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

LEONARDO GARCIA FISCHER

Otimização de Desempenho em Planejadores de Caminho Usando Campos Potenciais

Trabalho de Graduação

Prof[a]. Dra. Luciana P. Nedel
Orientador

MSc. Renato Silveira
Co-orientador

Porto Alegre, novembro de 2008.

AGRADECIMENTOS

Gostaria de agradecer a todos aqueles que contribuíram com algo de bom para mim, durante todos os anos em que estive na graduação. O Leonardo que aqui apresenta o seu trabalho de graduação é o resultado de um somatório de muitas coisas boas, vindas de muitas pessoas diferentes.

Agradeço à minha família, sem os quais eu não teria tido condições de chegar até aqui. Todo o alicerce em que me sustento é obra de Vilson, Regina, Victor e Beatriz. Em especial, agradeço ao meu pai, por ter me incentivado a estudar e passar no vestibular. Sem o seu incentivo, não sei se teria passado no vestibular e continuado meus estudos.

Agradeço à minha namorada, Francele. A mais de três anos, ela tem me dito palavras de conforto nos momentos difíceis, e palavras muito divertidas nos momentos mais alegres. Com certeza, minha vida mudou para muito melhor depois que eu a conheci.

Agradeço a dois amigos especiais, Jefferson e Renato. Apesar de eu ter deixado de acompanhá-los por muitas vezes para poder estudar (sim, foram muitas), eles continuam sendo meus melhores amigos. Também agradeço a todos os amigos que fiz na faculdade. São tantos que não vou citar os nomes. Basta dizer que fomos uma turma muito mais unida do que eu poderia imaginar nos meus melhores sonhos.

Também quero agradecer a todos os professores do Instituto de Informática da UFRGS. Sinto-me um grande profissional neste fim de curso, e ainda assim acho que sei apenas uma ínfima parte daquilo que cada um tem para oferecer. A todos os professores com quem tive aula, sem nenhuma exceção, meu muito obrigado.

Mas em especial, agradeço aos professores do grupo de computação gráfica. Coincidência ou não, todos me proporcionaram momentos muito especiais. À professora Luciana Nedel, obrigado pelos primeiros passos na área de CG, e por me acompanhar neste trabalho de conclusão. À professoral Carla dal Sasso Freitas, obrigado pelo ano e meio de bolsa de IC. Ao professor João Comba, obrigado pela disciplina onde projetei um jogo por completo, um sonho que tinha desde os 7 anos de idade. E ao professor Manuel Menezes de Oliveira Neto, obrigado pela incrível empolgação científica que demonstrou em suas aulas (não posso deixar de citar a apresentação de um episódio do seriado “Cosmos” em sua aula, ato que considero o “divisor de águas” na minha vida acadêmica).

Por fim, agradeço ao Instituto de Informática pela estrutura oferecida. Na minha humilde opinião, a estrutura de prédios, salas de aula, laboratórios e biblioteca, é apenas o reflexo da altíssima qualidade do curso de Ciência da Computação da UFRGS, referência no Brasil e reconhecido internacionalmente.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CUDA	Compute Unified Device Architecture
HTML	Hyper Text Markup Language
GPU	Graphics Processor Unit
OGRE	Open Graphics Rendering Engine
PDF	Portable Document Format
PVC	Problema de Valor de Contorno
SOR	Successive Over-Relaxation
SVN	Subversion

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A animação de personagens autônomos interessa a muitas áreas, tais como filmes, jogos e simulações. O seu principal objeto de estudo é a definição de maneiras de reproduzir, de forma realística, um dado ser (seja ele vivo, mecânico ou mesmo imaginário). Diversas técnicas já foram propostas tentando se alcançar alta qualidade na animação final. Porém, percebe-se que essas propostas são cada vez mais complexas, dificultando o seu uso em aplicações de tempo real.

Dessa forma, o desempenho do algoritmo que executa a animação pode ser tão importante quanto à qualidade da própria animação. Uma técnica que gere resultados de alta qualidade, mas que não possa ser executada em tempo real será bastante limitada quanto à sua aplicabilidade.

Nesse sentido, esse trabalho apresenta o estudo de um desses algoritmos, apresentando uma forma de melhorar o seu desempenho, visando o seu uso em aplicações de tempo real. Esse ganho de desempenho será alcançado utilizando as características paralelizáveis do algoritmo estudado.

Para alcançar esse objetivo, este trabalho utiliza o alto paralelismo das placas gráficas atuais para estender o trabalho iniciado por Dapper (DAPPER et al., 2007). Em seu trabalho, Dapper utilizou os campos potenciais gerados a partir da solução numérica de problemas de valores de contorno envolvendo a equação de Laplace (funções harmônicas). Esses campos potenciais são capazes de gerar caminhos livres de mínimos locais, com trajetórias suaves, e por isso foram utilizados para gerar o caminho a ser seguido pelo agente autônomo. Como será visto, esses campos potenciais podem ser obtidos mais rapidamente se os cálculos envolvidos forem paralelizados.

Também será apresentado uma série de testes que demonstram que a implementação paralela do algoritmo é capaz de melhorar o seu desempenho. Nos testes, foi possível detectar que a nova implementação é capaz de ser até 56 vezes mais rápida que a versão seqüencial. Isso torna o algoritmo aplicável para uso em tempo real, mesmo em situações com centenas de personagens autônomos na cena.

Por fim, será apresentada uma série de outras contribuições feitas ao projeto. Entre essas, destaca-se um sistema de ramificação de trajetórias para definir caminhos, e a melhoria da qualidade do código existente no início dos trabalhos, como forma de melhorar os trabalhos futuros.

Palavras-Chave: Planejamento de Caminhos, GPGPU, CUDA, Paralelismo, Computação Gráfica

ABSTRACT

The autonomous agent animation concerns many areas, such as movies, games and simulations. Its main objective is how to define methods of reproducing a certain being in a realist way (be it alive, mechanic or even imaginary). Several techniques have already been proposed in order to achieve high quality in the final animation. However, we can notice that these proposals are more and more complex, which difficults its use in real time applications.

Thus, the algorithm's performance which accomplishes the animation can be as important as the quality of the animation itself. A technique which produces high quality results, though it cannot be performed in real time, will be limited enough when it comes to applicability.

Therein, this work shows the study of one of these algorithms, featuring a way of improving its performance and aiming at using it in real time applications. This performance advantage will be achieved by using parallel characteristics of the algorithm.

Then, this work uses the actual video boards' high parallelism to extend Dapper's work (DAPPER et al., 2007). In his work, Dapper used the potential fields developed from a numerical solution of boundary value problems involving the Laplace's equation (harmonic functions). These potential fields can create paths with mild courses, free of local minimun. Because of this they were used to produce the path to be followed by the autonomous agent. As we will see, these potential fields can be achieved much more quickly if we parallelize the calculations involved in.

It will be presented as well a series of tests which showed that the algorithm's parallel implementation can improve its performance. The tests showed that the new implementation is up to 56 times faster than the sequential implementation. This makes the algorithm applicable for use in real time, even in situations with thousands of autonomous agents in the scene.

Finally, a set of other contributions made to the project will be showed, among which two of them stand out: a system of ramification of paths to define routes and the improvement of the already existent code as a form of enriching later works.

Keywords: Path Planning, GPGPU, CUDA, Parallelism, Computer Graphics

1 INTRODUÇÃO

A animação de personagens virtuais é de grande interesse para diversas áreas da computação. A produção de filmes animados, os jogos de computador e simulações de fugas de emergência em edificações são exemplos de aplicação prática desse tipo de animação. Em todos esses casos, é perceptível o interesse de que a animação seja o mais realista possível, seja na simulação de uma pessoa caminhando no shopping, um robô que caminha em um ambiente hostil, ou um monstro imaginário em um filme infantil.

Diversas técnicas já foram propostas, e muitas ainda estão em estudo, com o objetivo de gerar animações de personagens autônomos navegando em ambientes virtuais. Essas técnicas envolvem a especificação do ambiente, a posição inicial do agente virtual e o objetivo a ser alcançado. Essas técnicas também podem envolver outras variáveis particulares a cada agente, tais como seu humor, personalidade, disposição, etc.

Um problema que surge com a sofisticação dessas técnicas é a quantidade de processamento necessário para realizá-las. A quantidade de variáveis e a forma como elas são utilizadas nos cálculos são decisivas para o desempenho do algoritmo. Em alguns casos, a qualidade da animação é mais importante que a quantidade de cálculos necessários para gerá-la. Por exemplo, um filme pode ser gerado quadro a quadro, e os quadros resultantes serem combinados em um segundo momento. Mas em um jogo de computador, um algoritmo muito complexo pode deteriorar sensivelmente a sua jogabilidade, reduzindo drasticamente a qualidade final do produto.

Com essa preocupação, este trabalho irá apresentar o algoritmo proposto por Dapper (DAPPER et al., 2007) para o planejamento de movimentos para personagens virtuais. E como resultado, será demonstrado como esse algoritmo se beneficia do paralelismo disponível no hardware gráfico atual, permitindo que ele possa ser utilizado em aplicações em tempo real, com alto desempenho e sem perda de qualidade.

1.1 Motivação

Por um lado, novas técnicas para o controle de personagens virtuais estão surgindo, cada vez gerando melhores resultados. Mas por outro lado, o desempenho de uma solução é essencial para definir sua aplicabilidade. O balanceamento entre qualidade e desempenho sempre deve ser analisado ao estudar uma solução proposta.

Em um jogo de computador, por exemplo, o desempenho de uma técnica pode ser decisivo na escolha entre utilizá-la ou não. Existe a necessidade de se desenhar muitos quadros por segundo para que o jogador tenha a sensação de uma animação suave. Isso faz com que um algoritmo de planejamento de movimentos tenha apenas alguns poucos milissegundos por frame para ser executado. Assim, a melhoria do desempenho deve ser buscada a todo esforço ao se desenvolver uma aplicação de tempo real.

O planejamento de movimentos, conforme já citado, deve levar em conta muitas características particulares do agente em questão. Por exemplo, um indivíduo em um estado "cansado" dará preferência por percorrer o caminho mais curto entre dois pontos. Já um indivíduo em um estado "disposto" poderá percorrer um caminho maior, chegando mais perto de algum outro ponto ou caminho que lhe atraia mais. O algoritmo proposto por Dapper (DAPPER et al., 2007) mostrou-se promissor ao conseguir planejar movimentos com diversas características, tais como a citada no exemplo. Porém, o desempenho exposto nos resultados de seu trabalho pode comprometer o seu uso em aplicações de tempo real, com muitos agentes na cena.

1.2 Objetivos

Esse trabalho apresenta uma proposta de implementação paralela do algoritmo apresentado por Dapper (DAPPER et al., 2007), visando a melhoria do seu desempenho. Essa nova implementação irá se aproveitar do paralelismo disponível nas placas gráficas atuais, sendo executado em GPU. Os recursos das placas gráficas serão acessados através da linguagem de programação CUDA (CUDA, 2008), disponibilizada pela nVidia.

1.3 Organização do texto

Nos próximos capítulos, este trabalho será organizado da seguinte forma:

- O Capítulo 2 apresenta uma descrição das diversas técnicas utilizadas para planejar caminhos;
- O Capítulo 3 apresenta a técnica utilizada neste trabalho como base da nova implementação;
- O Capítulo 4 apresenta detalhes sobre o paralelismo existente na técnica, e apresenta a forma como esse paralelismo foi implementado, utilizando a linguagem de programação CUDA;
- O Capítulo 5 apresenta os resultados obtidos com os testes de desempenho da implementação paralela;
- O Capítulo 6 apresenta outras contribuições realizadas;
- O Capítulo 7 apresenta as conclusões obtidas neste trabalho, e apresenta algumas sugestões de trabalhos futuros.

2 PLANEJAMENTO DE CAMINHOS EM AMBIENTES VIRTUAIS

O planejamento de caminhos pode ser definido, de maneira informal, da seguinte forma: dada a posição inicial de um agente e a posição do seu objetivo, definir um caminho livre que leve o agente até esse objetivo. Esse planejamento deve levar em conta diversos fatores, como os obstáculos existentes na cena (tanto fixos quanto móveis), outros agentes que possam influenciar no caminho tomado, ou mesmo uma mudança de posição no próprio objetivo. O planejamento de caminhos também deve ser eficiente para ser executado em tempo real (permitindo assim a interferência do usuário). Por fim, deve resultar em um movimento convincente, para manter a sensação de naturalidade que o usuário tem ao assisti-lo.

Além da técnica que será descrita aqui, diversos outros trabalhos já foram realizados propondo técnicas para o planejamento de caminhos em ambientes virtuais. Dentre elas, podem ser destacadas as baseadas em scripts, em grades, em roadmaps e em campos potenciais. Nas próximas seções, essas técnicas serão apresentadas.

2.1 Planejamento Baseado em Scripts

No planejamento de caminhos baseado em scripts, os diversos caminhos possíveis de serem tomados são definidos previamente (em geral, durante a modelagem do cenário virtual, realizada por um artista de design). Esses caminhos são definidos manualmente, baseados na percepção do artista de quais caminhos são *bons* e quais são *ruins*.

Por serem definidos manualmente, o artista precisa dedicar um grande tempo desenvolvendo os scripts. Pelo mesmo motivo, são grandes as chances de um mesmo caminho ser tomado diversas vezes durante a execução da cena. Isso é bastante indesejável, visto que o usuário pode perceber essa repetição muito facilmente (NIEUWENHUISEN; KAMPHUIS; OVERMARS, 2007). E em casos onde vários agentes navegam em grupo, eles poderão precisar negociar espaço entre si para percorrer o caminho de forma natural.

Por fim, tais técnicas não são capazes de tratar obstáculos dinâmicos de forma trivial. Se um cenário virtual permite que esse tipo de obstáculo exista, outra abordagem ou extensão à técnica de scripts acaba sendo necessária.

2.2 Planejamento Baseado em Grades de Células

Nesse tipo de técnica, o ambiente virtual é inicialmente discretizado em um mapa de células. Essas células são então marcadas como livres ou ocupadas, representando os obstáculos existentes na cena. A partir daí, um caminho entre dois pontos pode ser en-

contrado utilizando um algoritmo de pesquisa nas células livres. Por exemplo, esses algoritmos podem ser baseados no *A** (DELOURA, 2000; RUSSELL; NORVIG, 2003).

A figura 2.1 exemplifica um caso de planejamento de caminhos utilizando grades de células. Nela, o agente se encontra inicialmente na célula azul, e irá percorrer o caminho formado pelas células verdes, até alcançar seu objetivo, na célula amarela. As regiões vermelhas representam obstáculos no cenário.

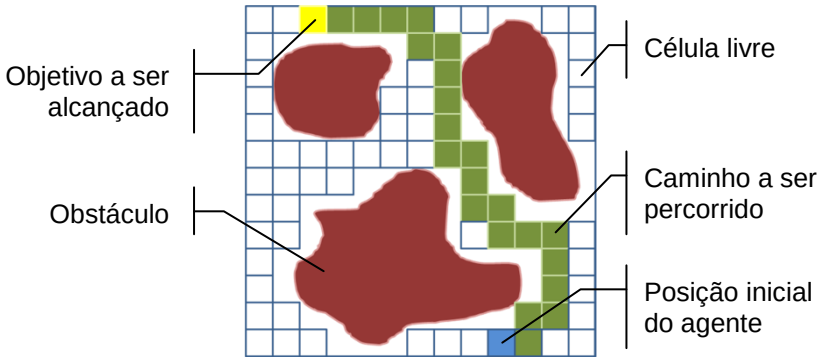


Figura 2.1: Exemplo de abordagem baseada em grades de células

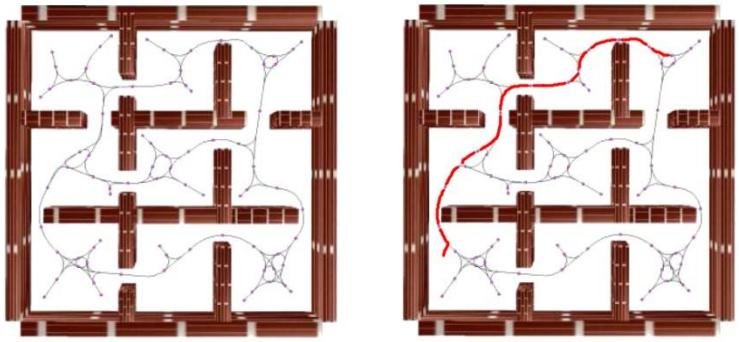
Essa técnica tende a apresentar bom desempenho em ambientes pequenos, com poucas células. Mas em ambientes grandes, com muitos obstáculos e agentes para se mover, esta técnica necessita de muita memória e grande quantidade de processamento. Heurísticas são capazes de reduzir o processamento necessário, mas podem gerar caminhos inválidos ou muito estereotipados.

Um dos problemas com essa técnica, é que ela tende a gerar caminhos pouco naturais. Os caminhos gerados podem ser suavizados, mas isso requer um processamento extra, aumentando o custo já existente na técnica.

2.3 Planejamento Baseado em Roadmaps

Os roadmaps constituem uma espécie de mapa dos possíveis caminhos que podem ser tomados no cenário virtual. Esse mapa é gerado em uma fase de pré-processamento, gerando um conjunto de vértices e curvas que os ligam. Pode ser feito uma analogia entre os roadmaps e um mapa de estradas de uma região. Durante a execução da animação, o agente autônomo precisa encontrar “a estrada” mais próxima do ponto onde se encontra atualmente, e então decidir qual percurso o levará para o ponto mais próximo do seu objetivo.

A seguir, a figura 2.2(a) ilustra o roadmap de um cenário virtual composto de diversas salas. Perceba que qualquer sala pode ser alcançada a partir de qualquer ponto, apenas seguindo uma rota nesse roadmap. Na figura 2.2(b), um caminho foi escolhido para se navegar entre duas salas no ambiente.



(a) (b)

Figura 2.2: Planejamento de caminhos baseados em roadmaps. (a) roadmap de uma cena. (b) escolha de um roadmap para alcançar um objetivo (NIEUWENHUISEN; KAMPHUIS; OVERMARS, 2007)

Os roadmaps possuem um custo de processamento bastante baixo, independente da complexidade do cenário virtual. Porém, a qualidade dos caminhos gerados é bastante dependente da etapa de pré-processamento. Uma possibilidade para melhorar a qualidade desses caminhos é incluir uma etapa de suavização para gerar os roadmaps.

Ainda assim, os roadmaps podem sofrer do mesmo problema descrito na técnica de scripts, onde vários agentes autônomos podem precisar negociar espaço para atingirem um objetivo em comum.

2.4 Planejamento Baseado em Campos Potenciais

Os campos potenciais foram propostos para planejar caminhos por Khatib (KHATIB, 1980). Nessa técnica, um campo potencial é criado a partir dos objetivos e obstáculos existentes no ambiente. Os obstáculos influenciam o campo potencial, criando uma força repulsiva. Já os objetivos influenciam no campo potencial criando uma força atrativa. Por fim, os agentes autônomos são influenciados por essas forças, sendo então guiados até o objetivo.

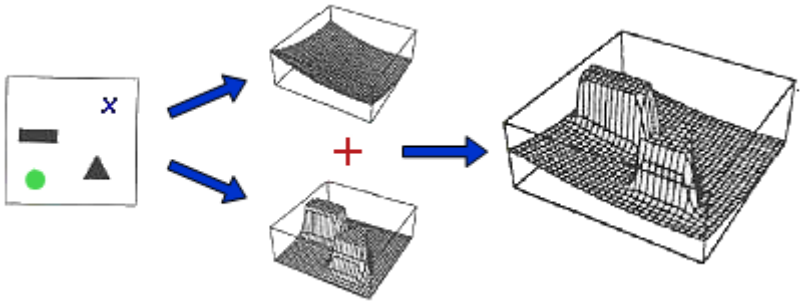


Figura 2.3: Abordagem usando Campos Potenciais (DAPPER et al., 2007)

Na figura 2.3, tem-se uma demonstração visual de como os campos potenciais funcionam. Em um primeiro momento, tem-se uma representação em duas dimensões do cenário. Na figura, a posição do agente é marcada com um círculo verde, o objetivo com

um “X”, e os obstáculos estão destacados com um retângulo e um triângulo. Cria-se então uma superfície cujo ponto mais baixo (dito de menor valor potencial) representa a posição do objetivo. Também cria-se uma segunda superfície, onde os pontos de alto potencial representam os obstáculos e os pontos de baixo valor potencial representam os espaços livres. Então, combinam-se as duas superfícies para obter o campo potencial final. Note que as duas superfícies possuem transições suaves entre os pontos de alto e baixo potencial. Ao final, para obter o caminho seguido pelo agente, coloca-se uma partícula no ponto que representa a posição do agente no cenário virtual, e anota-se o caminho que a partícula percorre até chegar ao vale da superfície.

Uma variação da técnica consiste em mapear uma pequena parte do cenário virtual no campo potencial, ao invés do cenário todo. Assim, é possível utilizar outra técnica para o planejamento de caminhos, utilizando os campos potenciais apenas para “corrigir” o caminho planejado pela outra técnica. Com isso, pode se evitar obstáculos dinâmicos que não foram previstos inicialmente (REIF; WANG, 1995; LAMIRAUX; BONNAFOUS; LEFEBVRE, 2004). Outra grande vantagem em se utilizar campos potenciais localmente é o ganho em desempenho e consumo de memória, sem afetar significativamente a qualidade dos resultados gerados pelos campos potenciais.

Mas o uso de campos potenciais locais pode levar a um problema, onde o agente não consegue decidir como dar o próximo passo. Esse problema ocorre quando o campo potencial possui mínimos locais. Quando o campo potencial possui um mínimo local, o agente pode ser atraído para ele, acreditando ser o melhor caminho para alcançar o objetivo. Mas quando o agente alcança o mínimo local, a força de atração do objetivo (que ainda não foi alcançado) é anulada pela força de repulsão dos obstáculos próximos, fazendo com que o agente pare de se mover.

Uma forma de se evitar os mínimos locais foi apresentada por Trevisan (TREVISAN et al., 2006), e constitui a base do trabalho de Dapper (DAPPER et al., 2007). Esses trabalhos serão apresentados e detalhados no próximo capítulo, e serviram de base para o trabalho apresentado nos capítulos seguintes.

3 USANDO FUNÇÕES HARMÔNICAS PARA PLANEJAR CAMINHOS

Como já foi citado, existem diversas técnicas para se planejar caminhos. Dentre elas, o uso de campos potenciais destaca-se pela qualidade e generalidade dos resultados gerados.

Mas apesar dessas características, o desempenho dessa técnica pode não ser suficiente para aplicações de tempo real com muitos personagens na tela. Assim, é natural que se tente aperfeiçoar a técnica dos campos potenciais para ter maior desempenho ao utilizá-la em uma aplicação de tempo real. Com esse intuito, Dapper utilizou os campos potenciais gerados a partir de funções harmônicas, até então uma abordagem utilizada na robótica, para a navegação de personagens virtuais. O objetivo deste trabalho é demonstrar que a técnica de campos potenciais, utilizada por Dapper, pode se beneficiar do uso das placas gráficas para melhorar o seu desempenho.

Mas para alcançar o objetivo proposto, antes é necessário expor a técnica de forma mais detalhada. Assim, o objetivo deste capítulo é a descrição do método proposto por Dapper (DAPPER et al., 2007), que serviu de base para este trabalho. Nas páginas seguintes, o trabalho citado será descrito em detalhes, incluindo a base matemática que o sustenta.

3.1 Considerações Iniciais

Antes de alguém iniciar uma viagem, é natural que essa pessoa planeje o caminho que fará. Ela irá escolher a rodovia por onde vai viajar, e escolhe as ruas que utilizará no trajeto até a rodovia e ao final dela. Também leva em consideração qual o caminho mais rápido ou mais seguro, de acordo com as suas preferências. Diz-se que esta é a fase de planejamento da viagem.

É na fase de execução da viagem que o caminho escolhido é percorrido. Mas durante a viagem, podem ocorrer imprevistos que não poderiam ser previstos na fase de planejamento. Um acidente pode bloquear um trecho da estrada e forçar um desvio até ultrapassar o ponto bloqueado. Esse tipo de ajuste só pode ser feito na fase de execução do caminho planejado, e deve afetar o caminho inicialmente planejado da menor forma possível.

O trabalho de Dapper (DAPPER et al., 2007), baseia-se na idéia de dividir o planejamento de caminho em duas fases, uma inicial e outra de ajustes. Para isso, Dapper usa campos potenciais nas duas fases. Na primeira, usa um campo potencial global para guiar o agente até o seu objetivo final. Na segunda fase, usa um campo potencial local, de menor tamanho em relação ao global, para realizar pequenos ajustes. Assim, só é necessário criar o campo potencial global uma vez para cada mudança de objetivo. En-

quanto isso, pequenos ajustes no caminho podem ser feitos com os campos potenciais locais, que podem ser obtidos com menor esforço computacional.

Um detalhe muito importante (citado anteriormente) é que o uso de campos potenciais para planejamento local pode levar o agente a mínimos locais. Para evitar este problema, Dapper utilizou um campo potencial gerado a partir da solução de funções harmônicas, ou seja, gerado pela solução de um Problema de Valor de Contorno (PVC).

3.2 Problemas de Valor de Contorno para Planejar Caminhos

A primeira proposta de um planejador de caminhos usando PVC foi feita por Connolly (CONNOLLY; BURNS; WEISS, 1990), com o uso de funções harmônicas. Por definição, uma função no domínio $\Omega \subset \Re^n$ é dita harmônica se ela satisfaz a equação de Laplace:

$$\nabla^2 p(r) = \sum_{i=1}^{n} \frac{\partial^2 p(r)}{\partial x_i^2} = 0 \qquad 3.1$$

A equação de Laplace é muito utilizada em diversas áreas da física por ser capaz de descrever campos de força, tais como eletromagnetismo, gravitação e dinâmica de fluídos. Ao aplicar a equação, obtêm-se linhas equipotenciais. Se em seguida forem calculadas as linhas de fluxo (normais às linhas equipotenciais), então é possível definir caminhos para que uma partícula possa se mover de um ponto de alto valor potencial para outro de baixo valor potencial.

Em 2006, Trevisan (TREVISAN et al., 2006) propôs uma extensão da equação de Laplace, onde suas soluções também não apresentam mínimos locais. Essas funções são obtidas através da solução de problemas de valor de contorno, usando as condições de contorno de Dirichlet. Essas funções satisfazem à seguinte equação

$$\nabla^2 p(r) + \varepsilon v . \nabla p(r) = 0 \qquad 3.2$$

onde ε é um valor escalar e v é um vetor unitário. Estes dois parâmetros servem para distorcer o campo potencial, onde ε é utilizado para indicar o grau da distorção, e v é utilizado para indicar a direção da distorção. Trevisan utilizou a equação 3.2 em um trabalho onde robôs exploram ambientes esparsos. Segundo Trevisan, a inclusão dos parâmetros ε e v à equação reduz o tempo de exploração do ambiente.

Em seu trabalho, Dapper (DAPPER et al., 2007) utilizou a equação 3.2, manipulando os parâmetros ε e v para obter diferentes comportamentos para os agentes autônomos. Ele sugere diversos valores e formas para variar esses valores durante o planejamento do caminho para obter tais comportamentos.

3.3 Solução Numérica de Problemas de Valor de Contorno

A estratégia utilizada por Dapper para resolver a equação 3.2 é discretizar o ambiente virtual em uma grade de células de mesmo tamanho cada. A cada célula (*i*,*j*), é associada uma área de dimensões proporcionais às dimensões da célula.

Também é associado um valor de potencial $p_{i,j}^{t}$ a cada uma das células no instante *t*. De acordo com as condições de contorno de Dirichlet, o valor do potencial de cada célula será alto caso ela contenha um obstáculo. De modo similar, uma célula terá um valor de potencial baixo se contiver um objetivo. Para as células livres, o potencial deverá ser calculado.

Para calcular o valor de potencial das células livres, a idéia é substituir o valor do potencial de cada célula por uma média ponderada dos valores de potencial das células vizinhas, durante um número *t* de iterações. Note que apenas as células livres terão o valor potencial substituído. As células com obstáculos ou objetivos permanecerão inalteradas. Após todas as iterações serem realizadas e não haver mudança significativa no valor das células, diz-se que o sistema convergiu.

O modo como as células são calculadas já foi estudado em outros trabalhos, e percebe-se que a escolha do método de substituição tem impacto direto no desempenho e na qualidade dos resultados obtidos. Dapper (DAPPER et al., 2007) optou por utilizar o método de Gauss-Seidel, baseado nos resultados obtidos por Prestes (PRESTES et al., 2002). Os autores deste trabalho perceberam que, para um número pequeno de iterações, o método de Gauss-Seidel gerava curvas mais suaves, permitindo que resultados parciais possam ser utilizados. A seguir, serão apresentados os métodos classicamente utilizados: Gauss-Seidel, Jacobi e SOR.

3.3.1 O Método de Gauss-Seidel

O método de Gauss-Seidel pode ser descrito da seguinte forma para o caso bidimensional. A cada iteração t, o valor do potencial *p* da célula (*i,j*) será atualizado de acordo com a seguinte equação:

$$p_{i,j}^{t} = \frac{1}{4}\left(p_{i+i,j}^{t-1} + p_{i-1,j}^{t} + p_{i,j+1}^{t-1} + p_{i,j-1}^{t}\right) \qquad 3.3$$

Estudando atentamente a equação 3.3, percebe-se que ela envolve valores da iteração atual (que não foram calculados ainda) e valores da iteração anterior (que já foram calculados). Esse detalhe é muito importante, já que essa fórmula tem uma codificação bastante simples em máquinas mono-processadas. Assim, a equação consiste basicamente em substituir o valor atual pela média dos vizinhos de uma célula, durante um número *n* de iterações. Como já foi dito, apenas as células livres terão o seu valor de potencial atualizado durante esse processo, chamado de relaxamento.

3.3.2 O Método de Jacobi

Este método é bastante semelhante ao método de Gauss-Seidel, com a diferença de que todos os valores utilizados na fórmula são obtidos na iteração anterior. Assim, o método de Jacobi será definido da seguinte forma para o caso bidimensional. A cada iteração *t*, o valor do potencial *p* da célula (*i,j*) será atualizado de acordo com a seguinte equação:

$$p_{i,j}^{t} = \frac{1}{4}\left(p_{i+i,j}^{t-1} + p_{i-1,j}^{t-1} + p_{i,j+1}^{t-1} + p_{i,j-1}^{t-1}\right) \qquad 3.4$$

Por utilizar todos os valores da iteração anterior, o método de Jacobi geralmente precisa de um número maior de iterações para convergir em relação a Gauss-Seidel. Porém, ele é mais adaptável à codificação em arquiteturas multi-processadas, permitindo que uma iteração completa possa ser executada em paralelo, para todas as células. Assim, uma arquitetura altamente paralela pode ser capaz de executar o método de Jacobi mais rapidamente do que se executasse o método de Gauss-Seidel, mesmo necessitando de mais iterações. As placas gráficas são o tipo mais popular desse tipo de arquitetura.

3.3.3 O Método SOR

O método Successive Over-Relaxation (SOR) envolve células da iteração atual e da iteração anterior, como o modo Gauss-Seidel. Mas ele inclui também alguns outros termos que, segundo Fortuna (FORTUNA, 2000), faz com que ele venha a convergir mais rapidamente que Gauss-Seidel ou Jacobi. Assim a atualização do valor de potencial *p* de uma célula (*i,j*) na iteração *t* é dada pela seguinte equação:

$$p_{i,j}^{t} = p_{i,j}^{t-1} + \frac{w}{4}\left(p_{i+i,j}^{t} + p_{i-1,j}^{t} + p_{i,j+1}^{t-1} + p_{i,j-1}^{t-1} - 4p_{i,j}^{t-1}\right) \qquad 3.5$$

onde *w* é uma constante de aceleração.

Apesar da vantagem de convergir mais rapidamente, o método tem o inconveniente de que os resultados parciais sofrem muito com oscilações. Como Dapper previu a possibilidade de utilizar resultados parciais, este método foi descartado por poder gerar resultados pouco convincentes na trajetória final. Além disso, ele não é tão adaptável às arquiteturas paralelas (de forma semelhante a Gauss-Seidel), como é o método de Jacobi.

3.4 O Cálculo do Campo Potencial Utilizado

Conforme já foi citado, foi utilizado o método de Gauss-Seidel para o cálculo do campo potencial. Assim, é possível obter à seguinte equação para atualização do campo potencial:

$$p_{i,j}^{t} = \frac{p_l + p_r + p_t + p_d}{4} + \varepsilon\frac{(p_l + p_r)v_x - (p_t + p_d)v_y}{8} \qquad 3.6$$

onde $p_l = p_{i-1,j}^{t}$, $p_r = p_{i+i,j}^{t-1}$, $p_t = p_{i,j-1}^{t}$, $p_d = p_{i,j+1}^{t-1}$ e $v = (v_x, v_y)$. Além disso, o parâmetro ε deve estar no intervalo $(-2,2)$, já que valores fora deste intervalo podem fazer com que o agente não alcance o objetivo. Para uma descrição detalhada da limitação ao intervalo $(-2,2)$, o leitor poderá consultar Dapper (DAPPER et al, 2007) e Silveira (SILVEIRA, 2008). Na figura 3.1, é possível se visualizar as células envolvidas na equação 3.6.

Após o campo potencial ter sido calculado, o gradiente descendente em cada uma das células pode ser obtido. O gradiente descendente será utilizado para orientar o agente virtual no seu movimento. Assim sendo, para uma célula (*i,j*), o seu gradiente descendente, no caso bidimensional, é definido pelo seguinte vetor:

$$(\nabla p)_{i,j} = \left(\frac{p_{i+1,j} - p_{i-1,j}}{2}, \frac{p_{i,j+1} - p_{i,j-1}}{2}\right) \quad 3.7$$

Com o resultado da equação 3.7, é definido a direção que o agente deverá seguir para alcançar o objetivo, a partir da célula (*i,j*).

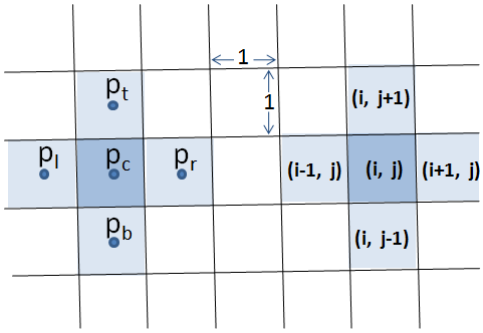


Figura 3.1: Localização das células envolvidas nos cálculos

3.5 Melhorando a Qualidade do Movimento do Agente

Já é possível definir uma trajetória utilizando os resultados obtidos com a equação 3.7. Porém, os resultados obtidos dessa forma demonstraram ser pouco naturais em algumas situações. Uma delas é devido ao fato de que o agente deverá tomar um desvio se houver um obstáculo à sua frente. Se a região coberta por todas as células do mapa local for muito pequena, o obstáculo somente se tornará visível quando estiver muito próximo do agente. Isso irá provocar uma forte força de repulsão, fazendo com que o agente se vire bruscamente. Para contornar este problema, Dapper (DAPPER; PRESTES; NEDEL, 2007) sugeriram a seguinte equação para o ajuste da nova posição do agente:

$$\Delta d = v(cos(\varphi^t), sen(\varphi^t)) \quad 3.8$$

onde v representa a velocidade máxima do agente e φ^t é definido por:

$$\varphi^t = \eta\varphi^{t-1} + (1-\eta)\zeta^t \quad 3.9$$

com $\eta \in [0,1)$ e ζ^t representando a orientação do gradiente descendente no momento *t*.

Utilizando a equação 3.8, é possível controlar a reação do agente, fazendo com que ele não reaja de forma automática quando surge um novo obstáculo no ambiente. Quando $\eta \longrightarrow 1$, o agente tende a reagir mais lentamente, podendo até mesmo colidir com o obstáculo. Observou-se que isso dá mais realismo à cena, apesar da possibilidade de colisão.

Outro caso onde os movimentos gerados parecem pouco naturais é quando existe uma grande quantidade de obstáculos no ambiente. Nesse caso, a ocorrência de colisões é muito grande, devido à velocidade do movimento do agente. Para resolver este problema, outro fator foi incluído na equação 3.9

$$\varphi^t = \eta\varphi^{t-1} + (1-\eta)\zeta^t\,\psi(|\varphi^{t-1} - \zeta^t|) \qquad 3.10$$

onde a função $\psi : \Re \longrightarrow \Re$ é definida por:

$$\psi(\mathrm{x}) = f(x) = \begin{cases} 0 & se\ x > \pi/2 \\ \cos(x) & caso\ contrário \end{cases}.$$

Se a diferença entre a orientação do agente no momento $t-1$ e a orientação do gradiente descendente no momento t (o que é representado por $|\varphi^{t-1} - \zeta^t|$) for muito grande (no caso, maior que $\pi/2$), então a função retorna 0, o que faz com que o agente pare de se mover. Caso contrário, a velocidade do movimento do agente será proporcional à diferença entre as duas orientações citadas, sendo maior quanto menor a probabilidade de uma colisão ocorrer. Isso acaba se mostrando correto, visto que uma pessoa tende a andar muito devagar em ambientes muito ocupados.

3.6 Descrição da Arquitetura do Sistema

O trabalho de Dapper (DAPPER et al., 2007) teve como resultado prático um sistema em C++. Este sistema foi posteriormente estendido por Silveira (SILVEIRA, 2008). A seguir, serão descritos os componentes básicos do sistema já implementado.

3.6.1 O Agente Virtual

O ambiente possui um conjunto de agentes $\{a_k\}$. A cada agente a_k é associado um vetor unitário v_k e um valor escalar ε_k no intervalo $(-2,2)$. Esses parâmetros permitem que cada agente possa ter um comportamento particular ao planejar o caminho a ser seguido.

Do ambiente virtual, o agente conhece a sua posição atual e a posição do seu objetivo O_k. Ele também é capaz de perceber os obstáculos e os outros agentes que estejam em um cone de visão de ângulo α_k. Esse cone de visão é limitado pela região definida por um mapa local, a ser descrito adiante.

Existe também a possibilidade de se manter uma lista de objetivos para cada agente virtual, ao invés de um único objetivo. Nesse caso, o agente só pode perseguir um objetivo por vez, e o objetivo O_k refere-se ao objetivo que o agente está perseguindo no momento considerado.

3.6.2 O Mapa Global do Ambiente Virtual

A representação do ambiente é feita por um conjunto de matrizes $\{M_k\}$, onde cada matriz M_k é associada a um objetivo O_k. Desta forma, todos os agentes virtuais que desejam alcançar o objetivo O_k podem compartilhar a mesma matriz M_k.

Cada matriz M_k é composta por $L_x * L_y$ células, onde cada célula $C_{i,j}^k$ possui um valor potencial associado $P_{i,j}^k$, e é associada a uma região $R_{i,j}$ do ambiente, quadrada e centrada no ponto (R_i, R_j), também do ambiente.

O valor do potencial de cada uma das células é obtido da seguinte forma. Para as células $C_{i,j}^k$ onde o objetivo O_k encontra-se sobre a região associada $R_{i,j}$, o valor do potencial associado $P_{i,j}^k$ é 0. Para as células $C_{i,j}^k$ onde existe um obstáculo sobre a região associada $R_{i,j}$, o valor do potencial associado $P_{i,j}^k$ é 1. Para as demais células, o valor do potencial associado é calculado através das funções harmônicas, conforme citado anteriormente.

3.6.3 O Mapa Local dos Agentes Virtuais

A cada agente virtual a_k é associado um mapa local, composto por uma matriz m_k. Cada matriz m_k é composta por $l_x^k * l_y^k$ células, com cada célula $c_{i,j}^k$ sendo associada a uma região $r_{i,j}$ centrada no ponto (r_i, r_j) do ambiente. Cada célula também possui um valor potencial associado $p_{i,j}^k$. Cada mapa local é centrado na posição do agente associado.

Para definir o potencial associado a cada célula $c_{i,j}^k$, as células são divididas em três grupos, da seguinte forma. As células pertencentes à fronteira do mapa local pertencem ao conjunto *b-zone* (*boundary zone*). As células próximas à fronteira não pertencentes ao conjunto *b-zone* pertencem ao conjunto *f-zone* (*free zone*). As células restantes (mais próximas ao centro do mapa) pertencem ao conjunto *u-zone (updateable zone)*. Na figura 3.2, as células de um campo potencial estão apresentadas com cores distintas, permitindo a visualização desses grupos de células.

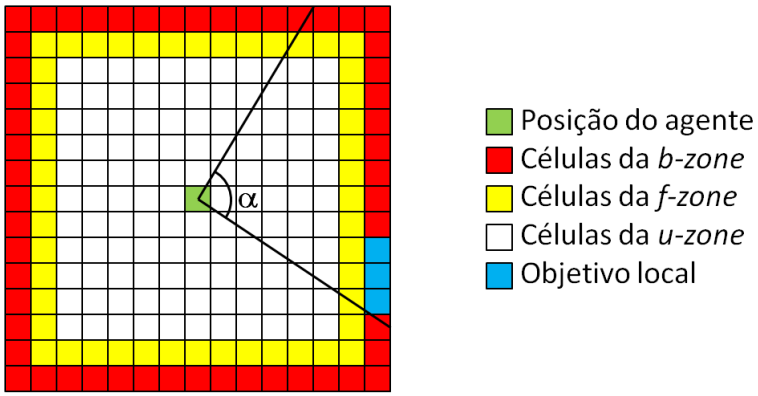


Figura 3.2: O mapa local de um agente

3.6.4 Atualizando o Mapa Local

A atualização do mapa local de um agente a_k inicia definindo o valor do potencial das células na *b-zone*. A essas células é atribuído o valor potencial 1, exceto por algumas células onde o valor do potencial atribuído é 0. Essas células onde o valor do potencial atribuído é 0 são denominadas como o "objetivo local" do mapa. As células para compor o objetivo local são definidas pela direção entre a posição do centro do mapa local e do objetivo atual do agente associado.

Os obstáculos e outros agentes percebidos pelo agente a_k também são mapeados no seu mapa local. Esse mapeamento é feito da seguinte forma. Para cada célula $c_{i,j}^k$ na *u-zone* que cobre uma região onde há um obstáculo ou um outro agente, é atribuído o valor potencial 1. Note que as células da *f-zone* não são modificadas. Isso é necessário para que não haja uma quebra da conexão entre as células livres e as células do objetivo local, caso algum obstáculo seja mapeado próximo a elas. Se ocorresse tal quebra de conexão, então não seria possível encontrar um caminho entre a posição do agente e o objetivo local, impossibilitando o agente de prosseguir sua navegação.

Em seguida, o potencial das células da *u-zone* e da *f-zone* que não foram modificadas nos passos anteriores é calculado, com o uso das funções harmônicas já estudadas, incluindo os atributos v_k e ε_k do agente em questão. Este é o processo de convergência, também conhecido como relaxamento, e é o que mais impacta no seu desempenho.

3.6.5 Movimento do Agente

Após o relaxamento do mapa local, é calculada a orientação do próximo passo a ser seguido pelo agente. Essa orientação é dada pela orientação do gradiente descendente na célula central do mapa local. Para obter essa orientação, devemos resolver a equação 3.7 utilizando $i = \lceil l_x^k/2 \rceil$ e $j = \left\lceil l_y^k/2 \right\rceil$.

O vetor obtido é utilizado para calcular a nova posição agente no ambiente virtual. É neste momento que as equações 3.8 ou 3.10 podem ser utilizadas para melhorar a qualidade do movimento realizado.

3.6.6 Algoritmo

A arquitetura do sistema é agora apresentada na forma de um algoritmo iterativo. Repare que os passos 4 a 6 podem ser executados em uma fase de pré-processamento.

Algoritmo Planejador de Caminhos BVP

1: **Para cada** agente a_k, **faça**
2: Definir atributos comportamentais v_k e ε_k
3: **Fim Para cada**
4: **Para cada** objetivo o_k, **faça**
5: Calcular o mapa global M_k
6: **Fim Para cada**
7: **Enquanto** existe agente a_k que não alcançou o objetivo o_k, **faça**
8: **Para cada** agente a_k que não alcançou o objetivo o_k, **faça**
9: Detectar outros agentes e obstáculos (estáticos e dinâmicos) que estejam no campo de visão
10: Atualizar o mapa local, utilizando os dados obtidos no passo anterior
11: Gerar o objetivo intermediário no mapa local
12: Relaxar o campo potencial do mapa local
13: Calcular o gradiente descendente na célula central do mapa local
14: Calcular a nova posição do agente e movê-lo
15: **Se** o agente a_k alcançou o objetivo o_k, **faça**
16: Parar de mover o agente a_k
17: **Fim Se**
18: **Fim Para cada**
19: **Fim Enquanto**

4 IMPLEMENTAÇÃO PARALELA DO ALGORITMO

Alguns algoritmos são muito propensos à execução paralela dos seus passos, podendo ter o seu tempo total de execução reduzido em várias vezes quando implementados dessa forma. Quando tais algoritmos são executados em máquinas mono-processadas, seu desempenho é muito limitado, já que o processador é capaz de realizar apenas uma tarefa por vez. Em uma máquina multi-processada, *n* passos podem ser executados ao mesmo tempo, podendo fazer com que o tempo total de execução destes *n* passos seja proporcional a 1 passo, no caso ótimo.

Em um ambiente com muitas pessoas em movimento, nenhuma delas conversa com as outras para decidir como dar o próximo passo. Para que elas decidam como será seu próximo passo, basta que cada indivíduo saiba a posição atual dos obstáculos no ambiente[1]. Ou seja, as pessoas tomam a decisão de como será o próximo passo em paralelo, utilizando apenas informações provenientes dos momentos anteriores.

Pensando dessa forma, parece natural que um algoritmo de planejamento de caminhos seja paralelizável. E conforme é possível perceber na técnica proposta por Dapper (DAPPER et al., 2007), diversos cálculos podem ser realizados em paralelo, sendo o relaxamento aquele que mais aparenta se beneficiar dessa característica.

Neste capítulo, será apresentada de forma mais clara as características paralelas do algoritmo. Também será apresentada a implementação da técnica, visando aproveitar esse paralelismo. Esta proposta foi implementada utilizando as linguagens de programação C++ e CUDA, esta última disponibilizada pela nVidia para uso em suas placas gráficas para programação massivamente paralela.

4.1 Paralelismo na Técnica Descrita

De maneira geral, o algoritmo descrito é bastante paralelizável. A descrição da técnica na forma de um algoritmo de passos simples, feita no item 3.6.6 deste trabalho, nos ajudará a perceber como isso pode ser feito.

4.1.1 Inicialização

Os passos indicados nas linhas 1 a 3 do algoritmo referem-se à definição de atributos dos agentes virtuais, e são bastante simples de serem executados. A definição desses atributos deve levar em consideração apenas o tipo de comportamento a ser simulado. Dessa forma, foi considerado que tais atributos são definidos manualmente, baseados nas diversas considerações feitas por Dapper (DAPPER et al., 2007), e que foram sinte-

[1] Seres humanos reais também podem utilizar outras informações, como a direção, velocidade e probabilidade das outras pessoas mudarem de direção e velocidade. Como o algoritmo descrito não utiliza esse tipo de informação, elas serão desconsideradas.

tizadas no capítulo 3. Assim, esses atributos servem de entrada para o código executável, e por isso não há codificação paralela a ser feita.

4.1.2 Cálculo dos Mapas Globais

Assim como no item 4.1.1, considera-se que os objetivos são valores de entrada para o algoritmo, já que o trabalho descrito não trata da sua obtenção. Porém, o cálculo dos mapas globais (linhas 4 a 6 do algoritmo) pode ser paralelizável em dois níveis.

Em um nível mais alto, a obtenção dos diversos mapas globais é independente, pois o caminho para se alcançar um objetivo é independente do caminho para se alcançar outro. Esse cálculo depende apenas dos obstáculos estáticos, o que se considera estar definidos já no início da execução.

Já em um nível mais baixo, o cálculo de um único mapa global também pode ser paralelizado. Basicamente, a idéia consiste em paralelizar o cálculo dos valores de potencial de cada célula. A forma como isto pode ser feito é a mesma que pode ser feita para os mapas locais, e será descrita adiante.

4.1.3 Detecção dos Obstáculos e Atualização do Mapa Local

Intuitivamente, detectar obstáculos é uma tarefa bastante paralelizável, pois consiste basicamente em selecionar os objetos do cenário virtual que estão no campo de visão de um agente em um determinado momento. Para o algoritmo, inclusive a ordem em que os obstáculos são selecionados é irrelevante, pois o algoritmo não trata de forma diferente os objetos que estão mais próximos ou mais distantes do agente virtual.

Na linha 9 do algoritmo, todos os obstáculos que satisfazem simultaneamente às seguintes condições são selecionados:

- Estar em uma região coberta por alguma célula do mapa local;
- Estar dentro do cone de visão do agente.

Durante esses passos, o conceito de obstáculo é generalizado para qualquer elemento da cena que deva interferir no planejamento do caminho de um determinado agente virtual, ou seja, obstáculos estáticos, dinâmicos e outros agentes virtuais. Após a descoberta desses obstáculos, as células que cobrem a região ocupada por eles recebem o valor de potencial 1. A ordem em que cada obstáculo é descoberto e a respectiva célula do campo potencial é atualizada não modifica os resultados gerados, desde que todas as células que cobrem um obstáculo estejam atualizadas antes de ocorrer o relaxamento.

Assim, essa tarefa pode ser realizada em paralelo da seguinte forma. Cria-se uma linha de execução para cada obstáculo, para cada um dos agentes na cena. Essa linha de execução irá verificar se um único obstáculo satisfaz às condições acima e, se pertencer, atualizará os valores das células que o cobrem.

Considerando que existem k agentes e s obstáculos (dinâmicos e estáticos) na cena, o número de linhas de execução é dado por:

$$l = k * [(k - 1) + s] \tag{4.1}$$

pois cada agente terá que verificar se $k - 1$ agentes e s obstáculos estão presentes no seu campo de visão.

4.1.4 Definição dos Objetivos Intermediários

Gerar um objetivo intermediário e mapeá-lo no mapa local (linha 11 do algoritmo) é uma atividade bastante simples, dependente apenas do número de agentes na cena. Para isso, basta que cada agente tome a direção em que se encontra o seu objetivo em relação à sua própria posição, e utilize-a para definir quais são as células do mapa local a serem marcadas com o valor de potencial 0. Como existe apenas um mapa local por agente, e cada agente possui um único objetivo a cada momento, é necessária uma linha de execução por agente na cena.

Deve ser observada a dependência de dados entre o passo anterior e este passo. A definição dos objetivos intermediários não pode ser executada em paralelo com a etapa de detecção dos obstáculos e mapeamento no mapa local. Se isso ocorresse, o objetivo intermediário poderia ser calculado antes que um obstáculo que ocupa a mesma célula que cobre o objetivo fosse detectado[2]. Quando o obstáculo for detectado, a célula que já foi atualizada com o valor de potencial 0 teria o seu potencial modificado para 1. Nesse caso, a informação do objetivo intermediário se perde, e o agente não terá condições de definir o seu próximo passo.

4.1.5 Relaxamento do Mapa Local

Esta etapa (linha 12 do algoritmo) é a mais custosa da técnica. Dado que existe um número k de agentes na cena, que o mapa local do agente k deve ser relaxado em t_k iterações, e que a cada iteração devem ser atualizadas $l_x^k * l_y^k$ células, o número de operações que serão executadas é dado por:

$$c = \sum_{i=1}^{k} t_k * l_x^k * l_y^k$$

Tal número de operações é o principal responsável pelo desempenho do algoritmo, visto que será executado a cada passo a ser dado pelos agentes virtuais.

Porém, esse processamento pode ser paralelizado. O cálculo do campo potencial de cada agente é independente, permitindo que todos possam ser executados em paralelo. Ou seja, é possível dividir esse processamento em k linhas de execução, uma para cada campo potencial.

Além disso, durante o relaxamento do campo potencial, a cada iteração o valor de potencial de cada célula pode ser calculado em paralelo, desde que seja utilizado o método de Jacobi ao invés do método Gauss-Seidel. Note que diferentes iterações de um mesmo campo potencial não podem ser executadas em paralelo, pois o resultado do valor de potencial em uma iteração t é dependente do resultado do valor de potencial de outras células obtidas na iteração $t - 1$.

Sendo assim, a execução em paralelo da etapa de relaxamento necessita de:

---

[2] Aqui, refere-se ao caso onde um objetivo e um obstáculo estão tão próximos que, durante a discretização do ambiente, eles são mapeados para a mesma célula do mapa local. O caso onde um objetivo está dentro da região coberta por um obstáculo é considerado um erro de entrada de dados e não é tratado pelo algoritmo.

$$e = \sum_{i=1}^{k} l_x^k * l_y^k$$

linhas de execução, uma para cada célula de cada mapa local dos agentes. O tempo total de execução de toda a etapa de relaxamento é proporcional a $\max(t_k)$.

4.1.6 Últimas Etapas do Algoritmo

Os passos 13 a 17 do algoritmo são pouco custosos em sua execução. Resumidamente, as três execuções precisam de um tempo proporcional a k, onde k é o número de agentes.

O passo 13 é constituído apenas da solução da equação 3.7, que por sua vez envolve o valor de potencial de quatro células em um cálculo simples. O passo 14 envolve o cálculo de uma nova posição, utilizando as equações 3.8 ou 3.10. Por fim, as linhas 15 a 17 podem ser resumidas a um teste e uma marcação no agente virtual, que será utilizada nas linhas 7 e 8, no passo seguinte.

Dessa forma, tais passos do algoritmo podem ser executados em paralelo, utilizando uma linha de execução por agente virtual. Esse custo por agente virtual é independente do tamanho do campo potencial ou número de iterações necessárias para relaxá-lo.

4.2 Modelo de Programação do CUDA

Conforme já citado, foi feita a opção pelo uso da linguagem de programação CUDA (Compute Unified Device Architecture) para a implementação do algoritmo em placas gráficas. Essa decisão foi tomada por dois motivos. O primeiro refere-se ao paralelismo presente nas placas gráficas atuais, que possuem dezenas (ou até centenas) de processadores trabalhando em paralelo, por um custo bastante acessível. O segundo motivo é que a linguagem de programação CUDA é uma extensão da linguagem C que aproveita as capacidades das placas gráficas da marca nVidia. Com ela, é possível ter um controle bastante completo da execução do código em uma placa gráfica, de forma bastante simples e eficiente, com instruções de iteração e sincronização.

Mas como qualquer linguagem de programação, o CUDA tem certas características que, se forem ignoradas, impactam negativamente no desempenho da aplicação. Essas características guiaram o desenvolvimento do algoritmo de planejamento de caminhos, de forma que o resultado final pudesse beneficiar-se delas.

Assim, para haver uma melhor compreensão das decisões tomadas, serão apresentadas algumas das características estudadas do CUDA. Essas características influenciaram diretamente as decisões de organização da estrutura de dados e da implementação paralela.

4.2.1 Introdução ao CUDA

Basicamente, o CUDA é um conjunto de extensões à linguagem C. Essas extensões são baseadas em algumas palavras-chave, operadores novos, e variáveis globais declaradas em tempo de compilação. Assim, apesar de ser sintaticamente bastante parecido com C, CUDA pode ser considerada uma nova linguagem por não ser reconhecido por

nenhum compilador de C padrão. Ainda assim, é possível integrar código compilado com CUDA em código C ou C++, através do uso de bibliotecas pré-compiladas.

Mas apesar da semelhança com uma linguagem já bastante difundida, a forma com que o código deve ser escrito é bastante diferente do usual. Isso ocorre porque os algoritmos escritos em CUDA devem ter o seu paralelismo muito bem exposto pelo programador. Além disso, o código deve levar em conta também diversos aspectos da arquitetura das placas gráficas nVidia. Por isso, compreender a arquitetura dessas placas gráficas é importante ao desenvolver soluções para elas.

4.2.2 Arquitetura Básica do CUDA e das Placas Gráficas nVidia

O desenvolvimento de um algoritmo paralelo usando CUDA deve levar em consideração que o código poderá rodar na CPU (chamado de *Host* pela sua documentação) ou nas placas gráficas instaladas na máquina (chamadas de *Device* pela documentação).

A divisão entre *Host* e *Device* define dois ambientes de execução, cada um com um espaço de memória separado do outro. A comunicação entre os dois ambientes é feita através de cópia de memória e chamada de funções da API do CUDA. Essa comunicação é sempre realizada com a iniciativa do *Host*.

O código executado no *Host* é geralmente mono-processado. O CUDA não oferece suporte à criação de threads nesse ambiente de execução, mas é possível utilizar a funcionalidade oferecida por outras API de programação multi-thread. A criação de uma nova thread e a troca de contexto entre duas já criadas nesse ambiente é realizada pelo sistema operacional, sendo bastante custosa. Além disso, threads diferentes podem executar funções diferentes.

Já o código executado no *Device* é totalmente multi-processado. Quando um tipo de função especial (chamada de *Kernel*) é chamada a partir do *Host*, diversas threads são criadas no *Device*, e todas elas irão executar o código da função chamada. O número de threads é definido no momento da chamada do Kernel, e cada uma delas recebe um identificador único. Esse identificador é utilizado pelo código para tomar decisões e fazer cálculos específicos por thread. Além disso, o custo de criação e troca de contexto entre threads é muito baixo, pois é realizado por um escalonador em hardware na GPU, sem interferência do sistema operacional. A documentação do CUDA inclusive recomenda um grande número de threads para se obter um bom aproveitamento do paralelismo da placa gráfica.

4.2.3 Mapeamento de Threads na GPU

Apesar de ser possível criar centenas de threads de uma única vez, todas elas não são executadas ao mesmo tempo. E, para haver uma otimização do uso da GPU, as threads são agrupadas de um modo bastante peculiar.

Basicamente, quando um Kernel é executado, é necessário definir dois parâmetros de execução: o número de blocos e o número de threads por bloco. O número total de threads executadas é dado pelo número de blocos vezes o número de threads por bloco. Esses números também podem ser definidos em vetores de duas ou três dimensões, criando assim identificadores de duas ou três dimensões, respectivamente.

A figura 4.1 exemplifica a divisão das threads que executam um determinado Kernel. No exemplo, o número de blocos é dado pelo vetor (3,2) e o número de threads por bloco é dado pelo vetor (4,3). Assim, o número total de threads que executa esse kernel é dado por:

$$\underbrace{3 * 2}_{\substack{número \\ de\ blocos}} * \underbrace{4 * 3}_{\substack{número\ de \\ threads \\ por\ bloco}} = \underbrace{72}_{\substack{número \\ total\ de \\ threads}}$$

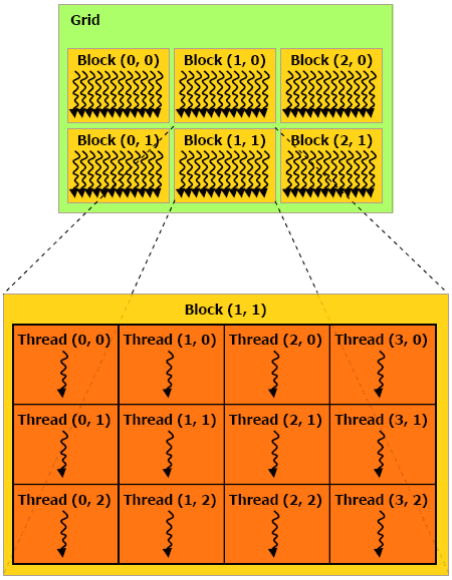


Figura 4.1: Divisão das threads em blocos de execução (CUDA, 2008)

O motivo de separar as threads desta forma é simplificar o trabalho do escalonador da GPU, que irá mapear cada bloco de threads em um bloco de processadores da GPU (chamado de multiprocessador)[3]. Além disso, essa organização permite que o hardware da placa gráfica possa obter um melhor desempenho.

Cada multiprocessador deve ser capaz de executar um bloco de threads encaminhado para ele. Por isso, o CUDA restringe em 512 o número máximo de threads por bloco. Quando um número maior que este é requerido, o Kernel não é executado e um erro é gerado. Quanto ao número de blocos, não há limite. Mas quando a placa gráfica não é capaz de processar todos os blocos em paralelo, eles são serializados, e o processamento dos blocos excedentes só inicia quando os primeiros blocos terminam a sua execução.

4.2.4 Arquitetura de Memória

Um multiprocessador é um agrupamento de processadores da GPU, que compartilham registradores e uma área de memória compartilhada (*Shared Memory*). Essa área de memória é tão rápida quanto os registradores, e é através dela que as threads podem trocar informações eficientemente. O tamanho da memória compartilhada é de 16KB por multiprocessador, em todas as placas gráficas capazes de executar código CUDA, até a escrita deste trabalho.

É necessário ressaltar que as threads que executam em um multiprocessador possuem acesso à memória compartilhada daquele multiprocessador somente. Threads execu-

[3] Na verdade, o escalonador da GPU pode decidir mapear mais de um bloco por multiprocessador. Mas as restrições de comunicação entre dois multiprocessadores são estendidas aos blocos de threads, e por isso é conveniente considerar que blocos distintos executam em multiprocessadores distintos.

tando em multiprocessadores diferentes podem trocar informações apenas através da memória global (*Device Memory*).

A memória global é a área de memória da placa gráfica para uso geral, e possui tempo de acesso muito maior que o da memória compartilhada (na ordem de centenas de vezes). Dessa forma, a troca de informações entre threads de multiprocessadores diferentes é bastante ineficiente, não sendo recomendado pela documentação. O seu tamanho depende do modelo da placa gráfica, mas os tamanhos típicos nos modelos atuais variam entre 256MB e 2GB. Por fim, a comunicação entre o *Host* e o *Device* é feita através da memória global, e não pode ser feita através da memória compartilhada.

Na memória global, existem duas regiões especiais. Uma é a área de memória constante, que possui acesso através de cache (o que melhora o seu desempenho). Mas ainda assim, o tempo de acesso a ela é maior que o da memória compartilhada. O seu tamanho é de 64KB, compartilhado entre todos os multiprocessadores. A outra área é a memória de texturas. Ela possui um cache menor (entre 6 e 8KB por multiprocessador) que é otimizado para acesso localizado em 2D. Essas duas áreas de memória somente podem ser lidas por um Kernel, e a escrita de dados é feita pelo *Host*.

O esquema exibido na figura 4.2 demonstra o relacionamento entre os processadores de um multiprocessador e os diversos tipos de memória da placa gráfica. Nele, as setas indicam as direções que os dados podem trafegar, permitindo a leitura e escrita nessas áreas.

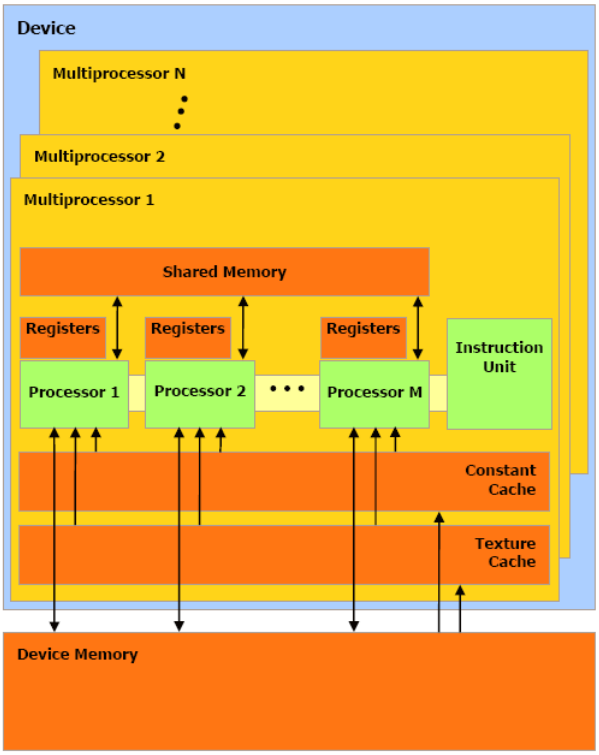


Figura 4.2: Arquitetura de memória de um multiprocessador (CUDA, 2008)

4.2.5 Execução do Código

Um algoritmo implementado com CUDA sempre inicia executando no *Host*, como se fosse um programa comum. O *Host* é o responsável por controlar o código que executa no *Device* (e nunca o contrário). Inicialmente, o *Host* utiliza a API do CUDA para alocar e trocar informações com a memória do *Device*. Em seguida, baseado na quantidade de informações a serem processadas e na forma com que o algoritmo foi implementado, o *Host* define o número de blocos e threads por bloco. Em seguida, esses parâmetros são passados para o Kernel, que então executa o código em paralelo, na GPU. Após todas as threads terminarem sua execução, o *Host* copia para a sua memória os dados processados no *Device*.

A troca de dados entre o *Host* e o *Device* deve ser minimizada, evitando muitas chamadas para a cópia de informações. Segundo a documentação do CUDA, a cópia de um grande bloco de memória é muito mais eficiente do que a cópia de vários pequenos blocos.

Durante a execução de um Kernel, uma técnica bastante recomendada pela documentação é que cada thread copie parte dos dados da memória global para a memória compartilhada, e execute seus cálculos nesta área de memória. Após a conclusão dos cálculos, a thread é responsável por escrever de volta o seu resultado na memória global, permitindo que o *Host* possa buscá-los. Dessa forma, é possível evitar muitos acessos desnecessários à memória global.

A sincronização entre threads é feita de forma a minimizar o número de threads ociosas na GPU. Por isso, a sincronização é feita apenas entre as threads de um bloco, ignorando as threads que executam em outros blocos. Visto de outra forma, a sincronização entre threads de blocos diferentes não é possível.

O uso de desvios condicionais (instruções *if, switch, for, do, while*) deve ser utilizado com bastante cuidado. Quando uma thread executa uma instrução desse tipo, todas as threads daquele bloco precisam ser serializadas. Com isso, as threads que não irão executar a instrução condicional ficarão ociosas, até que as outras threads retornem ao fluxo de execução principal, seguido por todas as threads.

Além destes, existem diversos outros cuidados a serem tomados durante a programação de um algoritmo utilizando CUDA que, se não forem tomados, terão impacto direto no desempenho. Porém, eles não serão apresentados por não terem influenciado diretamente a forma com que este trabalho foi desenvolvido.

4.3 Implementação do Planejador de Caminhos com o CUDA

Após entender os pontos onde o algoritmo proposto por Dapper (DAPPER et al., 2007) pode ser paralelizado e quais são as principais características e restrições do CUDA (CUDA, 2008), é possível estudar uma maneira de implementá-lo eficientemente com esta linguagem de programação.

Nos itens a seguir, será visto a estrutura de dados utilizada, e como os campos potenciais são resolvidos.

4.3.1 Estrutura de Dados

O mapa local de um agente k é composto de diversos atributos: uma matriz de células (m_k), a dimensão dessa matriz (l_x^k e l_y^k), o número de iterações para relaxar o campo potencial (t_k), além de outros relativos à área coberta pelo campo potencial. Para os

agentes, existe o objetivo atual (O_k), os atributos v_k e ε_k, além da orientação e cone de visão.

Para evitar que o número de cópias entre o *Host* e o *Device* seja grande, cada um desses atributos é armazenado em um vetor, onde a posição k do vetor contém o atributo do agente k. Assim, o número de cópias de dados entre o *Host* e o *Device* é igual ao número de atributos de um agente mais o número de atributos de um campo local, sendo independente do número de agentes na cena.

Em particular, todas as matrizes de células são armazenadas em um mesmo vetor de células. Como não existe nenhuma garantia de que todos os mapas locais terão o mesmo número de células, cada mapa local k possui um atributo s_k, que indica a posição no vetor de células em que inicia o campo potencial.

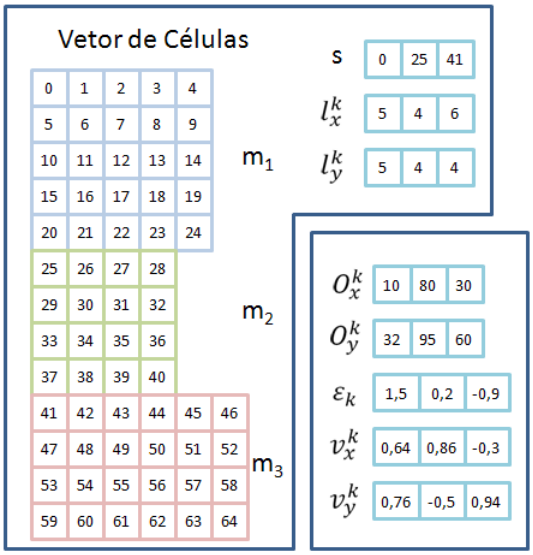


Figura 4.3: Estrutura de dados utilizada

Na figura 4.3, está demonstrado um exemplo de como a estrutura de dados é organizada para 3 agentes na cena. Na região demarcada mais à esquerda, são apresentados os dados dos campos potenciais. Na região mais à direita, são apresentados os atributos dos agentes. Os atributos referentes ao objetivo atual O e ao parâmetro v foram decompostos em dois vetores, pois esses atributos são definidos com coordenadas no plano XY. Para simplificar a figura, outros atributos (como o número de iterações ou a região ocupada pelas células) não estão exibidos. Mas como eles possuem a mesma estrutura lógica dos atributos apresentados, eles podem ser facilmente deduzidos pelo leitor.

Definido a organização desses dados, eles precisam transitar entre o *Host* e o *Device*. A idéia então é manter uma cópia dessa estrutura nos dois ambientes de execução, fazendo cópias entre os dois apenas quando necessário. Na prática, a maior parte das informações permanece constante durante a execução do algoritmo. Dessa forma, a cada passo do algoritmo é necessário enviar apenas o objetivo atual e a área de cobertura do campo potencial, pois os outros parâmetros permanecem constantes durante a sua execução. Os atributos v e ε também costumam manter-se constantes durante a execução do algoritmo, sendo poucos os casos onde eles são modificados com freqüência. Após o relaxamento, apenas os valores de potencial das células são buscados do *Device*.

Os atributos *s*, l_x^k e l_y^k são constantes de tal forma que uma mudança em algum deles significa reconstruir toda a estrutura de dados. Isso é algo que raramente ocorre durante o planejamento de caminhos. Por isso, eles são replicados na memória constante do *Device*, obtendo o benefício do cache dessa área. Devido ao espaço bastante limitado da memória constante, os outros atributos são replicados na memória global.

4.3.2 Detecção dos Obstáculos e Atualização do Mapa Local

A detecção dos obstáculos foi dividida em duas etapas. Na primeira, ocorre a detecção de obstáculos estáticos, e na segunda é feita a detecção dos obstáculos dinâmicos. A ordem em que elas são executadas pode ser alternada, sem influência nos resultados.

A detecção dos obstáculos estáticos é realizada num passo separado, aproveitando-se do pré-processamento realizado no mapa global. Conforme foi visto no item 3.6.2, os obstáculos estáticos são mapeados no mapa global nessa etapa inicial do algoritmo. Assim, torna-se redundante haver uma estrutura de dados para armazenar os obstáculos estáticos no *Device* (incluindo a sua posição e dimensão).

A idéia então passa a ser a seguinte: para cada célula da *u-zone* do mapa local de um agente, é criada uma thread. Essa thread irá buscar a célula do mapa global que cobre a mesma região coberta pela célula correspondente no mapa local. Então, o valor de potencial da célula do mapa global é atribuído à célula do mapa local. Dessa forma, se houver um obstáculo sobre aquela região no mapa global, o valor de potencial 1 será copiado. Caso contrário, o valor do potencial da célula relaxada (e por conseqüência, livre) será copiado.

Se forem consideradas todas as células da *u-zone* de todos os agentes virtuais, o número de threads necessárias pode ser bem maior que o limite de 512 threads por bloco, imposto pelo CUDA. Por esse motivo, é realizado o mapeamento de um mapa local por bloco. Dessa forma, em cada bloco são utilizadas as threads necessárias para atualizar cada uma das células do mapa local correspondente.

Um pequeno problema nesta forma de uso das threads é que, se os mapas locais tiverem tamanhos diferentes, será necessário que cada bloco tenha um número de threads suficiente para atender ao maior campo potencial. Nesse caso, é possível que algumas threads precisem ser criadas, mas não executem atividade alguma.

Na figura 4.4, pode ser visto como é feito este mapeamento entre o mapa global e o mapa local. Na cena, um agente virtual caminha próximo a um obstáculo. O obstáculo está representado pelas células do mapa global que cobrem a região ocupada por ele (o obstáculo em si está oculto, mas é percebido pelo agente virtual). Cada um dos quadrados marcados com um "X" representam a região de uma célula do mapa global ou do mapa local com valor de potencial igual a 1. Os quadrados marcados com um tom esverdeado representam algumas das células do mapa global aonde o obstáculo foi mapeado na etapa de pré-processamento. Perceba que algumas células do mapa local do agente se sobrepõem às células marcadas do mapa global. As células marcadas com um tom azulado são células da *u-zone* do mapa local do agente cujo valor de potencial é igual a 1, e que foram copiados do mapa global.

Após o mapeamento dos obstáculos estáticos, é realizado o mapeamento dos obstáculos dinâmicos. Para este caso, é criado uma thread para cada um dos obstáculos dinâmicos, para cada agente virtual. Essa thread irá verificar se o obstáculo está contido na região do mapa local e, se estiver, irá atribuir às células que cobrem a posição ocupada por ele o valor de potencial 1.

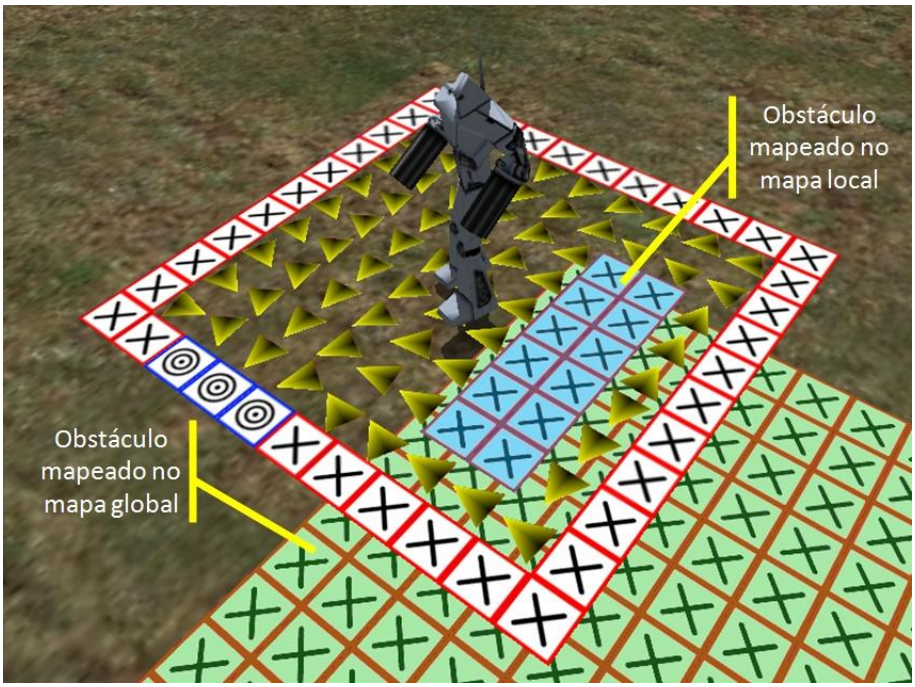


Figura 4.4: Mapeamento de obstáculos estáticos

Da mesma forma que a etapa anterior, o processamento é dividido entre blocos de threads. Ou seja, é criado um bloco de threads para cada agente virtual, e cada thread do bloco irá tratar um único obstáculo dinâmico do cenário.

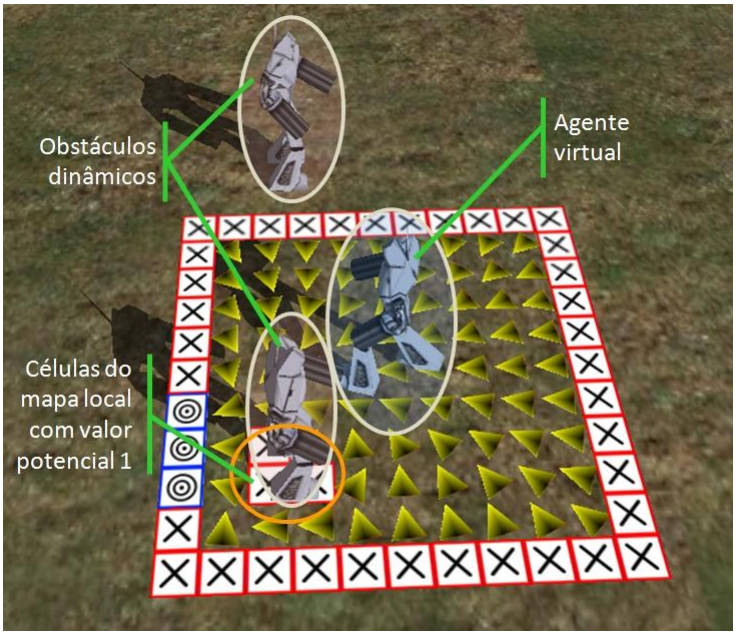


Figura 4.5: Mapeamento de obstáculos dinâmicos

A situação de mapeamento de obstáculos dinâmicos está representada na figura 4.5. Nela, o campo potencial de um agente virtual está em exibição. Na cena existem outros dois agentes virtuais que são vistos como obstáculos dinâmicos pelo primeiro agente. Porém, apenas um desses agentes está no campo de visão do primeiro agente. Dessa forma, o valor potencial 1é atribuído a essas células que cobrem a área onde o obstáculo dinâmico se encontra.

4.3.3 Criação de um Objetivo Intermediário no Mapa Local

Esta etapa é semelhante ao estudo realizado no item 4.1.4 deste trabalho. Para cada agente, é criada uma thread, que calcula quais são as células da fronteira do mapa local (*b-zone*) que devem receber o valor de potencial 0.

O único detalhe onde foi necessário realizar uma adaptação da idéia é relativo ao número de agentes na cena. No caso de haver mais de 512 agentes, é necessário dividir em dois ou mais grupos, onde cada grupo terá até 512 agentes e será processado por um bloco.

Nas figuras 4.4 e 4.5, as células marcadas com pequenos círculos e uma borda azul possuem o valor de potencial 0.

4.3.4 Relaxamento do Mapa Local

Conforme já citado, foi utilizado o método de Jacobi ao invés do método de Gauss-Seidel para fazer o relaxamento do campo potencial. Com isso, é possível calcular o valor de potencial de cada célula em uma iteração em paralelo, pois os cálculos envolvem apenas valores calculados na iteração anterior.

Com isso, é possível criar uma thread para cada célula. Essa thread irá calcular o valor de potencial da célula associada, e irá se sincronizar com as outras threads, que também estão realizando a mesma tarefa. Essa seqüência é realizada durante o número de iterações utilizado para relaxar o campo potencial. Ao final, o campo potencial estará relaxado.

De forma semelhante a outros passos, cada mapa local é relaxado em um bloco de threads. Com isso, a sincronização das threads é realizada apenas entre aquelas relacionadas ao mesmo campo potencial, não influenciando na execução de outros campos potenciais, que também são calculados em paralelo.

Uma otimização importante a ser destacada é quanto ao uso da memória compartilhada do multiprocessador. Como o valor de potencial de cada célula será lido e reescrito diversas vezes durante o relaxamento, no inicio dessa etapa os valores de potencial são copiados para a memória compartilhada, e escritos na memória global apenas no fim do relaxamento. Conforme já foi citado, a memória compartilhada é muito mais rápida do que a memória global, tendo um impacto direto na execução do relaxamento.

Outro ponto que deve ser ressaltado é quanto ao uso do método de Jacobi. Para ser corretamente utilizado, os valores envolvidos nos cálculos devem ser todos da iteração anterior, conforme já foi citado. Para que isso pudesse ser implementado, seria necessário duplicar a memória utilizada para armazenar os mapas locais durante o relaxamento, o que permitiria trabalhar os valores entre cada iteração entre os dois espaços da forma correta. Porém, devido à limitação de tamanho da memória compartilhada e à velocidade de acesso à memória, optou-se por não se fazer desta forma. A cada iteração, as threads utilizam os valores das células, que já podem ter sido calculadas ou não durante a iteração. Ou seja, não foi garantido que a cada iteração, os valores utilizados eram da

iteração anterior. Mas mesmo sem garantir essa característica do método de Jacobi, não foi possível se perceber nenhum problema ao se fazer dessa forma, com os caminhos gerados tendo o mesmo nível de qualidade obtido com a técnica original.

4.3.5 Cálculo dos Mapas Globais

A solução adotada para relaxar os mapas locais requer que seja criada uma thread por célula do mapa local, e que todas as threads pertençam ao mesmo bloco. Se as threads não pertencerem ao mesmo bloco, elas não poderão ser sincronizadas, e por isso não é possível garantir que a cada iteração será utilizado o valor da iteração anterior.

Como já foi citado, o CUDA limita em 512 o número máximo de threads por bloco. Ou seja, se um campo potencial possuir mais do que 512 células, um bloco de threads não é suficiente para relaxá-lo, e uma estratégia diferente deve ser estudada. Como os mapas globais podem possuir centenas de vezes mais células do que este limite, a solução aplicada para mapas locais não é aplicável a mapas globais.

Pelo mesmo motivo, mapas locais muito grandes não podem ser calculados da forma proposta. Mas como os mapas locais são intencionalmente pequenos, na prática essa limitação é aplicada apenas para os mapas globais.

Além disso, se um campo potencial for relaxado em vários blocos, o compartilhamento de dados só poderá ocorrer através da memória global, o que é muito ineficiente.

Mas apesar da limitação, os campos potenciais globais precisam ser calculados apenas uma vez por objetivo no cenário virtual. E como esse cálculo é realizado na fase de pré-processamento, não há impacto no desempenho em tempo real do planejador de caminhos.

Dessa forma, apesar das características paralelas apresentadas, o cálculo dos mapas globais não foi implementado de forma paralela neste trabalho. O fato de a solução adotada para o relaxamento dos mapas locais não ser escalável para mapas com muitas células exige que outra forma deva ser estudada. Mas ainda assim, considera-se que o relaxamento dos mapas globais não tem influência no uso do algoritmo em tempo real, e por isso não foi encarado como uma deficiência grave deste trabalho.

5 EXPERIMENTOS REALIZADOS

Ao realizar a implementação de um algoritmo em uma arquitetura paralela, é natural que diversos testes sejam realizados, a fim de se obter números que ilustrem o ganho obtido.

Com a conclusão da versão paralela do algoritmo, foi realizada uma série de testes para verificar se houve ganho de desempenho, e de quanto foi esse ganho. Para tal, o algoritmo foi executado em sua versão seqüencial e em sua versão paralela, e estatísticas sobre a sua execução foram anotadas. Tais estatísticas vão permitir a sua comparação e extração de outras informações interessantes sobre a sua execução.

5.1 Trabalhos Anteriores Relacionados ao Desempenho do Algoritmo

Em seu trabalho, Silveira (SILVEIRA, 2008) percebeu que a maior parte do tempo de processamento era gasto na etapa de relaxamento dos campos potenciais, ou seja, para se encontrar o resultado da equação 3.6. Além do tamanho do campo potencial, o número de iterações necessário para obter o resultado da equação 3.6 tem influência direta no processamento necessário. Dessa forma, Silveira procurou formas para reduzir o número de iterações necessárias.

No trabalho realizado por Dapper (DAPPER et al., 2007), antes de calcular a equação 3.6, eram utilizados valores de potencial fixos para inicializar cada uma das células livres. Estes valores estavam fixos em 0,5 para as células livres, 1 para as células ocupadas, e 0 para as células que continham o objetivo. As células livres eram inicializadas com o valor constante a cada iteração do algoritmo.

Silveira percebeu que, após relaxar o campo potencial, muitas células livres passavam a ter valor potencial muito próximo a 1. Apenas as células próximas àquelas que continham o objetivo possuíam o valor de potencial mais baixo. Dessa forma, a escolha do valor fixo 0,5 para inicializar as células fazia com que o algoritmo precisasse de mais iterações para que fosse relaxado.

Dessa forma, Silveira fez diversos testes, inicializando as células com valores diferentes e contabilizando o número de iterações necessárias para se alcançar o relaxamento. Analisando os resultados obtidos, ele percebeu que, quanto maior o campo potencial, mais próximo de 1 os valores de inicialização deveriam ser para que o menor número de iterações seja necessário. Por exemplo, para relaxar um mapa com até 25 × 25 células deveriam ser utilizados valores próximos a 0,9 para inicializar as células vazias. Já para mapas maiores que 50 × 50, um valor mais próximo a 1 é recomendado.

Ainda com o objetivo de reduzir o número de iterações necessárias para relaxar os mapas locais, Silveira percebeu outros detalhes que lhe auxiliaram a alcançar o objetivo. Em primeiro lugar, o método Gauss-Seidel aproxima os valores de potencial das células

a cada iteração. Ou seja, quanto mais próximo do valor final estiver o potencial de cada célula, mais rapidamente o método alcança a solução. Em segundo lugar, a cada iteração do algoritmo o espaço de configuração do agente muda discretamente. Ou seja, a cada passo do agente virtual, os obstáculos se afastam ou se aproximam levemente do agente virtual, sendo então mapeados no mapa local do agente em uma posição próxima àquela mapeada no passo anterior.

Assim, Silveira passou a inicializar o mapa local com o valor constante apenas na primeira iteração do algoritmo. Nas seguintes, o mapa não é inicializado. Os valores obtidos em uma iteração do algoritmo são utilizados na iteração seguinte para inicializar o mapa local. Dessa forma, como os obstáculos e o objetivo mudaram apenas um pouco em relação ao passo anterior, muitas células serão inicializadas com um valor de potencial bastante próximo do seu valor final, e com isso o número de iterações é reduzido.

Como se percebe, as otimizações sugeridas por Silveira tem um foco na forma com que os campos potenciais são inicializados. Nas próximas seções, será discutido os testes executados sobre a implementação que foi desenvolvida com base nos aspectos já discutidos no capítulo 4. As otimizações de inicialização foram utilizadas na implementação paralela, e já estavam implementadas na versão seqüencial do algoritmo.

5.2 Comparação Entre a Versão Seqüencial e a Paralela do Algoritmo

Segundo Mazarakis e Avaritsiotis (MAZARAKIS; AVARITSIOTIS, 2005), uma pessoa caminhando devagar realiza novos passos em uma freqüência de cerca de 0,9Hz. Já uma pessoa correndo rapidamente pode alcançar a freqüência de 3,5Hz ao dar seus passos.

Dessa forma, pode-se considerar que o algoritmo pode ser executado em tempo real se for possível executar o algoritmo de planejamento de caminhos para cada agente nas freqüências indicadas. Mas, além disso, deve se levar em conta que a cena deverá ser desenhada numa freqüência de 30Hz a 60Hz para que a animação seja suave ao ser visualizada.

Como é de se imaginar, quanto maior a freqüência de execução do algoritmo, maiores são os recursos computacionais necessários para que tal freqüência seja sustentada. Dessa forma, a freqüência com que o algoritmo pode ser executado é uma métrica que pode ser utilizada para comparar as suas configurações de execução e as suas implementações. Quanto maior a freqüência de execução, mais eficientemente o algoritmo foi implementado, e mais eficientemente ele usa os recursos computacionais disponíveis.

Por isso, a comparação entre a versão seqüencial e a versão paralela do algoritmo será feita com base na freqüência de execução obtida. O algoritmo foi executado em um computador com a seguinte configuração: Intel Core 2 6300 1,86Ghz, 2 Gb de memória RAM, uma placa gráfica nVidia GeForce 9800 GX2, sistema operacional Microsoft Windows XP SP3. Durante os testes com a versão paralela e seqüencial, não houve nenhuma alteração na configuração do sistema.

Para executar os testes, foram definidas três configurações do algoritmo, variando o tamanho do mapa local dos agentes. Os tamanhos dos mapas locais foram escolhidos para os testes por que apresentam resultados satisfatórios quanto à qualidade da animação gerada, e por isso tendem a ser os mais utilizados em outras situações. Como a noção de “animação com qualidade satisfatória” é subjetiva, a escolha dos tamanhos dos mapas locais também acabou sendo feita de forma subjetiva.

Após definir o tamanho do mapa local, o algoritmo foi executado diversas vezes, onde a cada execução o número de agentes foi alterado. Para cada combinação entre tamanho do mapa local e número de agentes, o algoritmo foi executado por cerca de 30 segundos. Então, foi anotado a freqüência média que foi possível executar o algoritmo seqüencial e paralelo, considerando-se e desconsiderando-se a renderização da cena. Com os resultados obtidos, foi dividida a freqüência obtida para a versão paralela pela freqüência obtida para a versão seqüencial, e o resultado é apresentado na coluna "Melhoria", representando quantas vezes a versão paralela foi melhor do que a seqüêncial.

Deve-se ressaltar que, para renderizar a cena, foram utilizadas apenas as otimizações fornecidas automaticamente pelo motor gráfico Ogre. Otimizações, como nível de detalhes (LOD, *Level of Detail*), não foram utilizadas.

Além disso, o algoritmo de planejamento de caminhos foi executado a cada frame do programa. Isso é útil para testar a sua execução e desempenho. Porém, em aplicações de uso real, o algoritmo deverá ser executado em uma freqüência fixa, independente da taxa de quadros da aplicação.

5.2.1 Mapas Locais com Tamanho 11×11

Para a primeira configuração de execução, foram utilizados campos potenciais de tamanho 11×11 (121 células) para os mapas locais de todos os agentes. Durante os testes, o número de agentes na cena foi sendo modificado. Os resultados obtidos estão listados na tabela 5.1. Esses dados também estão apresentados no gráfico da figura 5.1. Considerando a fase de rendering da aplicação, os resultados obtidos estão apresentados na tabela 5.2.

Tabela 5.1: Freqüência de execução utilizando campo potencial 11×11

Agentes na Cena	Versão Seqüencial	Versão Paralela	Melhoria (vezes)
100	304,785000	871,051000	2,8579195170
500	42,580500	290,661000	6,8261528164
1000	17,235200	93,297000	5,4131660787
1500	8,451870	35,810800	4,2370268355
2000	5,224490	22,210000	4,2511326464
2500	3,559220	14,904700	4,1876309978
3000	2,691690	11,703800	4,3481232980
3500	2,003260	8,743560	4,3646655951
4000	1,402000	6,733030	4,8024465050

Tabela 5.2: Freqüência de execução utilizando campo potencial 11×11, considerando o rendering da aplicação

Agentes na Cena	Versão Seqüencial	Versão Paralela	Melhoria (vezes)
100	111,618000	146,277000	1,3105144332
500	19,699200	32,785000	1,6642807830
1000	8,928160	15,553000	1,7420162721
1500	5,029110	9,227410	1,8347997956
2000	3,338610	6,559680	1,9647937315
2500	2,410600	4,968240	2,0609972621
3000	1,873870	4,067130	2,1704440543
3500	1,449740	3,302180	2,2777739457

4000	1,066860	2,750460	2,5780889714

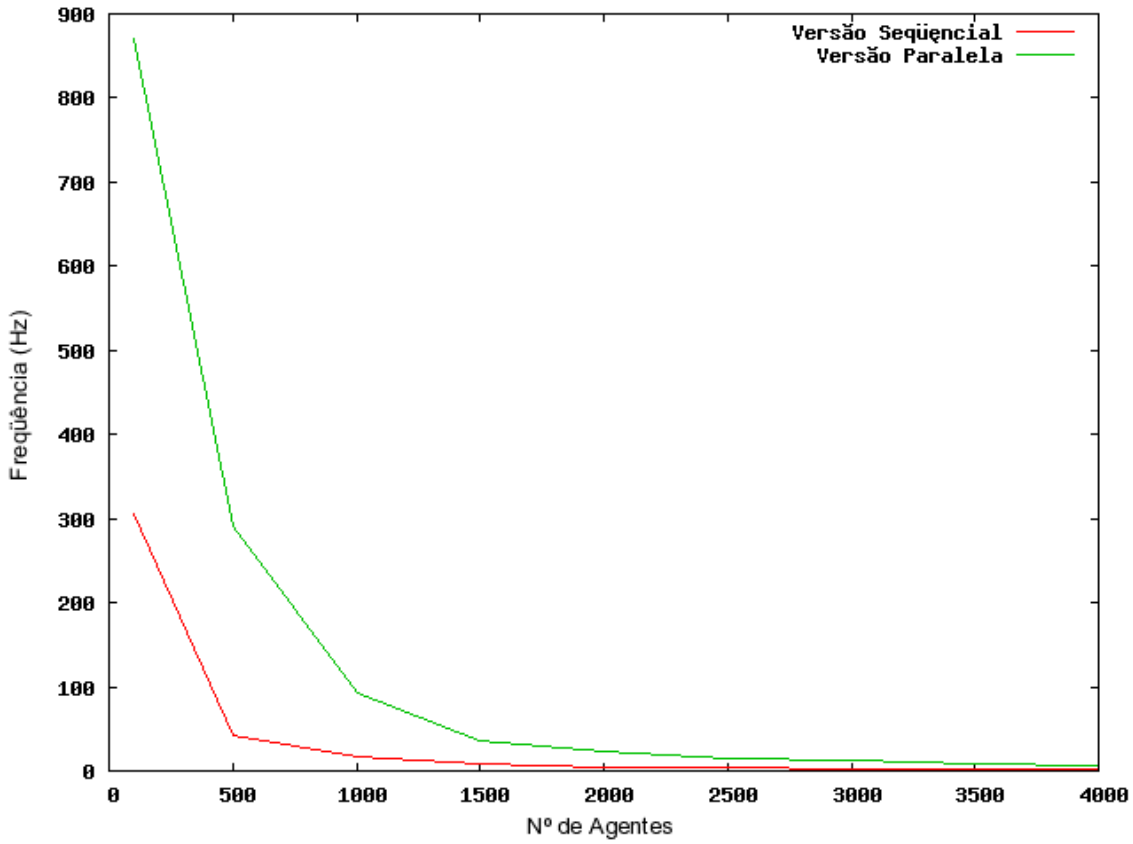


Figura 5.1: Freqüência de execução utilizando campo potencial 11×11

5.2.2 Mapas Locais com Tamanho 16×16

Na segunda bateria de testes, os campos potenciais dos agentes tinham o tamanho 16×16 (256 células). Assim como na primeira parte dos testes, apenas o número de agentes na cena foi modificado. Os resultados obtidos estão listados na tabela 5.3, e também estão apresentados no gráfico da figura 5.2. Ao levar em consideração o rendering da cena, o número de quadros por segundo obtido está apresentado na tabela 5.4.

Tabela 5.3: Freqüência de execução utilizando campo potencial 16×16

Agentes na Cena	Versão Seqüencial	Versão Paralela	Melhoria (vezes)
100	55,3408	736,449000	13,3075235631
500	11,5075	212,364000	18,4543993048
1000	2,77825	77,656900	27,9517322055
1500	1,753010	32,382900	18,4727411709
2000	1,285090	20,401900	15,8758530531
2500	0,940056	15,144400	16,1101040789
3000	0,709194	10,983200	15,4868766515
3500	0,650381	8,243820	12,6753702830
4000	0,570495	6,382470	11,1876002419

Tabela 5.4: Freqüência de execução utilizando campo potencial 16×16, considerando o rendering da aplicação

Agentes na Cena	Versão Seqüencial	Versão Paralela	Melhoria (vezes)
100	41,794700	140,873000	3,370595
500	8,795430	31,408400	3,570991
1000	2,413690	15,020200	6,222920
1500	1,534840	8,902260	5,800122
2000	1,127550	6,332540	5,616194
2500	0,834685	4,948890	5,929051
3000	0,633918	3,925290	6,192110
3500	0,578883	3,193580	5,516797
4000	0,507333	2,685730	5,293821

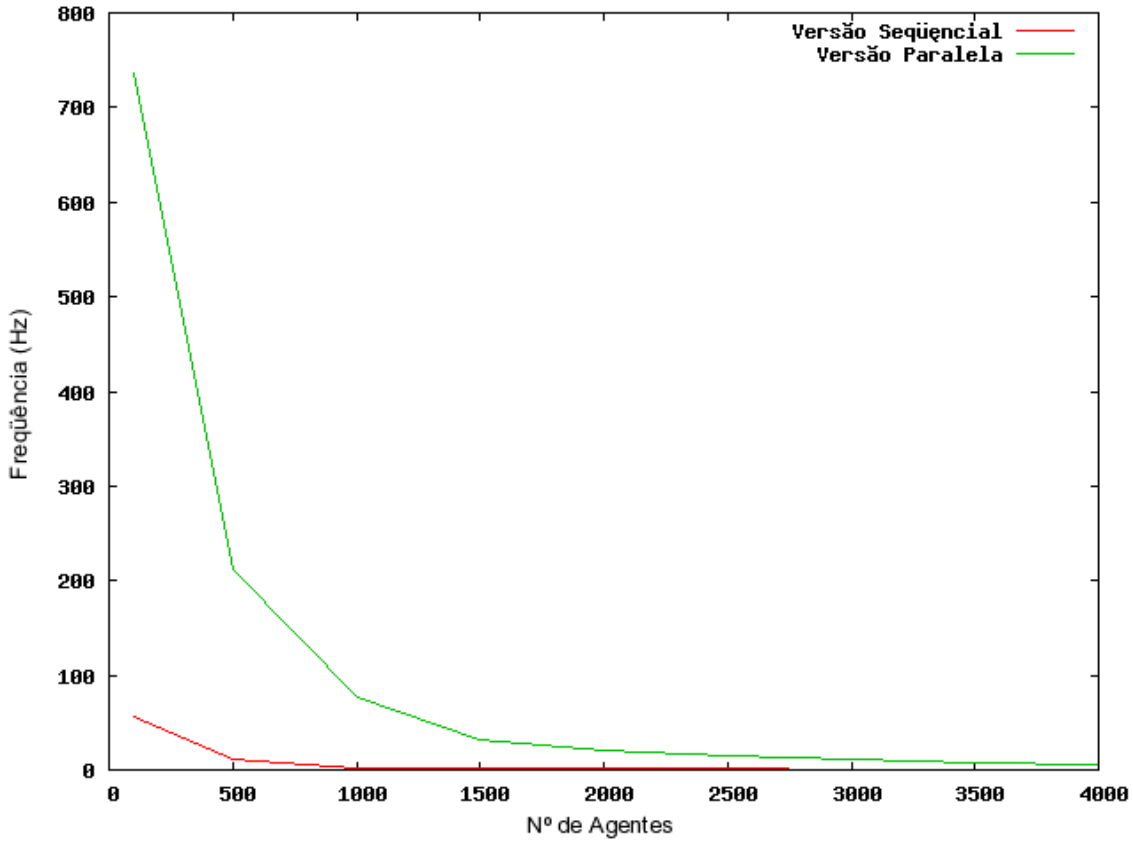


Figura 5.2: Freqüência de execução utilizando campo potencial 16×16

5.2.3 Mapas Locais com Tamanho 21×21

Na terceira bateria de testes, o tamanho do mapa local dos agentes na cena passou para 21×21 (441 células). Da mesma forma que nos testes anteriores, o número de agentes foi modificado. Os resultados obtidos estão listados na tabela 5.5, e estão apresentados no gráfico da figura 5.3. Considerando o rendering, os resultados obtidos estão apresentados na tabela 5.6.

Tabela 5.5: Freqüência de execução utilizando campo potencial 21×21

Agentes na Cena	Versão Seqüencial	Versão Paralela	Melhoria (vezes)
100	13,028100	257,328000	19,7517673337
500	2,174100	123,072000	56,6082516904
1000	1,046800	51,376300	49,0793847917
1500	0,637785	24,523900	38,4516725856
2000	0,459638	15,907900	34,6096275765
2500	0,360970	12,203800	33,8083497244
3000	0,322672	8,998950	27,8888468786
3500	0,260701	6,886950	26,4170448138
4000	0,235294	5,470090	23,2478941239

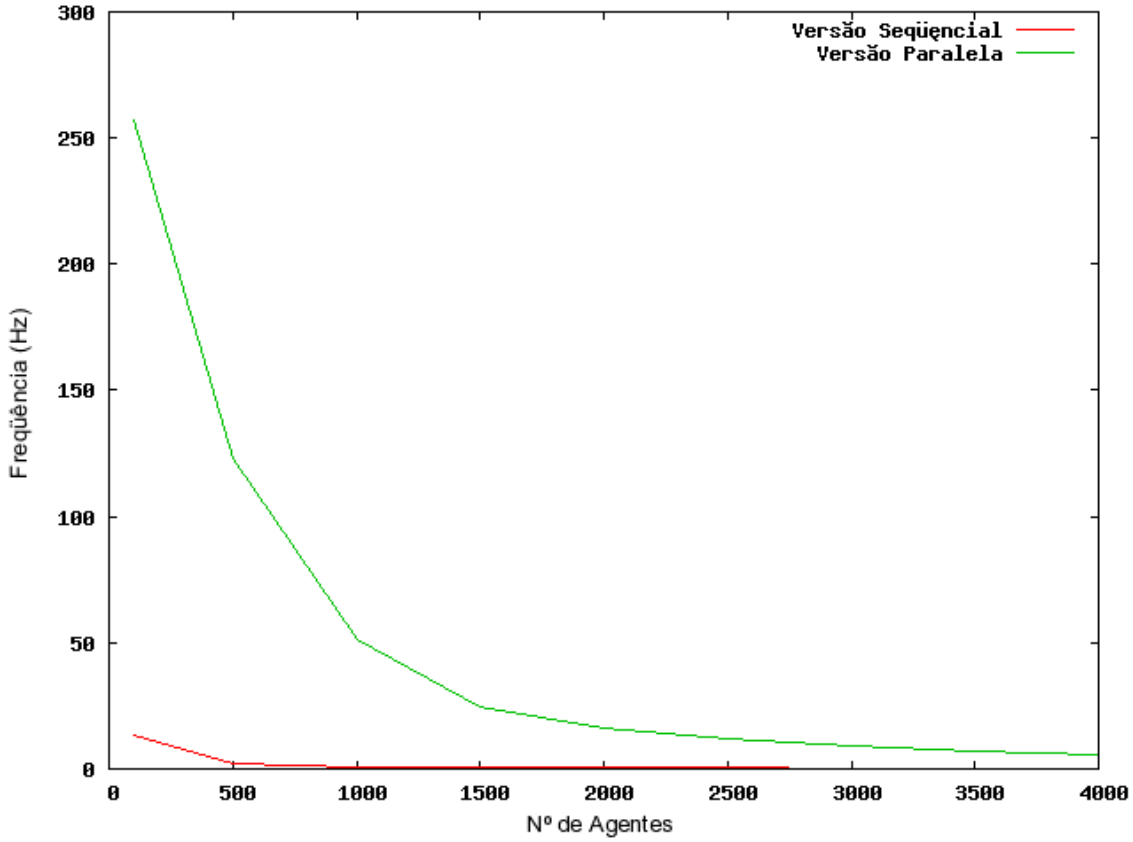


Figura 5.3: Freqüência de execução utilizando campo potencial 21×21

Tabela 5.6: Freqüência de execução utilizando campo potencial 21×21, considerando o rendering da aplicação

Agentes na Cena	Versão Seqüencial	Versão Paralela	Melhoria (vezes)
100	12,117100	127,901000	10,5554134240
500	2,051690	28,366800	13,8260653413
1000	0,989053	13,568100	13,7182739449
1500	0,605917	8,179980	13,5001658643
2000	0,437517	5,834140	13,3346589961
2500	0,344132	4,591220	13,3414503737
3000	0,306312	3,650140	11,9164120243
3500	0,248259	2,949150	11,8793276377
4000	0,223624	2,509800	11,2233034021

5.2.4 Tempo de Transferência de Dados entre Host e Device

Durante a execução dos testes anteriores, foi levantada a dúvida sobre qual o impacto da cópia de dados entre a memória do processador e a da placa gráfica sobre o tempo total de processamento dos mapas locais. Desta forma, além do tempo de execução total do algoritmo, foi também anotado o tempo utilizado apenas para transferir dados entre os dois espaços de memória. Os resultados obtidos estão apresentados na tabela 5.7.

A partir dos dados obtidos na tabela 5.7, foi calculado o quanto o tempo de cópia de dados entre o *Host* e o *Device* representa do tempo total de processamento necessário para executar o algoritmo de planejamento de caminhos, em sua versão paralela. Os resultados estão apresentados na forma de gráfico, na figura 5.4. Ressalta-se que o tempo total de execução do algoritmo na placa gráfica já inclui o tempo de transferência entre os dois espaços de memória.

Tabela 5.7: Tempos de execução do algoritmo e da cópia de dados entre *Host* e *Device*

Agentes na Cena	Mapa Local 11×11		Mapa Local 16×16		Mapa Local 21×21	
	Tempo Total	Tempo de Cópia	Tempo Total	Tempo de Cópia	Tempo Total	Tempo de Cópia
100	0,00114804	0,00075158	0,0180698	0,00085491	0,07675730	0,00094915
500	0,00344043	0,00071633	0,0869001	0,00089688	0,45996100	0,00131431
1000	0,01071850	0,00086805	0,3599390	0,00148191	0,95529500	0,00197545
1500	0,02792460	0,00099571	0,5704470	0,00188925	1,56793000	0,00261415
2000	0,04502470	0,00143914	0,7781580	0,00222402	2,17563000	0,00315691
2500	0,06709280	0,00178180	1,0637700	0,00257365	2,77031000	0,00381348
3000	0,08544230	0,00183824	1,4100500	0,00295188	3,09912000	0,00433683
3500	0,11437000	0,00199855	1,5375600	0,00333659	3,83582000	0,00514221
4000	0,14852200	0,00347222	1,7528600	0,00378418	4,25000000	0,00548475

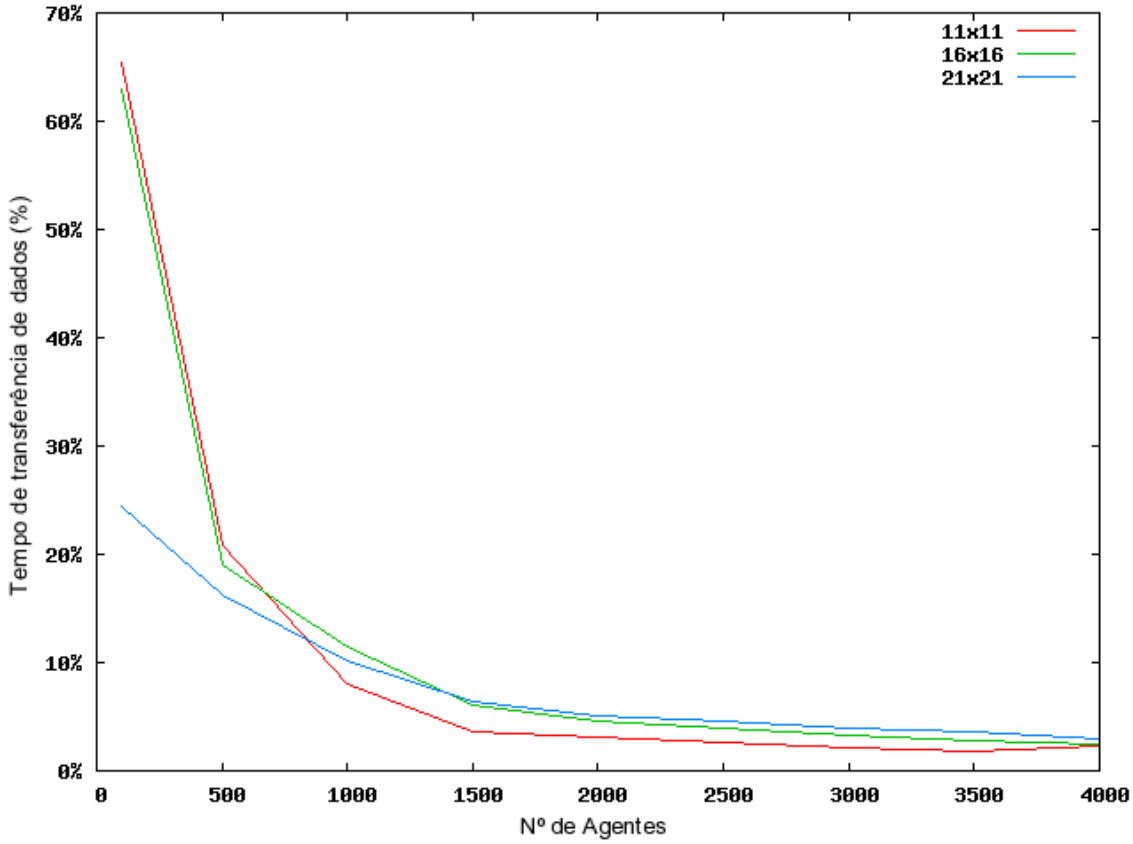


Figura 5.4: Proporção entre o tempo de transferência de dados entre *Host* e *Device* e o tempo total de execução do algoritmo

5.2.5 Análise dos Resultados Obtidos

Antes de analisar os resultados, outro gráfico será apresentado. O gráfico da figura 5.5 refere-se à coluna "Melhoria" das tabelas 5.1, 5.3 e 5.5, reunindo os valores ali obtidos de forma a facilitar a sua comparação.

Como já foi dito, a melhoria obtida em um teste é calculada dividindo a freqüência de execução da versão paralela pela freqüência de execução da versão seqüencial. Além disso, uma maior freqüência de execução indica um melhor desempenho do código executado. Desta forma, é possível dizer que, sempre que a melhoria obtida no gráfico for maior que 1, a versão seqüencial apresentou desempenho superior à versão paralela.

Ao observar o gráfico, percebe-se que em todos os testes, a versão paralela apresentou um desempenho superior à versão seqüencial. O ponto mais baixo do gráfico é representado pela curva de melhoria do mapa local de tamanho 11×11 para 100 agentes, e foi mais de 2 vezes mais rápida. Já o ponto mais alto é apresentado na curva do mapa local de tamanho 21×21 com 500 agentes, onde o desempenho foi mais de 56 vezes o da versão seqüencial.

Porém, a diferença de desempenho obtida pelas diversas configurações deve ser estudada mais profundamente. Afinal, para mapas locais pequenos, o ganho em desempenho foi bem menor que o ganho obtido para mapas locais grandes. Além disso, a curva de melhoria para mapas locais de tamanho 11×11 é bem mais suave do que as outras curvas.

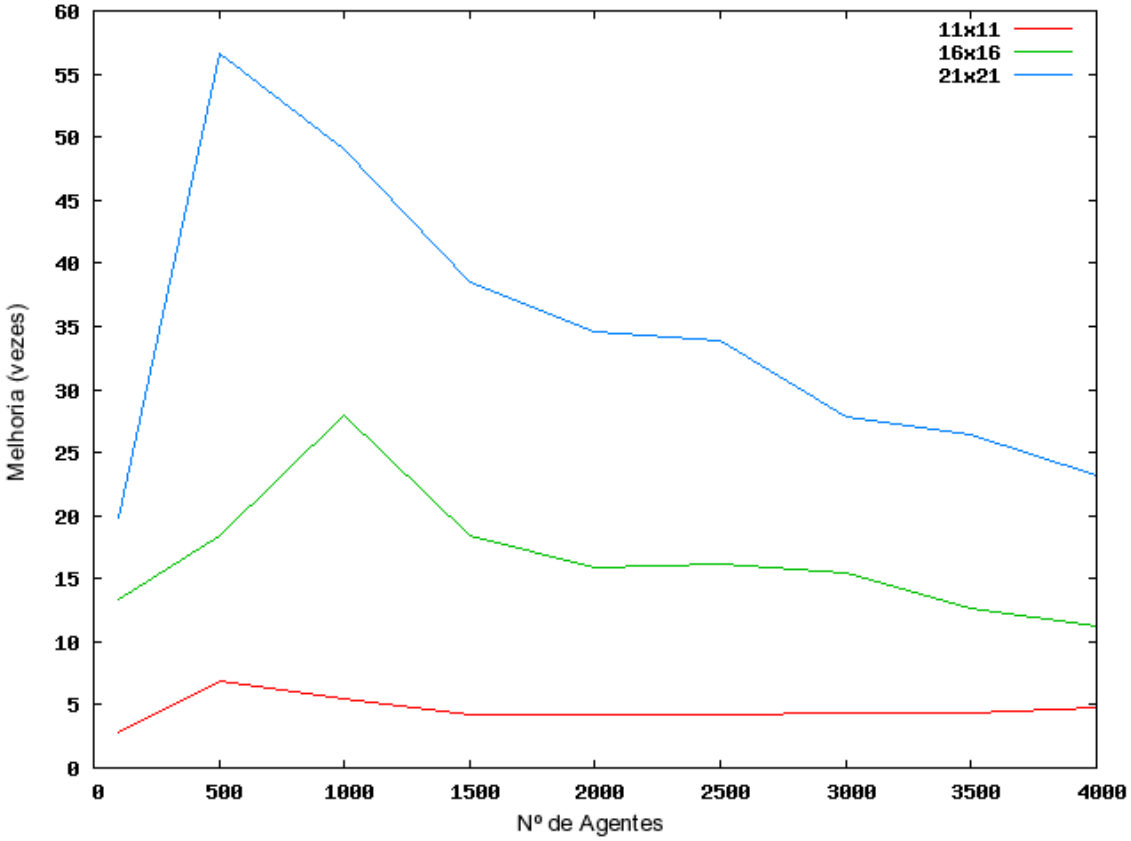


Figura 5.5: Melhoria obtida nas três baterias de testes realizados

Uma das principais recomendações da documentação do Cuda (CUDA, 2008) é de que quanto maior for o número de threads por bloco, maior será o aproveitamento do hardware da placa gráfica. Com o mapeamento de um bloco por mapa local descrito no capítulo anterior, ao utilizar o tamanho 11×11 serão criadas 121 threads por bloco. Este número representa menos de 25% do limite de 512 threads por bloco. Já para os mapas locais maiores, o tamanho 16×16 precisa criar 256 threads, e o tamanho 21×21 precisa de 441 threads. Dessa forma, quanto maior o tamanho do mapa local, maior o aproveitamento dos recursos da placa gráfica.

Por outro lado, mapas potenciais pequenos podem ter um grande benefício da cache do processador do *Host* utilizado, já que estes podem ser armazenados em espaços muito pequenos de memória. Além disso, a freqüência de trabalho dos processadores atuais é algumas vezes maior que a freqüência de trabalho dos processadores das placas gráficas. Por esses fatores, mais o fato de mapas locais pequenos precisarem de poucas iterações para convergirem, o desempenho da versão seqüencial foi tão grande que acabou por reduzir o ganho obtido pela versão paralela. Já para os mapas locais maiores, um maior espaço de memória é necessário para armazená-los, e um maior número de iterações deve ser feito. Nestes casos, o ganho obtido com a implementação paralela foi mais "visível".

Também deve se levar em conta que, apesar de a implementação ter sido paralela, ela não é executada totalmente em paralelo. Cada modelo de placa gráfica é capaz de trabalhar com um número máximo de blocos por vez, e os blocos excedentes são "enfileirados" para uma execução posterior. De forma semelhante, em cada bloco apenas algumas threads são executadas realmente em paralelo, enquanto outras aguardam por sua execução. A troca de contexto entre as threads possui custo muito pequeno, mas isso não elimina o fato de que muitas delas são executadas de forma seqüencial. Segundo a documentação do Cuda (CUDA, 2008), um grupo de threads que executa realmente em paralelo se chama *warp*, e atualmente esse número é limitado a 32 threads por bloco.

Isso impede que, na prática, por exemplo, o relaxamento de um mapa local possa ser realizado em *n* passos, onde *n* é o número de iterações utilizado para relaxar o campo potencial. Por exemplo, ao relaxar um mapa local de tamanho 11×11 serão criadas 121 threads paralelizáveis. Mas no máximo 32 de cada vez irão executar realmente em paralelo.

De forma semelhante, as três curvas desenhadas no gráfico parecem convergir, cada uma, para uma constante, pois elas parecem tender a tornarem-se horizontais conforme aumenta o número de agentes na cena. Não foi possível detectar se existe essa constante com os testes realizados. De qualquer forma, o número máximo de blocos em execução em um determinado momento combinado com a forma com que os mapas locais são mapeados nos blocos sugere que com um grande número de agentes na cena a melhoria em relação à versão seqüencial do algoritmo seja constante.

Outro ponto a ser analisado são os picos nas 3 curvas do gráfico 5.5. Segundo a documentação do Cuda (CUDA, 2008), cada algoritmo possui uma configuração de execução ótima em um determinado hardware. Tal configuração, que envolve o número de blocos e threads por bloco, é dita ótima por tirar o melhor proveito do paralelismo oferecido pela placa gráfica. A documentação do Cuda sugere que essa combinação entre número de threads e blocos seja encontrada por experimentação, tal como realizado neste trabalho.

Dessa forma, os picos nas 3 curvas são explicados por representarem uma boa combinação entre número de threads e blocos. Conforme já foi dito, o número de threads depende do tamanho do mapa local, e o número de blocos depende do número de agentes na cena. Como pode ser percebido, ao se manter o número de agentes fixo (e por conseqüência, o número de blocos em execução), o aumento do tamanho do mapa local (e, dessa forma, o aumento do número de threads) implica no aumento do ganho obtido. Porém, ao se fixar o tamanho do mapa local, o ganho máximo se dá entre 500 agentes (para os mapas de tamanho 11×11 e 21×21) e 1000 agentes (para os mapas de tamanho 16×16).

Em particular, o ponto máximo no gráfico da figura 5.5 se refere à combinação ótima de número de threads por bloco e número de blocos executados. Nesta configuração, são executados 500 blocos, cada um com 441 threads, totalizando 220.500 threads executadas. Esta combinação de threads e blocos demonstrou ser a mais eficiente nos testes realizados, mas outras combinações não testadas podem demonstrar um ganho ainda maior.

6 OUTRAS CONTRIBUIÇÕES

Antes de implementar a versão paralela do algoritmo de planejamento de caminhos, foram feitas diversas outras contribuições ao projeto. Essas contribuições tiveram o objetivo inicial de permitir que o autor pudesse se integrar ao projeto e conhecer melhor o código existente. Os resultados serão descritos nos itens a seguir.

6.1 Reescrita do Código Existente

Ao ingressar no grupo, a primeira tarefa seria compreender o código existente, escrito na linguagem C++. Nessa etapa ocorreram muitas dificuldades de entendimento desse código existente do planejador de caminhos. Ele estava codificado de uma forma bastante complicada, e os conceitos estavam bastante dispersos.

Alguns dos problemas detectados foi a mistura dos conceitos agente virtual e campo potencial, o conceito de trajetória (usado para definir objetivos de forma iterativa) estava disperso em duas classes distintas, e muitos métodos possuíam uma interface bastante complexa, sendo alguns com muitos parâmetros. Também havia nomes de atributos de classes pouco intuitivos, o que dificulta a sua memorização e o desenvolvimento do código focado no algoritmo em si.

Dessa forma, a primeira atividade prática executada foi a reescrita do código visando separar melhor os conceitos e deixar o código mais limpo e legível. Isso facilitaria os trabalhos seguintes. Essa tarefa é chamada de refatoração de código pela área de engenharia de software. Após a sua conclusão, a tarefa de compreensão do código seria cumprida ao mesmo tempo em que a qualidade do código seria melhorada.

Como resultado desta tarefa, o código foi significativamente melhorado. As classes foram bem definidas, e os métodos foram simplificados. Muitos métodos diferentes, com funções muito semelhantes também foram reescritos em um único, de uma forma mais genérica. A interface dos métodos também foi simplificada, reduzindo o número de parâmetros. É provável que muitas das idéias implementadas na versão paralela do algoritmo surgiram após a refatoração do código existente, e outras teriam uma implementação muito mais complexa se a refatoração não fosse realizada.

Um segundo benefício obtido durante a refatoração foi a documentação do código. Durante este trabalho, as classes foram documentadas justamente para ser possível entendê-las melhor e facilitar a sua refatoração. Atualmente, essa documentação pode ser facilmente extraída do código com o uso da ferramenta Doxygen, e exportada em formatos diversos, como HTML ou PDF.

6.2 Biblioteca Path-Planning

Com a refatoração do código, muito das suas características ficaram mais visíveis. E algumas classes que possuíam varias funções foram separadas em duas ou mais classes, enquanto que outras foram reduzidas a uma única classe. Entre as outras melhorias já citadas, foi possível se separar as funcionalidades de desenho das funcionalidades de planejamento de caminho.

Com a separação de funcionalidades, o código de melhor qualidade e documentado, foi possível desenvolver uma biblioteca independente, apenas com as funcionalidades de planejamento de caminho.

No momento, esta biblioteca (batizada de Path-Planning) está escrita totalmente em C++, e é independente de quaisquer outras bibliotecas (tais como OpenGL ou DirectX). O código pode ser compilado no Microsoft Visual Studio (versões 2005 e 2008), mas acredita-se que a portabilidade para outras plataformas possa ser realizada sem muitas dificuldades.

Por fim, o código está sendo publicado em um sistema de controle de configurações (SVN). O acesso, tanto em leitura quanto em escrita, está restrito aos desenvolvedores do grupo de pesquisa.

6.3 Nova Aplicação Gráfica

Durante a refatoração do código, foi levantada também a possibilidade de melhorar visualmente a aplicação utilizada para testes e gravação de demonstrações. Em particular, considerou-se interessante incluir sombras nos objetos exibidos nas cenas.

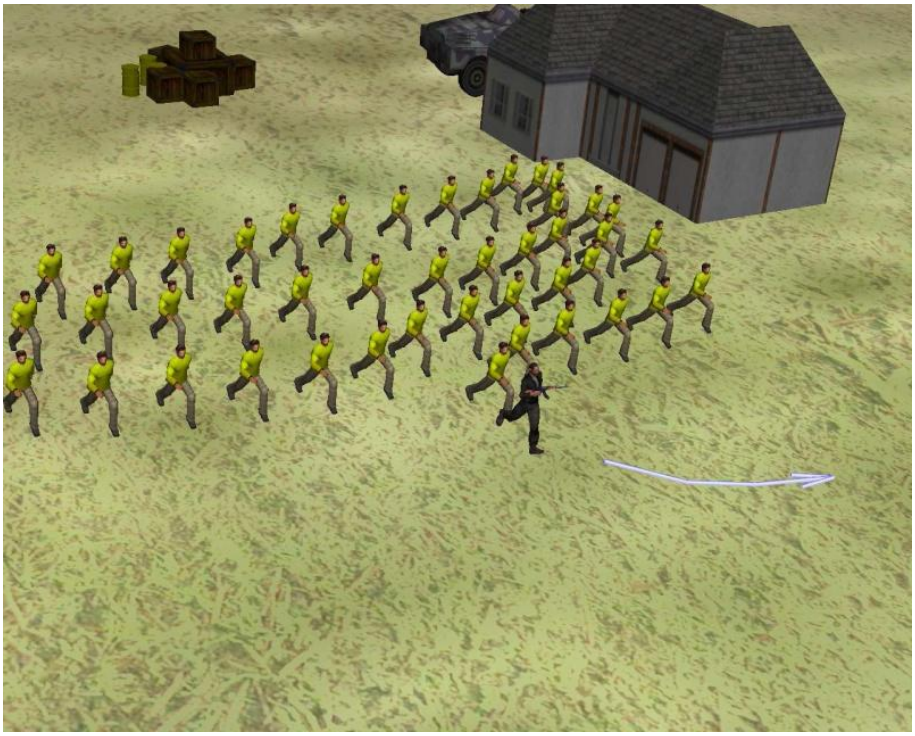


Figura 6.1: Imagem gerada pela versão antiga da aplicação de testes, utilizando OpenSceneGraph (SILVEIRA, 2008)

Como a OpenSceneGraph (OSG, 2008), biblioteca utilizada para gerar as cenas animadas, possui suporte limitado à sombras, o grupo de pesquisa decidiu que a biblioteca de controle de cena da aplicação seria substituída.

Após pesquisar as características de diversas bibliotecas, foi decidido pelo uso da OGRE, Open Graphics Rendering Engine (OGRE, 2008). Esta biblioteca tem como principais características: ser multi-plataforma, OpenSource, orientada a objetos (C++), suporte a grafo de cena, compatível com DirectX e OpenGL, suporte otimizado a diversos tipos de cenário (terrenos abertos, áreas fechadas). Outro fator considerado importante é o suporte oferecido pela sua comunidade de desenvolvedores e usuários, que é bastante ativa. Por fim, o seu suporte a sombras é bastante eficiente e flexível, gerando resultados de boa qualidade.

Com a migração da biblioteca de controle de cena, foi possível melhorar a qualidade visual dos vídeos e testes realizados, incluindo sombra para os objetos estáticos e dinâmicos na cena. A figura 6.1 exibe uma cena renderizada com a biblioteca antiga (OpenSceneGraph), sem sombras. Já as figuras 6.2 e 6.3 exibem cenas renderizadas com a nova biblioteca (OGRE), com sombras e terrenos baseados em mapas de alturas.

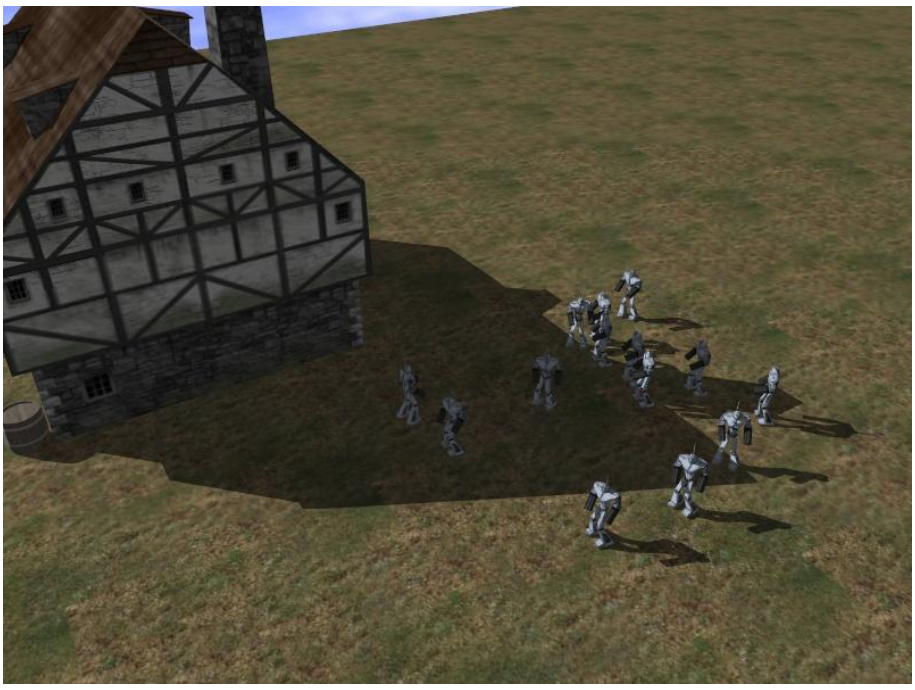


Figura 6.2: Imagem gerada pela aplicação de testes, utilizando a OGRE

Devido às melhorias realizadas até o momento, ficou simples programar um sistema de carga de cenários. Isso passou a facilitar a manutenção de cenários já desenvolvidos e a sua carga durante os testes. Com isso, a construção do cenário de teste é parametrizada por arquivos externos, sendo desnecessário compilar o código para se criar ou utilizar um novo cenário.

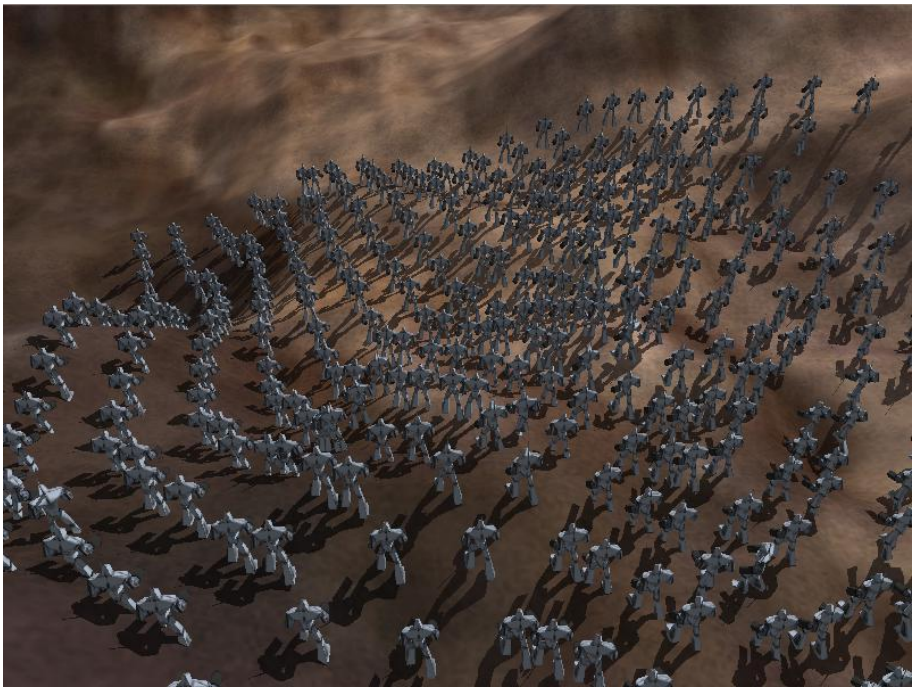


Figura 6.3: Imagem da aplicação executando um cenário com terreno gerado por mapa de alturas

6.4 Divisão Dinâmica dos Agentes em Grupos

Uma das contribuições realizadas pelo trabalho de Silveira (SILVEIRA, 2008) foi incluir a abordagem descrita por Dietrich (DIETRICH et al, 2008) no planejador de caminhos descrito nos capítulos anteriores.

Em aplicações onde o usuário pode interagir com diversos agentes virtuais, é muito comum o uso do mouse para que tal iteração ocorra. Mas para que um comportamento qualquer possa ser definido, é comum que o usuário precise pressionar muitas vezes o botão do mouse nos momentos certos. Esse tipo de iteração exige muita atenção do usuário, e se o local onde são feitos os cliques não forem bem escolhidos, os agentes virtuais podem exibir comportamentos indesejáveis.

Além do mais, a técnica proposta por Dapper (DAPPER et al., 2007) impede que o usuário interaja com os agentes virtuais durante a simulação. Na técnica descrita por ele, os campos potenciais globais precisam ser relaxados a cada definição de objetivo. Como o relaxamento desses campos potenciais é uma tarefa custosa, os objetivos deveriam ser definidos numa etapa de pré-processamento, eliminando qualquer possibilidade de iteração com o usuário.

A seguir, a abordagem citada será descrita. Em seguida, este trabalho propõe uma extensão a essa abordagem, permitindo que diversos agentes possam seguir um determinado caminho até certo ponto, dividindo-se então em grupos com objetivos diferentes.

6.4.1 Especificação de Caminhos em Alto Nível

A idéia é baseada no desenho de esboço do caminho a ser seguido pelos agentes. Tal ação é demonstrada na figura 6.4, onde uma pessoa desenha trajetórias a serem seguidas

por jogadores de basquete em uma jogada. O usuário da aplicação literalmente indica para o agente virtual qual será o caminho que ele irá seguir, e o agente cuida dos detalhes como desviar de obstáculos e dos outros agentes. Na prática, o usuário interage com o sistema como se fosse um comandante dando ordens a um exército, ou um técnico definindo a estratégia a ser seguida pelos jogadores.

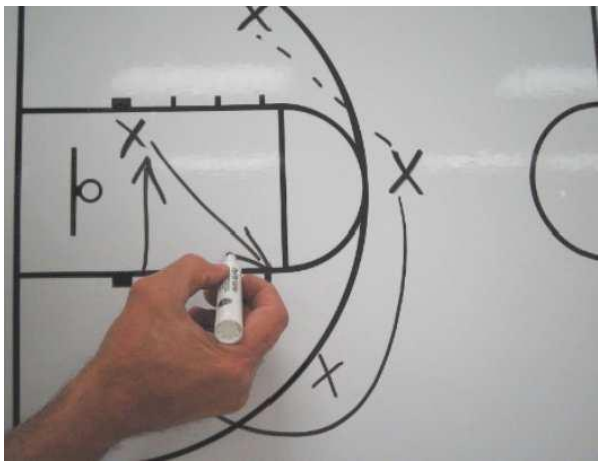


Figura 6.4: Esboço de uma estratégia de um time de basquete (SILVEIRA, 2008)

Durante a execução do programa, o usuário utiliza o mouse para desenhar sobre o terreno o caminho que os agentes virtuais irão seguir. Da linha desenhada, são amostrados alguns pontos. Esses pontos são ordenados pela ordem em que foram desenhados pelo usuário.

Os pontos obtidos são utilizados para definir objetivos intermediários para os agentes. Inicialmente, o primeiro ponto da lista se torna o objetivo de todos os agentes envolvidos. Quando um dos agentes o alcança, o segundo ponto se torna o objetivo intermediário de todos, e assim por diante. Esse processo se repete até que todos os pontos sejam alcançados, na ordem em que foram definidos. Na figura 6.5, o usuário desenhou a trajetória indicada pela seta azul, e alguns pontos foram amostrados. Os agentes então passam a perseguir o primeiro ponto da seta, como sendo seu objetivo intermediário.

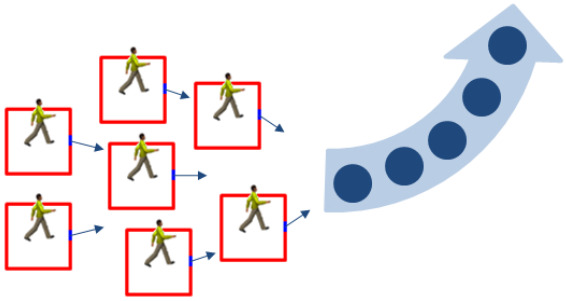


Figura 6.5: Amostra de pontos e mapeamento de objetivos intermediários (SILVEIRA, 2008)

O mapeamento dos objetivos intermediários é feito conforme descrito no item 3.6.4. Dessa forma, os objetivos não são mais buscados no mapa global, evitando seu relaxamento. Com isso, passa a ser possível definir os objetivos em tempo de execução do algoritmo. Além disso, a forma com que essa iteração é feita permite que comportamentos mais complexos do que tomar o menor caminho entre dois pontos sejam descritos muito facilmente.

6.4.2 Ramificação de Caminhos e Divisão em Subgrupos

Como já foi citado, o uso dessa abordagem pode simplificar a descrição de ações mais complexas. Uma das tarefas imaginadas ocorre quando vários agentes virtuais precisam caminhar em grupo até certo ponto, e dividirem-se em grupos distintos em seguida. Um exemplo é o caso onde um grupo de mergulhadores e soldados caminham juntos, até encontrarem um lago. Nesse ponto, dois grupos distintos são criados: o grupo de mergulhadores, que atravessa o lago diretamente, e o dos soldados, que contorna o lago até chegar ao outro lado.

Para poder descrever ações como esta, foi desenvolvida uma estrutura que permite ao usuário desenhar diversas curvas, onde umas iniciam em outras, descrevendo as divisões que irão ocorrer com o grupo. Tal estrutura utiliza uma árvore para armazenar os pontos amostrados das curvas desenhadas pelo usuário. Nessa estrutura, cada nodo n contém um ponto p_n amostrado da curva, e pode conter zero ou mais nodos filhos. A relação entre um nodo pai e seus filhos indica que o ponto associado ao nodo pai foi desenhado pelo usuário antes dos seus filhos.

Além disso, foi desenvolvida uma função de pesquisa nesta árvore. Tal função recebe um ponto p qualquer no cenário e a árvore A, e retorna o nodo p_n tal que a distância entre p e p_n é a menor para todos os nodos de A.

Com essa estrutura de dados, a iteração com o usuário é realizada da seguinte maneira. Ao desenhar uma trajetória inicial, alguns pontos dessa trajetória são amostrados e inseridos na estrutura em árvore descrita. Para cada ponto é criado um nodo, e cada um é inserido como filho do nodo criado para armazenar o ponto previamente amostrado. Para o primeiro ponto amostrado, é criado um nodo que será a raiz da árvore.

Quando o usuário termina de desenhar a primeira trajetória, tem-se uma árvore A_1 onde cada nodo terá exatamente um filho, exceto pelo último. Em seguida, o usuário desenha uma segunda trajetória, que inicia aproximadamente onde ele gostaria que os grupos se dividissem. Com essa segunda trajetória, uma árvore A_2, semelhante à primeira, é criada. Mas como já existe uma árvore A_1, é feita uma busca nesta árvore pelo nodo mais próximo do ponto armazenado no nodo raiz da árvore A_2, e então a segunda árvore é anexada como um filho do nodo retornado.

A figura 6.6 pode ser útil para compreender como isso acontece. No momento 1 destacado, o usuário desenhou a trajetória t_1, destacada em vermelho. Os pontos p_1, p_2, p_3, p_4 e p_5 foram amostrados. Para cada ponto, é criado um nodo, que é inserido na árvore. O ponto p_1 será associado ao nodo raiz da árvore. De forma semelhante, a trajetória t_2 foi desenhada no momento 2. Para essa trajetória, os nodos p_6, p_7 e p_8 foram amostrados. Após criar uma árvore com os pontos da segunda trajetória, é realizada a busca na primeira árvore, passando como parâmetro o ponto p_6 (que é a raiz da segunda árvore). Essa busca retorna o nodo associado ao ponto p_3. A segunda árvore então é anexada ao nodo da primeira árvore retornado pela busca.

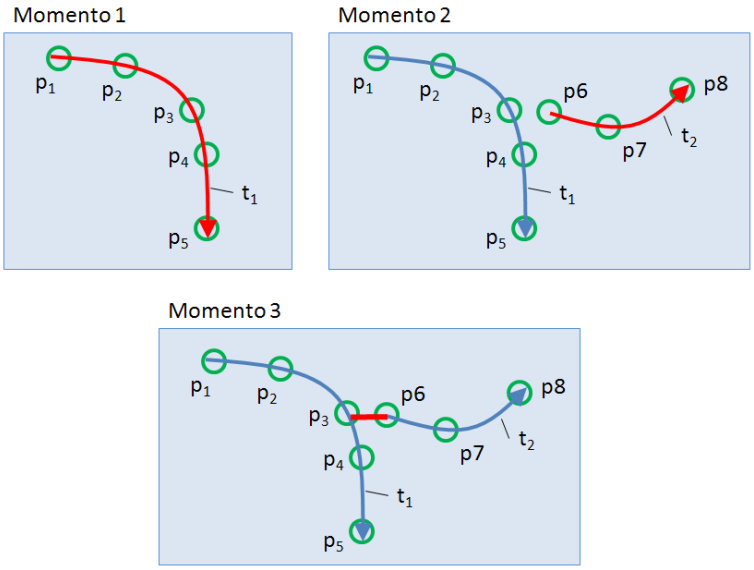


Figura 6.6: Desenho de trajetórias e a estrutura em árvore

Note que o usuário é livre para iniciar a segunda trajetória em um ponto qualquer do cenário. É muito provável que esse ponto não pertença à trajetória traçada anteriormente. Por isso, é necessário incluir uma pequena conexão, ligando as duas trajetórias criadas pelo usuário. Na figura 6.6, isso ocorre no momento 3, de forma automática.

Quando o usuário interage com os agentes virtuais dessa forma, os agentes que perseguirão a trajetória definida formarão um grupo. Após a definição do grupo de agentes, a árvore contendo a trajetória é atribuída àquele agrupamento de agentes.

Para que os agentes sigam a trajetória, é tomado o ponto associado à raiz da árvore como primeiro objetivo intermediário. Esse objetivo intermediário é atribuído a todos os agentes do grupo. Quando um desses agentes alcança esse objetivo, o nodo raiz da árvore é descartado. Se o nodo descartado não possuir mais filhos, os agentes param de caminhar. Se o nodo descartado possuir um único filho, esse filho torna-se a raiz da árvore, e o ponto associado torna-se o novo objetivo intermediário. Se o nodo descartado possuir dois ou mais filhos, os agentes do grupo são divididos em tantos sub-grupos quantos nodos filhos o nodo tiver. Cada nodo filho (com a sua respectiva sub-árvore) é atribuído a um sub-grupo, e então o ciclo de obter o ponto associado à raiz para definir o objetivo intermediário dos seus agentes se reinicia.

O uso da estrutura em árvore permite que um número qualquer de trajetórias possa ser desenhado pelo usuário, sem nenhum tipo de restrição. Com isso, dividir um grupo de agentes virtuais em dois grupos menores é apenas o caso mais simples da infinita quantidade de formas de dividir os grupos. Com a forma proposta, o usuário passa a ter bastante liberdade na forma de descrever como os agentes se dividem em grupos menores.

A função que divide um grupo de agentes em vários sub-grupos deve ser especificada, de acordo com o caso que se deseja simular. Por exemplo, é possível definir que, após a divisão, cada grupo terá um mesmo número de agentes. Ou então, é possível utilizar propriedades dos agentes e das trajetórias para definir em que grupo cada agente

irá prosseguir. O caso citado inicialmente, com mergulhadores e soldados, pode ser facilmente simulado incluindo um atributo no agente, indicando se ele será um mergulhador ou soldado. Também deve ser incluído um atributo na trajetória, indicando se ela deve ser seguida por soldados ou mergulhadores (tal atributo será definido pelo usuário, ao desenhar a trajetória sobre o terreno). Quando os agentes precisarem se dividir em grupos, basta levar em conta os atributos definidos para que os mergulhadores sigam uma trajetória e os soldados sigam outra.

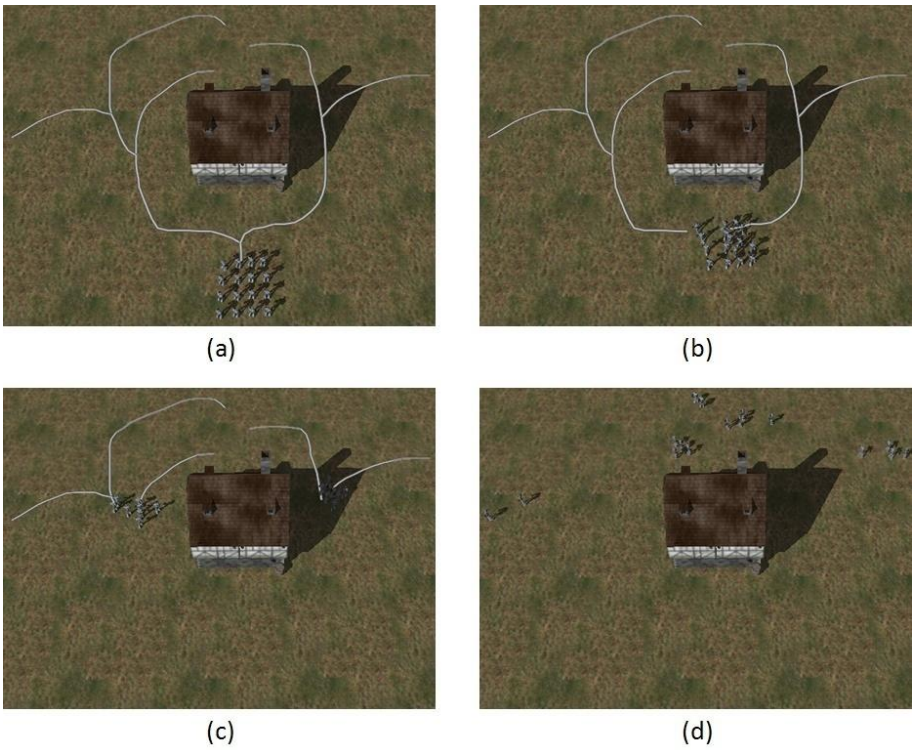


Figura 6.7: Divisão de um grupo de agentes para alcançar cinco pontos distintos do cenário virtual

A figura 6.7 ilustra o caso onde um grupo com vários agentes parte de um ponto do cenário, para alcançar vários outros pontos em volta da construção. Na figura 6.7-a, os agentes estão agrupados, iniciando o movimento. Na figura 6.7-b, os agentes começam a se dividir em dois grupos, um andando pela direita e o outro pela esquerda da casa. Na figura 6.7-c, eles se dividem novamente em mais grupos, enquanto aproximam-se do seu objetivo. Na figura 6.7-d, cada pequeno grupo de agentes atinge um dos objetivos especificados. Nesta simulação, foi feita uma divisão simples dos agentes em grupos menores, onde a cada divisão os grupos formados possuem aproximadamente o mesmo número de agentes.

7 CONCLUSÕES E TRABALHOS FUTUROS

Baseado nos trabalhos de Dapper (DAPPER et al., 2007) e Silveira (SILVEIRA, 2008), este trabalho propôs uma estrutura de dados e uma forma de interagir com placas gráficas atuais para planejar caminhos de humanos virtuais, tendo como resultado uma implementação mais eficiente.

Inicialmente, o código já existente foi reescrito, permitindo que os conceitos apresentados nas referências citadas estivessem mais claros. Com o código reescrito, foi possível desenvolver uma biblioteca de classes, cuja facilidade de extensão e manutenção foram as principais características ganhas com esta tarefa.

Como contribuição adicional, e para facilitar os testes do código que foi reescrito, foi desenvolvida uma extensão ao trabalho de Dietrich (DIETRICH et al, 2008), com o intuito de descrever divisões de grupos ao percorrer caminhos. Com esta extensão, o usuário descreve esboços de como os agentes virtuais se dividirão em grupos, de forma bastante simples e intuitiva. O resultado obtido com esta extensão demonstrou ser bastante simples e eficiente, permitindo que o usuário descreva ações bastante complexas com poucos requisitos de processamento e memória.

Em seguida, o algoritmo de planejamento de caminhos foi analisado com o intuito de detectar os pontos paralelizáveis do algoritmo. Neste estudo percebeu-se que o relaxamento dos mapas locais e globais são as etapas que mais se beneficiariam de um processamento paralelo. Mas também percebeu-se que os outros passos do algoritmo também poderiam ser beneficiados com uma implementação deste tipo.

Desta forma, a linguagem de programação Cuda foi estudada, e diversos passos do algoritmo de planejamento de caminhos foram reescritos com ela. Esta linguagem permitiu que esses passos se aproveitassem do paralelismo disponível nas placas gráficas nVidia, apresentando ganhos sensíveis de desempenho.

Por fim, as duas versões disponíveis do algoritmo de planejamento de caminhos (uma seqüencial, que executa apenas em CPU, e outra paralela, que utiliza as placas gráficas para realizar o processamento) foram comparadas. Diversos testes foram realizados, alternando número de agentes virtuais na cena e tamanho dos mapas locais. Nesses testes, foi calculada a freqüência que o algoritmo pode ser executado nas duas versões. Em todos os testes realizados, a versão paralela do algoritmo demonstrou-se superior, sendo no melhor caso 56 vezes mais rápida que a versão seqüencial.

Como trabalhos futuros, são feitas as seguintes sugestões:

- Propor uma solução eficiente para relaxar campos potenciais com mais de 512 células. Este trabalho focou-se no relaxamento de campos potenciais pequenos, e a solução apresentada não pode ser utilizada diretamente em campos potenciais grandes, principalmente devido a restrições impostas pela linguagem Cuda;
- Estudar a viabilidade de implementar o relaxamento dos campos potenciais utilizando outros métodos desenvolvidos para arquiteturas paralelas. O método de Jacobi exige a sincronização de todas as threads a cada iteração. Outros métodos podem precisar de menos sincronizações, tornando-se mais eficiente. Um exemplo não tradicional, também voltado para essas arquiteturas é o método ADI (PEACEMAN, RACHFORD, 1955);
- Realizar um estudo mais preciso da complexidade do algoritmo proposto por Dapper. Neste trabalho, foi apresentado uma visão superficial da complexidade de execução do algoritmo, mas um estudo mais preciso do número de operações deve ser realizado. Também poderá ser feito um estudo da complexidade de uso de memória, o que não foi estudado neste trabalho.

REFERÊNCIAS

CONNOLLY, C. I.; BURNS, J. B.; WEISS, R. Path Planning Using Laplace's Equation. In: IEEE International Conference On Robotics and Automation, 1990., 1990. Proceedings. . . [S.l.: s.n.], 1990. p.2102–2106.

DELOURA, M. Game Programming Gems 1. [S.l.]: Charles RiverMedia, 2000. p.443–440.

DAPPER, F.; NEDEL, L. P; PRESTES, E. S. Planejamento de Movimento para Pedestres Utilizando Campos Potenciais. 2007. 73 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

DAPPER, F.; PRESTES, E.; NEDEL, L. P. Generating Steering Behaviors for Virtual Humanoids using BVP Control. Proc. of CGI, [S.l.], 2007.

DIETRICH, C. A.; NEDEL, L. P.; COMBA, J. L. D. A Sketch-Based Interface to Real-Time Strategy Games Based on a Cellular Automaton. In: Game Programming Gems 7. [S.l.]: Charles River Media, 2008. p.59–68.

FORTUNA, A. de.; Técnicas Computacionais para Dinâmica dos Fluidos: Conceitos Básicos e Aplicações. [S.l.]: EDUSP, 2000. ISBN 85-3140-5262.

KHATIB, O. Commande dynamique dans l'espace opérational des robots manipulatersen présence d'obstacles. 1980. Tese (Doutorado em Ciência da Computação) — École Nationale Supérieure de l'Aéronatique et de l'Espace, France.

LAMIRAUX, F.; BONNAFOUS, D.; LEFEBVRE, O. Reactive Path Deformation For Nonholonomic Mobile Robots. In: IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATIONS, 2004. [S.l.: s.n.], 2004. p.967–977.

MAZARAKIS, G. P.; AVARITSIOTIS, J. N. A prototype sensor node for footstep detection. In: WIRELESS SENSOR NETWORKS, 2005. PROCEEEDINGS OF THE SECOND EUROPEAN WORKSHOP ON, 2005. Anais. . . [S.l.: s.n.], 2005. p.415–418.

NIEUWENHUISEN, D.; KAMPHUIS, A.; OVERMARS, M. H. High quality navigation in computer games. Sci. Comput. Program., Amsterdam, The Netherlands, The Netherlands, v.67, n.1, p.91–104, 2007.

nVidia CUDA Compute Unified Device Architecture Programming Guide Version 2.0, 2008, [S.l], disponível em <http://developer.download.nvidia.com/compute/cuda/2_0/windows/sdk/NVIDIA_CUDA_SDK_2.02.0811.0240_win32.exe>, acesso em 29 de outubro de 2008.

OGRE Open Graphics Rendering Engine, 2008, [S.l], disponível em <http://www.ogre3d.org/docs/manual/>, acesso em 20 de novembro de 2008.

OSG Open Scene Graph, 2008, [S.l], disponível em <http://www.openscenegraph.org/projects/osg/wiki/Support>, acesso em 20 de novembro de 2008.

PEACEMAN, D. W.; RACHFORD, H. H. The Numerical Solution of Parabolic and Elliptic Differential Equations. In: Journal of the Society for Industrial and Applied Mathematics, Vol. 3, No. 1, Mar. 1955, pp. 28-41.

PRESTES, E.; ENGEL, P. M.; TREVISAN, M.; IDIART, M. A. Exploration Method using Harmonic Functions. Robotics and Autonomous Systems, [S.l.], v.40, n.1, p.25–42, 2002.

REIF, J. H.; WANG, H. Social Potential Fields: a Distributed Behavioral Control for Autonomous Robots. In: WAFR: Proceedings of The Workshop on Algorithmic Foundations of Robotics, 1995, Natick, MA, USA. A. K. Peters, 1995. p.331–345.

RUSSELL, S. J.; NORVIG, P. Artificial Intelligence: a modern approach. [S.l.]: Pearson Education, 2003.

SILVEIRA, R; NEDEL, L. P; PRESTES, E. S. Planejamento de Movimento para Grupos Utilizando Campos Potenciais, 2008, 78 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

TREVISAN, M.; IDIART, M. A.; PRESTES, E.; ENGEL, P. M. Exploratory Navigation based on Dynamic Boundary Value Problems; Journal of Intelligent and Robotic Systems, [S.l.], v.45, p.101–114, 2006.

ANEXO: COMPARAÇÃO ENTRE CÓDIGO SEQUENCIAL E PARALELO

Como um dos objetivos deste trabalho foi desenvolver uma implementação paralela do algoritmo para planejamento de caminhos descrito, para então compará-la com a versão seqüencial, também faz sentido haver uma comparação entre os códigos escritos.

Assim, para que seja possível fazer essa comparação, foi selecionado o trecho referente ao relaxamento dos mapas locais, na versão seqüencial e paralela. Esses dois trechos de código estão apresentados a seguir, para que o leitor possa compará-los.

Em seguida, será realizada uma breve explicação do código, facilitando o seu entendimento pelo leitor. Note que, apesar de esse código estar de fato implementado, ele precisa do restante do código para que funcione corretamente. O seu próprio entendimento só pode ser completo ao se estudar o restante do código. Dessa forma, o código apresentado a seguir serve apenas para ilustrar ao leitor as diferenças entre um código seqüencial escrito em C++ e um código paralelo escrito com Cuda. Também deve se ressaltar que não será feita a comparação de todo o código, mas apenas de alguns trechos. A comparação total do código seria cansativa e pouco interessante para este trabalho.

Versão seqüencial do método de relaxamento

```
1.  int PP::PotentialField::relax() {
2.     unsigned int k = 0;
3.     float mi;
4.     float error= 0.0f;
5.     do {
6.        for (register int i = 1; i < dimensionX-1; ++i) {
7.           for (register int j = 1; j < dimensionY-1; ++j) {
8.              if (getTypeAt(i,j) == Cell::T_FREE) {
9.                 mi = average(i,j);
10.                error += fabs( mi - getPotentialAt(i,j) );
11.                setPotentialAt(i,j,mi);
12.             }
13.          }
14.       }
15.       if (error < this->maxError ) {
16.          return k;
17.       }
18.       error = 0.0f;
19.       ++k;
20.    } while ( k < this->maxIterations );
21.    return k;
22. }
```

Versão paralela das funções de relaxamento, implementadas com Cuda

```
1.  void callRelaxKernel(int potentialFieldCount, int dimBlockX,
2.           int dimBlockY, const unsigned int iterations,
3.           float* globalPotentialMatrix,
4.           const int* globalTypeMatrix) {
5.     dim3 dimBlock(dimBlockX,dimBlockY);
6.     int sharedMemSize = dimBlockX*dimBlockY*sizeof(float);
7.     relaxKernel<<<potentialFieldCount,dimBlock,sharedMemSize>>> (
8.        iterations, globalPotentialMatrix, globalTypeMatrix
9.     );
10.    cudaSynchronize();
11. }
12.
13. __global__ void relaxKernel(unsigned int iterations,
14.           float *globalPotentialMatrix,
15.           const int *globalTypeMatrix) {
16.    extern __shared__ float sharedPotentialMatrix[];
17.    const int start = startIndex[blockIdx.x];
18.    const int dimensionX = dimX[blockIdx.x];
19.    const int dimensionY = dimY[blockIdx.x];
20.    const int idx = threadIdx.y*dimensionX+threadIdx.x;
21.    sharedPotentialMatrix[ idx ] =
22.       globalPotentialMatrix[ start + idx ];
23.    bool relaxar = false;
24.    if( threadIdx.x<dimensionX && threadIdx.y<dimensionY ) {
25.       relaxar = (globalTypeMatrix[ start + idx ]=='f');
26.    }
27.    const float* left = sharedPotentialMatrix +
28.       threadIdx.y*dimensionX + (threadIdx.x-1);
29.    const float* right = sharedPotentialMatrix +
30.       threadIdx.y*dimensionX + (threadIdx.x+1);
31.    const float* up = sharedPotentialMatrix +
32.       (threadIdx.y-1)*dimensionX + threadIdx.x;
33.    const float* down = sharedPotentialMatrix +
34.       (threadIdx.y+1)*dimensionX + threadIdx.x;
35.    __syncthreads();
36.    while(iterations>0) {
37.       if( relaxar ) {
38.          sharedPotentialMatrix[idx] = (*left + *right +
39.                                        *up + *down) * 0.25f;
40.       }
41.       iterations--;
42.       __syncthreads();
43.    }
44.    if(relaxar) {
45.       globalPotentialMatrix[ start + idx ] =
46.          sharedPotentialMatrix[ idx ];
47.    }
48. }
```

O primeiro trecho citado refere-se ao método que faz o relaxamento de um campo potencial de forma seqüencial. Tal método faz uso de outros métodos da mesma classe. Apesar de esses o código desses outros métodos e atributos não estar explicito, a funcionalidade por trás deles pode ser subentendida pelos seus nomes.

O método apresentado não executa um número fixo de iterações. Ele possui um limite no número de iterações (linha 20), podendo terminar de executar caso o campo potencial tenha convergido antes (linhas 15 e 16). Dentro deste laço, dois outros laços são executados (linhas 6 a 14), atualizando o valor das células livres do mapa local usando o método Gauss-Seidel. O método chamado na linha 9 é responsável pela solução da equação 3.6. As linhas restantes são responsáveis por controlar o erro do campo potencial na iteração atual.

Já o segundo trecho de código apresentado é mais complexo, tanto por ter detalhes referentes à linguagem Cuda quanto por ter otimizações de baixo nível, como o uso de aritmética de ponteiros. Esse trecho de código é dividido em duas funções. A primeira delas (linha 1) tem duas responsabilidades. A primeira é servir como uma interface para o restante do código. Dessa forma, os trechos que não precisam manipular código Cuda ficam separados daqueles que tratam diretamente com o código Cuda. A segunda funcionalidade é declarar o número de blocos e threads por bloco que serão executadas. Isso é feito na linha 7, com o uso de um operador especial do Cuda (os sinais "<<<" e ">>>"). Esse operador também é responsável por declarar a quantidade de memória compartilhada a ser utilizada por bloco.

Quando a linha 7 é executada, o Cuda inicia a execução da função declarada nas linhas 13 a 48, em diversas threads. Essa função será responsável por atualizar o valor de uma única célula no mapa local. Para isso, ela utiliza a informação do identificador da thread (`threadIdx.x` e `threadIdx.y`) para definir qual a célula será atualizada.

Como foi citado, uma das otimizações de código realizadas foi o uso da memória compartilhada do multiprocessador para realizar os cálculos. Para isso, a linha 16 declara a variável de acesso à memória compartilhada, as linhas 21 e 22 buscam o valor na memória global, e após os cálculos as linhas 44 a 47 enviam o valor de volta para a memória global, permitindo que o host tenha acesso ao campo potencial relaxado.

Por fim, nas linhas 36 a 43 é realizado o relaxamento do campo potencial em si. Como o controle do erro global do campo potencial iria gerar muitas sincronizações, optou-se por apenas executar o número fixo de iterações. O laço executado nessas linhas é equivalente ao laço *while* executado no código seqüencial, nas linhas 5 a 20. Note que não é necessário realizar nenhuma iteração sobre as células do mapa local, pois isso é feito implicitamente pelo escalonador da placa gráfica.