste em identificar o fluxo máximo de material que pode trafegar nesta rede, respeitando-se a capacidade de transporte dos canos que a compõem.

O algoritmo de Ford-Fulkerson, um dos primeiros algoritmos que resolvem esse problema, é apresentado abaixo. Sua explicação é baseada no livro *Introduction to Algorithms*, de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein [24]. Serão apresentadas algumas definições para melhor entendimento do algoritmo.

Sejam $G = (V, E)$ um grafo com capacidades (custos) nas arestas, s o vértice fonte, t o vértice sorvedouro, a função capacidade $c : E \to \mathbb{R}$ e a função fluxo $f : E \to \mathbb{R}$, tais que as seguintes restrições sejam satisfeitas:

1. **Restrição de capacidade:** O fluxo não viola a capacidade máxima da aresta.

$$f(u, v) <= c(u, v), \forall u, v \in V$$

2. **Conservação de fluxo:** O saldo do fluxo que entra e sai de um vértice do grafo é nulo, com exceção dos vértices s (fonte) e t (sorvedouro).

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v), \forall u \in V - \{s, t\}$$

3. **Anti-simetria:** O fluxo de arestas anti-paralelas possui fluxo inverso.

$$f(u,v) = -f(v,u), \forall u,v \in V$$

Definimos o fluxo de arestas inexistentes como nulo:

$$f(u,v) = 0, \forall (u,v) \notin E$$

Definimos o **grafo residual** $G_f = (V, E_f)$ e a função capacidade residual $c_f(u,v) = c(u,v) - f(u,v)$, onde $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$.

O algoritmo de Ford-Fulkerson é um algoritmo iterativo, em que a cada passo é selecionado um caminho saindo de s até t no grafo residual, chamado de **caminho aumentador**. Em seguida, o fluxo das arestas deste caminho é incrementado em um valor da menor capacidade residual das arestas de tal caminho e por fim, o valor da capacidade residual das arestas é atualizado.

Algoritmo 5 Algoritmo de Ford-Fulkerson [24]

Entrada: G, s, t

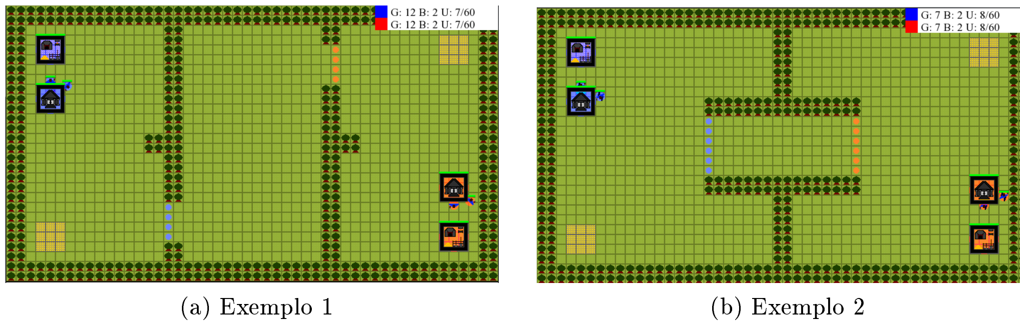
1: **para todos** arestas $(u,v) \in E(G)$ **faça**
2: $\quad f(u,v) \leftarrow 0$
3: **enquanto** existir caminho P de s até t em G_f **faça**
4: $\quad c_f(P) \leftarrow min\{c_f(u,v) : (u,v) \in P\}$
5: $\quad$ **para todos** arestas $(u,v) \in P$ **faça**
6: $\quad\quad f(u,v) \leftarrow f(u,v) + c_f(P)$
7: $\quad\quad f(v,u) \leftarrow f(v,u) - c_f(P)$
8: $\quad\quad c_f(u,v) \leftarrow c(u,v) - f(u,v)$
9: $\quad\quad c_f(v,u) \leftarrow c(v,u) - f(v,u)$
10: **devolva** f

Após o término da execução do algoritmo é realizada uma busca em profundidade a partir da fonte, passando somente pelas arestas as quais a capacidade ainda não foi saturada. Após a busca, obtemos o conjunto S de vértices visitados e o conjunto T de não visitados. A partir do conjunto definimos o corte $(S, V - S)$ do grafo G.

Aplicação no simulador

No contexto do simulador, consideramos que o material são as unidades e que a rede de canos é o mapa do jogo. Para a identificação do ponto de estrangulamento entre duas regiões, utilizamos o algoritmo apresentado em [22], que também resolve o problema do fluxo máximo e rotula os dois conjuntos apresentados anteriormente.

No algoritmo utilizamos as duas regiões em que estão localizadas a base do jogador e a base do oponente, como fonte e sorvedouro. O conjunto de vértices que representa um ponto de estrangulamento são os vértices andáveis em S que possuem um vizinho andável em T. A figura 4.12 ilustra os pontos de estrangulamento encontrados pelo algoritmo no simulador. O algoritmo é executado uma vez no início da partida e não leva em consideração os edifícios dos jogadores.



(a) Exemplo 1 (b) Exemplo 2

Figura 4.12: Pontos de estrangulamento entre as regiões contendo as bases dos jogadores. Os pontos estão destacadas com círculos.[1]

4.3.2 Sistema baseado em regras

O **sistema baseado em regras** foi um método amplamente utilizado em pesquisas de IA na década de 80 [28]. Ela possui uma estrutura composta de:

- Base de conhecimento utilizada pela IA;
- Conjunto de regras do tipo se-então.

No contexto do simulador, a base de conhecimento são as informações relacionadas aos recursos, unidades e edifícios do jogador. As regras são compostas de pares (condição, ação). A condição pode ser simples ou combinada através dos operadores **E**, **OU** e **NÃO**. A ação é executada assim que a condição associada for verdadeira.

O conjunto de regras implementado no simulador é exibido a seguir:

Condição	**Ação**
Mina de ouro com vagas disponíveis	Produzir unidade trabalhadora e enviá-la à mina de ouro
Quantidade de ouro disponível maior que N_1 **E** todos os quartéis produzindo pelo menos uma unidade	Construir novo quartel
Quantidade de ouro disponível maior que N_2 **E** pelo menos um quartel ocioso	Produzir uma unidade de combate no quartel ocioso
Quantidade de unidades de combate menor que N_3	Mover unidades de combate ao ponto de estrangulamento
Quantidade de unidades de combate maior ou igual a N_3	Enviar ordem de ataque ao local do edifício inimigo mais próximo da base do jogador

Para realizar a criação de múltiplas IAs, basta criar conjuntos de regras diferentes. Por exemplo, uma IA defensiva poderia ter uma regra que faça com que ela acumule uma grande quantidade de tropas no ponto de estrangulamento antes de atacar.

[1] Fonte: elaborado pelo autor

Parte II

Parte subjetiva

Capítulo 5

Fabiano

5.1 Desafios e frustrações

A principal dificuldade para a realização do trabalho foi o tempo disponível. Acredito que este seja também um dos principais problemas para a maioria dos alunos do BCC. Como já trabalhava em tempo integral desde o início do projeto, este problema tornou-se maior ainda, já que restavam somente os períodos da noite/madrugada durante a semana e os fins de semana para a realização do trabalho. Foi necessário também que eu tirasse umas "férias" na empresa em que trabalho, para que o andamento do TCC não fosse prejudicado.

Uma frustração foi que muitos dos problemas encontrados durante o desenvolvimento do simulador, tiveram que ter suas soluções simplificadas ou simplesmente deixados de lado. Dois exemplos foram: a simplificação da movimentação em grupo e a não implementação da névoa de guerra. Este último é um dos conceitos básicos encontrados em um RTS, que afetaria diretamente a jogabilidade e a IA do simulador.

5.2 Disciplinas relevantes

De modo geral, considero que a grande maioria das disciplinas contribuíram para meu crescimento pessoal. Em relação ao trabalho, as disciplinas mais relevantes, em nenhuma ordem particular, foram:

- **MAC0110 - Introdução à Computação:** foram apresentados conceitos fundamentais em programação, fornecendo uma base para duas outras disciplinas de grande importância: MAC0122 e MAC0323;
- **MAC0122 - Princípios de Desenvolvimento de Algoritmos:** essencial tanto na formação acadêmica quanto para o trabalho. Nela aprendemos a importância da elaboração de algoritmos que sejam eficientes e bem escritos;
- **MAC0323 - Estruturas de Dados:** introduziu indiretamente conceitos de Programação Orientada a Objetos (POO), já que a disciplina foi ministrada em Java.

MAC0110 e MAC0122 haviam sido ministradas em C. Foram apresentados também alguns algoritmos de busca em grafos, que foram vistos mais a fundo na disciplina MAC0328;

- **MAC0211 - Laboratório de Programação:** disciplina na qual trabalhei em um primeiro projeto de maior porte, com duração de cerca de quatro meses. Seu desenvolvimento foi realizado em três etapas, durante o semestre;
- **MAC0328 - Algoritmos em Grafos:** essencial para o trabalho, já que foram utilizamos algoritmos em grafos para a resolução dos problemas de movimentação de unidades;
- **MAC0414 - Linguagens Formais e Autômatos:** importante para o tratamento do laço principal em que utilizamos uma FSM, também conhecido como autômato finito;
- **MAC0425 - Inteligência Artificial:** uma das mais importantes para o trabalho, já que ele trata diretamente da IA de jogos do gênero RTS. Utilizamos alguns conceitos apresentados na disciplina, como funções heurísticas para a implementação do algoritmo A*.

5.3 Próximos passos

Caso fosse dado prosseguimento ao trabalho, gostaria de implementar o módulo de microgerenciamento da IA que tivesse um desempenho semelhante aos dos jogadores de alto nível. Outros itens interessantes a serem implementados seriam a névoa de guerra e a melhoria da interface do simulador com o módulo da IA.

Capítulo 6

Leandro

6.1 Desafios e frustrações

Uma das partes que considero mais difíceis do nosso projeto é a resposta de colisão. Foram realizadas muitas consultas na internet e em jogos comerciais para termos um conjunto de possibilidades. Em seguida, foram realizadas muitas tentativas de implementação sendo que a maioria não trouxe resultados agradáveis. Depois de muito tempo gasto nesse problema, decidimos utilizar o modo que obteve melhores resultados mas que ainda possui falhas.

Um dos aspectos que considero importante, em relação a eficiência, é a questão da movimentação em grupo. Quando as unidades são selecionadas para mover em grupo, uma busca é feita para cada unidade separadamente. Essa solução, mesmo não sendo eficiente, ficou agradável junto com o sistema de colisão implementado.

Em relação a IA implementada, uma das dificuldades foi com o balanceamento do jogo pois afeta todo modo de pensar em como a IA deve agir no decorrer da partida. Desse modo, decidimos implementar apenas as funcionalidades básicas que uma IA deve ter, sem se preocupar com características mais específicas como composição de unidades.

6.2 Disciplinas relevantes

As disciplinas mais importantes para o desenvolvimento do nosso projeto foram:

- **MAC0110 - Introdução à Computação e**
  MAC0122 - Princípios de Desenvolvimento de Algoritmos: serviram como base do curso, desde o primeiro contato com programação até o desenvolvimento de alguns algoritmos;
- **MAT0139 - Álgebra Linear para Computação:** ajudou no desenvolvimento da resposta de colisão, a qual envolve manipulação de vetores e foram utilizadas alguns conceitos aprendidos nessa disciplina;
- **MAC0323 - Estruturas de Dados:** disciplina onde foram estudadas diversas estruturas de dados importantes que foram utilizadas no projeto;

- **MAC0211 - Laboratório de Programação I e**
  MAC0242 - Laboratório de Programação II: tivemos o primeiro contato com o desenvolvimento de jogos, foram introduzidas alguns conceitos de Orientação a Objetos as quais foram colocadas em prática;
- **MAC0328 - Algoritmos em Grafos:** utilizamos grafos como estrutura da grade do simulador. Foram vistos alguns problemas usando grafos como o problema do caminho mínimo, que é um dos problemas dos trabalho. Uma das soluções implementadas para resolver esse problema foi o algoritmo de Dijkstra, que é visto nessa disciplina;
- **MAC0327 - Desafios de Programação:** nessa disciplina tive o primeiro contato real com a linguagem de programação C++. Um dos aspectos que considero importante nessa disciplina é a familiarização com as rotinas existentes na STL. Além disso, minha capacidade de resolver problemas computacionalmente foi melhorado consideravelmente ao cursar essa disciplina;
- **MAC0414 - Linguagens Formais e Autômatos:** essa disciplina ajudou a entender o conceito de Máquina de Estado, que foram utilizados em algumas partes do nosso projeto;
- **MAC0425 - Inteligência Artificial:** foram introduzidas conceitos e métodos importantes da área de Inteligência Artificial. Um dos algoritmos visto nessa disciplina é o algoritmo A*, que é o algoritmo principal de busca de caminho do trabalho.

6.3 Próximos passos

Os próximos passos do trabalho que considero mais importantes no estado atual são:

- Implementação de novas técnicas de IA;
- Implementação de névoa de guerra;
- Implementação de árvore tecnológica;

Capítulo 7

Agradecimentos

Gostaríamos de agradecer ao professor Paulo Miranda por todo o suporte oferecido na orientação deste trabalho. Estendemos nossos agradecimentos aos professores e aos colegas do IME.

Referências Bibliográficas

[1] Battle.net. http://web.archive.org/web/20100924090612/http://us.battle.net/sc2/en/blog/363375? Acessado em setembro de 2013.

[2] Bwapi. https://code.google.com/p/bwapi/. Acessado em outubro de 2013.

[3] Bwapi. https://code.google.com/p/bwapi/wiki/Competitions. Acessado em outubro de 2013.

[4] E-sports earnings. http://www.esportsearnings.com/games. Acessado em setembro de 2013.

[5] Extreme tech. http://www.extremetech.com/gaming/102413-starcraft-ii-playing-artificial-intelligence-shows-promise. Acessado em outubro de 2013.

[6] Forbes. http://www.forbes.com/sites/insertcoin/2012/12/20/2012-the-year-of-esports/. Acessado em outubro de 2013.

[7] Gamespot. http://www.gamespot.com/gamespot/features/all/real_time/p2_02.html. Acessado em setembro de 2013.

[8] Ign. http://www.ign.com/articles/2006/04/08/the-state-of-the-rts. Acessado em setembro de 2013.

[9] Nova, a starcraft bot. http://nova.wolfwork.com/. Acessado em outubro de 2013.

[10] Oxeye game studios. http://www.oxeyegames.com/game-play-mechanics-of-real-time-strategy-games/. Acessado em agosto de 2013.

[11] Red blob games. http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html. Acessado em outubro de 2013.

[12] Spiegel online. http://www.spiegel.de/international/spiegel/the-boys-with-the-flying-fingers-south-korea-turns-pc-gaming-into-a-spectator-sport-a-399476.html. Acessado em setembro de 2013.

[13] Uol jogos.
http://jogos.uol.com.br/pc/ultnot/2010/06/30/ult182u8395.jhtm.
Acessado em setembro de 2013.

[14] Usa today. http://usatoday30.usatoday.com/tech/gaming/2007-05-21-starcraft2-peek_N.htm. Acessado em outubro de 2013.

[15] Wikia. http://dune.wikia.com/wiki/Dune_II. Acessado em setembro de 2013.

[16] Wikipedia. http://en.wikipedia.org/wiki/Chronology_of_real-time_strategy_video_games. Acessado em setembro de 2013.

[17] Wikipedia. http://en.wikipedia.org/wiki/Electronic_sports. Acessado em outubro de 2013.

[18] Wikipedia. http://en.wikipedia.org/wiki/Tile-based_video_game. Acessado em outubro de 2013.

[19] Wikipedia. http://en.wikipedia.org/wiki/Collision_detection. Acessado em outubro de 2013.

[20] Wikipedia. http://en.wikipedia.org/wiki/Real-time_strategy. Acessado em abril de 2013.

[21] Wikipedia. http://en.wikipedia.org/wiki/Actions_per_minute. Acessado em setembro de 2013.

[22] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(9):1124–1137, 2004.

[23] Michael Buro. Call for ai research in rts games. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, pages 139–142, 2004.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[25] Jason Darby. *Going to War: Creating Computer War Games*. Cengage Learning PTR, 2009.

[26] M. Tim Jones. *Artificial Intelligence: A Systems Approach (Computer Science)*. Jones and Bartlett Publishers, Inc, 2008.

[27] John B. Ahlquist Jr. and Jeannie Novak. *Game Development Essentials: Game Artificial Intelligence*. Cengage Learning, 2007.

[28] Ian Millington and John Funge. *Artificial Intelligence for Games*. CRC Press, 2009.

[29] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[30] Philip Sabin. *Simulating War: Studying Conflict through Simulation Games*. Bloomsbury Academic, 2012.

[31] Loki Software, John R. Hall, and Loki Software Inc. *Programming Linux Games*. No Starch Press, 2001.