UM FRAMEWORK PARA PESQUISA DE INTELIGÊNCIA ARTIFICIAL EM JOGOS DE ESTRATÉGIA POR TURNOS

ISRAEL HERINGER LISBOA DE CASTRO

UM FRAMEWORK PARA PESQUISA DE INTELIGÊNCIA ARTIFICIAL EM JOGOS DE ESTRATÉGIA POR TURNOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Luiz Chaimowicz

Coorientadora: Gisele Lobo Pappa

Belo Horizonte

Julho de 2012

Castro, Israel Heringer Lisboa de
C355f Um framework para pesquisa de inteligência artificial em jogos de estratégia por turnos / Israel Heringer Lisboa de Castro. — Belo Horizonte, 2012.
xxiv, 131 f. : il. col. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Luiz Chaimowicz
Coorientadora: Gisele Lobo Pappa

1. Jogos por computador – Teses. 2. Inteligência artificial – Teses. I. Título. II. Orientador. III. Coorientadora

CDU 519.6*82(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Um framework para pesquisa de inteligência artificial em jogos de estratégia por turnos

ISRAEL HERINGER LISBOA DE CASTRO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. LUIZ CHAIMOWICZ - Orientador
Departamento de Ciência da Computação - UFMG

PROFA. GISELE LOBO PAPPA - Co-orientadora
Departamento de Ciência da Computação - UFMG

PROF. FLÁVIO SOARES CORREA DA SILVA
Departamento de Ciência da Computação - USP

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

PROF. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 19 de julho de 2012.

Este trabalho é dedicado a todos os jogadores que, como eu, são ávidos por experiências mais desafiadoras contra adversários controlados por Inteligência Artificial.

Agradecimentos

Em primeiro lugar agradeço de todo meu coração a Deus, que ao longo da minha vida tem guiado passos de forma que eu sozinho certamente não seria capaz. Agradeço a Ele que, mesmo sem eu merecer, se mantém fiel com Seu amor e misericórdia todos os dias.

De forma mais do que especial, agradeço à minha querida esposa Ariana, com quem me casei durante o período de estudos do mestrado. Obrigado por me amar e apoiar nesta empreitada, além de suportar minha ausência logo no primeiro ano do nosso casamento, devido ao volume de trabalho, especialmente nos períodos finais de conclusão deste trabalho.

À minha família – principalmente meus pais Marcelo e Anavera minha irmã Rebeca – que sempre me deram todo tipo de apoio que precisei em todos os momentos da minha vida, além de sempre terem me incentivado nos estudos. Provavelmente nem sei o tanto que vocês já se sacrificaram por mim para que hoje eu esteja concluindo mais esta etapa, mas de toda forma reconheço e agradeço por tudo que são na minha vida.

Também agradeço muito aos meus dois orientadores, Luiz Chaimowicz e Gisele Pappa, que com sua experiência me ajudaram a conduzir o trabalho até o escopo atual – por sinal bem diferente do original – e contribuíram bastante com diversas sugestões e revisões. Obrigado pela paciência e pelas cobranças em meio a tantas mudanças e ausências durante este tempo.

Agradeço a todos os meus amigos, aos quais tão pouco tempo dediquei nos últimos tempos. Finalmente poderemos voltar a sair, conversar e nos divertir. Obrigado especialmente ao pessoal da célula, que me apoiou muito com as orações durante esses últimos meses. Gostaria ainda de destacar meu obrigado ao Flávio e ao Lucas, que além de compartilhar a paixão por jogos, me ajudaram diretamente neste trabalho.

Também agradeço aos meus chefes de trabalho, tanto no meu emprego atual quanto no anterior, onde estava quando comecei o curso. Obrigado pela compreensão

ao permitirem horários de trabalhos mais flexíveis que permitiram que eu frequentasse as aulas durante o dia. Obrigado também aos meus colegas de trabalho que, especialmente nessa reta final, trabalharam ainda mais para suprir a minha ausência nos momentos em que houve o projeto demandava algum esforço extra.

Ao DCC e a UFMG, pela oportunidade de realizar este curso e permitir que conduzisse um trabalho que para mim foi muito interessante e motivador.

Agradeço também a todos os que participaram das avaliações relatadas neste trabalho. Vocês foram de grande ajuda ao dedicarem tempo a este projeto e apresentarem abertamente suas opiniões e sugestões.

Por fim, agradeço aos criadores da série *Sid Meier's Civilization*, especialmente do jogo *Civilization* II. Foi em meio a centenas de horas neste jogo que por volta de 15 anos atrás comecei a me interessar pela área da inteligência artificial nos jogos, o que me motivou a estudar tanto a graduação quanto o mestrado. Certamente este trabalho seria bem diferente se não fosse por essa experiência.

"Nossa abordagem para criar jogos é encontrar a diversão em primeiro lugar, e então usar a tecnologia para aumentar a diversão."

(Sid Meier)

Resumo

Em jogos digitais, a Inteligência Artificial (IA) é um dos elementos mais importantes para garantir o desafio e a diversão, especialmente em gêneros como os jogos de estratégia. No entanto, devido à grande complexidade dos jogos desse gênero, quase sempre o desempenho da IA é muito fraco quando comparado aos jogadores razoavelmente experientes.

Nesse cenário, a necessidade de técnicas e algoritmos de IA mais robustos trazem desafios muito interessantes para as pesquisas da área. Uma grande dificuldade existente, porém, é a ausência de ferramentas adequadas para os experimentos em jogos reais, seja porque a manutenção dos códigos é muito complicada ou porque em jogos comerciais não se tem acesso a todas as modificações necessárias.

Nesse contexto, propomos a criação de um *framework* propício à pesquisa de algoritmos de IA em jogos de estratégia por turnos (TBS) através da criação de um jogo com arquitetura especialmente planejada para a pesquisa de IA em jogos TBS de forma que futuros trabalhos na área possam ter uma ferramenta adequada aos experimentos necessários.

O objetivo principal do trabalho é fornecer uma ferramenta completa para o desenvolvimento e a experimentação de IA em jogos TBS. Os principais requisitos que devem ser atendidos por esta ferramenta são (i) permitir a realização de confrontos entre diferentes agentes de IA sem a necessidade de um jogador humano, (ii) possibilitar o desenvolvimento de algoritmos de IA sem a necessidade de se escrever muito código não relacionado ao algoritmo utilizado, (iii) possuir arquitetura bem modularizada de forma que a IA esteja em módulos independentes do restante do jogo e não possa trapacear e (iv) conter um jogo TBS que contemple os principais elementos existentes em outros jogos do gênero, de forma que os experimentos realizados sejam aplicáveis em jogos reais.

Após a conclusão do desenvolvimento do *framework*, conduzimos experimentos de avaliação com participantes de diferentes níveis de experiência, tanto em relação à

área de IA quanto a desenvolvimento de jogos de forma geral. De acordo com os resultados obtidos, verificamos que a ferramenta construída conseguiu alcançar os objetivos propostos e tem grande potencial na utilização em experimentos de novas técnicas e algoritmos de IA em jogos TBS.

Palavras-chave: Jogos Digitais, Jogos de Estratégia por Turnos, Inteligência Artificial, *Framework* para Experimentos.

Abstract

In digital games, the Artificial Intelligence (AI) is one of the most important elements to ensure both challenge and fun, especially in genres such as strategy games. However, due to the high complexity of games of this genre, the AI performance is often very weak when compared to fairly experienced players.

With this scenario, the need for more robust AI techniques and algorithms bring very interesting challenges to researches in the area. However, a major difficulty faced by researchers is the lack of proper tools for conducting experiments in real games, either because the maintenance of the code is too complicated or because real commercial games do not offer the access for modifying everything needed.

In this context, we propose the creation of a framework appropriate for the research on AI algorithms in Turn-Based Strategy (TBS) games by creating a game with its architecture specially designed for AI research on TBS games, in order to have a proper tool for experiments for future works.

The main goal of the work is to provide a comprehensive tool for the development and experimentation of AI in TBS games. The main requirements which must be with are to (i) allow the performance of contests between different AI agents without the need for a human player, (ii) enable the development of AI algorithms with no need of writing much code not related to the algorithm itself, (iii) have a well modularized architecture so the AI is totally in independent modules from the rest of the game and is not allowed to cheat and (iv) contain a TBS game which includes the main elements present on other games of the genre, so that the experiments on it are applicable in real games.

Upon completion of the development phase, we conducted evaluation experiments with participants from different experience levels, both relating to AI area and games development in general. According to the results, we found that the tool built on this work achieved the proposed objectives and has a great potential for use in experiments of new algorithms and techniques applied to TBS games.

Keywords: Videogames, Turn-Based Strategy Games, Artificial Intelligence, Framework for Experiments.

Lista de figuras

Lista de siglas

Sigla	Descrição
DLL	*Dynamic-link Library* (Biblioteca de vínculo dinâmico)
IA	Inteligência Artificial
MMOG	*Massive Multiplayer Online Game* (jogo eletrônico *online* multijogador em massa)
NPC	*Non Player Character* (Personagem Não Jogável)
RBS	*Rule-Based System* (Sistema Baseado em Regras)
RTS	*Real-Time Strategy* (Estratégia em Tempo Real)
SDK	*Software Development Kit* (Pacote de Desenvolvimento de Software)
TBS	*Turn-Based Strategy* (Estratégia por Turnos)
UI	*User Interface* (Interface com o usuário)

Sumário

Capítulo 1

Introdução

1.1 Motivação

A indústria de jogos eletrônicos, que atraía mera curiosidade nos anos 70, tornou-se extremamente relevante para a indústria do entretenimento como um todo. Apenas nos Estados Unidos, foi responsável por um acréscimo de 4,9 bilhões de dólares no PIB em 2009, e o crescimento anual real da indústria foi de 16,7% no período de 2005 a 2008 [Siwek, 2010].

Com esse crescimento, a necessidade de novas tecnologias e experiências também aumentou, o que pode ser facilmente observado pela evolução de vários aspectos dos jogos como, por exemplo, os recursos gráficos e os mecanismos de controle utilizados, como sensores de movimento e outros. A evolução de outros aspectos, no entanto, não é facilmente percebida pelos jogadores por se tratarem de mecanismos internos, como as técnicas e algoritmos de inteligência artificial (IA) utilizados.

Em alguns gêneros de jogos – como quebra-cabeças e jogos de plataforma – normalmente não há necessidade de algoritmos de IA complexos ou muito avançados, pois o uso de estratégias simples e predefinidas é suficiente para manter o jogo em um nível de desafio bastante razoável. Outros gêneros – especialmente jogos de estratégia – já demandam ações mais complexas ou coordenadas dos adversários, e algoritmos muito simples costumam ser muito fracos se comparados ao desempenho de jogadores humanos. Nos jogos de estratégia, portanto, a Inteligência Artificial (IA) é um aspecto de grande importância, fundamental para que o jogo seja considerado divertido e, ao

mesmo tempo, desafiador.

Os jogos de estratégia são normalmente ambientados em um mapa bidimensional e representados através de visão isométrica ou superior (como se fosse vista por cima do mapa). O jogador pode executar diversas ações com os elementos controláveis do jogo e de acordo com as suas regras. Os exemplos mais comuns são a movimentação de unidades, ataque, investimentos em pesquisa ou avanços tecnológicos, construção de novas bases (ou cidades), etc. Assim, o jogador deve gerenciar os recursos que possui tendo em vista o objetivo final da partida, que normalmente envolve eliminar os adversários através de batalhas, embora haja algumas variações.

Os jogos de estratégia podem ser divididos em duas categorias principais: os jogos de estratégia em tempo real (RTS, do inglês *Real-Time Strategy*) e os de estratégia por turnos (TBS, do inglês *Turn-Based Strategy*). A principal diferença entre as duas categorias envolve o momento em que os jogadores podem efetuar suas jogadas. Enquanto nos jogos RTS o jogo não sofre pausas ou interrupções e os jogadores devem executar suas ações simultaneamente e de forma contínua (ou em tempo real[1]), nos jogos TBS o jogo se desenvolve em turnos, quando cada jogador possui sua vez de jogar, de forma semelhante ao que ocorre nos jogos de tabuleiro como o xadrez ou o jogo *War*®, exibido na Figura 1.1. Neste trabalho, estamos interessados na aplicação da inteligência artificial a esta última categoria – os jogos TBS.

Nos jogos de tabuleiro clássicos, em que o ambiente é completamente observável e as ações são totalmente determinísticas, já existem IAs capazes de vencer os grandes mestres. Um exemplo é o jogo de damas, que foi completamente resolvido recentemente [Schaeffer et al., 2007]. Já nos jogos de estratégia comerciais, não temos conhecimento de nenhum exemplo semelhante, pois quase sempre o desempenho da IA é muito fraco em comparação com jogadores razoavelmente experientes. Em grande parte, as dificuldades podem ser atribuídas à grande complexidade dos jogos do gênero,

[1] Em jogos, a definição de "tempo real" não é tão forte quanto nos sistemas de tempo real normalmente tratados em outras áreas da computação, nos quais existem severas restrições de tempo para a manutenção do sistema. No contexto de jogos, em geral "tempo real" se refere apenas a um "tempo contínuo", onde o jogo decorre sem pausas e todos os jogadores atuam simultaneamente.

que possuem informações imperfeitas e centenas de unidades, construções, árvores tecnológicas e outros elementos, o que torna um mapeamento completo das decisões computacionalmente inviável.

Figura 1.1: Tabuleiro do jogo *War*®, baseado no jogo norte-americano *Risk*.

Para contornar esse problema e tentar manter o nível de desafio contra jogadores mais experientes, é comum a utilização de regras diferenciadas para os jogadores controlados por IA, de forma a torná-los mais fortes. Alguns dos exemplos mais comuns que encontramos envolvem aspectos como visão completa do mapa (nos casos em que o mapa não é completamente visível a todos), bônus em batalhas, diminuição de penalidades, construções e outros recursos mais baratos e outros. O grande problema dessa estratégia é que normalmente, ao invés de tornar as partidas mais interessantes e desafiadoras, acaba por torná-las desbalanceadas, o que pode se tornar motivo para desânimo dos jogadores. O motivo para a insatisfação dos jogadores é que, quando estes percebem essas distorções das regras e a inexistência de um "adversário" (IA) bom o suficiente, o nível de recompensa obtido no jogo pode se tornar significativamente mais baixo.

Um recurso que é bastante explorado na tentativa de minimizar esses problemas é a possibilidade de que as partidas sejam jogadas entre vários jogadores humanos, local ou remotamente, podendo assim até mesmo dispensar o uso de jogadores controlados pela IA na partida. No entanto, esse recurso encontra certas restrições nos TBS devido

ao grande tempo exigido para conclusão das partidas e, principalmente, aos longos tempos de espera entre cada turno. Ao contrário dos jogos RTS, em que é possível jogar partidas mais rápidas (até 1 hora – embora em alguns jogos modernos, partidas de campeões podem durar menos de 15 minutos), os jogos por turno exigem um tempo muito maior, podendo durar várias horas, o que torna mais difícil a realização de partidas entre vários jogadores, tornando ainda maior a necessidade de uma IA desafiadora.

Ainda em comparação com jogos de tempo real, os jogos por turnos apresentam uma importante vantagem para o desenvolvimento e execução da IA: como há tempo disponível para realização das jogadas sem que os adversários interajam com o mapa, é possível utilizar técnicas computacionalmente mais robustas e que exigem um tempo maior de processamento, pois alguns segundos de espera entre os turnos são facilmente aceitáveis pelos jogadores humanos. Em jogos em tempo real, a demora de alguns segundos no processamento da IA pode implicar em uma IA mais fraca, por não ser capaz de responder adequadamente a mudanças no ambiente provocadas pelos demais jogadores, sendo assim mais facilmente derrotável.

Jogos comerciais normalmente são desenvolvidos sob grande restrição de tempo e orçamento e, por isso, normalmente não dedicam o investimento necessário para a pesquisa em IA, já que do desenvolvimento considerada um aspecto secundário do jogo. Jogos nos quais a IA é considerado o ponto mais importante – como "*Black & White*" (2001), produzido pela *Lionhead Studios* – são a exceção. De acordo com Fairclough et al. [2001], até por volta do ano 2000, a IA podia tipicamente utilizar apenas 10% dos ciclos do processador, e mesmo assim não era investido tempo suficiente para desenvolver um mecanismo de IA eficiente para os poucos recursos disponíveis.

Como será discutido ao longo deste trabalho, há muito mais trabalhos sobre jogos RTS em comparação aos jogos TBS. Para a pesquisa de IA em jogos TBS, temos um cenário em que há poucas ferramentas disponíveis para os pesquisadores focarem apenas em seus objetos de pesquisa, e normalmente é necessário um trabalho muito grande para obter um ambiente onde os experimentos possam ser feitos de maneira satisfatória.

A pesquisa de algoritmos e técnicas de IA em jogos TBS normalmente demanda uma ou mais dentre três opções: (1) modificar jogos com código aberto, os quais infelizmente não costumam ser de fácil adaptação para as necessidades de pesquisas, por não terem sido criados com esse objetivo; (2) criar um ambiente/jogo próprio no qual possam ser feitos os experimentos, o que demanda esforço considerável e muitas vezes é uma tarefa inviável; (3) modificar jogos comerciais que possuam alguma opção de adaptação ou modificação. Esta última opção, no entanto, é bastante difícil de ser executada em certos tipos de pesquisa, pois normalmente não são permitidas modificações em mecanismos pertencentes ao núcleo do jogo, como a IA, o que inviabiliza a utilização desses jogos quando essas modificações se fazem necessárias. Assim, percebemos que há uma grande demanda de um jogo TBS que permita criar ou alterar agentes[1] de IA de forma simples e direta, permitindo a exploração de algoritmos e técnicas de IA com maior liberdade.

1.2 Objetivos do trabalho

Nesse contexto, este trabalho tem por objetivo a criação de um *framework* propício à pesquisa de algoritmos de IA em jogos TBS. De forma mais específica, propomos a criação de um jogo com arquitetura especialmente planejada para a pesquisa de IA em jogos TBS, de forma que futuros trabalhos na área possam ter um ponto de partida comum. Com isso, pretendemos também incentivar as pesquisas envolvendo IA em jogos TBS através da criação de um ambiente que proporcione redução do esforço necessário para modificações ou adaptações na ferramenta, permitindo aos pesquisadores focar apenas no interesse do trabalho que venham a desenvolver.

O jogo criado deve contemplar as diversas tarefas normalmente presentes em jogos TBS, como gerenciar recursos, gerar e executar planos, compreender e reagir

[1] Na IA, um agente inteligente é uma entidade capaz de observar e agir em determinado ambiente e direciona suas ações para alcançar um objetivo [Russel & Norwig, 2003]. Normalmente, agentes controlados por IA também são chamados de bot (termo mais comum em jogos de ação em primeira pessoa) ou Personagem Não-Jogável (NPC, do inglês Non-Playable Character)

diante de ações de adversários, entre outras, bem como um agente que permita a demonstração do *framework* nas diversas ações disponíveis no jogo.

Em resumo, o principal objetivo deste trabalho é fornecer uma ferramenta completa para o desenvolvimento e a experimentação de inteligência artificial em jogos de estratégia por turnos. Os principais requisitos que devem ser atendidos por esta ferramenta são (i) permitir a realização de confrontos entre diferentes agentes de IA sem a necessidade de um jogador humano, (ii) possibilitar o desenvolvimento de algoritmos de IA sem a necessidade de se escrever muito código não relacionado ao algoritmo utilizado, (iii) possuir uma arquitetura bem modularizada de forma que a IA esteja em módulos independentes do restante do jogo, e não possa trapacear e (iv) conter um jogo TBS que contemple os principais elementos existentes em outros jogos do gênero, de forma que os experimentos realizados sejam aplicáveis em jogos reais.

1.3 Organização da dissertação

Esta dissertação está dividida em seis capítulos, e o restante do trabalho está organizado da seguinte forma:

- Capítulo 2, Referencial Teórico e Trabalhos Relacionados: neste capítulo apresentamos uma revisão dos principais conceitos envolvidos em jogos TBS e dos principais trabalhos relacionados ao desenvolvimento de jogos, especialmente na área de inteligência artificial em jogos de TBS.
- Capítulo 3, O Jogo Território: após a introdução dos principais aspectos em jogos TBS, apresentamos o jogo criado para uso no *framework*, as regras envolvidas, os conceitos específicos e suas características únicas.
- Capítulo 4, O *Framework*: com base nas regras e conceitos do jogo Território, este capítulo apresenta o *framework* criado, as definições técnicas, a modelagem utilizada e como cada elemento deve ser utilizado para a criação de um agente de IA.
- Capítulo 5, Avaliação: com o objetivo de avaliar o jogo e o *framework*

criados neste trabalho, apresentamos neste capítulo a metodologia utilizada para avaliação, os experimentos realizados e a avaliação dos resultados obtidos.

- Capítulo 6, Considerações Finais: No último capítulo, concluímos apresentando as considerações finais acerca do trabalho, os próximos passos e recomendações de trabalhos futuros.

Capítulo 2

Referencial Teórico e Trabalhos Relacionados

Neste capítulo, apresentamos o referencial teórico para o desenvolvimento do trabalho e uma revisão dos trabalhos relacionados encontrados na literatura. Em primeiro lugar, apresentaremos o que são jogos TBS e as principais características encontradas em jogos deste gênero. Em seguida, mostraremos como normalmente a IA é organizada nos jogos TBS e os componentes em que costuma ser dividida. Logo após, apresentaremos três ferramentas que foram analisadas do ponto de vista dos experimentos de IA. Por fim, complementaremos com trabalhos relacionados a jogos de estratégia por turnos e inteligência artificial.

O conceito de Inteligência Artificial é bastante abrangente, e pode apresentar até mesmo divergências entre diferentes trabalhos. Neste trabalho, uma IA deve ser entendida como um agente – ou jogador – capaz de interagir de forma autônoma no jogo. Para isso, deve ser capaz de perceber o estado atual do jogo, tomar decisões e efetuar suas jogadas.

2.1 Jogos TBS

Os jogos TBS são uma categoria de jogos de estratégia onde os jogadores atuam no jogo alternando-se através de turnos, onde apenas um jogador pode efetuar jogadas a cada vez. Devido a essa característica, os jogos TBS costumam ser muito mais longos

que os demais jogos de estratégia, com partidas que duram muitas horas ou até mesmo vários dias. Outra implicação importante é que as estratégias adotadas costumam ser muito diferentes das presentes nos jogos em tempo real, principalmente no que se refere à velocidade em que as decisões precisam ser tomadas.

A representação mais comum para o ambiente dos jogos TBS é através de um mapa bidimensional discreto, seja ele plano, cilíndrico ou até mesmo toroidal. Não há, no entanto, nenhuma restrição para que sejam utilizados mundos tridimensionais e espaços contínuos.

Estes jogos costumam ser ambientados em uma guerra ou batalha, onde o jogador pode controlar e gerenciar os recursos de sua cidade, império, reino, civilização, etc. Esses recursos se referem tanto aos recursos naturais disponíveis no mapa, e que o jogador precisa extrair (e.g. construir minas para obter ferro) quanto às tropas e exércitos com os quais enfrentará batalhas.

Para controlar esses recursos, o jogador tem à sua disposição cidades, construções ou bases militares, onde pode armazenar os recursos obtidos e criar novas tropas, além de obter avanços tecnológicos que lhe garantam acesso a recursos mais modernos e avançados.

Uma vez que na grande maioria dos jogos TBS o ambiente é de uma guerra, o objetivo também costuma estar relacionado à conquista de territórios e eliminação dos demais adversários através de batalhas militares. No entanto, isso não é uma regra, e a vitória também pode estar associada a condições econômicas, soberania tecnológica ou relacionamentos diplomáticos.

Dentre os jogos comerciais do gênero TBS, a série *Civilization* criada por Sid Meier é o exemplo mais conhecido, e vamos tomá-la como exemplo para ilustrar essas características em um jogo real. No *Civilization*, cada jogador é o líder de uma civilização e deve guiá-la ao longo do tempo através de aspectos econômicos, científicos e militares. Para isso ele deve fundar cidades, nas quais poderá criar construções e unidades militares, além de ter acesso a recursos tecnológicos e econômicos.

A forma mais comum de se obter a vitória é através da dominação militar, mas também é possível obter vitória através de aspectos tecnológicos (por exemplo, o

primeiro jogador a produzir e enviar com sucesso uma nave para o espaço), diplomáticos (o jogador que, em determinado momento, receber apoio formal da maioria dos outros jogadores, através de uma votação), culturais (o jogador que tiver produzido maior influência cultural no planeta) ou mesmo após um limite de tempo.

Uma partida se inicia em um período pré-histórico, no qual o jogador pode descobrir tecnologias primitivas como mineração e agricultura e criar unidades igualmente primitivas, como guerreiros munidos de clavas ou porretes. À medida que o jogador vai evoluindo tecnologicamente, ganha-se acesso a recursos que permitem a criação de unidades e construções mais avançadas e fortes. O jogador deve guiar o crescimento da civilização tentando balancear aspectos militares, econômicos, científicos e culturais, aliando-se ou declarando guerra com outros jogadores, na tentativa de se tornar a civilização mais importante.

As unidades são as entidades que podem movimentar, atacar, fundar novas cidades e executar as ações diretamente no mapa do jogo. Alguns exemplos de unidades existentes são os guerreiros, trabalhadores, catapultas, tanques, porta-aviões e mísseis nucleares. Elas podem se movimentar pelo mapa e interagir com os demais elementos do jogo, seja através de batalhas (ataque e defesa) ou de manipulação dos recursos (construção de minas para extração de ferro, fazendas para produção de alimentos etc.).

Figura 2.1: Captura de tela do jogo *Civilization* V.

As cidades são os locais onde o jogador recebe e processa os recursos naturais do jogo, como comida (para crescimento da população), dinheiro, produção científica e, principalmente, a criação de novas unidades e construções. À medida que as cidades crescem, os jogadores podem acessar recursos que se encontram a uma distância maior de sua localização. Nas cidades, também é possível criar unidades ou construções, de acordo com o estágio tecnológico em que o jogador se encontra e os recursos naturais que tem à sua disposição.

A Figura 2.1 exibe uma captura da tela do mais recente jogo da série, "*Civilization* V", lançado em 2010. Nela podemos observar alguns elementos do jogo, como o mini-mapa no canto inferior direito (1), indicadores da civilização e dos recursos disponíveis na barra superior à esquerda (2), a unidade ativa e suas ações possíveis no canto inferior esquerdo (3) e o mapa do jogo (4), ao centro, com a exibição dos recursos, geografia e estado das cidades e unidades.

2.2 Arquitetura de IA em Jogos TBS

A arquitetura necessária para a construção de uma IA em jogos TBS pode variar bastante de acordo com a ferramenta utilizada, devido às próprias restrições apresentadas pelo jogo envolvido. Apesar dessas diferenças, no entanto, Millington & Funge [2009] apresentam uma arquitetura básica para a construção de uma IA em jogos TBS de uma forma geral.

De acordo com essa arquitetura, os principais componentes para a construção de uma IA em jogos TBS são: (i) gerenciamento de execução, responsável por manter o tempo de execução da IA aceitável; (ii) análise tática, responsável pela observação do ambiente, interpretação do estado do jogo e definição de parâmetros que serão utilizados na tomada das decisões estratégicas; (iii) decisões estratégicas: responsável por definir as ações do jogador do ponto de vista estratégico (alocação de recursos, destino de unidades, etc.); (iv) planejamento de rotas, responsável pelo planejamento de rotas das unidades para que elas alcancem da melhor forma possível o destino definido pelo módulo estratégico.

Na Figura 2.2 apresentamos um diagrama dessa arquitetura e do fluxo entre os componentes durante a execução de jogadas. O componente de gerenciamento de execução é uma tecnologia de suporte, para garantir que a jogada seja executada em tempo aceitável. Ele se comunica com o componente da análise tática, que por sua vez se comunica com o componente de decisões estratégicas. Aqui é onde ocorre a tomada de decisão da IA, e onde são planejados caminhos que serão tomados pelas unidades. Esses caminhos, por sua vez, podem impactar nas decisões que devem ser tomadas. Por fim, depois de tomadas as decisões, as ações são executadas sobre o mapa do jogo.

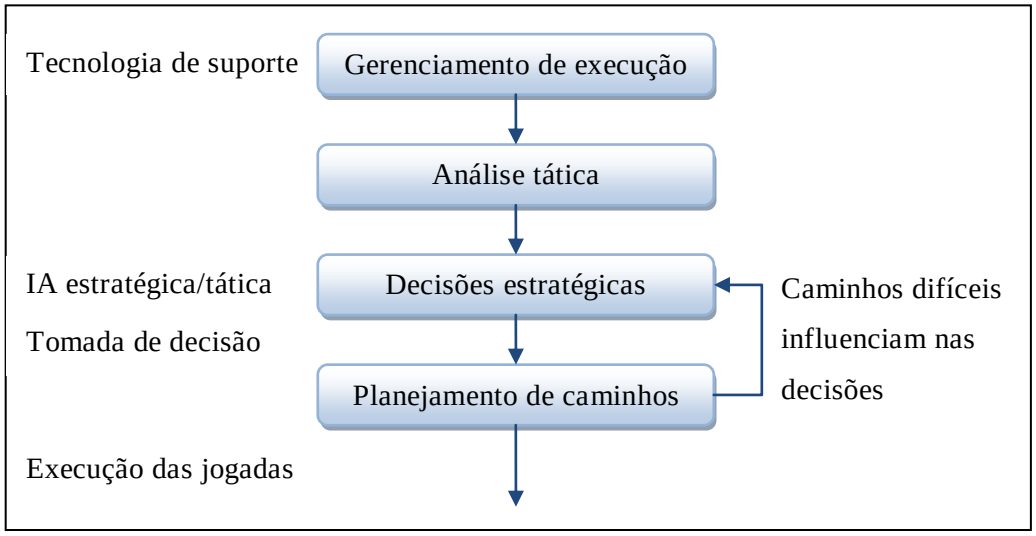

Figura 2.2: Arquitetura básica da IA para jogos TBS.

2.2.1 Gerenciamento de execução

Nos jogos em tempo real, o gerenciamento de execução é necessário para garantir a consistência temporal dos dados que estão sendo processados pelo sistema da IA e alterados pelos demais jogadores de maneira simultânea. Já nos jogos TBS, onde a consistência temporal não é um problema, o gerenciamento de execução é responsável por garantir que o tempo de processamento da IA se mantenha em um nível aceitável.

Em alguns jogos, há um limite de tempo para que os jogadores possam efetuar suas jogadas. Nesse caso, o gerenciamento de execução precisa garantir que o tempo disponível seja bem utilizado, para não ocorrer do jogador perder a vez sem ter concluído todas as ações que poderia executar. Mesmo em jogos onde não há essa limitação, o gerenciamento de execução é igualmente importante, pois caso as jogadas de uma IA sejam longas demais, o aspecto de diversão para os jogadores humanos será duramente afetado, e pode levar o jogador a abandonar a experiência.

2.2.2 Análise Tática

Para efetuar uma jogada, é necessário que a IA processe as informações do jogo, especialmente o estado atual e as últimas jogadas realizadas pelos adversários, de

forma que o jogo seja interpretado taticamente. A análise tática é responsável por analisar essas informações disponíveis e estruturá-las, de forma que possam fornecer insumos para que as decisões estratégicas sejam tomadas.

Os insumos gerados nessa etapa podem variar de acordo com os objetivos ou o contexto do jogo. Alguns exemplos que podem ser utilizados são a identificação de áreas do mapa interessantes por disponibilidade de recursos, marcação de áreas de "controle militar" e regiões perigosas onde se deve evitar planejar rotas. Existem várias técnicas e algoritmos capazes de estruturar essas informações e processá-las de maneira adequada, o que pode variar principalmente de acordo com a complexidade. De acordo com Millington & Funge [2009], as técnicas mais utilizadas são os *waypoints* táticos e mapas de influência.

A primeira técnica, denominada *waypoints* táticos, consiste na marcação de pontos no mapa seguida de associação de uma informação estratégica a cada um desses pontos, como locais de controle militar inimigo ou onde há recursos naturais de interesse. A segunda técnica, chamada mapas de influência, consiste em manter o registro da influência de cada jogador sobre determinada região. Essa influência normalmente é calculada com base na proximidade de unidades militares, cidades ou do tempo em que determinado jogador detém o controle da região.

2.2.3 Decisões Estratégicas

Com base na análise tática previamente realizada, passa-se então à etapa das decisões estratégicas, onde são tomadas as decisões de alto nível, como o que construir nas cidades, para onde movimentar as tropas e qual tecnologia pesquisar. As decisões estratégicas tomadas são, então, mapeadas nas ações do jogo que o jogador efetuará em sua jogada, interagindo com o ambiente do jogo.

Tipicamente, as decisões estratégicas em jogos TBS costumam ser tomadas através de técnicas simples, como máquinas de estado finito (FSM, do inglês *Finite State Machine*) e árvores de decisão. De acordo com Millington & Funge [2009], muitas vezes a IA dos jogos é altamente baseada em scripts, e as decisões militares estão representadas por códigos fixos do tipo "IF... THEN", o que muitas vezes resulta em IAs altamente previsíveis.

2.2.4 Planejamento de Rotas

Durante o processo de decisões estratégicas da IA, algumas decisões acerca do destino das unidades que podem ser movimentadas no jogo devem ser tomadas. Para minimizar o custo de movimentação (seja o custo em tempo, turnos, ou o custo de determinados recursos) dessas unidades, é desejável que o caminho percorrido seja o menor possível. Abordagens tipicamente gulosas, como simplesmente tentar se aproximar do destino a cada turno, não são muito interessantes, pois normalmente caminhos mais curtos estão associados não apenas a distâncias, mas também à movimentação em diferentes tipos de terrenos ou até mesmo a existência de facilitadores, como estradas. O planejamento de rotas é responsável, então, por encontrar o caminho que se quer percorrer até o destino, considerando-se as informações disponíveis no momento.

Dentre os algoritmos de planejamento de rotas, os mais utilizados são o algoritmo de Djikstra [Djikstra, 1959] e o A* [Hart et al., 1968]. Destes dois, o A* é o mais utilizado em jogos devido a sua eficiência e simplicidade de implementação [Millington & Funge, 2009].

Nos jogos TBS, o planejamento de rotas possui ainda uma utilidade especial: auxiliar os jogadores humanos na movimentação. Muitas vezes, o jogador deseja simplesmente movimentar uma unidade até determinado local, mas a tarefa de ter encontrar o melhor caminho a cada rodada pode ser um tanto trabalhosa. Além disso, é praticamente inviável que um jogador humano se lembre das rotas planejadas individualmente para dezenas ou até mesmo centenas de unidades. A Figura 2.3 apresenta um trecho da tela do jogo *Civilization* V (2010) que ilustra essa utilização, onde podemos observar uma unidade (seta vermelha) que o jogador humano deseja movimentar até determinado destino (seta verde), e o planejamento de rotas do jogo mostra o caminho que vai ser percorrido a cada turno (setas alaranjadas). A seta azul indica a borda (linha azul no mapa) que indica o limite que a unidade pode se movimentar naquele turno. Assim, o jogo automatiza parte da movimentação das unidades, o que auxilia o jogador no gerenciamento de aspectos mais relevantes da partida.

Figura 2.3: Planejamento de rota no jogo *Civilization* V.

2.3 Trabalhos Relacionados

A pesquisa em jogos digitais, quando comparada a outras áreas da computação como Inteligência Artificial, é bastante recente. Embora existam registros de pesquisas envolvendo jogos há algumas décadas [Malone, 1981], apenas com o crescimento da indústria e a demanda por profissionais mais especializados houve aumento considerável no interesse acadêmico na área. Dentre os trabalhos revisados, a maior parte deles tem o foco nos componentes de análise tática e decisões estratégicas, de acordo com a divisão apresentada na seção anterior.

Um dos primeiros trabalhos a trazer motivação para a pesquisa de IA em jogos é

de Laird & Jones [1998], que mostrou ser tecnicamente possível em jogos o uso das técnicas utilizadas em seu sistema de IA para pilotos artificiais, devido ao seu baixo custo de processamento. Laird & van Lent [2000] também publicaram um trabalho importante para motivar a pesquisa em jogos, afirmando que jogos de computador oferecem ambientes interessantes e desafiadores para muitos problemas da IA.

Analisando os trabalhos relacionados a IA em jogos de estratégia, percebemos que o volume de trabalhos publicados acerca de jogos TBS é muito baixo quando comparado a jogos RTS. Pesquisas em grandes bibliotecas científicas como a ACM[1] e o IEEE[2] retornam quase 10 vezes mais artigos relacionados a jogos RTS que a jogos TBS[3]. No entanto, como grande parte dos algoritmos e técnicas utilizados em um gênero são aplicáveis ao outro, não é necessário nos ater apenas aos trabalhos realizados em jogos TBS. Se houver necessidade de alguma ressalva ou restrição, apresentaremos no momento adequado.

Em sua dissertação de mestrado, Marc Ponsen [Ponsen, 2004] utilizou com sucesso um algoritmo evolucionário para descobrir novas táticas e estratégias para jogos RTS. Nesse trabalho, foi possível utilizar táticas e estratégias descobertas através de um pré-processamento – não em tempo real – de forma que melhorassem o desempenho da base de regras dinâmica utilizada no jogo RTS.

Em relação à análise tática e tomada de decisões, [Cheng & Thawonmas, 2004] analisam a possibilidade de utilização de *case-based planning* (planejamento baseado em casos, técnica que consiste na reutilização de planos anteriores onde se obteve sucesso para solucionar novos planos) para tentar diminuir a previsibilidade de personagens não jogáveis (NPCs, do inglês *Non-Playable Characters*). Já Sanchéz-Ruiz et al. [2007] apresentam uma abordagem que utiliza bases de conhecimento, numa arquitetura que combina *case-based planning* e conhecimento ontológico do ambiente de jogo.

Em [Hinrichs & Forbus, 2007], os autores propõem uma forma de alocação de

[1] http://dl.acm.org/, acessado em outubro de 2010.

[2] http://ieeexplore.ieee.org/Xplore/guesthome.jsp, acessado em outubro de 2010.

[3] Pesquisas realizadas em outubro de 2010.

recursos através de uma combinação de analogia estrutural, experimentação e modelagem qualitativa para otimizar a produção de comida em um jogo de estratégia. [Bergsma & Spronck, 2008] propõem uma arquitetura para um agente de IA para jogos TBS chamada Adapta, baseada na decomposição de tarefas usando alocação de recursos, e promove o uso de técnicas de aprendizado de máquina.

Como mencionado no Capítulo 1, em jogos, um fator de grande importância para uma IA é que se mantenha o nível de desafio para jogadores com diferentes níveis de experiência. Potisartra & Katrajaras [2009] propõem um protótipo de IA bastante interessante, capaz de jogar uma partida tão bem quanto seu adversário, para que as partidas sejam sempre equilibradas. Outro trabalho interessante seguindo essa linha de pesquisa é [Salge et al., 2008], no qual foi utilizada uma IA evoluída através de um algoritmo genético para prover uma inteligência que permita um jogo mais profundo e divertido.

Dentre trabalhos que têm o foco em jogos comerciais, podemos citar [Wender & Watson, 2008], no qual é descrito um agente inteligente para a tarefa de seleção de locais de fundação de cidades no jogo *Civilization* IV através de uma abordagem de aprendizado de máquina. O mesmo jogo foi utilizado por [Spronck & Teuling, 2010] e [Machado et al., 2011] na construção de modelos para os jogadores controlados por IA e suas preferências.

2.4 Ferramentas para IA em Jogos TBS

Durante as pesquisas iniciais para este trabalho, procuramos avaliar algumas ferramentas onde poderiam ser realizados experimentos de IA em jogos TBS. A dificuldade em encontrar uma ferramenta realmente adequada a modificações e implementações na IA foi, inclusive, a principal motivação para esse trabalho.

De uma forma geral, percebemos que esta é uma grande lacuna nas pesquisas da área, pois há poucos jogos comerciais onde é possível fazer modificações de forma satisfatória e com complexidade e liberdade necessárias. Já nos jogos de código aberto, normalmente não é uma tarefa fácil alterar apenas aspectos da IA sem que sejam

necessárias modificações em outros componentes do jogo. Nesta seção apresentaremos três ferramentas que foram analisadas, e os pontos fortes e fracos encontrados em cada uma delas.

2.4.1 *Civilization* SDK

Dentre os jogos comerciais do gênero TBS, a série *Civilization* criada por Sid Meier é provavelmente o exemplo mais conhecido. O primeiro jogo da série – *Sid Meier's Civilization* – foi desenvolvido pela *MicroProse*® e lançado em 1991 para diversas plataformas, como Amiga 500, DOS e Macintosh. O mais recente – *Sid Meier's Civilization* V – foi desenvolvido pela *Firaxis*® e lançado em novembro de 2010. Uma característica interessante é que a série sempre foi bastante utilizada para criação de modificações, e a partir dessa demanda a *Firaxis*®, empresa atualmente responsável pelo desenvolvimento do jogo, passou a disponibilizar um SDK (pacote de desenvolvimento de software, do inglês *software development kit*) oficial desde o penúltimo jogo da série, *Civilization* IV, lançado em 2005.

Por se tratar de um SDK oficial para modificações em um jogo comercial, esta é uma ferramenta bastante robusta e tem como base um jogo bastante completo e complexo, além de uma grande base de jogadores. A adição de novos conteúdos, como novos mapas ou unidades, é bastante simples, e o mesmo vale para a alteração de certos recursos e regras do jogo. No entanto, como o código do jogo é proprietário, nem todos os componentes estão disponíveis para modificação, e o núcleo da IA é um deles.

No SDK do jogo *Civilization* V existem vários arquivos XML[1] que definem certos pesos e preferências que são utilizados pelas IAs do jogo, e alguns algoritmos em scripts LUA[2], voltados principalmente para a geração procedural dos mapas do jogo e outras funções utilitárias da IA, como escolha de locais de fundação de cidades. Esses

[1] *Extensible Markup Language*, linguagem de marcação bastante utilizada em diversos mecanismos e situações, inclusive em arquivos de configuração como no jogo *Civilization* V.

[2] Linguagem de *script* extensível e bastante leve projetada para aplicações em geral e que é utilizada em diversos jogos comerciais como *Grim Fandango* (1998), *Far Cry* (2004) e *Angry Birds* (2009), além do próprio *Civilization* V.

arquivos – tanto os XML quanto os *scripts* em LUA – podem ser personalizados com facilidade para proporcionar algumas modificações na IA do jogo, principalmente na modelagem de preferências das IAs do jogo. No entanto, como o jogo possui código proprietário, não é possível ver ou alterar os algoritmos presentes no núcleo da IA, utilizados internamente no jogo, o que reduz bastante a possibilidade de exploração do SDK para pesquisas de IA. Uma última desvantagem é que, mesmo nas alterações possíveis, todos os jogadores de IA são afetados e não é possível utilizar diferentes configurações em uma mesma execução do jogo, o que torna difícil comparar o efeito que as alterações tiveram em situações de jogo real.

O SDK do jogo *Civilization* IV já foi utilizado em diversos trabalhos, como em [Wender & Watson, 2008], relacionado à escolha de locais para criação de cidade, e [Spronck & Teuling, 2010] e [Machado et al., 2011], relacionados à modelagem do comportamento adotado pela IA no jogo. Como o jogo *Civilization* V ainda é muito recente (lançado em 2010), ainda não temos conhecimento de trabalhos acadêmicos que utilizam seu SDK.

2.4.2 FreeCiv

Após a análise das possibilidades de modificações em um jogo de código proprietário e perceber que haveria certa limitação nas possibilidades de exploração, decidimos então analisar um jogo de código aberto onde não haveria este problema, pois este permitiria qualquer tipo de modificação que fosse necessária. O jogo TBS de código livre mais conhecido é o *Freeciv*[1], voltado principalmente às plataformas GNU/Linux (na qual é inclusive incluído em algumas distribuições) e Sun Solaris, embora haja versões para outras plataformas.

O projeto do jogo *Freeciv* foi iniciado em novembro de 1995 por três estudantes do curso de Ciência da Computação da Universidade de Aarhus, na Dinamarca. A primeira versão – 1.0 – foi lançada em janeiro de 1996, e a última versão disponível na data em que este trabalho foi desenvolvido – 2.3.0 – foi lançada em agosto de 2011. As regras do jogo foram fortemente baseadas nas regras o primeiro jogo da série

[1] http://freeciv.wikia.com/wiki/Main_Page, acessado em 17/06/2012

Civilization, lançado em 1991 e influenciadas pelo segundo jogo da série, *Civilization* II, lançado em 1996. Dentre os principais recursos disponíveis estão a possibilidade de até 126 jogadores em uma mesma partida e a possibilidade de jogos entre jogadores humanos (através da Internet ou rede local) ou contra jogadores controlados por IA.

Um ponto que se mostrou bastante positivo foi a atuação da comunidade de desenvolvedores, que possui uma lista de discussão[1] onde as perguntas levantadas são respondidas rapidamente. Para uma avaliação mais completa da ferramenta e também para que pudesse discutir as dúvidas com a comunidade, participamos da lista de discussões por cerca de 40 dias, no período entre 25/05/2010 e 05/07/2010, período no qual também analisamos a documentação existente e o código-fonte.

Devido à idade do projeto, à linguagem de programação utilizada (linguagem C) e a certas decisões arquiteturais tomadas ao longo do projeto, a compreensão e manutenção do código-fonte são tarefas difíceis. Além disso, os códigos relacionados à IA não estão isolados de outros componentes do jogo, dificultado a localização dos pontos onde se deve alterar a IA, por exemplo as funções de automação das tarefas dos colonos, que está no núcleo do jogo mas é utilizada pela IA para tomar as decisões com estas unidades. Outro ponto negativo é que o jogo não foi concebido para permitir diversas IAs simultâneas em uma mesma partida. Há alguns projetos para evoluções nesse sentido, tanto um projeto para tornar a IA um módulo à parte quanto outro que permitiria partidas entre diferentes IAs. No entanto, até o momento em que finalizamos a análise e acompanhamos as listas de discussão, nenhum dos dois estava em desenvolvimento pela comunidade nem havia sinais de que seriam concluídos em breve.

Em um e-mail enviado à lista de discussões da comunidade desenvolvedora, apresentamos o desejo de entender e trabalhar em alguns algoritmos e técnicas com o objetivo de melhorar a IA existente no jogo atualmente. As respostas recebidas, no entanto, não foram muito encorajadoras quanto a realizar modificações na IA. Um dos desenvolvedores sugeriu que a IA fosse literalmente jogada fora e construída novamente devido ao número de mudanças pelas quais o jogo passou ao longo do

[1] freeciv-dev@gna.org

tempo e que tornaram o código relacionado à IA muito obsoleto. Com as respostas recebidas e a análise das documentações e do código-fonte, percebemos que embora o jogo seja estável, mantenha uma comunidade de desenvolvedores ativa e permita em teoria realizar qualquer mudança desejada na IA, o esforço necessário para essas mudanças é muito grande quando comparado ao que seria estritamente necessário para a implementação dos algoritmos desejados.

Apesar disso, cabe mencionar que o jogo *Freeciv* já foi utilizado para experimentos em IA em diversos trabalhos, como [Forbus & Hinrichs, 2006], que utilizaram o jogo para validação dos experimentos de um sistema colaborador e [Jones & Goel, 2009], que utilizaram o jogo para criação de redes de abstração para análise da situação das cidades do jogo.

2.4.3 C-Evo

Com base nas principais dificuldades enfrentadas no uso do *Freeciv*, percebemos que seria mais interessante encontrar uma ferramenta onde fosse possível trabalhar apenas na IA, sem necessidade de entendimento ou alteração de códigos não relacionados à pesquisa em si. Essa busca nos levou ao jogo *C-Evo*[1], criado por Steffen Gerlach. *C-Evo* é um jogo gratuito baseado no *Civilization* II, criado inicialmente com o propósito de corrigir algumas das falhas deste jogo, principalmente a fraca IA. O jogo foi escrito em Delphi e a primeira versão foi disponibilizada em 1999.

A grande vantagem dessa plataforma vem da arquitetura projetada para uma IA bastante forte e totalmente modular. A interface utilizada pela IA é aberta, o que permite a criação de uma IA totalmente nova em uma DLL (biblioteca de vínculo dinâmico, do inglês *dynamic-link library*) e incorporá-la ao jogo. Como foi criado com esse objetivo, o jogo permite nativamente a competição entre diferentes IAs, e no site oficial há diversas IAs disponíveis que podem ser utilizadas para efeitos de comparação.

A documentação existente é razoavelmente boa, e a flexibilidade para implementação dos algoritmos é grande. Os principais pontos negativos encontrados

[1] http://www.c-evo.org, acessado em 17/06/2012

podem ser associados à linguagem utilizada e à idade do projeto. A interface disponível para a IA é fortemente baseada em estruturas de baixo nível, o que torna o entendimento do código um pouco confuso. Outro ponto é que o núcleo do jogo não é executado em uma estrutura segura, o que permite que uma IA utilize de trapaças, por exemplo através da leitura e gravação de determinados endereços de memória que são utilizados pelo núcleo do jogo. É claro que isso não invalida as IAs construídas de forma "correta", mas esse fator, juntamente com o fato de o código não ser muito moderno e não apresentar muitos recursos de alto nível, motivaram a criação do *framework* desenvolvido neste trabalho. É interessante notar aqui que boa parte das características positivas encontradas no *C-Evo* foram incorporadas ao *framework*, buscando corrigir os seus pontos fracos.

Dentre os trabalhos que utilizaram o *C-Evo*, destacamos [Sanchéz-Pelegrín et al., 2005] e [Díaz-Agudo et al., 2005], nos quais os autores desenvolveram uma IA para o jogo baseada em *case-based reasoning* (raciocínio baseado em casos, de forma semelhante ao que já foi explicado sobre *case-based planning*) e criaram um módulo de decisão inteligente.

Capítulo 3

O jogo Território

Este capítulo apresenta o jogo Território, criado para ser usado juntamente com o *framework*. Os conceitos e regras do jogo foram definidos de forma que o jogo possua todos os principais elementos presentes em outros jogos TBS. Com isso, pretende-se que todos os experimentos realizados através do *framework* sejam aplicáveis a outros jogos TBS e, da mesma forma, os algoritmos aplicados em outros jogos TBS sejam aplicáveis ao jogo Território.

Aqui não entraremos em detalhes acerca do uso do *framework* ou da sua implementação, o que será explicado no próximo capítulo. Aqui estamos interessados nos conceitos principais, elementos do jogo, ações que podem ser executadas e as regras envolvidas. Nos casos onde houver alguma característica única (não existente em outros jogos TBS que temos conhecimento), mencionaremos os motivos de tal decisão.

3.1 Visão geral

O jogo Território é disputado em um mapa bidimensional entre dois ou mais jogadores. Cada jogador deve controlar unidades (ou tropas) e construções (como cidades ou quartéis) com o objetivo de eliminar completamente todas as unidades e construções dos adversários. Cada espaço (ou célula) do mapa é chamado de território, e cada unidade ou construção de um jogador ocupa um território. As construções são centros de produção, onde o jogador pode criar novas unidades ou itens, usados para equipar unidades. As unidades são as entidades que o jogador pode movimentar pelo mapa para

localizar os inimigos, atacar, criar novas construções, etc.

Assim como nos demais jogos TBS, o jogo é dividido em turnos, nos quais os jogadores alternam a vez de jogar. Na vez de cada jogador, ele escolhe quais ações tomar (e.g. mover uma unidade e iniciar produção de um item em uma construção), e ao final da sua vez passa-se à vez do próximo jogador. O processo é repetido até que restem apenas elementos de um jogador no mapa.

3.2 Conceitos básicos

Antes de iniciarmos a apresentação dos conceitos básicos do jogo, vamos explicar porque o jogo recebeu o nome de "Território". A principal razão é devido a uma das características únicas do jogo, o domínio de territórios. Esse conceito será explicado em detalhes na Seção 3.2.2, mas em linhas gerais temos que os jogadores podem obter domínio sobre os territórios, e com isso obter bônus em certas ações do jogo. Assim, o domínio de territórios se torna um dos aspectos mais importantes nas decisões estratégicas do jogo, dando origem ao nome do jogo: Território.

Os conceitos do jogo Território são bastante semelhantes a outros jogos do gênero TBS. Cada jogador controla unidades e construções e o objetivo é eliminar todas as unidades e construções dos demais jogadores. O último jogador a sobreviver (ou seja, aquele que ainda possuir unidades e/ou construções) é considerado o vencedor.

Assim como nos demais jogos TBS, os jogadores alternam entre si a vez de jogar, sendo que na sua vez é possível realizar diversas ações em sequência. O jogador, no entanto, não é obrigado a executar nenhuma ação em sua vez, podendo apenas passar a vez ao próximo jogador se considerar essa ação vantajosa para sua estratégia. Após todos os jogadores executarem suas jogadas na vez de cada um, é encerrado o turno e a vez passa novamente ao primeiro jogador. Caso o jogador da vez já tenha sido eliminado, passa-se a vez ao próximo jogador seguindo a ordem inicial dos jogadores, e isso é feito até que se encontre um jogador que ainda esteja vivo na partida, ou seja, não tenha sido eliminado.

Para o início da partida, cada jogador recebe uma unidade e uma construção. A localização inicial pode ser personalizada de acordo com o tipo do experimento a ser realizado. Por padrão, os elementos dos jogadores são colocados em um dos quatro cantos existentes no mapa.

Os territórios são os espaços (ou células) do mapa onde são posicionadas as unidades e construções ao longo da partida. Cada território do mapa pode receber apenas uma construção, mas é possível que haja múltiplas unidades em um mesmo território, desde que todas pertençam ao mesmo jogador. Além disso, não é possível que haja unidades de um jogador em um território onde haja uma construção de outro jogador.

As unidades são as entidades que atuam diretamente sobre o mapa, podendo executar ações como movimentação e ataque, enquanto as construções são elementos estáticos, capazes apenas de produzir elementos novos para o jogador ao qual ela pertence. Nas construções é possível produzir novas unidades ou itens que podem ser equipados nas unidades para melhorar determinadas características destas. Os itens que equipam as unidades não executam nenhuma ação, pois têm comportamento apenas passivo alterando determinadas características da unidade na qual está equipado, de acordo com o tipo do item.

De maneira geral, o jogo não possui uma variedade muito grande de unidades e itens. A inspiração principal para essa opção vem de jogos como Combate (lançado no Brasil pela Estrela, que também conhecido como *Stratego*, nome original em inglês), nos quais há um número limitado de tipos de unidade e mapas de tamanho reduzido. Como o jogo Território possui também a possibilidade de equipar itens nas unidades, unidades do mesmo tipo que possuam itens diferentes apresentam características diferentes, o que aumenta a diversidade e torna a exploração de estratégias mais interessante. A seguir serão explicados em maiores detalhes alguns aspectos do jogo.

3.2.1 Mapas Hexagonais

Como mencionado anteriormente, Território utiliza um mapa bidimensional. Ao contrário da grande maioria dos jogos TBS, no entanto, Território utiliza uma grade hexagonal ao invés de uma grade quadrada. Esta representação não é uma novidade no

mundo dos jogos, pois vários jogos já a utilizam desde que a companhia norte-americana *Avalon Hill* lançou a segunda edição do jogo de tabuleiro "Gettysburg"[1], em 1961. Nos jogos eletrônicos, podemos citar como exemplos recentes o jogo de código aberto "A Batalha por Wesnoth" (do inglês, *The Battle for Wesnoth*)[2] e o *Civilization* V, lançado pela *Firaxis* em 2010[3].

A grande vantagem dos mapas hexagonais quando comparados aos mapas quadrados é que a distância do centro de uma célula para o centro das células adjacentes é constante. Isso se deve ao fato de que nos mapas quadrados, para cada célula há quatro células vizinhas nos lados e quatro células vizinhas nas quinas. Com isso, a distância entre o centro de uma célula até o centro das células vizinhas através de um vértice é maior que a distância até o centro das células vizinhas através de uma aresta. Já em mapas hexagonais, os vizinhos sempre compartilham uma aresta, o que faz com que a distância do centro de uma célula até o centro de todas as células vizinhas seja constante.

Assim, a utilização de mapas hexagonais traz benefícios em aspectos do jogo que têm alguma relação com distâncias no mapa, especialmente a movimentação de unidades. A Figura 3.1 mostra um exemplo de mapa hexagonal com seis linhas e oito colunas. É importante notar que, embora o eixo y esteja alinhado verticalmente, o mesmo não ocorre com o eixo x, que faz um zigue-zague, como pode ser visto na cor de fundo destacada para as linhas pares.

Por outro lado, uma desvantagem na representação através de mapas hexagonais é nas coordenadas utilizadas para movimentação nas diagonais, pois há diferença entre células que estão em colunas pares e as que estão em colunas ímpares. Ainda sobre a Figura 3.1, vamos tomar como exemplo duas células do mapa, [1,1] e [2,2], e seja necessário obter as coordenadas das células vizinhas localizadas a nordeste de cada uma delas. No caso da primeira célula, a posição desejada é [2,1], enquanto no caso da segunda célula é [3,1]. Podemos perceber que, no primeiro caso, a mudança foi apenas

---

[1] http://en.wikipedia.org/wiki/Gettysburg_(game) – acessado em 17/04/2012

[2] http://pt.wikipedia.org/wiki/The_Battle_for_Wesnoth – acessado em 17/04/2012

[3] http://www.civilization5.com/ – acessado em 26/05/2012

de uma unidade no eixo x, o que podemos representar por um vetor (1,0), enquanto na segunda célula a mudança é representada, analogamente, por (1,-1). Essa necessidade de tratamento diferenciado para colunas pares e ímpares torna o planejamento de rotas e cálculo de distâncias mais complexo em mapas hexagonais do que em mapas quadrados, nos quais o vetor de deslocamento é sempre o mesmo para uma determinada direção.

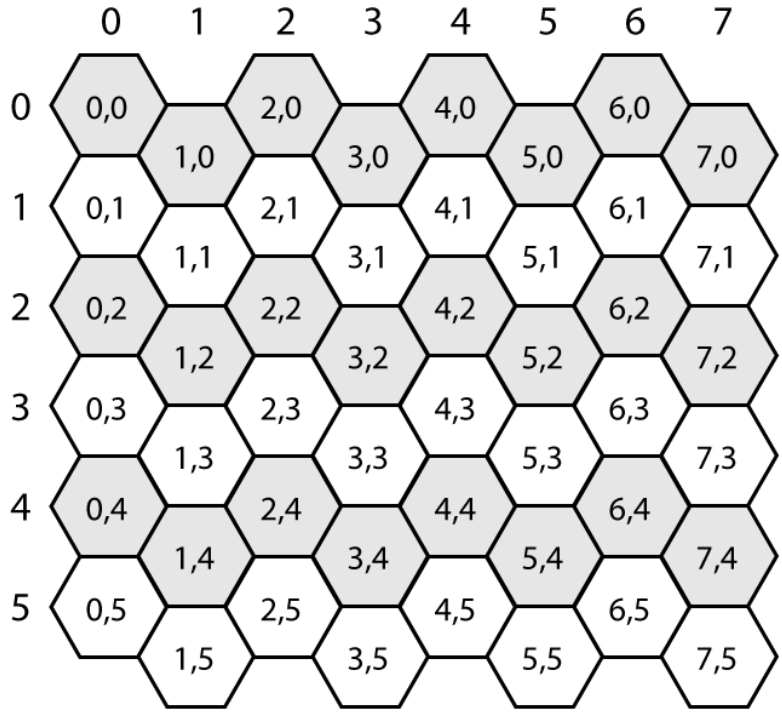

Figura 3.1: Representação de um mapa hexagonal.

Com base no que identificamos como vantagens e desvantagens nos mapas com células hexagonais, consideramos que a adoção destas apresenta vantagens que superam as desvantagens, considerando o objetivo principal do jogo, que é a utilização para pesquisa de novas técnicas de IA em jogos TBS.

3.2.2 Domínio de territórios

No que se refere à inovação em relação a outros jogos do gênero, a característica que pode ser considerada como a mais marcante no jogo Território é o domínio dos territórios do jogo pelos jogadores. Como já foi citado anteriormente, essa é a razão

para que o jogo fosse batizado com o nome Território.

No jogo, cada território pode estar neutro (sem nenhum domínio) ou dominado por um ou mais jogadores. Inicialmente todos os territórios do jogo são considerados neutros, e os jogadores podem conquistar domínio sobre os territórios movendo as unidades estrategicamente ou executando o comando de influenciar território utilizando alguma unidade, o que será explicado na Seção 3.4.4. Um território do mapa possui uma matriz de influência, que representa quanto determinado jogador possui do domínio do território em questão. Se apenas um jogador possuir influência no território, pode-se afirmar que o território está 100% dominado por este jogador. Caso dois jogadores possuam influências 0,6 e 0,4 sobre um território, isso é o mesmo que afirmar que eles possuem 60% e 40% de domínio sobre este território, respectivamente.

Como forma de simplificar a visualização da situação atual de domínio de um território, os valores são sempre normalizados para que a soma das influências de determinado território seja sempre igual a 1. Vejamos um exemplo em que um território possui 0,75 de domínio do jogador A e 0,25 do jogador B, mas o jogador B esteja influenciando o território em mais 0,25. Se somarmos esse valor sobre o domínio atual do jogador B sobre o território, teremos o valor de 0,5. No entanto, a soma dos domínios deste território (0,75 + 0,5) passaria a ser 1,25. Ao efetuarmos a normalização, teremos que o domínio do jogador A passaria a ser 0,6 (0,75 / 1,25), enquanto o domínio do jogador B passará a ser 0,4 (0,5 / 1,25).

Além da influência direta exercida através de comandos executados pelas unidades, o que será mais bem detalhado nas próximas seções do documento, as unidades também exercem influência indireta sobre os territórios onde elas permanecem ao fim de cada turno. Quando um turno é finalizado, cada unidade executa certa influência sobre o domínio do território em que se encontra, assim como em todos os seus vizinhos. Essa característica torna bastante atraente a exploração de estratégias de rápida expansão. Para que isso não se tornasse um problema no jogo, foi criada uma regra na qual as unidades que permanecem unidas recebem bônus especiais, o que será descrito na próxima seção.

3.2.3 Agrupamento de unidades

Outro aspecto que pode ser considerado inovador no jogo Território – embora não tanto quanto o domínio de territórios – é o fato de que unidades sobre uma mesma célula recebem um bônus em algumas de suas características, como ataque ou defesa, permitindo que tenham desempenho melhor que se estivessem isoladas. Não temos conhecimento desse conceito em outros jogos TBS.

No *Civilization* V há um conceito que pode ser considerado parecido, onde é possível obter uma habilidade chamada "Disciplina" para todas as unidades do jogador, e assim elas passam a ter ataque 10% mais forte quando há outras unidades amigas em territórios adjacentes. No nosso caso, no entanto, ao invés de um bônus para unidades próximas foi criado um bônus para unidades que estão compartilhando um mesmo território do jogo (ou seja, uma célula do mapa). Além disso, o bônus aplicado é de 50% sempre que houver mais de uma unidade no território. É importante esclarecer que a regra vale apenas quando houver mais de uma unidade no território, independente de qual é o número real de unidades, ou seja, não há aumento do bônus quando há três, quatro, cinco ou dez unidades.

Do ponto de vista estratégico, essa regra cria novas possibilidades para os jogadores, sendo possível adotar estratégias mais expansionistas ou defensivas. É interessante também que, em conjunto com o conceito de domínio de território, essa regra pode servir para não deixar estratégias de rápida expansão fortes demais, o que tornaria o jogo desbalanceado. Dessa forma, o jogador precisa balancear as necessidades de domínio de território com a maior segurança advinda do fato das unidades estarem sempre agrupadas.

3.2.4 Tempo para jogadas

Um dos fatores que torna menos atraentes as partidas multijogador em jogos TBS é a duração das jogadas de cada jogador, especialmente jogadores humanos. Considerando esse fator, é desejável que um jogador controlado por uma IA também não utilize tempo demais, pois do contrário, quando a IA for utilizada em partida contra humanos, o jogo se tornará moroso e desagradável.

Por esse motivo, o jogo conta com um tempo limite para que as jogadas sejam efetuadas. Se ao final do tempo o jogador ainda não houver terminado sua jogada, ele não poderá mais jogar no turno e será a vez do próximo jogador. Por padrão, esse tempo foi definido em 30 segundos, mas pode ser alterado para valores maiores ou menores de acordo com as necessidades de pesquisa apresentadas.

Em partidas multijogador, o tempo limite para jogadas já é utilizado há vários anos. *Civilization* II, lançado em 1996, permite que em partidas *online* os jogadores definam se haverá um tempo limite para que as jogadas sejam efetuadas. Em alguns casos, a utilização de tempos reduzidos é inclusive uma característica que favorece a competitividade e exige maior agilidade dos jogadores.

3.3 Elementos do jogo

No início do capítulo, os elementos do jogo como terrenos e unidades foram apresentados de forma superficial, para que fosse possível compreender de forma geral o funcionamento do jogo e as principais regras envolvidas. Nesta seção apresentaremos em maiores detalhes esses elementos, características que possuem, quando podem ser utilizados etc.

3.3.1 Territórios

Os territórios são os espaços que compõem o mapa do jogo. Cada território equivale a uma célula do mapa, e sobre os territórios são realizadas todas as ações do jogo, como movimentação de unidades e ataques. É possível haver apenas uma construção por território, mas é permitido haver várias unidades de um mesmo jogador sobre um mesmo território. Outro atributo dos territórios é o domínio dos jogadores, o que já foi explicado nas seções anteriores do capítulo.

Assim como em todos os jogos TBS que temos conhecimento, cada território possui um tipo de terreno, o que traz características distintas para os territórios. Os tipos de terreno existentes no jogo Território são grama, areia, floresta e água. Cada

terreno possui características distintas, de acordo com os seguintes atributos:

- Custo de movimentação: Representa o custo para que uma unidade se movimente até o território;
- Possibilidade de receber unidades terrestres: Indica se o território poderá receber unidades terrestres. Embora por razões de tempo não tenham sido criadas unidades que não fossem terrestres, o atributo foi mantido para visando maior facilidade de expansão ou adaptação do *framework* de acordo com a necessidade das pesquisas que vierem a utilizá-lo, uma vez que o jogo já estava mais preparado a considerar estes fatores;
- Possibilidade de receber unidades marinhas: Indica se o território poderá receber unidades marinhas. Conforme citado no atributo acima, inicialmente não foram definidas unidades não terrestres e o atributo não é utilizado na versão atual do jogo;
- Possibilidade de receber unidades aéreas: Indica se o território poderá receber unidades aéreas. O comportamento é análogo ao atributo acima;
- Possibilidade de receber construções: Indica se o território poderá receber construções.

A Figura 3.2 mostra a configuração de atributos para cada tipo de terreno do jogo. Foram criados três tipos de terreno sobre os quais as unidades podem movimentar (lembrando que todas as unidades são terrestres), cada um com um custo de movimentação diferente para que seja possível explorar algoritmos de movimentação e planejamento de rotas.

Tipo de terreno	Custo de movimentação	Unidades terrestres	Unidades marinhas	Unidades aéreas	Construções
Grama	1	Sim	Não	Sim	Sim
Areia	2	Sim	Não	Sim	Sim
Floresta	3	Sim	Não	Sim	Não
Água	1	Não	Sim	Sim	Não

Figura 3.2: Atributos dos tipos de terreno.

3.3.2 Unidades

As unidades são o principal elemento do jogo, pois é através delas que o jogador atua sobre o mapa, executando as ações em busca da vitória. Existem unidades de diversos tipos, cada uma com características particulares. Além disso, cada unidade pertence a apenas um jogador, e não é possível transferi-la para outro.

As unidades possuem atributos fixos e variáveis. Os atributos fixos são as características herdadas de acordo com o tipo da unidade, como ataque e quantidades de movimentos que pode realizar por turno. Já os atributos variáveis representam o estado corrente da unidade, como a posição da unidade no mapa e a quantidade de movimentos que ainda pode realizar no turno. Na lista abaixo são apresentados todos os atributos variáveis e a descrição de cada um deles:

- Posição: Indica qual a posição da unidade no mapa, com as coordenadas X e Y;
- Saúde: Representa a saúde da unidade através de um valor contínuo em que o valor mínimo é zero e o valor máximo varia de acordo com o tipo da unidade. Se a saúde de uma unidade atinge o valor zero a unidade é eliminada do jogo. A cada turno em que a unidade não executar nenhuma ação, a saúde dela é recuperada em uma pequena quantidade;
- Movimentos restantes: Representa a quantidade de movimentos que a unidade ainda pode realizar no turno corrente, apenas com valores inteiros. Se o valor for zero, a unidade não pode executar mais nenhuma ação no turno;
- Último turno movimentado: Indicação do último turno que a unidade executou alguma ação. Os turnos são indicados no jogo através de números inteiros, onde o primeiro turno é zero e os turnos seguintes vão incrementando o valor em uma unidade. Essa informação não influencia nas regras do jogo, mas foi criada como forma de auxiliar os jogadores no planejamento e identificação das ações executadas.

Como mencionamos acima, os atributos fixos das unidades são herdados de acordo com o tipo da unidade. Por esse motivo, unidades do mesmo tipo apresentam sempre os mesmos atributos fixos. No entanto, cada unidade pode ser equipada com

itens que trazem bônus aos seus atributos, fazendo com que a unidade atue usando atributos com valores modificados em relação ao valor original do seu tipo. Os itens serão explicados mais à frente na próxima seção. A lista abaixo apresenta os atributos fixos e o significado de cada um deles para a unidade:

- Saúde máxima: Representa o valor máximo da saúde que a unidade poderá receber;
- Movimentos: Representa a quantidade de movimentos que a unidade pode realizar por turno;
- Ataque: Representa o poder de ataque da unidade, em valores inteiros positivos;
- Defesa: Representa o poder de defesa da unidade, em valores inteiros positivos;
- Alcance: Representa o alcance das ações da unidade, em valores inteiros positivos. Um alcance igual a um indica que a unidade só pode realizar ações até os territórios vizinhos, ou seja, com distância igual a um. Um alcance igual a dois indica que a unidade pode realizar ações até os territórios vizinhos e os vizinhos dos vizinhos, ou seja, com distância igual a dois. O mesmo comportamento entendido para valores maiores, sempre de acordo com a distância a partir do território de origem da unidade;
- Visão: Representa o alcance de visão da unidade, em valores inteiros positivos. O comportamento é semelhante ao atributo de alcance, mas representa a distância que a unidade é capaz de enxergar o mapa, e não a distância que ela é capaz de executar ações;
- Fator de influência: Representa a capacidade que a unidade tem de influenciar os territórios ao seu redor, de acordo com o conceito de domínio de territórios que já foi apresentado. O fator de influência é um valor contínuo entre zero e um;
- Fator aleatório mínimo: O fator aleatório mínimo é utilizado para calcular o valor real de um atributo da unidade no momento em que este for utilizado, e pode ser entendido como a precisão da unidade. É representado através de um valor contínuo entre zero e um. O comportamento desse atributo será explicado na Seção 3.4;

- Custo de produção: Representa o custo para que a unidade seja produzida em uma construção. A produção de elementos do jogo será explicada nas próximas seções.

Foram criados nove tipos de unidades, sendo que uma é voltada para auxiliar no domínio dos territórios (ataque e defesa mais baixos, mas fator de influência alto), enquanto outras oito são voltadas para aspectos militares (ataque e defesa mais altos, mas fator baixo). Na Figura 3.3 podemos ver os atributos definidos para cada tipo de unidade.

As ações disponíveis para o jogador realizar com as unidades são movimentar, atacar e influenciar território. As regras envolvidas em cada ação serão descritas nas próximas seções.

Tipo de unidade	SM	MV	AT	DF	AC	VI	FI	FA	CP
Sacerdote	3	2	0	1	1	2	0,50	0,60	10
Soldado	1	2	1	1	1	1	0,00	0,45	2
Sargento	2	1	1	1	2	2	0,00	0,45	3
Tenente	4	1	1	2	1	1	0,00	0,50	5
Capitão	5	1	2	1	1	1	0,00	0,50	8
Major	6	2	4	2	1	1	0,00	0,55	13
Coronel	10	1	6	5	1	1	0,00	0,55	21
General	15	2	9	7	2	3	0,05	0,60	30
Marechal	20	3	12	9	2	2	0,10	0,70	40

SM: Saúde máxima
MV: Movimentos
AT: Ataque
DF: Defesa
AC: Alcance
VI: Visão
FI: Fator de influência
FA: Fator aleatório mínimo
CP: Custo de produção

Figura 3.3: Atributos dos tipos de unidade.

3.3.3 Itens

Os itens são elementos do jogo que podem ser produzidos nas construções e, depois de produzidos, equipados em unidades que estejam no mesmo território da construção na qual eles foram produzidos. Uma vez equipados na unidade, um item possui efeito

passivo sobre um dos atributos fixos da unidade, e não pode mais ser removido da unidade ou transferido para outra unidade.

Tradicionalmente, os itens não costumam ser utilizados em jogos TBS, e a inspiração foi trazida dos RPGs, gênero de jogos onde o jogador cria e evolui um personagem através de melhores habilidades ou itens, à medida que executa tarefas ou ganha experiência. Já nos jogos TBS, há exemplos onde ao ganhar experiência as unidades conquistam novas habilidades, mas não temos conhecimento do uso de itens. Acreditamos que com essa característica, adiciona-se mais um elemento que influencia na estratégia dos jogadores, conferindo ao jogo uma nova dinâmica.

Uma unidade pode equipar mais de um item, desde que seja de um tipo que ainda não tenha sido equipado nela. O jogo conta com oito itens, cada um fornecendo bônus para um dos atributos fixos da unidade. A Figura 3.4 apresenta a lista dos itens, o atributo que é influenciado por cada um, em quanto o atributo da unidade será incrementado e o custo de produção de cada item.

Tomemos um exemplo como forma de ilustrar o funcionamento dos itens: seja uma unidade do tipo Coronel, equipada com os itens Espada e Capacete. De acordo com a Figura 3.4, o item Espada fornece um bônus de +3 sobre o ataque e o item Capacete fornece um bônus de 0,25 sobre o fator de influência. Dessa forma, o atributo ataque da unidade passará de 6 para 9, e o fator de influência passará de 0 para 0,25. A Figura 3.5 mostra os valores padrão para a unidade, o bônus oferecido por cada um dos itens e o valor final dos atributos da unidade.

Item	Atributo	Bônus	Custo de produção
Coração	Saúde máxima	+3	15
Sandália	Movimentos	+1	10
Armadura	Fator aleatório mínimo	+0,2	15
Espada	Ataque	+3	25
Escudo	Defesa	+2	20
Cinturão	Alcance	+1	20
Óculos	Visão	+1	10
Capacete	Fator de influência	+0,25	30

Figura 3.4: Bônus e custo de produção dos itens do jogo.

O efeito de um item só passa a valer a partir do momento em que ele for equipado

na unidade. Enquanto isso não é feito, os itens que já foram produzidos permanecerão guardados na construção que os produziu, até que sejam utilizados para equipar uma unidade. Acerca da produção de itens, explicaremos na próxima seção, que trata das construções.

	SM	MV	AT	DF	AC	VI	FI	FA
Atributos Coronel	10	1	6	5	1	1	0,00	0,55
Bônus espada			(+3)					
Bônus capacete							(+0,25)	
Coronel (totalizado)	10	1	9	5	1	1	0,25	0,55

SM: Saúde máxima
MV: Movimentos
AT: Ataque
DF: Defesa
AC: Alcance
VI: Visão
FI: Fator de influência
FA: Fator aleatório mínimo

Figura 3.5: Exemplo de uso de itens em uma unidade.

3.3.4 Construções

As construções são elementos do jogo que têm a função de produzir novas unidades ou itens para o jogador ao qual pertencem. Cada construção tem uma posição fixa e uma taxa de produção, o que determinará quantos turnos será necessário para que a produção seja concluída. Em todas as construções é possível produzir qualquer unidade ou item existente no jogo, não havendo nenhuma restrição em relação ao número de turnos que já tenham sido realizados, número de participantes ainda ativos ou eventos da partida. O tempo (em turnos) até que a produção seja concluída dependerá da taxa de produção da construção e do custo de produção do elemento. O custo de produção de cada unidade já foi apresentado na Figura 3.3 e o custo de produção dos itens na Figura 3.4. Detalhes sobre a definição e alteração da produção nas construções serão apresentados na próxima seção, que trata das ações do jogo.

Assim como as unidades, as construções também possuem atributos fixos e variáveis, sendo os primeiros herdados do tipo de construção e os segundos indicadores do estado atual da construção. Os atributos fixos das construções podem ser vistos na lista a seguir:

- Taxa de produção: indica quanto é produzido pela construção em cada turno, o que influencia no tempo (em turnos) para que se conclua a produção escolhida. Se uma construção possuir uma taxa de produção de 2/turno e estiver produzindo uma unidade do tipo Tenente, para a qual é necessária uma produção total de 5, após 3 turnos a unidade estará criada;
- Fator de influência: assim como o fator de influência das unidades, representa a capacidade que a construção tem de influenciar os territórios ao seu redor. O fator de influência das construções também é um valor contínuo entre zero e um. A grande diferença é que, como as construções são elementos estáticos no jogo, a influência se dará sempre nos mesmos territórios, que são adjacentes a ela.

Além dos atributos fixos, as construções possuem atributos variáveis, indicando o que está sendo produzido, quanto já foi produzido, sua localização e os itens existentes no inventário. Na lista abaixo podemos ver os atributos variáveis das construções:

- Posição: Indica qual a posição da unidade no mapa, com as coordenadas X e Y;
- Produção atual: Indica qual a produção atual da construção. É possível que seja uma unidade ou um item;
- Progresso da produção: Indica quanto já foi produzido até o momento da produção atual;
- Inventário de itens: É uma coleção onde são armazenados os itens que já foram produzidos pela construção e ainda não foram equipados em nenhuma unidade. A partir do momento que um item é equipado, ele é removido do inventário.

Construção	Taxa de produção	Fator de influência
Cidade	2	0,5
Quartel	3	0,0

Figura 3.6: Atributos dos tipos de construção.

Como podemos perceber pelos atributos, as construções são elementos do jogo que não possuem ataque nem defesa. Dessa forma, se uma unidade entrar em território

onde houver uma construção inimiga, a mesma será perdida, junto com os itens que estavam estocados nela aguardando que fossem equipados em uma unidade. Assim, se o jogador não desejar deixar suas construções vulneráveis, é recomendável que haja ao menos uma unidade no mesmo território em que está a construção. O jogo conta com dois tipos de construção, conforme apresentado na Figura 3.6.

3.4 Ações do jogo

A forma de atuação dos jogadores no jogo é através das ações, que podem ser executadas apenas pelo jogador que possuir a vez. É possível ao jogador movimentar suas unidades, atacar unidades inimigas que estejam ao alcance de suas unidades, equipar unidades com itens produzidos, influenciar territórios, definir e alterar a produção nas construções que possuir e criar novas construções. Ao final de sua jogada, o jogador deve também executar a ação de finalizar turno, informando o jogo que já encerrou suas jogadas para a vez.

O jogador é livre para executar as ações na ordem em que preferir, de acordo com a estratégia adequada. Obviamente, a única restrição é que as ações executadas devem ser válidas de acordo com as regras estabelecidas para cada uma delas. Os detalhes acerca de cada ação serão apresentados a seguir.

3.4.1 Movimentar

A movimentação de unidades é a ação mais utilizada no jogo, pois só através da movimentação é que o jogador poderá encontrar locais para executar outras ações como ataques ou criação de novas construções. Para executar uma movimentação, é necessário que o jogador forneça as seguintes informações:

1. Unidade: A unidade que o jogador quer movimentar, a qual deve possuir movimentos restantes (vide atributos das unidades);
2. Destino: As coordenadas do território no mapa para o qual a unidade deve ser movida, que deve ser diferente da posição na qual se encontra a

unidade.

Com base nas duas informações acima, o jogo irá verificar o custo de movimentação necessário até o destino escolhido, de acordo com o custo de movimentação de cada território na rota utilizada. Se o custo de movimentação até o destino for menor ou igual ao número de movimentos restantes da unidade, a movimentação é realizada com sucesso no jogo e o usuário recebe o mapa do jogo resultante da ação, para que possa prosseguir com as ações. Caso o custo de movimentação seja maior que o número de movimentos restantes da unidade, a movimentação não é realizada e o usuário é informado que o comando solicitado não pode ser executado.

Além da validação em relação ao custo de movimentação, também é necessário que o território de destino não possua unidades inimigas. Caso não haja unidades inimigas e o território de destino possuir uma construção inimiga, no momento da movimentação a construção destruída, removendo-a do território.

3.4.2 Atacar

Assim como na grande maioria dos jogos TBS, o ataque é a ação de maior importância em Território. É através do ataque que os jogadores podem eliminar unidades inimigas e, assim, vencer o jogo. Para executar um ataque, é necessário que o jogador forneça as mesmas informações que são exigidas no comando de movimentação, conforme explicamos a seguir:

1. Unidade: A unidade com a qual o jogador deseja atacar. Também é necessário que a unidade possua movimentos restantes (vide atributos das unidades);
2. Destino: As coordenadas do território no mapa que será atacado, que deve ser diferente da posição na qual se encontra a unidade.

Embora sejam necessárias as mesmas informações, as regras envolvidas são diferentes entre as duas ações. É necessário que o destino possua unidades inimigas e esteja dentro do raio de alcance da unidade que está efetuando o ataque. Caso haja mais de uma unidade inimiga no destino, a defesa será realizada pela unidade que possuir o atributo de defesa com maior valor, incluindo possíveis bônus fornecidos por itens

equipados.

Uma vez que a ação respeita todas as restrições do jogo, é feito então o cálculo do ataque e defesa para verificar o dano que cada unidade envolvida irá sofrer em sua saúde. Como exemplo, utilizaremos o cálculo para valor do ataque, mas o comportamento é o mesmo para o cálculo da defesa. Além do valor base do atributo (no caso, o ataque), são considerados também os bônus fornecidos pelos itens que a unidade possuir, o bônus por agrupamento (caso haja mais unidades no território em que a unidade se encontra), bônus por domínio do território e o fator aleatório mínimo, este último também levando em consideração bônus fornecidos pelos itens. A seguir apresentamos um exemplo de como o cálculo deve ser feito, supondo que a unidade de ataque seja do tipo Coronel, possua os itens Espada e Armadura, e esteja em um território que possua domínio de 60% do jogador e ainda possua outras unidades:

1. Em primeiro lugar, deve-se obter o valor do ataque da unidade considerando-se os bônus fornecidos por itens. No caso, o ataque original tem valor 6, e o item Espada fornece um bônus (+3), totalizando um ataque com valor 9;
2. Em seguida, como há outras unidades no território da unidade, acrescenta-se o bônus por agrupamento, que é de 50%, sobre o valor já agregado do ataque, o que representa um acréscimo de 4,5. Dessa forma, o ataque passa a contar com o valor 13,5;
3. Sobre o valor do ataque acumulado no passo 2, acrescenta-se também o bônus por domínio do território, que neste exemplo é de 60%, o que fornecerá um acréscimo de 8,1. Com isso, o valor total com o qual será obtido o ataque passa a ser de 21,6;
4. Por fim, é levado em consideração o fator aleatório mínimo, que no exemplo possui também um bônus fornecido pelo item Armadura, o que totaliza o valor 0,75. Dessa forma o ataque real utilizado pela unidade será calculado aleatoriamente, com o valor mínimo possível de $16{,}2$ $(21{,}6 \times 0{,}75)$ e o máximo de 21,6 conforme cálculo realizado no passo 3.

O cálculo da defesa funciona de maneira semelhante, e para completar o exemplo vamos supor que no território de destino do ataque haja apenas uma unidade, do tipo Marechal, que a unidade não possua nenhum item, e que o domínio inimigo sobre o

território seja de apenas 0,1. Com isso, a defesa da unidade, que originalmente é 9, será de 9,9. Através do fator aleatório mínimo (que para o Marechal é 0,7), temos então que a defesa terá valor mínimo de 6,93 ($9{,}9 \times 0{,}7$) e máximo de 9,9.

Apenas como demonstração, vamos supor que tanto o ataque quanto a defesa obtiveram, aleatoriamente, valores correspondentes ao ponto médio entre o mínimo e o máximo que seriam possíveis. Assim, o ataque real do Coronel será realizado com valor 18,9 e a defesa real do Marechal será realizada com valor 8,42 (trabalharemos com valores arredondados em no máximo duas casas decimais).

Dano provocado pelo ataque

$$\frac{ataque^2}{ataque + defesa}$$

Dano provocado pela defesa

$$\frac{defesa^2}{ataque + defesa}$$

Figura 3.7: Cálculo dos danos em um ataque.

A Figura 3.7 mostra como será calculado o dano causado pelo ataque à unidade de defesa e o dano causado pela defesa à unidade de ataque. Como podemos notar, no numerador é utilizado o quadrado do valor do ataque (ou defesa). Isso foi feito para aumentar os danos causados por batalhas, especialmente para as unidades mais fortes, tornando a partida mais dinâmica.

Com base nas fórmulas apresentadas na Figura 3.7, temos então que o dano causado pelo ataque será igual a $\frac{18{,}9^2}{18{,}9+8{,}42} = \frac{357{,}21}{27{,}32} = 13{,}08$. Já o dano causado pela defesa será igual a $\frac{8{,}42^2}{18{,}9+8{,}42} = \frac{70{,}9}{27{,}32} = 2{,}6$. Supondo ainda que as duas unidades contassem com a saúde máxima no momento do ataque, ou seja, o Coronel possuísse saúde igual a 10 e o Marechal saúde igual a 20, ao final do combate o Coronel receberia o dano de 2,6 e o Marechal o dano de 13,08. Finalizando o combate, então, o Coronel estará com a saúde de 7,4, enquanto o Marechal estará com a saúde de 6,92.

Além do mecanismo de ataque, esse exemplo também serviu para que vários aspectos do jogo fossem demonstrados. Podemos perceber que a combinação de domínio de territórios, itens e agrupamento de unidades fornece um mecanismo bastante interessante para balancear os combates, e é possível que até unidades

inicialmente mais fracas provoquem danos consideráveis em unidades mais fortes, podendo até mesmo derrotá-las.

Por fim, ao realizar um ataque, a unidade terá seus movimentos restantes reduzidos em um valor igual do custo de movimentação do território de destino do ataque. Assim, dependendo dos movimentos disponíveis para a unidade e do tipo de terreno onde o ataque foi realizado, é possível que a unidade realize mais de um ataque por turno. Em contrapartida, se a unidade não possuir movimentos restantes suficientes, o ataque não poderá ser feito no turno. A distância da unidade até o território de destino do ataque não influencia os movimentos restantes. Uma maneira mais fácil de entender essa questão é considerando que unidades com alcance baixo são unidades de combate corpo-a-corpo (como infantarias), enquanto unidades com alcance mais alto são unidades de cerco (como catapultas).

3.4.3 Equipar unidade

A ação de equipar uma unidade é bastante simples. Formalmente, é preciso que o jogador forneça as seguintes informações:

1. Unidade: A unidade que o jogador deseja equipar;
2. Item: O item que deseja equipar na unidade.

Para que seja possível equipar a unidade, a mesma deve se encontrar em um território onde haja uma construção, e na construção é necessário que o item escolhido esteja disponível no inventário. Conforme já foi explicado acerca dos itens, o item estará disponível no inventário da construção que o produziu até que seja equipado em alguma unidade. Não é possível transferir itens entre unidades, nem devolver à cidade um item que tenha sido equipado em uma unidade.

3.4.4 Influenciar território

A influência de território é uma ação com objetivo de aumentar o domínio do jogador sobre determinado território. Para executar a ação, é necessário que o jogador forneça as mesmas informações que o ataque, conforme listado abaixo:

1. Unidade: A unidade com a qual o jogador deseja influenciar o território. É necessário que a unidade possua movimentos restantes (vide atributos das unidades);
2. Destino: As coordenadas do território no mapa que será influenciado, que deve ser diferente da posição na qual se encontra a unidade.

Assim como para o ataque, é necessário que o destino esteja dentro do raio de alcance da unidade. No entanto, não é necessário que o território de destino possua unidades, podendo a influência ser realizada sobre qualquer território dentro do alcance da unidade, independente das unidades e da construção que nele estejam.

O bônus de agrupamento de unidades também é válido para a ação de influenciar território, assim como os bônus fornecidos pelos itens equipados na unidade. Para exemplificar a influência de território, vamos supor que uma unidade do tipo Sacerdote, não equipada com nenhum item, esteja num território cujo domínio é 20% do jogador e possua também outras unidades. A unidade irá influenciar um território no qual o domínio seja 100% de um jogador inimigo. Com o bônus de 20% devido ao domínio do território, a influência da unidade passará de 0,5 para 0,6. Além disso, com o bônus por agrupamento de 50%, a influência da unidade passará de 0,6 para 0,9. Por fim, a influência real será calculada considerando o fator aleatório mínimo, da mesma forma como é feito para o ataque. Temos, então, que o território de destino receberá a influência de no mínimo 0,54 (uma vez que o fator aleatório mínimo da unidade é de 0,6) e no máximo 0,9.

Vamos supor, então, que aleatoriamente tenha sido definido que a influência real será de 0,6. De acordo com o cálculo do domínio de territórios explicado na Seção 3.2.2, teremos então que o território passará a ter domínio de 37,5% do jogador, enquanto o inimigo passará a ter o domínio de 62,5%.

Assim como na ação de atacar, ao realizar uma influência sobre um território a unidade terá seus movimentos restantes reduzidos em um valor igual do custo de movimentação do território de destino da ação. Se a unidade não possuir movimentos restantes suficientes, a ação não poderá ser feita no turno.

3.4.5 Definir produção

A ação de definir a produção é utilizada para que o jogador escolha o que uma construção irá produzir nos próximos turnos. Para definir a produção para uma construção, é necessário que o jogador forneça as seguintes informações:

1. Construção: A construção na qual o usuário deseja definir a produção;
2. Produção: O elemento que será produzido, podendo ser uma unidade ou item.

A produção de unidades e itens pode ser feita em qualquer construção que o jogador possua. Caso a construção não esteja produzindo nada no momento, é iniciada a produção do elemento passado pelo jogador. Caso a construção já esteja produzindo alguma coisa, a produção será alterada para a desejada e todo o progresso da produção anterior será perdido. Caso a produção passada seja a mesma já em execução pela construção, o comando não terá efeito.

Conforme explicado acerca das construções, o tempo (quantidade de turnos) para que a produção seja concluída será dado de acordo com o ritmo de produção da construção e o custo de produção do elemento desejado. Se uma construção do tipo Quartel (produção de 3 por turno) estiver produzindo uma unidade do tipo Sacerdote, para a qual é necessária uma produção total de 10, após 4 turnos a unidade estará criada.

3.4.6 Criar construção

Para criar uma nova construção, o jogador deverá utilizar uma unidade do tipo Sacerdote, que criará a construção no território em que estiver localizada, e em seguida a unidade será perdida. Para criar uma construção, o jogador precisa fornecer as seguintes informações:

1. Unidade: A unidade que será utilizada para criar a construção;
2. Tipo de construção: O tipo de construção que será criado.

A criação de construções possui apenas três restrições: o território onde se encontrar a unidade não pode ter uma construção, a unidade deve ser do tipo Sacerdote

e o jogador que a possuir deve dominar o território onde ela estiver localizada (e consequentemente onde será criada a construção) em um valor superior a 50%.

3.4.7 Finalizar turno

A última ação disponível para o jogador é a finalização de sua jogada. Ao finalizar sua jogada, o jogador encerra sua atuação no turno e não lhe é permitido executar nenhuma nova ação até que seja novamente sua vez de jogar. Para finalizar sua jogada, o jogador não necessita ter executado nenhuma ação. Assim que essa ação é executada, o jogo informa ao próximo jogador que é sua vez de jogar e envia a disposição atual do mapa para que ele possa efetuar suas ações.

3.5 Limitações do Jogo

Com a explicação dos conceitos e regras do jogo Território, é necessário esclarecermos algumas limitações que podem ser percebidas quando comparado a outros jogos TBS.

A primeira dessas limitações é em relação às características que afetam estratégias globais. Em jogos mais simples, como *Battle for Wesnoth*, temos a quantidade de vilarejos controlados pelo jogador, que influenciam na quantidade de tropas que o jogador pode utilizar. Já na série *Civilization*, cujos jogos são bastante complexos, existem vários aspectos que afetam globalmente, como a construção das "maravilhas do mundo", que garantem privilégios exclusivos ao jogador que finalizar a construção. O jogo Território possui o domínio de territórios, que influencia nas habilidades das unidades controladas pelo jogador. No entanto, o domínio de territórios não influencia em outros aspectos do jogo, como a capacidade de produção das construções ou a quantidade de unidades que o jogador pode ter em um determinado momento. Outra limitação no jogo Território são os elementos que possuem dependências para uso ou exploração, como as descobertas tecnológicas que abrem novas possibilidades de unidades e outros elementos do jogo.

Com essas limitações, percebemos que o jogo Território possui regras mais

simples que as utilizadas nos jogos comerciais mais complexos, como o jogo *Civilization*. Apesar disso, o jogo ainda possui complexidade comparável aos jogos TBS mais simples, como o já citado *Battle for Wesnoth*.

Capítulo 4

O *Framework*

Neste capítulo, vamos apresentar o *framework* criado, tanto do ponto de vista de sua estrutura e arquitetura quanto do ponto de vista de como será utilizado para a implementação de um agente de IA. O objetivo aqui é uma apresentação mais técnica, que permita demonstrar o trabalho realizado e também servir de referência para o uso do *framework*, evidenciando assim como ele pode solucionar aspectos não atendidos por outras ferramentas já existentes.

Para que o capítulo não se torne excessivamente extenso, não apresentaremos aqui todas as propriedades e métodos públicos existentes em cada classe, mas apenas aquelas que forem necessárias para o entendimento de algum ponto, ou que apresentarem características muito importantes que justifiquem um detalhamento maior. A explicação completa de toda a interface pública (constantes, propriedades e métodos) das classes será apresentada no Apêndice A.

4.1 Linguagem e plataforma

A escolha da linguagem e plataforma a serem utilizados para o desenvolvimento foi um dos aspectos mais importantes na definição do *framework*. Pela natureza acadêmica do trabalho, era fortemente desejável que os *softwares* necessários possuíssem licenças gratuitas. Além disso, desejávamos uma linguagem moderna e ferramentas próprias ao desenvolvimento de jogos.

Com base nessa necessidade, após avaliarmos algumas opções como usar C++ com Open GL, ou até mesmo a *engine* Unity 3D, optamos, então pela utilizados do XNA 4.0, um *framework* da Microsoft voltado para o desenvolvimento de jogos para Windows, Xbox 360, Windows Phone 7 e Zune. A relação custo/benefício da utilização do XNA se mostrou mais vantajosa, uma vez que o XNA fornece vários recursos próprios para a criação de jogos (como recursos gráficos e utilização de efeitos sonoros), e já possuíamos uma experiência prévia com a tecnologia.

A linguagem utilizada no XNA é o C#, uma linguagem orientada a objetos bastante moderna e que atende de forma satisfatória a todas as necessidades do projeto. Todos os softwares necessários para o desenvolvimento em XNA (e com isso para desenvolvimento sobre o *framework*) são de uso gratuito, e estão listados abaixo:

- **Microsoft .Net Framework 4.0**: *Framework* sobre o qual os códigos criados são executados nas plataformas desejadas;
- **Microsoft Visual C# Express 2010**: Edição gratuita do ambiente de desenvolvimento para a linguagem C#;
- **Microsoft XNA Game Studio 4.0**: Conjunto de ferramentas, bibliotecas e *templates* voltado para o desenvolvimento de jogos, usado em conjunto com o .Net Framework e o Visual C#.

Embora não seja uma característica da linguagem de programação ou das plataformas envolvidas, é importante mencionar que todo o código-fonte foi escrito tendo o inglês como idioma base. Dessa forma, todos os métodos, variáveis, comentários e quaisquer outros elementos contidos no código-fonte se encontram em inglês, e serão apresentados da forma como se encontram originalmente. Caso em algum trecho de código incluído neste documento haja necessidade de uma tradução para que o entendimento seja possível, a tradução será feita e indicada em cada contexto.

4.2 Arquitetura

A arquitetura do *framework* foi projetada tendo em vista facilitar o desenvolvimento

dos agentes de IA no jogo. Com esse princípio, o *framework* foi modularizado de forma que seja possível aos pesquisadores e desenvolvedores atuar apenas sobre os aspectos necessários, ou seja, sem que haja necessidade de alterar regras do jogo ou incluir informações ou interfaces de comunicação com o jogo.

O *framework* conta com três módulos que se comunicam através de uma interface bem definida. Além desses três módulos, o *framework* ainda pode contar com diversos agentes (ou jogadores) de IAs diferentes e que atuam de forma independente. Um módulo de exemplo para um agente de IA que pode ser utilizado como referência pelos desenvolvedores é fornecido juntamente com o *framework*.

Tecnicamente, cada módulo é compilado para uma DLL, assim como cada agente de IA que será usado no jogo. A única exceção é o módulo principal (Game), sobre o qual é gerado o executável do jogo.

Nas seções a seguir, apresentaremos em linhas gerais o escopo de cada módulo e como a interação entre eles ocorre de fato. A Figura 4.1 mostra os módulos do jogo e o fluxo de comunicação entre eles.

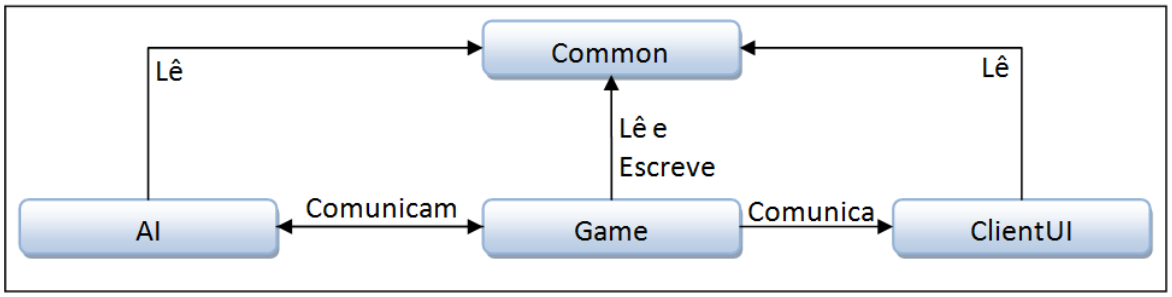

Figura 4.1: Diagrama de comunicação entre os módulos.

4.2.1 Módulo “Game”

O módulo “Game” (do inglês, “Jogo”) é o mais importante dos módulos, pois atua como o núcleo do *framework*. Ele é o responsável pela criação e manutenção das partidas, indicação dos turnos aos jogadores, validação das jogadas enviadas pelos jogadores e diversos outros aspectos relacionados ao funcionamento do jogo.

A comunicação com os agentes de IA é realizada através da interface de

realização de jogadas, na qual o jogo solicita a um jogador que efetue suas jogadas e, sempre que uma jogada é enviada pelo jogador ele efetua as validações necessárias e envia uma resposta ao jogador acerca do comando solicitado. Os detalhes acerca do controle das jogadas serão explicados nas próximas seções.

Apenas o módulo "Game" possui acesso privilegiado a alguns métodos e propriedades do módulo "Common" (que será explicado adiante), para definição das informações públicas do jogo como regras e quantidade de jogadores. Para os demais módulos, essas propriedades estão expostas apenas para leitura. Da mesma forma, as informações para que o módulo "ClientUI" (que também será explicado adiante) possa desenhar o estado do jogo também só podem ser enviadas pelo módulo "Game".

Por fim, o conteúdo deste módulo, é inacessível aos outros módulos como forma de garantir que um jogador não tenha acesso a informações sobre o jogo de forma indevida. Essa restrição foi a principal motivação para que fosse necessário criar o módulo "Common", um módulo de acesso comum aos demais módulos, que será apresentado na próxima seção. É importante mencionar aqui que apenas uma cópia do estado do jogo é enviada pelo módulo "Game" diretamente para as IAs executarem as jogadas, e por isso o estado do jogo contido neste módulo permanece inacessível para os demais módulos.

4.2.2 Módulo "Common"

O módulo "Common" (do inglês, "Comum") é onde estão localizadas as classes e métodos de uso comum pelos demais módulos do *framework*. Ele é composto tanto de elementos estáticos (como as entidades do jogo ou as regras) quanto os que são alterados ao longo do tempo (como o estado do jogo), mas sempre apenas os que são de uso comum pelos demais módulos. O principal motivo para a existência de um módulo apenas para esses recursos comuns é manter a modularização de forma que as dependências entre os módulos seja a menor possível, tornando-os componentes mais independentes. Além disso, isso evita um possível problema de referência circular entre os módulos, o que seria um grande entrave no desenvolvimento.

Os elementos deste módulo podem ser categorizados basicamente em três grupos:

Entidades do jogo, interface para os jogadores e classes auxiliares. As entidades são os elementos utilizados para representar o jogo, como o mapa, unidades etc. A interface para os jogadores é a definição da interface que será utilizada para que os jogadores se comuniquem com o jogo. As classes auxiliares, por fim, são compostas por métodos diversos que podem ser úteis para o núcleo do jogo, para o planejamento de jogadas de um jogador e até mesmo para obter informações que serão exibidas na interface com o usuário. Vamos nos ater a essa definição básica por enquanto, pois cada um desses grupos será detalhado nas próximas seções.

Como já foi mencionado, apenas o módulo Game possui permissão de acesso em alguns métodos e propriedades de escrita neste módulo. Os demais módulos têm acesso apenas para leitura, conforme pode ser visto também no diagrama da Figura 4.1.

4.2.3 ClientUI

Este é o módulo responsável por toda a parte de interface com o usuário (UI, do inglês *User Inteface*) do jogo, projetada para apresentar tanto a parte gráfica quanto música e efeitos sonoros. Embora nenhuma interface visual ou sonora tenha sido implementada para este trabalho devido à restrição de tempo, este aspecto do jogo não foi esquecido, e por isso toda a estrutura para trabalhos futuros relacionados a este tópico já foi criada.

Do ponto de vista da comunicação, este módulo recebe uma comunicação do módulo “Game” (para que desenhe as informações novamente) e não envia nenhum retorno ou mensagem de resposta. Para que as características do jogo possam ser exibidas de maneira satisfatória, este módulo possui acesso de leitura ao módulo “Common”. Por fim, nenhuma comunicação pode ser feita entre este módulo e os módulos das IAs.

Com essa arquitetura, acreditamos que foi possível um isolamento bastante satisfatório do componente de UI e, com isso, trabalhos futuros podem criar uma UI sem que haja necessidade de alterações nos demais componentes do *framework*.

4.2.4 Inteligência Artificial

Como já foi dito, o jogo pode contar com vários jogadores controlados por diferentes

agentes de IA. Para permitir maior flexibilidade na criação dos agentes e garantir a independência na execução e construção de cada um deles, cada agente de IA é tratado como um módulo adicional. Assim, se algum trabalho possuir alguma restrição quanto à exposição de código-fonte do agente, é possível inclusive que ele seja utilizado sem acesso ao seu código-fonte, vinculando apenas à biblioteca (DLL) correspondente. Para simplificar a escrita neste capítulo, vamos chamar os agentes (ou jogadores) de IA apenas de IA.

Como forma de garantir a proteção aos dados e estratégias adotadas por cada IA, o *framework* não permite a comunicação entre as diferentes IAs em uso no jogo. Na verdade, uma IA não sabe quais são os seus adversários, e a única informação que recebe sobre os demais jogadores é o número de identificação de cada um. A comunicação de uma IA com o jogo se dá no momento de efetuar as jogadas da sua vez, na qual a IA recebe o estado do jogo e envia as jogadas que deseja fazer. Qualquer tentativa de comunicação de uma IA com o jogo em um momento que não seja sua vez de jogar será rejeitada. Como já havia sido mencionado, cada IA também tem acesso de leitura às propriedades e métodos do módulo “Common”.

O modo de criação das IAs e a interface usada para comunicação com o jogo serão explicados nas próximas seções. Por enquanto, vamos apenas adiantar que os objetos (mapas, unidades etc.) recebidos pela IA são apenas clones (objeto diferente, mas com valores iguais) dos originais manipulados pelo módulo “Game”, o que permite às IAs alterar livremente as informações que desejar durante o planejamento das jogadas, sem que isso tenha qualquer efeito colateral nas informações reais do jogo.

4.3 Entidades

As entidades do jogo (mapa, unidades e demais elementos apresentados no capítulo anterior) utilizadas no jogo estão definidas no módulo “Common”, para que possam ser consultadas por todos os demais módulos. Nesta seção apresentaremos como essas entidades são representadas no *framework*, as propriedades e métodos que possuem e as classes relacionadas a cada uma delas.

Também apresentaremos as entidades e classes que, embora não possuam um mapeamento direto com as entidades percebidas no jogo (apresentadas no Capítulo 3), são necessárias para o entendimento da modelagem utilizada, do relacionamento entre as classes e da forma como o *framework* deve ser utilizado para a criação de agentes de IA.

Na Figura 4.2 podemos ver as entidades do jogo e a classe que representa cada uma no *framework*. Comparando essas entidades com os elementos do jogo apresentados, podemos perceber que três entidades – "elemento de jogo", "posição" e "elemento produzível" – existem apenas quando tratamos do ponto de vista da modelagem das classes e do código-fonte do *framework*. Entraremos em detalhes sobre o que elas representam nas seções que seguem neste capítulo.

Entidade	Classe
Elemento de Jogo	GameElement
Mapa	Board
Posição	Position
Território	Tile
Terreno	Terrain
Unidade	Unit
Item	UnitAttributes
Construção	Building
Elemento Produzível	ProduceableElement

Figura 4.2: Mapeamento entre entidades e classes.

A Figura 4.3 apresenta um diagrama simplificado das classes que representam as entidades do jogo, onde podemos ver as relações de herança e dependência entre as entidades. Para que o diagrama não ficasse confuso devido ao grande número de classes auxiliares e enumeradores utilizados na implementação, esses elementos foram omitidos, mantendo-se apenas as classes que representam as entidades listadas na Figura 4.2 e os relacionamentos entre elas. Nas seções a seguir apresentaremos cada uma dessas entidades, as classes envolvidas, enumeradores, relacionamentos e como são acessados no *framework*.

4.3.1 Elemento de Jogo

A primeira entidade que apresentaremos é o elemento de jogo. De fato, este elemento não pode ser considerado uma entidade de verdade, pois é na verdade uma abstração para todos os elementos dinâmicos do jogo, ou seja, o mapa, territórios, unidades etc.

Um elemento de jogo é representado pela classe “GameElement”, da qual devem herdar todas as demais classes que representam entidades do jogo. A funcionalidade por trás dessa entidade está na criação de um ID único para cada entidade do jogo, o que é utilizado para sobrescrever os métodos de comparação de igualdade (Equals) e criação de *hash* (GetHashCode) utilizados em diversas funcionalidades da linguagem C#. Com isso, a manipulação dos elementos do jogo se tornou muito mais fácil tanto para o núcleo quanto para as IAs, além de oferecer também melhor performance.

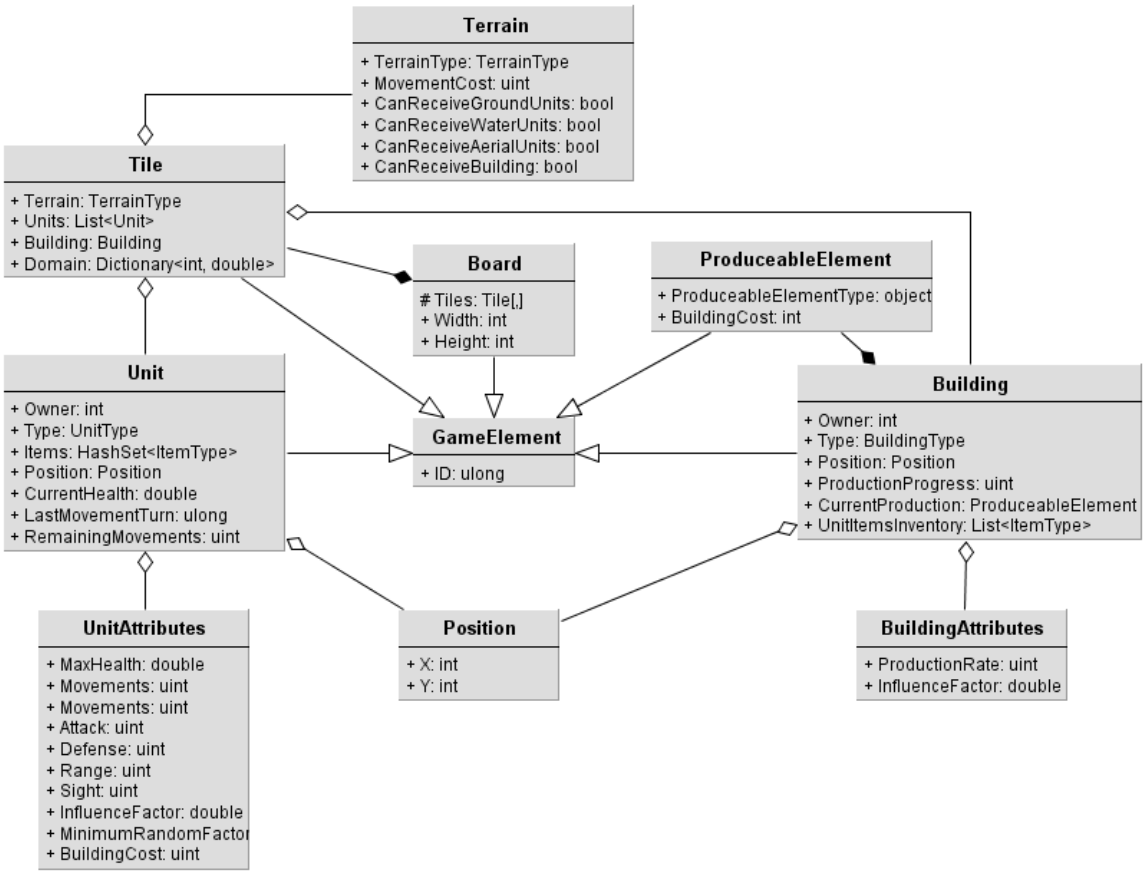

Figura 4.3: Diagrama de classes das entidades do jogo.

A classe “GameElement” é abstrata e implementa a interface ICloneable, permitindo assim um mecanismo unificado para clonar os objetos de todas as classes de entidades do jogo (uma vez que herdam desta). A única propriedade pública desta classe é “ID”, do tipo *ulong*, que contém o ID único mencionado no último parágrafo. O ID é gerado no momento de criação do objeto e não é possível alterá-lo.

4.3.2 Mapa

O mapa é a entidade que representa o estado do jogo, parcial ou totalmente, e é representado pela classe “Board”. É importante explicar que, embora o termo *board* em inglês signifique tabuleiro (e não mapa), preferimos este nome para a representação do mapa para não causar confusão de nomenclatura com a estrutura de dados “Map”, existente em diversas linguagens de programação (e.g. Java), embora em C# o nome não estivesse reservado. No entanto, continuaremos nos referindo a mapa quando tratarmos da entidade, adotando o nome “Board” apenas quando se referindo à classe.

No *framework*, o mapa é bidimensional onde cada célula é um hexágono. Cada célula representa um território do jogo, e no mapa esses territórios estão organizados em uma matriz bidimensional onde a primeira dimensão é a largura (eixo X) e a segunda dimensão é a altura (eixo Y). Por convenção, o eixo X cresce da esquerda para a direita e o eixo Y cresce de cima para baixo. Assim, um mapa de tamanho 6x8, por exemplo, tem 6 colunas e 8 linhas. Além disso, a posição (0,0) sempre será no canto superior esquerdo, uma vez que um mapa não aceita coordenadas negativas. Um exemplo de mapa pode ser visto na Figura 3.1, apresentada no capítulo anterior.

Para representar os territórios do mapa, foi utilizada uma matriz bidimensional, de forma que qualquer posição do mapa pode ser obtida da forma mais transparente e direta possível, uma vez que as posições da matriz correspondem às posições do mapa.

Como forma de facilitar o acesso às informações do mapa, bem como a execução de algumas tarefas durante o jogo, a classe *Board* conta também com diversas propriedades e métodos auxiliares, como para localizar todas as unidades de determinado jogador.

Como já foi explicado no capítulo anterior, o mapa do jogo é parcialmente observável, e por isso era necessário que os mapas do jogo pudessem representar

apenas uma visão parcial de acordo com o estado das unidades e construções de determinado jogador. Assim, um mapa visto por determinado jogador pode não ser igual ao visto por outro jogador. Ao tentar acessar um território ao qual o jogador não possui visibilidade, não será possível obter nenhuma informação acerca do território, incluindo-se aí tanto as unidades ou construções inimigas quanto o tipo de terreno (o território será exibido como um "terreno desconhecido").

É importante lembrar que, para garantir esse isolamento da visão do mapa, o jogador recebe um clone do mapa original, que contem apenas a sua visão do jogo disponível. Embora essa cópia tenha um custo de processamento para ser realizada, ela foi mantida para que a arquitetura cliente/servidor existente entre o núcleo do jogo e os jogadores fosse mantida, e com isso futuramente seja possível expandi-la para realizar jogadas através de uma rede local ou da Internet.

4.3.3 Posição

Esta é uma entidade bastante simples, criada para representar uma posição no mapa. Embora o XNA já forneça alguns tipos que podem ser usados para essa funcionalidade, como o "vector2", neles as coordenadas são representadas através de números reais ao invés de números inteiros como era necessário. Assim, para simplificar o acesso e manipulação de posições no jogo, foi criada a classe "Position", que armazena apenas as coordenadas X e Y de uma posição.

Através desta classe, as posições do jogo podem ser manipuladas como se fossem vetores geométricos de duas dimensões. Para isso, os operadores de soma (+), igualdade (==) e diferença (!=) foram sobrescritos na classe, efetuando assim as operações necessárias em cada caso. Como uma posição representa um ponto fixo e imutável do mapa, as coordenadas de uma posição não podem ser alteradas. Caso deseja-se alterar uma coordenada, é necessário criar uma nova posição. Para alterar a posição de uma unidade, por exemplo, não é possível alterar as coordenadas X ou Y da posição dela (objeto *Position*), pois é necessário criar uma nova posição (ou seja, um novo objeto da classe *Position*) e atribuí-la como a nova posição da unidade.

Para facilitar a manipulação das posições do mapa, a classe *Position* conta ainda

com diversas propriedades e métodos que já retornam informações acerca de determinada posição, como identificação dos vizinhos e obtenção de todas as posições a determinada distância. Esses métodos (que estão detalhados no Apêndice A) são particularmente úteis pelo fato de ser no jogo utilizado um mapa dividido em hexágonos, no qual a obtenção dos vizinhos de uma determinada posição não é tão simples quanto em mapas divididos em quadrados.

4.3.4 Territórios

O território é a entidade que representa cada célula do mapa e, com isso, é a entidade sobre a qual as ações do jogo são executadas. As características de um território são determinadas pelo seu tipo de terreno (o que será explicado na próxima seção), e cada território pode possuir unidades e uma construção de um jogador. Além disso, cada território pode possuir também domínio de um ou mais jogadores.

A classe criada para representar o território é a classe “Tile” (do inglês ”telha, azulejo, ladrilho”, o que dá a ideia de composição de uma superfície, o que no caso do jogo é o próprio mapa). As unidades no território (se houver alguma, é claro) são representadas através de uma lista. Conforme explicado no capítulo anterior, não é permitido que um território possua unidades de diferentes jogadores no mesmo instante. Os terrenos e a influência deles sobre o território serão explicados no próximo tópico.

O domínio é representado através de um mapeamento chave/valor utilizando a estrutura *Dictionary*, onde a chave é o identificador do jogador e o valor é a quantidade de domínio que o jogador possui no território. Essa estrutura foi escolhida primeiramente para garantir que não haja duplicações no domínio de jogadores (o que provocaria uma grande inconsistência no jogo), e também porque dessa forma a obtenção do domínio é feita de forma mais rápida.

Assim como já foi explicado sobre a classe *Board*, a classe *Tile* possui também diversas propriedades e métodos auxiliares, como para verificar se existe alguma unidade ou construção de determinado jogador ou executar a normalização do domínio no território.

4.3.5 Terrenos

Como mencionamos na seção anterior, um território possui um tipo de terreno, que determinará quais são as características desse território. Os atributos associados a um tipo de terreno não podem ser modificados ao longo do jogo, assim como um território não pode ter seu tipo de terreno alterado. Assim, se dois territórios possuírem o mesmo tipo de terreno, terão as mesmas características.

O terreno é representado através da classe *Terrain*, que contém todas as características de um terreno do jogo. Os tipos de terreno estão definidos no enumerador público *TerrainType*, existente na classe *Terrain*. Com isso, pode-se referenciar sempre apenas o tipo do terreno desejado (como é feito dentro do território), pois a partir do tipo de terreno é possível obter suas características correspondentes na classe *Terrain*. Para evitar confusão, chamaremos de terreno quando a referência for à classe *Terrain*, e tipo de terreno quando for ao enumerador *TerrainType*.

A associação entre o tipo de terreno e o terreno correspondente é feita através da classe estática *Terrains*. Esta classe possui um único método público que, para cada tipo de terreno, retorna terreno correspondente. As características dos terrenos são definidas na classe *Terrains* pelo módulo *Game* na inicialização da partida. Com essa estrutura, o custo de memória utilizado pelo *framework* para os terrenos é a menor possível, sem duplicação de instâncias de objetos equivalentes. Além disso, a obtenção das características de terreno é bastante simples e sem burocracias, sendo necessária apenas uma linha de código. Os tipos de terreno e suas características podem ser vistos na Figura 3.2, apresentada no capítulo anterior.

4.3.6 Unidades

A classe criada para representar cada unidade do jogo é a classe *Unit*. Conforme explicado no capítulo anterior, cada unidade possui atributos fixos (determinados pelo tipo da unidade) e atributos que representam o estado atual da unidade. Da mesma forma como foi feito com os terrenos, não havia necessidade de repetir os atributos fixos em cada unidade do mesmo tipo, e por isso foi criada a classe *UnitAttributes*, que encapsula todos os atributos fixos das unidades. Os tipos de unidade são definidos no

enumerador *UnitType*, presente na classe *Unit*.

Ainda em analogia aos terrenos, foi criada a classe *Units*, responsável pela associação de um tipo de unidade aos seus atributos, com funcionamento análogo à classe *Terrains*. Assim, as características das unidades são definidas na classe *Units* pelo módulo *Game* na inicialização da partida, e esta classe possui um único método público que, para cada tipo de unidade, retorna os atributos (*UnitAttributes*) correspondentes.

Por fim, como as unidades podem possuir itens que trazem bônus para os atributos fixos, a utilização da classe *UnitAttributes* para encapsular todos estes atributos permitiu que esta mesma classe fosse utilizada para os itens, tornando a manipulação dos itens e dos atributos das unidades mais simples e transparente, conforme será explicado na próxima seção.

4.3.7 Itens

Os itens do jogo são definidos e referenciados de forma bastante semelhante ao que é feito com os terrenos e unidades. A grande diferença é que, ao contrário das duas entidades já citadas, os itens não possuem nenhum atributo que possa ser alterado durante a partida e, por esse motivo, todas as referências a itens (ou seja, nas construções em que estão armazenados ou nas unidades em que foram equipados) são feitas apenas ao tipo do item, que são definidos em um enumerador. Assim, não é necessária nenhuma classe para instanciar os itens que estão no jogo.

Os tipos de item são definidos no enumerador *ItemType*, na classe *UnitItems*. O funcionamento dessa classe é semelhante às classes *Terrains* e *Units*, já apresentadas, onde o núcleo do jogo define os atributos para cada tipo de item na inicialização da partida e essas informações podem ser consultadas através de um método *Get* que retorna um objeto *UnitAttributes* correspondente ao tipo do item (*ItemType*) desejado.

É importante ressaltar aqui que como os atributos que os itens modificam nas unidades são os mesmo que as unidades possuem através da classe *UnitAttributes*, essa mesma classe foi utilizada também para a definição dos itens. Com isso, a manipulação dos atributos de unidades que possuem itens se tornou muito mais fácil e transparente através do operador de adição (+), que foi sobrescrito na classe *UnitAttributes*. Com

isso, para obter determinado atributo de uma unidade levando-se em consideração seus itens, basta utilizar a operação de adição entre todos os objetos do tipo *UnitAttributes* envolvidos (ou seja, o objeto originado do tipo da unidade mais os objetos originados dos tipos de itens que a unidade possuir) e obter o atributo desse objeto resultante. Essa operação já é feita automaticamente quando se acessa qualquer um desses atributos a partir de um objeto da classe *Unit*.

4.3.8 Construções

A classe criada para representar cada construção do jogo é a *Building* (do inglês, construção). A manipulação das construções é feita de forma bem semelhante ao que é feito com as unidades. Cada construção é um objeto da classe *Building*, sendo que os atributos de cada construção são definidos de acordo com o tipo de construção. Os tipos de construção estão definidos no enumerador *BuildingType*, na classe *Building*.

Assim como é utilizada a classe *Units* para obter os atributos de uma unidade a partir do seu tipo, a classe *Buildings* é responsável pela manipulação das construções. No caso das unidades, os atributos eram retornados em um objeto da classe *UnitAttributes*, enquanto no caso das construções os atributos são retornados em um objeto da classe *BuildingAttributes*.

4.3.9 Elemento produzível

O elemento produzível, representado pela classe *ProduceableElement*, é a entidade utilizada para os elementos do jogo que podem ser produzidos dentro das construções. Os objetos desta classe podem representar um tipo de unidade ou um tipo de item, cada um instanciado por um dos dois construtores públicos. A principal motivação para esta classe é que haja uma interface comum para tratar a produção dentro das construções. Como os tipos de unidades e itens foram implementados utilizando enumeradores (*enum*, em C#), não era possível a utilização direta de herança ou interfaces.

A classe *ProduceableElement* possui apenas duas propriedades públicas. A primeira é o tipo do elemento (public object ProduceableElementType) que está em

produção, armazenado como um *object* e verificado em tempo de execução se é uma instância de *UnitType* ou *ItemType*. A segunda propriedade é o custo de produção (public uint BuildingCost), que já retorna o custo de produção da unidade ou do item que estiverem armazenados na propriedade *ProduceableElementType*, de acordo com a referência correta (*Units* para *UnitType* e *Items* para *ItemType*).

4.4 Inicialização das partidas

Ao iniciar o jogo, o módulo *Game* é executado e inicia as rotinas de inicialização dos recursos do jogo através do método *Initialize*, local padrão para as inicializações dos recursos não gráficos do jogo no XNA. A Figura 4.4 apresenta o diagrama contendo o fluxo da inicialização das partidas no *framework*, realizado a partir da chamada ao método *Initialize*.

Em primeiro lugar, é inicializado o *log* do jogo, responsável pelo registro de todas as ações ocorridas na partida. O *log* será explicado em uma seção exclusiva mais à frente. Em seguida, é utilizada a classe estática *GameInitializer* para inicializar as regras do jogo. As regras são definidas no arquivo de configurações *Rules.settings*, localizado no módulo *Game*, visando facilitar a alteração das regras do jogo quando desejado. Assim, elas são carregadas para o jogo durante a inicialização e armazenadas na classe estática *GameSettings*, existente no módulo *Common*, para consulta pelos demais módulos do jogo, e não podem ser alteradas em tempo de execução.

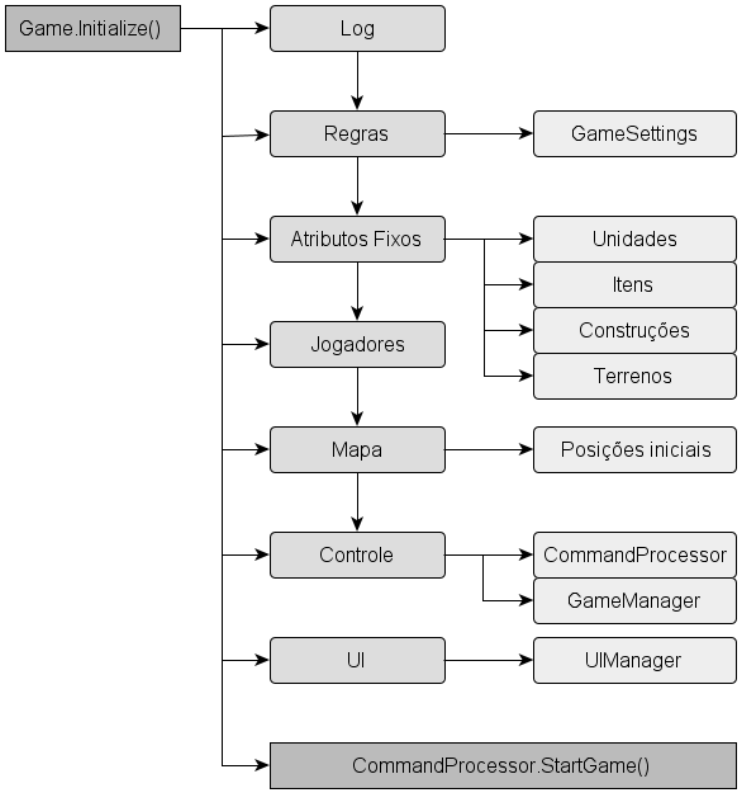

Figura 4.4: Diagrama do fluxo de inicialização da partida.

Uma vez carregadas as regras do jogo, a classe *GameInitializer* é utilizada para inicializar os atributos fixos dos elementos do jogo, ou seja, as características das unidades, itens, construções e terrenos. Assim como as regras, estas características não podem ser alteradas em tempo de execução. Cada uma dessas entidades possui uma classe própria para a inicialização, nas quais são definidos e instanciados os objetos correspondentes. As unidades são inicializadas na classe *UnitsInitializer*, os itens na classe *UnitItemsInitializer*, as construções na classe *BuildingsInitializer* e os terrenos na classe *TerrainsInitializer*. Dessa forma, foi possível deixar a configuração de cada entidade em um arquivo isolado, tornando a alteração de uma ou mais características mais fácil.

Em seguida, a classe *GameInitializer* é utilizada para inicializar os jogadores (ou IAs) que atuarão na partida. Esta inicialização, por sua vez, é feita pela classe estática *Players*, responsável pelo gerenciamento dos jogadores. Esta classe será explicada na próxima seção. Para os testes e a avaliação deste trabalho, optou-se por utilizar duas

IAs, sendo uma a IA de exemplo fornecida junto com o *framework* e outra a IA criada pelos avaliadores do *framework*. Caso seja necessário utilizar uma configuração diferente, basta alterar o método *Initialize*, da classe *Players*.

Após a inicialização dos jogadores, é feita a inicialização do mapa do jogo, através da classe estática *BoardManager*. As dimensões do mapa criado são obtidas através das regras do jogo, e os terrenos de cada território do mapa são obtidos aleatoriamente. Após a geração do mapa, são definidos os territórios onde cada jogador iniciará a partida. Nesses territórios, é colocada uma construção e uma unidade para que o jogador inicie a partida, e o tipo de terreno é definido como *Grama* (*Grass*, no enumerador *TerrainType*), para que não haja risco de ocorrer situações inválidas no jogo, como unidades sobre água ou construções sobre florestas. A posição inicial dos jogadores é definida de forma aleatória de acordo com algumas restrições, para que não haja jogadores em territórios muito próximos. A implementação atual foi feita para aceitar até quatro jogadores simultâneos. Para que seja possível inicializar mais que quatro jogadores no mapa, é necessário alterar o método *InitializeRandomBoardElements*, na classe *BoardManager*. É importante ressaltar aqui que a restrição de quatro jogadores existe apenas na inicialização das partidas. O restante do *framework* não possui restrições quanto ao número de jogadores simultâneos.

Quando o mapa do jogo já está criado, passa-se à inicialização da camada de controle do *framework*, mais especificamente as classes estáticas *CommandProcessor* e *GameManager*, responsáveis pelo processamento dos comandos e do andamento do jogo, respectivamente. Estas duas classes serão explicadas em maiores detalhes nas duas próximas seções.

O último elemento a ser inicializado no jogo é o gerenciador da interface com o usuário, na classe *UIManager*. Esta classe será explicada em maiores detalhes nas próximas seções, mas é importante esclarecer que neste momento são apenas inicializadas as classes e controles necessários para o gerenciamento e exibição dos elementos de UI. Os elementos gráficos e sonoros devem ser carregados pelo módulo de interface gráfica logo após as inicializações, no momento de carregamento dos conteúdos.

Por fim, após a inicialização de todos os recursos explicados acima, a partida é

iniciada acionando-se a classe *CommandProcessor*. A partir desse momento, o *loop* do jogo é iniciado e passa-se ao controle das jogadas, dos turnos, gerenciamento do jogo e exibição da interface gráfica.

4.5 Gerenciamento dos recursos

Com o objetivo de tornar o gerenciamento e a manipulação dos diversos recursos do jogo, o módulo *Game* possui algumas classes criadas exclusivamente para este fim. Estas são classes estáticas e visíveis apenas no escopo do módulo (todas possuem visibilidade *internal*), o que torna a manipulação dos recursos do jogo bastante simples.

A primeira classe que vamos apresentar é a *GameElementsManager*, responsável pelo gerenciamento das entidades do jogo que podem ser alteradas, ou seja, as unidades e construções. Esta classe armazena todas essas entidades que são criadas no mapa do jogo em um mapeamento chave-valor através de uma estrutura do tipo *Dictionary*. A chave é o ID da entidade – lembrando que esse ID é criado para todos os objetos de classes que herdam a classe *GameElement* – e o valor é o próprio objeto. Esta classe foi criada para otimizar a busca dos elementos do jogo no mapa ao receber as jogadas de determinado jogador. Como as jogadas referenciam apenas o ID dos objetos, o *framework* precisaria localizar o objeto no mapa, o que traria um grande problema de desempenho por ser necessário varrer o mapa para localizar os elementos em cada jogada recebida. Com essa estrutura, a busca das entidades tem complexidade praticamente O(1), uma vez que segundo a documentação do .Net Framework a implementação do *Dictionary* é feita através de uma tabela *hash*. Para o correto funcionamento dessa classe, todo elemento adicionado ao jogo (ou seja, unidades ou construções que são criadas) devem ser adicionados a ela, e toda vez que esses elementos são eliminados do mapa devem ser removidos dela.

A segunda classe responsável pelo gerenciamento de recursos do jogo é a classe *Players*. Ela é responsável por gerenciar os jogadores da partida e a situação em que cada um se encontra no momento, como quais jogadores ainda estão vivos e o jogador que está com a vez. Os jogadores são armazenados em uma lista de elementos da classe

Player, que é a classe que representa as informações de um jogador. Cada jogador possui uma instância, correspondente à DLL do módulo de IA usado para o jogador em questão, utilizada para informar a vez de jogar. Além disso, cada jogador possui um atributo booleano que indica se o jogador está vivo na partida ou não. Assim, a classe *Players* utiliza essa lista de jogadores para alternar entre eles a vez de jogar. Internamente, a classe possui um apontador indicando qual o jogador da vez atual, retornando-o na propriedade *CurrentPlayer* (do inglês, jogador atual) quando solicitado. Quando solicitado o próximo jogador (através da propriedade *NextPlayer*), o ponteiro é movido para o próximo jogador ainda vivo na partida. Se o ponteiro chegar ao final da lista, a busca continua a partir do início e um turno é finalizado na classe *GameManager* (que será explicada a seguir). Se o ponteiro retornar ao jogador sobre o qual estava ao início da busca, é lançada uma exceção indicando que não há mais jogadores disponíveis e a partida deveria ter sido encerrada. No controle das jogadas, que será explicado na próxima seção, esta exceção é tratada para verificar a finalização da partida e chamar os métodos para finalização do jogo.

A próxima classe que apresentaremos é a *GameManager*, responsável pelo gerenciamento do andamento do jogo, mais especificamente as tarefas de finalização de turnos e finalização da partida. Toda vez que um turno é encerrado, é preciso processar algumas mudanças no jogo antes de iniciar o novo turno, como avançar o progresso da produção em curso em cada construção e a recuperação da saúde de unidades. Essas tarefas são executadas a partir do método *NextTurn*, que por sua vez é chamado a partir da classe *Players*, como já foi explicado. Os jogadores podem consultar o turno atual da partida através da classe *GameState*, presente no módulo *Common*. Além do fim dos turnos, quando uma partida é finalizada (ou seja, apenas um jogador restou no mapa), é chamado o método *EndGame*, que envia ao *log* as informações de fim de partida e informa ao *loop* do jogo que a partida foi encerrada e a execução deve ser finalizada.

A última classe responsável pelo gerenciamento de recursos é a *BoardManager*, na qual é gerenciado e manipulado o mapa do jogo. Esta classe é de fundamental importância para a proteção das informações do jogo às quais as IAs não podem ter acesso. Como no jogo Território o mapa é parcialmente observável, não se poderia permitir que um jogador tivesse acesso a informações de territórios que estejam fora do alcance de visão de suas unidades e construções. Por esse motivo, esta classe possui o método *GetPlayerBoard* que, dado um jogador, retorna uma representação do mapa que

pode ser visto pelo jogador. Além disso, toda representação do mapa obtida através desta classe retorna na verdade uma cópia do mapa, contendo objetos diferentes (novas instâncias), mas que contenham os mesmos valores. Com isso, uma cópia do mapa é enviada aos jogadores, e por isso as alterações realizadas pelos jogadores nos objetos enviados não são refletidas nos objetos originais, garantindo proteção aos dados do jogo e não permitindo acesso a informações restritas. Já para acessos de dentro do módulo *Game*, o mapa pode ser lido e alterado sem restrições através da propriedade *Board*.

4.6 Controle de jogadas

O controle das jogadas é um dos mecanismos mais importantes do *framework*, responsável por notificar cada jogador da sua vez de jogar, validar e executar as jogadas recebidas, impedir trapaças, enfim, garantir o bom funcionamento da mecânica do jogo e o fluxo dos turnos. A principal classe envolvida nesse processo é a *CommandProcessor*.

De forma resumida, podemos explicar o processo da seguinte forma: ao iniciar a partida, é dado início ao primeiro turno, e a vez é enviada ao primeiro jogador. A classe entra em um estado de espera, onde aguarda que o jogador envie as jogadas que deseja fazer. Ao terminar a vez, aciona-se o próximo jogador e novamente aguarda que ele envie as jogadas até o término da vez. Esse processo se repete até que reste apenas um jogador no mapa. Nesse caso, a execução do controle de jogadas é interrompida pela classe *GameManager*, responsável pela finalização da partida, conforme explicado na última seção.

Para garantir o correto gerenciamento das jogadas e garantir que o fluxo do jogo não será interrompido, todo o controle de jogadas é realizado através de *threads*. Uma nova *thread* é criada sempre que for o momento de entregar a vez de jogar para um jogador e a chamada à IA correspondente ao jogador é feita a partir dessa nova *thread*. Num fluxo sem erros, essa *thread* é executada durante o planejamento e execução das jogadas desse jogador, e ao final da sua vez ela é finalizada. No entanto, não é possível

garantir que esse fluxo sem erros ocorra para todas as implementações de IAs feitas com o *framework*, podendo assim prejudicar o desempenho do jogo devido a um número muito grande de *threads* em execução simultaneamente. Para que esse problema não ocorra, todas essas *threads* criadas para as jogadas das IAs são armazenadas enquanto estiverem ativas (verifica-se quais *threads* ainda estão ativas periodicamente, mantendo em memória o registro apenas das *threads* ainda ativas), o que permite tratar a situação da forma adequada, por exemplo forçando o encerramento de uma *thread* que não está mais respondendo. Outra vantagem dessa abordagem é que, no encerramento do jogo, todas as *threads* que foram criadas podem ser finalizadas da maneira correta.

Existe ainda uma *thread* responsável por encerrar a vez de um jogador que não responde à sua vez após determinado período de tempo (por padrão, 30 segundos). Se isso ocorrer, a *thread* do jogador é abortada e o jogador perde a vez, que é passada a ao próximo jogador, dando continuidade ao fluxo do jogo. Com isso, o *framework* garante que o fluxo do jogo não é impactado caso haja uma falha na execução de uma IA, ou então se o processamento de uma IA for mais longo do que o tempo permitido.

Devido à arquitetura descentralizada de execução (cada jogador é processado de maneira independente), era necessário garantir que um jogador não pudesse efetuar uma jogada fora da sua vez, ou até mesmo efetuar uma jogada fingindo ser outro jogador. Isso foi feito através de um mecanismo de chaves, que são geradas aleatoriamente para cada vez de cada jogador. Quando a vez é enviada para um jogador, este recebe também um código de validação do turno, gerado aleatoriamente e de conhecimento apenas pela classe *CommandProcessor*, no núcleo do jogo, e pelo módulo de IA do jogador em questão. Esse código deve ser enviado junto com cada comando que a IA enviar para execução no jogo, junto com o identificador do jogador. Assim, uma jogada só é processada pelo jogo se o for enviada pelo jogador da vez e se for fornecido o código de validação de turno correto. Com isso, uma IA não poderá obter vantagens sobre outra através do envio de comandos fora do seu turno de maneira ilícita.

Quando a vez de jogar é passada para um jogador (seja porque o último jogador passou o turno ou porque perdeu a vez devido ao tempo excedido), o controlador de jogadas utiliza a classe *Players* para obter o próximo jogador, conforme explicado na

seção anterior. Se por acaso for lançada uma exceção indicando que não há mais jogadores disponíveis, esta exceção é tratada e os métodos para finalização do jogo são chamados na classe *GameManager*.

A última funcionalidade tratada pelo controle de jogadas é a validação e execução dos comandos recebidos das IAs. Ao receber um comando, o controlador de jogadas verifica o tipo do comando (mover, atacar etc.) e aciona o método responsável por tratar esse comando. Nesse método, o comando é validado e, caso o comando não possa ser executado (por exemplo, tentando atacar um território que está fora do alcance da unidade selecionada), a jogada não é processada e o jogador é avisado do motivo de o comando não ter sido aceito. Se o comando for validado com sucesso, ele é processado, as devidas alterações são realizadas no mapa e o jogador é informado do sucesso da execução, recebendo também o mapa do jogo resultante da ação executada. Na próxima seção explicaremos como é feita a validação e persistência dos comandos, bem como o escopo de cada comando e como devem ser usados na implementação da IA.

4.7 Execução de comandos dos jogadores

A execução dos comandos com o *framework* é bastante simples. Em sua vez de jogar, a IA recebe o mapa (objeto da classe *Board*) com o estado atual do jogo (de acordo com a visibilidade do jogador, é claro) e o código de validação do turno (tipo *long*). Para enviar uma jogada para execução no jogo, basta criar um comando e enviá-lo no método de retorno passado pelo módulo *Game* na construção da instância do jogador. Ao receber esse comando, o módulo *Game* processa a jogada de acordo com as propriedades do comando recebido e envia de volta ao jogador uma resposta com o resultado da execução do comando.

Um comando é um objeto da classe *PlayerCommand*, que deve ser criado através desta mesma classe, usando o método correspondente ao tipo de comando desejado. Os comandos possíveis são movimentar, atacar, equipar unidade, influenciar território, definir produção, criar construção e finalizar turno, como já apresentamos no capítulo anterior. Explicaremos os métodos relacionados a cada tipo de comando e como usá-los

para criar os comandos desejados na próxima subseção.

A resposta enviada de volta à IA é um objeto da classe *PlayerCommandResponse*, e contém o resultado da execução (se o comando estava OK ou não), o mapa resultante da ação e o erro apresentado. Na lista abaixo apresentamos essas propriedades:

- Resultado: Indica se o comando enviado foi executado ou não. O resultado é obtido na propriedade *Result*, representada pelo enumerador *PlayerCommandResult*;
- Mapa resultante: Quando o comando foi executado com sucesso, o jogador receberá o mapa resultante da execução do comando através da propriedade *ResultBoard*, que é um objeto da classe *Board*.
- Erro: Quando a execução não foi executada, uma exceção é lançada dentro do módulo *Game* e colocada na resposta dentro da propriedade *Error*, que é justamente da classe *Exception*. Assim, o jogador pode identificar qual foi o problema e continuar sua jogada, enviando ou não um novo comando.

4.7.1 Definição dos comandos

Como vimos no início da seção, cada jogada feita por um jogador é representada por um objeto da classe *PlayerCommand*. Para simplificar a criação dos comandos e diminuir a possibilidade de criação de comandos inválidos, todos os construtores da classe são privados, e a criação dos comandos é feita através de métodos estáticos que já exigem os parâmetros de acordo com o tipo de comando correspondente ao método.

Vamos tomar como exemplo um comando de ataque (tipo de comando *Attack*), no qual o jogador precisa definir a unidade que está atacando e a posição de destino do ataque. Para criar esse comando, portanto, chama-se o método estático *AttackAt* na classe *PlayerCommand*, passando por parâmetro a unidade que está atacando (classe *Unit*) e o destino do ataque (classe *Postition*). Além dos parâmetros específicos de cada ação, para construir um comando deve-se passar também a instância do jogador que está criando o comando, o que será usado para validar a jogada dentro do módulo Game.

A seguir estão listadas as ações do jogo e o método estático correspondente na

classe *PlayerCommand*. Os parâmetros necessários e as validações que serão realizadas pelo módulo *Game* na classe *ComandProcessor* serão detalhados no Apêndice A.

- Movimentar: MoveTo
- Atacar: AttackAt
- Equipar unidade: EquipUnit
- Influenciar território: InfluenceAt
- Definir produção: SetProduction
- Criar construção: CreateBuilding
- Finalizar turno: EndOfTurn

4.7.2 Processamento dos comandos

No início da seção, já vimos que a classe *CommandProcessor* é a responsável pelo controle das jogadas, passar a vez aos jogadores e receber de volta os comandos que o jogador deseja executar no turno. Agora explicaremos em maiores detalhes como é feito o processamento dos comandos recebidos, a validação e alteração das informações do mapa.

Quando o jogador executa um comando, é feita uma chamada ao método estático *ExecuteCommand*, na classe *CommandProcessor*. Esse método recebe como parâmetros apenas o comando do jogador (objeto da classe *PlayerCommand*) e o código de validação do turno (tipo *long*), que já foi explicado anteriormente. Se em qualquer etapa do processamento do comando houver algum erro (seja na validação prévia ou na própria execução do comando), a exceção levantada é capturada neste método e repassada ao jogador na resposta do comando (objeto da classe *PlayerCommandResponse*), conforme vimos na seção anterior.

Logo após a validação inicial da jogada, onde se verifica o identificador do jogador e o código de validação do turno, inicia-se então o processamento do comando recebido, com a chamada do método responsável pela execução do tipo de comando recebido. A primeira ação executada é a validação do comando, de acordo com as regras de cada comando, já apresentadas nesta seção. Se não for encontrado nenhum erro, as ações envolvidas no comando são executadas sobre o mapa do jogo, e o

comando é concluído com sucesso, retornando-se então ao jogador o mapa resultante da ação realizada.

No caso dos comandos de movimentar, atacar, influenciar território e criar construção, que envolvem alterações ou regras mais complexas, a execução das ações envolvidas no comando é realizada na classe *UnitActionsPerformer*. Esta classe é responsável justamente pela execução das ações de jogadas mais complexas, nas quais as unidades são o ator principal. O motivo para essa divisão é tornar o código mais limpo e claro, sem regras ou ações muito específicas dos comandos na classe *CommandProcessor*, que já executa diversas outras tarefas.

Para ilustrar como é feito o processamento, vamos tomar como exemplo uma ação de ataque de uma unidade. O comando é recebido na classe *CommandProcessor*, onde são feitas as validações e, caso esteja tudo correto, é chamado o método estático *AttackAt* na classe *UnitActionsPerformer*, passando o mapa do jogo, a unidade que está atacando e a posição do território alvo do ataque. Nesse método, obtêm-se do mapa os territórios de ataque e defesa do mapa. Em seguida, calcula-se o valor do ataque total da unidade, de acordo com seu tipo, itens equipados e os bônus de agrupamento e domínio de territórios. Após esse cálculo calcula-se o valor de defesa total da unidade, seguindo-se as mesmas regras utilizadas para a unidade atacante. Com base no ataque e defesa calculados, calcula-se o dano sofrido na batalha por cada unidade, de acordo com a fórmula já apresentada na Figura 3.7. Por fim, é processado o dano causado em ambas unidades, reduzindo o valor correspondente na saúde atual da unidade e, quando a unidade for eliminada (saúde atual for menor ou igual a zero), a unidade é removida do mapa do jogo e do gerenciador de recursos *GameElementsManager*.

Uma vez que o comportamento é feito de maneira bastante similar para as demais ações do jogo, não é necessário aqui detalharmos o processamento de todas as ações. As regras envolvidas em cada uma já foram devidamente apresentadas no Capítulo 3.

4.8 Interface com o Usuário

A criação da interface com o usuário (UI) é feita no módulo *ClientUI*. A comunicação

utilizada entre os módulos *Game* e *ClientUI* para esse fim é bastante simples. O XNA já fornece o mecanismo de *loop* do jogo, e a renderização dos elementos gráficos é feita em toda chamada ao método *Draw*, na classe *TerritoryGame*, principal classe do módulo *Game*. Quando esse método é chamado, o módulo *Game* aciona então a classe *UIManager*, responsável pelo gerenciamento da UI do jogo. Esta classe, por fim, aciona o método *Draw* na classe *UI* do módulo *ClientUI*, enviando o mapa do jogo para que seja desenhado.

Para tornar a UI mais flexível e compatível com as necessidades de cada projeto que venha a utilizar o *framework*, a UI pode ser usada tanto para exibir o mapa completo do jogo quanto o mapa visto por um jogador específico. A alteração do tipo de visualização pode ser realizada facilmente na classe *UIManager*, no método *Draw*. No caso da exibição do mapa visto por um jogador, a definição desse jogador é realizada na inicialização da partida, no método *Initialize* dessa mesma classe.

É importante ressaltarmos aqui que, embora o *framework* conte com a estrutura para a UI, não era o objetivo deste trabalho a criação de uma interface gráfica, o que por si só já poderia ser considerado um projeto à parte por ser bastante extenso. Por esse motivo, foi criada apenas a estrutura já modularizada que possa receber a implementação da UI em um trabalho futuro.

4.8.1 Logs

Embora a criação de uma interface gráfica para o jogo não estivesse dentro do escopo deste trabalho, sabemos que a falta de uma UI pode dificultar o uso do *framework* devido ao difícil acompanhamento das ações que ocorrem no jogo. Por esse motivo, foi criado um mecanismo de *log* onde são registradas as ações do jogo, o que permite que o autor da implementação acompanhe o que está ocorrendo na partida sem a necessidade de investigação da execução em modo debug.

A classe responsável pelo *log* é a *GameLogger*, presente no módulo *Game*. O *log* é inicializado através do método *Initialize* durante a fase de inicialização do jogo, e na finalização da execução do jogo ele é terminado através do método *Close*. Isso é necessário por causa das operações de abrir e fechar o arquivo. O *log* possui um

arquivo de configuração (Logger.settings) que possui três itens:

- LocalFile: Caminho no sistema de arquivos onde será gravado o arquivo do *log*;
- LogLocalFile: Indicador booleano, onde verdadeiro significa que os registros do *log* serão gravados em um arquivo e falso significa que não serão. Quando estiver marcado como falso, a configuração *LocalFile* não é utilizada;
- LogConsole: Indicador booleano, onde verdadeiro significa que os registros do *log* serão exibidos no console da execução e falso significa que não serão.

O *log* permite que todas as ações do jogo sejam exibidas no *console* e gravadas em um arquivo. Assim, o jogador é informado de qualquer modificação ocorrida no mapa do jogo (criação de unidade, ataque, movimentação etc.). No entanto, esse mecanismo de *log* é apenas um artifício para minimizar os impactos da ausência de uma interface gráfica para o jogo, especialmente para a avaliação do *framework* realizada neste trabalho. Uma vez que a implementação da UI tenha sido realizada, mesmo que de maneira bastante simples, o *log* não será mais necessário com a finalidade original, de informar os jogadores o que se passa no jogo. Com isso, poderá ser desabilitado por padrão, para que seja usado apenas durante *debugs* de implementação. É importante ressaltar aqui que a IA não deve acessar os registros do *log* em hipótese nenhuma, pois toda informação extraída do jogo deve ser obtida através do mapa recebido em sua vez de jogar. O *log* possui registro de todas as informações do jogo (incluindo as ações que o jogador não poderia ver), e por isso deve ser utilizado apenas para acompanhamento por parte do implementador humano, nunca usado para fornecer informações privilegiadas à IA. Isso não impede, no entanto, que a IA gere seu próprio *log* ou outra forma de registro e nem que informações sejam gravadas em um arquivo. Um exemplo de *log* de uma partida completa pode ser vista no Apêndice B.

4.9 Classes auxiliares

O *framework* conta com algumas classes estáticas de métodos utilitários, disponíveis no módulo *Common*. Esses métodos podem ser utilizados para auxiliar a implementação das IAs, diminuindo a quantidade de código gerado.

A primeira classe utilitária é a *ListUtil*, criada para conter utilitários na manipulação genérica de listas. O método contido nela é o *GetListWithoutDuplicates*, que recebe uma lista e retorna uma nova lista com os mesmos elementos, mas sem que haja repetições. A seguir apresentaremos as demais classes e os principais métodos disponíveis.

4.9.1 RandomUtil

A classe *RandomUtil* possui algumas propriedades e métodos que fornecem valores pseudoaleatórios. Ela é construída utilizando a classe *Random*, fornecida no *.Net Framework*, mas oferece mais opções de geração de números pseudoaleatórios do que a classe *Random*. Além disso, evita que sejam criadas várias instâncias de um gerador pseudoaleatório, o que pode reduzir drasticamente a qualidade da distribuição obtida. As propriedades e métodos disponíveis nessa classe estão apresentados no Apêndice A.

4.9.2 MovementUtil

A classe *MovementUtil* possui métodos que auxiliam em tarefas que envolvem planejamento de caminhos e movimentação. A seguir apresentamos os principais métodos e a funcionalidade deles:

- CalculateBestRouteTo: Obtém a rota com menor custo de movimentação entre duas posições do mapa. Se o mapa não estiver completamente visível (ou seja, haja territórios desconhecidos), eles não serão utilizados para a rota. A rota obtida é uma lista com as posições do melhor caminho, na ordem em que devem ser utilizadas. O algoritmo utilizado é o A* [Hart et

al., 1968];

- CalculateRouteCost: Calcula o custo de movimentação com base em uma rota, ou seja, uma lista de posições. O método não verifica se a rota é possível ou não, apenas realiza o somatório do custo de movimentação para cada território existente na rota;
- CalculateManhattanDistance: Calcula a distância entre duas posições considerando que o custo de movimentação para cada território é constante. O algoritmo utilizado é uma adaptação da Distância de Manhattan [Black, 2006] para mapas hexagonais;

4.10 Agentes de Inteligência Artificial

Finalizando a apresentação técnica acerca do *framework*, vamos enfim apresentar a criação dos agentes de IA, relacionamentos envolvidos e implementações necessárias para a integração correta com o restante do jogo. Do ponto de vista do objetivo principal deste trabalho, esta é sem dúvida a parte mais interessante, onde podemos ver, na prática, o que os pesquisadores terão à disposição para usar o *framework* na criação de um agente de IA capaz de agir de forma autônoma no jogo.

Um dos objetivos do trabalho era permitir a implementação de algoritmos em um agente de IA da forma mais simples possível, buscando reduzir ao máximo a necessidade do uso de códigos relacionados à integração com o jogo ou o *framework*, permitindo que o pesquisador foque apenas na implementação dos algoritmos relacionados à IA. Aqui veremos como a interface para a implementação de um agente é bastante simples e direta, e a comunicação das jogadas é feita de forma bastante enxuta.

Para a implementação de um jogador de IA capaz de interagir com o jogo, basta que a classe abstrata AIBasePlayer, presente no módulo Common, seja herdada e o método abstrato PlayTurn seja implementado. Para o controle do jogo realizado no módulo Game, todo jogador é simplesmente uma instância da classe AIBasePlayer, e na vez de jogar chama-se esse método PlayTurn.

O construtor da classe do jogador deverá receber o ID do jogador (número gerado aleatoriamente pelo módulo *Game*) e o método utilizado para envio de comandos (este é definido através do *delegate PlayerCommandCallBack*), e ambos serão repassados para o construtor da classe *AIBasePlayer*. A partir da classe filha, o ID do jogador poderá ser consultado através da propriedade pública ID, e o método para envio de comandos poderá ser acionado através da propriedade CallBack, de visibilidade protegida (modificador *protected*).

Como já mencionamos, o método abstrato *PlayTurn(Board board, long turnValidationCode)* da classe *AIBasePlayer* deve ser implementado para que o jogador possa receber a vez de jogar. Este método recebe por parâmetro o mapa (objeto da classe *Board*) com o estado atual do jogo (de acordo com a visibilidade do jogador, é claro) e o código de validação do turno (tipo *long*), que será usado para envio dos comandos, conforme já explicamos nas seções anteriores.

Para envio das jogadas que deseja realizar no jogo, deve-se criar o comando através da classe *PlayerCommand*, como já explicamos neste capítulo e, em seguida, chamar o método para envio de comandos através da propriedade *Callback*. O método recebe como parâmetros apenas o comando do jogador (objeto da classe *PlayerCommand*) e o código de validação do turno (tipo *long*), recebido como parâmetro do método *PlayTurn*. O retorno da chamada a este método é um objeto da classe *PlayerCommandResponse*, que já foi explicado nas seções anteriores.

Os requisitos para a criação de uma IA capaz de interagir completamente com o jogo são apenas esses. A partir dessa implementação inicial, a IA já estará pronta para interagir com o jogo, e fica a critério de quem está criando ou modificando o agente as decisões de implementação, estruturas utilizadas, algoritmos, modelos e quaisquer recursos que forem necessários.

Para integrar no jogo o agente de IA construído, é necessário referenciá-lo dentro do módulo *Game*. Atualmente, para que isso seja feito é necessário ter acesso a todo o código-fonte do projeto, incluir a referência no método *Initialize* da classe *Players* e compilar o projeto novamente. Essa referência à IA pode ser feita tanto com um projeto que será compilado junto com o jogo quanto através de uma DLL já compilada. Com isso, é possível incorporar uma IA sem que se tenha acesso ao código-fonte. Em

trabalhos futuros pretendemos alterar esse mecanismo no módulo *Game* para que não seja necessário recompilar o projeto todo, bastando, por exemplo, alterar um arquivo de configurações para indicar o *namespace* e nome da classe utilizada no agente.

No Apêndice D pode ser visto o código-fonte utilizado para a IA de exemplo fornecida com o *framework*. Esse agente, criado apenas com o intuito de demonstrar a utilização do *framework*, utiliza algoritmos mais simples, baseado em uma máquina de estados finitos (e.g. decisão de atacar quando encontrar uma unidade inimiga) com algumas decisões aleatórias (e.g. movimentação das unidades enquanto procuram adversários).

A implementação de um novo agente de IA pode abranger todos os componentes apresentados na Seção 2.2, mas acreditamos que o principal alvo das pesquisas serão os componentes de análise tática e decisões estratégicas. Na implementação, podem ser utilizados tanto algoritmos simples, como as máquinas de estado utilizadas no exemplo, quanto outros mais complexos, como técnicas de aprendizado de máquina.

4.11 Comparação com Outras Ferramentas

Neste capítulo apresentamos a arquitetura do *framework* e as principais decisões de implementação tomadas durante o desenvolvimento. Esta arquitetura foi criada a partir das principais vantagens e desvantagens observadas em cada uma das ferramentas analisadas no Capítulo 2. Nesta seção vamos explicar quais características foram adaptadas para o *framework*, quais foram evoluídas e quais foram propostas de forma diferente do que se encontrava nas demais ferramentas.

A arquitetura contendo a IA em um módulo totalmente independente foi inspirada no *C-Evo*, mas no *framework* foi utilizada uma interface mais simples e direta para comunicação entre o núcleo do jogo e as IAs. No *C-Evo*, não há um módulo para os recursos compartilhados do jogo, o que faz com que alguns elementos sejam referenciados por estruturas de mais baixo nível (e.g. inteiros representando o tipo da unidade, ao invés de um enumerador, classe ou estrutura). A partir dessa lacuna que percebemos a necessidade do módulo *Common*, referenciado por todos os demais

módulos do *framework*.

A competição nativa entre diferentes IAs também foi inspirada no *C-Evo*, bem como a possibilidade da importação de IAs com base apenas em uma DLL, sem a necessidade de se ter o código-fonte para compilá-la. Os recursos para segurança e validação de jogadas foram criados para que no *framework* não fosse possível efetuar trapaças, como é possível fazer no *C-Evo*.

A dificuldade encontrada em realizar pequenas alterações no *FreeCiv* devido ao grande acoplamento do código influenciou na decisão de tornar as funcionalidades bem definidas e independentes. Um ponto positivo que buscamos aproveitar foi que todos os recursos do *FreeCiv* são gratuitos (enquanto as ferramentas para criação do *C-Evo* são proprietárias, embora o código seja aberto), uma vez que utiliza a linguagem C. Buscando uma linguagem que pode ser utilizada apenas com ferramentas gratuitas, mas que seja mais moderna e com recursos de mais alto nível que a linguagem C, optamos pela utilização do C#, no .Net Framework 4.0, conforme já explicamos no início do capítulo.

Por fim, o SDK do *Civilization* não influenciou de forma significativa nas decisões sobre a arquitetura do *framework*, pois como se trata de um código proprietário não havia como avaliar a forma como os mecanismos estavam construídos. Um ponto positivo que pode vir a ser incorporado futuramente ao *framework*, no entanto, é a utilização de scripts em LUA, que pode tornar a implementação dos métodos de IA ainda mais flexível.

Capítulo 5

Avaliação

Uma vez concluída toda a implementação do *framework*, conduzimos uma avaliação qualitativa com o objetivo de verificar se o ele atende aos objetivos propostos. Uma vez que o objetivo (i) do trabalho – permitir a realização de confrontos entre diferentes agentes de IA sem a necessidade de um jogador humano – faz parte da própria estrutura do *framework* e uma avaliação não faria sentido, o foco na avaliação foi concentrado nos outros três objetivos: (ii) possibilitar o desenvolvimento de algoritmos de IA sem a necessidade de se escrever muito código não relacionado ao algoritmo utilizado, (iii) possuir arquitetura bem modularizada de forma que a IA esteja em módulos independentes do restante do jogo e não possa trapacear e (iv) conter um jogo TBS completo, que contemple os principais elementos existentes em outros jogos do gênero, de forma que os experimentos realizados sejam aplicáveis em jogos reais.

A avaliação foi realizada com cinco usuários com diferentes níveis de experiência em relação a jogos TBS, Inteligência Artificial, modificações de jogos, linguagem de programação C# e diferentes níveis de escolaridade. Um dos usuários foi escolhido para executar um teste piloto para validar a metodologia adotada.

5.1 Metodologia

A avaliação é composta pelos seguintes passos: explicação do projeto e do escopo teste, apresentação e aceite de um termo de consentimento, preenchimento de um questionário pré-teste, execução das atividades da avaliação e preenchimento de um questionário pós-teste. O material utilizado para a avaliação está disponível no

Apêndice C.

O termo de consentimento é requerido pela legislação brasileira para condução de experimentos que envolvam participação humana. Ao fornecer o consentimento, os usuários concordam com a participação anônima e voluntária na avaliação, sem qualquer bônus ou ônus pela participação ou não.

O objetivo do questionário pré-teste é traçar o perfil dos usuários, de acordo com a faixa etária, escolaridade e contatos prévios com desenvolvimento de jogos, inteligência artificial, jogos de estratégia por turnos, modificações em jogos e a linguagem de programação utilizada.

Para a execução das tarefas da avaliação, os usuários receberam o código-fonte do *framework*, a documentação acerca das regras do jogo (correspondente ao Capítulo 4 desta dissertação), um exemplo de *log* de uma partida (vide Seção 4.8.1), instruções acerca dos softwares necessários (de acordo com o que foi apresentado na Seção 4.1), explicação resumida da arquitetura do *framework* e a descrição das tarefas que deveriam ser executadas.

No teste piloto, não havíamos fornecido o *log* de exemplo, mas ao final da avaliação percebemos que a ausência da interface gráfica havia sido muito sentida. Quando perguntamos sobre isso para o usuário, ele respondeu que um *log* de exemplo poderia ter ajudado a entender as mensagens do *log* e a execução das tarefas. Além disso, a explicação acerca da arquitetura do *framework* havia sido muito superficial, e ele sentiu falta de uma documentação mais detalhada do funcionamento. Por restrições de tempo, não poderíamos aguardar a finalização da documentação mais técnica (Capítulo 4 desta dissertação) para prosseguir com as avaliações, e então passamos a fazer uma explicação um pouco mais detalhada (embora ainda bastante resumida) do que a que havia sido feita para o teste piloto.

O código-fonte do *framework* disponibilizado para a avaliação estava parametrizado para partidas entre dois jogadores, sendo um a IA construída pelo usuário e o adversário a IA de exemplo fornecida com o *framework*.

As tarefas escolhidas para compor a avaliação foram (1) criar uma unidade em uma construção, (2) localizar uma construção inimiga no mapa e (3) destruir uma unidade inimiga em uma batalha. Como uma avaliação longa demais poderia dificultar

que conseguíssemos voluntários, estas tarefas foram escolhidas por não exigirem a construção de algoritmos muito complexos, mas demandarem implementação suficiente para que o usuário adquirisse o conhecimento necessário de como trabalhar no *framework*, de forma que estivesse apto a responder às questões de avaliação no questionário pós-teste. A ordem das tarefas foi definida de forma que a dificuldade esperada para execução das tarefas seja crescente, ou seja, espera-se que a tarefa (1) seja a mais fácil e a tarefa (3) a mais difícil de ser executada.

Após a conclusão das tarefas da avaliação por parte dos usuários, eles então responderam ao questionário pós-teste com o objetivo de coletar a opinião acerca de diversos aspectos do *framework* e a aderência do trabalho aos objetivos propostos. Aqui também houve uma pequena modificação após o teste piloto. Inicialmente, a pergunta acerca do tempo necessário para execução das tarefas (pergunta 4 do questionário pós-teste) possuía granularidade muito baixa, e a menor resposta possível era “menos de 5 horas”. No entanto, enquanto coletávamos as opiniões no questionário pós-teste percebemos que seria mais interessante alterar as respostas para a forma final apresentada no Apêndice C. Além disso, após o teste piloto também foi incluída a pergunta 12 no questionário, acerca da dificuldade esperada para executar tarefas mais complexas do que as solicitadas na avaliação.

5.2 Perfil dos Usuários Avaliadores

Para a avaliação piloto, escolhemos dentre os usuários candidatos à avaliação o que possuía maior experiência com a área de IA e possuía conhecimento em jogos TBS, o que nos permitiria obter um indicador mais preciso acerca do teste. Este usuário avaliador – que chamaremos de U0 – está na faixa etária de 21 a 25 anos, cursando pós-graduação e possuía muito contato tanto com a área de desenvolvimento de jogos quanto com a de inteligência artificial. Já conhecia jogos TBS – jogou um pouco – e o contato com modificações de jogos e com a linguagem C# era pequeno.

Todos os demais usuários avaliadores – que chamaremos de U1, U2, U3 e U4 – já haviam jogado jogos TBS bastante, com a exceção de U3, que respondeu ter jogado um pouco. No entanto, após o início da avaliação U1 pediu para reconsiderar sua resposta

como tendo jogado um pouco.

Em relação à faixa etária metade (U1 e U4) estava na faixa de 21 a 25 anos e a outra metade (U2 e U3) na faixa de 26 a 30 anos. Dois deles (U1 e U3) possuíam pós-graduação completa (o primeiro com mestrado e o segundo com especialização), enquanto U2 estava cursando graduação e U4 possuía graduação completa.

Quanto ao contato com a área de desenvolvimento de jogos, U1 e U4 responderam possuir pouco contato, enquanto U2 e U3 responderam possuir muito contato. No entanto, com base nas suas experiências relatadas por U1, reconsideramos sua resposta para incluí-lo no grupo com muito contato. Em relação ao contato com a área de inteligência artificial, todos responderam possuir pouco contato.

U2 não possuía contato com modificações de jogos existentes, e os outros três possuíam pouco contato. Por fim, em relação à experiência com a linguagem de programação C#, dois deles (U1 e U4) possuíam muita experiência, enquanto U3 possuía experiência média e U2 nenhuma experiência.

Usuário	Idade	Escolaridade	Desenvolvimento de Jogos	Inteligência Artificial	Jogos TBS	Modificações de Jogos	Linguagem C#
U0	21 a 25 anos	Pós-graduação incompleta	Muito contato	Muito contato	Já jogou um pouco	Pouco contato	Pouca experiência
U1	21 a 25 anos	Pós-graduação completa	Pouco contato	Pouco contato	Já jogou bastante	Pouco contato	Muita experiência
U2	26 a 30 anos	Ensino superior incompleto	Muito contato	Pouco contato	Já jogou bastante	Nenhum contato	Nenhuma experiência
U3	26 a 30 anos	Pós-graduação completa	Muito contato	Pouco contato	Já jogou um pouco	Pouco contato	Experiência média
U4	21 a 25 anos	Ensino superior completo	Pouco contato	Pouco contato	Já jogou bastante	Pouco contato	Muita experiência

Figura 5.1: Perfil dos avaliadores.

5.3 Resultados

Para a análise dos resultados, vamos considerar os avaliadores divididos em dois grupos, de acordo com a sua experiência prévia em desenvolvimento de jogos. Assim, os usuários U1 e U4, que relataram pouco contato com essa área, compõem o primeiro

grupo (ou G1), e os usuários U2 e U3, com muito contato com a área, compõem o segundo grupo (ou G2). Os resultados obtidos no teste piloto por U0 não são considerados de nenhum grupo.

Na Figura 5.2 podemos ver o sumário das respostas às perguntas de múltipla escolha respondidas no questionário pós-teste. As questões 2, 12 e 15 não se encontram na figura por se tratarem de questões de resposta aberta. O questionário completo com as perguntas e as respostas possíveis se encontram no Apêndice C.

	Perguntas													
Usuário	1	3	4	5	6	7	8	9	10	11	13	14	16	17
U0	b	c	a	d	a	a	d	c	e	-	b	b	b	d
U1	b	a	e	d	a	a	b	b	b	a	a	b	b	e
U2	b	c	c	d	a	b	c	a	d	b	b	b	b	d
U3	b	b	b	d	b	c	a	b	d	b	b	a	b	e
U4	b	b	a	d	b	d	c	b	d	b	c	b	b	e

Figura 5.2: Sumário das respostas às perguntas de múltipla escolha.

As três primeiras perguntas, na primeira parte da avaliação, eram relativas ao jogo Território. Em primeiro lugar, todos os usuários consideram que Território pode ser considerado um representante significativo de um jogo TBS. Em relação à dificuldade de aprendizado das regras do jogo (terceira pergunta), U0 considerou a dificuldade média, G1 considerou fácil ou muito fácil, e G2 considerou fácil ou média.

O tempo que cada um levou para concluir a avaliação – o que considera tanto o tempo para entendimento e preparação quanto o tempo de implementação – variou bastante, especialmente em G1. U0 concluiu as tarefas em menos de 5 horas (lembrando que, na avaliação piloto, essa resposta era a menor possível) e G2 levou de 2 a 10 horas. Já em G1, U1 concluiu as tarefas em mais de 20 horas, enquanto U4 precisou de menos de 2 horas.

Todos os usuários conseguiram executar as três tarefas. A primeira (criar uma nova unidade) foi considerada muito fácil por U0, enquanto tanto em G1 quanto em G2 foi considerada fácil ou muito fácil. Já a segunda tarefa (encontrar uma construção inimiga) foi considerada muito fácil por U0, mas apresentou resultados dispersos em G1 e G2. Em G1, U1 considerou muito fácil, mas U4 considerou difícil. Já em G2, U2

considerou fácil, enquanto U3 considerou média. Por fim, a terceira e última tarefa (destruir uma cidade inimiga) foi considerada difícil por U0, mas fácil ou média por G1 e G2.

O nível de dificuldade na implementação da IA no *framework* foi considerado médio por U0, mas fácil ou muito fácil em G1 e G2.. A IA de exemplo foi muito utilizada como base para a implementação nas avaliações por todos os usuários, com a exceção de U1 (pertencente a G1), que se baseou pouco nela. Na escala de 1 (pouco) a 5 (muito) em relação a quanto a implementação foi baseada nela, quatro avaliadores (U0, U2, U3 e U4) basearam-se muito (respostas 5, 4, 4 e 4, respectivamente), enquanto U1 baseou-se pouco (resposta 2).

Tanto em G1 quanto em G2, os usuários acreditam que seria fácil ou muito fácil utilizar o *framework* para tarefas mais complexas. Como já mencionamos anteriormente, no teste piloto essa pergunta não havia sido feita, por isso não há resposta para U0. O único avaliador que já havia feito ou tentado fazer em outras ferramentas algo parecido com o que foi feita nas avaliações foi U3, pertencente a G2.

Em relação à produtividade da implementação no *framework*, U0 precisou escrever pouco código além do estritamente necessário para o funcionamento dos algoritmos, o mesmo resultado obtido em G2. Em G1, U1 precisou de muito pouco (resposta 1) e U4 de uma quantidade média (resposta 3).

Por fim, todos responderam que o *framework* pode ajudar na implementação e pesquisa de IA em jogos, sendo U0 considera a utilização válida, G1 considera muito válida e G2 considera válida ou muito válida.

5.4 Discussão

Avaliando o jogo Território, acreditamos que o objetivo foi alcançado de forma bastante satisfatória, uma vez que todos os avaliadores responderam afirmativamente à primeira pergunta, o que foi também complementado por U4 de forma bastante interessante na resposta à segunda pergunta: “Não é necessário um conjunto de regras

muito complexo para se criar um bom jogo de estratégia por turnos. Os jogos 'Combate' (jogo de tabuleiro) e '*Battle for Wesnoth*' (jogo multiplataforma open-source desenvolvido em LUA) mostram que 'Território' não precisa de mais regras e ou elementos. Além disso, o jogo pode ser expandido sem muitas dificuldades dado a maneira em que foi desenvolvido." Também acreditamos que as regras do jogo são claras e razoavelmente fáceis de compreender, uma vez que nenhum dos avaliadores achou a curva de aprendizado difícil. Complementando a resposta, U1 disse em resposta à segunda pergunta que "as regras [do jogo Território] são bem claras e bem definidas".

Uma vez que todos os avaliadores conseguiram concluir as tarefas com sucesso, a questão mais divergente em toda a avaliação foi o tempo gasto para conclusão das tarefas, especialmente pelo avaliador U1, no grupo G1, que gastou mais de 20 horas. Neste caso, especificamente, o motivo não se deveu a dificuldades encontradas na execução das tarefas propriamente ditas, uma vez que ele considerou a execução de todas as tarefas fáceis ou muito fáceis. Tampouco pode ser atribuído a dificuldades encontradas no próprio *framework*, uma vez que considerou a implementação fácil. O motivo foi, na verdade, que ele decidiu extrapolar o escopo da própria avaliação e criou uma IA muito mais avançada que a criada pelos demais avaliadores. Ele explicou que pesquisou um pouco acerca de IA nos jogos TBS para poder criar algo mais interessante e complexo do que o solicitado na avaliação. U4, o outro usuário do grupo G1, possuía muita experiência tanto com jogos TBS quanto com a linguagem C#, o que explica a execução em tempo menor do que os demais. Já em G2, no entanto, o tempo gasto esteve dentro do que esperávamos para desenvolvedores com alguma experiência em jogos, nas faixas de 2 a 5 horas ou de 5 a 10 horas.

Sobre a dificuldade de implementação no *framework* de forma geral, os resultados também foram muito interessantes, uma vez que apenas o avaliador do teste piloto (U0) não considerou a implementação fácil ou muito fácil. Acreditamos que, em boa parte, isso se deve às explicações muito superficiais fornecidas no teste piloto (como já explicamos na seção anterior), o que foi melhorado para as avaliações posteriores. Um aspecto bastante interessante é que a IA fornecida junto com o *framework* para servir de exemplo para a implementação foi bastante utilizada como referência nas implementações, e com isso acreditamos que serviu bem ao seu propósito. As respostas à pergunta 11 também corroboram com a facilidade de implementação, uma vez que

todos os avaliadores acreditam que a implementação de tarefas mais complexas seria fácil ou muito fácil.

Quanto à produtividade oferecida pelo *framework*, os resultados também são bastante satisfatórios. Tanto U0 quanto os avaliadores do G2 precisaram de pouco ou muito pouco código não relacionado ao próprio algoritmo que estavam implementando. A exceção foi apenas U4, pertencente a G1, que por sua vez apresentou a sugestão de incluir no *framework* alguns comportamentos largamente utilizados, "como atacar a unidade com menos pontos de vida, ou se movimentar para uma posição adjacente a uma unidade amiga", o que explica a resposta não ter sido a mesma que os demais avaliadores. Ainda em G1, U1 sugeriu a criação de uma interface gráfica, o que tornaria a depuração mais fácil. Acreditamos que as duas sugestões são interessantes, e ambas foram incorporadas aos próximos passos do trabalho, que serão apresentados no próximo capítulo.

Em resposta à última pergunta (comentários gerais do avaliador), não há uma consolidação em grupos por ser uma resposta aberta à opinião geral dos avaliadores, conforme apresentado a seguir: U0 "acha que a ideia do projeto é muito interessante". U1 "gostaria muito de ver a continuidade do projeto", e sugeriu a utilização do *framework* para criar competições entre alunos de disciplinas de IA. U2 acredita que "um *framework* de IA pode ser útil no sentido de estabelecer boas práticas e incentivar o reuso." U4 considerou que "o processo de avaliação foi muito bem definido e facilitou o andamento do desenvolvimento", e elogiou a arquitetura utilizada para os turnos através da classe *AIBasePlayer*.

Finalmente, com base na análise realizada sobre as avaliações levando-se em consideração a experiência prévia dos usuários com a área de desenvolvimento de jogos, acreditamos que os resultados obtidos com o trabalho foram muito bons. Alguns pontos para melhoria foram levantados, especialmente em relação à criação da interface gráfica, e foram analisados e já incorporados aos próximos passos do projeto.

5.5 Expansão da Avaliação

Devido às restrições de tempo para conclusão deste trabalho, não foi possível realizar uma avaliação mais abrangente do *framework*, e por isso foi feita apenas a avaliação inicial, com foco qualitativo, apresentada neste capítulo, e que contou com apenas cinco avaliadores. No entanto, acreditamos que a realização de uma nova avaliação, com um grupo maior e mais homogêneo de avaliadores, seria de grande importância para se obter resultados estatisticamente significativos.

Propomos que essa avaliação seja realizada com uma turma de alunos de graduação em uma disciplina relacionada à IA ou ao desenvolvimento de jogos, com uma amostra desejada de 20 a 30 alunos. Com essa amostra mais significativa, deseja-se avaliar o *framework* de forma mais profunda do que foi possível realizar na avaliação inicial, para verificar a aderência do *framework* quanto à sua proposta original.

Para isso, pretendemos analisar os dados sob a ótica de três dimensões: de acordo com a experiência prévia dos avaliadores quanto a desenvolvimento de jogos, inteligência artificial e jogos de estratégia por turnos. O objetivo é verificar, em cada nível de conhecimento dessas áreas, as dificuldades e facilidades encontradas para uso do *framework*, de forma que seja possível traçar de forma mais precisa os pontos fortes e fracos da arquitetura e da implementação do *framework*.

Tendo em vista esse perfil de avaliação, o questionário pré-teste será reduzido para possuir apenas as perguntas 3, 4 e 5. Acreditamos que as demais perguntas (escolaridade, idade, experiência com C# e experiência com modificações de jogos) não são necessárias por não influenciarem significativamente nos elementos necessários para a avaliação.

Por fim, as tarefas solicitadas na avaliação serão as mesmas, pois na avaliação inicial percebemos que já demandaram até 10 horas dos avaliadores, o que é um tempo razoavelmente grande para exigir a participação na avaliação. Se a avaliação for parte de atividades da própria disciplina, pode ser expandida com uma quarta tarefa – “criar uma IA capaz de vencer o jogo” – o que pode ser associado a algum bônus extra para os alunos que conseguirem conclui-la.

Capítulo 6

Conclusões

Neste capítulo, apresentamos um resumo das contribuições desta dissertação e algumas possibilidades de trabalhos futuros que podem ser realizados a partir deste.

6.1 Resumo dos resultados

Neste trabalho foi desenvolvido um *framework* para a experimentação de algoritmos de Inteligência Artificial em jogos de estratégia por turnos. A necessidade de criação dessa ferramenta foi observada com base na análise e avaliação de três ferramentas disponíveis atualmente.

Com base nos pontos fortes e fracos levantados durante a avaliação realizada sobre essas ferramentas, propusemos a criação de um *framework* para pesquisa e experimentação de algoritmos de IA em jogos TBS que atenda os seguintes objetivos: (i) permitir a realização de confrontos entre diferentes agentes de IA sem a necessidade de um jogador humano; (ii) possibilitar o desenvolvimento de algoritmos de IA sem a necessidade de se escrever muito código não relacionado ao algoritmo utilizado; (iii) possuir arquitetura bem modularizada de forma que a IA esteja em módulos independentes do restante do jogo e não possa trapacear; (iv) conter um jogo TBS que contemple os principais elementos existentes em outros jogos do gênero, de forma que os experimentos realizados sejam aplicáveis em jogos reais.

Tendo em vista esses objetivos, foi criado o jogo Território, com arquitetura

especialmente planejada tendo em vista o desenvolvimento de algoritmos de IA. Após a conclusão do desenvolvimento, realizamos uma avaliação com cinco usuários para medir o quanto o jogo e o *framework* criados atendiam aos objetivos propostos. Os resultados obtidos foram muito bons, e com isso acreditamos que o *framework* atende aos objetivos propostos e se mostra como uma ferramenta para a realização de experimentos de IA em jogos TBS.

Em relação aos objetivos propostos, o *framework* já foi construído para realizar confrontos entre diferentes jogadores controlados por IA (objetivo i), e a arquitetura foi construída de forma que a implementação da IA seja totalmente independente do restante do jogo (objetivo iii). Com base nos resultados obtidos nas avaliações, os usuários relataram não necessitar de muito código para implementação no *framework* (objetivo ii) e concordaram de forma unânime que o jogo Território representa os jogos TBS de forma bastante satisfatória (objetivo iv).

Por fim, com base nos resultados obtidos acreditamos que o *framework* desenvolvido atingiu os objetivos esperados e poderá ser utilizado na realização de trabalhos e pesquisas de IA em jogos TBS.

6.2 Trabalhos futuros

Durante a preparação deste trabalho, especialmente nas etapas de implementação do *framework* e avaliação, percebemos alguns pontos onde o trabalho pode ser expandido ou melhorado. A seguir apresentamos os principais pontos identificados e de que forma podem ser explorados em trabalhos futuros.

1. Criar uma **interface gráfica** para o jogo Território, o que tornaria o uso do *framework* muito mais simples e intuitivo. Por se tratar de um jogo, a ausência de uma interface gráfica pode intimidar alguns pesquisadores, mesmo que o uso do *framework* seja muito fácil;
2. Usar o *framework* em **experimentos** com um grupo maior de usuários, como para realização de trabalhos em disciplinas relacionadas IA e jogos, o que permitiria avaliar os benefícios em uma escala bem maior do que a apresentada

neste trabalho;

3. **Referenciar dinamicamente as IAs** que são utilizadas na partida, permitindo alterar quais serão utilizadas sem a necessidade de recompilar o projeto. Essa alteração também irá facilitar bastante o uso do *framework* em competições que envolvam vários participantes, pois poderão ser usados mecanismos automatizados para configuração e execução das partidas. Com isso, será também uma tarefa mais fácil a utilização do *framework* para competições entre IAs.
4. Traduzir a **documentação** do jogo (Capítulo 3) e do *framework* (Capítulo 4) para o inglês, mesmo idioma usado no código-fonte e nos comentários. Acreditamos que isso seja muito importante para a publicação e distribuição do *framework* para uso global;
5. Expandir as regras do jogo para considerar estratégias com **aspectos globais**. Dentro da estrutura atual do jogo, já temos duas possibilidades de expansão nesta linha: (1) fazer com que o domínio de territórios influencie em outros aspectos do jogo, como a capacidade de produção das construções ou a quantidade de unidades que o jogador pode ter em um determinado momento; (2) criação de tecnologias que podem ser produzidas em conjunto pelas cidades e assim desbloqueiam a produção de novas unidades ou itens;
6. Adaptar a arquitetura do *framework* para que se crie um **motor genérico para IA em jogos TBS**. Para isso, é necessário dividir o núcleo do jogo em dois módulos, onde o principal manteria a mecânica de funcionamento genérica, e o novo módulo consolidaria as regras do jogo, de forma que seja fácil efetuar expansões ou alterações;
7. Incluir alguns **métodos implementados pela IA de exemplo** em classes utilitárias que fiquem disponíveis para todos os jogadores, o que pode agilizar a implementação de comportamentos comuns tornando a implementação no *framework* ainda mais produtiva.
8. Melhorar alguns **aspectos técnicos** do *framework*, como o desempenho dos métodos de desenho para quando a interface gráfica estiver criada, pois pode-se reduzir o custo de cópia do mapa do jogo a cada quadro, como é feito atualmente. Outro ponto passível de melhorias é o algoritmo de escolha das posições iniciais dos jogadores, para que as posições escolhidas aleatoriamente

sejam menos previsíveis e sejam bem balanceadas quando houver mais do que quatro jogadores na partida.

Referências bibliográficas

[Bergsma & Spronck, 2008] Bergsma, Maurice & Spronck, Pieter. (2008). Adaptative Intelligence for Turn-Based Strategy Games. In *Proceedings of the twentieth Belgian-Dutch Conference on Artificial Intelligence (BNAIC)*, pp. 17-24, Enschede, Netherlands

[Black, 2006] Black, Paul E. (2006). *Dictionary of Algorithms and Data Structures*, disponível em http://www.nist.gov/dads/HTML/manhattanDistance.html. Visitado em 27/06/2012.

[Cheng & Thawonmas, 2004] Cheng, Danny C. & Thawonmas, Ruck. (2004). Case-based plan recognition for real-time strategy games. In *Proceedings of the Fifth Game-On International Conference*, pp. 36-40, Reading, UK. University of Wolverhampton Press.

[Cunha, 2010] Cunha, Renato Luiz de Freitas. (2010). Um sistema de apoio ao jogador para jogos de Estratégia em Tempo Real. *Dissertação de Mestrado*, UFMG

[Díaz-Agudo et al., 2005] Diáz-Agudo, Belén; Sánchez-Pelegrín, Rubén. (2005). An Intelligent Decision Module based on CBR for C-evo. *Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*. pp. 90–94, Edinburgh, Scotland

[Djikstra, 1959] Djikstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik* **1**: 269-271

[Fairclough et al., 2001] Fairclough, Chris; Fagan, Michael; Namee, Brian Mac & Cunningham, Pádraig. (2001). Research Directions for AI in Computer Games. In *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science (AICS)*, pp. 333-344, Maynooth, Ireland

[Forbus & Hinrichs, 2006] Forbus, Kenneth D.; Hinrichs, Thomas R. (2006). Companion Cognitive Systems: A Step toward Human-Level AI. *AI Magazine*, **27**(2), pp. 83 - 95

[Hart et al., 1968] Hart, P. E.; Nilsson, N. J. & Raphael, B. (1968). A Formal Basis for

the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems, Science and Cybernetics (SSC4)*, **4**(2): 100-107

[Hinrichs & Forbus, 2007] Hinrichs, Thomas R.; Forbus, Kenneth D. (2007). Analogical Learning in a Turn-Based Strategy Game. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pp. 853 – 858, Hyderabad, India

[Jones & Goel, 2009] Jones, Joshua; Goel, Ashok. (2009). Metareasoning for adaptation of classification knowledge. *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, **2**: pp.1145 - 1146

[Laird & Jones, 1998] Laird, J. E. & Jones R. M. (1998). Building advanced autonomous AI systems for large scale real time simulations. In *Proceedings of the 1998 Computer Game Developers' Conference*, pp. 365-378

[Laird & van Lent, 2000] Laird, J. E. & van Lent, M. (2000). Human-level AI's killer application: Interactive computer games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pp. 1171-1178. AAAI Press / The MIT Press

[Machado et al., 2011] Machado, Marlos C.; Rocha, Bruno S. L.; Chaimowicz, Luiz (2011). Agents Behavior and Preferences Characterization in Civilization IV. *Proceedings of the Brazilian Symposium on Computer Games and Digital Entertainment, Computing Track.*

[Malone, 1981] Malone, Thomas. (1981). What makes computer games fun? (abstract only). In *Proceedings of the joint conference on Easier and more productive use of computers (CHI'81)*, volume 1981, pp. 143, Ann Arbor, USA

[Millington & Funge, 2009] Millington, Ian & Funge, John. (2009). *Artificial Intelligence for Games*. Morgan Kauffman, 2a edição. ISBN 978-0-12-374731-0

[Ponsen, 2004] Ponsen, Marc. (2004). Improving Adaptative Game AI With Evolutionary Learning. Master Thesis, Delft University of Technology

[Potisartra & Katrajaras, 2009] Potisartra, Kittisak & Kotrajaras, Vishnu. (2009). Towards an Evenly Match Opponent AI in Turn-based Strategy Games.

Computer Games, Multimedia and Allied Technology 2009 Conference (CGAT).

[Reynolds, 1987] Reynolds, C. W. (1987). Flocks, Herds, and Schools: A distributed behavioral model. *Computer Graphics*, **21**(4): 25-34

[Russel & Norwig, 2003] Russel, S. J. & Norwig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2a edição. ISBN 0-13-790395-2

[Salge et al., 2008] Salge, Christoph; Lipski, Christian; Mahlmann, Tobias & Mathiak, Brigitte. (2008). Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pp. 7-14.

[Sanchéz-Pelegrín et al., 2005] Sanchéz-Pelegrín, Rubén; Gómez-Martín, Marco Antonio; Díaz-Agudo, Belén. (2005). A CBR Module for a Strategy Videogame. *1st Workshop on Computer Gaming and Simulation Environments, at 6th International Conference on Case-Based Reasoning (ICCBR)*

[Sanchéz-Ruiz et al., 2007] Sanchéz-Ruiz, Antonio; Lee-Urban, Stephen; Muñoz-Avila, Héctor; Díaz-Agudo, Belén; González-Calero, Pedro. (2007). Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based Retrieval. In *Proceedings of the ICAPS 2007 Workshop on Planning in Games*.

[Schaeffer et al., 2007] Schaeffer, J.; Burch, N.; Björnsson, Y.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P. & Stutphen, S. (2007). Checkers is solved. *Science*, **317**: 1518-1522

[Siwek, 2010] Siwek, Stephen E. (2010). Video Games in the 21st Century: The 2010 Report. *Entertainment Software Association*

[Spronck & Teuling, 2010] Spronck, Pieter; Teuling, Freek den. (2010). Player Modeling in Civilization IV. *Association for the Advancement of Artificial Intelligence*

[Wender & Watson, 2008] Wender, S.; Watson, I. (2008). Using reinforcement learning for city site selection in the turn-based strategy game Civilization IV. In *IEE Symposium on Computational Intelligence and Games (CIG)*, pp. 372-377

Apêndice A – Classes Públicas do Módulo Common

Neste apêndice apresentamos os enumeradores, constantes, propriedades e métodos públicos e protegidos de todas as classes públicas do módulo *Common*. Foram omitidas apenas as classes que já foram completamente explicadas e detalhadas no Capítulo 4. Também foram omitidas as exceções (que herdam da classe *Exception*), pois estas não apresentam grande relevância para o entendimento e manipulação das classes, e contribuiriam apenas para tornar o apêndice excessivamente longo. Ao lado do nome de cada classe, entre parênteses, encontra-se o *namespace* correspondente.

A.1 Board (Common.Resources)

- **public int Width:** Retorna a largura do mapa, de acordo com o tamanho da dimensão X;
- **public int Height:** Retorna a altura do mapa, de acordo com o tamanho da dimensão Y;
- **public Tile GetTile(Position position):** Retorna o território (objeto do tipo "Tile") na posição (objeto do tipo Position) desejada do mapa. As classes "Tile" e "Position" serão explicadas nas próximas entidades;
- **public Tile GetTile(int xCoordinate, int yCoordinate):** Sobrecarga do método acima, que recebe as coordenadas X e Y separadamente ao invés de um objeto do tipo "Position";
- **public void NormalizeDomains():** Normaliza os domínios de todos os territórios do mapa. A princípio só precisa ser usado pelo núcleo do jogo, mas caso uma IA deseje realizar alguma simulação de jogadas no mapa o método pode ser chamado após alterações no domínio de vários territórios;
- **public bool ExistsPlayer(int playerID):** Retorna verdadeiro se, e somente se, determinado jogador estiver presente no mapa através de alguma unidade ou

construção. É importante ressaltar que quando este método for utilizado por um jogador, deve-se ter em mente que o mapa é parcialmente observável, e a não existência do jogador não indica que ele não esteja presente em pontos não visíveis do mapa;

- **public GameElement GetGameElement(ulong gameElementID):** Retorna o elemento de jogo existente no mapa com determinado ID. O método foi criado para auxiliar a busca de elementos no mapa, como no seguinte exemplo: Determinado jogador deseja mapear a movimentação de unidades inimigas, e pode então armazenar os IDs e as respectivas posições das unidades visíveis numa rodada para, na rodada seguinte, localizá-las através deste método;
- **public List<GameElement> GetGameElements(int playerID):** Retorna uma lista com todos os elementos pertencentes a determinado jogador e que estejam visíveis no mapa.

A.2 Position (Common.Resources)

- **public int X:** Retorna o valor da coordenada do eixo X;
- **public int Y:** Retorna o valor da coordenada do eixo Y;
- **public bool EvenColumn:** Retorna verdadeiro se, e somente se, a posição estiver em uma coluna par;
- **public Position North:** Retorna a posição do vizinho imediatamente ao norte da posição. A posição obtida possui coordenadas (X, Y-1);
- **public Position Northeast:** Retorna a posição do vizinho imediatamente a nordeste da posição. Quando a coluna da posição for par, as coordenadas a nordeste serão (X+1, Y-1), e quando a coluna for ímpar, as coordenadas a nordeste serão (X+1, Y);
- **public Position Southeast:** Retorna a posição do vizinho imediatamente a sudeste da posição. Quando a coluna da posição for par, as coordenadas a sudeste serão (X+1, Y), e quando a coluna for ímpar, as coordenadas a sudeste serão (X+1, Y+1);
- **public Position South:** Retorna a posição do vizinho imediatamente ao sul da posição. A posição obtida possui coordenadas (X, Y+1);

- **public Position Southwest:** Retorna a posição do vizinho imediatamente a sudoeste da posição. Quando a coluna da posição for par, as coordenadas a sudoeste serão (X-1, Y), e quando a coluna for ímpar, as coordenadas a sudoeste serão (X-1, Y+1);
- **public Position Northwest:** Retorna a posição do vizinho imediatamente a noroeste da posição. Quando a coluna da posição for par, as coordenadas a noroeste serão (X-1, Y-1), e quando a coluna for ímpar, as coordenadas a noroeste serão (X-1, Y);
- **public bool IsAdjacent(Position position):** Retorna verdadeiro se, e somente se, a posição passada por parâmetro for vizinha da posição corrente;
- **public List<Position> GetNeighbours():** Retorna a lista de vizinhos imediatos (ou seja, com distância igual a um) da posição. Neste método, não serão retornados os vizinhos inválidos, ou seja, aqueles que estão fora dos limites do mapa;
- **public List<Position> GetNeighbours(uint range):** Sobrecarga do método acima, retorna a lista de vizinhos da posição. A diferença é que não são retornados apenas os vizinhos imediatos, mas sim todos os vizinhos que estejam a uma determinada distância, de acordo com o parâmetro passado. Caso o valor do parâmetro passado seja igual a "1", o resultado é igual ao do método anterior.

A.3 Tile (Common.Resources)

- **public const double NORMALIZED_DOMAIN_SUM:** Constante que indica o valor da soma de todos os domínios de um território normalizado;
- **public TerrainType Terrain:** Retorna o tipo do terreno do território;
- **public List<Unit> Units:** Retorna a lista de unidades que estão no território;
- **public Building Building:** Retorna a construção que se encontra no território, se houver. Se não houver construção no território, será retornado *null*;
- **public Dictionary<int, double> Domain:** Retorna o mapa de domínio do território, onde a chave é o ID do jogador;
- **public void AddDomain(int playerID, double domain, bool normalizeAfterAdding):** Adiciona um valor ao domínio de um jogador sobre o

território. O parâmetro *normalizeAfterAdding* indica se o domínio do território deve ser normalizado logo após adicionar o domínio, e possui valor default igual a *true*;

- **public void NormalizeDomains():** Normaliza os domínios do território;
- **public void RemoveDomain(int playerID):** Remove do território o domínio de um jogador;
- **public bool ExistsPlayer(int playerID):** Retorna verdadeiro se, e somente se, determinado jogador estiver presente no território através de alguma unidade ou construção;
- **public List<GameElement> GetGameElements(int playerID):** Retorna uma lista com todos os elementos pertencentes a determinado jogador que estejam no território.

A.4 Terrain (Common.Resources.Terrains)

- **public enum TerrainType:** Enumerador que representa os tipos de terreno existentes;
- **public TerrainType TerrainType:** Retorna o tipo do terreno;
- **public uint MovementCost:** Retorna o custo de movimentação associado ao terreno;
- **public bool CanReceiveGroundUnits:** Indica se o terreno pode receber unidades terrestres;
- **public bool CanReceiveWaterUnits:** Indica se o terreno pode receber unidades aquáticas;
- **public bool CanReceiveAerialUnits:** Indica se o terreno pode receber unidades aéreas;
- **public bool CanReceiveBuilding:** Indica se o terreno pode receber construções;
- **public static TerrainType RandomTerrainType:** Retorna um tipo de terreno pseudoaleatório;

A.5 Terrains (Common.Resources.Terrains)

- **public static Terrain Get(TerrainType type):** Retorna os atributos do tipo de terreno desejado;

A.6 Unit (Common.Resources.Units)

- **public enum UnitType:** Enumerador que representa os tipos de unidade existentes;
- **public int Owner:** Retorna o ID do jogador ao qual a unidade pertence;
- **public UnitType Type:** Retorna o tipo da unidade;
- **public HashSet<ItemType> Items:** Retorna os itens que estão equipados na unidade;
- **public Position Position:** Retorna a posição da unidade no mapa;
- **public double CurrentHealth:** Retorna a saúde atual da unidade;
- **public uint RemainingMovements:** Retorna a quantidade de movimentos que a unidade ainda pode realizar no turno corrente;
- **public ulong LastMovementTurn:** Retorna o último turno em que a unidade executou alguma ação;
- **public double MaxHealth:** Retorna a saúde máxima da unidade, levando-se em consideração tanto o tipo da unidade quanto os bônus fornecidos por itens equipados;
- **public uint Movements:** Retorna a quantidade de movimentos que a unidade pode realizar por turno, levando-se em consideração tanto o tipo da unidade quanto os bônus fornecidos por itens equipados;
- **public uint Attack:** Retorna o poder de ataque da unidade, levando-se em consideração tanto o tipo da unidade quanto os bônus fornecidos por itens equipados;
- **public uint Defense:** Retorna o poder de defesa da unidade, levando-se em

consideração tanto o tipo da unidade quanto os bônus fornecidos por itens equipados;

- **public uint Range:** Retorna o alcance das ações da unidade, levando-se em consideração tanto o tipo da unidade quanto os bônus fornecidos por itens equipados;
- **public uint Sight:** Retorna o alcance de visão da unidade, levando-se em consideração tanto o tipo da unidade quanto os bônus fornecidos por itens equipados;
- **public double InfluenceFactor:** Retorna o fator de influência da unidade, levando-se em consideração tanto o tipo da unidade quanto os bônus fornecidos por itens equipados;
- **public double MinimumRandomFactor:** Retorna o fator aleatório mínimo da unidade, levando-se em consideração tanto o tipo da unidade quanto os bônus fornecidos por itens equipados;

A.7 UnitAttributes (Common.Resources.Units)

- **public double MaxHealth:** Saúde máxima da unidade;
- **public uint Movements:** Quantidade de movimentos que a unidade pode realizar por turno;
- **public uint Attack:** Poder de ataque da unidade;
- **public uint Defense:** Poder de defesa da unidade;
- **public uint Range:** Alcance das ações da unidade;
- **public uint Sight:** Alcance de visão da unidade;
- **public double InfluenceFactor:** Fator de influência da unidade;
- **public double MinimumRandomFactor:** Fator aleatório mínimo da unidade;
- **public uint BuildingCost:** Custo de produção da unidade/item;

A.8 Units (Common.Resources.Units)

- **public static UnitAttributes Get(UnitType type):** Retorna os atributos do tipo de unidade desejado;

A.9 UnitItems (Common.Resources.Units)

- **public enum ItemType:** Enumerador que representa os tipos de item existentes;
- **public static UnitAttributes Get(ItemType type):** Retorna os atributos do tipo de item desejado;

A.10 Building (Common.Resources.Buildings)

- **public enum BuildingType:** Enumerador que representa os tipos de construção existentes;
- **public int Owner:** Retorna o ID do jogador ao qual a construção pertence;
- **public BuildingType Type:** Retorna o tipo da construção;
- **public Position Position:** Retorna a posição da construção no mapa;
- **public uint ProductionProgress:** Retorna o progresso da produção atual em curso na construção;
- **public ProduceableElement CurrentProduction:** Retorna o elemento em produção na construção;
- **public List<ItemType> UnitItemsInventory:** Retorna a lista de itens no inventário da construção;

A.11 BuildingAttributes (Common.Resources.Buildings)

- **public uint ProductionRate:** Retorna a taxa de produção da construção;

- **public double InfluenceFactor:** Retorna o fator de influência da construção;

A.12 Buildings (Common.Resources.Buildings)

- **public static BuildingAttributes Get(BuildingType type):** Retorna os atributos do tipo de construção desejado;

A.13 GameElement (Common.General)

- **public ulong ID:** Retorna o ID do elemento de jogo;

A.14 ProduceableElement (Common.Resources)

- **public object ProduceableElementType:** Retorna o tipo do elemento produzível, que pode ser uma unidade (UnitType) ou item (ItemType);
- **public uint BuildingCost:** Retorna o custo de produção do elemento produzível;

A.15 AIBasePlayer (Common.AIInterface)

- **public delegate PlayerCommandResponse PlayerCommandCallBack(PlayerCommand command, long turnValidationCode):** Define a assinatura do método que será usado para que o jogador envie as jogadas;
- **protected PlayerCommandCallBack CallBack:** Método que deve ser chamado pelo jogador para enviar as jogadas;
- **public int PlayerID:** Retorna o ID do jogador;
- **public abstract void PlayTurn(Board board, long turnValidationCode):**

Método chamado pelo módulo *Game* para acionar o jogador indicando que é sua vez de jogar;

A.16 PlayerCommand (Common.Commands)

Nos métodos para criação dos comandos nesta classe, o primeiro parâmetro é sempre "AIBasePlayer player", que corresponde ao jogador que está construindo o comando (basta passar o objeto *this* para o parâmetro), e por isso a explicação deste parâmetro será omitida para evitar redundância.

- **public enum CommandType:** Enumerador que representa os tipos de comando existentes;
- **public CommandType Command:** Retorna o tipo de comando;
- **public Unit Unit:** Retorna a unidade envolvida no comando, se houver;
- **public Position Destination:** Retorna a posição do destino do comando, se houver;
- **public int PlayerID:** Retorna o ID do jogador que está enviando o comando;
- **public ItemType? UnitItem:** Retorna o item envolvido no comando, se houver;
- **public Building Building:** Retorna a construção envolvida no comando, se houver;
- **public ProduceableElement Production:** Retorna o elemento produzível envolvido no comando, se houver;
- **public BuildingType? BuildingCreating:** Retorna o tipo de construção envolvido no comando, se houver;
- **public static PlayerCommand EndOfTurn(AIBasePlayer player):** Cria um comando do tipo "finalizar turno";
- **public static PlayerCommand MoveTo(AIBasePlayer player, Unit unit, Position destination):** Cria um comando do tipo "mover";
    - Unit unit: Unidade que fará o movimento;
        - Deve existir no mapa do jogo (a busca será feita pelo ID);

        - Deve pertencer ao jogador que está construindo o comando;
        - Deve possuir movimentos restantes no turno;
    - Position destination: Posição do território de destino da movimentação;
        - Deve ser válida, ou seja, estar dentro dos limites do mapa;
        - Deve ser diferente da posição atual da unidade;
        - O território do mapa na posição não pode possuir unidades pertencentes a outro jogador;
        - O custo de movimentação até o território de destino deve ser menor ou igual ao número de movimentos restantes da unidade no turno;

- **public static PlayerCommand AttackAt(AIBasePlayer player, Unit unit, Position destination):** Cria um comando do tipo "atacar";
    - Unit unit: Unidade que executará o ataque;
        - Deve existir no mapa do jogo (a busca será feita pelo ID);
        - Deve pertencer ao jogador que está construindo o comando;
        - Deve possuir movimentos restantes no turno;
    - Position destination: Posição do território de destino do ataque;
        - Deve ser válida, ou seja, estar dentro dos limites do mapa;
        - Deve ser diferente da posição atual da unidade;
        - O território do mapa na posição deve possuir unidades pertencentes a outro jogador;
        - A menor distância entre o território da unidade e o território de destino deve ser menor ou igual ao alcance de ataque da unidade;

- **public static PlayerCommand InfluenceAt(AIBasePlayer player, Unit unit, Position destination):** Cria um comando do tipo "influenciar território";
    - Unit unit: Unidade que executará a influência sobre o território;
        - Deve existir no mapa do jogo (a busca será feita pelo ID);
        - Deve pertencer ao jogador que está construindo o comando;
        - Deve possuir movimentos restantes no turno;
    - Position destination: Posição do território que sofrerá influência pela unidade;

        - Deve ser válida, ou seja, estar dentro dos limites do mapa;
        - A menor distância entre o território da unidade e o território de destino deve ser menor ou igual ao alcance de ataque da unidade;

- **public static PlayerCommand EquipUnit(AIBasePlayer player, Unit unit, ItemType itemType):** Cria um comando do tipo "equipar unidade";
    - Unit unit: Unidade que será equipada com o item;
        - Deve existir no mapa do jogo (a busca será feita pelo ID);
        - Deve pertencer ao jogador que está construindo o comando;
        - O território onde a unidade se encontra deve possuir uma construção;
    - ItemType itemType: Item que será equipado na unidade;
        - Não pode ser um item já equipado na unidade;
        - Deve estar presente no inventário da construção que estiver no mesmo território em que a unidade se encontra;

- **public static PlayerCommand SetProduction(AIBasePlayer player, Building building, ProduceableElement production):** Cria um comando do tipo "definir produção";
    - Building building: Construção onde será definida a produção;
        - Deve existir no mapa do jogo (a busca será feita pelo ID);
        - Deve pertencer ao jogador que está construindo o comando;
    - ProduceableElement production: Elemento que será produzido na construção;
        - Qualquer objeto da classe *ProduceableElement* será válido, podendo então ser um tipo de unidade ou tipo de item;

- **public static PlayerCommand CreateBuilding(AIBasePlayer player, Unit unit, BuildingType buildingType):** Cria um comando do tipo "criar construção";
    - Unit unit: Unidade que executará criará a construção;
        - Deve existir no mapa do jogo (a busca será feita pelo ID);
        - Deve pertencer ao jogador que está construindo o comando;
        - Deve ser do tipo *Priest* (sacerdote);

        - O jogador deve possuir domínio maior que 50% no território em que se encontra a unidade;
        - Não pode haver outra construção no território em que a unidade se encontra;
        - O território em que a unidade se encontra deve permitir a criação de construções (propriedade *CanReceiveBuildings*, já apresenta anteriormente);
    - BuildingType buildingType: Tipo da construção que será criada no território em que a unidade se encontra;
        - Não há restrições acerca dos tipos de construção que podem ser usados;

A.17 PlayerCommandResponse (Common.Commands)

- **public enum PlayerCommandResult:** Enumerador que representa os tipos de resposta de comando existentes;
- **public PlayerCommandResult Result:** Retorna o resultado da execução do comando;
- **public Board ResultBoard:** Retorna o mapa do jogo resultante da execução do comando, quando o comando não apresentou nenhum erro;
- **public Exception Error:** Retorna o erro ocorrido durante o processamento do comando, quando houver algum erro;

A.18 GameSettings (Common.Settings)

- **public static int NumberOfPlayers:** Retorna o número de jogadores na partida;
- **public static int BoardWidth:** Retorna a largura do mapa da partida;
- **public static int BoardHeight:** Retorna a altura do mapa da partida;
- **public static double GroupingUnitsBonus:** Retorna o bônus fornecido quando há unidades agrupadas no mesmo território (vide Seção 3.2.3);

- **public static uint GroupingUnitsThreshold:** Retorna o número de unidades necessárias no mesmo território para considerá-las agrupadas (vide Seção 3.2.3);
- **public static float HealthRecoverPerTurn:** Retorna o percentual da saúde da unidade que será recuperado a cada turno em que ela não executar nenhuma ação;

A.19 GameState (Common.Settings)

- **public static ulong Turn:** Retorna o número do turno atual da partida;

A.20 RandomUtil (Common.Util)

- **public static int NextInt:** Retorna um número inteiro de 32 bits;
- **public static long NextLong:** Retorna um número inteiro de 64 bit;
- **public static uint NextUInt:** Retorna um número inteiro não negativo de 32 bits;
- **public static double NextDouble:** Retorna um número real entre 0 e 1, incluindo os extremos;
- **public static double NextNormalizedDouble(double minimumValue, double maximumValue = 1):** Retorna um número real entre os valores passados por parâmetro, incluindo os extremos;

Apêndice B – Exemplo de log do jogo

O *log* a seguir foi obtido em uma partida completa entre dois jogadores utilizando um mapa.de dimensões 5x5.

2012-05-02 22:35:08 Player 1281130855 moved the unit 49 to (3, 0).

2012-05-02 22:35:08 Player 1281130855 set production of Marshal in the building 48.

2012-05-02 22:35:08 Player 1281130855 ended turn.

2012-05-02 22:35:09 Player 1210714868 moved the unit 47 to (1, 3).

2012-05-02 22:35:09 Player 1210714868 set production of Marshal in the building 46.

2012-05-02 22:35:09 Player 1210714868 ended turn.

2012-05-02 22:35:10 Player 1281130855 moved the unit 49 to (2, 0).

2012-05-02 22:35:10 Player 1281130855 ended turn.

2012-05-02 22:35:11 Player 1210714868 moved the unit 47 to (0, 4).

2012-05-02 22:35:11 Player 1210714868 moved the unit 47 to (1, 3).

2012-05-02 22:35:11 Player 1210714868 ended turn.

2012-05-02 22:35:12 Player 1281130855 moved the unit 49 to (1, 0).

2012-05-02 22:35:12 Player 1281130855 ended turn.

2012-05-02 22:35:13 Player 1210714868 moved the unit 47 to (1, 2).

2012-05-02 22:35:13 Player 1210714868 moved the unit 47 to (0, 2).

2012-05-02 22:35:13 Player 1210714868 ended turn.

2012-05-02 22:35:14 Player 1281130855 moved the unit 49 to (2, 0).

2012-05-02 22:35:14 Player 1281130855 ended turn.

2012-05-02 22:35:15 Player 1210714868 moved the unit 47 to (0, 3).

2012-05-02 22:35:15 Player 1210714868 ended turn.

2012-05-02 22:35:16 Player 1281130855 moved the unit 49 to (1, 0).

2012-05-02 22:35:16 Player 1281130855 ended turn.

2012-05-02 22:35:17 Player 1210714868 moved the unit 47 to (1, 3).

2012-05-02 22:35:17 Player 1210714868 ended turn.

2012-05-02 22:35:18 Player 1281130855 moved the unit 49 to (2, 1).

2012-05-02 22:35:18 Player 1281130855 ended turn.

2012-05-02 22:35:19 Player 1210714868 moved the unit 47 to (0, 3).

2012-05-02 22:35:19 Player 1210714868 ended turn.

2012-05-02 22:35:20 Player 1281130855 moved the unit 49 to (1, 0).

2012-05-02 22:35:20 Player 1281130855 ended turn.

2012-05-02 22:35:21 Player 1210714868 moved the unit 47 to (0, 4).

2012-05-02 22:35:21 Player 1210714868 moved the unit 47 to (1, 4).

2012-05-02 22:35:21 Player 1210714868 ended turn.

2012-05-02 22:35:22 Player 1281130855 moved the unit 49 to (2, 1).

2012-05-02 22:35:22 Player 1281130855 ended turn.

2012-05-02 22:35:23 Player 1210714868 moved the unit 47 to (1, 3).

2012-05-02 22:35:23 Player 1210714868 ended turn.

2012-05-02 22:35:24 Player 1281130855 moved the unit 49 to (3, 0).

2012-05-02 22:35:24 Player 1281130855 ended turn.

2012-05-02 22:35:25 Player 1210714868 moved the unit 47 to (2, 3).

2012-05-02 22:35:25 Player 1210714868 ended turn.

2012-05-02 22:35:26 Player 1281130855 moved the unit 49 to (2, 1).

2012-05-02 22:35:26 Player 1281130855 ended turn.

2012-05-02 22:35:27 Player 1210714868 moved the unit 47 to (3, 2).

2012-05-02 22:35:27 Player 1210714868 moved the unit 47 to (4, 2).

2012-05-02 22:35:27 Player 1210714868 ended turn.

2012-05-02 22:35:28 Player 1281130855 moved the unit 49 to (2, 0).

2012-05-02 22:35:28 Player 1281130855 ended turn.

2012-05-02 22:35:29 Player 1210714868 moved the unit 47 to (4, 1).

2012-05-02 22:35:29 Unit 47 destroyed the building 48 at (4, 0).

2012-05-02 22:35:29 Player 1210714868 moved the unit 47 to (4, 0).

2012-05-02 22:35:29 Player 1210714868 ended turn.

2012-05-02 22:35:30 Player 1281130855 moved the unit 49 to (3, 0).

2012-05-02 22:35:30 Player 1281130855 ended turn.

2012-05-02 22:35:31 Unit 47 attacked the unit 49 at (3, 0), damaged it by 0,577161761236216 and was damaged back by 0,422838238763784.

2012-05-02 22:35:31 Player 1210714868 attacked at (3, 0) with the unit 47.

2012-05-02 22:35:31 Player 1210714868 ended turn.

2012-05-02 22:35:32 Unit 49 died in the battle.

2012-05-02 22:35:32 Player 1281130855 was eliminated from the game.

2012-05-02 22:35:32 Unit 49 attacked the unit 47 at (4, 0), damaged it by 0,496767268165663 and was damaged back by 0,503232731834337.

2012-05-02 22:35:32 Player 1281130855 attacked at (4, 0) with the unit 49.

2012-05-02 22:35:32 Player 1281130855 ended turn.

2012-05-02 22:35:33 Player 1210714868 moved the unit 47 to (3, 0).

2012-05-02 22:35:33 Player 1210714868 ended turn.

2012-05-02 22:35:33 Player 1210714868 won the game.

2012-05-02 22:35:33 Game Over.

Apêndice C – Material da Avaliação

C.1 Termo de Consentimento de Participação

Título do trabalho: Um *framework* para pesquisa de Inteligência Artificial em jogos de estratégia por turnos

Data: Junho/2012

Instituição: DCC / UFMG

Pesquisador Responsável: Israel Heringer Lisboa de Castro (heringer@dcc.ufmg.br)

O objetivo deste teste é avaliar o *framework* para pesquisa de inteligência artificial em jogos de estratégia por turnos, criado para o projeto de dissertação de mestrado de Israel Heringer Lisboa de Castro.

Vamos pedir aos participantes que utilizem o *framework* para implementar em um agente inteligente (jogador controlado por Inteligência Artificial) algoritmos com o objetivo de executar três tarefas no jogo. Antes da execução das tarefas, deverá ser respondido um questionário com algumas perguntas iniciais para avaliar os conhecimentos prévios do participante. Ao final da execução das tarefas, deverá ser respondido um segundo questionário com o objetivo de coletar a opinião do participante acerca do *framework*, do jogo e das tarefas executadas.

O anonimato dos participantes da avaliação é garantido, sendo que as únicas pessoas que saberão dos seus nomes são os pesquisadores responsáveis por este teste e seus nomes aparecerão apenas neste termo de compromisso, caso os voluntários aceitem participar.

Se você decidir não participar na pesquisa:

Você é livre para decidir, a qualquer momento, se quer participar ou não desta pesquisa. Sua decisão não afetará sua vida estudantil ou profissional e nem qualquer relacionamento com os avaliadores, professores ou a instituição por trás desta.

Compensação

A participação nesta pesquisa é voluntária, e não será oferecida nenhuma remuneração aos seus participantes.

Consentimento Livre e Esclarecido (Acordo Voluntário)

O documento mencionado acima descrevendo os benefícios, riscos e procedimentos da pesquisa foram lidos e explicados. Eu tive a oportunidade de fazer perguntas sobre a pesquisa, que foram respondidas satisfatoriamente. Eu estou de acordo em participar como voluntário.

C.2 Questionário Pré-teste

1. Qual a sua faixa etária?
    a. 18 a 20 anos
    b. 21 a 25 anos
    c. 26 a 30 anos
    d. Acima de 30 anos
2. Qual o seu nível de escolaridade?
    a. Ensino médio completo
    b. Ensino médio com curso técnico/profissionalizante
    c. Ensino superior incompleto (graduação)
    d. Ensino superior completo (graduação)
    e. Pós graduação incompleto
    f. Pós graduação completo

3. Você já jogou jogos de estratégia por turnos (TBS)?

   Alguns exemplos de jogos TBS: "Civilization", "Heroes of Might and Magic", "Panzer General", "Galactic Civilizations", "Age of Wonders", "Colonization" e "Call to Power"

   a. Não conheço
   b. Conheço, mas nunca joguei
   c. Já joguei um pouco
   d. Já joguei bastante

4. Qual o seu contato com a área de desenvolvimento de jogos?

   Conta experiência em qualquer área do desenvolvimento de jogos (programação, arte, level design etc). O contato pode ser de várias formas, como pesquisa (meio acadêmico), desenvolvimento independente, experiência profissional etc.

   a. Nenhum contato
   b. Pouco contato (já pesquisou/trabalhou por um curto período de tempo)
   c. Muito contato (já pesquisou/trabalhou por um período médio a grande de tempo)

5. Qual o seu contato com a área de inteligência artificial?

   Seu contato com a área de Inteligência Artificial, podendo ou não estar relacionada a jogos.

   a. Nenhum contato
   b. Pouco contato
   c. Muito contato

6. Já teve algum contato com modificações em jogos existentes?

   Seu contato com modificações, mesmo que sem finalidade de distribuição de "*Mods*" completos

   a. Nenhum contato
   b. Pouco contato (já tentou modificar algo ou fez modificações muito pequenas)
   c. Muito contato (já fez modificações em alguma funcionalidade completa ou modificações de médio/grande porte)

7. Qual a sua experiência com a linguagem de programação C#?
    a. Nenhuma experiência
    b. Pouca experiência (menos de 6 meses)
    c. Média experiência (de 6 meses a 1 ano)
    d. Muita experiência (mais de 1 ano)

C.3 Questionário Pós-teste

Sobre o jogo

As perguntas a seguir devem ser respondidas tendo em vista o jogo Território, as regras e demais elementos. Sempre que possível deve-se tentar isolar as respostas das opiniões acerca do framework como um todo.

1. O jogo Território pode ser considerado satisfatório como um representante de um jogo TBS?

    De acordo com sua experiência anterior em jogos de estratégia por turnos, você considera que o jogo Território possui regras e elementos que sejam amostras significativas do que é visto em outros jogos do gênero, mesmo que em escala reduzida?

    a. Não sei avaliar
    b. Sim
    c. Não

2. Comente acerca da sua resposta anterior. [Questão aberta]
3. Como você definiria a curva de aprendizado sobre as regras do jogo Território? [Escala]

    Foi fácil ou difícil entender o jogo para executar as tarefas solicitadas?

    a. 1 (Muito fácil)
    b. 2
    c. 3
    d. 4

e. 5 (Muito difícil)

Sobre o framework

As perguntas a seguir devem ser respondidas tendo em vista o framework de maneira geral. Sempre que possível deve-se tentar isolar as respostas das opiniões acerca de questões específicas do jogo, como as regras ou o balanceamento de unidades.

4. Quanto tempo você dedicou para a execução das tarefas?
    a. Menos de 2 horas
    b. De 2 a 5 horas
    c. De 5 a 10 horas
    d. De 10 a 20 horas
    e. Mais de 20 horas
5. Das tarefas passadas, quantas conseguiu você conseguiu executar?
    a. Nenhuma
    b. 1
    c. 2
    d. 3
6. Qual foi a dificuldade que você teve para executar a primeira tarefa (criar uma nova unidade)? [Escala]
    a. 1 (Muito fácil)
    b. 2
    c. 3
    d. 4
    e. 5 (Muito difícil)
7. Qual foi a dificuldade que você teve para executar a segunda tarefa (localizar uma construção inimiga)? [Escala]
    a. 1 (Muito fácil)
    b. 2

c. 3

d. 4

e. 5 (Muito difícil)

8. Qual foi a dificuldade que você teve para executar a terceira tarefa (destruir uma unidade inimiga)? [Escala]

    a. 1 (Muito fácil)

    b. 2

    c. 3

    d. 4

    e. 5 (Muito difícil)

9. Como você avalia a facilidade de implementação da IA no framework? [Escala]

    a. 1 (Muito fácil)

    b. 2

    c. 3

    d. 4

    e. 5 (Muito difícil)

10. Quanto você se baseou na IA de exemplo para as implementações realizadas?

    a. 1 (Pouco)

    b. 2

    c. 3

    d. 4

    e. 5 (Muito)

11. Como você acha que seria utilizar o framework para tarefas mais complexas no jogo Território do que as solicitadas na avaliação?

    a. 1 (Muito fácil)

    b. 2

c. 3

d. 4

e. 5 (Muito difícil)

12. Quais seriam suas maiores facilidade e dificuldades para usar o framework na implementação de tarefas mais complexas que as solicitadas na avaliação? [Questão aberta]

13. Você teve que escrever muito código além do estritamente necessário para a implementação dos algoritmos de IA? [Escala]

    Quanto de código relacionado ao funcionamento do framework/jogo foi necessário escrever

    a. 1 (Pouco código)

    b. 2

    c. 3

    d. 4

    e. 5 (Muito código)

14. Já fez ou tentou fazer algo parecido com o que foi pedido utilizando outra ferramenta ou jogo?

    a. Sim

    b. Não

Sobre sua experiência prévia com outra ferramenta ou jogo

15. Qual foi a ferramenta? Compare a experiência anterior que você teve em outra ferramenta ou jogo com a experiência na execução das tarefas no framework do jogo Território. [Questão aberta, exibida apenas quando for escolhida a opção “Sim” na pergunta 14]

Considerações finais sobre o framework

16. Na sua opinião, o uso do framework pode ajudar na implementação e pesquisa de algoritmos de IA em jogos?
    a. Não sei avaliar
    b. Sim
    c. Não
17. Quanto você considera válida a utilização do framework para pesquisa de IA em jogos TBS? [Escala]
    a. 1 (Pouco válida)
    b. 2
    c. 3
    d. 4
    e. 5 (Muito válida)
18. Coloque seus comentários finais acerca do jogo, do framework ou do processo de avaliação. [Questão aberta]

    Se desejar, fique à vontade para apresentar pontos não abordados pelos dois questionários.

Apêndice D – Código da IA de Exemplo

```
1   using System.Collections.Generic;
2   using Common.AIInterface;
3   using Common.Commands;
4   using Common.General;
5   using Common.Resources;
6   using Common.Resources.Buildings;
7   using Common.Resources.Terrains;
8   using Common.Resources.Units;
9   using Common.Util;
10
11  namespace AIPlayerExample
12  {
13      /// <summary>
14      /// This is an example of a simple AI Player
15      /// It implements the AIBasePlayer interface
16      /// </summary>
17      public class AIPlayer : AIBasePlayer
18      {
19          #region Properties
20
21          /// <summary>
22          /// The current turn validation code
23          /// </summary>
24          private long CurrentTurnValidationCode
25          {
26              get;
27              set;
28          }
29
30          /// <summary>
31          /// The last board received from the game
32          /// </summary>
33          private Board CurrentBoard
34          {
35              get;
36              set;
37          }
38
39          #endregion
40
41          #region Interface AIBasePlayer
42
43          /// <summary>
44          /// Initializes the AIPlayerExample player
45          /// </summary>
46          /// <param name="playerID">The ID of the player in the game - cannot be
    changed</param>
```

```
47          /// <param name="callBack">The callback function that must be called to
    send the commands</param>
48          public AIPlayer(int playerID, AIBasePlayer.PlayerCommandCallBack
    callBack)
49              : base(playerID, callBack) { }
50
51          /// <summary>
52          /// This method is called by the game to invoke the commands of the
    players' turn.
53          /// The player must send all the commands to the callback and then
    finish its turn.
54          /// It is a simple example of AI, capable only of executing the basics
    of the game, no strategy:
55          /// </summary>
56          /// <param name="board">The game board as visible to the player</param>
57          /// <param name="turnValidationCode">The validation code to be sent back
    to the game in with the commands</param>
58          public override void PlayTurn(Board board, long turnValidationCode)
59          {
60              //stores the current turn validation code
61              CurrentTurnValidationCode = turnValidationCode;
62
63              //stores the current board
64              CurrentBoard = board;
65
66              //get the list of game elements from the player
67              List<GameElement> myGameElements =
    CurrentBoard.GetGameElements(PlayerID);
68
69              //iterates through my game elements to execute the actions
70              foreach (GameElement gameElement in myGameElements)
71              {
72                  //if it is a unit
73                  if (gameElement is Unit)
74                  {
75                      //create building with the unit
76                      PerformCreateBuilding(gameElement.ID);
77
78                      //equip the unit
79                      PerformEquipUnit(gameElement.ID);
80
81                      //perform influence with the unit
82                      PerformInfluenceAt(gameElement.ID);
83
84                      //attack with the unit
85                      PerformAttackAt(gameElement.ID);
86
87                      //move the unit
88                      PerformMoveTo(gameElement.ID);
89                  }
90                  //if it is a building
91                  else if (gameElement is Building)
92                  {
93                      //gets the building as a Building
94                      Building building = gameElement as Building;
95
96                      //sets the next building
97                      PerformSetProduction(building);
98                  }
99              }
100
```

```
101             //ends the turn
102             SendCommand(PlayerCommand.EndOfTurn(this));
103         }
104 
105         #endregion
106 
107         #region Methods
108 
109         #region Common
110 
111         /// <summary>
112         /// Sends the command to the game
113         /// </summary>
114         /// <param name="command">The command to be sent</param>
115         /// <returns>The response received from the game</returns>
116         private PlayerCommandResponse SendCommand(PlayerCommand command)
117         {
118             //sends the command and gets the response
119             PlayerCommandResponse commandResponse = CallBack(command,
    CurrentTurnValidationCode);
120 
121             //updates the current board
122             if (commandResponse.ResultBoard != null)
123                 CurrentBoard = commandResponse.ResultBoard;
124 
125             //returns the command response
126             return commandResponse;
127         }
128 
129         #endregion
130 
131         #region Commands Logic
132 
133         /// <summary>
134         /// Creates a building with the unit if it is a priest and is not in a
    building or adjacent to one
135         /// </summary>
136         /// <param name="unitID">The ID of the unit</param>
137         private void PerformCreateBuilding(ulong unitID)
138         {
139             //gets the unit
140             Unit unit = CurrentBoard.GetGameElement(unitID) as Unit;
141             if (unit == null)
142                 return;
143 
144             //create a building only with a priest
145             if (unit.Type == UnitType.Priest)
146             {
147                 //gets its position and neighbours
148                 List<Position> positions = unit.Position.GetNeighbours();
149                 positions.Add(unit.Position);
150 
151                 //iterates through the positions
152                 foreach (Position position in positions)
153                 {
154                     //if there is a building, the unit will not create another
    one
155                     if (CurrentBoard.GetTile(position).Building != null)
156                         return;
157                 }
158 
```

```
159                 //there is no building - create a new one (a city, just for
    example)
160                 SendCommand(PlayerCommand.CreateBuilding(this, unit,
    BuildingType.City));
161             }
162         }
163 
164         /// <summary>
165         /// Equips the unit with all the items in the inventory if it is in a
    building’s tile
166         /// </summary>
167         /// <param name="unitID">The ID of the unit</param>
168         private void PerformEquipUnit(ulong unitID)
169         {
170             //gets the unit
171             Unit unit = CurrentBoard.GetGameElement(unitID) as Unit;
172             if (unit == null)
173                 return;
174 
175             //gets the unit's tile
176             Tile tile = CurrentBoard.GetTile(unit.Position);
177 
178             //if there is a building and it has items, equip one of them on the
    unit
179             if (tile.Building != null && tile.Building.UnitItemsInventory.Count
    > 0)
180             {
181                 //send the equipUnit command
182                 SendCommand(PlayerCommand.EquipUnit(this, unit,
    tile.Building.UnitItemsInventory[0]));
183             }
184         }
185 
186         /// <summary>
187         /// Performs the influence over territories with the unit
188         /// </summary>
189         /// <param name="unitID">The ID of the unit</param>
190         private void PerformInfluenceAt(ulong unitID)
191         {
192             //gets the unit
193             Unit unit = CurrentBoard.GetGameElement(unitID) as Unit;
194             if (unit == null)
195                 return;
196 
197             //gets the unit's neighbours within range
198             List<Position> neighbours = unit.Position.GetNeighbours(unit.Range);
199 
200             //the unit may influence a territory - the chances are proportional
    to its influence factor
201             if (RandomUtil.NextDouble <= unit.InfluenceFactor)
202             {
203                 //searches for a neighbour where it does not have 100% domain
204                 foreach (Position neighbour in neighbours)
205                 {
206                     //verifies the tile's domain
207                     double currentDomain;
208                     CurrentBoard.GetTile(neighbour).Domain.TryGetValue(PlayerID,
    out currentDomain);
209 
210                     //if the domain is not full, influence it
211                     if (currentDomain < Tile.NORMALIZED_DOMAIN_SUM)
```

```
212                     {
213                         //sends the command to influence at
214                         SendCommand(PlayerCommand.InfluenceAt(this, unit,
    neighbour));
215
216                         //updates unit's status
217                         unit = CurrentBoard.GetGameElement(unitID) as Unit;
218                     }
219
220                     //if the unit has no movements left, stop influencing
221                     if (unit.RemainingMovements == 0)
222                         break;
223                 }
224             }
225         }
226
227         /// <summary>
228         /// Perform attacks with the unit
229         /// </summary>
230         /// <param name="unitID">The ID of the unit</param>
231         private void PerformAttackAt(ulong unitID)
232         {
233             //gets the unit
234             Unit unit = CurrentBoard.GetGameElement(unitID) as Unit;
235             if (unit == null)
236                 return;
237
238             //if the unit does not have remaining movements, return
239             if (unit.RemainingMovements == 0)
240                 return;
241
242             //gets the unit's neighbours within range
243             List<Position> neighbours = unit.Position.GetNeighbours(unit.Range);
244
245             //the response received from the game
246             PlayerCommandResponse commandResponse = null;
247
248             //may attack more than once in the turn
249             do
250             {
251                 //finds an enemy in a neighbour
252                 Position attackDestination = neighbours.Find(p =>
    CurrentBoard.GetTile(p).Units.Exists(u => u.Owner != PlayerID));
253
254                 //if there is an attack available to a neighbour, attack it
255                 if (attackDestination != null)
256                 {
257                     //send the attack command
258                     commandResponse = SendCommand(PlayerCommand.AttackAt(this,
    unit, attackDestination));
259
260                     //updates unit's status
261                     unit = CurrentBoard.GetGameElement(unitID) as Unit;
262                 }
263                 //no attack is available - stop attacking
264                 else
265                 {
266                     break;
267                 }
268             } while (commandResponse != null && commandResponse.Result !=
    PlayerCommandResult.NOK && unit != null && unit.RemainingMovements > 0);
```

```
269         }
270
271         /// <summary>
272         /// Performs the movements with the unit
273         /// </summary>
274         /// <param name="unitID">The ID of the unit</param>
275         private void PerformMoveTo(ulong unitID)
276         {
277             //gets the unit
278             Unit unit = CurrentBoard.GetGameElement(unitID) as Unit;
279             if (unit == null)
280                 return;
281
282             //if the unit does not have remaining movements, return
283             if (unit.RemainingMovements == 0)
284                 return;
285
286             //the response received from the game
287             PlayerCommandResponse commandResponse = null;
288
289             //may move more than once in the turn
290             do
291             {
292                 //gets the unit's neighbours
293                 List<Position> neighbours = unit.Position.GetNeighbours();
294
295                 //removes neighbours with enemies
296                 neighbours.RemoveAll(p => CurrentBoard.GetTile(p).Units.Exists(u
    => u.Owner != PlayerID));
297
298                 //removes neighbours where it cannot move to
299                 neighbours.RemoveAll(p =>
    !Terrains.Get(CurrentBoard.GetTile(p).Terrain).CanReceiveGroundUnits);
300
301                 //if no neighbour is available, don't move
302                 if (neighbours.Count == 0)
303                     return;
304
305                 //if there is an enemy building in a neighbour, go to it
306                 Position destination = neighbours.Find(p =>
    CurrentBoard.GetTile(p).Building != null &&
    CurrentBoard.GetTile(p).Building.Owner != PlayerID);
307                 if (destination == null)
308                     destination = neighbours[RandomUtil.NextInt %
    neighbours.Count];
309
310                 //moves randomly to a neighbour
311                 commandResponse = SendCommand(PlayerCommand.MoveTo(this, unit,
    destination));
312
313                 //updates unit's status
314                 unit = CurrentBoard.GetGameElement(unitID) as Unit;
315
316             } while (commandResponse != null && commandResponse.Result !=
    PlayerCommandResult.NOK && unit != null && unit.RemainingMovements > 0);
317         }
318
319         /// <summary>
320         /// Sets the production of a building
321         /// </summary>
322         /// <param name="building">The building to set the production on</param>
```

```
323         private void PerformSetProduction(Building building)
324         {
325             //gets the position
326             Position position = building.Position;
327 
328             //gets the Tile
329             Tile tile = CurrentBoard.GetTile(position);
330 
331             //if I have a building not producing, set its production
332             if (tile.Building != null && tile.Building.Owner == PlayerID &&
    tile.Building.CurrentProduction == null)
333             {
334                 //sets the production on the building
335                 SendCommand(PlayerCommand.SetProduction(this, tile.Building, new
    ProduceableElement(UnitType.Lieutenant)));
336             }
337         }
338 
339         #endregion
340 
341         #endregion
342     }
343 }
344 
```