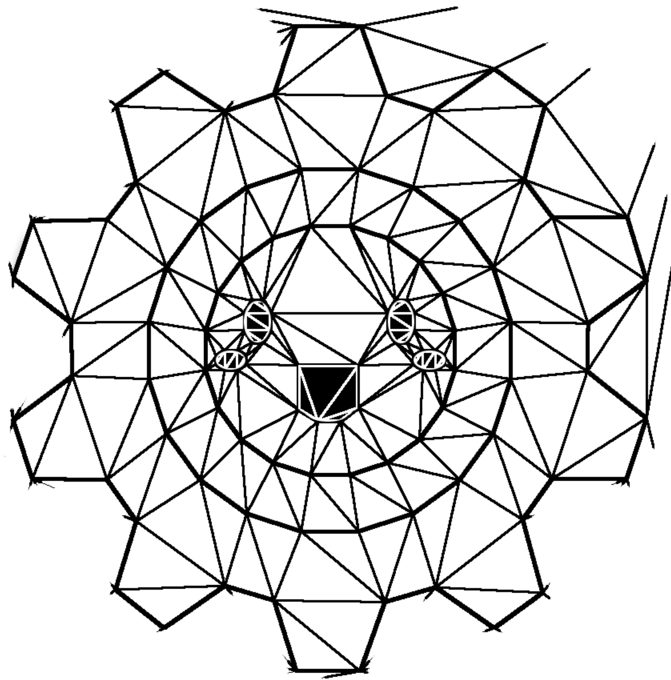
Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DeltaPath: Um módulo genérico de planejamento de trajetória



Luiggi Monteiro Reffatti

Brasília
2013

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DeltaPath: Um módulo genérico de planejamento de trajetória

Luiggi Monteiro Reffatti

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Guilherme Novaes Ramos

Coorientadora
Prof.ª Dr.ª Carla Denise Castanho

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof.ª Dr.ª Maristela Terto de Holanda

Banca examinadora composta por:

Prof. Dr. Guilherme Novaes Ramos (Orientador) — CIC/UnB
Prof.ª Dr.ª Carla Maria Chagas e Cavalcante Koike — CIC/UnB
Prof. Dr. Flávio de Barros Vidal — CIC/UnB
Prof.ª Dr.ª Carla Denise Castanho (Coorientadora) — CIC/UnB

CIP — Catalogação Internacional na Publicação

Reffatti, Luiggi Monteiro.

DeltaPath: Um módulo genérico de planejamento de trajetória / Luiggi Monteiro Reffatti. Brasília : UnB, 2013.
63 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. planejamento de trajetória, 2. busca por caminho,
3. particionamento espacial, 4. inteligência artificial, 5. jogos

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DeltaPath: Um módulo genérico de planejamento de trajetória

Luiggi Monteiro Reffatti

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Guilherme Novaes Ramos (Orientador)
CIC/UnB

Prof.ª Dr.ª Carla Maria Chagas e Cavalcante Koike
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
CIC/UnB

Prof.ª Dr.ª Carla Denise Castanho (Coorientadora)
CIC/UnB

Prof.ª Dr.ª Maristela Terto de Holanda
Coordenador do Bacharelado em Ciência da Computação

Brasília, 18 de julho de 2013

Dedicatória

À pessoa que me fez quem eu sou, Maria do Pranto "Luar" Monteiro, que apesar de não estar aqui estaria muito orgulhosa e seria com certeza a fã número um do Smile Wars.
À pessoa que me reensinou a ser feliz, FerNanda "Pan" Plentz, muito obrigado por tudo.
À pessoa que me manteve são, Eduardo "Kyrw" Freire, me ajudando a seguir em frente enquanto eu perdia as duas outras pessoas aqui mencionadas.

Agradecimentos

Obrigado aos professores que me orientaram e acreditaram em mim:
Guilherme Novaes Ramos
Carla Denise Castanho

Obrigado aos professores que aceitaram participar da banca de avaliação deste trabalho pelos valiosos comentários e sugestões:
Carla Maria Chagas e Cavalcante Koike
Flávio de Barros Vidal

Obrigado à equipe Delta Creatures por todos os anos de sábados de trabalho duro:
Eduardo "Kyrw" Freire
Guilherme "Shrek" Costa
Raul "braindf" Santana
Herman "Metaur" Ferreira
Mayara "Verde" Rosa
Guilherme "PC" Nunes

Obrigado aos membros das equipes Gear2D e FiraSoft, pela honra de trabalhar juntos:
Leonardo "Léo" Freitas
Igor "Bicão" Sousa

Muito obrigado à minha "nova" "velha" família, por todo o suporte:
Rebeca "Rosa" Monteiro
Michele "Verde" Monteiro
Marcos "Brigão" Monteiro
Gorete "Tia" Gomes

Obrigado a todas as fábricas de sonhos, pela inspiração. Em especial:
Blizzard Entertainment
Squaresoft
Maxis

Resumo

Inteligência artificial (IA) se tornou um aspecto importante e presente nos jogos eletrônicos. Uma área importante dentro de IA aplicada a jogos é o problema do planejamento de trajetória, ou seja, como fazer entidades se locomoverem de maneira autônoma e de forma coerente em um determinado ambiente. Várias soluções existem para este problema, porém nenhuma delas resolve o problema de forma eficiente para qualquer caso, e há espaço para contribuições.

Este tópico envolve uma combinação de diferentes técnicas e algoritmos com propósitos distintos, necessária devido à natureza dos jogos eletrônicos, que são aplicações complexas e multidisciplinares. Particionamento espacial, busca em grafos, suavização de trajetória, vetorização de ambientes e comportamento de movimentação são alguns dos assuntos envolvidos na pesquisa de movimentação de entidades autônomas.

A proposta dessa monografia é o desenvolvimento de um módulo para planejamento de trajetória, abrangendo representação espacial, busca de caminho e suavização de rota, que seja eficiente, simples e genérico. Esse módulo, denominado *DeltaPath*, é então aplicado em um jogo de estratégia e em um componente de um motor de jogos, para avaliar se esses objetivos foram alcançados, identificar potenciais problemas e propor soluções.

Após realizadas as aplicações e os testes do *DeltaPath*, são apresentados os resultados. Esses mostram melhoria no desempenho e jogabilidade do jogo ao qual foi integrado, sem problemas de compatibilidade com a arquitetura em componentes do motor de jogos onde foi utilizado e facilidade de uso em ambas aplicações.

Palavras-chave: planejamento de trajetória, busca por caminho, particionamento espacial, inteligência artificial, jogos

Abstract

Artificial intelligence (AI) has become an important aspect in commercial games. An important field within AI for games is the problem of path planning, i.e., how to make entities move autonomously in a well defined environment. Several solutions exist for this problem, but none of them solve the problem for all cases, and there is room for further contributions.

This topic involves a combination of various techniques and algorithms with different purposes, required due the nature of digital games, which are complex and multidisciplinary applications. Space partitioning, graph search, path smoothing, environment vectorization and steering behaviors are some of the issues involved in the research of autonomous moving entities.

The objective of this work is the development of an efficient, simple and generic path planning module that deals with spatial representation, pathfinding and path smoothing. This module, called *DeltaPath*, is applied in a strategy game and used to create a component for a game engine, to evaluate whether the goals have been achieved and to identify potential problems and propose solutions.

After integration and tests of the module, the results are presented. These show improvements in performance and gameplay in the game where it was integrated, no compatibility issues with the game engine's component-based architecture where it was used and ease of use in both applications.

Keywords: path-planning, pathfinding, space partitioning, artificial inteligence, games

Sumário

1 Introdução **1**

2 Trabalhos Correlatos **4**
2.1 IA em Jogos 4
2.2 Busca por Trajetória 6

3 Planejamento de Trajetória **8**
3.1 Descrição do Ambiente 8
3.1.1 Representação em Grade 9
3.1.2 Representação em Polígonos Convexos Variados 11
3.1.3 Triangulação Restrita 13
3.1.4 Triangulação de Delaunay Restrita e Dinâmica 15
3.2 Busca por Trajetória 21
3.2.1 Algoritmo *A** 21
3.2.2 *A** em uma Triangulação 23
3.2.3 Custo Acumulado e Heurística 25
3.2.4 *Triangulation A** 27
3.3 Suavização do Caminho 28
3.3.1 O Algoritmo do Funil 28

4 Módulo *DeltaPath* **33**
4.1 Implementação 33
4.2 Testes e Resultados 35

5 Aplicações **40**
5.1 *Smile Wars* 40
5.2 Gear2D 43

6 Conclusões e Trabalhos Futuros **47**
6.1 Conclusões 47
6.2 Trabalhos Futuros 48

Referências **50**

Lista de Figuras

3.1 Representação de três unidades e uma restrição. 9
3.2 Ambiente com obstáculos. 10
3.3 Representações de um ambiente em *grid*. 10
3.4 Movimentação possível para uma unidade u. 11
3.5 Grade com tamanho reduzido para melhorar a representação. 11
3.6 Ambiente com obstáculos usando polígonos convexos. 13
3.7 Ambiente com obstáculos usando uma triangulação restrita. 14
3.8 Comparativo: *grid* com precisão e triangulação restrita. 14
3.9 Um conjunto de pontos. 15
3.10 Triangulações possíveis para um conjunto de pontos. 16
3.11 Um conjunto de arestas representando um ambiente. 16
3.12 Triangulação restrita e de Dalaunay de um ambiente. 17
3.13 Inserção de um vértice sobre uma aresta. 18
3.14 Inserção de um vértice em uma face. 18
3.15 Virando arestas para restaurar a propriedade de Delaunay. 19
3.16 Inserindo um segmento restrito na triangulação. 19
3.17 Inserindo vértices entre o novo segmento e as arestas restritas. 20
3.18 Remoção das arestas não-restritas que cruzam o novo segmento. 20
3.19 Triangulação final. 20
3.20 Grafo representando um ambiente discretizado. 22
3.21 Um caminho inválido entre dois triângulos. 24
3.22 Caminho pelos centros de triângulos e sua estimativa do menor caminho. . 24
3.23 O menor caminho até um ponto depende da posição destino. 24
3.24 O Caminho calculado com heurística simples nem sempre é o mais curto. . 25
3.25 Uma sequência de triângulos com início, canal e fim. 29
3.26 A rota, o ápice e o funil durante a execução do algoritmo. 30
3.27 Subfunis correspondentes aos vértices do funil. 30
3.28 Adicionando um vértice no lado direito do funil. 31
3.29 O funil formado após a adição de um novo vértice à direita. 31
3.30 Adicionando um vértice no lado esquerdo do funil. 32
3.31 O funil formado após a adição de um novo vértice na esquerda. 32

4.1 Diagrama que mostra as interfaces internas e externa do *DeltaPath*. 34
4.2 Divisão do processamento antes da integração com o *DeltaPath*. 37
4.3 Divisão do processamento após a integração com o *DeltaPath*. 38
4.4 Subárvore de chamadas a funções após a integração com o *DeltaPath*. . . . 39

5.1 Captura de tela do *Smile Wars*. 41
5.2 Um mapa e sua triangulação no jogo *Smile Wars*. 42
5.3 Logo da *Gear2D* triangulado através do *DeltaPath* 44
5.4 Katch-Up - demonstração da capacidade do componente *Pathfinder2D*. . . 45

Capítulo 1

Introdução

Por que desenvolver jogos eletrônicos? A indústria de jogos digitais tem crescido substancialmente em relação a formas mais tradicionais de entretenimento [9]. Em 2011, constatou-se que o gasto com jogos, consoles e acessórios ultrapassou 25 bilhões de dólares no ano [2], um número expressivo comparado à indústria de cinema (aproximadamente 14 bilhões de dólares anuais [25]) e de música (aproximadamente 7 bilhões de dólares anuais [27]). O crescente interesse econômico neste mercado, cuja base é tecnologia e inovação, tem impulsionado a pesquisa científica na área de jogos digitais.

Além de altamente rentáveis, jogos são excelentes objetos de estudo por serem aplicações complexas envolvendo diversas áreas do conhecimento dentro e fora da computação [24], como renderização, interface com o usuário, áudio, inteligência artificial, psicologia, pedagogia, dentre outras. Por exemplo, grandes jogos *online* precisam de comunicação via rede e, geralmente, também de bancos de dados. Não são incomuns jogos que necessitam até de processamento de imagens para sensores de movimento ou utilizam *hardware* específico [24].

Nos últimos anos, em grande parte por causa dos grandes investimentos em pesquisa, a qualidade dos jogos eletrônicos avançou significativamente. O impacto desses avanços fica evidente nos detalhes e no realismo dos ambientes criados nos jogos atuais, desde salas e corredores detalhados a paisagens externas vastas. Esses jogos populam o ambiente com personagens controlados tanto por jogadores quanto pelo computador, propiciando um laboratório rico para a pesquisa em agentes autônomos [24].

Dentre os aspectos que mais tem evoluído nos últimos anos está a inteligência artificial (IA), que tem se destacado como um diferencial nos jogos eletrônicos. Quase todos os jogos produzidos hoje em dia tem algum elemento de IA, seja um modelo de comportamento de um personagem, de adaptação ao comportamento do usuário, de processamento dos sinais de entrada, etc [26].

A importância da IA fica evidente em jogos comerciais e de jogabilidade complexa, especialmente em modo de jogo individual (*single player*), onde é muito difícil criar uma experiência divertida sem que o oponente controlado pelo software aja de maneira desafiadora e ao mesmo tempo natural [26]. Em determinados estilos, como interpretação de papéis (*Role-Playing Game* - RPG) e jogos de estratégia em tempo real (*Real Time Strategy* - RTS), que são jogos que incluem movimentação indireta (onde o jogador não controla a trajetória exata das personagens), é desejável que a IA tenha um comportamento "inteligente", escolhendo uma ação coerente com a situação em que se encontra,

como por exemplo, movimentar-se através do menor caminho até seu objetivo. Tarefas como essa são de implementação complexa [26] e demandam muitos recursos computacionais.

Nesse sentido, os jogos RTS se destacam por apresentarem um contexto favorável à pesquisa e investigação [4] [24], pois esse estilo impõe diversos desafios como alocação de recursos, problemas espaciais e planejamento de ações em tempo real. Também é possível encontrar uma grande quantidade de pessoas conhecedoras desse estilo de jogo, o que torna o processo de testes mais fácil e expressivo. Além disso, o atual estado da inteligência artificial nos jogos ainda não atingiu um patamar ideal, portanto, há espaço para contribuições, principalmente quanto ao planejamento e aprendizado para a máquina.

Essa situação fica evidente quando, ao desenvolver um jogo de estratégia, nos deparamos com um problema cuja solução utiliza técnicas de IA. Não é possível encontrar soluções abertas, eficientes e genéricas para resolver vários desses problemas. Diante do problema de planejamento de trajetória, e após uma busca extensiva por soluções diretamente aplicáveis, sem sucesso, fez-se necessário desenvolver uma solução própria.

Este trabalho consiste na pesquisa, desenvolvimento e disponibilização de um módulo, denominado *DeltaPath*, para resolver o problema do planejamento de trajetória. Este módulo almeja ser genérico, para que possa ser usado em várias aplicações diferentes, eficiente, de modo a não impactar significativamente nos programas onde será utilizado e simples, para facilitar e propagar seu uso.

O *DeltaPath* é desenvolvido através da combinação de três técnicas de IA com propósitos diferentes: triangulação restrita, para particionar e representar o ambiente, o algoritmo *A** [16], para fazer a busca pela trajetória, e o algoritmo do funil [29] para suavizar o caminho.

A triangulação implementada neste trabalho é a Triangulação de Delaunay Restrita e Dinâmica (*Dynamic Constrained Delaunay Triangulation* - DCDT), desenvolvida por M. Kallmann [20], e foi baseada no trabalho de D. Demyen e M. Buro [10]. Há uma implementação em código aberto disponível como parte do motor de jogos *Open Real-Time Strategy - ORTS* [6], entretanto, esta é fortemente acoplada a este motor. Além disso, seu desenvolvimento foi interrompido em 2007 e a última atualização do projeto foi feita em 2010, praticamente tornando o código do *ORTS* (e de todo o módulo de planejamento de trajetória) inutilizável devido a atualizações de dependências e interfaces. Dadas estas dificuldades, e a intenção de se ter um módulo simples, genérico e com escopo reduzido (não englobando a parte de abstrações), optou-se por desenvolver uma nova implementação.

Após seu desenvolvimento, o *DeltaPath* foi integrado a um jogo de estratégia e utilizado para a criação de um componente de um motor de jogos, a fim de avaliar se os objetivos do módulo foram alcançados, identificar potenciais problemas e propor soluções. Após esses testes, são apresentados os resultados que mostram melhoria no desempenho e jogabilidade do jogo à qual foi integrado, nenhum problema de compatibilidade com a arquitetura em componentes do motor de jogos onde foi utilizado e facilidade de uso em ambas aplicações.

O restante desta monografia está organizado da seguinte forma: O Capítulo 2 faz uma breve revisão de trabalhos correlatos na área de IA, como técnicas gerais de pesquisa e específicas de busca de trajetória. O Capítulo 3 discute o problema da representação espacial para planejamento de trajetória e apresenta as técnicas existentes com seus prós e contras, e também descreve a técnica de busca de trajetória em uma triangulação e a

suavização dessa trajetória. O Capítulo 4 descreve o módulo implementado e apresenta os testes feitos e os resultados obtidos. O Capítulo 5 mostra a integração do *DeltaPath* em aplicações reais e os resultados dessas aplicações. O Capítulo 6 apresenta as conclusões e sugestões para trabalhos futuros.

Capítulo 2

Trabalhos Correlatos

O problema de planejamento de trajetória está presente na grande maioria dos jogos RTS. Apesar disso implementações com código aberto desses jogos não são comuns [4], pois a maioria desses jogos, especialmente os mais complexos, são feitos para o mercado e, por uma questão de segurança e competitividade, suas implementações não são divulgadas.

Uma implementação conhecida é a ORTS (*Open Real-Time Strategy*) [6], que é um ambiente para estudar problemas de inteligência artificial em tempo real como cálculo de rota, decisões com informações incompletas, agendamento e planejamento de ações, dentre outros. O objetivo do projeto ORTS é ser um sistema de jogo que permita que jogadores e computadores joguem de maneira justa, ou seja, de acordo as regras do jogo. Para isso, várias medidas de segurança foram tomadas, como por exemplo a comunicação com o servidor, que considera somente o que está visível para o cliente naquele momento (impossibilitando assim tentativas de obter informações sobre todo o ambiente por meios externos ao jogo).

O protocolo de comunicação utilizado no ORTS é público e todo seu código fonte e arte estão disponíveis sem custo. Usuários podem conectar qualquer programa cliente que desejarem, graças a uma arquitetura cliente/servidor que garante que cada jogador tenha acesso apenas às informações que foram obtidas conforme as regras do jogo. Essa abertura leva a várias novas possibilidades, desde torneios *online* de IAs autônomas até o uso de sistemas híbridos onde jogadores humanos podem utilizar uma interface com o usuário que os permita delegar tarefas para um ajudante virtual (IA).

Sua implementação está disponível como parte do motor de jogos *Open Real-Time Strategy - ORTS* [6]. Por ser um módulo feito para resolver o problema específico deste motor, sua implementação não é facilmente generalizável, e devido à falta de atividade no projeto desde 2007 o mesmo está incompatível com algumas bibliotecas utilizadas, que tiveram mudanças em sua interface nesse tempo.

2.1 IA em Jogos

De um modo geral, IA está presente nos jogos eletrônicos em várias situações, tais como, controlar personagens individualmente, fornecer uma orientação estratégica para grupos de personagens ou mudar parâmetros para fazer o jogo adequadamente desafiador. Jogos digitais fornecem um ambiente de pesquisa de baixo custo, confiável e surpreenden-

temente acessível, e alguns desses contam até com interfaces de IA disponíveis, onde é possível programar e utilizar módulos personalizados [24].

O uso de jogos eletrônicos como objeto de estudo evita algumas críticas levantadas contra a pesquisa baseada em simulações. Como na maioria das vezes os pesquisadores não são responsáveis pelo desenvolvimento dos jogos, estes evitam noções pré-concebidas sobre quais características do ambiente os desenvolvedores podem simular com facilidade e quais são complexas. Como exemplo disso, muitos dos jogos usados em pesquisas são produtos que estão no mercado e possuem ambientes virtuais onde milhares de jogadores podem interagir intensamente.

Custos de desenvolvimento e falta de poder de processamento impedem diversos jogos eletrônicos atuais de implementarem uma inteligência artificial eficiente. Atualmente, os jogos gastam muito mais recursos com renderização do que com IA. Apesar disso, os desenvolvedores estão sempre procurando novos meios de diferenciar seus produtos, e algumas companhias vendem jogos com ênfase em IA, como a Electronic Arts/Maxis com *SimCity*[1], e a Blizzard Entertainment com *StarCraft*[2]. Nestes casos, o objetivo é atrair a atenção dos jogadores para o quão realista é uma simulação ou o quão desafiador pode ser um oponente virtual.

Simuladores e jogos RTS apresentam diversos desafios computacionais [5]. Um problema de interesse nessa área é que, além das tarefas comuns a todos os jogos, que geralmente demandam um número enorme de computações por segundo, nesse gênero existem muitos objetos (edifícios, unidades, etc.), o que aumenta a complexidade do ambiente a ser considerado. Além disso, o ambiente geralmente é grande e variado, com áreas bem abertas, como campos, e áreas bem fechadas, como labirintos.

Dentre as muitas aplicações de IA em jogos, destacam-se [5]:

- Raciocínio espacial e temporal: análise estática e dinâmica do ambiente e entendimento das relações temporais entre as ações é de extrema importância em jogos RTS, e ainda hoje muitos jogos não executam essas análises de maneira satisfatória.

- Informação incompleta: na maioria dos jogos os jogadores não conhecem a localização e as ações de seus adversários. É necessário descobrir essas informações em tempo real. Geralmente não é possível se conseguir toda a informação necessária, e os jogadores devem tomar suas decisões ainda assim.

- Planejamento em tempo real: em simulações com grande número de agentes não é possível planejar com base em ações singulares. Nesses casos, é necessário encontrar abstrações que permitam que o computador execute buscas diretas em um espaço de dados aceitável e seja capaz de transformar a solução encontrada em ações válidas.

- Gerenciamento de recursos: muitas vezes os jogadores têm uma quantidade limitada de recursos do jogo necessários para a execução de diversas ações. Esses recursos devem ser extraídos do ambiente de acordo com as regras do jogo, através de determinadas ações que custam tempo. Gerenciar de maneira eficiente esses recursos é necessário para o sucesso do jogador.

[1] http://www.simcity.com/
[2] http://www.starcraft2.com/

- Colaboração: muitas vezes, é possível que vários jogadores se organizem em equipes com objetivos em comum. Nesses casos, é necessário coordenar as atividades de várias IAs para que sejam eficientes juntas.
- Aprendizado: um dos maiores problemas nas IAs da maioria dos jogos RTS é sua incapacidade de aprender com as experiências. Jogadores humanos aprendem naturalmente as fraquezas do adversário após algumas partidas, e como podem utilizar isso a seu favor nas próximas.

De todos os aspectos de IA em um jogo RTS, um de destaque especial é a movimentação [26], pois o usuário percebe rapidamente qualquer comportamento falho. É esperado que os agentes desviem de obstáculos, escolham a melhor rota até o destino, considerem partes desconhecidas do ambiente, etc. Por ser um aspecto básico, muitos acreditam que esse problema já foi definitivamente resolvido nos jogos atuais, porém isso nem sempre procede. Apesar de uma grande parte dos recursos em IA dos jogos serem investidos em movimentação, esta tarefa é o alvo da maior parte das insatisfações e problemas relatados pelos jogadores sobre IA [26].

Isso deve-se principalmente ao fato de que os recursos computacionais disponíveis para as computações de IA são limitados, pois existe a demanda de processamento de outros aspectos, como renderização, por exemplo. Outro fator que influencia na redução da capacidade de processamento destinado à IA nos jogos é o número cada vez maior de agentes que devem ser simulados.

Os requisitos de um jogo simplesmente reforçam as restrições sob as quais um sistema de movimentação de agentes deve operar: é necessário encontrar uma trajetória no menor tempo possível, pois diversos caminhos podem ser requisitados ao mesmo tempo, e a representação do ambiente deve comportar tanto grandes regiões quanto áreas detalhadas de maneira eficiente.

2.2 Busca por Trajetória

Além das técnicas de busca geral existentes [30] que podem ser usadas para procurar por uma rota, existem várias buscas especializadas que aproveitam as propriedades inerentes a esse problema. De fato, planejamento de trajetória é extremamente importante em áreas como jogos eletrônicos e robótica, e existem métodos que não só tratam da busca por trajetória (*pathfinding*) como um todo, mas a dividem para resolver separadamente cada uma de suas subtarefas com o objetivo de simplificar o problema e assim obter soluções melhores.

Em muitos casos, são utilizadas técnicas de busca por trajetória em múltiplos níveis. Por exemplo, para longas distâncias, um caminho aproximado entre determinados pontos é calculado primeiro, e então os trechos entre esses pontos são calculados de maneira mais precisa. Esse segundo passo é necessário para lidar com situações como informação incompleta, ambiente dinâmico e outros agentes.

O problema de calcular o menor caminho em um plano é bem definido e uma solução ótima é conhecida [18], cujo tempo de execução e memória são da ordem $O(n \cdot \log n)$, onde n é o número de vértices nos polígonos usados pela técnica para descrever o ambiente. Essa solução implementa um algoritmo eficiente que simula a propagação de frente de onda (*wavefront propagation*) entre obstáculos poligonais e, na prática, computa um mapa

planar que contém todos os melhores caminhos de um ponto fixo para todos os outros possíveis pontos no plano.

Essa técnica porém é de implementação complexa e, devido à grande quantidade de cálculos envolvidos, computacionalmente custosa. Na maioria dos casos encontrados em jogos eletrônicos, uma implementação mais simples é mais apropriada. Geralmente as soluções mais adequadas para esse problema são as que utilizam uma representação discretizada do ambiente, sendo a divisão em células quadradas (grade) a mais comum delas [22].

A técnica da divisão em grade, porém, acaba acarretando na perda de precisão da representação ou demandando muito tempo nas buscas. Para resolver esse problema outras técnicas foram desenvolvidas [31], como grafos de visibilidade. Isto envolve conectar os vértices de todos os obstáculos uns aos outros desde que seja possível traçar uma linha reta entre eles. Esse grafo pode então ser usado como uma representação do ambiente na busca por uma rota, e garante que, para objetos pontuais, é possível encontrar o menor caminho entre dois pontos (caso exista). Uma das desvantagens dessa técnica é que o número de arestas do grafo pode aumentar muito de acordo com a disposição dos obstáculos, o que pode causar uma perda de desempenho na busca. Outra é que essa técnica sozinha é capaz apenas de calcular trajetórias para objetos de um tamanho único.

Em ambientes com muitas dimensões, por exemplo, algoritmos completos e discretos podem não ser apropriados e, por isso, implementações como a *Rapidly-exploring Random Tree (RRT)* [23] foram desenvolvidas. Essas árvores exploram o ambiente rápida e aleatoriamente em uma tentativa de encontrar um caminho entre origem e destino. Apesar de úteis quando o ambiente é complexo ou tão grande que tornam outras abordagens ineficientes, esse tipo de técnica não garante completude, e geralmente o caminho encontrado não é ótimo. Em jogos RTS, essas duas características são importantes.

Para resolver os problemas dos métodos existentes, foi desenvolvida uma técnica chamada de *Triangulation A** [10], que executa sobre uma divisão espacial em triangulação, mais especificamente a *Constrained Delaunay Triangulation* [20]. Essa técnica garante as propriedades do espaço euclidiano, os benefícios da busca determinística e discretizada e uma representação fiel do ambiente. Com isso, é possível calcular o menor caminho de manira eficiente.

Capítulo 3

Planejamento de Trajetória

A grande maioria dos jogos tem dois tipos de personagens: controlados pelo jogador (*Playable Character* - PC) ou não (*Non-Playable Character* - NPC). Todo personagem que pode interagir com outros e/ou com o cenário é considerado um agente, e todo que execute ações independentes do controle direto do jogador é considerado uma unidade. Um NPC que age exclusivamente de acordo com uma IA é um agente e uma unidade, assim como é um PC que é indiretamente controlado por uma IA (por exemplo, para movimentação). Um personagem controlado de maneira direta e exclusiva pelo jogador é apenas um agente. Esses agentes interagem em um espaço limitado e bem definido, e define-se como ambiente a representação desse espaço. É o ambiente que indica a posição de eventuais obstáculos à movimentação, as colisões entre os agentes e se a posição de um agente é válida.

Dado um ambiente e uma unidade que deve deslocar-se por ele de forma autônoma, isto é, deve encontrar seu próprio caminho evitando obstáculos, temos um problema de planejamento de trajetória *pathfinding*. Chamaremos de *pathfinder* todo o algoritmo responsável por encontrar este caminho. O primeiro problema encontrado ao se implementar um *pathfinder* é a representação do ambiente. Para executar um algoritmo de busca sobre um ambiente, é necessário que este seja discretizado de alguma maneira. Esta divisão espacial depende de vários fatores, como por exemplo, a forma dos agentes, a precisão necessária no movimento e o algoritmo de busca que se pretende usar para encontrar uma trajetória.

Neste capítulo serão explicadas as três técnicas utilizadas pelo módulo de planejamento de trajetória. A Seção 3.1 apresenta algumas das técnicas mais comuns de particionamento e representação do ambiente e discute suas vantagens e desvantagens. Na Seção 3.2 é descrito o algoritmo $A*$ para busca de caminho e quais os problemas de se utilizar este algoritmo em um ambiente representado por uma triangulação. São então feitas considerações sobre como lidar com essas questões e uma solução é mostrada. Por fim, a Seção 3.3 descreve uma técnica de suavização de trajetória, aplicada depois de encontrado um caminho na abstração do ambiente.

3.1 Descrição do Ambiente

Agentes podem ter diversos atributos em jogos. Para o *pathfinder* os atributos mais relevantes são os espaciais, como posição no espaço, tamanho do agente (corpo de colisão)

e orientação. A posição indica onde exatamente no ambiente está localizado o agente. O corpo de colisão (*bounding box*) do agente determina o formato deste no espaço. A orientação (ou rotação) do agente define onde está a frente do mesmo em relação ao ambiente, além da direção e do sentido de sua movimentação em um dado momento.

O ambiente define onde um agente pode ou não estar, e por onde ele pode se mover. Em geral, os agentes se movem a uma determinada velocidade e em linha reta, ou seja, podem alterar sua posição em uma determinada distância em um dado tempo (normalmente, o tempo de um ciclo de processamento do jogo). Chamamos uma posição e um caminho de válidos quando os mesmos não se sobrepõe às restrições do ambiente. Para uma movimentação de um agente ser válida, não apenas suas posições iniciais e finais devem ser válidas após um movimento, mas o caminho entre elas tem também que ser válido.

De acordo com a representação usada para o corpo de colisão de uma unidade (elíptica ou quadrada, por exemplo) ela pode estar em uma posição válida ou não dependendo de sua orientação, como mostrado na Figura 3.1.

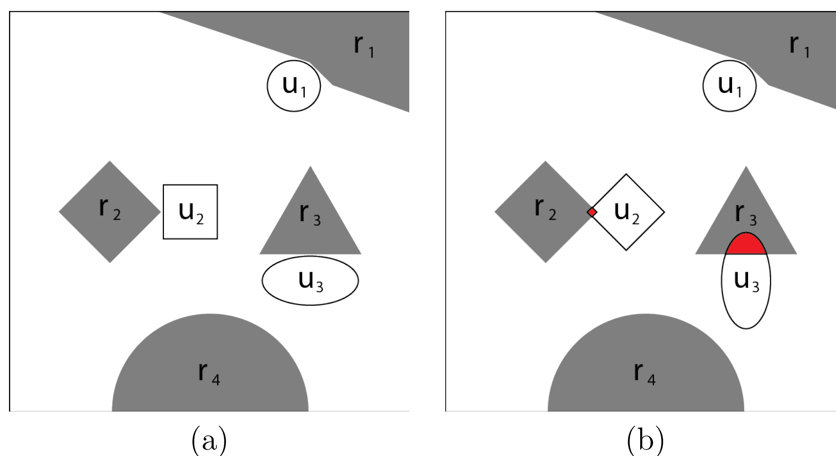


Figura 3.1: (a) Representação de três unidades u e quatro restrições r em posições válidas. (b) Unidades u_2 e u_3 com orientação alterada e em posições inválidas.

Por uma razão de desempenho, simplificaremos o problema assumindo, daqui por diante, que todos os agentes sejam simétricos em relação ao raio, ou seja, têm forma circular com raios de tamanhos variados. Desta forma, os cálculos ficarão mais simples e rápidos, pois não será necessário lidar com a orientação.

Existem várias formas de representar o ambiente, cada uma com suas vantagens e desvantagens, como demonstrado por Tozour [31]. Segue um breve resumo das mais conhecidas e utilizadas. Na Seção 3.1.1 será mostrada a técnica de divisão em grade. A Seção 3.1.2 descreve a abstração do ambiente através de polígonos convexos. Um subconjunto de polígonos convexos é a representação em triangulação, mostrada na Seção 3.1.3. Por fim, a Seção 3.1.4 apresenta a técnica de particionamento utilizada neste trabalho.

3.1.1 Representação em Grade

A representação mais comum encontrada nos *pathfinders* é baseada em grade (*grid*) [10], onde o ambiente é dividido em polígonos iguais, geralmente quadrados, independentes das restrições do ambiente. Deste modo, caso um ou mais obstáculos intersectem um desses

polígonos, este é considerado inválido para movimentação, e válido caso contrário. Esses polígonos são chamados de células e podem conter um ou mais agentes. A Figura 3.2 mostra um possível ambiente com obstáculos. Suas representações como grade de células quadradas e hexagonais estão ilustradas na Figura 3.3 (a) e (b).

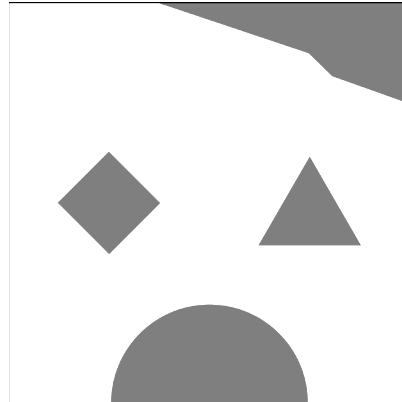


Figura 3.2: Ambiente com obstáculos.

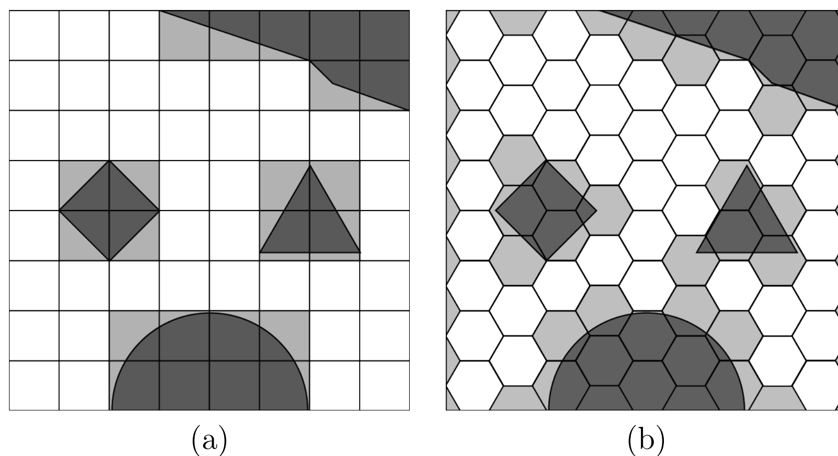


Figura 3.3: Representações do ambiente. (a) Grade quadrada e (b) hexagonal. As células em cinza claro estão ocupadas por obstáculos e, portanto, inválidas para posicionamento e movimentação dos agentes.

Nesse modelo, a tarefa de encontrar o caminho fica reduzida aos polígonos da grade, ou seja, um agente sempre está em uma célula válida e pode se mover apenas para outra célula válida, geralmente adjacente à primeira. Define-se a adjacência entre células como células fisicamente adjacentes no ambiente de jogo, e entre essas células é definida uma conexão por onde pode ser possível ou não movimentar-se. A Figura 3.4 ilustra essa situação no caso de movimentação da unidade u.

Pela simplicidade de implementação e pela existência de diversas técnicas de *path-finding* baseadas em *grid*, a maioria dos jogos comerciais utiliza esta representação de ambiente [10]. Entre as desvantagens dessa representação, ressalta-se a falta de precisão na representação dos objetos, que dependendo do tamanho das células pode delimitar um ambiente diferente do original. A Figura 3.3 (b) ilustra claramente isso, pois a representação hexagonal impede os agentes de moverem-se entre o obstáculo triangular e

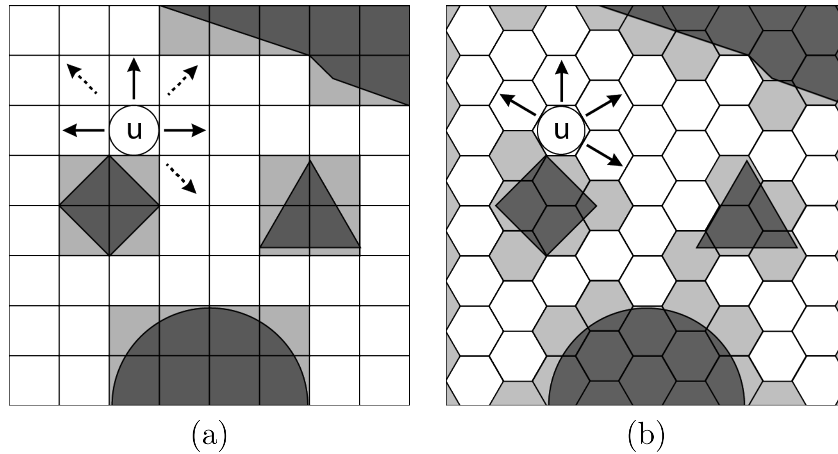


Figura 3.4: Movimentação possível para uma unidade u em uma grade (a) quadrada e (b) hexagonal.

o semi-circular. Para lidar com este problema, pode-se aumentar a resolução da grade, diminuindo o tamanho dos polígonos que representam cada célula, conforme ilustrado na Figura 3.5.

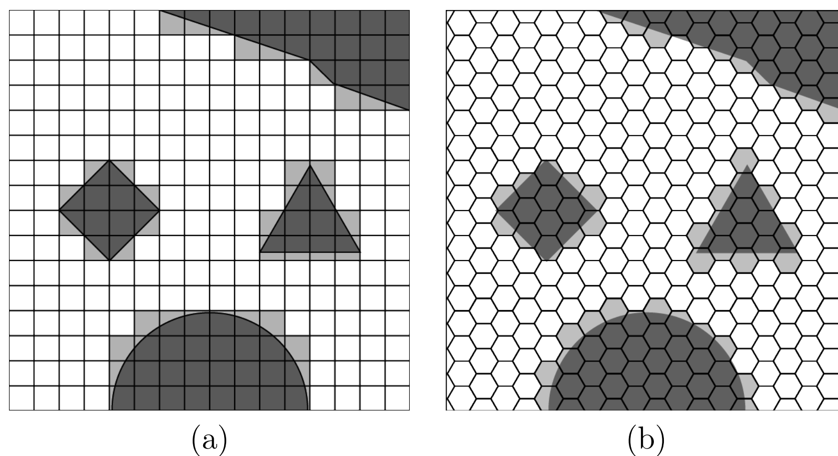


Figura 3.5: Grades com maior resolução para melhorar a representação. Grade (a) quadrada e (b) hexagonal.

Porém, mesmo aumentando muito a resolução da grade, esta será sempre uma aproximação imprecisa do ambiente. Além disso, aumentar demasiadamente a resolução faz com que o desempenho do *pathfinder* diminua muito, pois este está diretamente relacionado ao número de nós no grafo onde será efetuada a busca. A partir de um certo número de células, a busca se torna inviável. Por esses motivos, a representação em grade, embora simples, possui limitações que podem ser inaceitáveis.

3.1.2 Representação em Polígonos Convexos Variados

Uma alternativa à grade é a divisão do espaço em polígonos de formatos e tamanhos diversos [31]. Nesse caso, dentro de cada polígono um agente deve ser capaz de se locomover sem necessitar de uma técnica de *pathfinding*, andando em linha reta ou usando

alguma técnica de direcionamento como *steering behaviors* [28] para desviar de obstáculos menores e/ou dinâmicos.

Esse requisito é importante pois, se o agente não for capaz de se locomover de maneira direta dentro de uma célula do ambiente, será necessária uma técnica de planejamento de trajetória para calcular caminhos dentro das células.

Para tanto, os polígonos devem ser convexos, ou seja, deve ser possível traçar uma linha reta entre quaisquer dois pontos em seu interior. À rede de polígonos convexos que delimitam a área atravessável do ambiente é dado o nome de malha de navegação (*navigation mesh*) [31]. Existem vários métodos de particionar o espaço em setores poligonais. Alguns dos mais utilizados são: *quadtrees*, triangulações e divisões mistas ou arbitrárias.

Quadtrees [21] podem ser utilizadas eficientemente como técnica de indexação espacial. Nelas, cada nó representa um setor retangular que delimita uma seção do espaço que está sendo particionado, com o nó raiz cobrindo toda a área. Cada nó pode ser um nó de folha, sem filhos, ou um nó interno, que tem exatamente quatro nós filhos, um para cada quadrante obtido dividindo a área coberta na metade em ambos os eixos. Esse método é muito eficiente na representação de objetos pontuais, porém não é adequado para representar linhas e polígonos pois estas podem estar em mais de uma célula da árvore simultaneamente.

A triangulação é um caso específico da divisão em polígonos convexos [10] em que todos os polígonos tem três lados. Uma das vantagens da triangulação é que sua representação é uniforme (todas as células são representadas da mesma forma), o que facilita os cálculos efetuados no ambiente particionado.

Representações que usam polígonos de tamanhos e formas variadas podem ser construídas manualmente para cada ambiente de jogo e muitos jogos ainda utilizam este método de particionamento espacial (manual). Existem técnicas para particionar automaticamente o espaço em polígonos convexos, por exemplo, após calculada uma triangulação, pares de triângulos adjacentes podem ser unificados para formar quadriláteros convexos, o que gera uma malha com menos células. Vale notar que, nesse caso, o número de polígonos e arestas diminui, mas o número de vértices permanece o mesmo nos dois casos.

Quando comparadas a triangulações, malhas de navegação baseadas em polígonos convexos de N lados usam representações de espaço mais simples. Um octógono, por exemplo, que em uma malha de polígonos convexos é representado em apenas uma célula, precisa ser representado por no mínimo seis células em uma triangulação. Com isso obtém-se um ganho no custo de memória, já que se armazena um número menor de arestas. Há também uma redução no número de nós do grafo em relação à triangulação, já que, na maioria dos casos, é possível transformar dois triângulos adjacentes em um quadrilátero convexo, e por vezes é possível agregar ainda outros triângulos a um mesmo polígono, conforme a Figura 3.6.

Na Figura 3.6(a) é possível observar que o semi-círculo (na parte inferior do ambiente) foi transformado em meio octógono na Figura 3.6(b). Esta transformação é necessária devido ao requisito da divisão do espaço em polígonos, ou seja, o ambiente deve ser composto por segmentos de reta, e não por curvas. Logo, todos os obstáculos com forma arredondada devem ser aproximados para polígonos antes de particionar o espaço, o que pode causar imprecisão na implementação.

A desvantagem da representação em polígonos convexos variados quando comparada à triangulação é a maior complexidade dos cálculos realizados para obter informações sobre

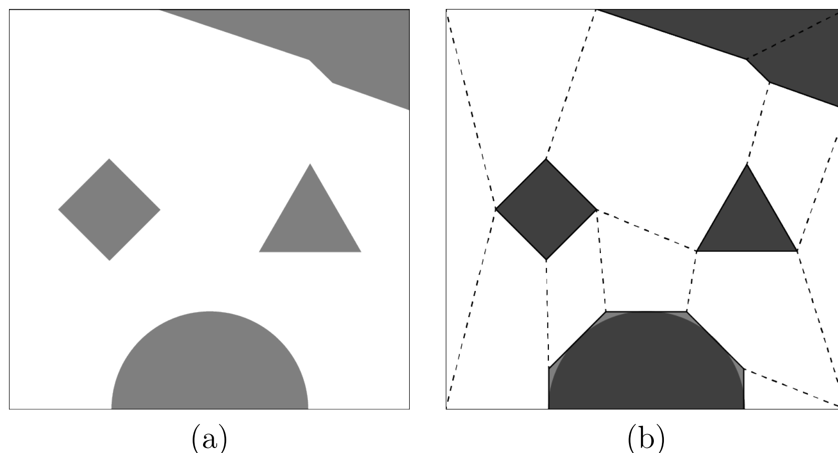


Figura 3.6: (a) Um ambiente com obstáculos e (b) sua representação usando polígonos convexos.

um nó. Por exemplo, se um ponto está contido em um polígono ou a qual polígono pertence uma aresta. Esses cálculos têm influência significativa no desempenho do algoritmo de *pathfinding* [31].

Essas desvantagens das outras técnicas apresentadas aliadas à existência algoritmos eficientes de construção e busca de trajetória em triangulações fizeram com que triangulação fosse a representação espacial escolhida para este trabalho.

3.1.3 Triangulação Restrita

Uma triangulação restrita é construída a partir de determinadas restrições, que podem ser pontos ou segmentos de retas (que podem formar polígonos). Para construir uma triangulação restrita, deve-se primeiro representar as restrições do ambiente como arestas restritas na triangulação. Arestas (não-restritas) são então adicionadas entre os pontos das arestas restritas (inclusive aos seus pontos de interseção), sem que estas arestas não-restritas se sobreponham, até que todos os pontos tenham sido conectados e não seja possível adicionar mais arestas. Ao final desse procedimento, o ambiente estará completamente dividido em triângulos, conforme ilustra a Figura 3.7.

A Figura 3.8 mostra um obstáculo adicionado a dois ambientes, sendo um representado por grade (Figura 3.8(a)) e outro por triangulação (Figura 3.8(b)). O obstáculo está representado de maneira precisa nos dois casos, porém no primeiro ambiente temos uma maior quantidade de células se comparado ao segundo, e consequentemente, um pior desempenho na busca. Isso porque, quando um obstáculo é adicionado a uma triangulação, ele é representado com precisão sem a necessidade de aumentar drasticamente a quantidade de células. É possível ver que a triangulação resolve de uma maneira mais eficiente o problema da representação de obstáculos diversificados e é capaz de lidar com o problema de agentes e obstáculos com tamanhos diferenciados.

Vale notar que a precisão da representação de objetos com formas arredondadas depende de quantos segmentos de reta serão utilizados na aproximação destes por polígonos. Quanto mais segmentos, mais precisa será a representação da curva, porém mais triângulos serão gerados em torno do objeto na triangulação, aumentando o custo da busca.

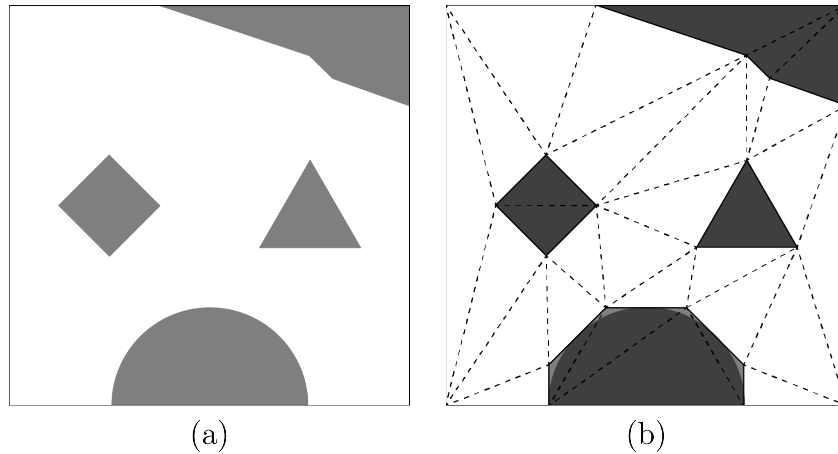


Figura 3.7: (a) Representação de um ambiente com obstáculos, e (b) uma triangulação restrita desse ambiente. Em (b), linhas sólidas representam arestas restritas, e linhas pontilhadas representam arestas não-restritas.

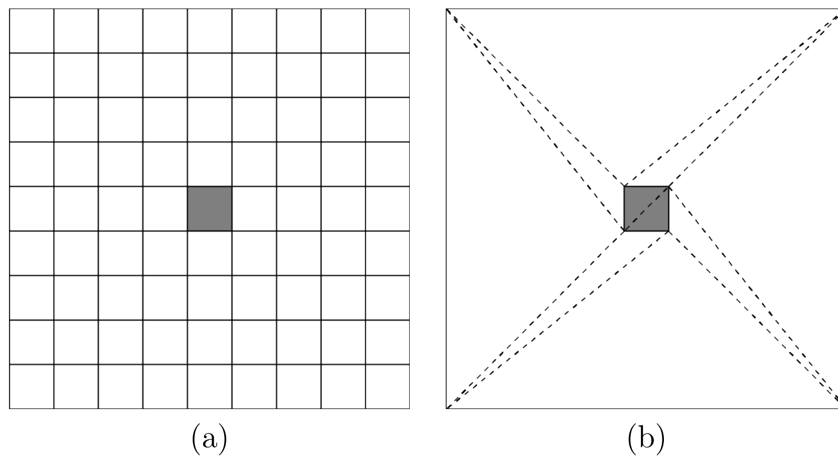


Figura 3.8: (a) Grade com precisão suficiente: 81 células. (b) Triangulação restrita: 10 polígonos.

Em geral, uma representação por triangulação tem muito menos células que uma representação por grade com resolução grande o suficiente para representar de maneira fiel o ambiente, logo, o desempenho de uma busca em uma malha triangular tende a ser melhor que o de uma busca feita em um *grid*.

Outra vantagem de malhas poligonais sobre grades é a resolução do problema de ter unidades de tamanhos diferentes no ambiente. Utilizando grade, é necessário fazer com que uma unidade ocupe exclusivamente mais de uma célula em um determinado momento, o que não acontece em malhas triangulares, por serem contínuas, ou seja, uma célula pode conter mais de uma unidade ao mesmo tempo.

Apesar disso, esta característica cria dois novos desafios: localização de pontos na malha e execução dos movimentos das unidades. Para descobrir a que triângulo pertence um ponto arbitrário, são necessários vários cálculos. Outra característica inerente à grade é que as unidades sempre ocupam no mínimo uma célula do espaço. Ou seja, mover uma unidade é simples, basta trocar quais células ela ocupa. Ao utilizar malhas triangulares,

as unidades podem efetivamente se mover distancias arbitrárias em alguma direção válida. Logo, é necessário pensar como será efetuada essa movimentação e como será evitada a colisão entre os agentes (algo muito simples de fazer utilizando-se grade).

3.1.4 Triangulação de Delaunay Restrita e Dinâmica

A triangulação implementada neste trabalho é a Triangulação de Delaunay Restrita e Dinâmica (*Dynamic Constrained Delaunay Triangulation* - DCDT), desenvolvida por M. Kallmann [20], e baseada no trabalho de D. Demyen e M. Buro [10], de implementação aberta e disponível como parte do motor de jogos *ORTS* [6].

Uma triangulação simples recebe como entrada um conjunto de pontos (Figura 3.9) e adiciona arestas entre eles de modo que estas não se cruzem. Quando não for mais possível adicionar essas arestas, todas as áreas serão triangulares dentro da envoltória convexa gerada pelos pontos, conforme ilustrado na Figura 3.10. Uma triangulação de Delaunay desse mesmo conjunto de pontos adiciona as arestas de modo que o ângulo mínimo de cada triângulo seja o maior possível. Essa propriedade visa evitar que triângulos estreitos sejam gerados, tornando a área dos triângulos mais uniforme e fazendo com que a distribuição de área do ambiente por célula também o seja. Um algoritmo eficiente para obter uma Triangulação de Delaunay dado um conjunto de pontos foi proposto por M. Anglada [1] e utilizado na implementação deste trabalho.

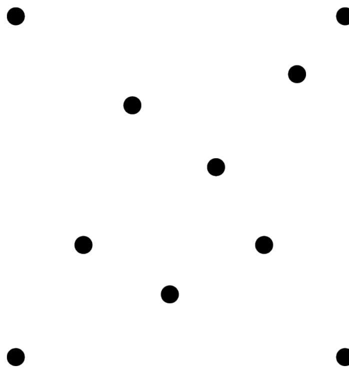


Figura 3.9: Um conjunto de pontos.

A representação do ambiente como uma triangulação geralmente é feita usando alguma forma de triangulação restrita (*Constrained Triangulation* - CT), onde segmentos de reta representam os limites entre espaços onde é possível a movimentação e onde não é. Esses segmentos são compostos por um par de pontos início e fim. Ao serem adicionados à triangulação, esses pontos são denominados vértices, pois os mesmos serão utilizados para compor os vértices dos polígonos na malha triangular.

Aos segmentos de reta que compõe essa malha é dado o nome de arestas restritas, pois os mesmos serão representados na triangulação por uma ou mais arestas de triângulos, que serão marcadas como restritas (impedindo a movimentação através delas). As outras arestas que compões os triângulos (e não impedem a movimentação através delas) são denominadas arestas não restritas. Os polígonos que compõe uma malha triangular são muitas vezes também denominados faces ou células.

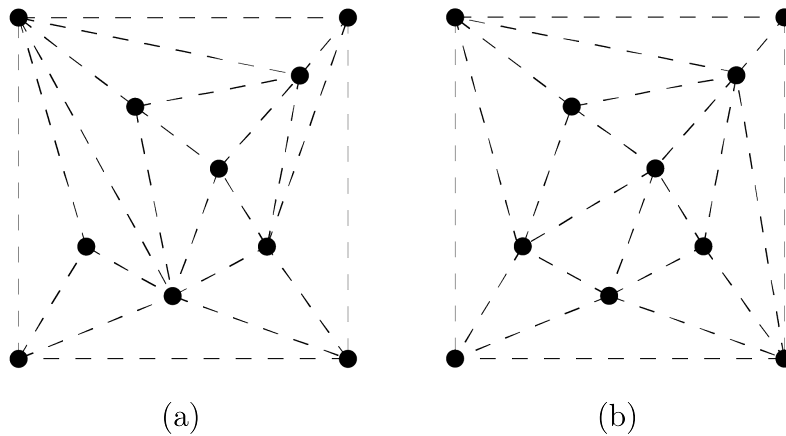


Figura 3.10: (a) Uma possível triangulação e (b) a triangulação de Delaunay de um mesmo conjunto de pontos.

Normalmente, o ambiente é limitado por uma envoltória convexa, que é composta de arestas restritas para especificar que os objetos dentro dela não podem deixá-la. Na maioria dos jogos, essa envoltória é um retângulo que delimita a área onde o jogo ocorre, também chamada de mapa ou ambiente. Um possível mapa com suas restrições é exemplificado na Figura 3.11.

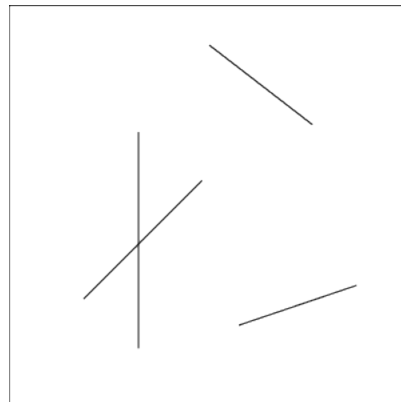


Figura 3.11: Um conjunto de arestas representando as restrições de movimentação de um ambiente.

Para fazer uma triangulação de Delaunay restrita (*Constrained Delaunay Triangulation* - CDT), adicionam-se arestas não-restritas ao conjunto de arestas restritas que formam o mapa, de modo a maximizar o ângulo interno mínimo dos triângulos compostos por essas arestas. Uma possível triangulação restrita do ambiente da Figura 3.11 é mostrada na Figura 3.12(a).

Uma triangulação restrita com a propriedade de Delaunay é, em geral, uma melhor abstração do ambiente para a execução do algoritmo de *pathfinding* (como será mostrado na Seção 3.2). Além disso, mais especificamente, essa propriedade garante que, se existe um caminho válido entre dois pontos da triangulação, existe um caminho válido que não passa mais de uma vez por cada triângulo [1].

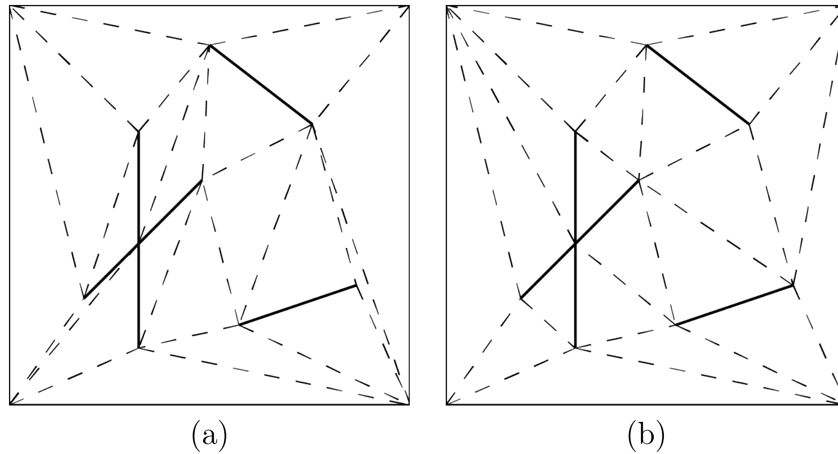


Figura 3.12: (a) Uma possível triangulação restrita do conjunto de arestas mostrado0000 e (b) a triangulação de Delaunay desse mesmo conjunto.

São consideradas alterações na triangulação a inserção e a remoção de vértices e arestas restritos. A movimentação de restrições é feita através da remoção da restrição de sua posição anterior e reinserção da mesma em sua posição nova. Para o *pathfinding*, vértices são considerados como obstáculos, por isso, na maioria das vezes, os vértices são usados apenas como pontos de início, fim, e interseção nas restrições. É possível, porém, adicionar um vértice para criar um obstáculo pontual.

Uma triangulação é dita dinâmica quando uma alteração pode ser ajustada localmente, sem a necessidade de refazer toda a malha. Kallmann [20] implementa um mecanismo eficiente para alterar a triangulação de modo a causar o número mínimo de mudanças na malha, algoritmo que foi implementado neste trabalho.

Construção da triangulação

A seguir são descritos os procedimentos necessários para construir uma triangulação dinâmica e restrita de Delaunay. Para adicionar um vértice é necessário, primeiramente, localizá-lo na triangulação e então verificar se ele já existe, e se está em uma aresta ou se está na face de um triângulo.

A localização de um triângulo na malha dado um ponto arbitrário pode ser feita de várias maneiras. Uma delas é escolher, por exemplo, um triângulo cujo centro seja o mais próximo do centro do mapa, e examinar seus vizinhos, e triângulos adjacentes um por um, sempre andando em direção ao ponto que se deseja encontrar, até que seja testado o triângulo que contém o ponto procurado. Em seguida verifica-se:

- Se o vértice coincide com outro, não é necessário adicioná-lo.
- Se está sobre uma aresta, essa deve ser dividida em duas outras arestas, usando os pontos de início e fim e o novo vértice inserido como ponto em comum entre as duas. Os triângulos formados pela antiga aresta são então divididos, através da adição de uma aresta (não-restrita) entre o novo vértice e os vértices opostos à aresta que foi dividida, conforme a Figura 3.13.

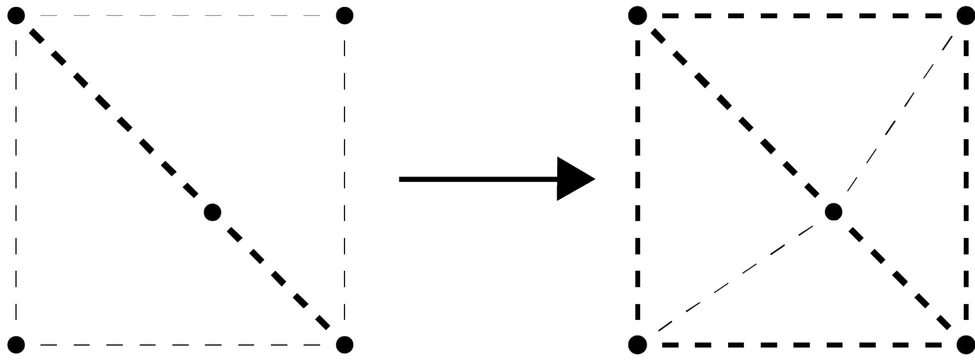


Figura 3.13: Inserção de um vértice sobre uma aresta.

- Se o vértice estiver na face de um triângulo, arestas não-restritas são adicionadas entre ele e os vértices deste triângulo, de modo a criar 3 novas faces triangulares, como na Figura 3.14.

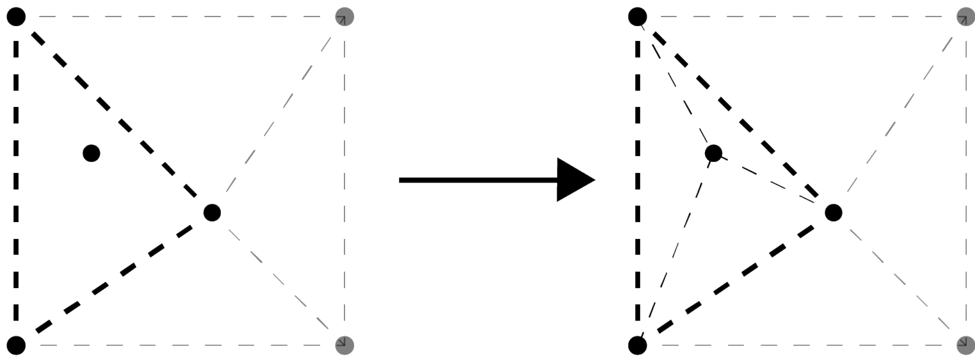


Figura 3.14: Inserção de um vértice em uma face.

Quando um vértice é adicionado sobre uma aresta ou uma face, os triângulos ao seu redor podem perder a propriedade de Delaunay. Por isso, é necessário verificar todas as arestas dos triângulos envolvidos na inserção, pois elas podem precisar ser alteradas para corrigir a malha. Segue o procedimento de verificação e correção, para cada aresta. Primeiro, forma-se um quadrilátero através da junção dos dois triângulos adjacentes a ela, e a propriedade de Delaunay é conferida nesses triângulos. Se foi perdida, a aresta que forma a diagonal desse quadrilátero deve ser "virada", para que ligue o outro par de vértices, conforme a Figura 3.15. Isso garante que a propriedade de Delaunay será restaurada [1].

Esta alteração pode, por sua vez, causar a perda da propriedade em outros triângulos adjacentes à mudança, sendo necessário verificar as arestas destes. Desse modo, a área afetada pela inserção de um vértice se expande a partir da região da inserção. No pior caso, seria necessário trocar todas as arestas da triangulação, porém o número médio de trocas necessárias por inserção, independente do estado da malha, é constante [15].

A inserção de uma restrição ocorre em vários passos, conforme Figuras 3.16 a 3.19. Uma restrição normalmente representa um único obstáculo que pode ser composto por

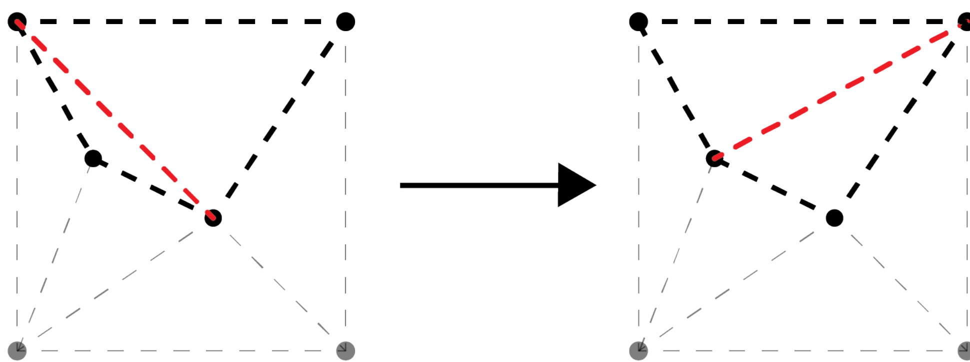


Figura 3.15: Procedimento de virar uma aresta irregular para restaurar sua propriedade de Delaunay.

vários segmentos de reta. A inserção de cada um desses segmentos será descrita a seguir e pode corresponder a múltiplas arestas restritas na triangulação.

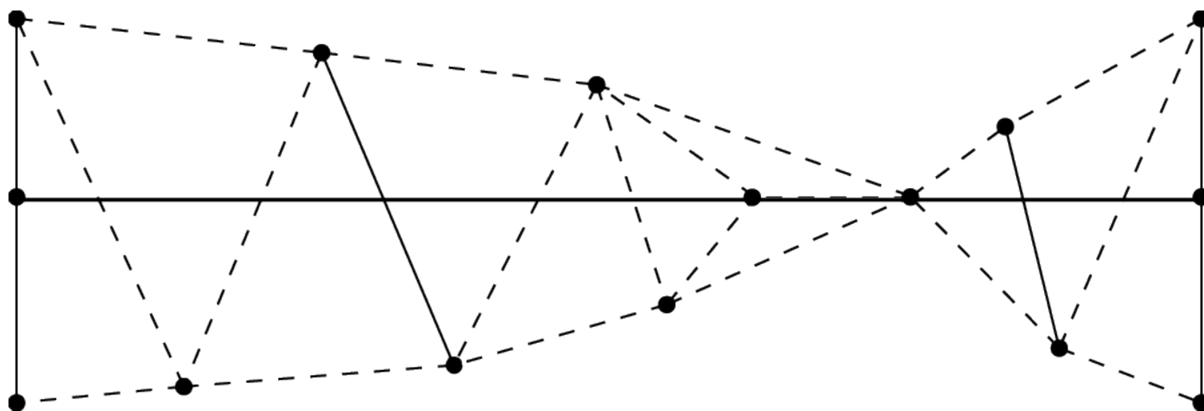


Figura 3.16: Inserindo um segmento restrito na triangulação.

Primeiro, é necessário localizar os pontos de início e fim de cada segmento. Esses pontos são adicionados como vértices à triangulação (Figura 3.16). Nesta figura, e nas Figuras 3.17, 3.18 e 3.19, a restrição sendo adicionada é o segmento de reta horizontal que corta a triangulação horizontalmente.

Em seguida, os pontos de interseção entre o novo segmento e as arestas restritas existentes são calculados e inseridos. Para cada ponto de interseção com uma aresta restrita, a mesma e o segmento são divididos em duas novas arestas e dois novos segmentos, respectivamente (Figura 3.17). Depois disso, todas as arestas não-restritas que cruzam o novo segmento restrito são encontradas e removidas da triangulação (Figura 3.18).

Como último passo, insere-se esse segmento da nova restrição como uma aresta restrita na malha. Note que se esse segmento fizer interseção com outra restrição em um vértice, isso também divide o segmento em outros menores, mas a outra restrição não é alterada. Se um segmento dessa restrição se sobrepuser a uma aresta existente na triangulação (pontos de início e fim iguais), só é necessário tornar aquela aresta restrita. Ou seja, uma aresta não-restrita pode se tornar uma aresta restrita, e uma aresta restrita pode

representar várias restrições diferentes. Por fim, as regiões não triangulares originadas no passo anterior são trianguladas (Figura 3.19).

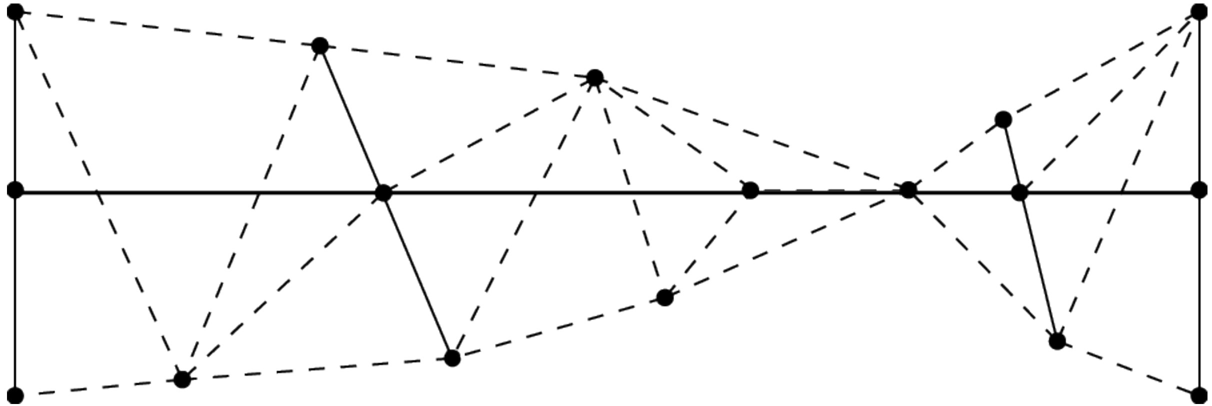


Figura 3.17: Inserindo vértices nos pontos de interseção entre o novo segmento e as arestas restritas.

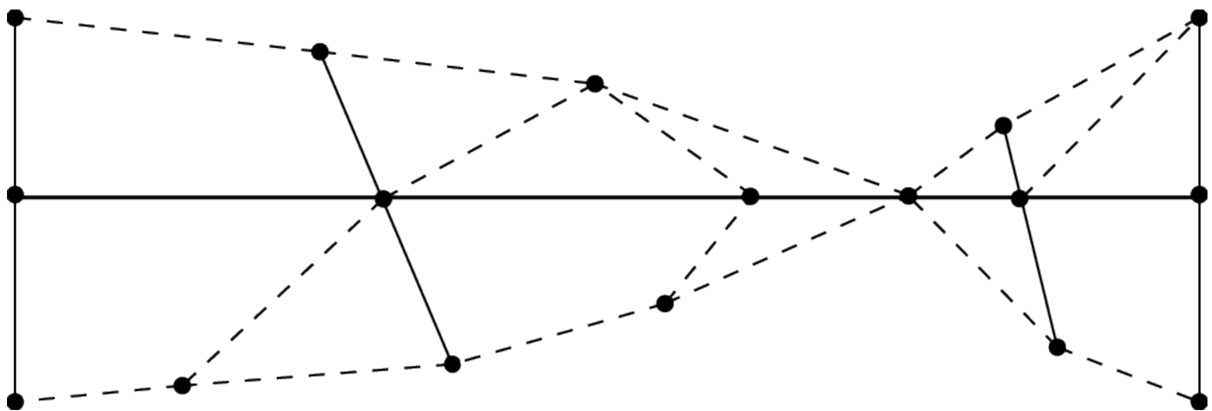


Figura 3.18: Remoção das arestas não-restritas que cruzam o novo segmento.

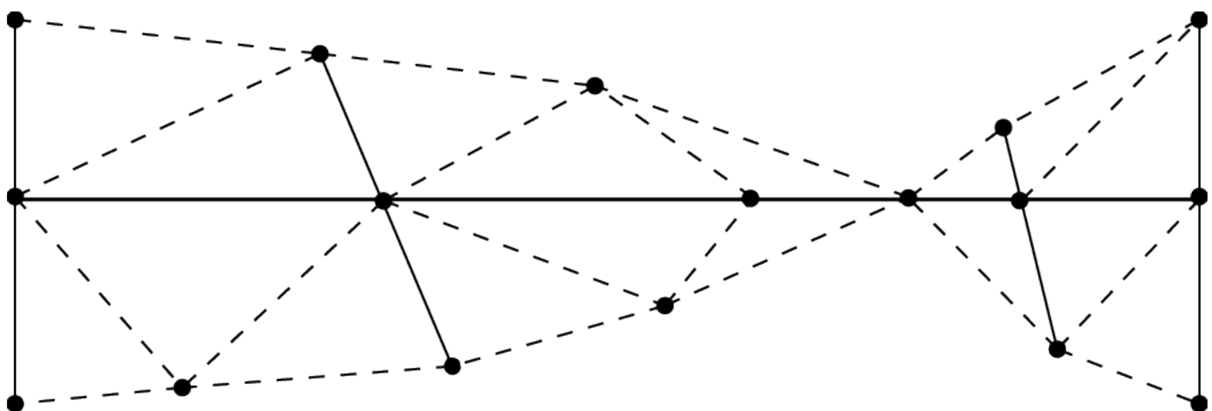


Figura 3.19: Triangulação final, após a inserção do novo segmento (que resultou na inserção de 5 arestas restritas).

A retirada de um vértice da triangulação ocorre pela remoção de todas as arestas em que esse vértice é um ponto de início ou fim, e então remove-se o vértice em si. Isso gera uma área não triangular ao redor do vértice removido, que deve ser ajustada. Assume-se que o vértice a ser removido não é um ponto de início ou fim para nenhuma aresta restrita, pois nesse caso ele não poderia ser removido (a restrição deveria ser removida primeiramente). Para a remoção de uma restrição, primeiramente localizam-se as arestas correspondentes à restrição, então remove-se a restrição das arestas. Caso as arestas em questão não representem outra restrição, elas são transformadas em arestas não-restritas.

Por fim, é necessário verificar se os pontos de início e fim da restrição que está sendo removida pertencem a alguma outra restrição. Caso não pertençam, esses vértices devem também ser removidos da triangulação, usando o procedimento descrito anteriormente. Se um vértice era ponto de interseção entre a restrição removida e uma outra restrição (como, por exemplo, duas arestas restritas separadas por este vértice, colineares e representando a mesma restrição), esse vértice e as arestas restritas separadas por ele devem ser removidos da triangulação e um único segmento deve ser inserido para substituir os dois segmentos menores que foram removidos.

Desta forma, obtém-se uma malha de navegação que representa o particionamento do ambiente, onde é possível realizar uma busca discreta por trajetória.

3.2 Busca por Trajetória

Dada uma unidade no ambiente em uma posição válida e um destino também válido neste mesmo ambiente, procuramos pelo melhor caminho válido que leve a unidade de sua posição atual (inicial) até seu destino. Definimos como melhor caminho o menor caminho válido entre esses pontos, ou seja, aquele com o menor custo para a unidade. Apesar disso, a metodologia aqui descrita pode ser utilizada com outras métricas de custo.

Como dentro de uma mesma célula do ambiente discretizado as unidades podem mover-se livremente (todos os caminhos dentro de uma mesma célula são válidos, conforme visto na Seção 3.1.2), o problema de encontrar uma trajetória fica reduzido a encontrar o conjunto de células do ambiente, ordenadas da origem para o destino, que contém este caminho.

Considerando o ambiente como um grafo, cada nó representa uma célula do ambiente discretizado e são conectados, se e somente se, as células representadas são adjacentes (tem uma aresta em comum) e é possível mover-se entre elas (essa aresta não é restrita). Tal grafo é exemplificado na Figura 3.20.

O algoritmo *A** [16] é amplamente utilizado para encontrar o caminho em grafos, por ser completo (sempre encontra uma solução, se existir), ótimo (a solução encontrada sempre é a melhor, assumindo o uso de uma heurística admissível) e eficiente (o mínimo necessário de nós é expandido até encontrar uma solução). Como o algoritmo *A** lida com células, será necessário porém fazer alguns ajustes para que o algoritmo seja ótimo quando aplicado em uma triangulação.

3.2.1 Algoritmo *A**

Suponha um espaço de estados S, onde será feita a busca. Para cada estado $s \in S$, temos o conjunto dos sucessores de s, $Suc(s) \subset S$, tal que $\forall s \in Suc(s)$, s' pode ser

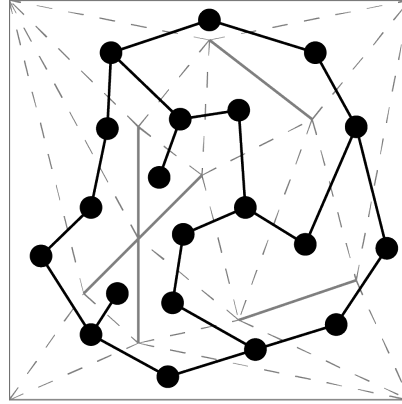


Figura 3.20: Grafo representando um ambiente discretizado.

alcançado através de uma única ação a a partir de s. Esses estados são chamados *filhos* de s. Além disso, existe um custo associado a esta ação $c(s, a, s') \geq 0$.

Define-se um estado s como *pai* de um estado s' se e somente se s' é filho de s. Um *predecessor* de um estado s' é qualquer estado tal que s é ou o pai de s' ou um ancestral do pai de s', ou seja, partindo de algum estado s, s' pode ser alcançado através de qualquer número de ações a. No algoritmo $A*$ todos os estados s' armazenam a informação de quem é seu pai s, pois são os antecessores de um estado qualquer que definem o caminho no grafo do estado inicial até este.

O valor g de um estado da busca, que representa a distância da trajetória já coberta, é o custo associado a chegar ao estado atual a partir do estado inicial. Para *pathfinding* em uma triangulação, esse custo é o comprimento do caminho da posição inicial até algum ponto no triângulo associado com o estado atual da busca. A esse valor é associada uma incerteza, pois não se sabe qual ponto no triângulo deve ser usado na avaliação (o ponto exato depende dos nós que serão expandidos nos estados sucessores).

O valor h, ou valor da heurística de um estado, é uma estimativa da distância entre o estado atual e o estado objetivo através do caminho de menor custo. Usaremos aqui a menor distância (ou seja, um segmento de reta) entre o ponto por onde a busca chegou ao triângulo atual e o objetivo, pois isso garante que o valor de h seja sempre o mais alto possível, e dessa forma garantimos que a heurística é admissível e, portanto, o algoritmo é ótimo.

O valor f é calculado para um estado s como $f(s) = g(s) + h(s)$. Isso é equivalente ao custo de todo o caminho do ponto de partida ao objetivo, sendo que o caminho entre o estado inicial e o atual é dado pelos estados que precedem s. f é então uma estimativa da distância entre as posições inicial e final, considerando que o caminho entre o estado atual e o objetivo é o menor possível.

O algoritmo $A*$ classifica os estados $s \in S$ em abertos ou fechados, não-visitados ou visitados. Os estados abertos são aqueles cujo valor de f dos filhos ainda não é conhecido. Os estados fechados são aqueles por onde a busca já passou, isto é, o valor f dos filhos já foi calculado. Primeiramente, o valor f do estado inicial s_i é calculado e s_i é colocado no conjunto de estados abertos. Então, a cada iteração do algoritmo o estado não-visitado s com o menor valor f é visitado (ou expandido), e seus filhos são gerados através da função de sucessão, ou seja, é calculado o valor f de cada um dos filhos de s, que são colocados

no conjunto de estados abertos e sua informação de parentesco é armazenada. Por fim, s é colocado no conjunto de estados fechados.

Este procedimento se repete até que o estado objetivo seja visitado (nó destino) ou que a lista de estados abertos seja vazia (não sendo possível expandir nenhum estado). Quando o destino é atingido, o algoritmo retorna a lista ordenada dos nós que contém o menor caminho até o destino. Caso não seja possível visitar mais nós, não existe um caminho até o destino, e um caminho vazio é retornado.

A função heurística h tem certas propriedades que podem influenciar no comportamento do algoritmo *A** . É fundamental que a heurística seja admissível, isto é, em nenhum estado o valor de h pode superestimar a distância daquele estado ao objetivo [16]. Ou seja, $\forall s \in S$, $h(s) \leq h^*(s)$, onde $h^*(s)$ é a menor distância real entre o estado s e o objetivo. Isso garante que a busca do *A** seja completa, e que na primeira vez que o estado objetivo seja visitado, o caminho pelo qual este foi atingido seja ótimo.

A heurística utilizada também deve ser consistente, ou seja, para qualquer estado $s \in S$ e seu sucessor $s' \in Suc(s) \subset S$, $h(s) \leq h(s') + c(s, a, s')$, e além disso, $h(s_g) = 0$, onde s_g é o estado objetivo. Isso significa que, ao mover de um estado para outro, não é possível se aproximar do objetivo mais do que o custo para se mover entre esses dois estados. Uma heurística com essa propriedade garante que a busca executada pelo *A** seja ótima, pois sempre que um estado $s \in S$ for expandido pela primeira vez o caminho através do qual ele foi expandido é um caminho ótimo até este estado. Além disso, por definição, toda heurística consistente é também admissível.

3.2.2 *A** em uma Triangulação

A principal vantagem de buscar a trajetória em uma representação em grade é que a distância percorrida até o ponto correspondente a cada estado da busca é exatamente conhecida. Isso se deve ao fato de que o objeto segue o caminho entre os pontos centrais de células adjacentes em linha reta. Se as células forem pequenas em relação ao tamanho dos objetos, esse caminho pode ser bem preciso (não é exato pois sempre sofre as restrições espaciais da grade).

Em contraste, assumir que o objeto se deslocará sobre um caminho simples ao executar o *pathfinding* em uma triangulação pode causar falhas no caminho calculado. Ao aproximar o deslocamento entre os triângulos por uma linha reta entre os centros, por exemplo, surgem dois problemas. Primeiro, esse caminho pode cruzar outros triângulos, até mesmo restritos, mesmo que os triângulos inicial e final sejam adjacentes, como é ilustrado na Figura 3.21. Em segundo lugar, mesmo que não sejam inválidos, os caminhos produzidos usando aproximações simples como essa podem possuir estimativas de distância ruins (como mostrado na Figura 3.22). Isso acontece porque os triângulos são muito grandes em relação aos objetos e seu posicionamento é resultado do ambiente, que geralmente forma um caminho entre os centros desses triângulos que não reflete o caminho que o agente deve percorrer.

Mesmo se durante a busca for aplicado um algoritmo de refinamento complexo, como o algoritmo do funil (que será detalhado na Seção 3.3), não é possível ter certeza da distância percorrida em um triângulo específico sem saber por onde a busca vai continuar, conforme Figura 3.23, que mostra três possíveis caminhos candidatos ao menor dentro de um mesmo

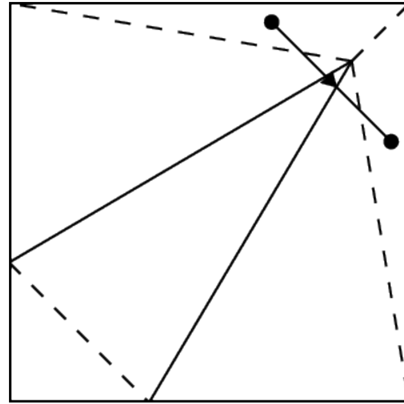


Figura 3.21: Um caminho entre centros de dois triângulos adjacentes cruzando outros triângulos (e restrições).

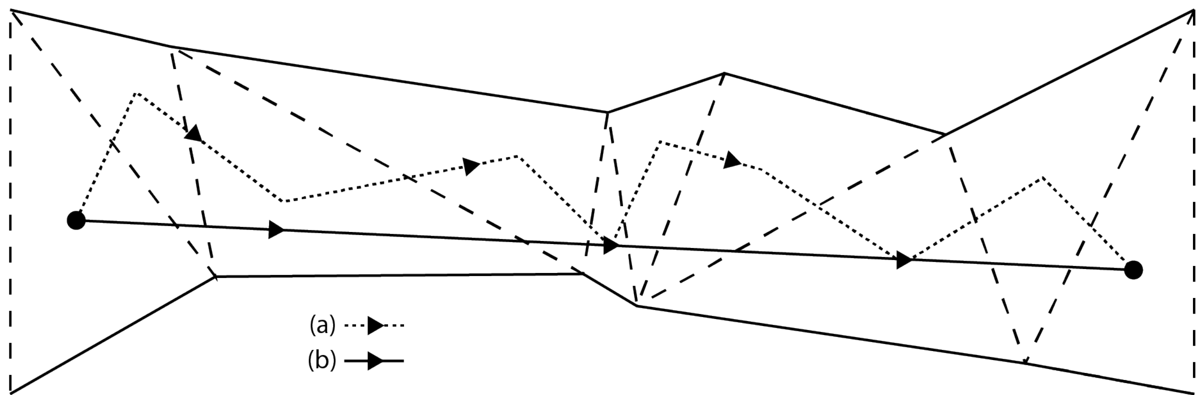


Figura 3.22: Um caminho gerado pelos centros dos triângulos (a). A estimativa do comprimento do menor caminho (b) fica prejudicada.

conjunto de triângulos. É necessário executar a busca com essa incerteza, mesmo sabendo que, após a conclusão da busca essa informação não é mais necessária.

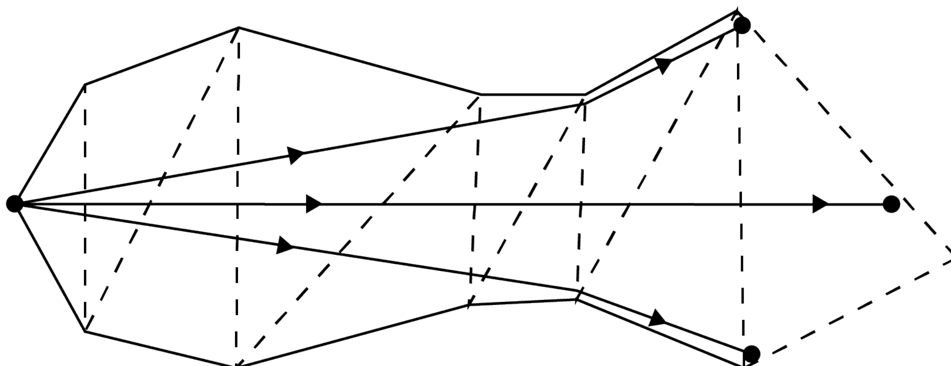


Figura 3.23: O menor caminho até um ponto em um triângulo específico depende do destino.

3.2.3 Custo Acumulado e Heurística

Vamos assumir que o comprimento exato de g seja conhecido para todos os triângulos durante a busca. Supondo, por exemplo, que o comprimento do caminho em um estado seja a soma da distância percorrida, em segmentos de reta, entre os pontos médios de cada aresta no caminho, ou seja, são considerados para os cálculos de distância os pontos médios das arestas (não restritas) por onde a pesquisa entrou em cada triângulo.

Nesse caso, o valor de g para um estado de busca é a soma dos comprimentos dos segmentos entre os pontos médios das arestas atravessadas por esse caminho até o triângulo correspondente ao estado atual. O valor de h é calculado como a distância euclidiana entre este ponto e o objetivo. Em duas dimensões, a distância euclidiana d entre dois pontos p e q é calculada como $d = \sqrt{(q_x - p_x)^2 + (q_y - p_y)^2}$.

A distância euclidiana é conhecida por ser uma heurística admissível e consistente [19]. Ao utilizá-la em uma busca com o algoritmo $A*$, assegura-se que o caminho pelo qual a busca chegou ao estado atual é ótimo sempre que qualquer triângulo é expandido durante a busca (assumindo que se sabe os valores exatos de g para cada estado). Portanto, cada triângulo é expandido apenas uma vez.

Naturalmente, como os valores g de cada triângulo não são conhecidos durante a busca, isso nem sempre é verdade. Já que a busca nunca vai expandir um mesmo triângulo duas vezes, assumir esse cálculo simples de g acaba resultando em caminhos subótimos. Nesses casos, a busca não faz a expansão de um triângulo em um estado da busca cuja distância real do caminho é ótima, mas cuja distância estimada (aqui o comprimento do caminho passando pelos pontos médios das arestas dos triângulos) é maior do que a de outro estado de busca. Tal situação é ilustrada na Figura 3.24.

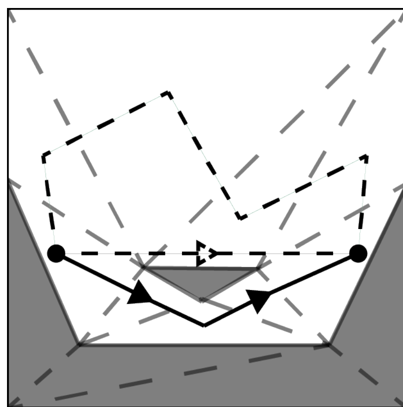


Figura 3.24: Um caso onde o caminho calculado utilizando uma heurística simples não é o mais curto.

Nesse caso, a distância através dos pontos médios dos triângulos entre os pontos inicial e destino é maior passando por cima do obstáculo do que passando abaixo. Assim, o caminho correspondente à parte inferior do ambiente é escolhido. No entanto, o caminho de menor custo passa pela parte superior do ambiente, mas não foi explorado devido à imprecisão na estimativa do valor de g e da busca expandir cada triângulo no máximo uma vez, além de parar ao encontrar a primeira solução.

A solução utilizada nessa abordagem para resolver o problema dos caminhos subótimos, proposta por D. Demyen [10], é, primeiro, não descartar a expansão de um triângulo

durante a busca, mesmo que um caminho melhor para este triângulo tenha sido encontrado. Como não se sabe exatamente quando o melhor caminho para um triângulo é encontrado não se deve descartar nenhum desses caminhos dado que eles podem fazer parte da solução ótima.

Da mesma forma, não é mais garantido que a primeira expansão do destino resulta no caminho ótimo. Isso é, para encontrar o caminho ótimo entre o início e o destino em uma triangulação requer a continuação da busca depois que o objetivo já foi alcançado, pois deve-se considerar a possibilidade de existirem caminhos cuja distância estimada é maior que a de outros caminhos, embora a distância real seja menor.

Sendo assim, é melhor fazer a busca utilizando um algoritmo que possa retornar um resultado a qualquer momento, assim é possível encontrar uma solução inicial (potencialmente subótima) e continuar a busca, procurando por possíveis caminhos melhores até convergir para uma solução ótima. Esses algoritmos são flexíveis pois encontram uma resposta inicial rapidamente e podem retornar uma solução se interrompidos a partir desse ponto. Essa solução pode melhorar se mais tempo for dado para a busca.

Todas as vezes que o objetivo for expandido, o comprimento exato do caminho pode ser determinado. Esse caminho é mantido se for o primeiro a ser encontrado ou se tiver um custo menor do que o melhor caminho encontrado até então. A busca continua assim, procurando por soluções melhores, até que uma solução ótima seja encontrada ou até que seja atingido um critério de parada.

Esta abordagem é bastante vantajosa quando se trabalha com problemas em tempo real, porque uma mesma busca consegue chegar a uma solução mesmo tendo diferentes quantidades de recursos disponíveis. Além disso, em muitos casos, a busca encontra o melhor caminho muito antes de determinar que este é ótimo. Portanto, mesmo parando o algoritmo a qualquer momento é possível conseguir uma solução ótima.

Isto levanta outra preocupação: como a busca determina se uma solução encontrada é ótima? Basicamente, o que se quer é saber se existe algum estado da busca em aberto (ainda não foi expandido) que poderia produzir uma solução de menor custo do que o melhor já encontrado, que é conhecido exatamente. Por isso a busca deve determinar quando os estados de pesquisa em aberto só resultam em caminhos mais longos.

A solução para este problema é forçar o valor de g para o limite inferior, ou subestimar o seu valor real, durante a busca. Isso significa, $\forall s \in S, g(s) \leq g'(s)$, onde $g'(s)$ é a distância real entre o início e o estado s pelo caminho determinado neste estado. Então, tem-se que $\forall s \in S, g(s) \leq g'(s) \wedge h(s) \leq h'(s) \Rightarrow f(s) \leq f'(s)$, onde $f'(s)$ é o menor caminho entre o início e o destino através do caminho definido no estado s.

Seja $s' \in S$ o estado correspondente ao caminho ótimo, ou seja, $\forall s \in S, f'(s') \leq f'(s)$. Seja $s' \in Q$ o primeiro estado na fila de busca. $Q \subset S$, isto é, $\forall s \in Q, f(s') \leq f(s)$. Então, se $f(s') \geq f'(s')$, temos que $\forall s \in Q, f'(s) \geq f(s) \geq f(s') \geq f'(s')$, o que significa que o custo atual do melhor caminho através de todos os estados na fila tem, no mínimo, o mesmo custo da solução já encontrada, dado que o valor de f do primeiro estado na fila de busca é maior do que o custo do caminho já encontrado.

Isto vale para todos os estados na fila e para todo o espaço de busca, dado que todos os filhos do estado inicial foram colocados na fila e que todos os caminhos originados no estado inicial devem passar pelos seus filhos. Dessa forma, não existem caminhos alternativos para considerar, o que fornece o conhecimento necessário para interromper a busca sabendo que o caminho encontrado é a solução ótima.

Portanto, para a busca convergir para uma solução ótima, mesmo com o valor impreciso de g, devemos considerar vários caminhos possíveis para um mesmo triângulo e continuar a busca após a solução inicial ser encontrada pelo algoritmo. Além disso, para chegar a uma condição de parada para a busca, o valor de g deve ser estimado como um limite inferior dos verdadeiros valores.

3.2.4 *Triangulation A**

Para fazer a busca em triangulação foi utilizado um algoritmo denominado *Triangulation A** (TA*) [10]. Segue o funcionamento do algoritmo.

Primeiramente, deve-se descobrir em qual triângulo o ponto de início está. Um estado inicial correspondendo a esse triângulo é então colocado na fila de prioridade da busca com os valores $g = 0$ e com a distância euclideana entre os pontos de início e destino para h.

A cada passo da busca, o estado com o menor valor f é retirado da fila e expandido. A função de sucessão gera um estado filho para cada triângulo adjacente ao triângulo atual cuja aresta adjacente a esse não é restrita. Essa aresta é então usada para calcular os valores de g e h para os novos estados. O valor de h é a distância euclideana entre o ponto destino e o ponto mais próximo a ele nessa aresta. Essa heurística é conhecida por ser admissível e consistente, e essas propriedades são usadas para calcular o valor de g.

A precisão do valor de g tem um impacto considerável no número de estados extras processados, logo é importante estimá-lo o melhor possível, sempre evitando uma superestimativa. Sabendo disso, calcula-se essa estimativa como o número máximo em um conjunto de limites inferiores, resultando em um limite inferior máximo. O limite inferior para um estado s' com pai s para um objeto de raio r, é estimado da seguinte forma:

- O primeiro e mais simples é a distância entre o ponto de início e o ponto mais próximo a ele na aresta de entrada do triângulo atual. Como com o valor h, essa estimativa não ultrapassa o valor real e satisfaz a desigualdade do triângulo tal que $g(s) \leq g(s') + c(s, a, s')$.

- O segundo é $g(s)$ somado à distancia entre os triângulos associados com s e s'. Assumimos que o valor g é um limite inferior e queremos adicionar a distância mais curta para chegar em outro limite inferior. Mais uma vez, a medida será feita utilizando as arestas através das quais os triângulos foram acessados pela busca. Como os triângulos são adjacentes, essa é a distância de mover-se através do triângulo associado com s. Se a as arestas de entrada dos triângulos correspondentes a s' e s formam um ângulo Θ. Essa estimativa é calculada por $g(s) + r\Theta$.

- Outro valor de limite inferior para $g(s')$ é $g(s)+(h(s)-h(s'))$, ou seja, o valor de g do pai somado à diferença entre seu valor h e o h de seu filho. Essa é uma subestimativa pois a distância Euclideana aplicada a essa heurística é consistente. Para provar que com $g(s') = g(s) + h(s) - h(s')$, $g(s') \leq g^*(s')$, podemos assumir que o valor g do pai é subestimado, ou $g(s) \leq g^*(s))$ para obter $g(s') \leq g^*(s) + h(s) - h(s')$, então as distâncias Euclideanas usadas para a heurística são consistentes, ou $h(s) \leq h(s') + c(s, a, s')$, e como $g^*(s') = g^*(s) + c(s, a, s')$ por definição do valor real de g, temos $g(s')) \leq g^*(s')$, como desejado.

O máximo dentre esse valores em geral é um valor suficientemente preciso para todo estado, sem ultrapassar o valor real. Vale notar que um filho de um estado da busca nunca será gerado para um triângulo em particular se o estado correspondente àquele triângulo já for um predecessor daquele estado. Essa exclusão nunca eliminará um caminho ótimo. Isso é importante pois, como existem múltiplos estados que correspondam a um mesmo triângulo, isso poderia levar a um espaço de busca infinito, por conta de ciclos no grafo. Essas eliminações reduzem o espaço de busca, evitando buscas desnecessárias.

3.3 Suavização do Caminho

3.3.1 O Algoritmo do Funil

O objetivo do *pathfinding* em uma triangulação é encontrar uma sequência de triângulos adjacentes de modo que o ponto de início esteja contido no primeiro triângulo e o destino esteja no último. Caso a sequência contenha duplicatas significa que o caminho encontrado possui um ciclo, e os triângulos entre tais pares podem ser removidos para encurtar o caminho. Assim, assume-se que uma sequência é um conjunto de triângulos distintos ordenados.

Para um caminho ser válido, o agente não deve sobrepor nenhum obstáculo quando posicionado nas posições inicial ou final. Pode-se supor que o ponto de partida é válido, dado que o agente já deve estar posicionado em uma posição desobstruída. Assumindo que exista um caminho válido entre as arestas de cada triângulo em tal sequência (ou seja, toda a área dos triângulos é válida para movimentação), segue-se que existe um caminho válido que atravessa todos estes triângulos. O caminho mais óbvio é através de triângulos adjacentes ligados por segmentos de reta. Além da validade dos pontos inicial e final, para um caminho ser válido o agente não deve sobrepor nenhuma aresta restrita enquanto se move entre esses pontos. Sendo assim, o comprimento das arestas deve ser de pelo menos do tamanho do diâmetro do agente para que esse caminho seja válido.

Dada a existência desse caminho, resta encontrar uma técnica para gerar o caminho de menor custo dentro desta sequência. Para isso, é necessário definir alguns elementos da sequência:

- Uma aresta interior é qualquer aresta irrestrita na triangulação que o agente vai cruzar quando estiver atravessando um caminho através desta sequência de triângulos. Ou seja, a aresta interior é compartilhada por dois triângulos adjacentes na sequência.
- Um triângulo interior é qualquer triângulo da sequência que não contenha os pontos inicial ou final. Isto é, todos os triângulos da sequência exceto o primeiro e último.
- Um canal é o polígono simples (formado pela sequência de triângulos) dentro do qual queremos encontrar um caminho válido para o agente. Os vértices do canal consistem nos pontos inicial e final, juntamente com os vértices de todos os triângulos interiores na sequência. As arestas deste polígono são as mesmas dos triângulos interiores exceto as arestas interiores. A Figura 3.25 mostra um ponto de início e destino, uma sequência de triângulos e o canal resultante. Este canal e as arestas interiores serão utilizados para encontrar o caminho mais curto entre os pontos

inicial e destino. Isto pode ser feito em tempo linear de acordo com o número de triângulos da sequência, usando o algoritmo do funil [7] [29].

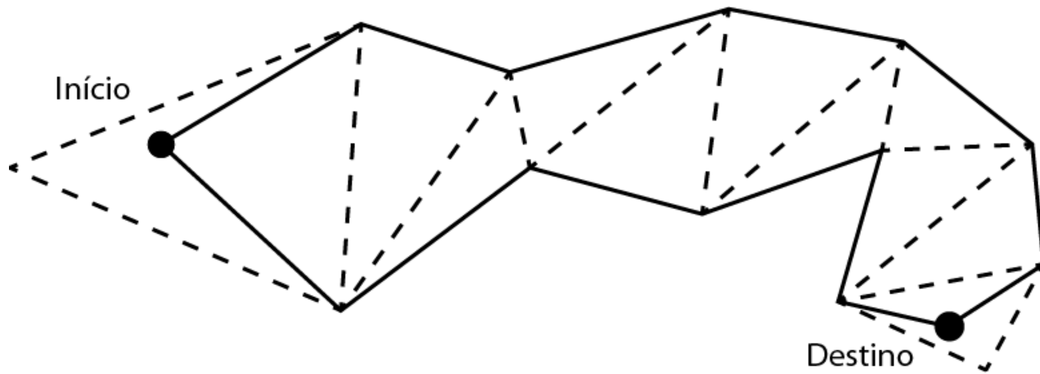


Figura 3.25: Uma sequência de triângulos, os pontos de início e destino e o canal formado por eles.

O algoritmo funil considera três estruturas: o caminho, o ápice e o funil. O caminho é a sequência de segmentos de reta que formam o trajeto mais curto conhecido no ponto atual no algoritmo. O funil é composto por duas sequências de segmentos de reta, uma girando no sentido horário e outra anti-horário, que representam a área onde estão todos os possíveis caminhos mais curtos para o destino que ainda não foram processados. Finalmente, o ápice é o ponto que une o caminho ao funil. A Figura 3.26 mostra essas estruturas.

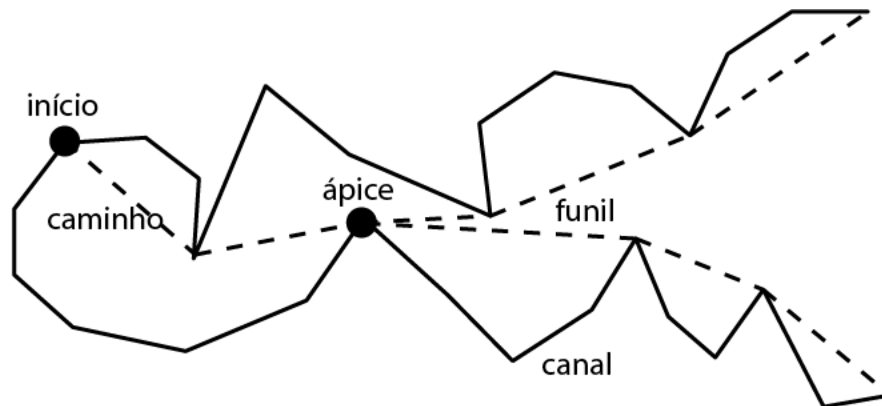


Figura 3.26: A rota, o ápice e o funil durante a execução do algoritmo.

No início do algoritmo o caminho está vazio, o ápice é definido como o ponto de partida, e o funil começa como os segmentos que ligam o ponto de partida na primeira aresta interior. O funil é armazenado em uma lista duplamente encadeada, pois a capacidade de percorrer essa lista em ambos os sentidos é muito explorada neste algoritmo. Cada uma das arestas interiores é processada por vez, e os vértices ainda não processados são adicionados na lista do lado correspondente no funil. Vértices são retirados daquele lado do funil até que o subfunil onde o vértice atual se encontra seja descoberto, quando então esse vértice é adicionado ao final da lista daquele lado do funil. Estes subfunis são ilustrados na Figura 3.27.

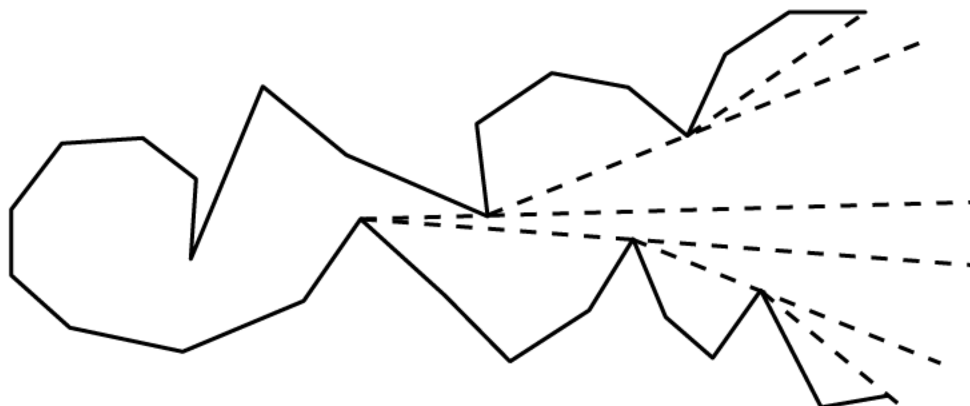


Figura 3.27: Subfunis correspondentes aos vértices do funil.

Se o ápice é retirado da lista desta forma, o próximo vértice passa a ser considerado como novo ápice, e um segmento ligando o ápice antigo ao novo é adicionado ao caminho. Uma vez que a aresta final interior do canal é adicionada ao ápice, nós adicionamos o ponto de destino no funil. Uma vez feito isso, o caminho é combinado com o lado certo do funil para formar o trajeto final entre o início e o destino. Este processo produz o caminho mais curto dentro do canal [17].

Segue um exemplo de como adicionar o mesmo vértice para cada lado da lista. Assumindo que o agente atravessasse a aresta superior do novo triângulo, o vértice seria adicionado ao lado direito, como mostrado na Figura 3.28. Aqui todos os vértices à direita daquele cujo subfunil contém o novo vértice, são removidos da lista, e o funil daquele lado é atualizado para incluir o segmento entre esses dois vértices (Figura 3.29).

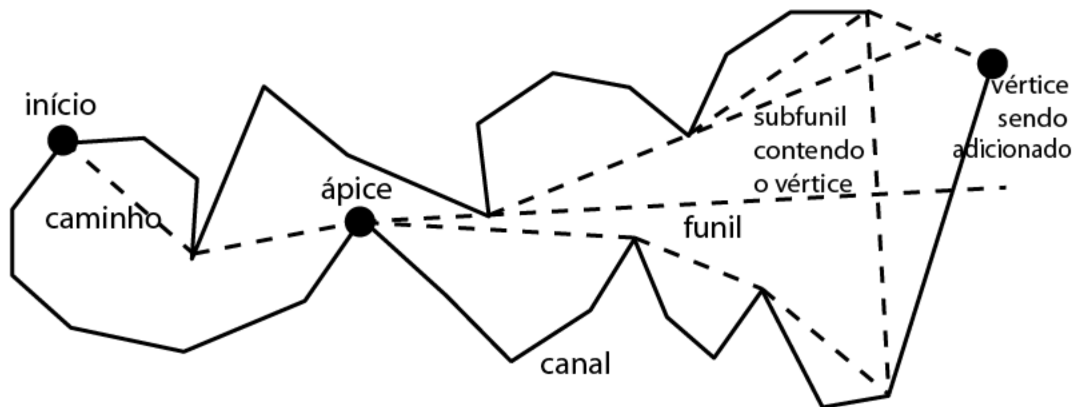


Figura 3.28: Adicionando um vértice no lado direito do funil.

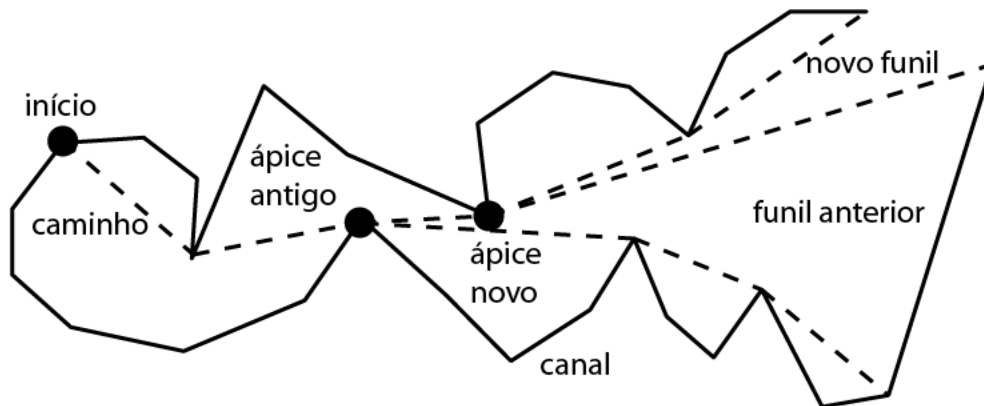


Figura 3.29: O funil formado após a adição de um novo vértice à direita.

Assumindo que o agente atravesse a aresta inferior do novo triângulo, o vértice deve ser adicionado ao lado esquerdo da lista, resultando na situação mostrada na Figura 3.30. Aqui todos os vértices do lado esquerdo da lista são retirados, e então o ápice é retirado, movendo o ápice um vértice à direita e estendendo o caminho para incluir o segmento entre o ápice antigo e o novo. Então chegamos ao vértice cujo subfunil contém o novo vértice. Paramos de remover vértices e substituímos o lado esquerdo do funil com o segmento entre o ápice e o novo vértice, como mostrado na Figura 3.31.

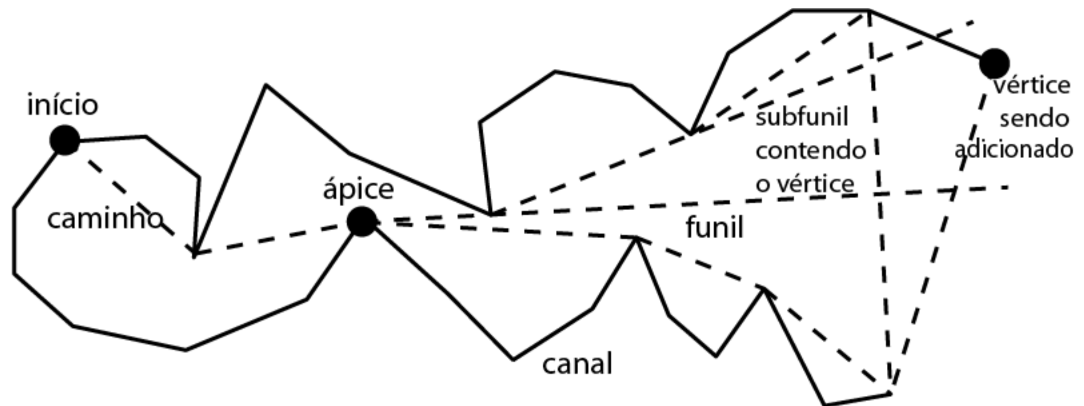


Figura 3.30: Adicionando um vértice no lado esquerdo do funil.

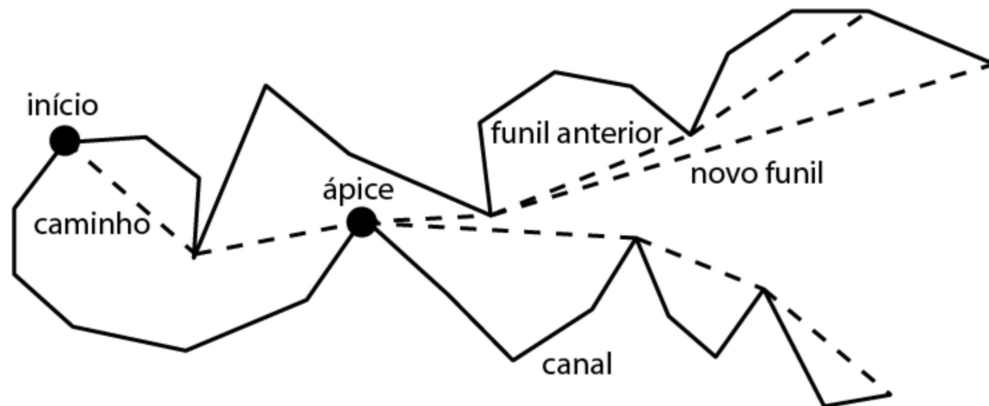


Figura 3.31: O funil formado após a adição de um novo vértice na esquerda.

Capítulo 4

Módulo *DeltaPath*

Este capítulo apresenta a implementação e os testes do módulo de planejamento de trajetória *DeltaPath*. Na Seção 4.1 serão dados detalhes de projeto e implementação do módulo, e na Seção 4.2 são apresentados os testes de desempenho do módulo e como foi identificada a necessidade de melhorar o sistema de planejamento de trajetória em um jogo RTS.

4.1 Implementação

A implementação do módulo de planejamento de trajetória *DeltaPath* foi dividida em três partes: triangulação, *pathfinding* e suavização, utilizando as técnicas demonstradas no Capítulo 3.

Em termos de arquitetura, o objetivo do *DeltaPath* é disponibilizar uma interface simples para o planejamento de caminho. A ideia é que uma única chamada de função com posição inicial e final como argumentos, depois de inicializado o módulo com a descrição do ambiente, seja toda a informação necessária para o planejador. Sendo assim foi desenvolvido um projeto de arquitetura, que está resumido na Figura 4.1.

Internamente, o *DeltaPath* foi dividido em 3 submódulos, sendo eles Triangulação, Malha de navegação (*navmesh*) e Funil. Cada um desses é composto por uma classe principal e várias classes auxiliares. Esses módulos podem operar juntos, de modo que as interfaces internas tornam-se completamente transparentes, ou separados, para que o desenvolvedor possa acessar essas interfaces diretamente se desejar.

A estrutura do módulo foi feita desta maneira com o objetivo de facilitar o acesso a membros internos da triangulação e malha de navegação. Isso é bom para o desenvolvedor, pois muitas técnicas, como comportamento de movimentação, colisão, cálculo de linha de visão e várias outras, precisam acessar diretamente a triangulação e/ou a malha de navegação. Do mesmo modo, caso esse acesso não seja de interesse do usuário, o mesmo pode simplesmente usufruir de toda a capacidade do planejador com apenas dois métodos: um para inicializar o ambiente e outro para calcular o caminho.

A linguagem utilizada na implementação foi C++ devido à sua eficiência e por ser a linguagem mais utilizada em jogos comerciais no mundo. Como um dos objetivos do módulo é a generalidade, para que possa ser utilizado em várias aplicações diferentes, o desenvolvimento do módulo teve como um de seus objetivos deixar o código bem modularizado, respeitando os padrões do paradigma orientado a objetos.

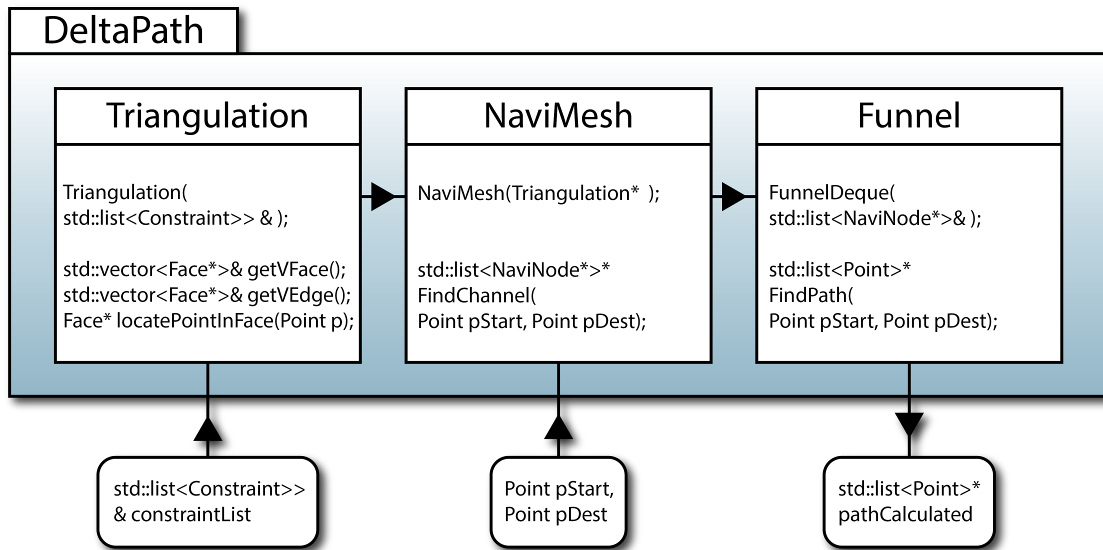


Figura 4.1: Diagrama que mostra as interfaces internas e externa do *DeltaPath*.

Outro requisito respeitado na implementação foi a eficiência do módulo. Sempre que possível foram aplicadas simplificações matemáticas aos cálculos espaciais e também foram utilizadas estruturas de dados simples para representar as diversas partes que compõe a malha. Otimizações matemáticas também foram feitas no algoritmo do funil. A grande maioria dessas melhorias baseou-se na implementação do planejador de trajetória do motor de jogos ORTS [6];

O triangulador é a maior parte do módulo e também a mais complexa. Sua entrada é um conjunto de restrições e sua saída são vários métodos de acesso a uma triangulação restrita de Delaunay. Para usá-lo, basta chamar seu construtor passando a lista de restrições desejada. A partir daí o ambiente será particionado e será possível navegar na malha triangular. Os métodos de inicialização e acesso à triangulação são demonstrados no Código 4.1.

```
1 // Constructor
2 Triangulation(std::list<Constraint>>& constraintList);
3
4 // Acessors
5 std::vector<Face*>& getVFace();
6 std::vector<Face*>& getVEdge();
7 Face* locatePointInFace(Point p);
```

Código 4.1: Triangulação - Inicialização e Uso.

O *pathfinder* é construído passando-se para ele uma triangulação. Usando essa triangulação, é construído um grafo de navegação entre triângulos, chamado de malha de navegação. Esse grafo é então usado para fazer buscas de caminhos entre triângulos através do algoritmo A*. Essa busca retorna uma lista ordenada de nós (que representa um canal de triângulos). Os métodos de inicialização e cálculo de caminho são demonstrados no Código 4.2.

```
// Constructor
NavMesh(Triangulation *triangulation);

// Usage
std::list<NavNode*>* FindChannel(Point pStart, Point pDest);
```
Código 4.2: NavMesh - Inicialização e Uso.

Depois de calculada a lista de triângulos que contém o menor caminho é necessário um algoritmo de suavização para definir, dentro do canal, qual o melhor caminho a ser utilizado. O suavizador recebe o canal, o ponto inicial e o destino, e sua saída é uma lista ordenada de pontos que representam o menor caminho dentro daquele canal. Os métodos de inicialização e de suavização de caminho são demonstrados no Código 4.3.

```
// Constructor
FunnelDeque(std::list<NaviNode*>&);

// Usage
std::list<Point>* FindPath(Point pStart, Point pDest);
```
Código 4.3: FunnelDeque - Inicialização e Uso.

Por fim, através do uso dos três submódulos juntos, é possível encapsular alguns métodos de inicialização e uso dentro do módulo da malha de navegação para que seja necessário apenas uma chamada de método para inicializar e usar o módulo *DeltaPath*. Os métodos finais de inicialização e cálculo do menor caminho pelo módulo *DeltaPath* são demonstrados no Código 4.4.

```
// Constructor
NavMesh(std::list<Constraint>>& constraintList);

// Usage
std::list<Point>* FindPath(Point pStart, Point pDest);
```
Código 4.4: DeltaPath - Inicialização e Uso.

O resultado produzido ao final é uma lista ordenada de pontos, que representam o menor caminho do ponto inicial ao objetivo dividido em segmentos de reta. Um agente deve ser capaz de, utilizando esta lista, mover-se de seu local inicial até seu objetivo. O modo como essa movimentação é feita, ou seja, a atualização da posição deste agente, não é responsabilidade do módulo *DeltaPath*.

4.2 Testes e Resultados

Identificando problemas

Durante o desenvolvimento de um jogo RTS, denominado *Smile Wars*, foi identificado um possível problema com o algoritmo de planejamento de rota. Esse procedimento

parecia demandar muito processamento, pois, ao executar diversas buscas simultâneas (problema identificado para mais de 50 unidades planejando trajetória ao mesmo tempo), o programa parecia parar de responder. Foi teorizado que a causa disso era o módulo de planejamento de rota, que, por usar uma técnica de particionamento do espaço muito simples (grade), acabava gerando um grafo com uma quantidade muito grande de nós, o que aumenta muito o custo da busca. Também por causa do tipo de particionamento, a movimentação das unidades no ambiente não era fluida, pois as mesmas tinham suas possíveis posições restritas pela grade usada para representar o ambiente. Outro problema identificado é que todas as colisões entre agentes no ambiente do jogo eram tratadas através de recálculo da rota, o que poderia fazer com que ainda mais requisições de busca de caminho fossem efetuadas.

Para comprovar a existência desses problemas foi usada uma técnica de *profiling*. A ferramenta escolhida para isso foi o Callgrind[1] , disponível dentro do conhecido *framework* de análise dinâmica de software Valgrind[2]. O Callgrind é capaz de registrar o histórico de chamadas entre funções de execução de um programa em um gráfico de chamadas. Por padrão, os dados coletados consistem no número de instruções executadas, sua relação com as linhas de código, a relação de chamadas/chamador entre as funções e a quantidade de tais chamadas. No término da execução, esses dados são gravados em um arquivo. Para a visualização dos dados, foi usado o Kcachegrind[3] , que é uma interface gráfica que torna fácil navegar na grande quantidade de dados que o Callgrind produz.

A versão do *Smile Wars* escolhida para executar esses testes de desempenho foi a número 184, por ser a versão mais estável disponível, com algumas pequenas modificações para facilitar o teste, sendo elas: criação automática de unidades e envio automático de pedidos de recálculo de rota para todas as unidades em todos os ciclos de processamento após todas as unidades estarem criadas. O tempo gasto em cada fase do teste também foi medido.

O teste procedeu da maneira relatada a seguir.

Fase do teste associada ao tempo decorrido (em minutos:segundos):

00:00 O jogo é iniciado dentro do ambiente Valgrind/Callgrind.

00:03 Interface gráfica do jogo é mostrada na tela inicial de carga.

00:59 A carga de todos os artefatos do jogo é finalizada, o ambiente de jogo é criado e o jogo inicia. A produção de 50 unidades é iniciada.

04:06 A produção de 50 unidades é concluída e iniciam-se os pedidos de cálculo de trajetória.

10:06 Após 6 minutos, os testes são finalizados e é enviado sinal para finalizar o programa.

10:30 Por estar preso em outros processamentos, o programa só finaliza vários segundos depois.

Os resultados produzidos por este teste podem ser vistos na Figura 4.2.

[1] http://valgrind.org/docs/manual/cl-manual.html
[2] http://valgrind.org/
[3] http://docs.kde.org/stable/en/kdesdk/kcachegrind/

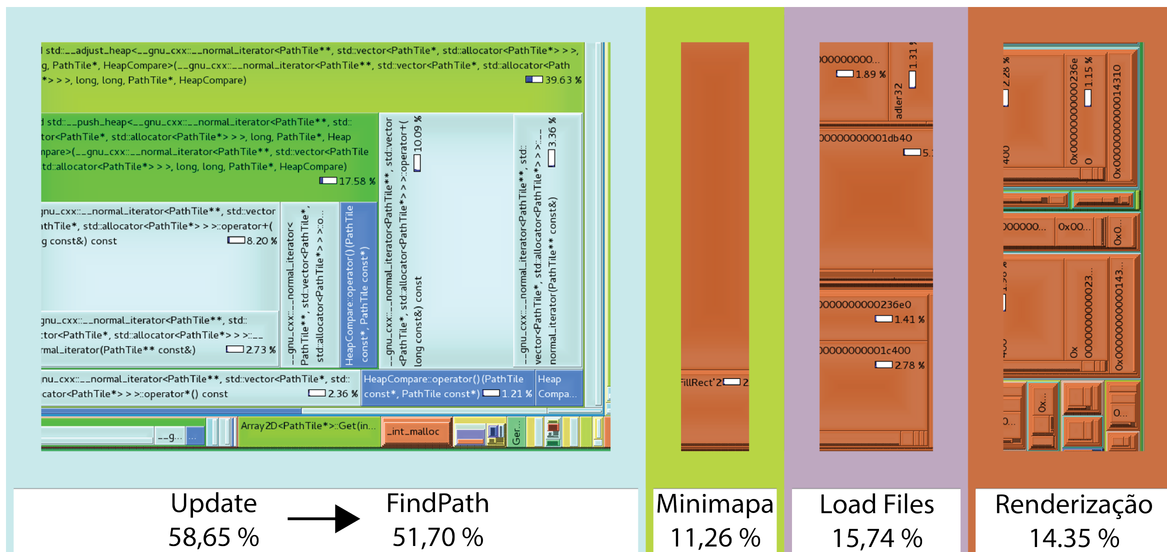


Figura 4.2: Representação da divisão do processamento na versão 184 do jogo em seus métodos e funções.

A análise dos resultados mostra que, descartadas as fases de carga do jogo, o planejamento de rota foi responsável por 51,70% do tempo de processamento do jogo, confirmando a hipótese de que isso era o gargalo do processamento e era o que causava problemas de desempenho no jogo.

Resultados

Após a integração do módulo *DeltaPath* no *Smile Wars*, foi feito um teste para comparar os resultados obtidos com o novo sistema de cálculo de rota.

A versão escolhida para executar os testes de desempenho foi a 444, por ser a versão mais estável disponível (após a integração), com algumas pequenas modificações para facilitar o teste, sendo elas: criação automática de unidades e envio automático de pedidos de recálculo de rota para todas as unidades em todos os ciclos de processamento após todas as unidades estarem criadas. O tempo gasto em cada fase do teste também foi medido.

O teste procedeu da maneira relatada a seguir.

Fase do teste associada ao tempo decorrido (em minutos:segundos):

00:00 O jogo é iniciado dentro do ambiente Valgrind/Callgrind.

00:31 Interface gráfica do jogo é mostrada na tela inicial de carga.

03:11 A carga de todos os artefatos do jogo é finalizada, o ambiente de jogo é criado e o jogo inicia. A produção de 50 unidades é iniciada.

04:07 A produção de 50 unidades é concluída e iniciam-se os pedidos de cálculo de trajetória.

10:07 Após 6 minutos, os testes são finalizados e é enviado sinal para finalizar o programa.

10:10 O programa finaliza alguns segundos depois.

Os resultados produzidos por este teste podem ser vistos na Figura 4.3. Esses resultados mostram que o tempo dedicado ao processamento do jogo foi dividido em 3 grandes grupos: inicialização (38,78%), atualização (11,87 %) e renderização (49,35 %). O cálculo de trajetória é feito dentro da atualização do jogo, por isso é necessária uma análise somente do método de atualização do jogo, que pode ser vista com mais detalhes na Figura 4.4.

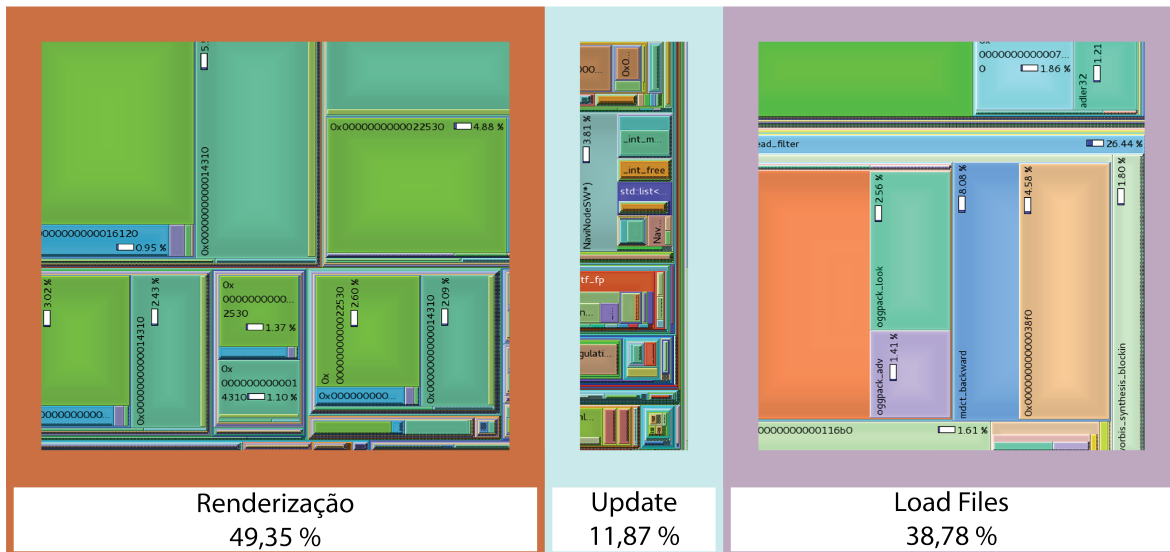


Figura 4.3: Representação da divisão do processamento na versão revisada do jogo em seus métodos e funções.

Por fim, o planejamento de rota foi responsável por 10,27% do tempo de processador do jogo, mostrando ser cinco vezes mais rápido que o modelo antigo. Nessa porcentagem está incluso o tempo de processamento gasto pela técnica de *steering behaviors*, que foi 1,81%. Com a implementação dessa técnica, foi possível evitar todos os recálculos de rota que eram necessários ao se usar o particionamento do espaço em grade devido a colisões e bloqueios por outras unidades. Também é notável a melhoria na fluidez da movimentação das unidades, uma vez que nesse novo modelo não há restrição no posicionamento das mesmas e, por causa das técnicas de *steering behaviors*, a movimentação de uma unidade afeta as outras à sua volta, ou seja, as unidades podem se empurrar.

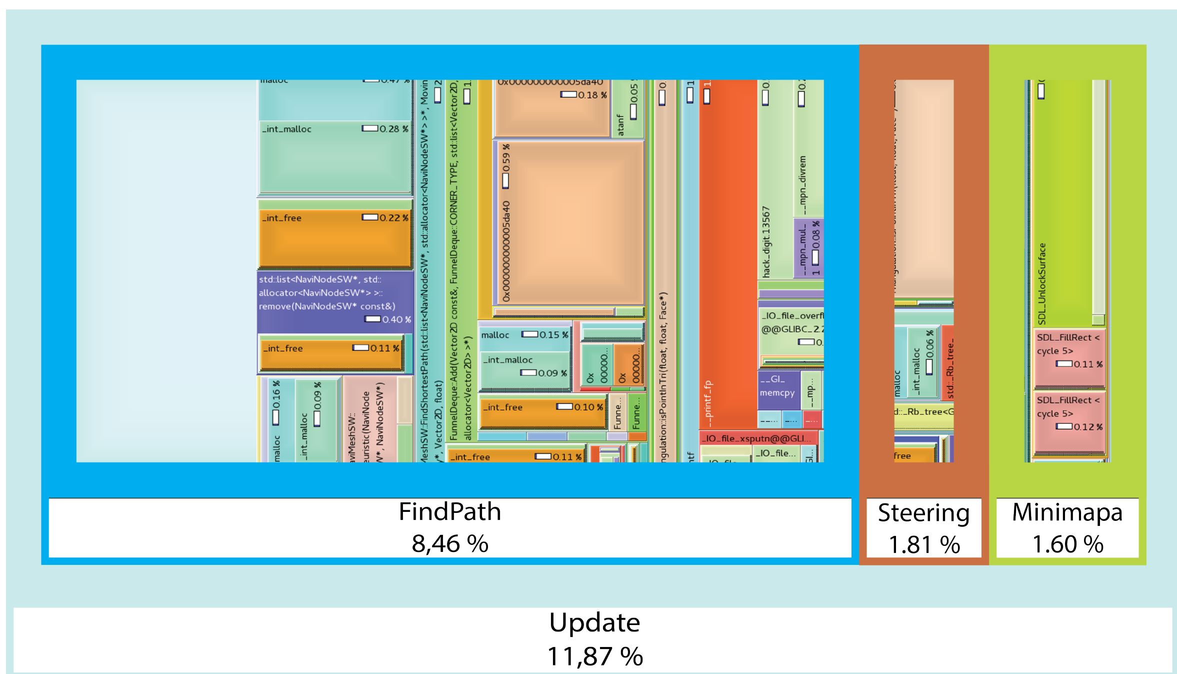


Figura 4.4: Representação da divisão do processamento no método de atualização.

Capítulo 5

Aplicações

O módulo *DeltaPath* foi utilizado em duas aplicações diferentes com o objetivo de avaliar seu desempenho em situações reais e distintas: no jogo *Smile Wars* e no motor de jogos *Gear2D* [12]. A aplicação no *Smile Wars* teve como objetivo testar a eficiência e facilidade de integração do módulo, enquanto na *Gear2D* o foco foi testar a generalização e a usabilidade.

5.1 *Smile Wars*

Smile Wars é um jogo de estratégia em tempo real cuja produção iniciou em agosto de 2012 por dois alunos do curso de Ciência da Computação da Universidade de Brasília, Luiggi Reffatti (autor desta monografia) e Eduardo Freire. O jogo encontra-se disponível para download gratuito[1].

Nesse gênero de jogo, o jogador controla várias unidades e o objetivo é usar essas unidades para atacar e destruir os adversários. No *Smile Wars*, por uma questão de jogabilidade e desempenho da renderização, este número é limitado em cinquenta unidades por jogador e, no máximo, trezentas unidades simultâneas pois o número de jogadores é limitado em seis (por questões de jogabilidade). O jogo foi projetado e desenvolvido para ser um jogo casual, ou seja, fácil de aprender, com partidas rápidas e controles simples. O público alvo é o grupo dos jogadores casuais em geral, mas a temática do jogo é direcionada para crianças, utilizando cores vivas e imagens simples.

Apesar de fácil de aprender, o jogo implementa uma mecânica complexa de vantagens e desvantagens entre as diferentes unidades. Cada unidade é composta de cinco partes, sendo elas corpo, pés, mãos, cabeça e especial. Essas partes podem ser básicas ou possuir um elemento entre normal, fogo, ar, água e terra. Cada parte concede às unidades diferentes habilidades, que são movimentação, ataque, vida, dano, visão e alcance, além de tornar a unidade mais forte ou mais vulnerável a outra dependendo de sua composição elemental (isto é, a porcentagem da presença de cada elemento na unidade). Considerando todas as possibilidades, é possível criar mais de cento e quarenta mil unidades diferentes, com atributos distintos, o que torna complexa a decisão de qual unidade criar em um determinado momento. A Figura 5.1 mostra uma captura de tela de uma partida no *Smile Wars*.

[1] http://smilewars.deltacreatures.net/download/

Figura 5.1: Captura de tela do *Smile Wars*.

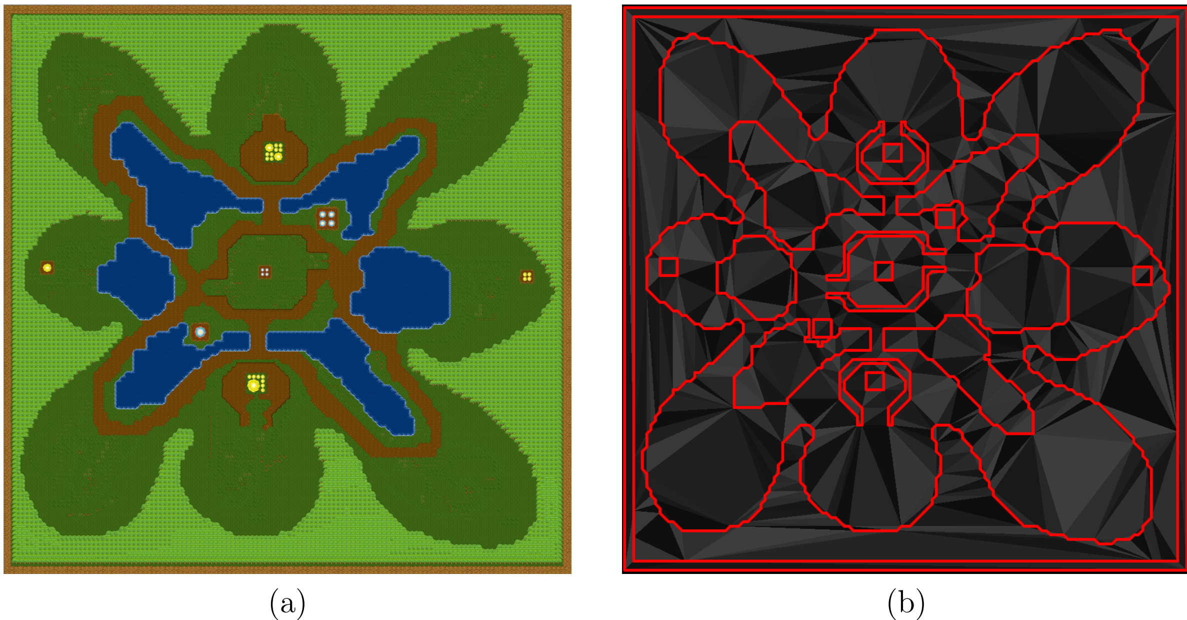
Aplicação do *DeltaPath*

O *Smile Wars* até sua revisão 184 utilizava um modelo de particionamento espacial em grade, escolhido devido à facilidade de implementação e ao fato dos ambientes do jogo serem definidos por um (*tilemap*). Um *tilemap* consiste em um *grid* que armazena, para cada célula, informações utilizadas para renderização, visibilidade, restrições de movimentação, etc. Após detectar os problemas da divisão do espaço em grade (tempo demorado de processamento e falta de fluidez na movimentação das unidades), a solução proposta foi utilizar o módulo *DeltaPath* para realizar o planejamento de trajetória.

Algumas adaptações tiveram que ser feitas para a aplicação do módulo, pois devido ao ambiente ser definido por *tilemaps* no *Smile Wars*, as restrições a serem passadas para a triangulação deveriam ser geradas a partir desses mapas.

Para realizar essa conversão, foi desenvolvido um vetorizador de mapas, ou seja, uma ferramenta capaz de criar linhas e polígonos representando as restrições de movimentação descritas por um *tilemap*. Resumidamente, o vetorizador funciona da seguinte maneira. Sua entrada é um *tilemap* e um intervalo de valores de restrições de movimentação. A grade do *tilemap* é então vasculhada, célula a célula, em busca de células que ainda não tenham sido percorridas. Inicialmente, nenhuma das células está marcada como percorrida. Ao encontrar uma célula não marcada, caso seu valor esteja fora do intervalo, é aplicada a técnica de enchente (*flood fill*) [11] para marcar como visitadas todos as células

pertencentes à mesma região (com valores de restrição fora do intervalo procurado). Ao encontrar uma célula não marcada cujo valor está dentro do intervalo, é aplicada a técnica de seguir paredes (*wall following*) [3] com algumas regras de vetorização para gerar um polígono representando a restrição encontrada. Após isso, é aplicada a técnica de enchente para marcar todas as células pertencentes a esta restrição (com valores de restrição dentro do intervalo procurado). O vetorizador para após percorrer todo o mapa e retorna uma lista de polígonos representando as células do *tilemap* (*tiles*) no intervalo desejado. Essa lista é usada para construir a triangulação, que pode ser vista na Figura 5.2.



(a) (b)

Figura 5.2: (a) Um mapa que define um ambiente no *Smile Wars* e (b) sua triangulação.

Também foi necessário implementar um modo de tratar as colisões entre unidades, dado que o novo planejamento de trajetória não é capaz de tratar esses casos. Fez-se necessária a implementação de uma técnica de *steering behaviors* [28], que calcula qual é o próximo passo de cada unidade baseada na sua trajetória e nas interações com outras unidades e com o ambiente à sua volta. Assim o problema da colisão dinâmica é resolvido e a movimentação das unidades torna-se mais realista e fluida.

A recepção do novo comportamento da movimentação das unidades pelos jogadores foi muito boa, pois, além de ter resolvido o problema de desempenho do jogo, o deixou mais agradável para jogar. Alguns outros efeitos não previstos da troca do modelo de movimentação foram notados, como por exemplo: unidades não ficam mais presas por outras unidades de um mesmo jogador, pois elas podem agora se empurrar. Algumas estratégias que baseiam-se na mobilidade e velocidade das unidades também ficaram melhores.

Até a data de publicação deste trabalho, o jogo continua em desenvolvimento, por uma equipe maior (sete integrantes, cujos nomes estão são citados nos Agradecimentos) que constitui um estúdio de desenvolvimento independente chamado DeltaCreatures Produções[2].

[2] http://deltacreatures.net/

O *Smile Wars* foi eleito, dentre mais de 150 jogos concorrentes, por votação popular o 2º melhor jogo no festival de jogos independentes do XI Simpósio Brasileiro de Jogos e Entretenimento Digital[3] (SBGames 2012).

5.2 Gear2D

A *Gear2D* [13], desenvolvida e mantida pelo aluno Leonardo de Freitas do curso de Ciência da Computação da Universidade de Brasília, é um motor de jogos que objetiva promover uma arquitetura baseada em componentes para a criação de jogos eletrônicos, com foco em flexibilidade e adaptação de entidades em tempo de execução. Tais objetivos foram alcançados com o uso extensivo de alguns padrões de projeto da *Gang of Four* [14] e alguns menos conhecidos na literatura. Por exemplo, para contornar os problemas ocasionados por longas hierarquias de entidades, o padrão *Composite* foi adotado pelo motor de jogos. Nesse cenário, personagens (e outras entidades do jogo) exercem o papel de *containers* para diversos componentes, permitindo que as características de uma entidade sejam atribuídas por meio de agregação.

Para evitar a necessidade da referência direta a atributos e métodos entre componentes, no intuito de promover reuso e evitar acoplamento, a metáfora do quadro-negro [8] foi implementada: componentes operam sobre uma tabela de parâmetros e atributos compartilhada entre eles para produzir um ou mais resultados desejados, sem a necessidade de fazer referência direta. A *Gear2D* utiliza a tabela de parâmetros não apenas para armazenar valores relevantes, mas também para a comunicação entre os componentes através do mecanismo *signal and slots*. Cada parâmetro aceita *listeners*, que são notificados para cada nova alteração realizada num parâmetro que lhes interessa, provendo meios para a realização de chamadas de métodos.

Em resumo, as características implementadas na *Gear2D* são:

- Carregar arquivos de configuração para a disposição inicial dos objetos e componentes da aplicação.
- Permitir a inserção e remoção de componentes em tempo de execução.
- Garantir que cada componente obtenha uma fatia de tempo para seu próprio processamento.
- Permitir a comunicação entre os componentes, disponibilizando para isso uma tabela extenível de parâmetros e atributos.
- Carregar novas entidades em tempo de execução através de uma API disponibilizada.

Utilizando tais características do *framework* é possível criar componentes que, juntos, completam um motor de jogos e, através do uso destes, é possível construir um jogo eletrônico. Atualmente existem componentes para posicionamento espacial, execução de áudio, renderização 2D, entrada via teclado, detecção de colisões, cinemática e dinâmica de corpo rígido, entre outros. Tanto a *Gear2D* quanto seus componentes estão disponíveis gratuitamente [4].

[3] http://www.sbgames2012.com.br/
[4] http://gear2d.com/

Apesar de contar com um bom conjunto inicial de componentes, a *Gear2D* ainda precisa de muitos mais para se tornar um motor atrativo para os desenvolvedores e pesquisadores. Por outro lado, devido à sua arquitetura genérica, ela é excelente para testar a capacidade de generalização e facilidade de utilização do *DeltaPath*. Por isso, com o intuito de testar essas características do módulo e adicionar à biblioteca da *Gear2D* um componente, não trivial, de planejamento de trajetória, que é estritamente necessário no desenvolvimento de jogos de determinados gêneros, foi implementado o componente *Pathfinder2D*.

Pathfinder2D

A arquitetura da *Gear2D* faz com que seja muito simples a criação e integração de componentes. O *Pathfinder2D* é capaz de encontrar o caminho mais curto entre dois pontos no ambiente de jogo de maneira genérica, para se adequar a uma grande diversidade de cenários de jogos. É também eficiente, para não causar impacto significativo no desempenho das aplicações feitas com o motor, mesmo com um número elevado de personagens e com cenários de tamanhos e complexidades variados. Na Figura 5.3 temos um cenário no formato da logo da *Gear2D* triangulado utilizando-se o *DeltaPath*.

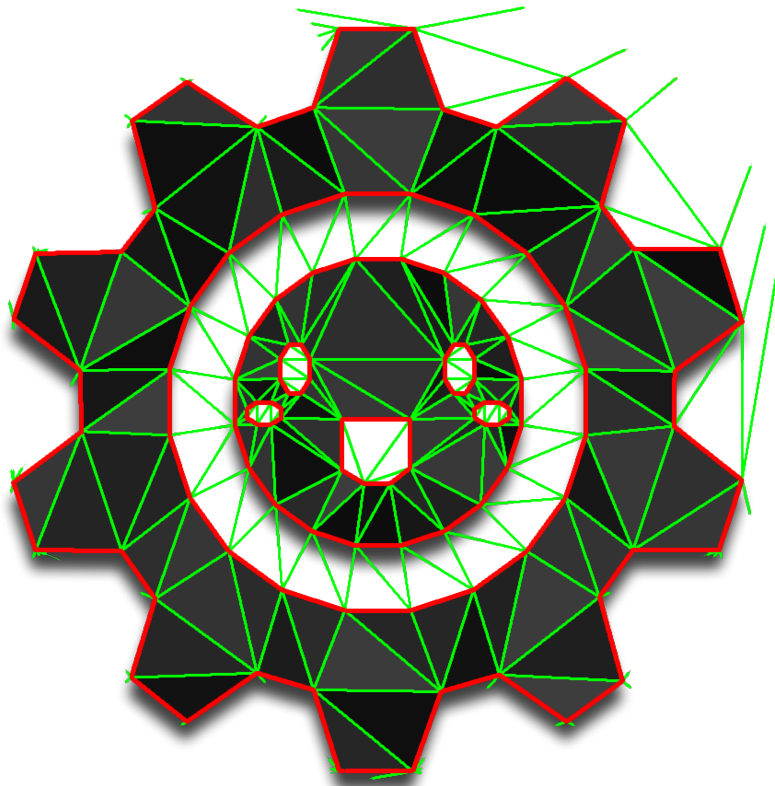


Figura 5.3: Logo da *Gear2D* triangulado através do *DeltaPath*

Na *Gear2D*, o ambiente é representado através de um espaço bidimensional contínuo, então o componente deve encontrar a trajetória partindo da posição atual da entidade (x_1, y_1) até o seu destino (x_2, y_2). Tais parâmetros são facilmente obtidos através da tabela de parâmetros, fornecidos por um componente espacial já existente chamado *space2d*. A trajetória resultante é representada em uma sequência de pontos (x, y) que a entidade pode usar para se locomover. Essa informação é disponibilizada para outros componentes na mesma tabela de parâmetros.

A fim de mostrar a funcionalidade do componente *Pathfinder2D* foi criado um pequeno jogo de demonstração denominado Katch-Up. O jogo é muito simples e envolve a movimentação de uma unidade em um ambiente retangular com vários obstáculos quadrados, como pode ser visto na Figura 5.4. Essa unidade deve se movimentar para a posição onde o usuário clicar na tela através do menor caminho e sem colidir com os obstáculos.

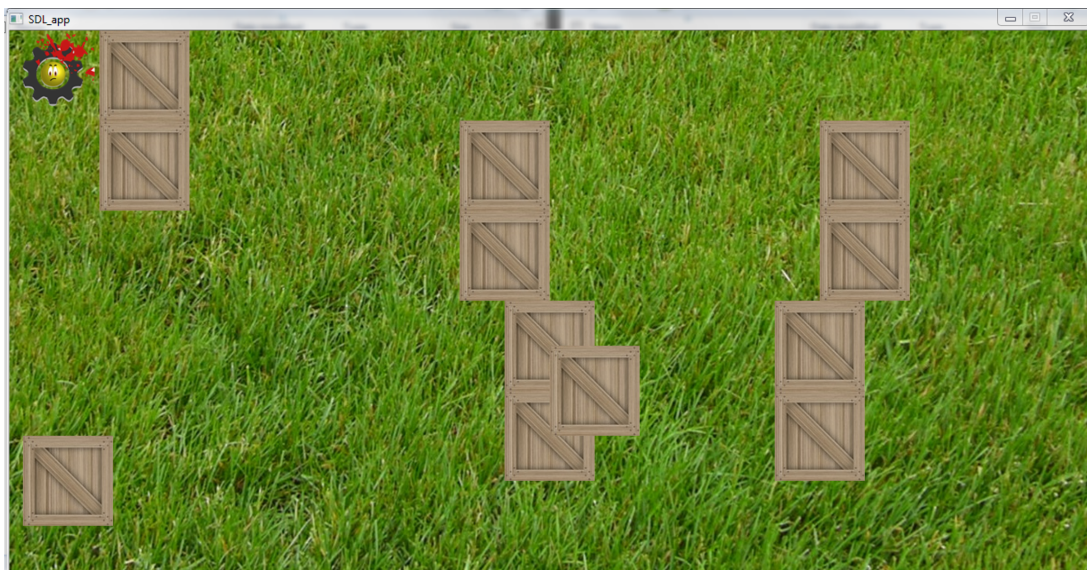


Figura 5.4: Katch-Up - demonstração da capacidade do componente *Pathfinder2D*.

Para desenvolver esse jogo foi necessário implementar algumas funcionalidades, como tratamento de eventos de entrada e movimentação de unidades, que não são responsabilidades do componente *Pathfinder2D*. Com o intuito de manter o componente responsável apenas por encontrar e retornar um caminho dois outros componentes foram criados para implementar essas funcionalidades: *Commander* e *Follower2D*.

O componente *Commander* age como uma ponte entre o usuário (usando o componente *mouse*) e as entidades do jogo, permitindo que o jogador dite as ações da unidade. Ele lida apenas com o comando 'mover para (x, y)', indicando o destino final da unidade. Esse comando é um dos mais usados em jogos do gênero estratégia, onde o usuário pode controlar múltiplas unidades e ordenar diferentes comandos a elas.

Uma vez que a trajetória é definida, o componente *Follower2D* especifica como a unidade vai se mover entre os pontos. Esse componente é responsável por alcançar a coordenada final, que inclui evitar colisões com objetos dinâmicos que podem estar no caminho. Idealmente, a rota tomada deve ser uma linha reta entre cada par de pontos, mas a trajetória pode se tornar mais complexa para evitar objetos que se movem dentro do mesmo triângulo, usando por exemplo técnicas de *steering behaviors* [28].

Os componentes da *Gear2D* se comunicam livremente uns com os outros através da tabela de parâmetros, enquanto a implementação do padrão *Observer* via métodos *hook* e *handle* permite que eles interajam. Isso, junto com a relação direta entre o espaço contínuo do componente *space2d* e a implementação do *Pathfinder2D*, permitiu que esses três componentes comunicassem entre si e com o motor. Além disso, graças ao desacoplamento garantido pela *Gear2D*, é permitido que os componentes *Follower2D* e *Commander* funcionem facilmente com outras implementações de busca por trajetória, bem como permite que o *Pathfinder2D* funcione com outros componentes de interface e movimentação.

O trabalho de criação da *Gear2D*, junto com a produção dos componentes *Pathfinder2D* e *LuaProxy*, resultaram em um artigo apresentado na conferência internacional *Foundations of Digital Games* [5] , em junho 2012, Raleigh, EUA, e publicado na *Association for Computing Machinery* [6] (ACM), intitulado: "*Gear2D: an extensible component-based game engine*" [13].

[5] http://www.foundationsofdigitalgames.org/

[6] http://dl.acm.org/citation.cfm?id=2282338.2282357

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Conclusões

Não foi encontrada uma solução com código aberto, eficiente, simples e genérica para resolver o problema do planejamento de trajetória. Como solução, foi implementado um módulo chamado *DeltaPath*, com todas as características desejadas, e esta implementação tem seu código disponível em `http://code.google.com/p/delta-creatures-pf2d/` . Isso foi possível através da combinação de três técnicas com propósitos diferentes: triangulação restrita de Delaunay, para particionar e representar o ambiente, o algoritmo A^*, para fazer a busca pela trajetória, e o algoritmo do funil, para suavizar esse caminho.

Os objetivos iniciais propostos para o *DeltaPath* foram atingidos. A saber: generalização, para que o módulo pudesse ser aplicado em vários contextos diferentes, eficiência, para que não houvesse grande impacto ao integrar às aplicações que vão utilizá-lo e simplicidade na sua interface para facilitar e encorajar seu uso. Esses resultados foram comprovados através de aplicações práticas do módulo em situações reais de uso.

Representação espacial

É possível ver as vantagens de usar representações poligonais para particionar e discretizar o espaço. Esse tipo de representação reduz o espaço de busca de forma significativa, especialmente quando o ambiente é composto de obstáculos formados por segmentos de reta, e mais ainda quando estes não são alinhados ao eixo, o que faz representações que usam divisões fixas, como grades, ficarem com imperfeições na representação ou aumentarem sua resolução. Mesmo quando o ambiente é definido por *tilemap*, onde a situação é propícia para trabalhar com buscas de trajetória baseadas em grade, representações poligonais geram um espaço de estados menor mantendo a representação original do ambiente. Os benefícios desse tipo de representação podem ser vistos nos resultados das aplicações do módulo *DeltaPath*.

Aplicações

Como ficou claro nos resultados, a utilização do módulo *DeltaPath* no jogo *Smile Wars* melhorou muito o desempenho do jogo, demonstrando a eficiência do módulo. Vale notar que, apesar de fortemente atrelado ao modelo de *grid*, a troca dos modelos de particio-

namento e *pathfinding* aconteceram sem maiores problemas, o que mostra a simplicidade da interface do *DeltaPath*.

A aplicação do módulo no motor de jogos *Gear2D* foi útil para mostrar que a implementação feita é desacoplada e genérica o suficiente para se adequar às diversas possíveis situações onde o componente *Pathfinder2D*poderá ser usado, ou seja, jogos e aplicações completamente diferentes umas das outras. O desenvolvimento desse componente também foi bem rápido, por causa das interfaces simples tanto da *Gear2D* quanto do *DeltaPath*.

6.2 Trabalhos Futuros

Algumas sugestões de trabalhos futuros surgiram ao longo do desenvolvimento deste. Essas incluem: a utilização de um nível maior de abstração para o grafo de busca, o aprimoramento das buscas por pontos na triangulação, a integração ao *DeltaPath* de um sistema de vetorização de *tilemap* e a criação de um componente de comportamento de movimentação para a *Gear2D*.

Abstrações

A forma homogênea que a triangulação representa o ambiente permite que o processo de abstração [10] seja executado nele. Essa técnica propõe a criação de um grafo de abstração um nível acima, através da identificação de becos sem saída, corredores, pontos de decisão, árvores, anéis e outras estruturas de grafos. Criado esse grafo, as buscas por trajetória são efetuadas primeiramente nesse nível de abstração e então no grafo de nível mais baixo, se necessário. Desse modo, a tarefa de *pathfinding* é simplificada para determinar qual é o melhor lado para contornar um obstáculo.

Essa abstração permite identificar várias situações em que uma trajetória pode ser calculada sem que seja preciso nenhuma busca, e quando a busca é necessária, o espaço de estados da mesma diminui consideravelmente. A principal vantagem desta abordagem é que o *pathfinding* não só independe do tamanho e da orientação das áreas, mas também das propriedades individuais dos agentes.

Localização de pontos

A parte que mais demanda recursos do *pathfinding* mostrou ser a localização de pontos na triangulação. Isso acontece pois o número de buscas necessárias é muito grande, a cada busca é necessário procurar os triângulos que contém os pontos inicial e objetivo de um caminho. Além disso, a localização de pontos também é utilizada em técnicas de *steering behaviors*. Para resolver isso é necessária a implementação de uma técnica melhor de localização de pontos.

Uma possibilidade é a divisão do ambiente em setores retangulares com pontos médios definidos [10]. Ao efetuar qualquer alteração na triangulação, o algoritmo deve conferir se o ponto médio de algum dos setores está sobre algum dos triângulos alterados, e, caso isso tenha acontecido, esse triângulo deve ser associado ao setor correspondente.

Ao executar a localização de triângulos, primeiramente é determinado qual ponto médio de qual setor está mais próximo ao ponto desejado, e a partir daí pode ser feita uma busca local na malha a partir do triangulo associado àquele setor.

Vetorização de *tilemaps*

Ao integrar o *DeltaPath* ao jogo *Smile Wars*, foi necessário implementar um módulo capaz de converter os obstáculos descritos em um *tilemap* e transformá-los em conjuntos de segmentos de reta representando restrições. A solução para esse problema, à primeira vista, parece muito simples, dado que o ambiente é especificado por uma grade quadriculada, cada *tile* restrito pode ser representado como um pequeno quadradinho restrito. Esse método, porém, não gera bons resultados, pois há uma fragmentação muito grande das restrições na malha além da criação de diversos pequenos triângulos nas bordas dos obstáculos do ambiente.

A implementação do módulo de vetorização de *tilemap* para gerar as restrições necessárias ao *DeltaPath* efetuada no *Smile Wars* utilizou a técnica de seguir paredes (*wall following*), associada à técnica de enchente (*flood fill*) para gerar restrições fiéis ao ambiente, suaves, sem serrilhamento nas diagonais e com o mínimo de pontos e segmentos de reta possíveis.

Essa solução pode ser generalizada e a partir dela pode ser criado um novo módulo, ou mesmo uma extensão do *DeltaPath* para facilitar a integração com jogos que utilizam *tilemap* como representação do ambiente.

Comportamento da movimentação

Durante o desenvolvimento do componente *Pathfinder2D*e do jogo demonstrativo Katch-Up, ficou claro que seria necessária a implementação de um componente auxiliar para definir, dado que um objeto no ambiente de jogo já tenha uma trajetória definida, como se daria sua movimentação por esse caminho.

Para isso, foi criado o componente auxiliar *Follower2D*, que simplesmente movimenta o objeto em uma linha reta entre os pontos. Este componente serve ao seu propósito, que foi de testar e demonstrar o funcionamento do componente *Pathfinder2D*. Porém, para ser de utilidade para a *Gear2D* e seus usuários, o módulo de movimentação dos objetos deve ser capaz de identificar colisões com outros objetos e reagir de forma aceitável.

Para resolver esse problema, propõe-se a implementação de um componente de movimentação que utilize *steering behaviors* [28] para mover os objetos através da trajetória calculada pelo componente *Pathfinder2D*.

Referências

[1] M. Anglada. An improved incremental algorithm for constructing restricted delaunay triangulations. *Computers & Graphics*, 21(2):215–223, 1997.

[2] Entertainment Software Association. 2011 sales, demographics, and usage data. http://www.theesa.com/facts/pdfs/ESA_EF_2011.pdf, 2011. Acessado em Outubro de 2011.

[3] G. Bauzil, M. Briot, and P. Ribes. A navigation sub-system using ultrasonic sensors for the mobile robot HILARE. *1st International Conference on Robot Vision and Sensory Controls*, pages 47–58, 681–698, 1981.

[4] M. Buro. Orts: A hack-free rts game environment. In *Computers and Games*, pages 280–291. Springer, 2003.

[5] M. Buro. Real-time strategy games: a new ai research challenge. *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1534–1535, 2003.

[6] M. Buro. Orts - a free software rts game engine. http://skatgame.net/mburo/orts/, 2011. Acessado em Julho de 2013.

[7] B. Chazelle. A theorem on polygon cutting with applications. *In Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 339–349, 1982.

[8] I. Craig. Formal techniques in the development of blackboard systems. *International journal of pattern recognition and artificial intelligence*, 7(02):197–219, 1993.

[9] R. Crandall and J. Sidak. Video games: Serious business for america's economy. http://papers.ssrn.com/sol3/papers.cfm?abstract_id=969728, 2006. Acessado em Novembro de 2011.

[10] D. Demyen and M. Buro. Efficient triangulation-based pathfinding. *Masters Abstracts International*, 45(03), 2006.

[11] B. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM (JACM)*, 39(2):253–280, 1992.

[12] L. Freitas, R. Bonifácio, and C. Castanho. Gear2d: um motor extensível de jogos baseado em componentes. *X Simpósio Brasileiro de Games e Entretenimento Digital*, 2011.

[13] L. Freitas, L. Reffatti, I. Sousa, A. Cardoso, C. Castanho, R Bonifácio, and G. Ramos. Gear2d: an extensible component-based game engine. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 81–88. ACM, 2012.

[14] E. Gamma, R. Helm, E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] L. Guibas, D. Knuth, and M. Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7(1):381–413, 1992.

[16] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[17] J. Hershberger and J. Ssnoeyink. Computing minimum length paths of a given homotopy class. *Computational geometry*, 4(2):63–97, 1994.

[18] J. Hershberger and S. Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999.

[19] M. Kallmann. Path planning in triangulations. *In Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 49–54, 2005.

[20] M. Kallmann, H. Bieri, and D. Thalmann. Fully dynamic constrained delaunay triangulations. *Geometric Modelling for Scientific Visualization*, pages 241–257, 2003.

[21] S. Kambhampati and L. Davis. Multiresolution path planning for mobile robots. *Robotics and Automation, IEEE Journal of*, 2(3):135–145, 1986.

[22] S. Koenig. A comparison of fast search methods for real-time situated agents. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 2:864–871, 2004.

[23] J. Kuffner Jr and S. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE Computer Society, IEEE Computer Society, 2000.

[24] John E Laird. Using a computer game to develop advanced ai. *Computer*, 34(7):70–75, 2001.

[25] Nash Information Services LLC. US movie market summary for 2010. http://www.the-numbers.com/market/2010.php. Acessado em Outubro de 2011.

[26] A. Nareyek. AI in computer games. *Queue*, 1:58–65, February 2004.

[27] Recording Industry Association of America. 2010 year-end shipment statistics. http://www.riaa.com/keystatistics.php?content_selector=2008-2009-U.S-Shipment-Numbers, 2010. Acessado em Outubro de 2011.

[28] C. Reynolds. Steering behaviors for autonomous characters. *Miller Freeman Game Group, San Francisco, California*, pages 763–782, 1999.

[29] P. Rezende, T. Lee, and P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *In Networks*, 14:393–410, 1985.

[30] S. Russell, P. Norvig, J. Canny, J. Malik, and D. Edwards. *Artificial intelligence: a modern approach*, volume 3. Prentice hall Englewood Cliffs, NJ, 1995.

[31] P. Tozour. Search space representations. *AI Programming Wisdom 2*, pages 85–101, 2003.