UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FERNANDO GABRIEL MACHADO COELHO

Estudo Sobre a Performance de Planejamento de Caminhos Utilizando Diagramas de Voronoi

Projeto de Diplomação

Prof[a]. Dr[a]. Luciana Nedel
Orientador

Me. Francisco Moura Pinto
Co-orientador

Porto Alegre, julho de 2012

AGRADECIMENTOS

Gostaria de agradecer aos meus pais por terem me dado as condições necessárias para minha formação intelectual e moral e, principalmente, por terem me ensinado valores de honestidade e sinceridade. Agradeço à minha irmã por ter sempre sido um grande exemplo de perseverança e ao meu irmão por ter me ensinado a questionar e a ter uma visão lógica e racional do mundo. Gostaria de agradecer também à minha namorada pela companhia e pela paciência nos últimos meses.

Gostaria de agradecer também à Professora Luciana Porcher Nedel por ter me orientado neste trabalho e pelo esforço achar horários em sua agenda já apertada para nossas reuniões. Agradeço ainda ao Mestre Francisco de Moura Pinto pelas explicações de como a técnica baseada em Voronoi funciona, pela ajuda no trabalho de integração entre meus programas e o dele e pelos muitos e-mails de dúvida que me respondeu.

Por fim, eu gostaria de expressar minha gratidão a UFRGS pelo excelente ensino que me proporcionou nos últimos anos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 5

LISTA DE FIGURAS 6

LISTA DE TABELAS 7

RESUMO 8

ABSTRACT 9

1 INTRODUÇÃO 10
1.1 Contexto 10
1.2 Contribuição 11
1.3 Organização do texto 11

2 TÉCNICAS DE REPRESENTAÇÃO E PLANEJAMENTO DE CAMINHOS 13
2.1 Técnicas de Representação de Cenário 13
2.2 Algoritmos de Busca 14
2.2.1 Busca por tabelas 14
2.2.2 A* 15
2.2.3 OpenSteer 15
2.3 Representação e Busca Utilizando Voronoi 17
2.4 Exemplos de Utilização 18

3 SOLUÇÃO PROPOSTA E FERRAMENTAS UTILIZADAS 19
3.1 Introdução 19
3.2 Panda3D 19
3.3 PandAI 20
3.4 Implementação com Voronoi 23
3.5 Aplicação Cliente Geradora de Simulações 25

4 EXPERIMENTOS E RESULTADOS 27
4.1 Introdução 27
4.2 Métricas Utilizadas 27
4.3 Hardware e Detalhes do Desenho da Cena 27
4.4 Agentes se Cruzando em um Campo Livre 28
4.5 Agentes Cercados de Obstáculos 31
4.6 Agentes em um Túnel 34

4.7 Agente com Muitos Objetivos 37
4.8 Simulação com Muitos Agentes 37

5 CONCLUSÕES E TRABALHOS FUTUROS 41
5.1 Conclusões 41
5.2 Trabalhos Futuros 42

REFERÊNCIAS 43

LISTA DE ABREVIATURAS E SIGLAS

PNJ	Personagem(ns) não jogador(es)
FPS	Frames per second
GUI	Guided user interface
E/S	Entrada e saída
BSD	Berkeley Software Distribution
ABI	Application Binary Interface
API	Application Programming Interface
GHz	Giga Hertz
GB	Gigabyte
RAM	Random Access Memory
GDDR5	Graphics Double Data Rate Version 5

LISTA DE FIGURAS

Figura 2.1: Esquerda: Representação fictícia de pontos marcados em Stormwind (World of Warcraft). Direita: Tabuleiro gerado. 13
Figura 2.2: Exemplo de um tabuleiro com pontos e uma tabela correspondente. . 14
Figura 2.3: Esquerda: Caminho retornado pelo algoritmo A*. Direita: Caminho retornado pelo algoritmo de Dijkstra. 16
Figura 2.4: Exemplo com dois comportamentos possíveis. 17
Figura 2.5: Representação do mundo e o diagrama de Voronoi gerado. 18

Figura 3.1: Fluxograma com a relação dos objetos da PandAI. 21

Figura 4.1: Estado inicial da primeira simulação com 32 agentes. 28
Figura 4.2: Gráfico completo da primeira simulação. 29
Figura 4.3: Rastros gerados na primeira simulação com 8 agentes. 29
Figura 4.4: Rastros gerados na primeira simulação com 16 agentes. 30
Figura 4.5: Rastros gerados na primeira simulação com 24 agentes. 30
Figura 4.6: Rastros gerados na primeira simulação com 32 agentes. 30
Figura 4.7: Estado inicial da segunda simulação com 10 agentes. 31
Figura 4.8: Gráfico completo da segunda simulação. 32
Figura 4.9: Rastros gerados na segunda simulação com 4 agentes. 32
Figura 4.10: Rastros gerados na segunda simulação com 6 agentes. 33
Figura 4.11: Rastros gerados na segunda simulação com 8 agentes. 33
Figura 4.12: Rastros gerados na sagunda simulação com 10 agentes. 33
Figura 4.13: Estado inical da terceira simulação com 32 agentes. 34
Figura 4.14: Gráfico completo da terceira simulação. 35
Figura 4.15: Rastros gerados na terceira simulação com 8 agentes. 35
Figura 4.16: Rastros gerados na terceira simulação com 16 agentes. 36
Figura 4.17: Rastros gerados na terceira simulação com 24 agentes. 36
Figura 4.18: Rastros gerados na terceira simulação com 32 agentes. 36
Figura 4.19: Estado inicial da quarta simulação. 37
Figura 4.20: Gráfico completo da quarta simulação. 38
Figura 4.21: Estado inicial da quinta simulação. 39
Figura 4.22: Gráfico completo da quinta simulação. 39
Figura 4.23: Agentes presos tentando chegar na mesma posição. 40

LISTA DE TABELAS

Tabela 4.1: Simulação 1 – Resultados. 28
Tabela 4.2: Simulação 2 – Resultados. 31
Tabela 4.3: Simulação 3 – Resultados. 34
Tabela 4.4: Simulação 4 – Resultados. 37
Tabela 4.5: Simulação 5 – Resultados. 38

Tabela 5.1: Comparação entre os resultado dos métodos. 41

RESUMO

O planejamento de caminhos é a ação de levar um agente de um ponto a outro em um mundo virtual ou real. As técnicas utilizadas para atingir este objetivo devem, também, evitar colisões com os objetos e outros agentes.

Neste estudo serão apresentados os algoritmos já usados e aceitos no mercado de jogos e simulações, assim como um novo baseado em diagramas de Voronoi. Este novo método apresenta como principais vantagens atualização dinâmica para cenários muito grandes e gera caminhos filtrados sem uma fase extra de refinamento.

Com o objetivo de gerar resultados úteis para a comparação entre as técnicas apresentas e encontrar pontos em que a nova técnica pode melhorar foram desenvolvidos testes. O foco principal foi criar situações envolvendo cenários nos quais os algoritmos normalmente erram e simulações com um número grande de objetos para testar a performance e escalabilidade.

Palavras-chave: Planejamento de caminhos, jogos, motores de jogos, A*, OpenSteer, Panda3D, PandAI, Voronoi.

A study on the Performance of Path Planning Using Voronoi Diagrams

ABSTRACT

Path planning is the action of moving an agent from one point to another in a virtual or real world. The techniques used to solve this problem must also avoid collisions with objects and other agents.

In this study we will show the algorithms that are most used in the game and simulation industry, as well as a new approach based on Voronoi diagrams. This new method's greatest features are the dynamic update of large scale scenarios and the path smoothing without the need of an extra phase.

With the objective of generating useful data to compare the techniques and finding out which points the new technique can be improved many tests were developed. The main focus was to create situations with scenarios where the algorithms are more likely to fail and simulations with a big number of objects to test the performance and scalability.

Keywords: path planning, games, game engines, A*, OpenSteer, Panda3D, PandAI, Voronoi.

1 INTRODUÇÃO

1.1 Contexto

O termo planejamento de caminhos é normalmente utilizado para descrever a busca do melhor caminho entre dois ou mais pontos. Seja em mundos virtuais (em aplicações de computação gráfica, como jogos e simuladores) ou no mundo real (cuja aplicação principal é a robótica) os problemas de planejamento de caminho apresentam um grande desafio pelo grande número de variáveis e pelo tempo de processamento disponível.

Inicialmente é importante observar que o melhor não é necessariamente o menor, pois existem vários fatores que podem impedir a solução mais curta. O caminho escolhido deve considerar primeiramente obstáculos, que podem ser fixos (no caso de um jogo, por exemplo, não é desejável que os PNJ atravessem os objetos do cenário) ou móveis (projéteis ou outros personagens). O algoritmo tem que levar em consideração as limitações de deslocamento que um agente pode apresentar, se o programa calcula o caminho de um peixe, ele não deve incluir pontos fora da água no caminho. Alguns agentes podem também possuir duas formas de movimentação. Por exemplo uma foca pode andar devagar ou nadar rápido. Neste caso, a técnica deve conseguir diferenciar se é melhor passar por um caminho de cinco metros de terra ou doze de água, por exemplo. Então, neste trabalho, referências ao termo melhor caminho devem ser entendidas como aquele que pode ser executado no menor tempo sem que o agente atravesse ou colida com um obstáculo.

Outra questão importante é que a técnica deve ser capaz de controlar múltiplos agentes. É importante que ela trate de forma especial o caso de agentes andando próximos ou em direção uns dos outros, pois se os dois se comportarem exatamente da mesma forma um irá bloquear a rota do outro. Este problema é especialmente agravado quando há pouco espaço livre em volta deles, pois muitos algoritmos de esquiva determinam que quando um agente está próximo do outro, ele deve se mover para um lado, que neste caso não será possível.

Os cenários das simulações também introduzem uma série de desafios. Quase todas as técnicas lidam somente com ambientes em 2D, mas na esmagadora maioria dos casos eles são, na realidade, 3D. Em cenários com múltiplos andares cada um deles é representado como um ambiente 2D separado e o algoritmo tem que tratar as transições entre eles. Outro grande problema na questão de ambientes é que eles nem sempre são estáticos, muitas vezes a interação dos personagens com eles pode fechar uma rota. Um exemplo deste problema é um dos monstros do jogo Diablo I, conhecido como Butcher, em que uma das estratégias mais simples de derrotá-lo é levá-lo para dentro de uma cela, sair dela e fechar a porta com ele dentro. O inimigo fica preso, pois não consegue abrir a porta. Como as paredes são grades você pode atacar através delas com uma arma à distância e matá-lo sem ele poder reagir [1].

Por fim, existem as limitações de tempo de processamento que variam conforme a aplicação. No caso dos jogos, a técnica deve ser muito rápida em comparação ao tempo de desenho dos gráficos, que idealmente varia de 30 a 72 FPS. Como o mundo muda a cada *frame* o algoritmo deve ser capaz de reagir a cada atualização. No entanto, ele não deve gastar tempo de processamento a ponto de impactar severamente a média de FPS.

1.2 Contribuição

O objetivo deste trabalho foi um estudo comparativo entre três algoritmos de planejamento de caminhos. As técnicas escolhidas foram A* (pois esta é a mais famosa e utilizada quando existe um grafo representando o cenário 3D), OpenSteer (pois esta é muito utilizada quando não existe representação do ambiente por um grafo) e um algoritmo baseado em Voronoi (pois este é novo e ainda não utilizado comercialmente). Desta forma espera-se gerar resultados sobre o quanto e em quais casos uma técnica é melhor ou pior do que as outras.

Para comparação de melhor técnica serão usadas medidas da taxa de FPS e comparação visual dos resultados. O objetivo final é avaliar a reação dos agentes guiados por cada método quanto a obstáculos e outros agentes ao seu redor, o custo computacional de cada método e como cada solução escala conforme o número de personagens controlados cresce.

Durante este trabalho foram adicionadas duas modificações principais à técnica baseada em Voronoi. Primeiro, para resolver qualquer problema de compatibilidade (neste, trabalho, ou em qualquer possível futuro), o programa foi transformado em um servidor e foi implementado um protocolo de comunicação via sockets.

Após resolver-se o problema de compatibilidade, existia a limitação dos agentes serem apenas pontos e, quando trazidos para um ambiente em três dimensões – no qual o agente era representado por um modelo – não existia noção de para onde ele olhava. Para resolver este problema o programa original recebeu uma extensão na qual ele salva a posição do agente antes e depois de movê-lo e, assim que concluído o deslocamento, calcula o ângulo do vetor entre elas.

1.3 Organização do texto

O presente trabalho está organizado da seguinte forma. No capítulo 2 são apresentados os algoritmos mais conhecidos usados atualmente. Na primeira seção encontra-se uma explicação das formas utilizadas para representar os cenários de forma computacional. Na segunda apresenta-se brevemente como funcionam os algoritmos assim como suas principais vantagens e desvantagens. Na terceira tem-se uma rápida análise das técnicas utilizadas nos principais motores de jogo do mercado e alguns jogos. Na última parte encontra-se uma simples explicação da solução baseada em Voronoi.

O capítulo 3 está dividido em cinco partes onde será descrito o motor de jogos utilizado para a implementação dos testes deste trabalho. Na primeira seção tem-se uma breve introdução. Na segunda, é apresentada a história da Panda3D, assim como suas principais funcionalidades. Na terceira é demostrada a arquitetura do módulo de inteligência artificial, assim como a descrição e nomenclatura dos algoritmos que ele implementa. Na quarta aborda-se as modificações necessárias na implementação do mestre Francisco de Moura Pinto, assim como uma descrição do desenvolvimento das novas funcionalidades. Na última seção apresenta-se o programa desenvolvido para gerar as simulações de teste

e medir os resultados.

No capítulo 4 são apresentados os testes e os resultados, assim como uma explicação das métricas e do hardware utilizados.

No capítulo 5 são apresentadas a conclusão do trabalho, assim como sugestões para possíveis expansões e trabalhos futuros.

2 TÉCNICAS DE REPRESENTAÇÃO E PLANEJAMENTO DE CAMINHOS

2.1 Técnicas de Representação de Cenário

Para iniciar o planejamento de caminhos precisa-se de uma representação do mundo em questão na qual um computador possa trabalhar. Para tal normalmente usa-se uma representação 2D de todo o cenário (também conhecida como rede de busca de caminhos [2]). Existem outras técnicas, que apenas salvam estruturas que representam seus elementos assumindo que todo o resto do cenário está vazio.

Uma das formas de representar a rede de busca de caminhos é através de um tabuleiro gerado automaticamente a partir de um modelo 3D do mundo. No final das contas tem-se uma estrutura que será vista e trabalhada como um grafo onde cada nodo é uma área do mundo e as arestas representam as conexões entre eles.

As representações mais simples abstraem cada nodo para formas geométricas estáticas, normalmente definidas por quadrados ou hexágonos, possibilitando que um personagem se mova para oito ou seis nodos vizinhos, respectivamente. As técnicas mais modernas representam as áreas em que os personagens podem se mover como polígonos de tamanho variável, esta técnica é muito comum e chamada de malha de navegação. O maior problema destas representações é que elas costumam consumir muita memória e trabalham com uma grande quantidade de dados.

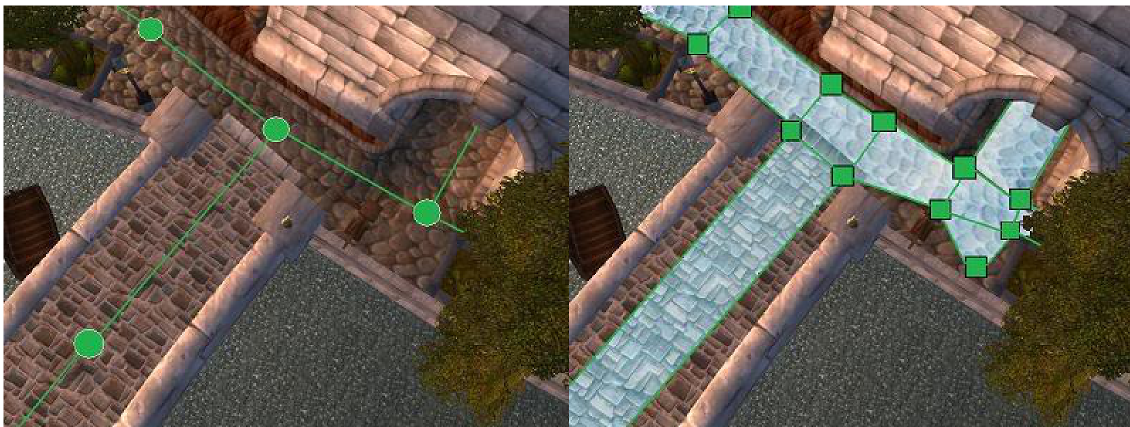
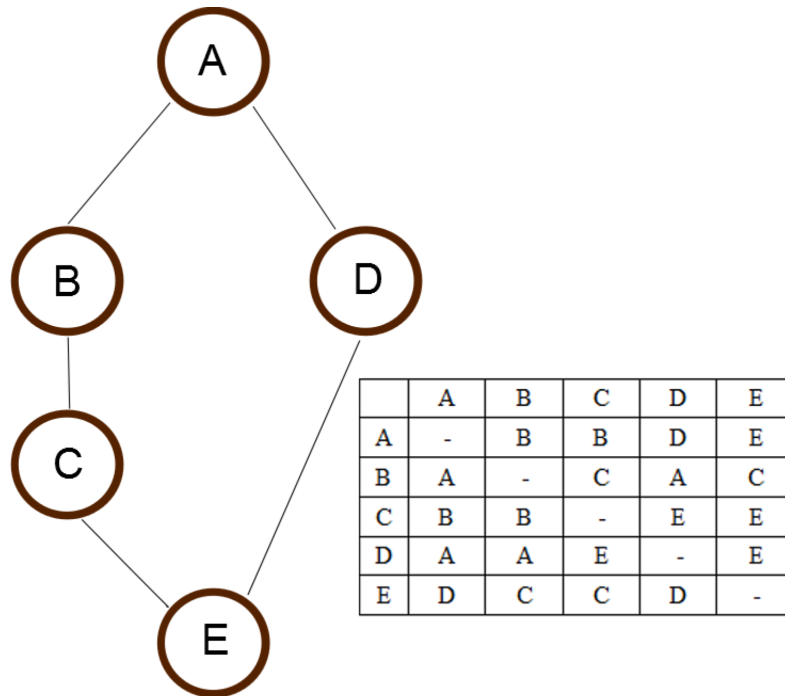


Figura 2.1: Esquerda: Representação fictícia de pontos marcados em Stormwind (World of Warcraft). Direita: Tabuleiro gerado. [3]

Outra aproximação possível é gerar um tabuleiro a partir de pontos marcados manualmente no cenário. Depois de gerados, é possível ligar os mais próximos, gerando uma



	A	B	C	D	E
A	-	B	B	D	E
B	A	-	C	A	C
C	B	B	-	E	E
D	A	A	E	-	E
E	D	C	C	D	-

Figura 2.2: Exemplo de um tabuleiro com pontos e uma tabela correspondente.

mapa de regiões navegáveis (figura 2.1). O maior problema desta aproximação é que os pontos precisam ser marcados por um humano, que gasta mais recursos e pode introduzir erros se o cenário for atualizado e os pontos não. Ainda assim, esta técnica tem duas grandes vantagens: ela pode reduzir consideravelmente a quantidade de dados e também pode-se usar múltiplos tabuleiros para simular comportamentos. Um exemplo deste último caso é se for necessário que um agente tente passar por um local sem ser visto, pode-se gerar um tabuleiro de forma que as regiões em que ele pode andar sejam sempre cobertas por terreno (por trás de muros, árvores, arbustos, etc), desta forma ele irá sempre caminhar por lugares onde ele ficará parcial ou totalmente escondido.

2.2 Algoritmos de Busca

Após criada uma representação utiliza-se um algoritmo de busca que opere sobre ela para descobrir o melhor caminho entre o ponto atual e o que queremos alcançar. Nesta seção serão explicados os algoritmos de busca por tabela, A* e Opensteer.

2.2.1 Busca por tabelas

Uma alternativa simples para buscar um caminho é utilizar uma tabela de busca. A estrutura da tabela é uma matriz n por n (onde n é o número de pontos marcados, na figura 2.2 tem-se um mapa e uma tabela referente a ele), normalmente pré-processada, que possui todas as referências possíveis entre dois nodos. Para descobrir como se vai de um ponto x para um y o algoritmo segue os seguintes passos:

1. Verificar na linha x da tabela a entrada que está na coluna y e salvar esta posição, que será chamada durante a explicação como z.

2. Deslocar-se até o ponto z no mapa.

3. Verificar se z é igual a y, se for finalizar a busca, senão igualar x a z e começar novamente.

Esta técnica apresenta duas grandes desvantagens: a primeira é que seu consumo de memória cresce exponencialmente e a segunda é que a tabela serve somente para um cenário estático. Caso o mundo seja alterado, a tabela precisa ser totalmente recriada, fazendo esta técnica pouco utilizável em jogos nos quais é possível bloquear ou abrir novas rotas. No entanto, para o caso específico de um cenário pequeno e sem alterações seu uso apresenta excelentes resultados.

2.2.2 A*

Quando os cenários são maiores ou muito dinâmicos precisa-se de um algoritmo que consiga ser rápido o suficiente para ser executado a cada atualização do mundo. Como a maioria das representações de mundo podem ser transformadas em grafos, o problema pode ser visto como achar o menor caminho entre dois vértices.

O algoritmo de Dijkstra [4] é uma maneira muito rápida e conhecida de calcular este valor. Ele utiliza a distância entre um ponto e seus vizinhos para calcular a menor entre todos os elementos de um tabuleiro. Um detalhe importante que se deve observar é que a área pesquisada pelo algoritmo de Dijkstra cresce para todas as possíveis direções conforme a busca segue.

O algoritmo A* funciona de forma similar, mas ele utiliza uma heurística para restringir a área de busca, minimizando o custo total de processamento. Na figura 2.3 tem-se uma uma comparação entre o Dijkstra e o A*, nela os pontos Azuis não preenchidos são os próximos que seriam analisados, os demais foram investigados. Neste exemplo pode-se ver facilmente que o número de vértices visitados pelo A* é consideravelmente menor.

A escolha da heurística depende do tipo de movimento que o agente pode executar, normalmente a heurística utilizada é a distância euclidiana (que deve ser usada quando o agente pode se mover em qualquer ângulo). Para um explicação mais extensa sobre o A* pode-se consultar [5] e uma análise matemática em [6].

O A* por si só ainda apresenta alguns problemas. Um deles é que a quantidade de dados processados cresce de forma não linear conforme a distância dos pontos. Outro problema é que o algoritmo não consegue detectar a priori se o ponto desejado é atingível, tendo que executar uma busca completa para definir com certeza que é impossível chegar na localização desejada. Por fim, o caminho resultante dificilmente será natural.

2.2.3 OpenSteer

Ao contrário dos demais algoritmos apresentados nas sessões anteriores, o OpenSteer é um algoritmo que trabalha sem uma representação do mundo, mantendo apenas dados sobre os agentes e os objetos que o habitam.

Ele é um algoritmo baseado em vetores de força, por exemplo se o agente deseja ir a um ponto x qualquer, ele terá uma força o empurrando em direção ao objetivo. Já os obstáculos criam vetores de força repulsiva. Portanto, se um agente se aproximar os dois vetores serão operados, gerando uma força resultante que, se seguida, gera um caminho que desvia do obstáculo, mas ainda vai em direção ao ponto final.

No modelo apresentado por Reynolds (dito simplificado) [9], todos os agentes tem os seguintes atributos: massa (escalar), posição (vetor), velocidade (escalar), força máxima (escalar), velocidade máxima (escalar) e orientação (n vetores). O número de componentes de cada vetor e n é definido pela dimensão do modelo, sendo três para 3D e dois para

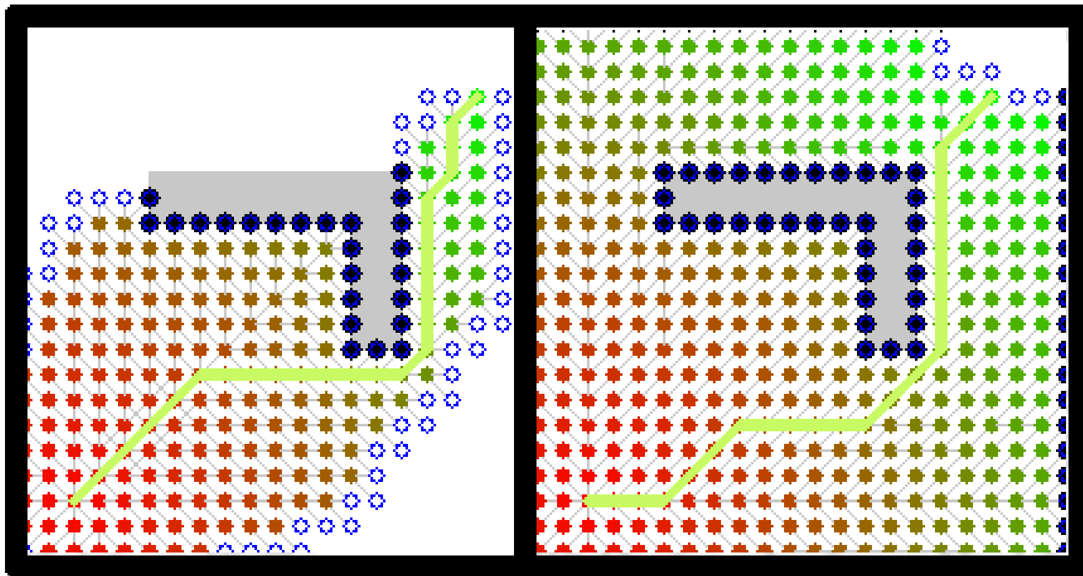


Figura 2.3: Esquerda: Caminho retornado pelo algoritmo A* [7]. Direita: Caminho retornado pelo algoritmo de Dijkstra [8].

2D.

A forma como os vetores serão operados depende dos comportamentos adicionados aos personagens controlados. Esses são divididos em duas categorias, os que tratam agentes únicos e os que manuseiam um grupo. Os comportamentos listados por Reynolds para agentes solitários são: busca, fuga, perseguição, evasão, perseguição a distância, chegada, desvio de obstáculos, vagar, acompanhamento de caminho, acompanhamento de campo de fluxo e desvio de colisão não alinhada. Já para os grupos temos: separação, coesão, alinhamento e acompanhamento de um líder.

Cada comportamento adiciona uma força repulsora ou de atração que após operadas geram um vetor resultante que será seguido. Na figura 2.4 tem-se o vetor de direção (em verde), o vetor desejado (em cinza) que representa a direção que deveria ser seguida para chegar no destino em linha reta, dois vetores de comportamento (em vermelho representa o comportamento de fuga do target, enquanto o azul representa o comportamento de busca) e o dois caminhos que serão gerados se o agente seguir os vetor resultante do comportamento escolhido (azul para busca e vermelho para fuga). Uma explicação completa de como cada comportamento altera o vetor direção pode ser encontrada em [9].

O OpenSteer apresenta como vantagem ser extremamente rápido, pois todos os cálculos são simples e ele escala de forma linear. Assim como o A* ele apresenta alguns problemas, inicialmente, o vetor de caminho resultante pode variar muito, fazendo com que o agente gire de forma não realista. Outro problema é que um agente pode acabar ignorando um obstáculo e passar pelo meio dele caso a força resultante dos outros comportamentos seja muito maior (ou um desvio de caminho for determinado muito próximo do objeto que queremos evitar). Assim como o A*, não existe garantia de que o caminho calculado aparente naturalidade.

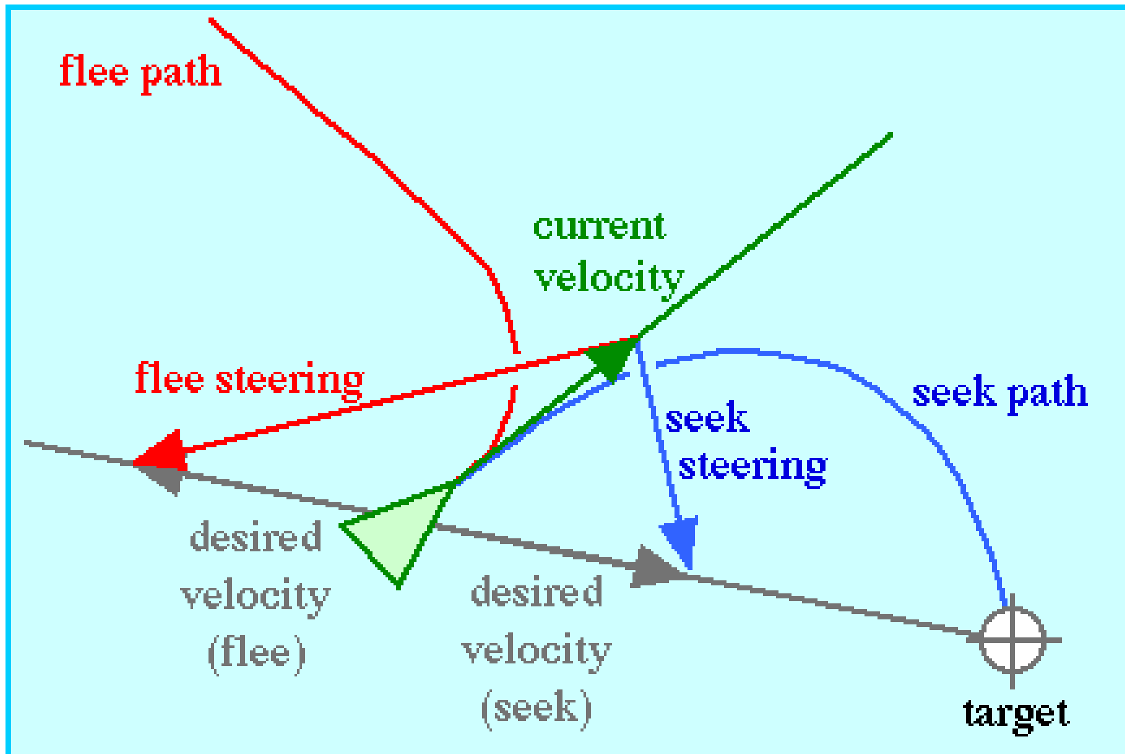


Figura 2.4: Exemplo com dois comportamentos possíveis. [9]

2.3 Representação e Busca Utilizando Voronoi

A técnica em questão utiliza um diagrama de Voronoi para a representação 2D do mundo. Neste caso o cenário pode representar as estruturas do ambiente como um círculo de raio qualquer (ideal para objetos cuja forma pode ser bem aproximada por um círculo, como postes, hidrantes ou buracos de bueiro) ou por polígonos 2D representados por quaisquer número de pontos. Os agentes são sempre representados por círculos de raio arbitrário. É importante usar sempre uma boa aproximação para os raios das entidades representadas, caso o contrário o algoritmo pode acabar ignorando colisões ou possíveis rotas.

Na figura 2.5 pode-se ver, à esquerda, um modelo 3D de um campo com uma caixa no centro e um agente. As setas são os pontos para os quais o agente deve se mover. Na direita encontra-se o diagrama de Voronoi que representa esta mesma cena. Podemos observar que o obstáculo e o agente afetam o diagrama, porém os pontos de locomoção não.

A cada movimentação de um agente ou de um obstáculo o diagrama de Voronoi fica desatualizado e precisa ser refeito. Na implementação do algoritmo baseado em Voronoi, a reconstrução do diagrama se dá de forma local, garantindo uma taxa de atualização extremamente rápida, mesmo para ambientes grandes.

O diagrama de Voronoi também é visto como um grafo. Portanto quando um agente precisa achar seu caminho entre um ponto e outro o algoritmo executa o A* para encontrar a melhor rota entre os dois pontos. O caminho escolhido é filtrado através de uma técnica que utiliza um dos princípios do diagrama de Voronoi. Como sempre existe pelo menos um círculo tangente a pelo menos três pontos de cada vértice (um destes, o agente), o algortimo faz com que o agente ande na borda deste círculo. Como o círculo é atualizado conforme o agente muda o resultado final desvia de obstáculos numa rota filtrada.

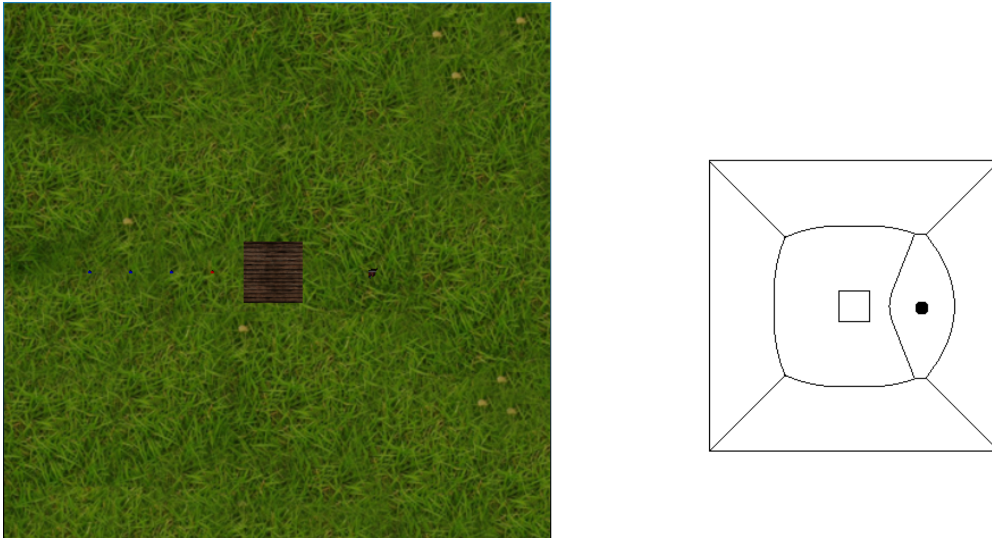


Figura 2.5: Representação do mundo e o diagrama de Voronoi gerado.

Uma explicação completa e mais detalhada de como o algoritmo funciona e das propriedades de Voronoi pode ser encontrada em [10].

2.4 Exemplos de Utilização

Atualmente soluções de planejamento de caminhos e para simulações com agentes estão presentes em todos os motores de jogos comercias pesquisados (Unity3D [11], CryEngine 3 [12], Unreal Engine [13] e Source [14]). Existem inclusive motores puramente dedicados para o planejamento de caminhos como a PathEngine [15].

A representação utilizada em todos os motores pesquisados é descrita como uma malha de navegação, porém somente a CryEngine afirma que o algoritmo de busca utilizado é o A* [16]. Na Unity3D o módulo de planejamento de caminhos só está disponível para usuários que possuem a licença PRO (que custa USD $1500,00), por isto existem módulos de expansão – também pagos, porém muito mais baratos – que utilizam tanto o A* quanto a OpenSteer.

A PathEngine também utiliza uma malha de navegação. Ela tem uma peculiaridade que ela age sobre os cenários, criando suas estruturas a partir deles e sua lógica é implementada via um SDK dentro do próprio jogo (ou seja, ela funciona como middleware). Isso difere dos motores de jogos que trabalham diretamente no cenário. Na documentação não foi encontrada nenhuma referência quanto ao algoritmo usado para encontrar os caminhos.

De Acordo com [2], Warcraft III usa um tabuleiro gerado automaticamente, já os jogos Unreal Tournament 2007 e Medal of Honor: Pacific Assault usam tabuleiros a partir de pontos. Na lista de clientes da PathEngine podemos encontrar mais de 50 jogos que a utilizam, entre eles The Witcher 2, Metro 2033: The Last Refuge, Banjo-Kazooie: Nuts & Bolts, Just Cause 2 e Stormrise.

3 SOLUÇÃO PROPOSTA E FERRAMENTAS UTILIZADAS

3.1 Introdução

A proposta deste trabalho é comparar as soluções baseadas em A* e OpenSteer com a implementada usando Voronoi, para testar se a última será mais rápida ou apresentará caminhos mais naturais. Para tal precisa-se de um motor de jogos que possa representar os cenários de teste e que implemente as técnicas para comparação. Por fim, precisa-se integrar a solução baseada em Voronoi com o motor de jogos.

Para resolver este problema foi utilizado a Panda3D, que é um motor de jogos que possui um módulo chamado PandAI que implementa os algoritmos A* e Opensteer. Já a integração foi resolvida através de comunicação via sockets. A descrição completa do motor de jogos, o módulo de IA, como foi feita a integração e como foram geradas as simulações está nas próximas quatro seções.

3.2 Panda3D

A Panda3D é um motor de jogos de código aberto portável que funciona em Linux, FreeBSD, Windows e MacOS[17]. Além de rodar diretamente nos sistemas operacionais como uma aplicação executável, ela também oferece um plugin para navegadores, permitindo que os usuários rodem os jogos diretamente online [18].

Ela foi desenvolvida inicialmente pela Disney para o jogo online Toontown [17] [19]. Em 2002 foi lançada como código aberto, porém alguns problemas na licença forçaram uma reformulação e desde 28 de maio de 2008 a licença foi refeita, desta vez baseada na Modified BSD License [20]. Atualmente o Centro de Tecnologia em Entretenimento da Universidade de Mellon se juntou ao projeto – ainda existem, também, grandes contribuições vindas da comunidade [17].

O código do motor é feito em C++, porém a linguagem de desenvolvimento sugerida (e com o maior suporte disponível) é Python. Todos os recursos são acessíveis via binds (mecanismo que traduz as classes e chamadas de métodos e funções de Python para C++), o que simplifica a criação de jogos (livrando o programador de ter que se preocupar com o gerenciamento de memória, por exemplo) e mantém a performance (pois a parte mais custosa em processamento é feita direitamente pelo C) [19].

Ao contrário de outras soluções populares de código aberto (como a OGRE [21] e a Irrlicht [22]), a Panda3D não é um motor puramente gráfico. Além de implementar uma solução gráfica bastante completa (possuindo, por exemplo: Shaders, Heightfields, Multithreaded Render Pipeline) ela também traz [23]:

- Som (integração com OpenAl, FMOD [1] e Miles[1]).
- Interface com teclado, mouse e outros dispositivos de E/S.
- GUI (pode-se usar o sistema próprio ou a integração com a Librocket).
- Sistema de colisão (integração com a Bullet)
- Física (integração com a Bullet e ODE)
- Gerenciamento automático de múltiplas threads (task system)
- Inteligência artificial (PandAI)
- Rede.

Comercialmente a Panda3D foi usada para jogos como Disney's Toontown, Disney's Pirates of the Caribbean Online e A Vampyre Story (Autumn Moon Entertainment). Além disso ela já foi utilizada no desenvolvimento de simuladores como Code3D (Sym Ops Studios) e é extensivamente empregada de forma educacional na cadeira Building Virtual Worlds no Centro de Tecnologia em Entretenimento da Universidade de Mellon [25].

A decisão de usar a Panda3D veio dela ter um módulo de inteligência artificial próprio e conseguir facilmente representar os cenários necessários. Outro fator que levado em conta foi que ela é de código aberto, em casos extremos pode-se simplesmente abrir o fonte e olhar o quê está acontecendo. Por fim, ela tem uma comunidade extremamente ativa, sendo muito raro uma dúvida no fórum não ter resposta em um dia.

3.3 PandAI

O módulo de inteligência artificial da Panda3D se chama PandAI, é um projeto que foi desenvolvido totalmente pela comunidade e adicionado recentemente na versão de distribuição. O módulo implementa busca de caminho através do algoritmo A* e também comportamentos de navegação baseados nos trabalhos de Reynolds [26] [27].

A primeira informação importante a se notar é que a implementação do A* é totalmente voltada para encontrar caminhos, desviando de obstáculos, ela não simula nenhum tipo de comportamento automaticamente. Também, é preciso gerar um arquivo com os pontos e pesos do cenário antes de podermos utilizar o A* [28]. Já os algoritmos baseados no OpenSteer podem ser usados diretamente com os modelos 3D.

Na estruturação da PandAI existem três módulos principais [29]. Na figura 3.1 pode-se ver a relação entre os módulos e como eles se relacionam, os retângulos verdes indicam classes e os azuis métodos.

O primeiro deles é módulo AIWorld. Ele cria o mundo que a inteligência artificial irá reconhecer a partir da cena. Caso queira-se utilizar os algoritmos baseados no OpenSteer, devemos adicionar os obstáculos a um objeto deste módulo, com o método addObstacle. Caso não existam obstáculos ou desejarmos usar o A*, devemos apenas adicionar os personagens ao objeto deste módulo, com o método addAiChar. Este método

O segundo módulo é o AICharacter, eles representam um personagem guiado pela inteligência artificial. Para criar um objeto dele é necessário passar como argumento um modelo 3D, sua massa, aceleração e velocidade máxima. Depois de criados e adicionados

[1] Estes módulos não são código aberto e é necessária uma licença individual para jogos comerciais [24].

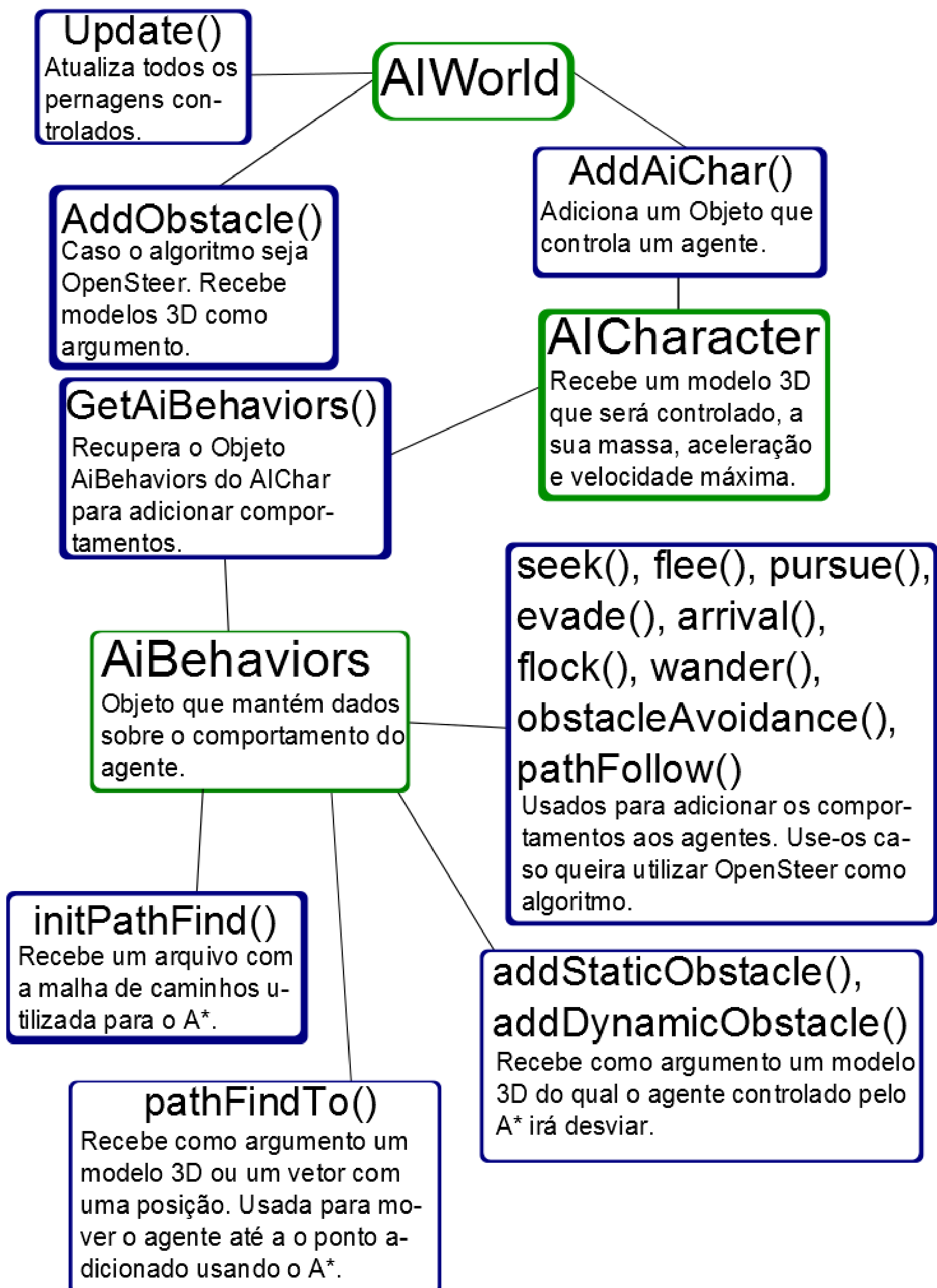


Figura 3.1: Fluxograma com a relação dos objetos da PandAI.

ao AIWorld, eles são automaticamente deslocados a cada atualização deste. O objeto é criado com um comportamento vazio inicializado, para definir o que o agente deve fazer, usa-se o próximo módulo.

Por fim, temos o módulo AIBehavior. No caso de utilizarmos o A*, é nele que devemos carregar o arquivo pré-processado e aqui que devemos adicionar os obstáculos. Já no caso do OpenSteer é aqui que definimos qual o tipo de comportamento queremos.

É importante ressaltar que no caso do A*, existe um tratamento diferenciado para obstáculos estáticos e dinâmicos. Administrar obstáculos estáticos requer muito menos uso de processamento, portanto é recomendável utilizá-los quando possível [30].

Existem no total seis tipos de comportamento de navegação, além de três modificadores (listados nesta ordem). São eles:

- Seek: Move o agente em direção a um alvo estático[9] [31].

- Flee: Move o agente na direção oposta a um alvo estático[9] [32].

- Pursue: Move o agente em direção a um alvo que pode sair da posição inicial[9] [33].

- Evade: Move o agente na direção oposta a um alvo que pode sair da posição inicial[9] [34].

- Path Follow: Comportamento similar ao seek, porém recebe diversos pontos. O agente irá se mover de sua posição inicial até o primeiro ponto e depois disso irá se mover entre todos os pontos, assim que chegar no último, ele retornará ao primeiro ponto. É usado para simular comportamento de patrulha. O comportamento descrito aqui está de acordo com a implementação [35] e não de acordo com o artigo de Reynolds [9].

- Wander: o agente irá se mover para pontos aleatórios dentro de uma região. É possível determinar a variação de ângulo entre cada mudança de posição, em quais eixos as posições serão escolhidas e o raio da região a partir da posição inicial[9] [36].

- Obstacle avoidance: modificador que faz com que o agente desvie de obstáculos[9] [37].

- Arrival: modificador para os comportamentos seek e pursue, ele faz com que o agente reduza gradativamente sua velocidade ao se aproximar do destino[9] [38].

- Flock: modificador que deve ser aplicado a múltiplos agentes, ele faz com que estes agentes se movam em grupo. Ele é comumente usado para simular cardumes, bandos de pássaros ou rebanhos[9] [39].

Após a criação de todos os objetos e de referenciá-los corretamente, devemos chamar o método update a cada *frame* redesenhado. O método de atualização irá automaticamente reposicionar os agentes e ajustar seu ângulo de visão de acordo[29].

3.4 Implementação com Voronoi

O algoritmo de path-planning baseado em Voronoi foi implementado no Visual Studio 2010, sem nenhuma consideração de portabilidade, isto acrescentou um grande limitante, pois a Panda3D só é compatível até a versão anterior. Infelizmente a ABI do compilador do Visual Studio muda a cada versão e não existe nenhum suporte a retrocompatibilidade.

Seria possível recompilar a Panda3D para a versão mais atual, porém para tal seria necessário compilar todas as dependências externas separadamente. Em uma busca rápida descobriu-se que diversos membros da comunidade já haviam tentado sem sucesso.

Outra opção de recompilação seria alterar o código do Voronoi para forçar a compatibilidade com o Visual Studio 2010. Esta opção também não se mostrou boa, pois além das mudanças serem grandes, seria um retrocesso no que a tecnologia oferece.

Por fim, decidiu-se que o melhor seria fazer os programas rodarem separadamente e se comunicarem. Inicialmente as ideias foram comunicação via socket ou blocos de memória comportalhada.

Após um estudo quanto as opções de memória compartilhada via a API do Windows [40] e através da biblioteca Boost [41], acabou-se preferindo usar a interface de sockets. O uso de memória compartilhada introduz muitos problemas de sincronização, os quais normalmente precisam ser resolvidos através de operações trancantes que podem comprometer a performance. Já os sockets, embora também implementados de forma trancante, usam apenas um canal de comunicação e só apresentariam problema (não de sincronização, mas sim de sobrecarga) se a produção de informação fosse maior que o consumo por um certo período de tempo.

Com a decisão de utilizar sockets foi necessário implementar um protocolo de comunicação entre os dois processos. Este protocolo deveria obedecer as seguintes regras:

- Deveria ser capaz de interromper a simulação a qualquer momento;
- Deveria ser capaz de gerar um cenário dinâmico para cada simulação, podendo adicionar um número ilimitado de agentes, obstáculos e caminhos para os agentes;
- Deveria ser capaz de responder com informações de atualização das posições do agente.

A partir das regras, foi gerada uma interface de comunicação tanto no lado da aplicação que calcula o algoritmo baseado em Voronoi quanto no lado das aplicações visuais que utilizam a Panda3D. A primeira foi desenvolvida em C++ – utilizando a API de sockets do Windows – já a segunda foi feita puramente em Python, utilizando os módulos de sockets e struct (para conversão dos blocos de dados recebidos em dados tipados para o Python).

O protocolo é implementado através de mensagens de tamanho variável, os primeiros 4 bytes são sempre um unsigned int que indica o seu código. As seguintes mensagens foram criadas:

- PROTOCOL_WORLD_SCALE: Esta mensagem contém mais 16 bytes que são duas variáveis do tipo double. Ela possui a informação da escala no eixo x e y, respectivamente. Estes dados são necessários porque a solução baseado em Voronoi trabalha sempre com a escala do mundo normalizada entre -1 e 1. Visto que as aplicações visuais normalmente trabalham com mundos muito maiores, é necessário converter as coordenadas, dividindo-as pela escala de ajuste.

- PROTOCOL_NEW_OBSTACLE: Esta mensagem contém mais 40 bytes que são oito variáveis do tipo double. Elas indicam quatro pares de pontos x e y que formam um obstáculo retangular (o único tipo suportado até o momento). Os pontos precisam ser informados no sentido horário.

- PROTOCOL_NEW_POINT: Esta mensagem contém mais 16 bytes que são duas variáveis do tipo double. Elas indicam um ponto que será adicionado no caminho a ser seguido pelo último agente criado.

- PROTOCOL_START_AGENT: Esta mensagem contém mais 16 bytes que são duas variáveis do tipo double. Elas indicam o ponto inicial de um novo agente.

- PROTOCOL_START_SIMULATION: Esta mensagem possui somente o código, ela sinaliza que a simulação deve começar. Antes de iniciar, o programa irá conferir se existe pelo menos um agente, se ele possui pelo menos um ponto e se as informações de obstáculo estão corretas.

- PROTOCOL_STOP_SIMULATION: Esta mensagem possui somente o código e pára a simulação.

- PROTOCOL_UPDATE_POSITION: Esta mensagem possui mais 28 bytes que são uma variável do tipo unsigned int e três do tipo double. O valor inteiro simboliza o identificador do agente, que é definido pela ordem crescente na qual eles foram gerados – partindo de zero. Os próximos três valores indicam as coordenadas x e y (já ajustadas para o mundo representado) e o ângulo de visão (em graus) do agente. A informação de ângulo pode ser NaN, a aplicação que o recebe deve tratar isto.

- PROTOCOL_POINT_REACHED: Esta mensagem possui mais 4 bytes que são uma variável do tipo unsigned int. O valor inteiro simboliza o identificador do agente, conforme descrito anteriormente. Ela é usada pelo programa cliente para cessar a animação do personagem quando ele já passou por todos os pontos do seu caminho.

- PROTOCOL_SIMULATION_FINISHED: Esta mensagem possui somente o código e é usada para informar que a simulação acabou.

Além da implementação do protocolo de comunicação e de seu uso dentro da solução original, também foi necessário expandi-la para calcular o ângulo que o agente está olhando. Para tal foi utilizada a seguinte equação:

$atan2(next_y - former_y, next_x - former_x) * 180/\pi$

Onde:

$atan2$ é a variação da operação arco tangente.

$next_y$ é o novo valor da posição y do agente após ele se movimentar.

$next_x$ é o novo valor da posição x do agente após ele se movimentar.

$former_y$ é o valor da posição y do agente antes dele se movimentar.

$former_x$ é o valor da posição x do agente antes dele se movimentar.

O resultado da expressão é dado em radianos e como a Panda3D trabalha com graus, o resultado é multiplicado por $180/\pi$ para convertê-lo. Esta função foi escolhida ao invés do produto vetorial por simplicidade já que seu conjunto imagem está entre $-\pi$ e π, enquanto o da segunda que está entre 0 e π.

3.5 Aplicação Cliente Geradora de Simulações

Para criar as simulações foi desenvolvido um programa cujo objetivo inicial era gerá-las e executá-las. O programa possui três módulos principais: edição, execução e desenho de rastro.

Quando o usuário entra no programa ele está em um cenário com grama, ao apertar a tecla 1 ele entra no modo de edição. Neste modo ele pode alterar a cena usando os seguintes passos:

1. Para adicionar um agente ou uma caixa o usuário deve utilizar as teclas 2 e 3, respectivamente.

2. Após selecionar o tipo elemento ele irá aparecer na posição (0,0). O usuário pode utilizar as setas do teclado para posicioná-lo e a tecla espaço para definir o ponto corrente como posição final.

3. Se o objeto escolhido foi um agente, o usuário poderá controlar o modelo de uma flecha (novamente com as setas) que aponta para uma posição que será adicionada ao seu caminho. Para finalizar o posicionamento, basta utilizar a tecla espaço.

4. Após adicionar o primeiro objetivo ao caminho do agente, o usuário pode incluir outros com a tecla 2 ou espaço para finalizar o processo de inclusão de um agente ao mundo.

Após finalizar a edição do mundo o usuário pode voltar ao estado inicial apertando a tecla 1. Neste estado pode-se conferir que está listado o algoritmo corrente, ele pode ser modificado utilizando a tecla 3. Para iniciar uma simulação basta usar a tecla espaço.

O programa também implementa a opção de salvar e carregar cenas. Em qualquer momento durante a edição ou o estado inicial o usuário pode apertar a tecla S para salvar um documento de texto que contém as informações do cenário corrente. Se na linha de comando para iniciar o programa for passado o nome de um destes arquivos como argumento a cena salva será carregada. O Formato no qual a cena é salva é extremamente simples e pode ser alterado em qualquer editor de texto. Ele se organiza da seguinte forma:

- Linha começando com A: Significa que nesta linha temos a descrição de um agente. Depois da letra inicial deve ter um espaço em branco e Point3(x, y, 0.0), onde x, y são variáveis que definem a posição inicial.

- Linha começando com P: Significa que nesta linha temos a descrição de um ponto pertencente ao caminho do último agente adicionado. Depois da letra inicial deve ter um espaço em branco e Point3(x, y, 0.0), onde x, y são variáveis que definem a posição deste ponto. Um agente precisa de pelo menos um ponto.

- Linha começando com O: Significa que nesta linha temos a descrição de um obstáculo. Depois da letra inicial deve ter um espaço em branco e Point3(x, y, 0.0), onde x, y são variáveis que definem a posição inicial.

O programa também apresenta a opção de desenhar rastros que será ativada se na linha de comando para iniciar o programa o terceiro elemento for 1 ou True. Neste caso, a cada 0.15 segundos o programa salva a posição de cada agente (até 2000 posições individuais

são salvas) e no fim da simulação ele desenhará uma linha representando o caminho e salvará uma foto.

As últimas funcionalidades relevantes são que a qualquer momento o usuário pode apertar a tecla A para salvar uma foto da cena atual. Também no fim de cada simulação o programa imprime a taxa de FPS da execução.

O fluxo da simulação funciona da seguinte forma: quando o usuário comanda que inicie uma simulação e o algoritmo selecionado é A* ou OpenSteer o programa irá criar os objetos conforme foram descritos na seção 3.3. Se o algoritmo selecionado for o Voronoi, o programa tentará conexão com o servidor que já deve estar rodando, assim que estabelecer a conexão ele enviará mensagens do tipo PROTOCOL_WORLD_SCALE, PROTOCOL_NEW_OBSTACLE, PROTOCOL_START_AGENT e PROTOCOL_NEW_POINT para que o servidor consiga gerar o mundo igual ao que foi desenhado no simulador. Depois de enviadas as mensagens para que os dois estejam trabalhando no mesmo cenário, uma mensagem do tipo PROTOCOL_START_SIMULATION é enviada para começar a simulação. O servidor então irá responder com mensagens PROTOCOL_UPDATE_POSITION e PROTOCOL_POINT_REACHED para que o cliente possa mover os agentes. Quando a simulação acaba o servidor envia uma mensagem do tipo PROTOCOL_SIMULATION_FINISHED e o cliente responde com uma mensagem PROTOCOL_STOP_SIMULATION, isto faz com que a thread que recebe informações no servidor volte aceitar a criação de novos mundo.

4 EXPERIMENTOS E RESULTADOS

4.1 Introdução

Neste capítulo é feita uma análise dos três algoritmos cujas implementações foram discutidas no capítulo anterior. Os problemas propostos tentam ver como os eles reagem à situações diversas, como por exemplo, agentes se cruzando ou tendo que se esquivar de muitos obstáculos.

Em todas as simulações são testados quatro variações nas quais são usados mais agentes ou mais obstáculos gradativamente. Assim pode-se ter uma noção de como o desempenho é afetado conforme a complexidade do cenário aumenta.

4.2 Métricas Utilizadas

A análise das três primeiras simulações será feita em três tipos de dados: a taxa de FPS, o rastro produzido e observações pertinentes. As duas últimas simulações são feitas com uma escala muito maior de dados, o objetivo delas é principalmente verificar a escalabilidade, portanto não serão observados os rastros – até mesmo porque os rastros de 100 agentes seriam confusos demais para alcançar qualquer conclusão.

As taxas de FPS observadas serão: desenho da cena (ou seja, todos os modelos carregados, mas nenhuma animação ou iteração com a IA) e depois a de cada algoritmo rodando. Os rastros serão demonstrados através de fotos que mostrarão os caminhos realizados pelos agentes. Por fim, conforme as simulações foram executadas, pode-se reparar em um padrão de erros ou comportamentos pouco naturais apresentado pelos agentes e eles serão descritos depois da apresentação dos outros dois dados.

Para as primeiras três simulações a taxa é a média entre três simulações. Para a quarta e a quinta apenas uma simulação (muito mais longa) foi executada. Todas as taxas de todas as simulações foram truncadas.

4.3 Hardware e Detalhes do Desenho da Cena

Todos os testes foram executados em um computador com processador AMD AM3 FX-6100 Zambezi com seis núcleos rodando a 3.3GHz, 4 GB de memória RAM e uma placa de aceleração gráfica ATI Radeon HD Série 4870 (Modelo de 1 GB GDDR 5). O Driver da placa de vídeo usado foi a versão 8.961-120405a-137813C-ATI usando o Catalyst versão 12.3.

Graficamente, os modelos usados são bastante simples e o único efeito extra oferecido pela Panda3D que foi usado foram luzes. Ainda, o Anti-aliasing foi desligado nas opções de otimização da placa de vídeo para garantir que ele não seria aplicado. A resolução

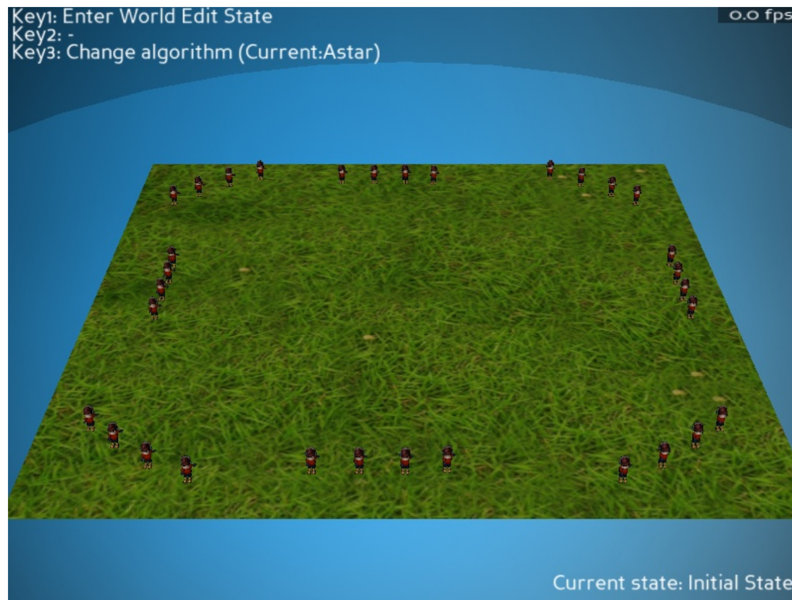


Figura 4.1: Estado inicial da primeira simulação com 32 agentes.

Agents	FPS Idle	FPS OpenSteer	FPS A*	FPS Voronoi
8	478	232	232	283
16	372	174	159	165
24	292	126	100	87
32	242	89	65	56

Tabela 4.1: Simulação 1 – Resultados.

utilizada nas simulações foi 800 x 600.

O Tamanho da cena é aproximadamente 56 vezes mais larga que o personagem. Assumindo que o personagem tem 1,80 e que a largura dos ombros é de aproximadamente um quarto da altura o mundo possui aproximadamente 25 metros de lado. Nas simulações quatro e cinco os agentes e os obstáculos assumem uma escala de 0.5, dobrando o tamanho do mundo.

4.4 Agentes se Cruzando em um Campo Livre

Neste primeiro teste o intuito é verificar como os agentes se esquivam uns dos outros. O cenário não terá obstáculos apenas diversos grupos de quatro agentes em uma linha lateral cujo objetivo é chegar na posição atrás da posição inicial do grupo em sua frente. O teste foi feito para dois, quatro, seis e oito grupos. A figura 4.1 mostra o estado inicial da simulação, com o número máximo de grupos.

Na tabela 4.1 podemos observar qual foi a taxa de FPS para cada algoritmo, o gráfico gerado a partir destes dados está na figura 4.2. O primeiro dado que pode-se observar é que a taxa de FPS cai drasticamente conforme os personagens se aproximam e, claramente, quantos mais personagens, maior a queda.

Neste teste nota-se um primeiro problema com o algoritmo baseado em Voronoi. Dois agentes podem acabar fazendo o mesmo caminho e ficar presos nesta situação por bastante tempo. Isto aconteceu tanto no caso com seis grupos. Nesta situação, além da falta de naturalidade, muito do processamento é jogado fora e, com isso, a taxa de FPS cai. Nos

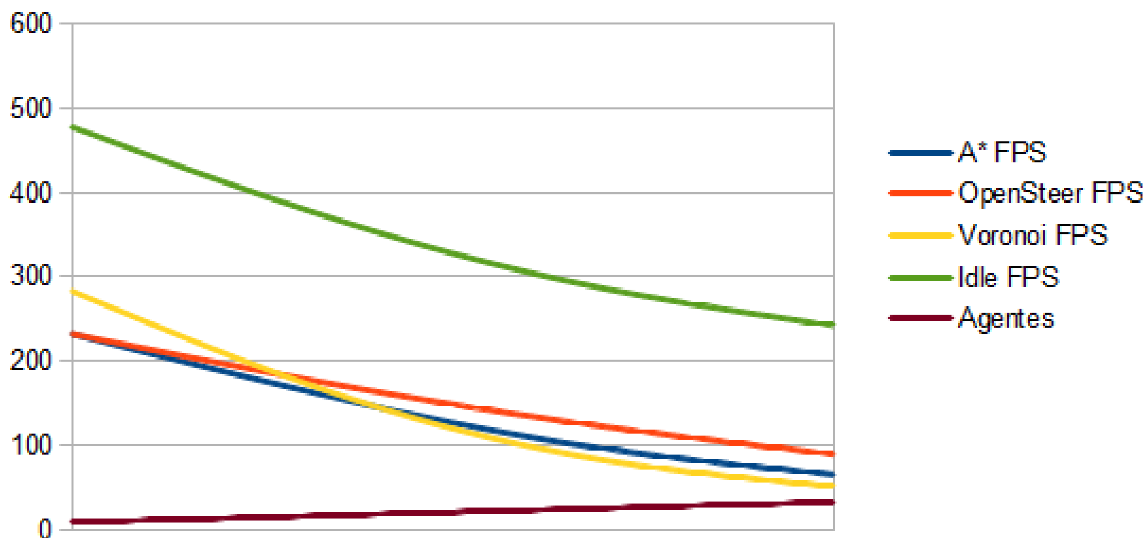


Figura 4.2: Gráfico completo da primeira simulação.

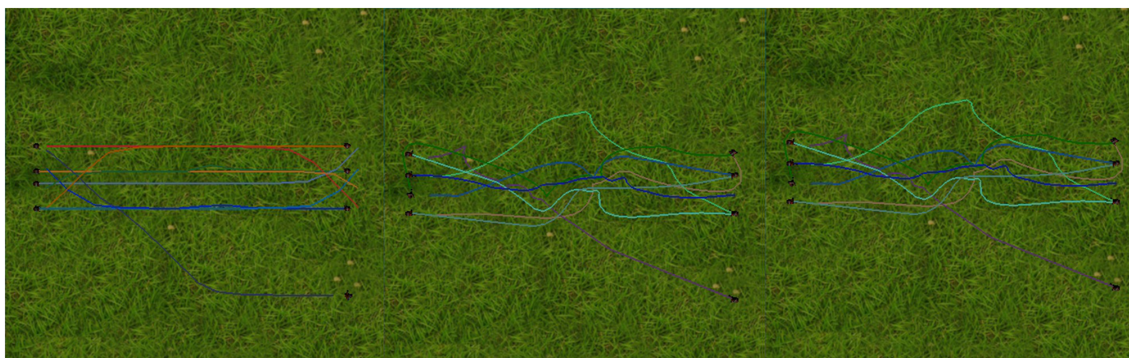


Figura 4.3: Rastros gerados na primeira simulação com 8 agentes.

caso em que tivemos este problema o resultado ficou 13% abaixo do o A*.

Pode-se ver os rastros que cada algoritmo gerou na figura 4.3 para dois grupos, 4.4 para quatro grupos, 4.5 para seis grupos e 4.6 para oito grupos. Em todas elas a ordem é A* no lado esquerdo, OpenSteer no centro e Voronoi no lado direito.

Analisando os rastros podemos observar algumas peculiaridades. O desvio que os agentes controlados pelo A* realizam é bastante curto e simples, sendo que algumas vezes eles acabam se cruzando. Também no caso do Opensteer acontece de os agentes se atravessarem quando existem muitos deles concentrados em uma área pequena. Em ambos os casos isto é bastante raro, no entanto no Voronoi isto nunca acontece. Outro comportamento que podemos observar é que no caso do A* os agentes tendem a caminhar em linhas retar, já no OpenSteer os caminhos apresentam um certo grau de atenuidade, enquanto no Voronoi os desvios de caminho sempre são atenuados.

Figura 4.4: Rastros gerados na primeira simulação com 16 agentes.

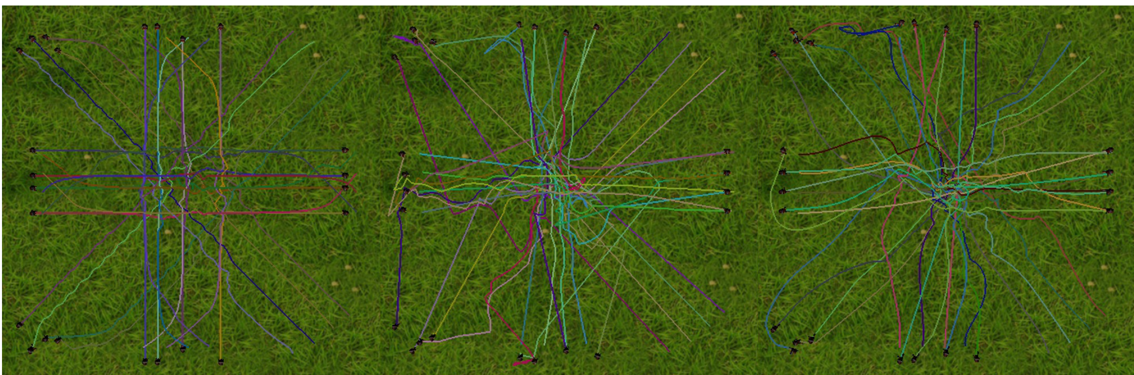


Figura 4.5: Rastros gerados na primeira simulação com 24 agentes.

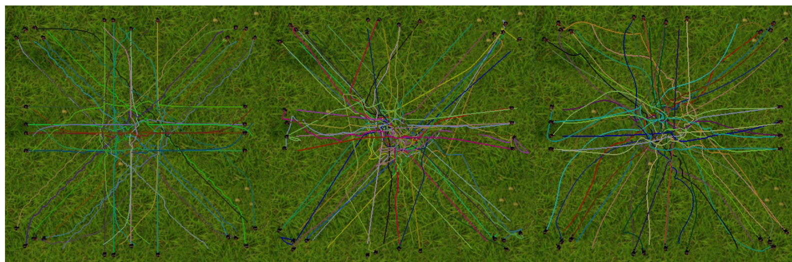


Figura 4.6: Rastros gerados na primeira simulação com 32 agentes.

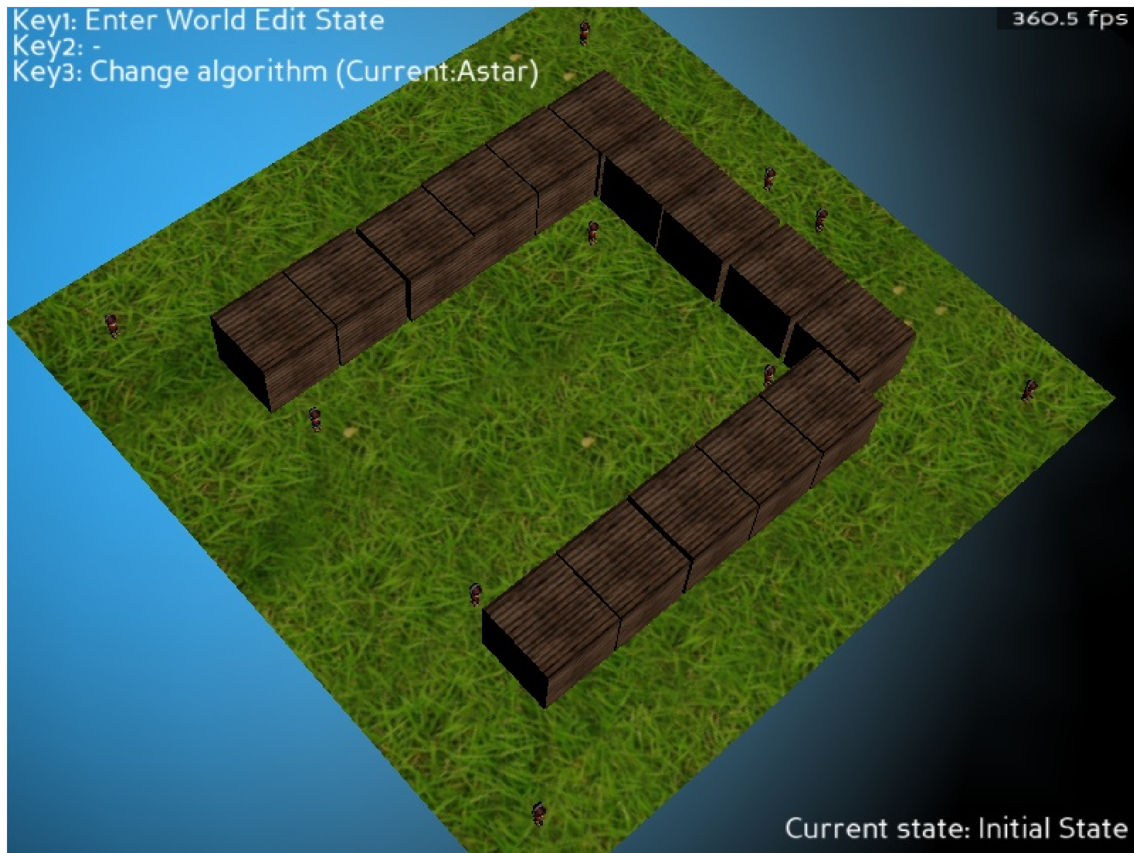


Figura 4.7: Estado inicial da segunda simulação com 10 agentes.

Agents	FPS Idle	FPS OpenSteer	FPS A*	FPS Voronoi
4	425	336	353	346
6	414	290	317	278
8	390	248	277	256
10	365	219	226	210

Tabela 4.2: Simulação 2 – Resultados.

4.5 Agentes Cercados de Obstáculos

Na segunda simulação o cenário tem diversos obstáculos que geram uma parede em formato de ferradura. Exitem alguns agentes que ficam dentro desta estrutura e outros que ficam fora, dependendo da simulação. O objetivo é rodear as paredes do cenário e retornar a posição inicial. A figura 4.7 mostra o estado inicial da simulação no caso com dez agentes.

Este teste foi feito para observar como os agentes se comportam em um espaço pequeno no qual eles possivelmente estariam cercados uns pelos outros. Outra situação interessante, neste cenário, é que os agentes, ao se moverem, podem fechar uma rota. Dessa forma, caso outro agente optasse por este caminho, ele não estaria mais disponível, forçando a escolha de outra rota completamente contrária.

Na tabela 4.2 podemos observar qual foi a taxa de FPS para cada algoritmo, o gráfico gerado a partir destes dados está na figura 4.8. Nesta simulação tanto o OpenSteer quanto o Voronoi apresentaram resultados visualmente ruins. O A* conseguiu concluir todos os caminhos e com uma taxa de FPS maior do que os outros.

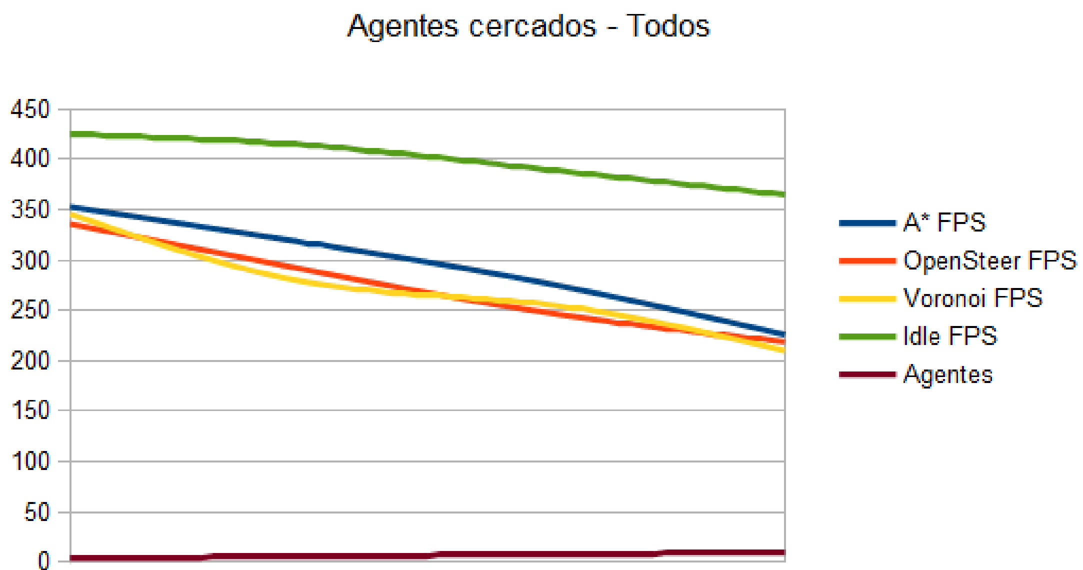


Figura 4.8: Gráfico completo da segunda simulação.

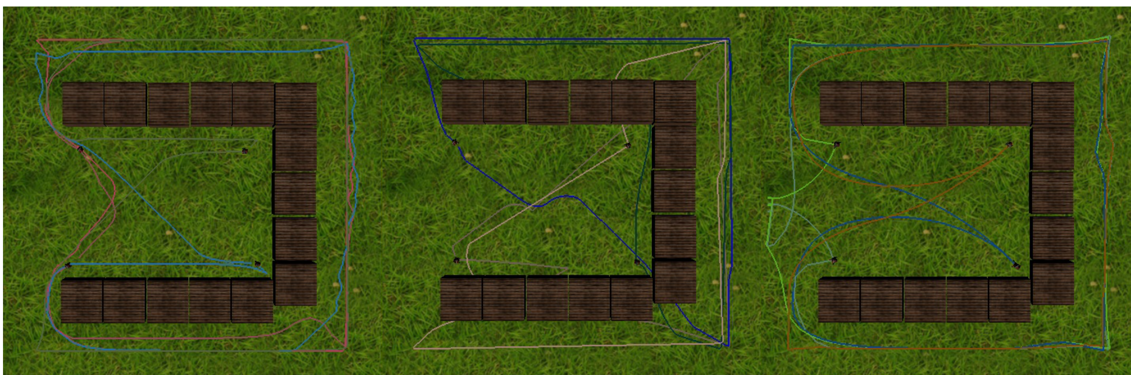


Figura 4.9: Rastros gerados na segunda simulação com 4 agentes.

O Opensteer apresentou seu problema já conhecido de ignorar obstáculos em algumas situações. Eu tentei aumentar o raio e prioridade da detecção de obstáculos, mas não houve melhoria. Também tentei mudar o cenário, colando as caixas para ver se o problema não seria que ele detectava uma rota entre elas, mas isto também não mudou em nada o resultado final.

Já o Voronoi apresentou o mesmo problema de os agentes ficarem presos, mas desta vez no canto do mundo e muito perto de um obstáculo. As simulações para seis, oito e dez agentes não foram concluídas, pois os agentes ficaram presos sem mudar em nada a posição por mais de cinco minutos – neste ponto eu cancelei a simulação.

Pode-se ver os rastros que cada algoritmo gerou na figura 4.9 para 4 agentes, 4.10 para seis agentes, 4.11 para oito agentes e 4.12 para dez agentes. Em todas elas a ordem é A* no lado esquerdo, OpenSteer no centro e Voronoi no lado direito.

Analisando os rastros podemos reparar facilmente que o OpenSteer ignorou os obstáculos em todos os testes. Já no caso do Voronoi podemos observar os agentes presos (retângulos azuis) nas figuras 4.10, 4.11 e 4.12. Um detalhe interessante é que em certo ponto um agente controlado pelo Voronoi resolveu mudar completamente de rota (retângulo vermelho, na figura 4.10), conforme inicialmente foi prevista a possibilidade. Por

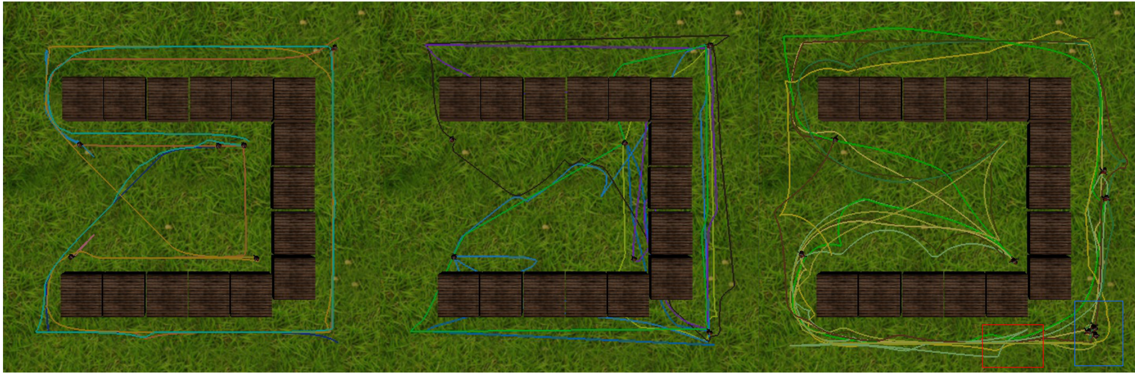


Figura 4.10: Rastros gerados na segunda simulação com 6 agentes.

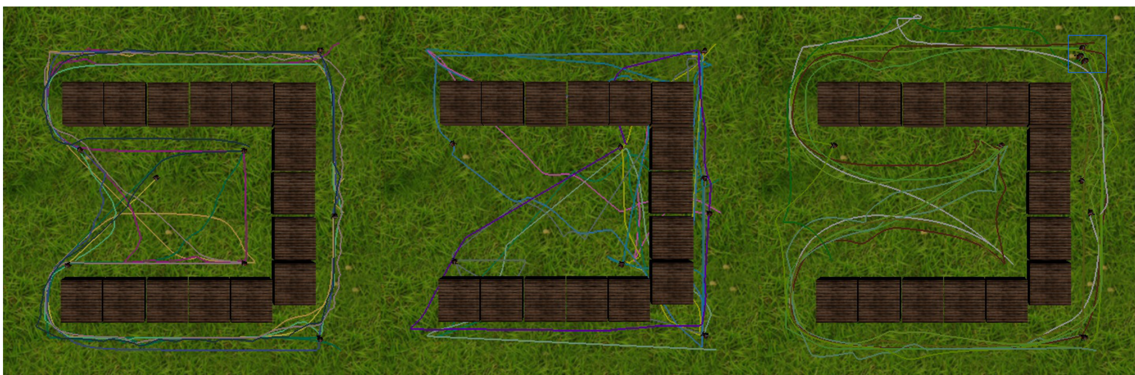


Figura 4.11: Rastros gerados na segunda simulação com 8 agentes.

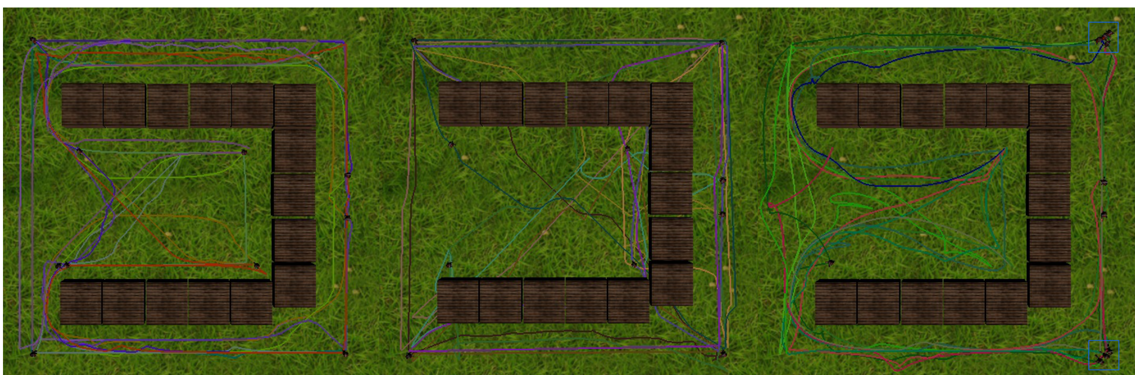


Figura 4.12: Rastros gerados na sagunda simulação com 10 agentes.

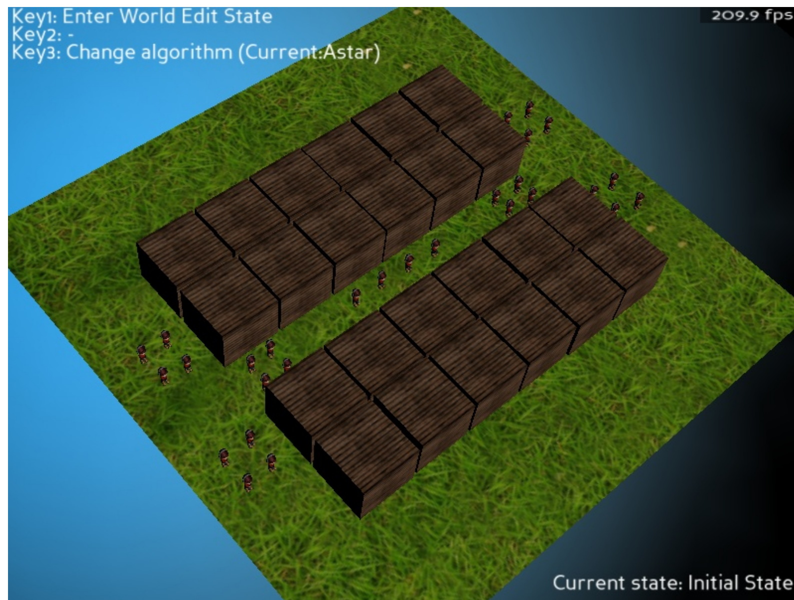


Figura 4.13: Estado inical da terceira simulação com 32 agentes.

Agents	FPS Idle	FPS OpenSteer	FPS A*	FPS Voronoi
8	292	226	225	202
16	244	166	153	123
24	190	94	121	78
32	189	82	84	59

Tabela 4.3: Simulação 3 – Resultados.

fim, podemos ver que os traços gerados pelo A* demonstram novamente algumas curvas pouco atenuadas.

4.6 Agentes em um Túnel

Nesta simulação o cenário terá duas paredes separadas que formam um túnel entre elas. A primeira situação será representada por dois grupos de quatro agentes, um em cada extremidade do túnel, cujo objetivo é alcançar o outro lado dele. O segundo e o terceiro teste adicionam grupos do lado de fora do túnel, também precisando chegar no outro lado. O último caso apresentará mais um grupo, desta vez dentro do túnel, com o objetivo de sair dele. A figura 4.13 mostra o estado inicial da simulação no caso com dez agentes

Esta simulação tem como objetivo testar como os agentes se comportam quando tem que escolher uma rota possivelmente ocupada. Também é um teste em que o Voronoi pode demonstrar mais de seu potencial, pois conforme foi observado no último teste, ele não funciona bem quando seus agentes precisam escolher rotas muito próximas aos limites do mundo.

Na tabela 4.3 podemos observar qual foi a taxa de FPS para cada algoritmo, o gráfico gerado a partir destes dados está na figura 4.14. Nesta simulação somente o Voronoi não apresentou nenhum erro grave, já os outros dois algoritmos tiveram problemas com os obstáculos. No caso do OpenSteer os agentes se moveram por dentro das paredes quando confinados dentro do túnel. No teste com o A*, dois agentes entraram na parede

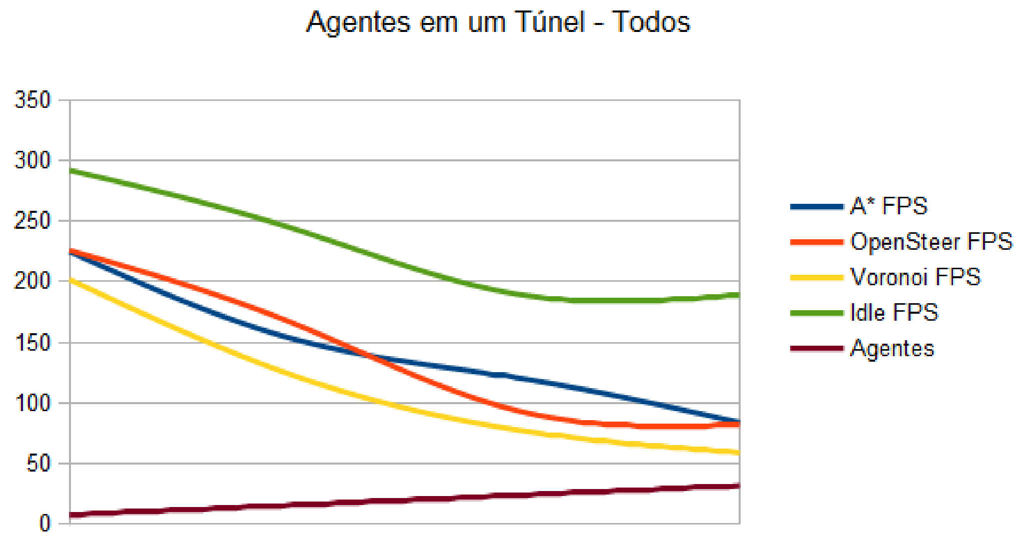


Figura 4.14: Gráfico completo da terceira simulação.

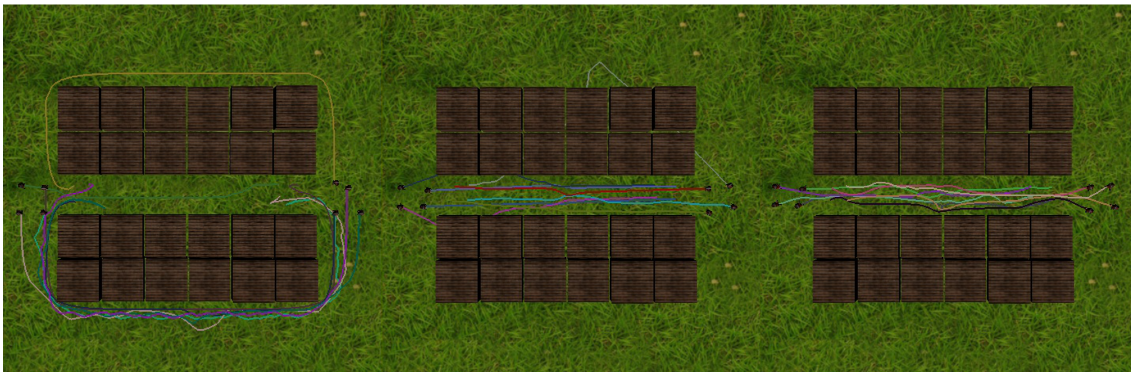


Figura 4.15: Rastros gerados na terceira simulação com 8 agentes.

e ficaram presos nela até o fim da simulação durante o caso com oito grupos (o algoritmo consegue detectar que os agentes estão presos, mas nenhuma solução foi implementada para contornar isto). O Voronoi apresentou um problema no qual dois agentes andaram juntos por bastante tempo, sendo que o objetivo de um deles era a direção oposta, no entanto eles conseguiram se desprender e concluir seus caminhamentos.

Pode-se ver os rastros que cada algoritmo gerou na figura 4.15 para 8 agentes, 4.16 para 16 agentes, 4.17 para 24 agentes e 4.18 para 32 agentes. Em todas elas a ordem é A* no lado esquerdo, OpenSteer no centro e Voronoi no lado direito.

Analisando os rastros podemos notar facilmente que o OpenSteer ignorou os obstáculos em todos os testes. No caso do A* a maioria dos agentes escolheu passar pelo lado de fora do túnel, um caminho muito maior, pois identificaram que a rota por dentro dele estava bloqueada. No último teste (figura 4.18), o retângulo vermelho mostra os agentes (controlados pelo A*) presos e os azuis esboçam quais as suas posições finais que eles deveriam ter atingido. Por fim, o rastro produzido pelo Voronoi mostra um resultado muito

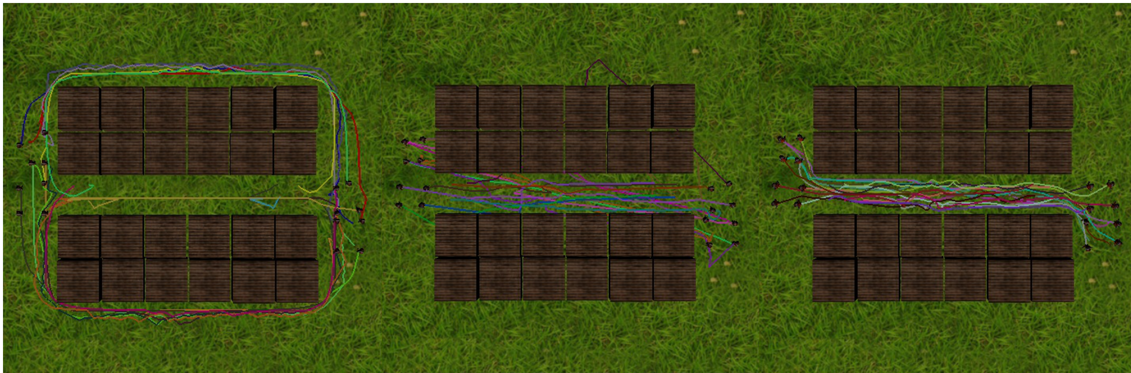


Figura 4.16: Rastros gerados na terceira simulação com 16 agentes.

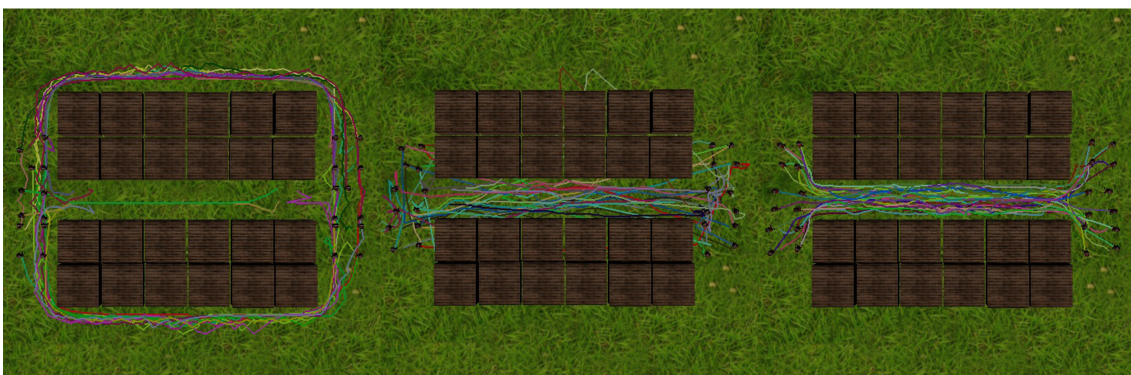


Figura 4.17: Rastros gerados na terceira simulação com 24 agentes.

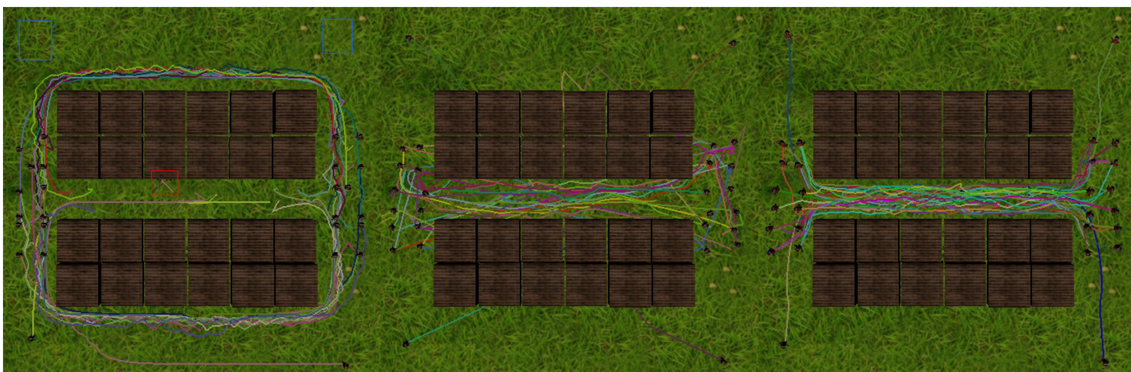


Figura 4.18: Rastros gerados na terceira simulação com 32 agentes.

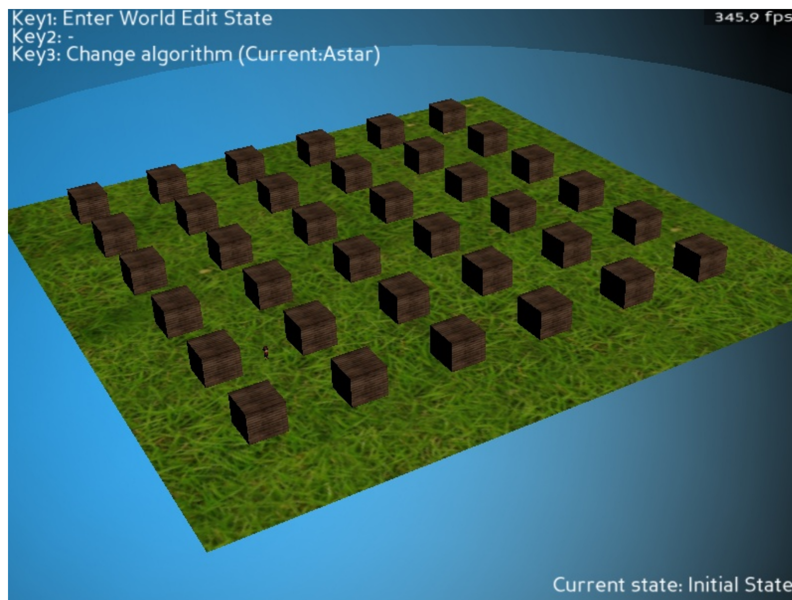


Figura 4.19: Estado inicial da quarta simulação.

Objectives	FPS Idle	FPS OpenSteer	FPS A*	FPS Voronoi
125	325	330	330	306
250	325	335	324	305
375	325	330	318	305
500	325	335	317	309

Tabela 4.4: Simulação 4 – Resultados.

bom pois todos os agentes escolheram a rota mais curta (por dentro do túnel), o único problema foi os agentes andando juntos que não esboçaram naturalidade.

4.7 Agente com Muitos Objetivos

Nesta simulação o cenário terá 36 caixas e somente um agente com muitos objetivos. Os testes foram executados para 125, 250, 375 e 500 pontos. A figura 4.19 mostra o estado inicial da simulação.

O Objetivo desta simulação é testar se alguma das técnicas perde desempenho quando um agente possui um caminho muito grande ou quando utilizados por muito tempo. A duração dos testes foi bastante longa, sendo que os casos mais extensos levaram mais de 10 minutos para acabar.

Na tabela 4.4 podemos observar como foi a taxa de FPS para cada algoritmo, o gráfico gerado a partir destes dados está na figura 4.19. Como o esperado, a variação de todos os algoritmos foi muito pequena. Opensteer e Voronoi apresentaram variação máxima de aproximadamente 2% e o A* sofreu perda máxima de aproximadamente 4%.

4.8 Simulação com Muitos Agentes

Nesta simulação o cenário terá 36 obstáculos e um número variável, porém grande de agentes. Os testes foram feitos para 25, 50, 75 e 100 agentes, com 20 objetivos aleatoriamente distribuídos. Os testes foram executados com um limitante de tempo de cinco

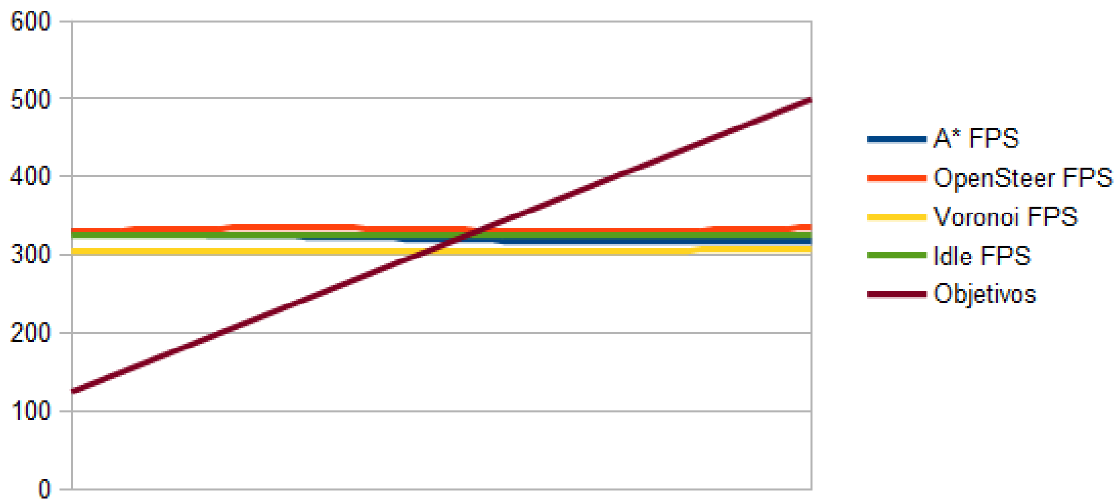


Figura 4.20: Gráfico completo da quarta simulação.

Agents	FPS Idle	FPS OpenSteer	FPS A*	FPS Voronoi
25	184	76	68	65
50	124	19	15	20
75	97	13	5	18
100	77	10	2	12

Tabela 4.5: Simulação 5 – Resultados.

minutos, pois a taxa de FPS em de alguns foi tão baixa que a conclusão dos maiores exemplos certamente levariam horas. A figura 4.21 mostra o estado inicial da cena com 100 agentes.

O objetivo desta simulação é testar ao máximo a performance de todos os algoritmos e verificar como cada um deles lida com uma escala realmente grande - no caso mais complexo cada agente tem 135 obstáculos para desviar no mundo. Para gerar as simulações foi gerado um conjunto de 104 pontos válidos no mundo do qual foram sorteadas posições iniciais únicas e objetivos aleatórios para os agentes. Para evitar que dois agentes ficassem presos no fim da simulação – por possuírem o mesmo objetivo – o último ponto do caminho que cada um seguiria era o mesmo que sua posição inicial.

Na tabela 4.5 podemos observar como foi a taxa de FPS para cada algoritmo, o gráfico gerado a partir destes dados está na figura 4.22. Neste teste podemos ver o Voronoi atuando em sua melhor área, mundos grandes com muitas mudanças. Infelizmente no caso de 75 agente ele apresentou um problema de memória dentro de sua Quadtree, terminando-o com menos de um minuto de execução, neste caso o dado da coluna e do gráfico são do tempo pelo qual ele rodou.

Neste ambiente o Opensteer ignorou novamente os obstáculos, é fácil reparar que isso acontece muito mais quando existem mais agentes no mundo. O A* demonstrou uma performance aceitável para o primeiro teste, mas nos demais decaiu completamente, chegando a 2 FPS no caso limite. O Voronoi demonstrou problemas de agentes de presos, desta vez é muito provável que o problema foi que os agentes tinham exatamente o mesmo ponto como objetivo e eles ficaram presos tentado alcançá-lo. A figura 4.23 mostra os

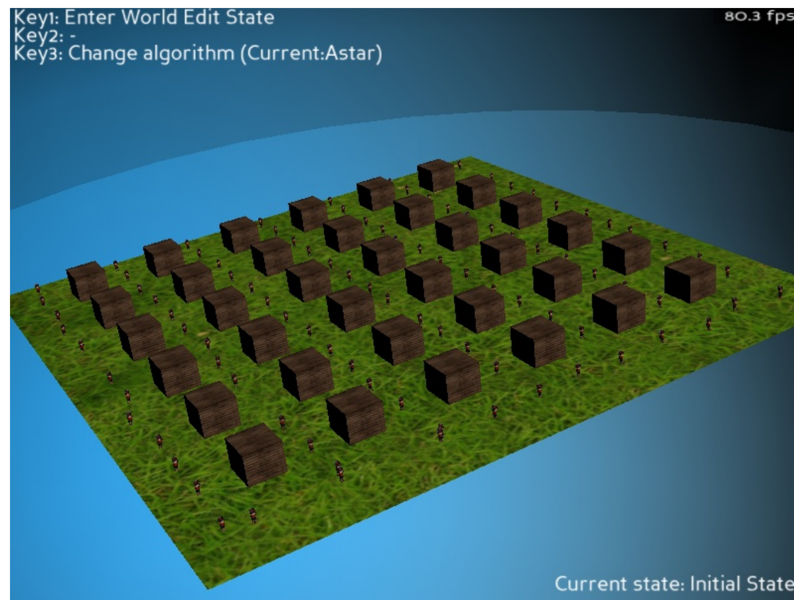


Figura 4.21: Estado inicial da quinta simulação.

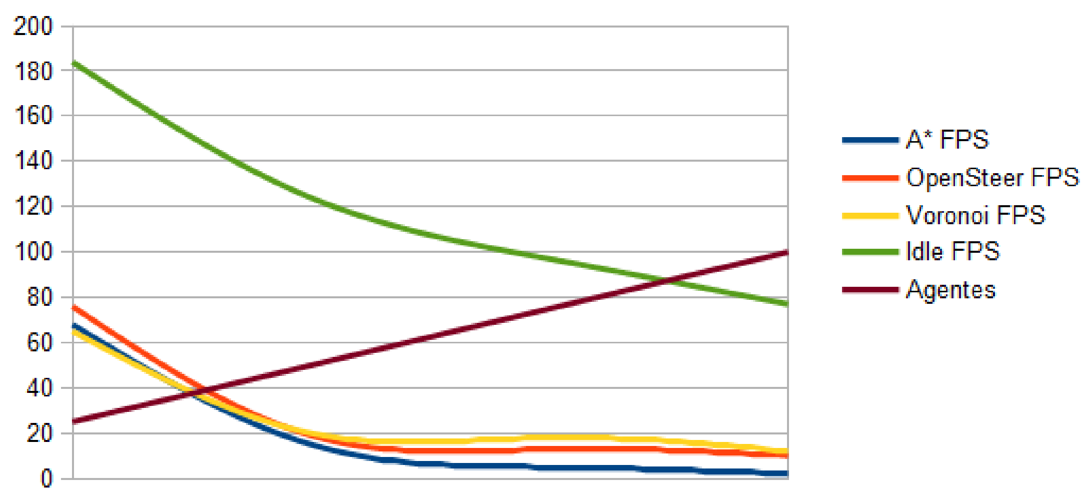


Figura 4.22: Gráfico completo da quinta simulação.

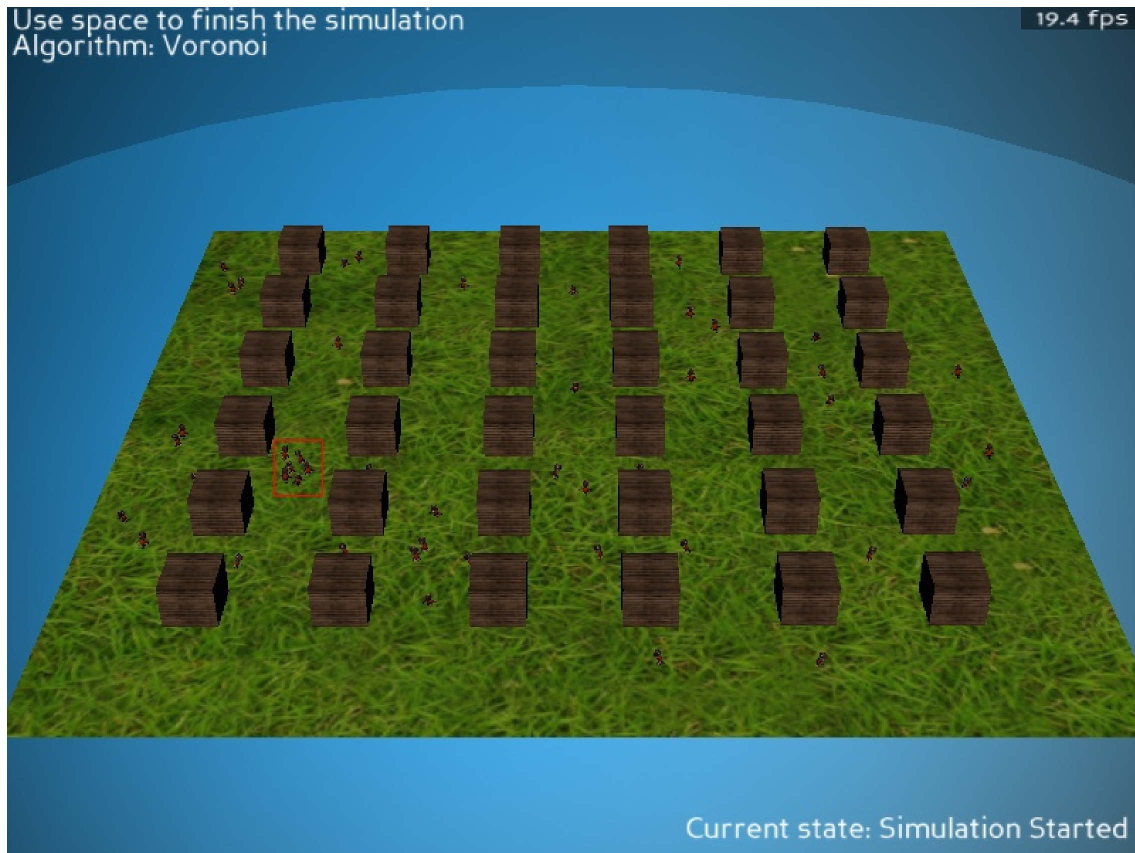


Figura 4.23: Agentes presos tentando chegar na mesma posição.

agentes trancados durante uma simulação (retângulo vermelho).

5 CONCLUSÕES E TRABALHOS FUTUROS

5.1 Conclusões

Neste trabalho testou-se o algoritmo de path-planning baseada em Voronoi, focando principalmente a comparação com outras técnicas utilizadas no mercado. Para tal criou-se um gerador de cenas dinâmicas e nele foram criados diversos cenários, sempre focados em forçar situações nas quais os algoritmos seriam testados ao máximo na questão de performance e induzidos a gerar erros.

Foram criados alguns testes para observar onde os algoritmos erravam e, a partir destes resultados gerou-se as cinco simulações apresentadas. Durante os testes sempre foram observados os buffers de comunicação e o consumo percentual de cada núcleo do processador para manter os testes similares para obter resultados mais confiáveis possíveis.

Na tabela 5.1 existe uma consideração entre os métodos testados e cada simulação. Onde os resultados são bons os algoritmos conseguiram resolver o problema sem precisar de nenhum ajuste. Nas colunas onde o resultado é dito médio as técnicas não resolveram o problema de forma satisfatório, mas os problemas não são inerentes a técnica e podem ser corrigidos. Quando os resultados são ditos ruins a técnica não consegue lidar com a situação de forma satisfatória.

Os maiores problemas encontrados na técnica baseada em Voronoi é que os agentes podem ficar presos e ela não consegue detectar isto. Por mais que esta situação também ocorra no A*, ele a identifica. Por fim, existe também o erro dos agentes andando juntos, tentando se livrar um do outro. Ainda que ele não seja terminal, ele acaba com a naturalidade do caminho, que é a maior vantagem da técnica baseada em Voronoi.

Após a comparação dos resultados e análise visual a técnica pareceu em bom estado para competir com as demais. O OpenSteer apresentou performance melhor, mas ele não garante que os obstáculos serão evitados em todos os casos. Ele realmente pode apresentar resultados melhores em casos simples, mas não pode ser utilizado como solução geral. Embora o A* tenha apresentado performance melhor nos casos mais simples, ele não tem o caminho filtrado para parecer natural. Em questão de escalabilidade, o Voronoi se

Simulação	OpenSteer	A*	Voronoi
1	Bom	Bom	Bom
2	Ruim	Bom	Médio
3	Ruim	Médio	Bom
4	Médio	Bom	Bom
5	Médio	Ruim	Médio

Tabela 5.1: Comparação entre os resultado dos métodos.

mostrou o melhor de todos, como os outros foi estável em simulações sem variações e ele foi o único que escolheu sempre a menor rota.

Ainda assim a técnica ainda precisa de mais algumas adições. Durante a realização deste trabalho encontrou-se um bug de implementação (referente a implementação da quadtree), problemas de personagens com o mesmo objetivo, problemas no algoritmo de filtragem de caminho nas bordas do mundo e, em casos raríssimos, os agentes ficavam presos, forçando um deles a andar na direção oposta do seu caminho original.

5.2 Trabalhos Futuros

Seria interessante estudar técnicas especiais de desvio de agentes para o algoritmo baseado em Voronoi, que poderiam ser ativadas em caso de proximidade do fim do mundo, já que – conforme os resultados deste trabalho – esta é a situação em que o algoritmo demonstra o pior resultado. A técnica também precisaria tratar a situação em que o personagem controlado está muito perto de obstáculos e de outros agentes, pois neste caso o algoritmo apresentou resultados ruins também. Para este trabalho futuro as simulações desenvolvidas neste trabalho poderiam ser utilizadas para testar o novo método, sendo assim, o trabalho poderia focar somente em resolver o problema, sem preocupações com implementações de cenários.

Outra alteração que seria importante é a implementação de uma técnica mais inteligente para determinar em qual ângulo está a frente do personagem. O método que usado neste trabalho é extremamente simples e pouco natural. Seria interessante se o personagem tivesse um limite do quanto ele pode girar a cada segundo, assim como se pudesse tomar a decisão de andar de costas ou de lado (sem alterar a orientação do modelo) por distâncias curtas – afinal, é assim que os humanos normalmente se comportam.

Por fim, desenvolver um método de detectar situações que o personagem está preso. Erros deste tipo sempre podem acontecer, principalmente quando se adiciona física ou movimentações em múltiplos eixos no jogo. Existem algumas técnicas de como tratar estas situações de erro em [2], além disso, outras novas podem ser adicionadas e testadas.

REFERÊNCIAS

[1] Blizzard Entertainment. Diablo 1 - 10 ways to kill the butcher, 1996. Disponível em: *http : //theory.stanford.edu/ amitp/GameProgramming/*. Acesso em: junho 2012.

[2] J. Ahlquist and J. Novak. *Game development essentials: game artificial intelligence*. Game Development Essentials Series. Thomson/Delmar Learning, 1st edition, 2007.

[3] Paul Tozour. Game/ai: Fixing pathfinding once and for all:, 2008. Disponível em: *http : //www.ai − blog.net/archives/000152.html*. Acesso em: junho 2012.

[4] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] Amit Patel. Amit's a* page, 2010. Disponível em: *http : //theory.stanford.edu/ amitp/GameProgramming/*. Acesso em: junho 2012.

[6] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[7] Subhrajit Bhattacharya. File:astar progress animation.gif wikipedia, the free encyclopedia:, 2011. Disponível em: *http : //en.wikipedia.org/wiki/File : Astar_progress_animation.gif*. Acesso em: junho 2012.

[8] Subhrajit Bhattacharya. File:dijkstras progress animation.gif - wikipedia, the free encyclopedia:, 2011. Disponível em: *http : //en.wikipedia.org/wiki/File : Dijkstras_progress_animation.gif*. Acesso em: junho 2012.

[9] C. W. Reynolds. Steering behaviors for autonomous characters. *In the proceedings of Game Developers Conference 1999*, pages 763–782, 1999.

[10] Francisco Pinto and Carla Freitas. Dynamic voronoi diagram of complex sites. *The Visual Computer*, 27:463–472, 2011. 10.1007/s00371-011-0581-z.

[11] Unity Technologies. Unity - pathfinding - unity 3 engine features, 2012. Disponível em: *http : //unity3d.com/unity/engine/pathfinding*. Acesso em: junho 2012.

[12] CRYTEK. Advanced modular ai system, 2012. Disponível em: *http : //www.crytek.com/cryengine/cryengine3/overview*. Acesso em: junho 2012.

[13] Epic Games. Game ai artificial inteligente, 2012. Disponível em: http : //http : //www.unrealengine.com/features/artificial_intelligence/. Acesso em: junho 2012.

[14] Valve Corperation. Source - game mechanics, 2012. Disponível em: http : //source.valvesoftware.com/gamemechanics.php. Acesso em: junho de 2012.

[15] PathEngine. Pathengine - intelligent agent movement, 2012. Disponível em: http : //www.pathengine.com/overview.php. Acesso em: junho de 2012.

[16] CRYTEK. Pathfinding - doc 2. sandbox manual - cryengine 3 free sdk, 2012. Disponível em: http : //freesdk.crydev.net/display/SDKDOC2/Pathfinding. Acesso em: junho de 2012.

[17] Carnegie Mellon University. Panda3d - free game engine, 2010. Disponível em: http : //www.panda3d.org/. Acesso em: junho de 2012.

[18] Carnegie Mellon University. Distributing via the web - panda3d manual, 2010. Disponível em: http : //www.panda3d.org/manual/index.php/$Distributing_via_the_web$. Acesso em: junho de 2012.

[19] Mike Goslin and Mark R. Mine. The panda3d graphics engine. *Computer*, 37:112–114, 2004.

[20] Carnegie Mellon University. Panda3d license, 2008. Disponível em: http : //www.panda3d.org/license.php. Acesso em: junho de 2012.

[21] Jacob Moen. Getting started, 2011. Disponível em: http : //www.ogre3d.org/tikiwiki/tiki − index.php?page = Getting + Started#Is_Ogre_a_game_engine_. Acesso em: junho de 2012.

[22] Nikolaus Gebhardt. Faq - irrlicht engine - a free open source 3d engine:, 2012. Disponível em: http : //irrlicht.sourceforge.net/faq/#gameengine. Acesso em: junho de 2012.

[23] Carnegie Mellon University. Panda3d manual: Features, 2012. Disponível em: http : //www.panda3d.org/manual/index.php/Features. Acesso em: junho de 2012.

[24] Carnegie Mellon University. Sound - panda3d manual, 2010. Disponível em: http : //www.panda3d.org/manual/index.php/Sound. Acesso em: junho de 2012.

[25] Carnegie Mellon University. Gallery of screenshots, 2010. Disponível em: http : //www.panda3d.org/screens.php. Acesso em: junho de 2012.

[26] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Documentation - pandai, 2010. Disponível em: https : //sites.google.com/site/etcpandai/documentation. Acesso em: junho de 2012.

[27] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Research - pandai, 2010. Disponível em: *https : //sites.google.com/site/etcpandai/what − is − pandai/research*. Acesso em: junho de 2012.

[28] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Research - pandai, 2010. Disponível em: *https : //sites.google.com/site/etcpandai/documentation/pathfinding/mesh − generator*. Acesso em: junho de 2012.

[29] Carnegie Mellon University. Getting started - panda3d manual, 2010. Disponível em: *http : //www.panda3d.org/manual/index.php/Getting*$_S$*tarted*. Acesso em: junho de 2012.

[30] Carnegie Mellon University. Pathfinding - panda3d manual, 2010. Disponível em: *http : //www.panda3d.org/manual/index.php/Pathfinding*. Acesso em: junho de 2012.

[31] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Seek - pandai, 2010. Disponível em: *https : //sites.google.com/site/etcpandai/documentation/steering − behaviors/seek*. Acesso em: junho de 2012.

[32] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Flee - pandai, 2010. Disponível em: *https : //sites.google.com/site/etcpandai/documentation/steering − behaviors/flee*. Acesso em: junho de 2012.

[33] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Pursue - pandai, 2010. Disponível em: *https : //sites.google.com/site/etcpandai/documentation/steering − behaviors/pursue*. Acesso em: junho de 2012.

[34] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Evade - pandai, 2010. Disponível em: *https : //sites.google.com/site/etcpandai/documentation/steering − behaviors/evade*. Acesso em: junho de 2012.

[35] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Path follow - pandai, 2010. Disponível em: *https : //sites.google.com/site/etcpandai/documentation/steering − behaviors/path − follow*. Acesso em: junho de 2012.

[36] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Wander - pandai, 2010. Disponível em: *https : //sites.google.com/site/etcpandai/documentation/steering − behaviors/wander*. Acesso em: junho de 2012.

[37] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Obstacle avoidance - pandai, 2010. Disponível em: *https :*

$//sites.google.com/site/etcpandai/documentation/steering - behaviors/obstacle - avoidance$. Acesso em: junho de 2012.

[38] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Arrival - pandai, 2010. Disponível em: $https : //sites.google.com/site/etcpandai/documentation/steering - behaviors/arrival$. Acesso em: junho de 2012.

[39] Kyle Dolan, Deepak Murali Chandrasekaran, John Kolencheryl, Srinavin Nair, Jy-Huey Lin, Mike Christel, and Ruth Comley. Flock - pandai, 2010. Disponível em: $https : //sites.google.com/site/etcpandai/documentation/steering - behaviors/flock$. Acesso em: junho de 2012.

[40] Microsoft. Creating named shared memory, 2012. Disponível em: $http : //msdn.microsoft.com/en - us/library/windows/desktop/aa366551.aspx$. Acesso em: junho de 2012.

[41] Ion Gaztanaga. Chapter 12 boost interprocess boost 1 49 0, 2012. Disponível em: $http : //www.boost.org/doc/libs/1_4 9_0/doc/html/interprocess.html$. Acesso em: junho de 2012.