UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RAFAEL PICCIN TORCHELSEN

Using the parametric domain for efficient computation

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. João Luiz Dihl Comba
Advisor

Prof. Dr. Rui Manuel Ribeiro de Bastos
Coadvisor

Porto Alegre, June 2010

CIP – CATALOGING-IN-PUBLICATION

Rafael Piccin Torchelsen,

Using the parametric domain for efficient computation /

Rafael Piccin Torchelsen. – Porto Alegre: PPGC da UFRGS, 2010.

107 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Advisor: João Luiz Dihl Comba ; Coadvisor: Rui Manuel Ribeiro de Bastos .

1. Mesh Parameterization. 2. Displacement Mapping. 3. Geodesics. 4. Agent Path Planning. I. , João Luiz Dihl Comba. II. , Rui Manuel Ribeiro de Bastos. III. Título.

Para meus pais, os quais admiro muito.

Não tá morto quem peleia!

AGRADECIMENTOS

Primeiramente agradeço aos meus pais, vocês são exemplos de honestidade e trabalho. Foram 4 anos nos quais não dediquei tempo a vocês. Esse talvez tenha sido o preço mais alto cobrado pelo doutorado. Vocês nunca deixaram de me apoiar e trabalharam muito para que eu tivesse as oportunidades que tive. Pai, prometo que vou pescar mais contigo. E mãe, prometo que vou falar para o pai não ir pescar tanto.

Aninha, minha flor, minha companheira, sem o teu apoio eu não teria agüentado o tirão. Quantas vezes eu chegava em casa é tinha um carreteiro de charque me esperando. E há quanto tempo que tu me coloca nos eixos? E sempre me deu uma mão. Só hoje de manhã, se não fosse a tua mão me colocar pra fora da cama eu ainda estaria lá. Tu é uma companheira para a vida toda. Te amo muito!

Meu irmão, índio veio ginete de éguas e ondas. Também estou na dívida contigo, também faz 4 anos que não tenho tempo para uma camperiada. Tenho um butiá no velho barreiro só esperando.

Preciso agradecer muito aos meus orientadores. Inicialmente por aceitarem me orientar e continuarem orientando até o final. Por me ensinarem como fazer pesquisa de alto nível e ser crítico sobre o próprio trabalho. Agradeço pelos elogios, mas principalmente pelos puxões de orelha, e foram muitos e as vezes um em cada orelha. Porém, foi isso que me vez chegar até aqui. Sei que não fui o único a deixar de ficar com a família para que eu pudesse chegar até aqui. E por isso eu agradeço muito. Foram 4 longos anos e uma infinidade de reports. Obrigado por acreditarem em mim, mesmo quando a batata estava queimando.

Também preciso agradecer aos meus orientadores do mestrado e graduação, Soraia e Rocha Costa. Sem a orientação de vocês nunca teria conseguido chegar ao doutorado.

E aos meus colegas, esses nem sei como agradecer, foram todos muito importantes durante esse período. Espero poder fazer outras carnes mala assadas.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
RTS	Real-Time Strategy Game

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The process of parameterizing a mesh to the plane is an ongoing research topic. Although there are several works dedicated to parameterization techniques the use of the resulting parameterizations has received less attention.

This work presents contributions related to the use of the parametric space to improve the computational efficiency of several algorithms. The main motivation comes from the fact that some algorithms are more efficiently computed on the parametric version of the mesh, compared to the 3-D version. For example, shortest distances can be computed, usually, an order of magnitude faster on the parametric space.

The contributions of this work can be applied to at least three research fields related to computer graphics: displacement mapping, distance computation on the surface of triangular meshes and agent path planning.

The contribution related to displacement mapping, presented in chapter 4, is used to increase the rendering performance and visual quality of terrains in games. The new method to compute distances, presented in chapter 5, increases the efficiency of several distance computation algorithms. This new method was also used on a novel agent path planning algorithm, to navigate agents on the surface of arbitrary meshes. This technique is presented in chapter 6.

The potential of the new distance computation method is not restricted to the applications presented in this thesis. In general, any technique that uses distance computation on the surface of triangular meshes can have the performance improved by the method. We can cite the following applications: procedural texture generation, surface labeling, re-meshing, mesh segmentation, etc.

Keywords: Mesh Parameterization, Displacement Mapping, Geodesics, Agent Path Planning.

Utilizando o espaço paramétrico para computação eficiente

RESUMO

O processo de parametrização de malhas em planos é um tópico de pesquisa bastante explorado. Apesar do grande esforço despendido no desenvolvimento de técnicas mais eficientes e robustas, pouco se tem investido no uso das representações paramétricas geradas por estas técnicas.

Este trabalho apresenta contribuições relacionadas ao uso do espaço paramétrico para computações eficientes. A principal motivação vem do fato de alguns algoritmos serem mais eficientes quando aplicados sobre a versão paramétrica da malha. Algoritmos para o cálculo de distância mínima, por exemplo, podem ter um aumento significativo de eficiência quando aplicados em versões paramétricas de malhas. Nossos resultados demonstram que esses aumentos de eficiência podem chegar a cerca de uma ordem de magnitude em alguns casos.

As contribuições deste trabalho possuem aplicação direta em três campos de pesquisa relacionados à computação gráfica: *displacement mapping*, cálculo de distâncias sobre superfícies e movimentação de agentes.

A contribuição relacionada a *displacement mapping*, apresentada no capítulo 4, é utilizada para aumentar a performance de renderização e a qualidade visual de terrenos em jogos. O novo método de cálculo de distâncias proposto, apresentado no capítulo 5, aumenta a eficiência de vários algoritmos de cálculo de distância sobre superfícies de malhas. Este novo método também é utilizado em uma nova técnica para cálculo de movimentação de agentes em superfícies de malhas arbitrárias. Esta técnica é apresentada no capítulo 6.

O potencial da nova técnica de cálculo de distância sobre malhas não está restrito aos exemplos apresentados. Em geral, qualquer técnica que utilize o cálculo de distância sobre superfícies de malhas de triângulos se beneficia das contribuições deste trabalho, podendo-se citar como exemplos a geração de texturas procedurais, rotulamento de superfícies, re-triangulação de malhas e segmentação de malhas, entre outros.

Palavras-chave: Parameterização de Malhas, *displacement mapping*, Geodesicas, Movimentação de Agentes.

1 INTRODUCTION

Mapping a 3-D triangular mesh over a 2-D triangular mesh has many applications. Texture mapping is a well know example (see Figure 1.1). This process establishes a relationship between points on the 3-D mesh surface and points on the 2-D parametric space.

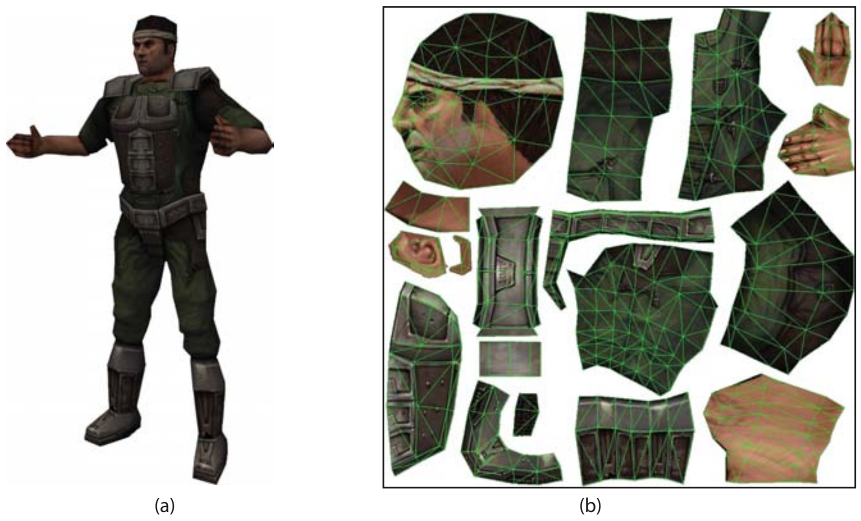


Figure 1.1: Texture mapping. Item (a) illustrates a 3-D triangular mesh. We can see on (b) the mapping of the mesh over a texture. Based on this mapping it´s possible to define a color for each point on the 3-D mesh surface as the color mapped in parametric space.

The processing of mapping a 3-D mesh over a 2-D mesh is also known as parameterization, see (SHEFFER et al., 2007), (HORMANN; LéVY; SHEFFER, 2007) and (SHEFFER; PRAUN; ROSE, 2006) for a survey. Most of the parameterization techniques are designed to generate a 2-D mesh based on a 3-D mesh, preserving geometric characteristics of the 3-D mesh on the 2-D mesh. For example, preserve the same triangle area in 3-D and 2-D.

This work explores the use of parameterized meshes in two situations to improve the performance of some existing algorithms. The first situation is related to the technique known as displacement mapping. We use the size of the mesh triangles in the parametric space as a level-of-detail metric to improve the technique efficiency. The second situation is related to computing the shortest distance between points on the surface of triangular meshes. We use the parametric space to improve the efficiency of the distance computation.

1.1 Displacement mapping

The technique known as displacement mapping encodes triangular meshes as textures. One of the main uses of this technique is in rendering terrains in virtual environments. Displacement mapping is used to that end, partially, due to the resemblance of the displacement map and a topographic map. Both describe a surface as displacements on the vertical axis. Figure 1.2 illustrates the displacement mapping technique. The work of Szirmay-Kalos (SZIRMAY-KALOS; UMENHOFFER, 2008) presents a survey about displacement mapping.

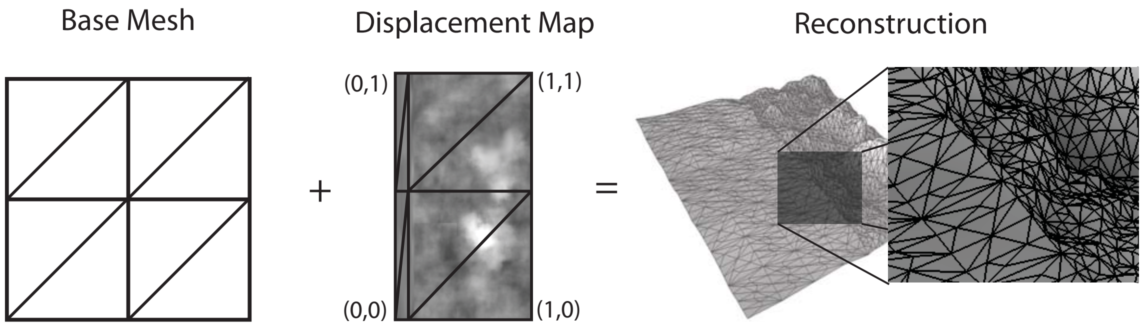


Figure 1.2: Displacement mapping overview. This method can decode a triangular mesh with small computational cost. Basically, the surface details are encoded into a texture called displacement map. This texture is mapped over a 3-D triangular mesh (base mesh), which has a small amount of details. The displacing of the base mesh surface, according to the displacement map, results on a detailed mesh. This process can be applied with efficiency on current GPUs.

The resulting geometry from displacement mapping has usually a uniform distribution of vertices, as we can see on Figure 1.3(a). However, this can result in under sampling of highly detailed regions and oversampling on low detailed regions of the displacement map. This thesis presents a technique, based on parameterizations, that allows an adaptive distribution of vertices according to the density of details on the displacement map. The adaptive approach is based on implicit information of the parameterization used on displacement map. On Figure 1.2 we can see a relationship between the size of the triangles in the parametric domain with the amount of details in the displacement map. We can use the size of the triangles, in the parametric domain, to guide the amount of samples over the displacement map, resulting in an adaptive reconstruction, as illustrated on Figure 1.3(b).

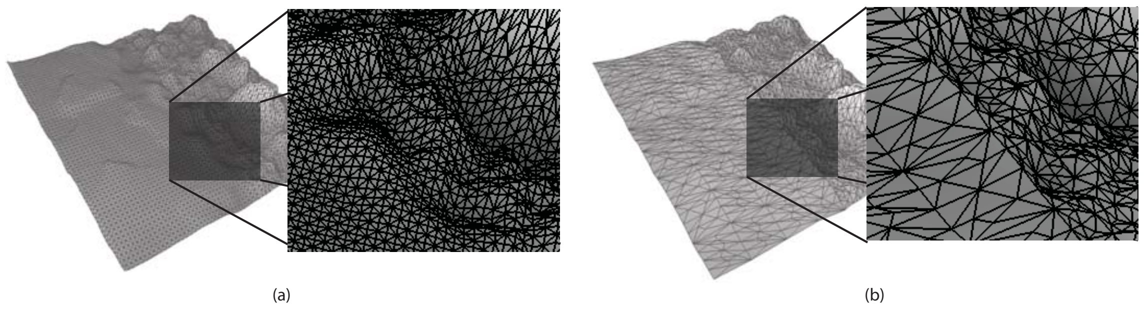


Figure 1.3: Displacement mapping: Uniform (a) vs Adaptive (b). Using an adaptive approach results in a better use of the computational resource, compared to a uniform reconstruction.

There are many variations to the displacement mapping technique; one of the most efficient is known as geometry clipmaps (LOSASSO; HOPPE, 2004). This technique is designed to efficiently represent terrains in virtual environments. However, there are limitations to the visual realism of the rendered terrains. One of such limitations is related to texture mapping, which is not described in the original work. Figure 1.4 illustrates the use of geometry clipmaps.

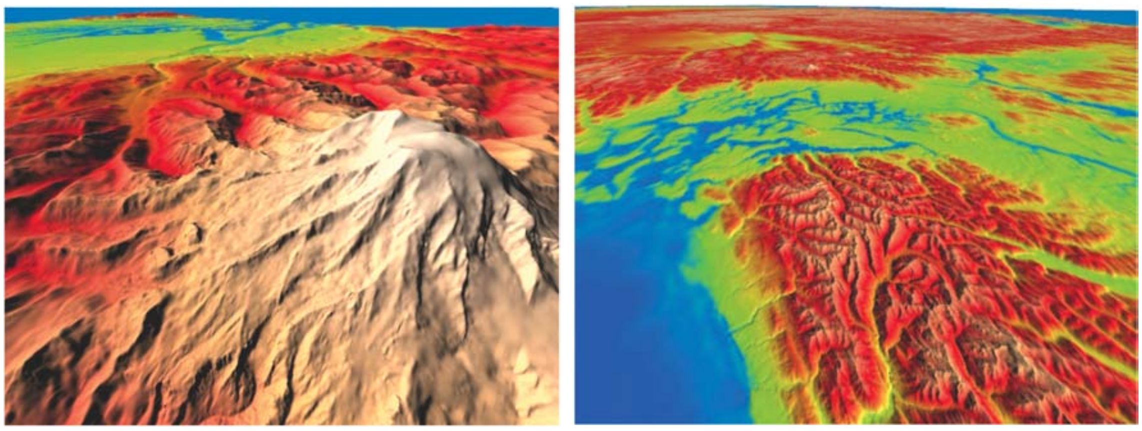


Figure 1.4: Results from the geometry clipmap technique (LOSASSO; HOPPE, 2004). This technique lacks support for texture mapping, which limits the visual fidelity of the terrain. The only described method to color the terrain is a simple gradient mapping according to the height of each point on the terrain. This figures use a gradient formed by blue, green, red and brown. The green is mapped to the lowest point on the terrain, while the brown is mapped to the highest.

Another contribution of this thesis is the improvement of geometry clipmaps. We present a method to map textures over the terrain, allowing an improvement in the visual fidelity. Figure 1.5 illustrates the results obtained.

The contributions related to displacement mapping are detailed in chapter 4 and in the following papers (TORCHELSEN; BASTOS; COMBA, 2008) and (TORCHELSEN et al., 2009).

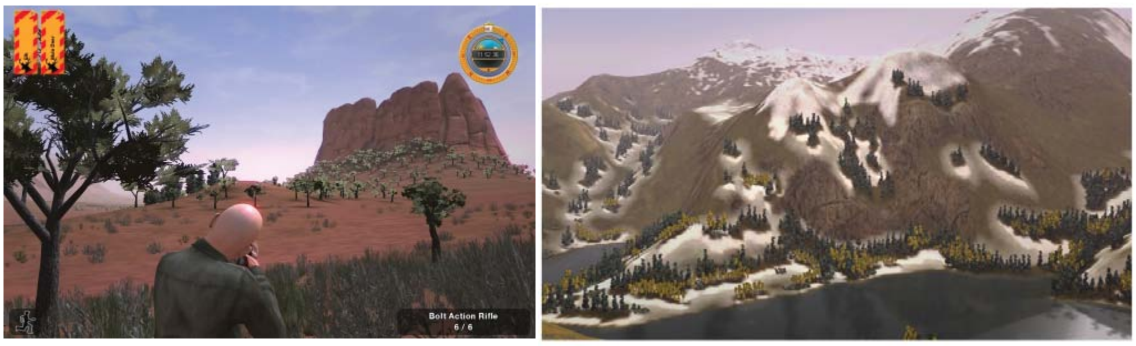


Figure 1.5: Screenshots from the game Deer Hunter Tournament®. The terrains on this figure are generated using the technique presented in this thesis.

1.2 On-surface distance computation on the parametric space

The minimal distance computation between two points on the surface of a triangular mesh has many applications. The navigation of agents on the surface of triangular meshes, as illustrated on Figure 1.6, is an example.

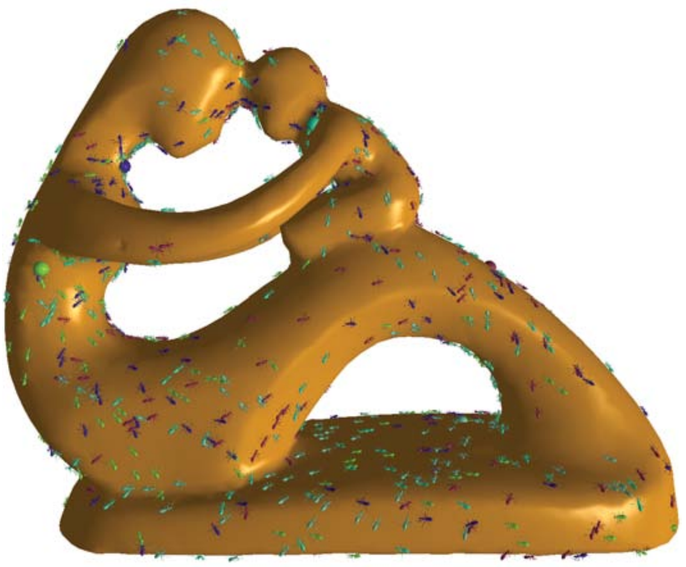


Figure 1.6: Results from the path planning algorithm presented in this thesis. There are 512 agents on this surface.

This work presents an efficient method to compute the shortest distance between two points over the surface of triangular meshes. As illustrated, the number of triangles has an impact on the computational cost of the distance computation. A reduction on the number of triangles can improve the algorithm performance. However, a simplification in 3-D can result in loss of precision during distance computation. On the other hand, reducing the number of triangles in parametric space has a high accuracy, as illustrated on Figure 1.7.

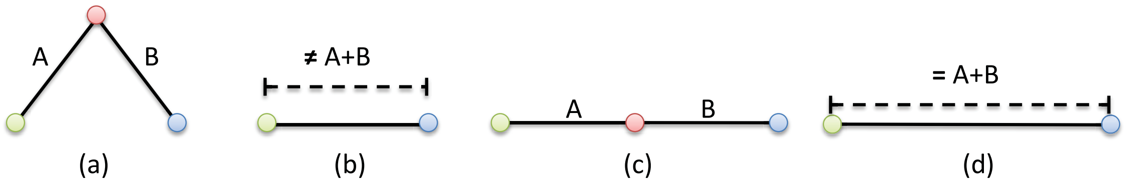


Figure 1.7: Difference between simplification in object space and parametric space. (a) is a simple mesh. The distance between the two bottom vertices is equal to the length of the edges A and B. If a simplification is applied in object space as removing the vertex on the center, the resulting mesh is (b). The distance between the two vertices on (b) is no longer equal to A+B. However, if a parameterization is used, here to 1-D (c), and the simplification applied in parametric space we have the mesh illustrated on (d). The distance between the two vertices on (d) is equal to A+B.

We present a novel method to compute approximated distances on a simplified mesh in the parametric space, as illustrated on Figure 1.8. This novel approach allows a control between precision and performance. This method is described in chapter 5 and in (TORCHELSEN et al., 2009). Although our method presents higher performance, on most cases, it is best suitable for large meshes due to the mesh segmentation and parameterization. On non-smooth meshes with few triangles our mesh simplification method, presented on chapter 5, can not reduce the number of triangles, which turns the best solution for this cases the use of distance computation algorithms on the original mesh without the mesh simplification presented in this work.

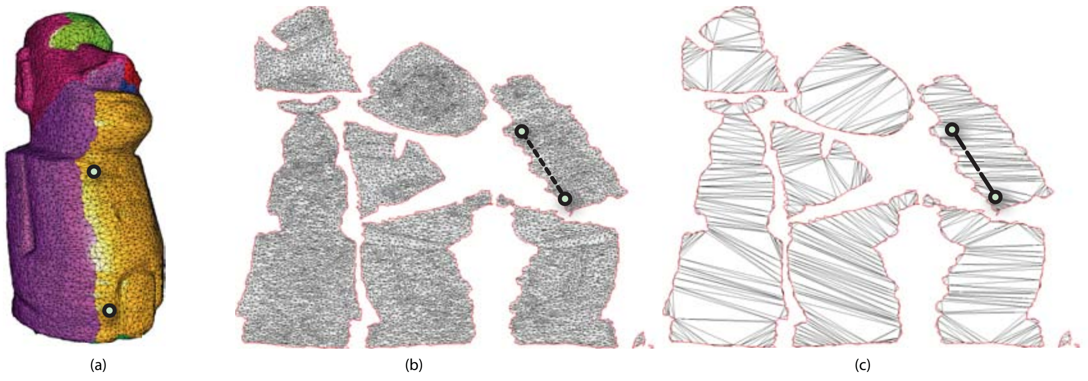


Figure 1.8: Distance computation in the parametric space. (a) is the mesh where distance computation between two points is needed. (b) is the mesh parameterization and (c) is the simplified version. The method presented in this thesis concerns the distance computation on a simplified and parametric version of the mesh (c).

Until now, agent path planning on the surface of complex meshes was limited, due to the high computational cost. A great portion of this cost is due to the distance computation. Usually, the surface where agents are allowed to move is limited to almost planar surfaces, which can be represented by few triangles. The ability to control precision and performance on the distance computation allows the use of more detailed surfaces in interactive applications, as games. Figure 1.9 illustrates this limitation on the complexity of the mesh in games, while Figure 1.10 illustrates the limitation on scientific publications.

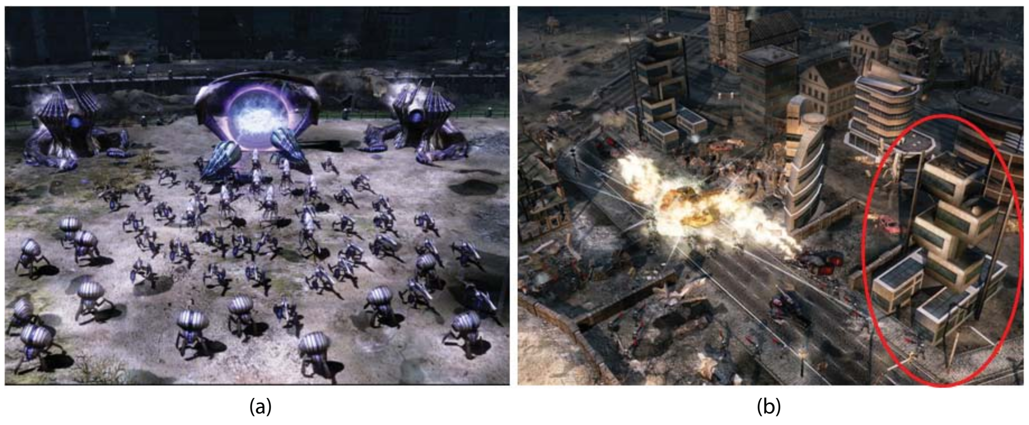


Figure 1.9: Screenshots from the game Command and Conquer^{TM}. This kind of game requires high performance agent path planning algorithms to sustain interactive frame rate. In that sense, the surfaces where agents are allowed to navigate are limited, usually to planar surfaces (a), to reduce the computational cost. This results in a limited use of the virtual environment. The agents usually can not cross over complex regions, as the building on (b).

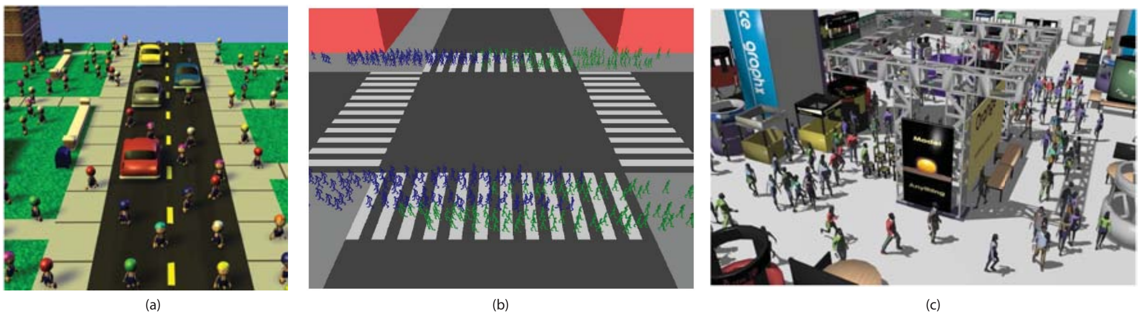

(a)

Figure 1.10: Results from agent path planning algorithms. We can see that the use of planar surfaces is frequent. The results are respectively from: (SUD et al., 2007), (SUD et al., 2007) and (BERG et al., 2008a).

To the best of our knowledge, the work (TORCHELSEN et al., 2010) was the first to present an agent path planning algorithm for arbitrary surfaces. This method is also parallelizable and is implemented using the GPU. This method is capable to navigate 512 agents on the surface of complex meshes with interactive frame rates. Figure 1.6 illustrates the results. This technique is detailed in chapter 6 and in (TORCHELSEN et al., 2010).

1.3 Contributions

This work presents the following contributions:

- **Efficient displacement mapping using the parametric space**:The improvements presented to the visual quality of the geometry clipmaps allow the use of this technique in applications that require high visual fidelity in the rendering of terrains. Additionally, the level-of-detail technique, for the reconstruction of displacement map, results in a more efficient use of computational resources compared to traditional displacement mapping.

- **Distance computation in the parametric space**: The computation of the shortest distance between two points on the surface of triangular meshes using the parametric space is novel. Improving the performance of an operation common to several applications is one of the main contributions of this thesis. The gain of performance on average is of one order of magnitude, compared to the exact solution. The ability to control performance and precision on the distance computation allows the use of this algorithm to interactive applications.

- **Agent path planning on arbitrary surfaces**: The novel agent path planning algorithm is capable of navigating a considerable number of agents on the surface of complex meshes, which until recently was not possible in interactive applications.

2 INTRODUÇÃO

O mapeamento de malhas de triângulos em 3-D para malhas de triângulos em 2-D possui várias aplicações. O mapeamento de texturas é um dos exemplos mais conhecidos (Figura 2.1). Esse processo define uma relação entre um ponto na superfície da malha em 3-D com um ponto no espaço paramétrico em 2-D.

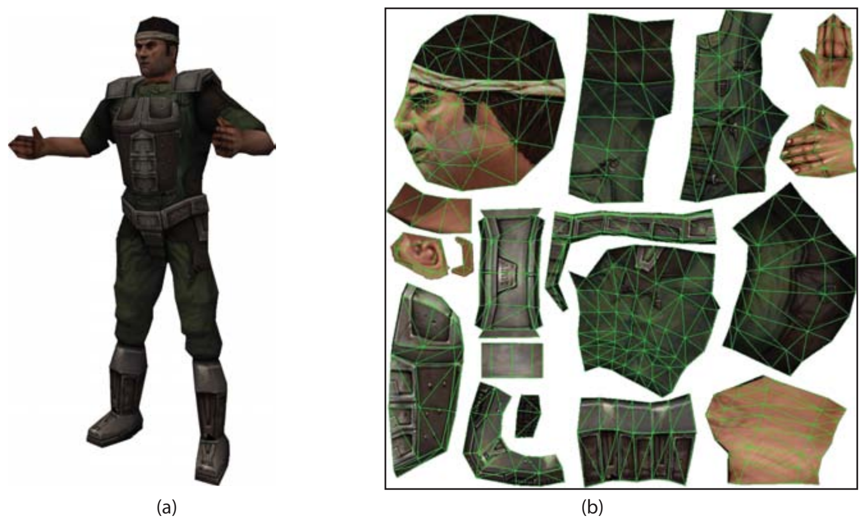


Figura 2.1: Mapeamento de textura. No item (a) podemos ver uma malha de triângulos em 3-D. Já no item (b) podemos ver o mapeamento dessa malha sobre uma textura. Com base nesse mapeamento podemos definir uma cor para cada ponto sobre a superfície da malha 3-D como sendo a cor correspondente no espaço paramétrico.

O processo de mapear uma malha em 3-D para uma malha em 2-D é conhecido como parametrização. A maioria das técnicas de parametrização são desenvolvidas para gerar uma malha em 2-D que compartilhe alguma informação geométrica com a sua versão em 3-D. Um exemplo são as técnicas de parametrização que procuram manter uma relação de igualdade entre as áreas dos triângulos no espaço 3-D e os seus mapeamentos correspondentes no espaço 2-D.

Este trabalho explora o uso da parametrização de malhas de triângulos em dois cenários diferentes, como forma de aumentar a performance de alguns algoritmos existentes. No primeiro cenário, os tamanhos dos triângulos no espaço paramétrico são utilizados como indicadores de nível de detalhe da malha. Isso permite uma reconstrução adaptativa da técnica de *displacement mapping*. No segundo, o processo de parametrização permite o cálculo eficiente de distâncias sobre malhas de triângulos arbitrárias, viabilizando este tipo de cálculo em aplicações de tempo real.

2.1 Displacement mapping

A técnica conhecida por *displacement mapping* codifica malhas de triângulos em texturas. Um dos usos mais comuns dessa técnica é a representação de terrenos em ambientes virtuais. O *displacement mapping* é utilizado para esse fim em parte pela analogia entre o *displacement map* e um mapa topográfico. Ambos descrevem uma superfície com base em um deslocamento vertical. A Figura 2.2 ilustra a técnica de *displacement mapping*. O trabalho de Szirmay-Kalos (SZIRMAY-KALOS; UMENHOFFER, 2008) apresenta uma revisão sobre técnicas de displacement mapping.

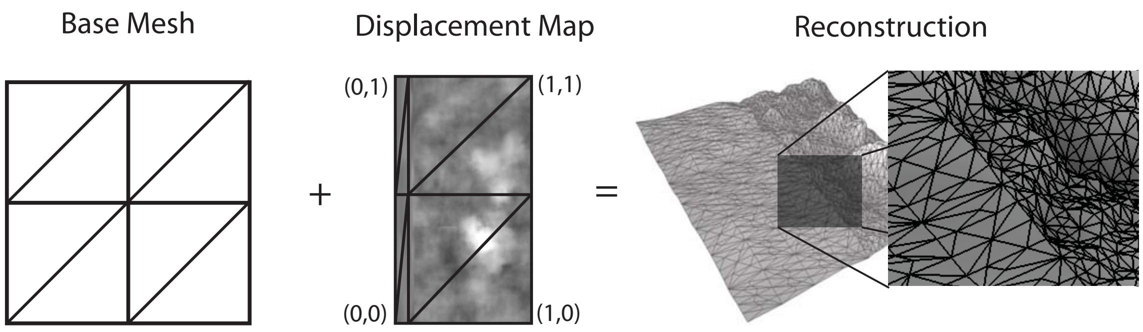


Figura 2.2: Visão geral da técnica de *displacement mapping*. Essa técnica é um método eficiente, em termos computacionais, de codificação e decodificação de malhas de triângulos. Basicamente os detalhes da superfície são codificados em uma textura chamada *displacement map*. Essa textura é então mapeada sobre uma malha que não possui detalhes, chamada malha base. A soma da malha base com os detalhes codificados no *displacement map* resultam e uma malha detalhada e que pode ser renderizada eficientemente na atual geração de GPUs.

A geometria construída com técnicas de *displacement mapping* geralmente apresenta uma distribuição uniforme de vértices, como pode ser visto na Figura 2.3(a). Entretanto, uma distribuição uniforme de vértices pode fazer com que regiões com poucos detalhes sejam super amostradas enquanto regiões com muitos detalhes sejam sub-amostradas. Esta tese apresenta uma técnica, baseada em parametrização, que permite a distribuição adaptativa de vértices que automaticamente varia a densidade de vértices de acordo com o nível de detalhe do *displacement map*. O cálculo da representação adaptativa é baseado numa informação implícita da parametrização utilizada na técnica de *displacement mapping*. Na Figura 2.2 podemos ver que existe uma relação entre o tamanho dos triângulos no espaço paramétrico e a quantidade de detalhes contida no *displacement map*. Se o tamanho dos triângulos no espaço paramétrico for utilizado para determinar a quantidade de detalhes por região da malha, temos uma reconstrução adaptativa do *displacement map*. O resultado desta nova técnica, baseada na amostragem adaptativa, é ilustrada na Figura 2.3(b).

Existem muitas variações da técnica de *displacement mapping*, e uma das mais eficientes é conhecida por geometry clipmaps (LOSASSO; HOPPE, 2004). Essa técnica foi desenvolvida para representar a superfície de terrenos no ambiente virtual utilizando uma variação do *displacement mapping*. Porém, limitações quanto à capacidade de representação de detalhes da técnica inviabilizam seu uso em reconstruções que exijam fotorealismo. Uma de suas limitações está relacionada à impossibilidade de se mapear texturas sobre a malha reconstruída. A Figura 2.4 ilustra o resultado dessa técnica.

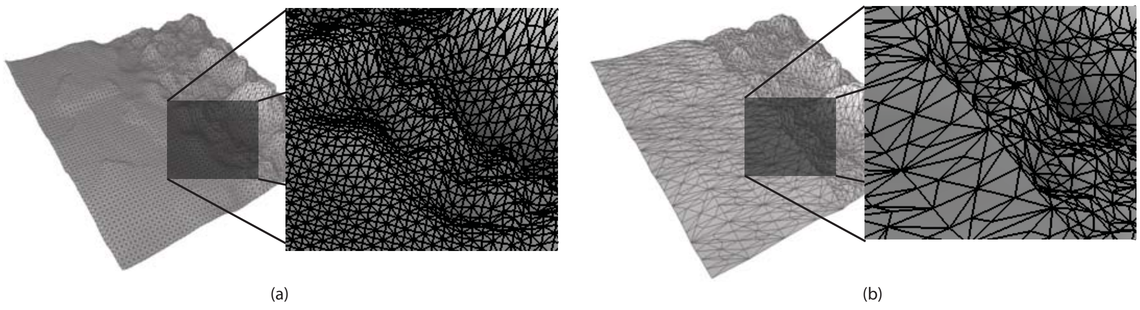


Figura 2.3: *Displacement mapping* uniforme vs adaptativo. Utilizando uma abordagem adaptativa temos uma economia no número de primitivas geométricas, dessa forma aumentado o desempenho da técnica.

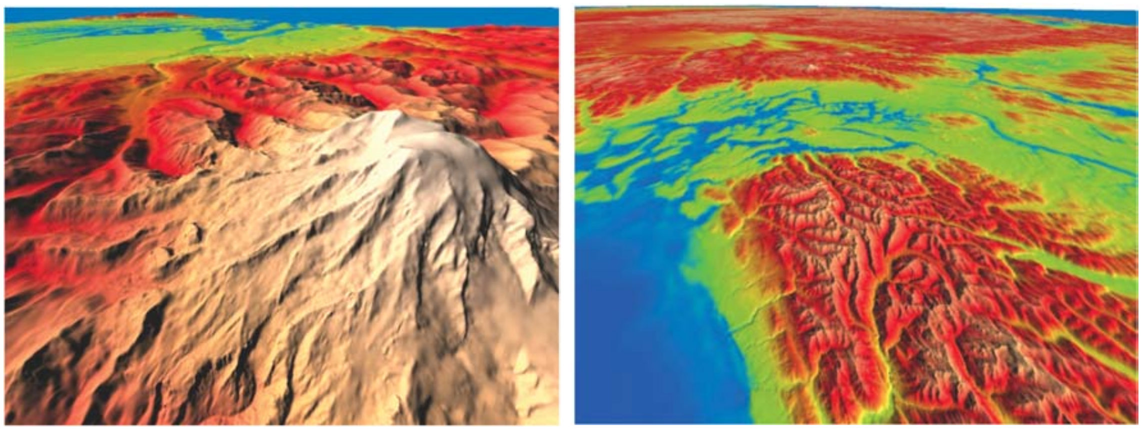


Figura 2.4: Resultados da técnica descrita em (LOSASSO; HOPPE, 2004). As representações virtuais geradas por essa técnica não são capazes de representar elementos naturais sobre a superfície do terreno, como por exemplo, grama. O único mapeamento de cores existente é uma variação de cor com base em um gradiente em relação à altura da superfície. Nesses exemplos as cores são definidas como azul para as áreas mais baixas, verde e vermelho para intermediarias e marrom para as mais altas.

Outra contribuição desta tese refere-se ao aprimoramento dessa técnica para a utilização de texturas. A Figura 2.5 ilustra os resultados desse aprimoramento.

As contribuições relacionadas a técnica de *displacement mapping* são detalhadas no capítulo 4 e nos seguintes artigos (TORCHELSEN; BASTOS; COMBA, 2008) e (TORCHELSEN et al., 2009).

2.2 Cálculo de distâncias no espaço paramétrico

O cálculo da distância mínima entre dois pontos sobre a superfície descrita por malhas de triângulos possui várias aplicações. A movimentação de agentes sobre uma superfície, como ilustrado na Figura 2.6, é um exemplo.

Este trabalho apresenta um método eficiente de cálculo de distância mínima entre dois pontos sobre uma superfície descrita por malha de triângulos. Como apresentado anteriormente, o número de triângulos tem impacto direto no custo computacional do cálculo de distância. A redução no numero de triângulos pode, então, aumentar sua performance. Uma simples redução no numero de triângulos da malha pode afetar gravemente a quali-

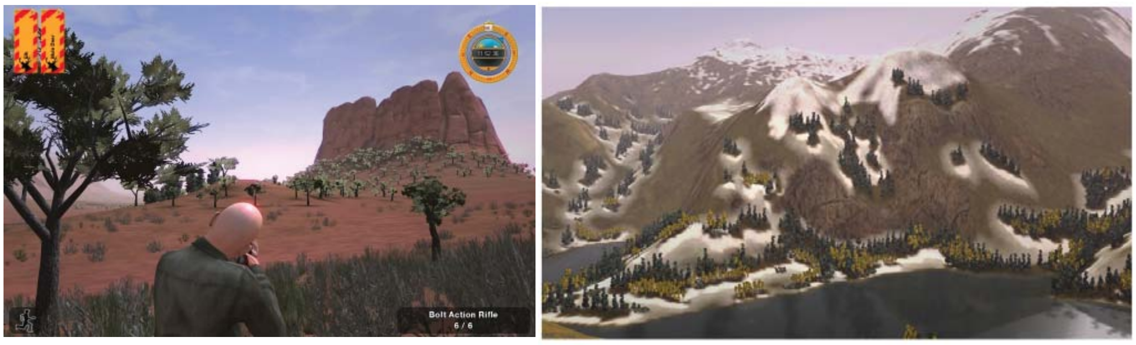


Figura 2.5: Captura de telas do jogo Deer Hunter Tournament®. Os terrenos ilustrados nessas figuras foram gerados com a técnica descrita nesta tese.

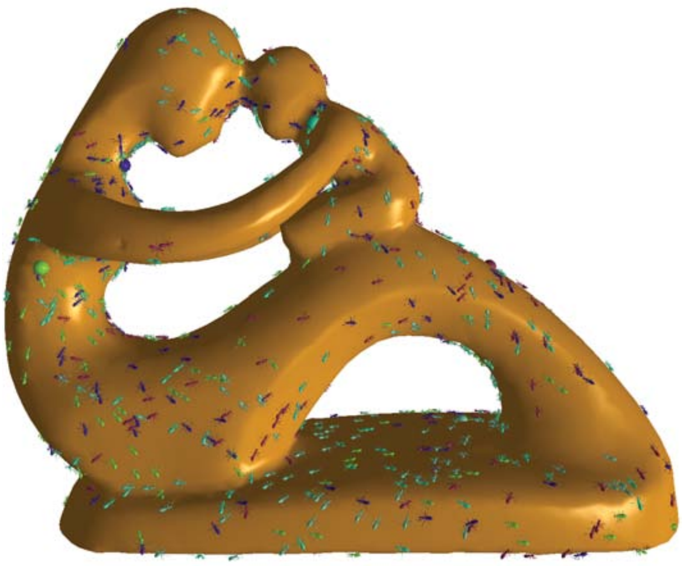


Figura 2.6: Essa figura ilustra o resultado do método de movimentação de agentes apresentado nesta tese. Existem 512 agentes sobre essa superfície.

dade do cálculo da distância. Por outro lado, reduzindo o número de triângulos no espaço paramétrico a perda de precisão é menor, como ilustrado na Figura 2.7.

Desta forma, é apresentado neste trabalho um novo método de cálculo aproximado de distâncias sobre malhas de triângulos que se baseia numa representação paramétrica e simplificada da malha, como ilustrado na Figura 2.8. Esta nova abordagem permite adicionalmente o controle da relação entre performance e precisão. Esse método é descrito no capítulo 5 e também é discutido em (TORCHELSEN et al., 2009).

Até então a utilização de algoritmos de movimentação de agentes, sobre superfícies arbitrarias, em aplicações interativas era limitada devido ao seu alto custo computacional, em parte devido ao custo do cálculo de distâncias. Normalmente o espaço de movimentação dos agentes ficava restrito a superfícies planas onde a complexidade da malha é baixa. A possibilidade de se poder controlar a relação entre performance e precisão nos cálculos de distância sobre malhas torna possível o seu uso em superfícies de maior complexidade e aplicações que demandem interatividade, como por exemplo os jogos de computador. A Figura 2.9 mostra essa limitação na complexidade das malhas em jogos, enquanto a Figura 2.10 ilustra essa limitação em artigos científicos.

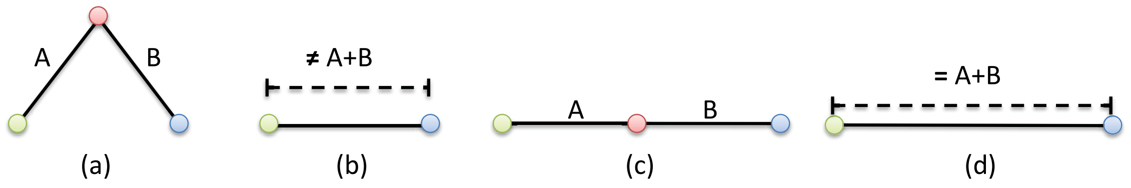


Figura 2.7: Diferença entre simplificar uma malha em espaço de objeto e espaço paramétrico. (a) ilustra uma malha simples. A distância entre os dois vértices inferiores dessa malha é igual à soma das arestas A e B. Se uma simplificação for aplicada em espaço de objeto removendo-se o vértice do centro temos como resultado a malha (b). A distância entre os mesmos dois vértices na malha (b) não é igual a A+B. Porém, se parametrizarmos a malha, para 1-D nesse caso (c), e depois removermos o vértice temos a malha (d). Nesse caso a distância entre os vértices é igual a A+B.

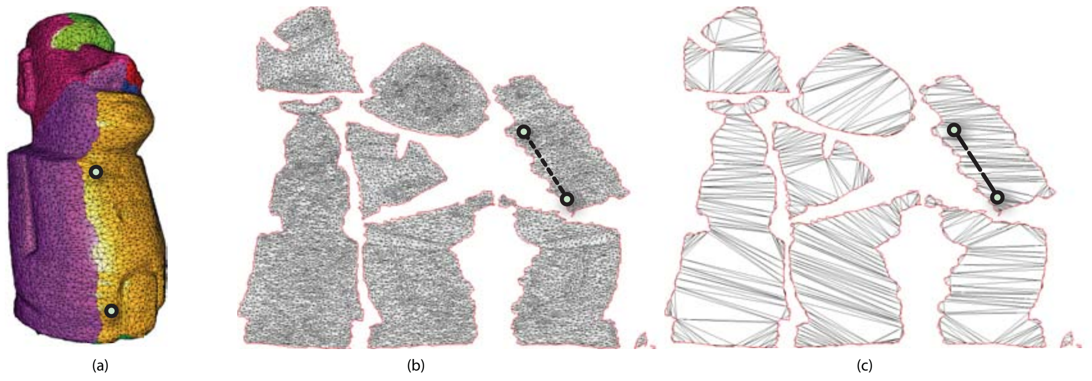


Figura 2.8: Cálculo de distância mínima sobre superfícies em espaço paramétrico. (a) é a malha onde gostaríamos de calcular a distância entre dois pontos. (b) é a parametrização da malha e (c) é a simplificação da malha no espaço paramétrico. O método apresentado nesta tese define como distâncias podem ser calculadas no espaço paramétrico (c).

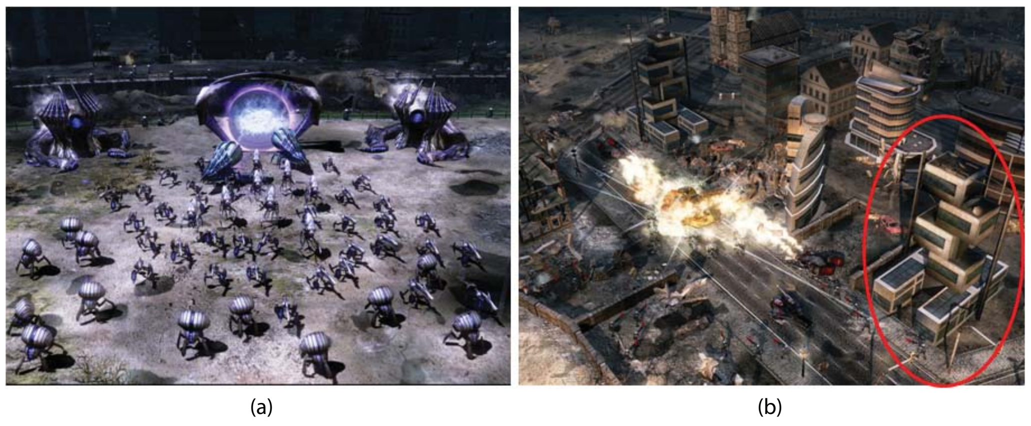
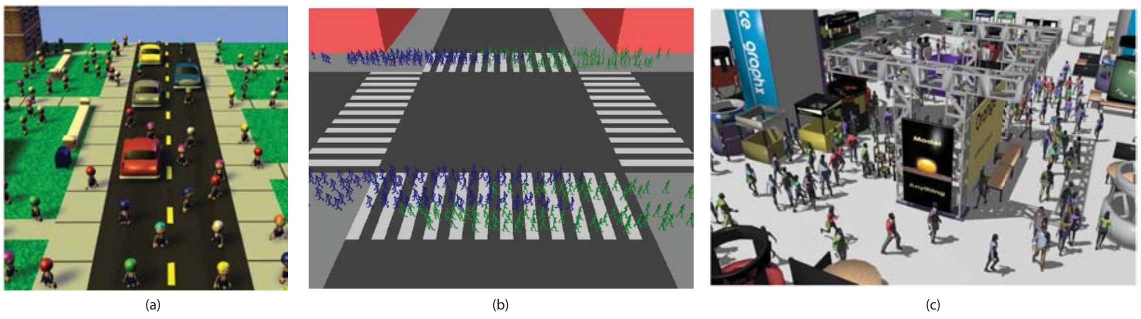


Figura 2.9: Captura de telas do jogo Command and Conquer^{TM}. Esse tipo de jogo usualmente necessita de técnicas de movimentação de agentes que possuam alto desempenho dado o número de agentes que precisam caminhar em tempo real. Dessa forma, as superfícies onde os agentes podem caminhar são limitadas, e geralmente são muito similares a superfícies planas como em (a). O resultado é um menor aproveitamento do ambiente virtual, por exemplo, os agentes não podem caminhar sobre o prédio do item (b).



(a)

Figura 2.10: Essa figura ilustra o resultado de diversos trabalhos que calculam caminhos para agentes. Veja que superfícies planares aparecem com freqüência. Os resultados são respectivamente dos seguintes trabalhos: (SUD et al., 2007), (SUD et al., 2007) e (BERG et al., 2008a).

Até o momento da escrita desta tese, não se tinha notícia sobre métodos de cálculo de colisão entre agentes que se movimentam sobre superfícies arbitrárias, exceto (TORCHELSEN et al., 2010). Esse método é paralelizável e foi implementado em GPU, sendo capaz de mover em tempo real 512 agentes sobre uma superfície complexa, como a ilustrada na Figura 2.6. Essa técnica é detalhada no capítulo 6 e também discutida em (TORCHELSEN et al., 2010).

2.3 Contribuições

Este trabalho apresenta as seguintes contribuições:

- **Displacement mapping eficiente utilizando o espaço paramétrico**: O aprimoramento do resultado visual da técnica de geometry clipmaps possibilitou a sua utilização em aplicações que necessitam de fidelidade visual. Já o cálculo de nível de detalhes utilizando o espaço paramétrico possibilita uma reconstrução adaptativa do *displacement map*, resultando em um melhor aproveitamento dos recursos computacionais.

- **Cálculo de distância no espaço paramétrico**: O cálculo de distância mínima entre pontos sobre a superfície de malhas de triângulo utilizando o espaço paramétrico é inovador. Resultando no aumento de desempenho de um algoritmo que possui muitas aplicações. Os ganhos de desempenho são em média de uma ordem de magnitude. O controle de precisão dos resultados e do desempenho possibilita a utilização dessa técnica em aplicações interativas, as quais eram limitadas dado o custo computacional dessa operação.

- **Movimentação de agentes sobre superfícies complexas**: A nova técnica de planejamento de caminhos para agentes é capaz de mover um número considerável de agentes sobre superfícies com complexidade nunca antes utilizadas em aplicações interativas.

3 RELATED CONCEPTS

This chapter introduces concepts relevant for understanding the topics discussed in this thesis. Other concepts related to a specific chapter are discussed in that chapter.

3.1 Displacement Mapping

The displacement mapping technique was introduced by Cook (COOK, 1984). The main idea is to describe surface details as elevations, which are encoded as a map, in much the same way as the surface of a terrain is encoded as a topographical map. In the context of displacement mapping, a topographical map is called a height map, whem the surface represented by the map is a terrain, for any other surface the map is called a displacement map. Given a displacement map (or a height map) and a surface without fine grained details, those details can be applied to the surface by disturbing the surface according to the displacement map. For example, figure 3.1 illustrates the process of reconstructing the surface of a terrain based on a displacement map. The displacement map representing a terrain can be constructed in an image editor or acquired by satellite scan, for example.

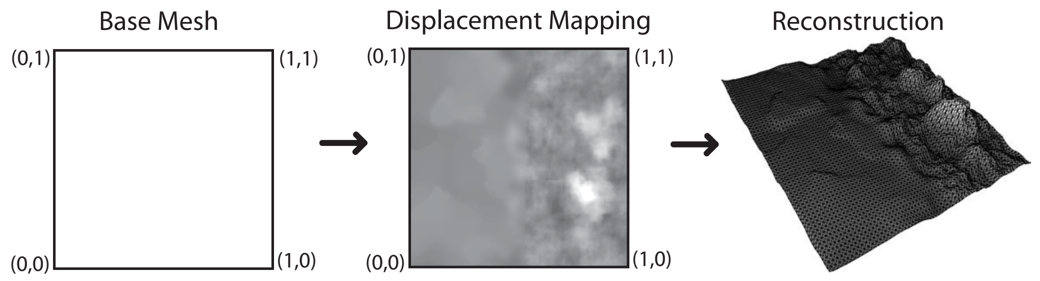


Figure 3.1: Reconstruction of a terrain encoded in a displacement map. The surface where the displacement map is mapped is usually called the base mesh. This mapping defines for each point on the surface of the base mesh a corresponding displacement value from the displacement map. Applying all the displacements to the base mesh results in the reconstruction of the terrain surface that was encoded as the displacement map.

Each point on the base mesh is displaced in the direction defined by the normal vector of the triangle where the point is located. However, before the base mesh can be disturbed it must have enough vertices to represent the displacement map. The ideal number of vertices is equal to the number of pixels of the displacement map. Without at least a 1:1 relationship between pixels (displacements) and vertices of the base mesh, the reconstruction of the terrain may lack resolution. Figure 3.2 illustrates three base meshes with increasing levels of resolution and the resulting reconstruction of the same displacement map. We can see that base meshes with different resolutions can result in different levels of reconstruction. This characteristic is one of the many advantages of the displacement mapping technique, which is explored in several works to generate different levels of reconstruction based on the content of the displacement map or position of the camera, for example.

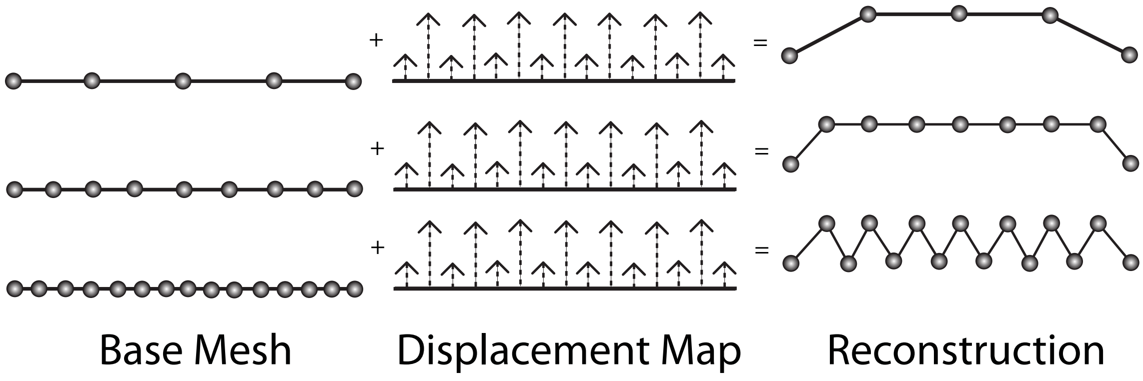


Figure 3.2: Three displacement map reconstructions based on three base meshes. We can see that increasing the resolution of the base mesh results in a closer approximation of the displacement map.

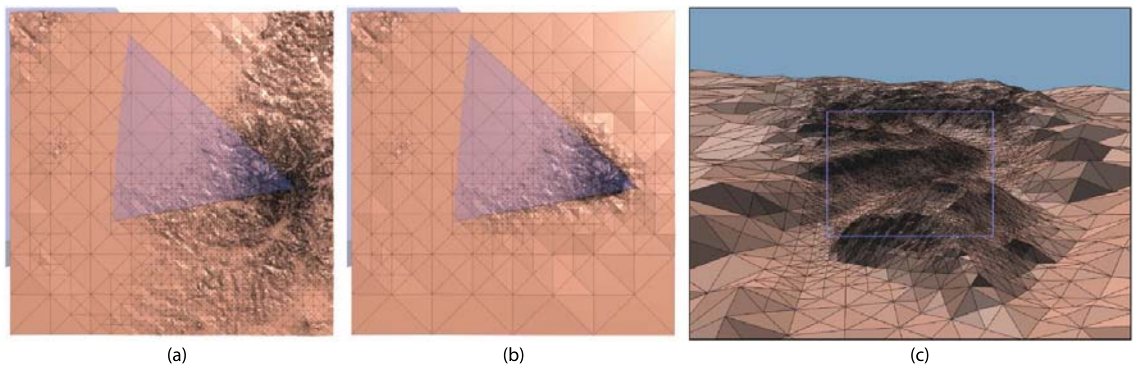


Figure 3.3: Reconstruction of a terrain from a displacement map on a rectangular base mesh, as in the work of Lindstrom and Pascucci (LINDSTROM; PASCUCCI, 2001). Item (a) illustrates the top view of the reconstruction, where the blue region denotes the view frustum of a camera located on the ground. We can see the larger density of vertices near the mountains valleys, where the differences between displacements (pixels on the displacement map) are larger (high signal frequency). Item (b) illustrates the same viewpoint, however, the resolution is not only determined by the signal frequency, but also considers the view frustum. The blue rectangle on item (c) illustrates the view of the camera denoted on the blue region of item (b).

The work of Lindstrom and Pascucci (LINDSTROM; PASCUCCI, 2001) introduced a technique to control the resolution of the displacement map reconstruction by inserting more vertices in the base mesh following two strategies. A strategy that inserts more vertices in regions of the base mesh that map to high frequency portions of the displacement map. The other strategy only subdivides the base mesh in regions inside the view frustum. Those two strategies can be combined, resulting in a multi-resolution reconstruction that is also view-dependent, optimizing the rendering efficiency. Figure 3.3 illustrates the reconstruction of a terrain as presented by Lindstrom and Pascucci (LINDSTROM; PASCUCCI, 2001). Other works that follow similar strategies to reconstruct displacement maps include Duchaineau et al. (DUCHAINEAU et al., 1997) and Hoppe (HOPPE, 1998).

The practical applications of an adaptive displacement mapping reconstruction, as in figure 3.3, have been limited due to the better performance of reconstructing a displacement map using pre-defined resolutions, in contrast to to a local analysis of the displacement map signal frequency. In most techniques that use a pre-defined resolution, the base mesh is subdivided according to the position of the camera, as in the work of Losasso and Hoppe (LOSASSO; HOPPE, 2004). In that work, the base mesh is subdivided in several rings around the camera, each ring with the same number of vertices. At each ring, from the camera to the horizon, the distance between vertices, on the rectangular base mesh, is increased. Figure 3.4 illustrates such mesh rings. The result is a constant resolution, which is higher near the camera, but is the same on low and high frequency regions of the displacement map. The advantage comes from the constant amount of data processed per frame, resulting in a smooth frame-rate, and a reduced amount of data sent from the CPU to the GPU, which is a common performance bottleneck. The work of Clasen and Hege (CLASEN; HEGE, 2006) and Livny et al. (LIVNY et al., 2008) follow similar strategies. A broader survey about interactive terrain rendering is presented by Pajarola and Gobbetti (PAJAROLA; GOBBETTI, 2007).

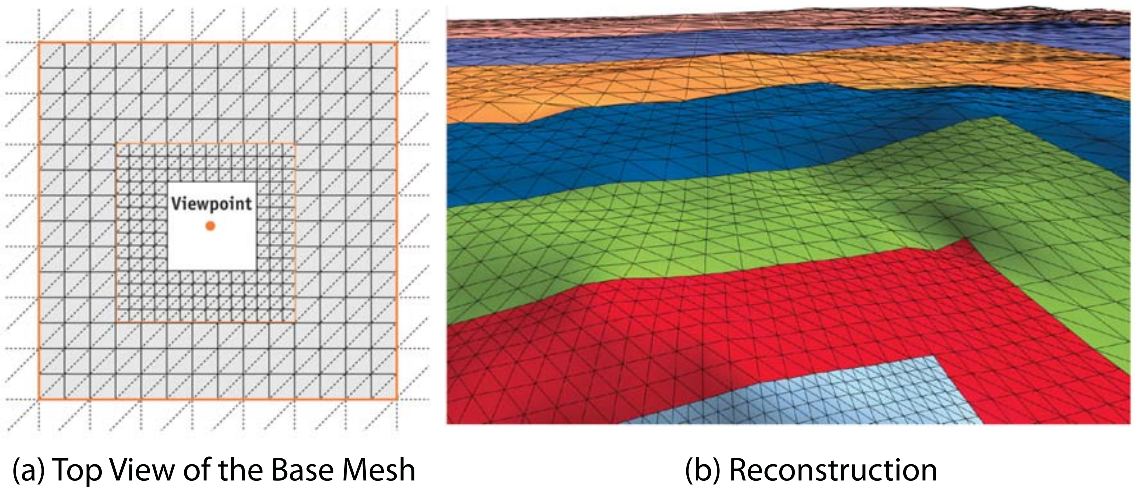


Figure 3.4: Geometry clipmaps technique, presented by Losasso and Hoppe (LOSASSO; HOPPE, 2004). The main idea is to define several rings surrounding the camera (item (a)) with fixed resolution, and store the rings on the GPU to avoid the need to transmit the mesh from CPU to GPU each frame. Item (b) illustrates the view of the resulting reconstructed mesh. We present more details about this technique in chapter 4, where we present an extension of the cited work.

The ability to insert details into surfaces with controllable resolution and its GPU-friendly characteristics are some of the features that make the displacement mapping technique a common choice to represent terrains in games. Furthermore, the same mapping used to place the displacement map over the base mesh can be used to map color textures onto the reconstructed surface. This is usually done by a combination of color textures, each representing some features of nature as snow, rocks or grass, for example. Figure 3.5 illustrates the process of coloring a terrain geometry. The mapping of colors

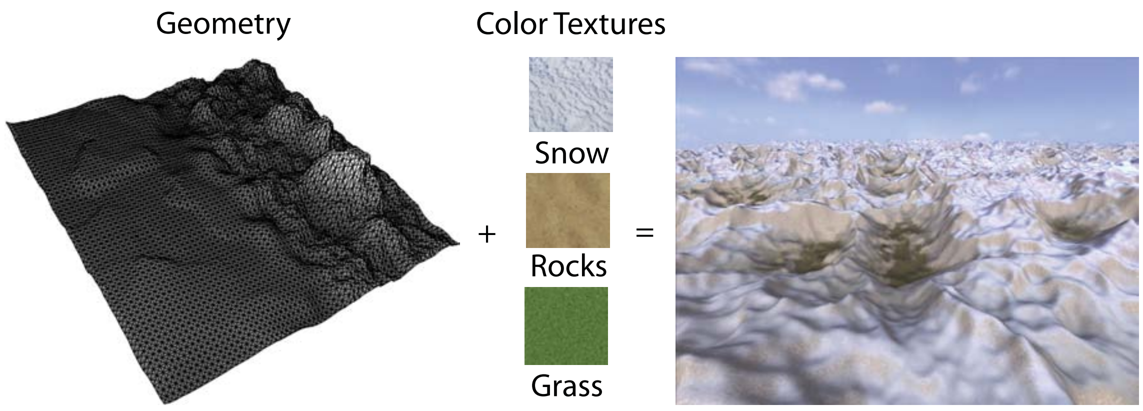


Figure 3.5: The displacement mapping technique used on terrain rendering involves a mapping of a displacement map in to a rectangular region, which is displaced. Using the same mapping we can color the resulting terrain, using textures, which can represent features of nature as snow, rocks and grass, for example. The compact storage of a complex terrain in the form of a texture, the real-time reconstruction and the reuse of the texture mapping for color mapping are some of the features that turn the displacement mapping technique popular in games and other interactive applications.

onto the reconstructed surface is not trivial for all displacement mapping reconstruction techniques, as in Losasso and Hoppe (LOSASSO; HOPPE, 2004). We present, in chapter 4, an extension to the work of Losasso and Hoppe (LOSASSO; HOPPE, 2004) to allow its practical uses including an efficient technique to map color textures.

Usually the base mesh used in terrain rendering is a rectangle in 3-D, which has a trivial mapping to a 2-D rectangular displacement map. However, displacement mapping is not restricted to rectangular base meshes. It can be applied to any mesh with a mapping from 3-D to 2-D; in other words, texture coordinates are needed to map each vertex of the base mesh to 2-D, which is the space where the displacement map is defined. In the next section we discuss the characteristics of such mapping (parameterization).

3.2 Mesh Parameterization

The interest about parameterization techniques has increased considerably in the past few years, as we can note from the number of surveys: (GRIMM; ZORIN, 2005), (FLOATER; HORMANN, 2005), (BOTSCH et al., 2006), (SHEFFER; PRAUN; ROSE, 2006), (HORMANN; LéVY; SHEFFER, 2007). Mesh parameterization is the process of establishing a mapping between two meshes. In this work we are only interested in the mapping between a mesh defined in $\Re^3$ and a mesh defined in $\Re^2$; a process usually called mesh unfolding. This kind of parameterization is particularly useful in computer graphics and probably the most notorious usage is texture mapping. Figure 3.6 illustrates texture mapping.

The parameterization of a mesh may have several properties. The most notorious are those that sustain some property from the folded mesh into the unfolded mesh, for example:

1. **Bijectivity**: Each point on the surface of any of the two domains has only a corresponding point in the other domain;

2. **Angle**: The internal triangle angles is the same in both domains;

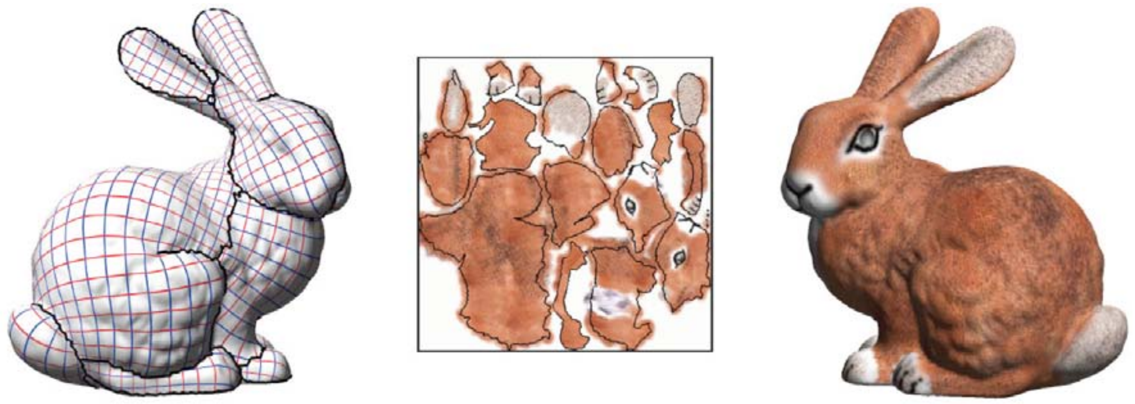


Figure 3.6: This figure, from the work of Lévy et al (LéVY et al., 2002), illustrates two texture mappings from the same parameterization.

3. **Area**: The area of corresponding triangles is the same in both domains;

These properties are used to classify each parameterization. A parameterization that preserves internal triangle angles is called a conformal parameterization, while a parameterization that preserves the triangle area is called authalic. The most difficult to obtain and most useful of the parameterizations is called isometric, which is both conformal and authalic. The isometric parameterization is ideal to map a color texture onto a mesh. A texture has a regular structure, which is a uniform distribution of texels. This means that a texel has always the same distance to the neighboring texels. Mapping a texture onto a mesh using an isometric parameterization sustains this characteristic over the mesh, in other words, there is no distortion in the texture over the mesh.

Other characteristics can be manually defined into the parameterization, according to the application. For example, mapping a texture onto a mesh usually begins with an automatic unfolding of a mesh. In some applications as games, coloring a mesh with a texture is an artistic process that requires manual manipulation of the parameterization. This is required because an automatic parameterization will map a small triangle in $\Re^3$ to a small triangle in $\Re^2$, resulting in few texels mapped into the triangle which means few details can be mapped. However, the artist may need more details and it can allocate more texels by increasing the size of the triangle in $\Re^2$. This information of how much detail is allocated to a triangle was explored in this work to develop an efficient displacement mapping reconstruction technique and is presented in chapter 4.

The difference, or the distortion, between the mapping of a triangle in $\Re^3$ and a triangle in $\Re^2$ can be measured using any of several existing techniques. A common metric to measure the stretch distortion was defined by Sander et al (SANDER et al., 2001). The method consists of defining a unit circle on the triangle in $\Re^2$, that maps to an ellipse on the same triangle in $\Re^3$ (Figure 3.7). The length of the principal axes of this ellipse are called γ and Γ. The principal axis, γ is equal to the smallest radius of the ellipse, while Γ is the largest radius. Both values are equal to 1 in the parametric domain because we have a unit circle. However, in the 3-D mesh the corresponding triangle may not have the same shape or area, compared to the same triangle in 2-D, turning the circle into an ellipse. The difference between the values of γ and Γ, between domains, gives an initial estimation of how much distortion is present in the parameterization.

An isometric parameterization corresponds to both, γ and Γ, equal to 1 for all triangles of the 3-D mesh. In a conformal parameterization both values are equal. However, those

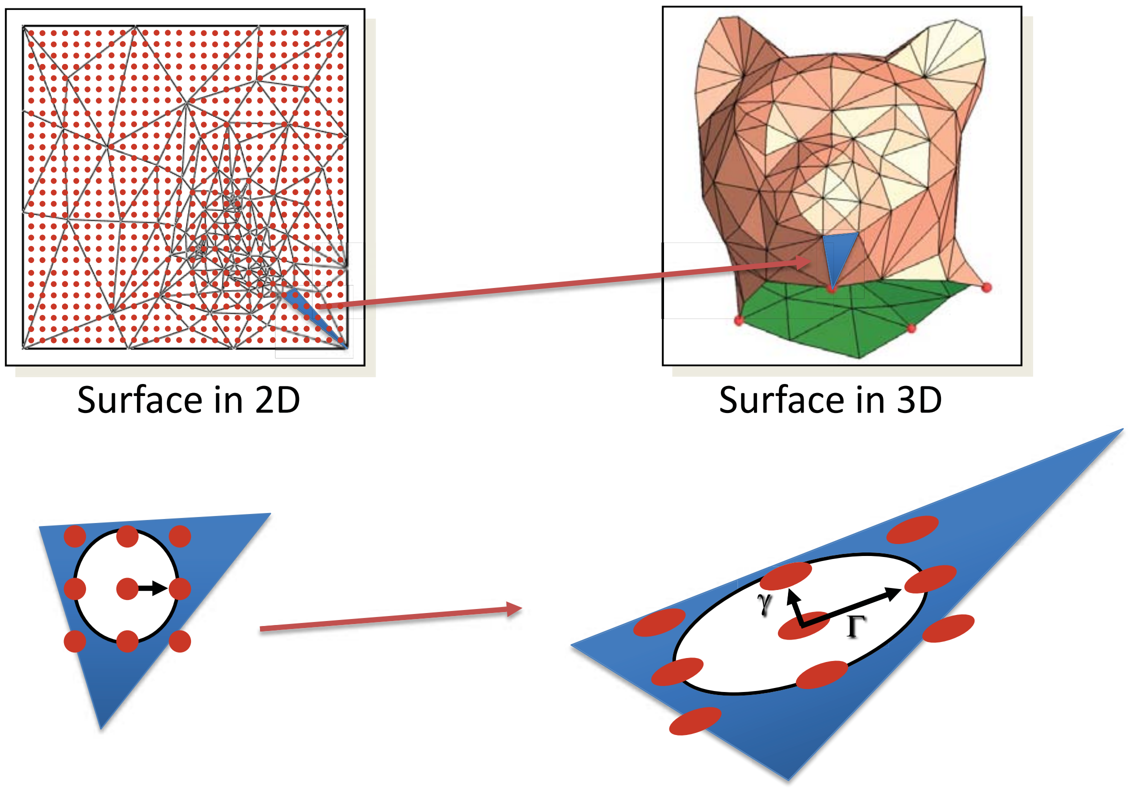


Figure 3.7: Distortion metric (SANDER et al., 2001). The method consists of defining a unit circle inside a mesh triangle, in the parametric domain, and mapping the circle back to the original triangle on the 3-D surface. Note the change from a unit circle to an ellipse, result of the difference in shape of the triangle between domains. This difference is called distortion.

values can be arbitrarily large, causing a triangle in $\Re^3$ to be mapped to a small triangle in the parametric domain. In an authalic parameterization the product of both values is 1. Still, one of those values can be significantly larger than the other. Therefore, an equilateral triangle can map to a thin long triangle of the same area. Finally, Sander et al (SANDER et al., 2001) used two metrics to compute the stretch distortion. The first is called the L^∞, which is equal to the larger of the values. The second metric is called L^2, which corresponds to the root-mean-square of both values. In other words, $L^\infty = \Gamma$ and $L^2 = \sqrt{(\gamma^2 + \Gamma^2)/2}$.

Figure 3.8(a) illustrates a parameterization and the color coding of the distortion present in the parameterization. Note the hot spots that represent regions with high amounts of distortion. The distortion can be reduced by increasing the number of charts. A chart is a group of connected triangles in the parameterization, for example, Figure 3.8(a) has only one chart while 3.8(b) has 3. Note from item (a) to (c) that by increasing the number of charts the distortion tends to decrease. The number of charts can be increased until each chart surrounds only one triangle, resulting in a parameterization without distortion. A triangle defined in $\Re^3$ mapped to $\Re^2$ would not suffer any distortion.

There are several works dedicated to finding a mesh segmentation that result in a low distortion parameterization. As mentioned by Julius et al (JULIUS; KRAEVOY; SHEFFER, 2005), finding an optimal set of charts is NP-Hard. Most methods use heuristics in the search for local minima, instead of the optimal solution.

A developable mesh is a mesh that can be unfolded without distortion, while a mesh

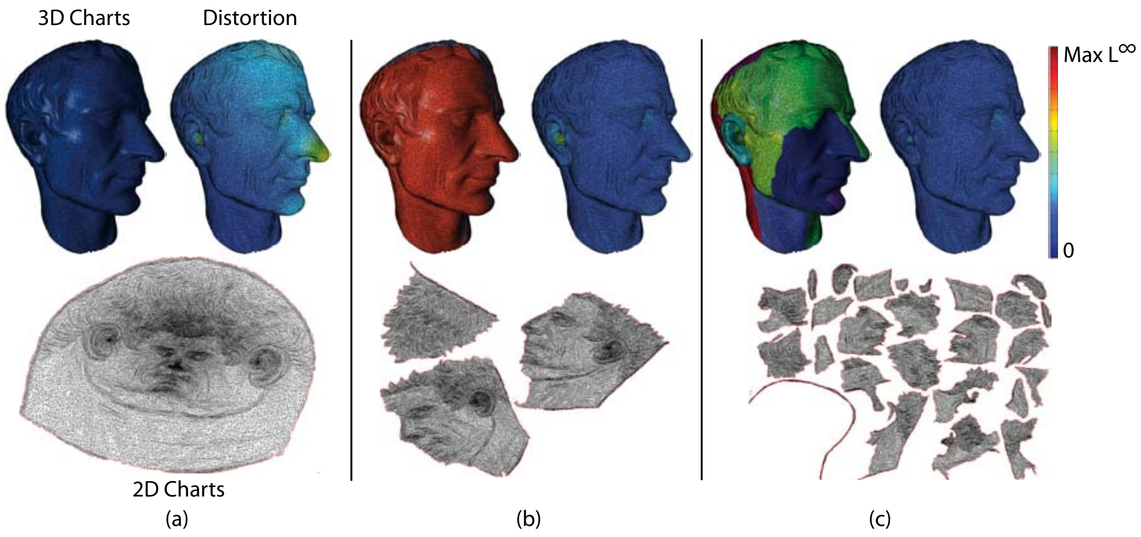


Figure 3.8: The distortion present in the parameterization can be reduced by increasing the number of charts. The maximum and average L^{∞} present in (a), respectively, is 7.8253 and 1.3189. In (b) 5.3271, 1.0482 and (c) 5.1411, 1.0112.

that is unfolded with low distortion is called a quasi-developable mesh. There isn't a specific distortion threshold that determines that a mesh is quasi-developable. It is only a term to indicate that a mesh can be unfolded with low distortion. As proved by the Minding's theorem any two surfaces with the same Gaussian curvature, at corresponding points, are locally isometric. In other words, mapping one surface into the other results in a parameterization without distortion. Meshes with zero Gaussian curvature everywhere are considered developable meshes, because the Gaussian curvature of the plane is zero. Figure 3.9 illustrates two developable meshes.

A common example of a developable surface is mechanical parts. Those parts are commonly constructed by folding or welding plane pieces of metal, as illustrated in figure 3.10. We can see on the figure that printing on paper the parameterization of the parts and gluing the pieces together results in the same surface of the triangular mesh; no distortion is present. However, most surfaces are not developable, as a result distortion will be present in the parameterization. Figure 3.11 illustrates the same process of reconstruction of the mesh surface, however, those meshes are not developable. Note that the reconstructed surfaces are distorted in relation to the mesh surface.

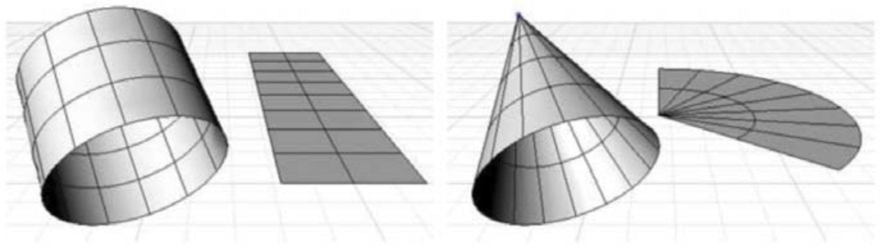


Figure 3.9: Examples of developable surfaces, from the work of Julius (JULIUS, 2006). Note the rectangular shapes mapped over the surfaces. They have the same shape and area in $\Re^3$ and $\Re^2$. This is only possible because those surfaces are developable, which means they can be unfolded without distortion.

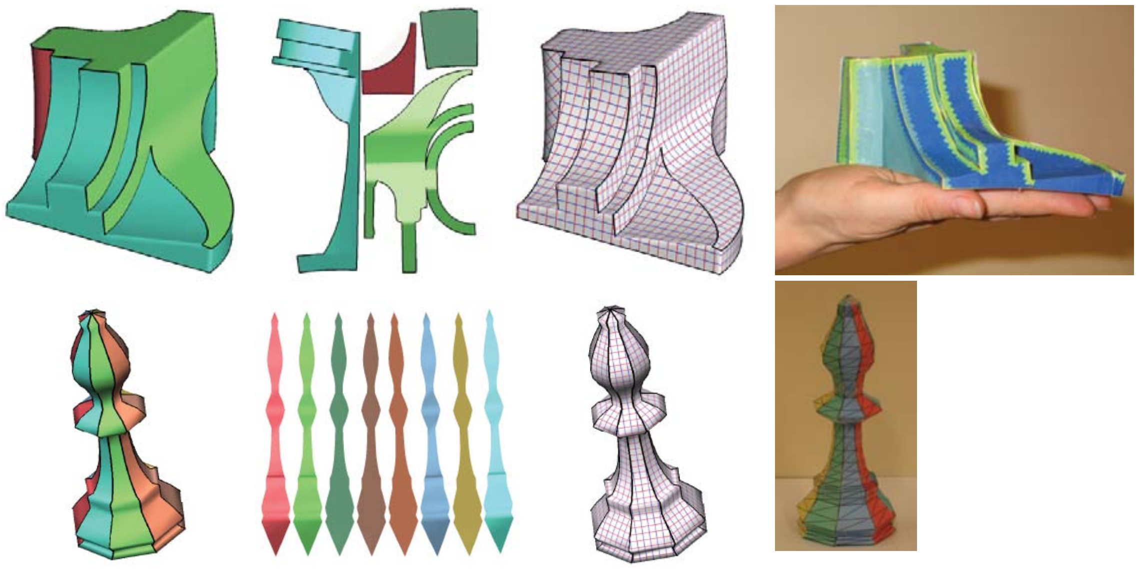


Figure 3.10: Examples of mechanical parts from the work of Julius et al (JULIUS; KRAEVOY; SHEFFER, 2005). Note the similarity between the hand-made mechanical part and the virtual counterpart. This level of precision is only possible because the parameterization of a developable surface preserves distances. This characteristic was explored in the contribution presented in chapter 5.

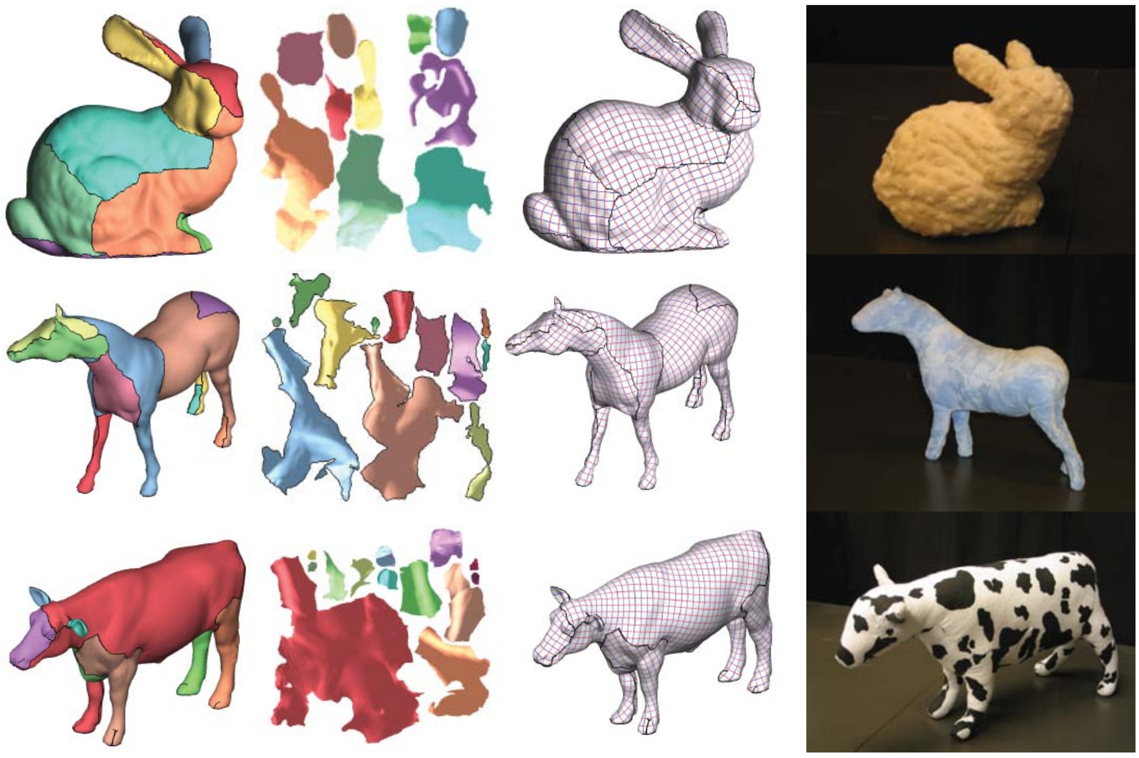


Figure 3.11: Examples, from the work of Julius et al (JULIUS; KRAEVOY; SHEFFER, 2005), of triangular meshes that are not developable. However, quasi-developable charts can be found. Note that the reconstructed models are similar to the virtual models. However, some distortion in the shape of the models is visible.

3.3 Graphics Processing Unit - GPU

The Graphics Processing Unit (GPU) was initially designed only to generate images. Currently the GPU is becoming a general processing unit. However, with a different philosophy compared to the CPU. The GPU data and processing organization is designed to maximize the performance of massively parallel algorithms. The current GPU design implies on each thread executing the same code, but on different data. On the other hand, the CPU is optimized to execute different algorithms on each thread. This difference in design is illustrated on Figure 3.12.

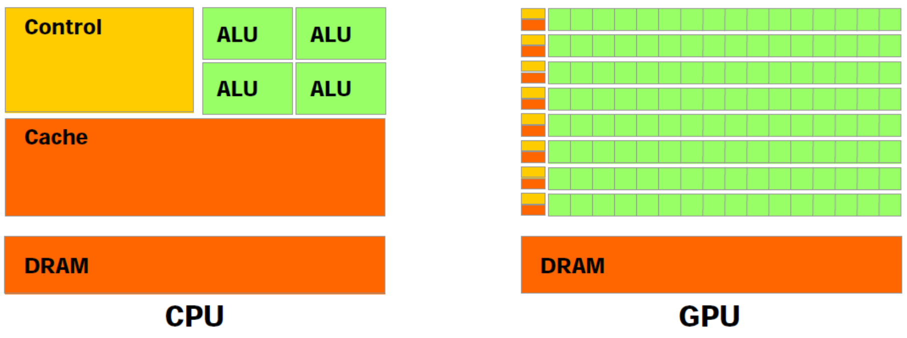


Figure 3.12: Difference between CPU/GPU designs. Note the number of arithmetic logic units (ALU) in each processor. The GPU emphasizes arithmetic operations, while the CPU emphasizes memory access. This figure is part of the NVIDIA CUDA Programming Guide (NVIDIA, 2009).

The CPU is optimized to predict data access patterns and pre-cache the data before it is requested, thus reducing the latency between request and delivery. Note, on Figure 3.12, the large CPU chip area dedicated to Cache compared to the GPU. In contrast, the GPU emphasizes the number of ALU units. This difference between processors imposes advantages and limitations to each one of them.

Some limitations of the GPU philosophy guided the development of this work. For example, in an application where multiple threads can read/write from the same memory location, atomic operations are necessary. An atomic operation is an operation, or a group of operations, that block the execution of other operations until the atomic operation is completed. An atomic operation implies a blocking on the execution of all threads, besides one, while it is accessing the data shared by all threads. In a massively parallel environment, as the GPU, minimizing the number of atomic operations is crucial; otherwise, the execution pattern would converge to a serial pattern, which is the best pattern to be executed on the CPU.

Another limitation imposed by the philosophy of current GPUs is the computational cost of memory allocation. Due to the GPU massive number of thread and single algorithm in each thread, the only variable on each thread is the data. Before any thread can execute, data must be distributed to all threads. This means that any new amount of data must be redistributed across all threads before the processing can continue. Every time a thread requires more memory all threads must stop while memory is allocated. The result is a high cost for any memory allocation. Any algorithm implemented on the GPU should minimize the use of memory allocation during the data processing, and maximize the allocation before the algorithm execution.

The philosophy of multiple instances of the same algorithm designed on the GPU is very similar to the agent path planning problem, which motivated the agent path planning algorithm presented in chapter 6. Each agent in a crowd can be controlled by the same algorithm and a crowd implies a massive number of agents, exactly the best pattern to be implemented on the GPU. However, in a real crowd, agents share information, visual or verbal, to avoid collision. Note that sharing information across instances (threads) has a high computational cost in the GPU; due to the concurrent nature of the operation. Usually, this cost can be reduced by increasing the number of computations and reducing the number of data accesses. That way, the GPU can execute portions of the algorithm while it is accessing data for the next portion. This is possible because each operation, data access and data processing, are implemented in different portions of the GPU chip, see Figure 3.12.

The algorithm presented in chapter 4 was implemented on the GPU due to another specific design of the GPU: hardware texture mapping support. The displacement mapping technique is based on texture mapping, turning the GPU the ideal processor. This support is present on current GPUs due to the origin and most notorious application of GPUs: computer graphics. In computer graphics, triangular meshes and textures are common techniques used to represent surfaces, resulting in GPU native support. The GPU was initially designed only for graphical applications, particularly the rasterization pipeline. The rasterization pipeline, illustrated on Figure 3.13, was until recently the only available processing pipeline on GPUs.

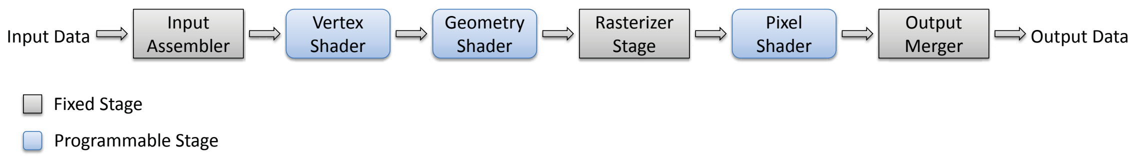


Figure 3.13: This figure illustrates the GPU graphics pipeline. The vertex shader operates on vertices while the geometry shader operates on groups of primitives (i.e. triangles or quadrilaterals). The rasterizer stage is responsible for the color definition of each pixel resulting from the discretization of each primitive.

Initially, only some stages of a fixed pipeline (as seen in the rasterization pipeline of Figure 3.13) were programmable. Currently, though, a complete pipeline can be defined. However, the rasterization pipeline still is present on current GPUs. A fixed pipeline allow the GPU designers to optimize the hardware performance. There are programming languages for each pipeline. The GLSL and HLSL are some of the most used in the graphics pipeline, while CUDA and OpenCL are the programming languages for the general pipeline. The choice between the programmable pipeline and the graphics pipeline, when implementing an algorithm, is mostly dependent on the algorithm output. Some algorithms when implemented in the graphics pipeline achieve higher performance if the final output is a image. A rendering algorithm implemented in the programmable pipeline would require a cycle on the programmable pipeline and an additional cycle in the graphics pipeline to render the algorithm result. However, if the entire algorithm can be implemented in the graphics pipeline there is a gain of performance. That is the main reason for the contributions to displacement mapping presented in this thesis to be implemented in the graphics pipeline instead of the programmable pipeline.

However, as mentioned, some algorithms are not suitable to be implemented in the GPU, which is the case of the on-surface distance computation technique presented in chapter 5. Most of the algorithm is serial and memory allocations are frequent during the algorithm execution, turning the CPU the ideal processor.

4 DISPLACEMENT MAPPING ON THE PARAMETRIC DOMAIN

In this chapter two contributions are presented, each derived from efficient usage of the parametric domain. The first is dedicated to efficient rendering of displacement mapping, with focus in its practical use in games. The second contribution is a novel level-of-detail method for the reconstruction of surfaces encoded in displacement maps. A new refinement criteria was derived from the area of the parameterization dedicated to each triangle of the base mesh, resulting in efficient level-of-detail reconstruction of displacement maps.

4.1 Efficient Rendering of Displacement Mappping

Rendering terrains with a high degree of realism is an ongoing need of the computer game community. To render scenes with increasing sizes and complexity, several terrain rendering algorithms have been proposed in the literature. An important approach is called Geometry Clipmaps (LOSASSO; HOPPE, 2004), and relies on the position of the viewpoint to create a multi-resolution representation of the terrain using nested meshes. This approach was further improved and most of the processing computation was transferred to the GPU (ASIRVATHAM; HOPPE, 2005). Although very efficient and allowing support for large terrain models, such an approach presents shortcomings when used for computer games. For instance, there was no support for color texturing, real-time deformations, solutions for older shader models, and a mesh tessellation designed for artistic editing.

In this section, the geometry clipmaps approach is revisited and several modifications to the original algorithm are presented. The contributions to geometry clipmaps allow its use in a wider variety of GPUs and shader models. Speedups and modifications based on needs of computer games resulted in a technique with high FPS rendering capability, and in an attractive solution for terrain rendering in games.

4.1.1 Review of Geometry Clipmaps

The geometry clipmaps technique is based on building a set of nested meshes (called clipmaps) surrounding the viewpoint with varying levels of detail, keeping the higher resolution clipmap near the viewpoint. One of the main advantages of this technique is a reduction on the CPU-GPU bandwidth, due to only sending updated heights to vertices defined on a predefined 2-D mesh already stored in GPU memory. Figure 4.1 illustrates a top-down view of a terrain using different shades to illustrate each clipmap (lighter shades are associated with higher mesh resolution), and the resulting rendering of clipmaps that

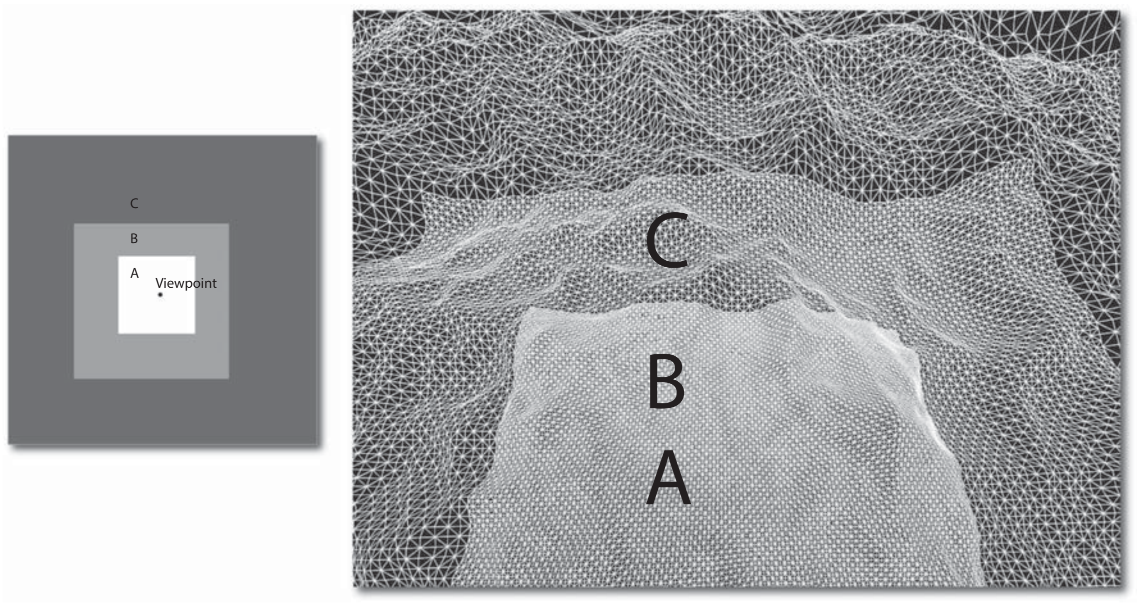


Figure 4.1: Top-view representation of three clipmaps (A, B and C) that form a terrain mesh (left) and rendering of several clipmaps mesh (right). Clipmaps A and B use the same resolution because they are close to the viewer, and there is no visible distinction between them in the rendered image, while the following clipmaps use decreasing resolutions to represent the terrain.

represent a terrain mesh.

Each clipmap is formed by grouping rectangular blocks, forming a "ring" surrounding the viewpoint. Because every clipmap ring must lie inside another ring and present a smooth transition across different resolutions, certain conditions must be imposed on the number of vertices in the perimeter of each ring. A more detailed discussion is presented on (ASIRVATHAM; HOPPE, 2005). Here we simply state that the number of vertices must be odd and represented by one minus a giving power of $2^n - 1$.

In figure 4.2 we illustrate how clipmaps are represented by rectangular grids chosen from a basic set of building blocks (each represented internally as a mesh patch). In order to keep the same amount of blocks on each clipmap ring, only three different types of building blocks are used: Block, formed by m x m vertices, Ring Fix-up formed by m x 3 vertices, and Interior Trim formed by (2m +1) x 2 vertices. The Ring Fix-up exists because we cannot close the ring only with the m x m Blocks, while the Interior Trim is used to move the clipmaps over the height field.

The way each clipmap is represented by building blocks depends heavily on the position of the viewpoint, and changes as the viewpoint moves. Figure 4.3 illustrates the movement of the Interior Trims while the viewpoint moves to the right. Moving the Interior Trims changes only the clipmap where it belongs, as well as the inner clipmaps, but the outer clipmaps are kept unchanged, avoiding an update of its values. Details on how building blocks change while the viewpoint move can be found in (ASIRVATHAM; HOPPE, 2005).

In the sections to come several modifications to the geometry clipmap algorithm are presented to make it amenable for computer games.

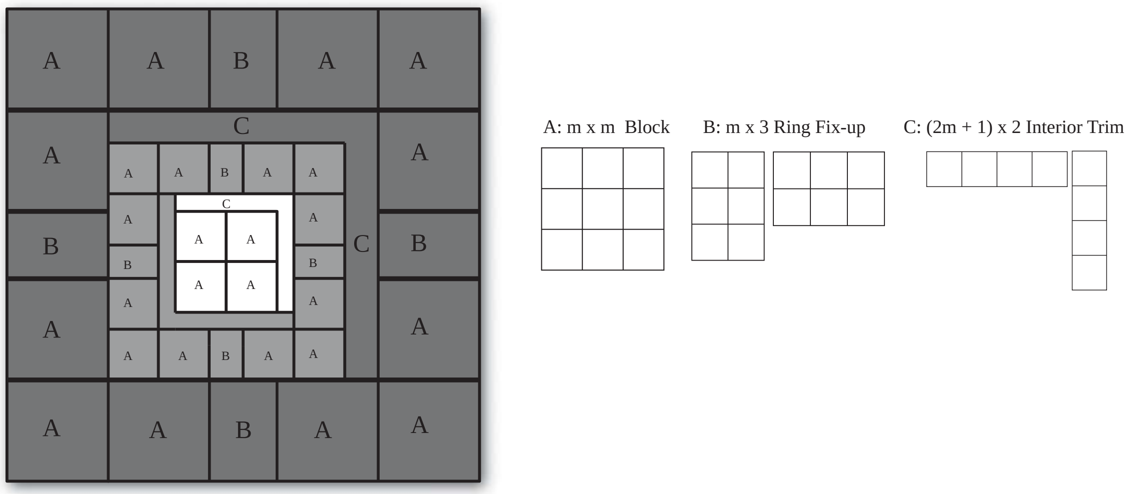


Figure 4.2: Three Clipmap rings, white, light gray and gray (left) are formed from a basic set (right) of three different rectangular building blocks.

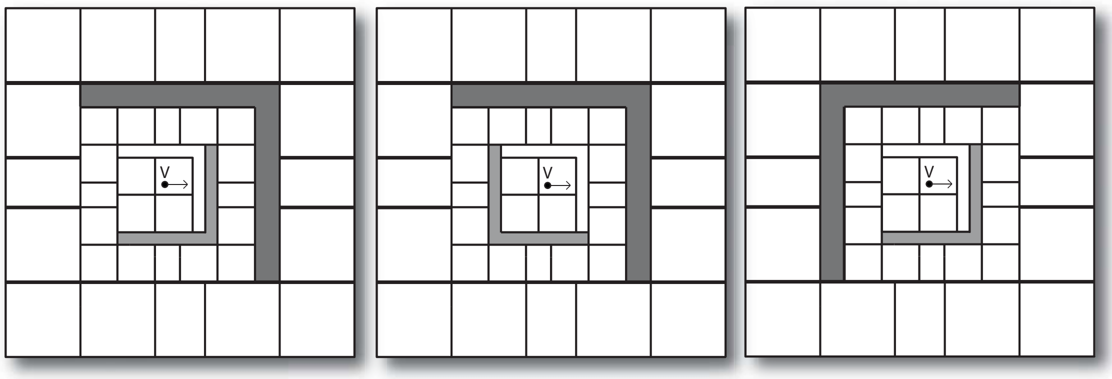


Figure 4.3: Moving the viewpoint three consecutive vertices to the right. The Interior Trim changes sides in different ways on each clipmap ring. In the most internal ring, the Interior Trim changes sides at every movement, while the next nested ring moves at intervals every other movement.

4.1.2 Terrain Tessellation

Editing capabilities is a major concern for a game developer. Terrains represented by a height field are parameterized into a support planar mesh (also known as base mesh), with each vertex associated a given height that is displaced along the vertex normal vector. Editing a terrain mesh based on a height field is simple, the artist just increases or decreases the heights of each vertex; in other words, only the heights of the displacement map are changed. The impact on adjacent triangles while changing the height of a given vertex (item A from figure 4.4) depends on the tessellation pattern used. The original tessellation pattern used in the clipmaps technique is illustrated in figure 4.4 (original pattern). The drawback of this pattern is that there is a hexagonal region of neighboring triangles that is affected when displacing vertices, which was found to confuse the artist in some situations. A modification to a symmetrical pattern is illustrated in figure 4.4 (practical pattern). This simple modification allowed the artists a better estimation on the impact of each displacement, even to a point where the artist could edit the terrain without

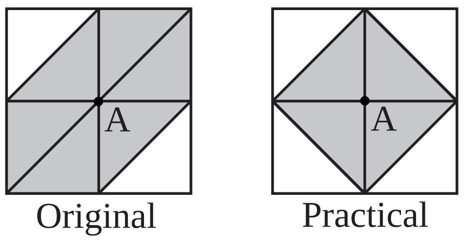


Figure 4.4: Tessellation used in the original clipmaps work (left) and practical solution (right). Moving the vertex A changes the surface in the shaded region. The surface resulting from the move of the vertex A in the practical tessellation is more easily estimated, even without wireframe.

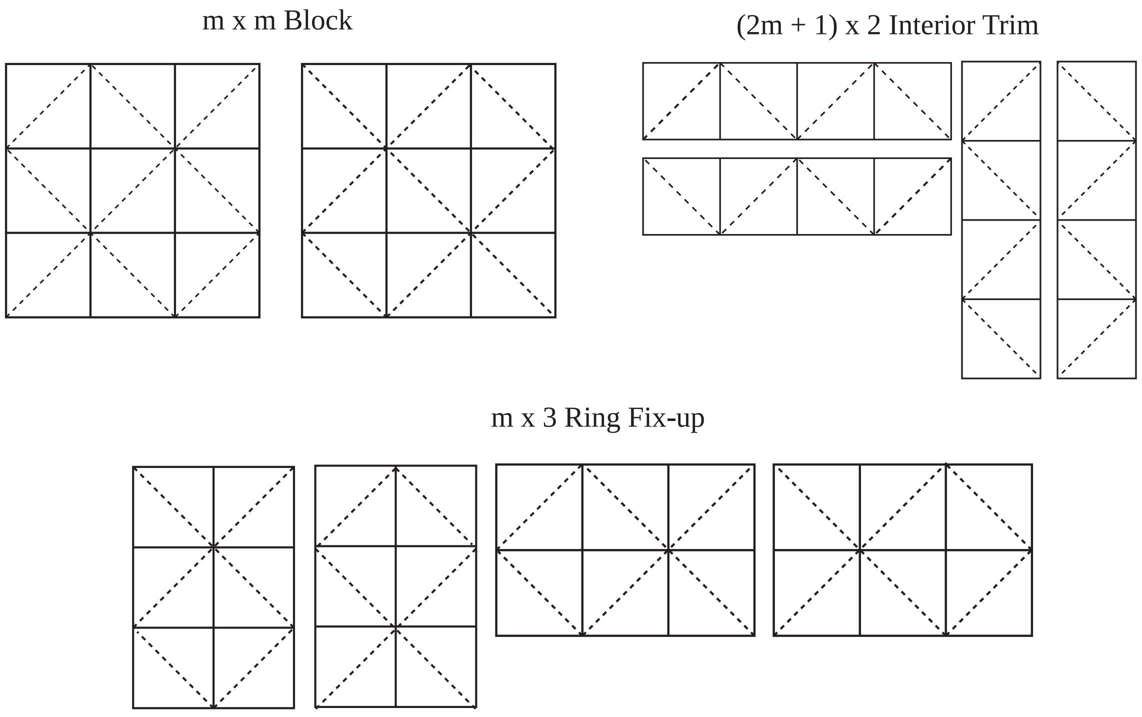


Figure 4.5: Regular grids triangulations using an artist-friendly tessellation.

using wireframe, which they prefer.

The only issue that needs to be addressed when using this new tessellation is that it requires pairs of shifted regular grids, so that their combination tessellates the plane. It is important to observe that using twice the number of tessellations per terrain only doubles the number of triangle indices, but does not double their coordinate locations. Therefore, it suffices to have an additional index buffer per regular grid, but we need to identify which one to use to keep tessellation at the grids frontiers. Figure 4.5 illustrates the final tessellation of the regular grids.

The choice of which of the two tessellation patterns (figure 4.5) will be used can be made by checking the parity of the texture (height field) coordinates associated to the top-leftmost vertex of the grid. By simply checking the parity, we can select a tessellation based on the predefined solutions above, as illustrated in figure 4.6. The same can be done for all other configurations, thus enforcing correct match between neighboring grids over the entire height field.

A final terrain triangulation using this new tessellations is illustrated in figure 4.7.

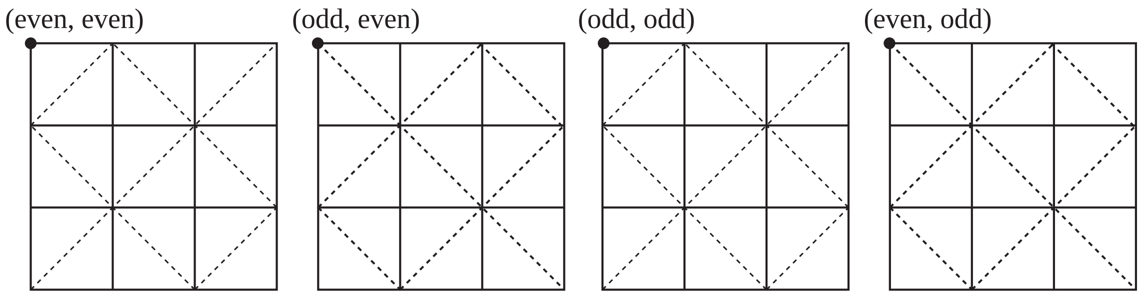


Figure 4.6: Choosing tessellation patterns is based on the parity of the texture (height field) coordinates associated to the top-leftmost vertex.

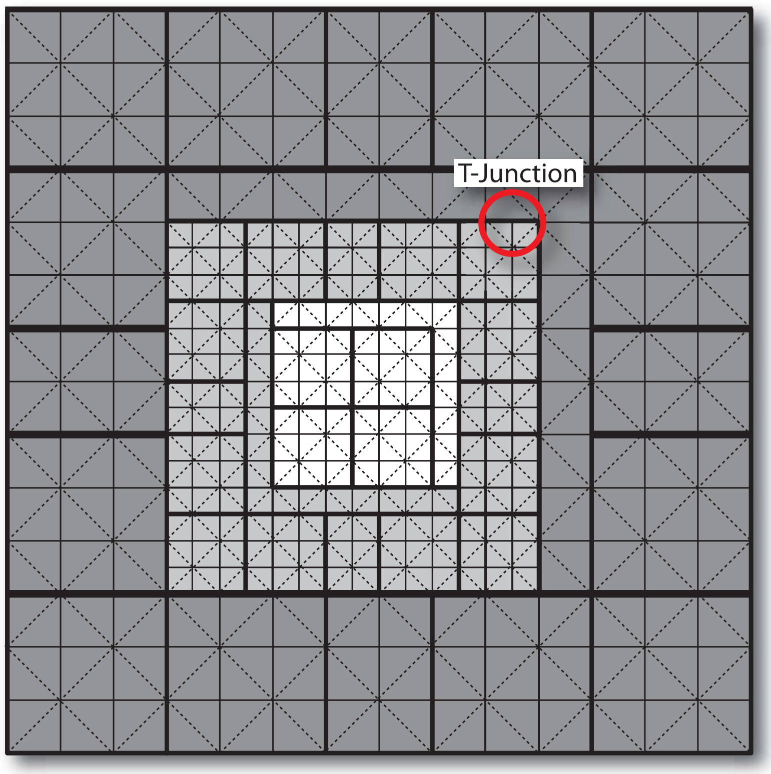


Figure 4.7: Terrain triangulation, illustrating tessellation matching between regular grids, from same clipmap ring.

4.1.3 Height-Field

The original Clipmap work uses a compression scheme for the storage of the height field that might not be attractive for a game implementation. Since it only decompresses the height field around the viewpoint, checking for collision of objects in regions away from the viewpoint requires a local decompression that might be costly when a lot of objects are present, for example in RTS games. Also, real-time editing updates of the height field would require a decompression and compression scheme that guarantees frame rate, which is not present in the original work. Therefore, in this case, it is preferred to avoid the compression scheme. This solution obviously limits the size of the terrain, but it is still possible to obtain reasonably large terrains.

The height field in the Clipmap work of (ASIRVATHAM; HOPPE, 2005) was stored on the GPU using a texture. It requires texture reads in the vertex shader that can only be accomplished in shader model 3 (SM3) and higher. In addition, handling collisions is also an issue using this approach. In this work, the height field is manipulated in the CPU, and the GPU is used to apply the displacements (vertex shader) and lighting (pixel shader). This way, allowing the technique to be implemented on previous generations of GPUs.

Every clipmap ring has an associated resolution that represents the mapping of their vertices to the vertices of the original height field. Full mapping (1-to-1 mapping) is associated to the first and second clipmaps closer to the viewpoint (item A and B from figure 4.1). The following clipmaps decrease their resolution by a factor of two with respect to the inner clipmap.

Processing every vertex requires three heights to be accessed from the height field, one for the level of detail in the clipmap which the vertex belongs and two others used to generate the vertex height in the outer clipmap. Both are used in the geometry morphing in the perimeter of the clipmap.

4.1.4 Morphing Between Different Resolutions

Different levels-of-detail might cause T-junction vertices (see figure 4.7), which must be removed to avoid cracks in the mesh. Similar to the original clipmap work, we use a morphing area (figure 4.8) to morph between different resolution levels.

This is accomplished by subdividing the work between the CPU and the GPU. Instead of sending to the GPU just one height associated to each vertex in the mesh, we send two heights, corresponding to the level-of-detail in the ring where the vertex belongs and one corresponding to its adjacent ring. Using these two heights we can slowly interpolate them, producing a smooth transition from one resolution to the next, thus avoiding cracks. Since the bandwidth is one of the most common bottlenecks the two heights are sent using 16 bits each, resulting in 32 bits per vertex, the minimum data structure that can be used to create a vertex on the GPU. The height from the ring where the vertex belongs is directly read from the height field but the height that corresponds to the next ring is generated on-the-fly.

The size of the morphing area depends on how heights vary in the terrain. For smoother regions, only a small area might be necessary, but for highly irregular regions a larger region might be needed. The morphing area is illustrated with different shades in figure 4.8.

In figure 4.9, we illustrate crack problems that can happen across different rings. Vertices B and E from figure 4.9 can generate a crack in the mesh if they use one of the heights from the height field. Those vertices must be moved to a position that exists in the next ring to avoid cracks. The only directions the vertices can go are up or down with respect to the supporting plane (base mesh). The only position in the next ring that exists in those directions is over one of the edges formed by the neighbors of B and E. Figure 4.9 illustrates the four existing neighboring configurations used to identify that edge. Those configurations exist because all rings use the same tessellation.

Morphing vertices across the frontier of two rings avoid cracks but does not create a smoother transition between rings, and therefore all vertices inside the morphing area are required to be morphed. Vertices H and K in Figure 4.9 illustrate the other two configurations of neighbors for vertices inside the ring that must be handled to identify the morphing target (position of the vertex in the surrounding ring).

The morphing target is the second height sent to the GPU and is generated for all vertices that do not exist in the level-of-detail from the surrounding ring. The vertices that exist in both rings like A and C have the same height, which is duplicated in both channels.

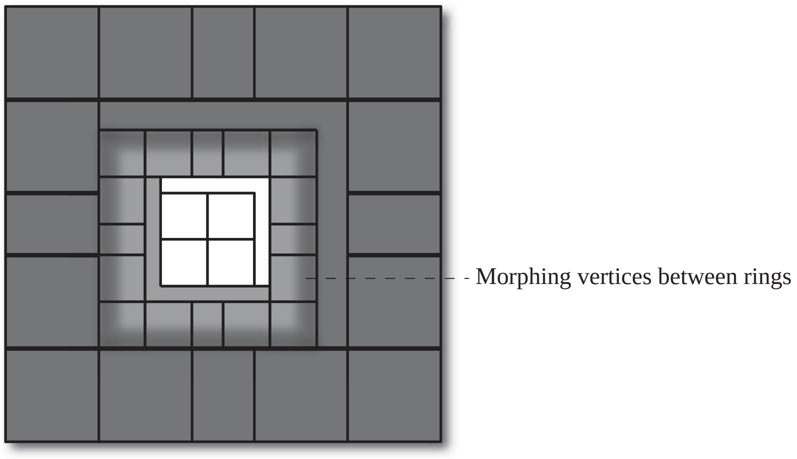


Figure 4.8: Morphing area denoted by the gray gradient between the first and second ring illustrates the smooth morphing of the vertices from one resolution to the next.

4.1.5 Texture Layers

The traditional texturing technique used in terrain rendering is based on texture layers. A texture layer is a covering of the entire terrain that can represent different materials, such as grass, rocks, snow, etc. To determine the final color of the pixels, an alpha texture between every layer defines the contribution of the layer to the final color of the pixel.

The layers are formed by three components:

1. **Color Texture**: A texture corresponding to grass, rocks, snow, etc;

2. **Tiling**: Defines how many times the color texture repeats itself between vertices of the terrain, defined as α;

3. **Alpha Texture**: Defines the contribution the color texture has to the final color of the pixel;

Figure 4.10 illustrates an example of a terrain mesh defined over the support plane with two texture layers. The first layer uses an alpha texture to render the color texture at regular intervals. The second layer uses the alpha texture in such way that the color

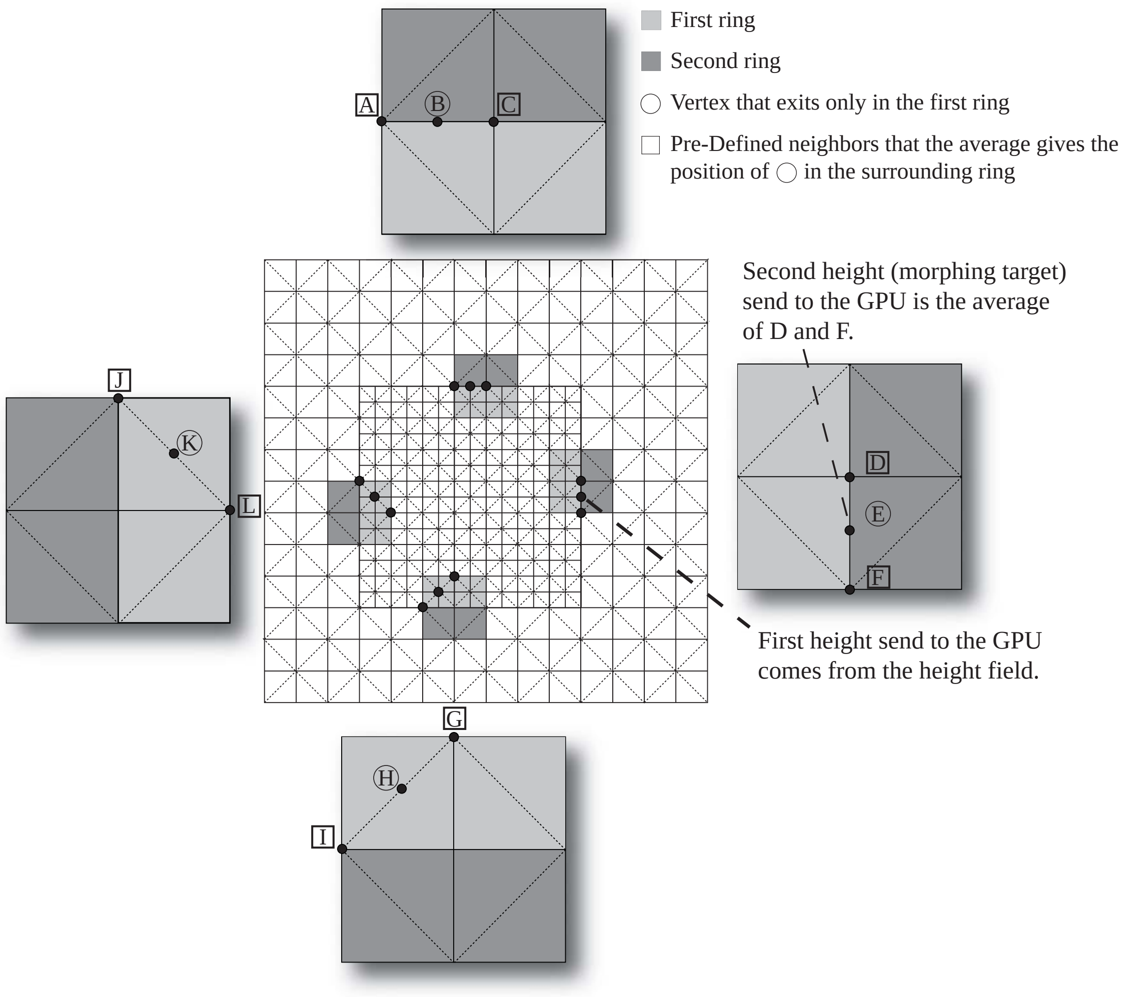

First ring
Second ring
Vertex that exits only in the first ring
Pre-Defined neighbors that the average gives the position of ◯ in the surrounding ring
A
B
C
Second height (morphing target) send to the GPU is the average of D and F.
D
E
F
J
K
L
First height send to the GPU comes from the height field.
G
H
I

Figure 4.9: Smooth transition between rings: vertices of a ring that do not exist in the surrounding ring are morphed to positions in the middle of the edges formed by neighbors of the vertex (B, E, H and K). Shaded squares illustrate the existing vertices configurations and the pre-defined vertices neighbors that are considered for the morphing target calculation.

texture appears at regular intervals, but applies gradients to slowly show the color texture. The gradient can be used to represent smooth changes between layers, such as to simulate discontinuities on materials in the terrain (e.g. start of grass after a sidewalk).

The size of the alpha texture does not need to match between layers, but using the same size in all layers allows it to be optimized by concatenating four alpha textures in one and reading them all at the same time in the pixel shader. Normally the alpha texture is edited by an artist, but it can be procedurally generated based on the height of the vertex. Lower regions can be covered by grass, while mountains are covered by snow (see figure 3.5).

The texture coordinates are generated per frame and only in the rendered vertices, avoiding unnecessary calculations and storage of large amounts of data per vertex (one texture coordinate per texture layer would be necessary). This process is illustrated in figure 4.11 by the regular replication of a sample texture (containing a simple shape - an arrow).

Every regular grid has a default texture coordinate system with its origin at the top left

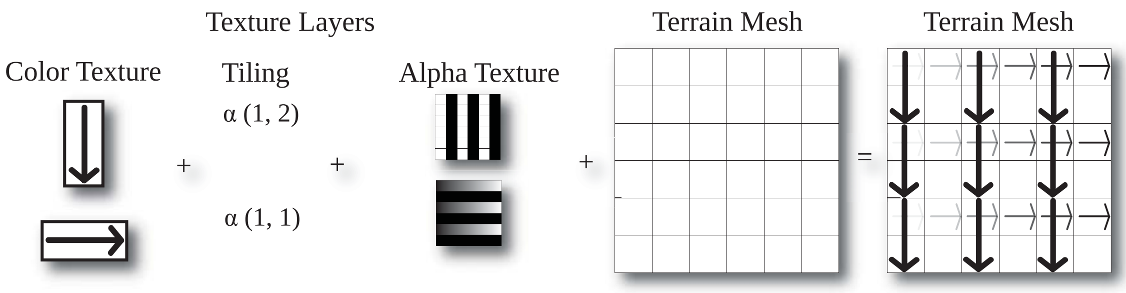


Figure 4.10: Application of texture layers to a terrain.

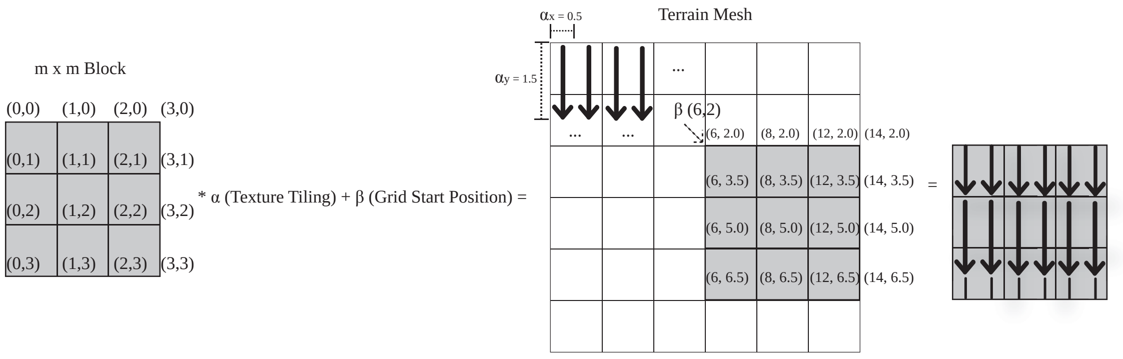


Figure 4.11: The texture layer used in the terrain mesh uses a color texture in the shape of a down arrow with tiling (0.5, 1.5). The layer covers all the terrain (denoted by the "...") but we are only interested in the shaded region.

most vertex as illustrated in the m x m Block from figure 4.11. Mapping a texture layer to the grid is a two steps process.

In the first step, the texture coordinates are scaled based on the 2-D texture tiling α associated to the texture layer, creating a space between vertices to fit the color texture. The example from figure 4.11 uses a tiling of (0.5, 1.5), creating space for two color textures between intervals of vertices in the x axis and 1.5 in the y axis. The mapping is illustrated by the arrow texture that repeats itself two times horizontally and 2/3 vertically between intervals of vertices.

The second step is adding a translation to the grid sending it to its position in the height field, based on the viewpoint. The amount of translation is denoted by β in figure 4.11, which is a parameter valid for all vertices of the grid, so it is calculated on the CPU and sent to the GPU.

Mapping the default coordinate system to the texture layer is a process that must be done for all texture layers, because they all have unique color textures and tilings. Therefore, for all texture layers there is a constant in the shader for the color texture, α and β, but only the latter changes between grid renderings.

4.1.5.1 Hole Layer

Sometimes game objects, like caves, may intersect the terrain, which requires a method to open holes in the mesh. Instead of modifying the terrain mesh to create holes or using transparent regions that would require ordered rendering, we use a feature of the raster operations that avoid the contribution of some pixels, leaving an unpainted region that resembles a hole in the mesh.

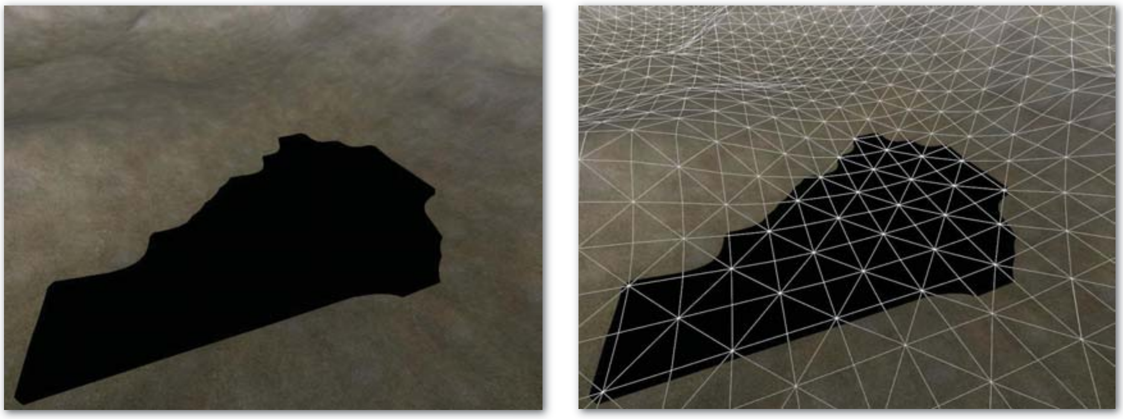


Figure 4.12: A hole in a terrain is defined as a region of the mesh where there is no output pixel in the pixel shader, leaving unpainted regions.

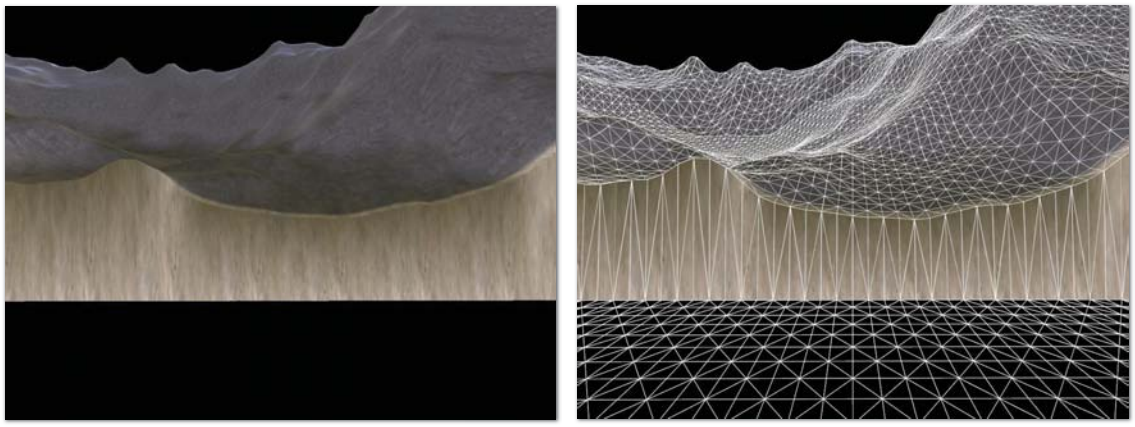


Figure 4.13: Visualization of the end of the world with a wall created by vertices outside the borders. The base of the wall is not rendered, because the hole layer kills the pixels that sample outside the alpha texture.

The hole layer is composed by an alpha texture that defines regions on the terrain mesh that must not be taking into account while rendering the terrain. In DirectX 8, holes are defined setting the render states Alpharef to 1 and Alphafunc to Equal. That way, any output pixel of the terrain that is mapped to a value different than 1 in the alpha texture will not be applied to the frame buffer. An example of creating a hole layer is illustrated in figure 4.12.

Another use for the hole layer is to avoid rendering beyond the height field limits. This can be done by setting any sample in the alpha texture (hole layer) outside the range ([0.0, 1.0], [0.0, 1.0]) to a value different from 1. The alpha texture may not have the exact same size of the height field. This results in precision issues that cause problems while killing pixels at the borders of the height field. We set the height for all vertices outside the borders to the minimum height possible, resulting in a wall like the one showed in figure 4.13. Note that this wall is not visualized, because the camera in a game rarely gets close enough to the limits of the terrain. Actually, the only vertices that are sent to the minimum height are those that belong to a regular grid with at least some vertex inside the borders; otherwise, the grid is not rendered.

4.1.6 Performance

The terrain from figure 4.14 runs at an average of 580 fps, using 6 texture layers. The height field size is 2048 x 2048 with distance between vertices of 8 meters, resulting in a terrain with 256 km x 256 km. Performance was measured on the following machine: AMD Atlhon 64 3500+, 2 GB ram, NVIDIA GeForce 7800 GTX. On a Pentium 4 2.8GHZ, 1 GB ram and a NVIDIA GeForce 6800 GT 256MB, the average fps was 300.

The performance is barely affected by the content of the height field or the color textures, the main bottleneck is the number of textures accessed in the pixel shader. For example, a terrain with 4 layers requires 4 color textures, 1 alpha texture (3 channels with alphas from the texture layer plus an alpha from the hole layer), and a normal map texture.

The performance of the presented technique is appropriated for games where other tasks besides the rendering are computed per frame and interactive frame rate is required. An example is the use of the technique in a commercial game called Deer Hunter Tournament 2008® that is heavily focused in the capabilities of the terrain, see figure 4.15.

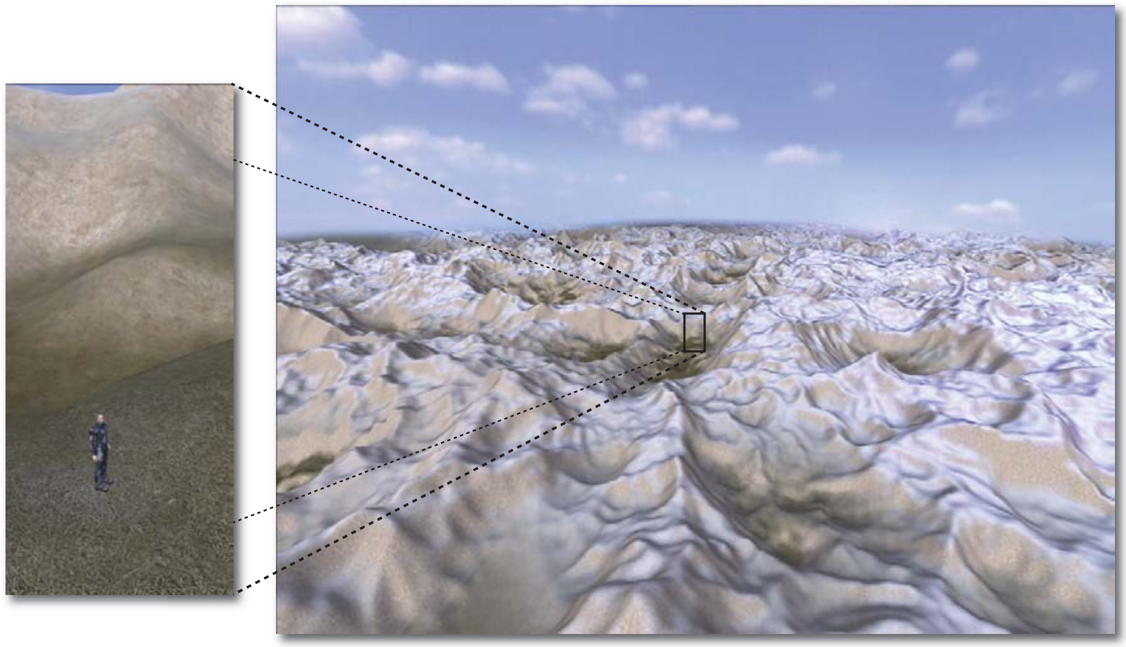


Figure 4.14: Technique demonstration, note the scale of the terrain compared to a person.

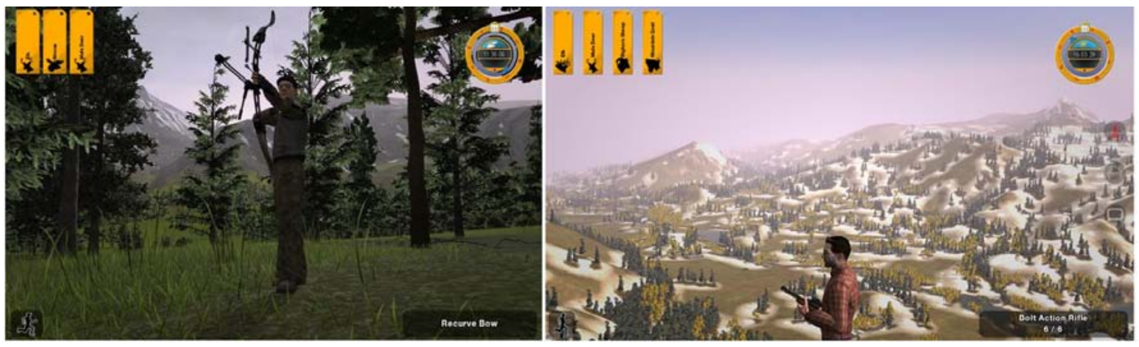


Figure 4.15: Screenshots from the game Deer Hunter Tournament®, as we can see it is heavily based on our terrain technique.

4.2 Displacement Mapping LOD using the Parametric Domain

One of the reasons that displacement mapping is used in real-time applications is its level-of-detail (LOD) capabilities. Usually for terrains the level of details of the reconstructed displacement mapping is defined according to the distance from the viewpoint to the surface of the base mesh. The closer a region of the base mesh is to the viewpoint the higher the resolution, resulting in an almost smooth distribution of resolution, as in previous section. However, in this section a different approach is introduced. Instead of only modulating the level of detail by the distance between the viewpoint and the surface, a contribution from the amount of detail defined in the displacement map and mapped to each region of the base mesh is considered. That way, avoiding a waste of resources in regions close to the viewpoint but without details associated from the displacement map.

The displacement mapping technique is defined by two elements: a coarser representation of a mesh (base mesh) and a map that encodes information to displace the geometric coordinates of mesh vertices (displacement map). There are several factors that might be taken into consideration when establishing a correspondence between mesh vertices and a displacement map. For instance, if the displacement function has a great change of values in a certain area, this will only be properly applied if there are enough mesh vertices to sample that area. Re-meshing or subdividing the base mesh to increase sampling can solve this problem, which can be done in a uniform or adaptive fashion. A uniform subdivision might be a suitable solution, depending on how the base mesh is created (SZIRMAY-KALOS; UMENHOFFER, 2008), (ASIRVATHAM; HOPPE, 2005), but often tends to create unnecessary vertices on regions where they are not needed. Alternatively, adaptive subdivision is more involved, but gives control on where subdivision should be performed (DYKE; REIMERS; SELAND, 2008), (MOULE; MCCOOL, 2002), (DOGGETT; HIRCHE, 2000), (AMOR et al., 2005).

These subdivision approaches usually do not guarantee that all information stored in a displacement map is well sampled by the base mesh. To overcome this sampling problem a new subdivision algorithm that defines a near one-to-one relationship between the number of texels of the displacement map and the vertices of the base mesh is presented in this section. The procedure is also adaptive, and therefore aims to obtain an optimal representation of the information contained in a displacement map, while avoiding unnecessary subdivision of uniform-subdivision approaches.

The main characteristics of the technique are summarized as follows:

1. **Adaptability**: The subdivision algorithm is guided by the displacement map information in such way to establish a near one-to-one mapping between mesh vertices and texels.

2. **GPU-friendly**: The algorithm presented here is fully parallel, and was implemented entirely in the geometry shader of the GPU.

3. **LOD rendering guided by displacement map information**: Details are preserved based on the frequency of the signal stored in the displacement map.

4. **Simplicity**: Although the method is based on recent parameterization techniques (which can be complicated), it is simple to implement.

4.2.1 Displacement Map

Displacement maps can be manually constructed or extracted automatically by measuring the distance between a base mesh and the original mesh. The purpose of the technique is to reduce the memory usage of an original mesh by replacing the original mesh with a base mesh and a displacement map. The latter has a smaller memory footprint than the original mesh and allows the reconstruction of the original mesh at runtime.

A possible approach to extract displacement information is described in Tewari et al (TEWARI et al., 2004), where the extraction is guided by the amount of detail in the original mesh. In regions with higher frequency details, they increase the size of the area allocated to encode the displacement map. Similarly, they reduce the area in lower frequency detail regions. Figure 4.16 illustrates how the same information encoded in a uniform subdivision can be represented in a more compact displacement map using the approach described above.

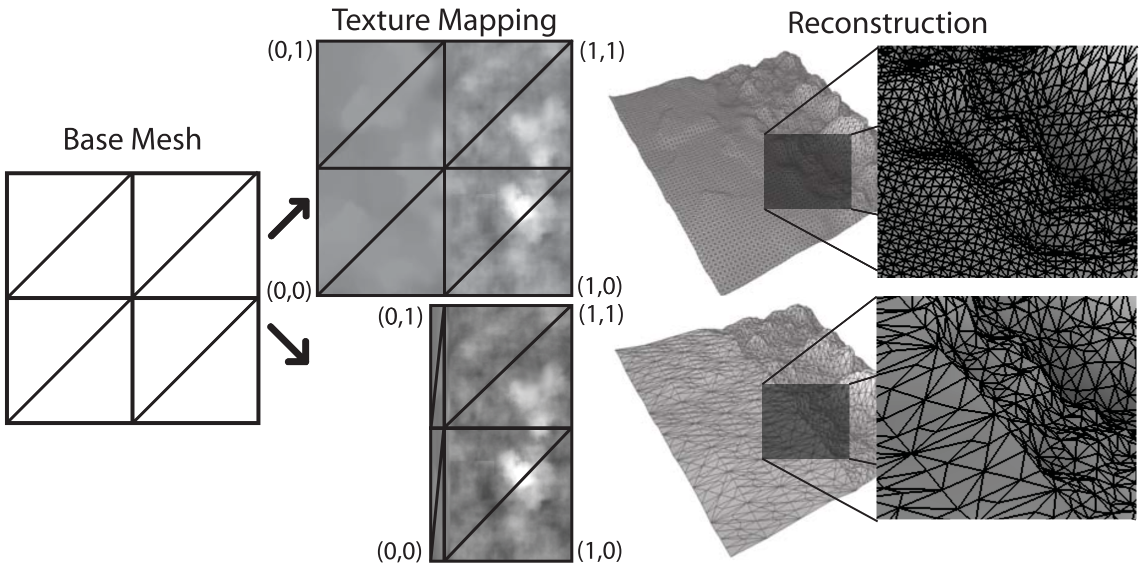


Figure 4.16: Two reconstructions of the original mesh from the same base mesh: Mapping between mesh vertices and the displacement is displayed in the Texture Mapping column. The one on top uses a uniform distribution of displacements, while the one in the bottom uses an adaptive distribution. The Reconstruction column shows the resulting meshes, with detail images highlighting the differences between these approaches.

An important point to observe is that the reconstruction step is always performed at the highest resolution needed, even if an adaptive distribution of samples in the displacement map were used. Therefore, computation is wasted in regions where a lower resolution reconstruction would be sufficient (which correspond to the smaller texels in the texture domain column of figure 4.16). This waste of computations can be reduced using the technique described as follow.

4.2.2 Adaptive Re-Meshing for Displacement Mapping

The reconstruction step is responsible for constructing a mesh based on the information encoded in the base mesh and the displacement map. We generate a mesh with more detail by subdividing the triangles of the base mesh. The detail encoded in each triangle of the base mesh is estimated by looking to the area of the displacement map reserved to them. Triangles mapped to larger areas of the displacement map encode more information than the triangles mapped to smaller areas, and thus should be subdivided accordingly (figure 4.17).

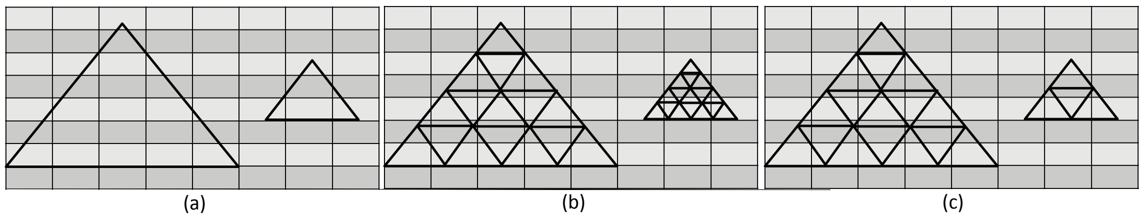


Figure 4.17: Relationship between the triangle area in texture space and the levels of subdivision: In (a), two triangles of the base mesh are mapped to the texture space represented by the underlying grid. In (b), we show a uniform subdivision for all triangles of the base mesh. Our approach, illustrated in (c), subdivides each triangle according to the number of texels mapped to it; avoiding computation waste.

The parameterization illustrated in Figure 4.16, between the base mesh and the texture, was generated manually. The parameterization was defined according to the amount of details desired for each region of the terrain. The left side of the terrain encodes regions with few details while the right side is composed by high detailed mountains, which require a large region of the displacement map. The parameterization could be defined automatically using the method presented by Teware et al (TEWARI et al., 2004), however, due to time limitations we choose to use a manual definition of the parameterization.

Central to this proposal is the computation of the triangle area, which can be expensive and result in different subdivision levels in adjacent triangles. We overcome this problem by looking at the length of triangle edges in texture space. This guarantees consistency between subdivisions, needs only local information, is easily parallelizable, and avoids T-junctions (inconsistent subdivisions between adjacent triangles, see Figure 4.7).

We illustrate this solution with the example in figure 4.18. Suppose we have a horizontal edge crossing from one side to the other of the displacement map. Since the number of texels crossed by this line is equal to the displacement map width, a simple equation to calculate the number of vertices that should be inserted in the edge is to divide the edge length, in object space, by his width in pixels over the texture space. Equation 4.1 represents this where α is the number of subdivisions to be applied to an edge, β is the width in pixels of the displacement map and ϕ is the edge length in texture space. This can be extended to edges with other orientations, and will result in adding more vertices to better sample long edges.

$$\alpha = \frac{1}{\beta}\phi \tag{4.1}$$

Once we know how many vertices to insert on each edge, based on the solution given by Equation 1 according to the edge length, we need a tessellation procedure to generate the connections inside the triangle. This procedure must be simple, fast, and connect the new vertices consistently. We use pre-defined subdivision patterns (figure 4.19), store them directly in GPU memory, and choose which one to apply according to the edges that have been subdivided and the number of vertices inserted on each edge. For example, if only two edges require insertion of vertices we choose the middle pattern, while the left pattern is used when all 3 edges of a triangles had been subdivided, and the right pattern

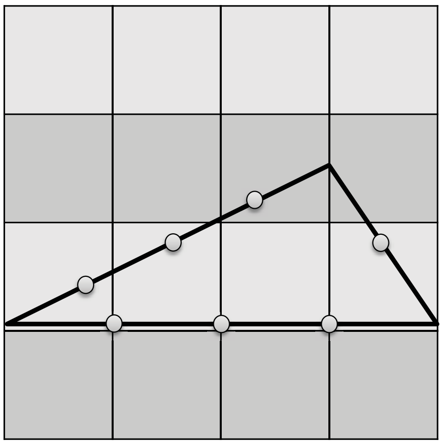


Figure 4.18: Subdivision of a triangle edge based on its length.

is used when only one edge is subdivided. The subdivision pattern is applied recursively until the length of the remaining edges reaches a threshold.

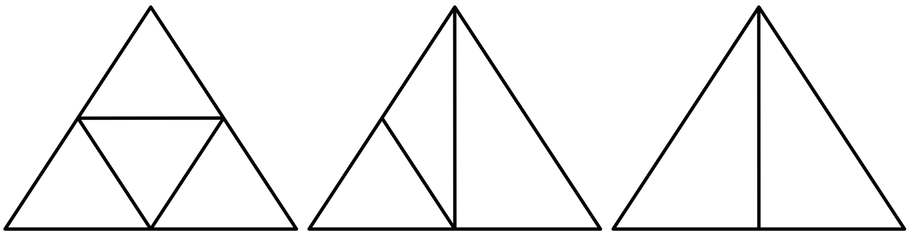


Figure 4.19: Tessellation patterns.

The use of subdivision patterns requires recursive passes when tessellating some triangles. This recursive approach explores the parallelism of current GPUs by distributing the computation through different geometry shader pipelines, thus avoiding the overload of a single pipeline with a given subdivision level. In fact, triangles with different subdivision levels are the whole point of adaptive re-meshing. Figure 4.20 illustrates recursive subdivision applied to adapting the base mesh for displacement mapping.

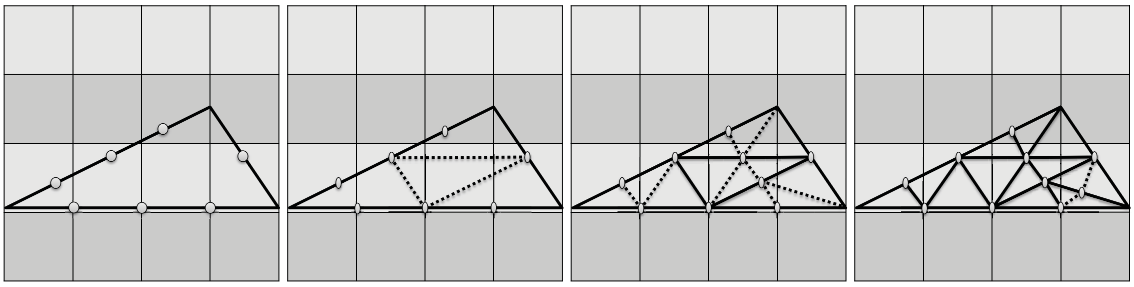


Figure 4.20: Recursive subdivision of a triangle: Process repeats until the edge length becomes smaller than a pre-defined threshold. Using the side of a texel as threshold, for example, we have a near one-to-one mapping between texels and new vertices.

4.2.3 Level of Detail

One of the main motivations for using displacement mapping is that the base mesh represents a coarser version of the original mesh. This allows the use of the base mesh even without a displacement map. The use of the base mesh alone is useful, for instance, when the model is far away from the camera and the details inserted by the displacement map would not be perceived. However, if we use the base mesh when the camera is far away, then we need a level-of-detail approach that smoothly adds detail to the base mesh when the model gets closer to the camera.

We modulate the amount of detail by changing the level of subdivision. By decreasing the number of subdivisions in the base mesh, we decrease the amount of displacements applied to the model, which results in a less detailed representation of the original mesh. The maximum allowed edge length in texture space (the side of a texel, for instance) is used to modulate the level of subdivision. Decreasing the maximum allowed length represents coarser subdivisions, while bigger edge lengths represent more levels of subdivision.

The distance between the model and the camera is used to modulate the maximum edge length. This provides a level-of-detail technique that increases details near the camera and decreases details as the model moves away from the camera. It is important to observe that while the model is moving away from the camera, regions of the model mapped to small portions of the displacement map converge to the base mesh faster than other regions of the model that map to large areas of the displacement map.

The resulting level-of-detail technique can smoothly insert and remove details, while avoiding drastic changes in the model. Furthermore, not only we can smoothly insert details by increasing the subdivision level, but also we can smoothly displace the vertices using geomorph. Figure 4.21 illustrates the same dataset from figure 4.16 now using our LOD scheme.

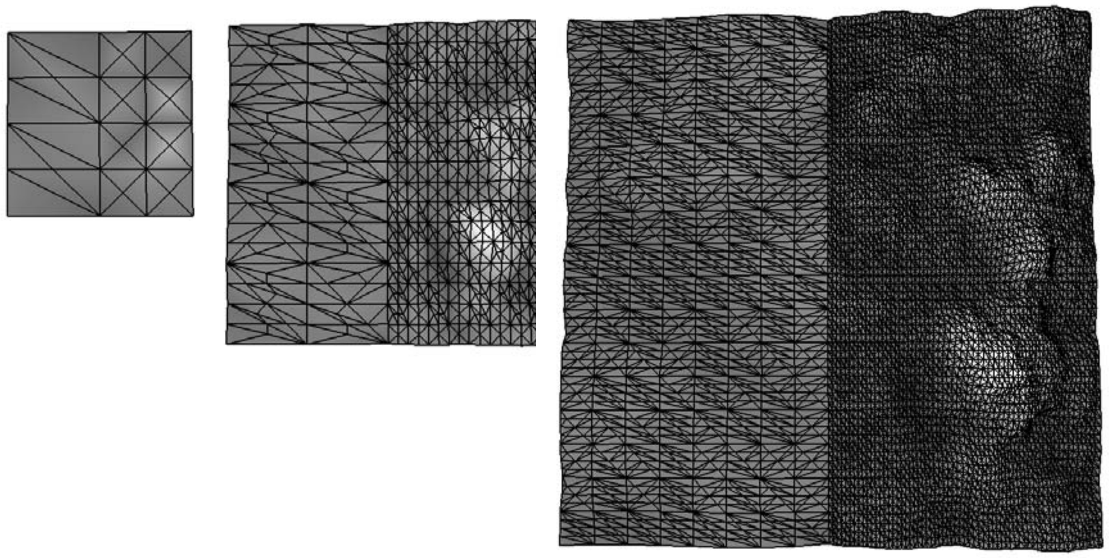


Figure 4.21: Reconstruction of the original mesh using our LOD scheme: From left to right, the mesh moves closer to the camera (which is perpendicular to the ground). Observe that we have a finer tessellation in areas with high frequency detail.

Instead of computing the LOD according to the distance between a camera and a reference point in the base mesh, we also locally modulate the LOD. For this purpose, we use the middle point of each edge, which defines the subdivision resolution in terms of the proximity to the surface, which is suitable for rendering large surfaces like a terrain. Figure 4.22 illustrates the local modulation of the LOD using the same terrain of figure 4.21. Observe that closer to the camera we have more resolution, but the high frequency regions (mountains) always have enough resolution to represent the encoded information, even when the camera is far away, as depicted by the left image.

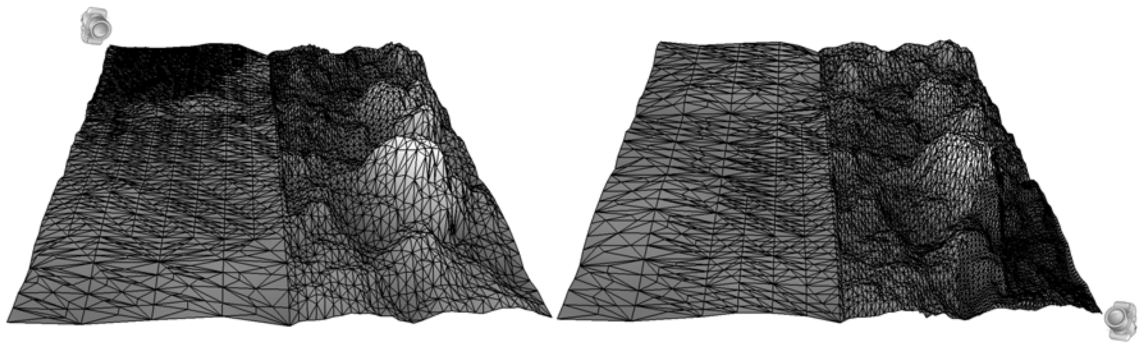


Figure 4.22: Distribution of resolution according to the distance of the edges to the camera (depicted with an icon in the figure): Distance to the camera is not the only factor that affects base-mesh resolution. High frequency regions remain with enough resolution, even when they are away from the camera (left).

4.2.4 Results

We measured performance results in a computer with an AMD Athlon(tm) 64 Processor 3700+, 2GB RAM and an NVIDIA GeForce 8600 GTS. The first mesh of figure 4.22 has 34,422 vertices and 11,474 triangles, and is rendered in 112 frames-per-second (FPS). The second mesh has 50,451 vertices and 16,817 triangles running at 108 FPS. Those times include the re-meshing and the displacement of the base mesh at each frame. Note that this is a stress test, because the base mesh has only 8 triangles and the output mesh, on the second mesh, has 2102 times more triangles.

4.3 Conclusion

In this chapter we presented two approaches to improve the efficiency and visual characteristics of the displacement mapping technique. Both have being developed based on the characteristics of the parameterization between the displacement map and the base mesh.

The original clipmap work (ASIRVATHAM; HOPPE, 2005) was developed for rendering of large terrain datasets using the GPU. However, some essential features to its practical application, particularly in games, have being neglected. Our modifications, presented on section 4.1, improved the clipmap technique to a point where its practical application on games is possible. The results show high frame rates with a flexible technique not only in the media creation side but also on the range of GPUs that are capable of performing it.

The other improvement to the displacement mapping technique was presented on section 4.2, and is related to the geometry reconstruction of the displacement map. The novel multi-resolution displacement-mapping technique presented in this work has many desirable features, especially for games: multi-resolution, GPU-friendly, LOD-enabled rendering, simple to implement, and capable of interactive results.

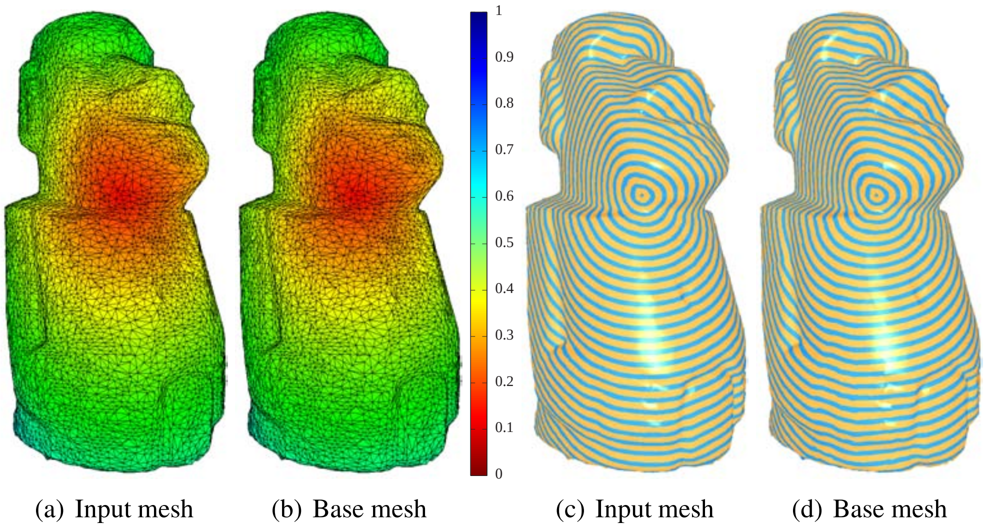
5 GEODESIC DISTANCE COMPUTATION ON THE PARAMETRIC DOMAIN

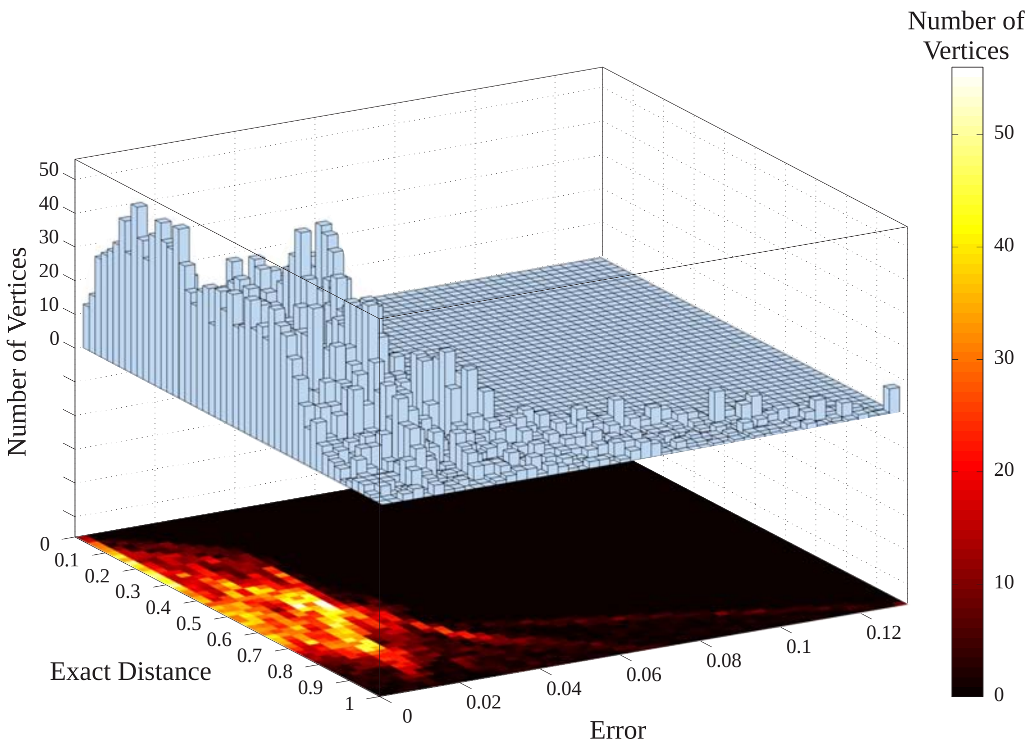
Computing distances on triangular meshes is required in several fields, including procedural textures (WALTER; FOURNIER; MENEVAUX, 2001), surface labeling (CIPRIANO; GLEICHER, 2008), remeshing (PEYRé; COHEN, 2005), parameterization (PEYRé; COHEN, 2005) and segmentation (FUNKHOUSER et al., 2004). Distance evaluation for interactive applications has been limited to poor approximations, due to the computational cost of exact solutions. Most interactive applications that require on-the-fly distance calculation, such as path planning (see chapter 6), use variations of the Dijkstra algorithm. However, resulting paths are not smooth, since they are constrained to pass through the edges of the triangulation. Recent efforts on computing exact solutions more efficiently are described in Surazhsky et al. (SURAZHSKY et al., 2005), Liu et al. (LIU; ZHOU; HU, 2007) and Balasubramanian et al. (BALASUBRAMANIAN; POLIMENI; SCHWARTZ, 2009). However, due to the greedy nature of these algorithms, their computational time and large memory requirements make them too expensive for interactive applications.

In this chapter is presented an approximate algorithm that produces distance computation with minimal precision loss at an order of magnitude faster than an exact algorithm. The method relies on computing distance in the parametric domain. However, instead of using manually created parameterizations, an automatic parameterization is used to parameterize quasi-developable regions, which allow geodesic distance computation to be performed over a simplified version of the input mesh. Distance computation is performed over simplified charts, thus reducing the computational cost and memory footprint. In figure 5.1, we compare distance fields computed on the input mesh (items (a) and (c)) against results computed over the base mesh using our approach (items (b) and (d)). Colormaps and isolines in both cases are mapped and displayed over the input mesh, even for the one computed on the base mesh. Figure 5.1(e) presents an error histogram of the approximation error. While the distance field using the input mesh required 13 seconds to compute, the method presented on the following sections took only 0.4 seconds and the memory footprint was reduced 19 times.

The main contributions of our approach can be summarized as follows:

- **Developability as a simplification criteria**: The input mesh is divided into quasi-developable segments that are unfolded into charts and then simplified (remeshed) with no additional distortion. Distance computation algorithms over simplified charts lead to better performance with an acceptable loss of accuracy, caused by distortion in mesh unfolding.


(a) Input mesh (b) Base mesh (c) Input mesh (d) Base mesh



(e) Base mesh error histogram

Figure 5.1: Distance field computed on the input mesh and on the simplified base mesh; all results mapped on the input mesh, the captions denote where each result was computed. Distance fields are represented by the color-coded, normalized distance between each point, on the surface, to the source point (distance 0) - (a) and (b). Isolines extracted from (a) and (b) are shown in (c) and (d), respectively. Approximation error is presented in a 3-D histogram (e): difference between distances computed on each mesh. The distances computed on the input mesh are exact. We can control the accuracy of our method.

- **Accuracy control**: Most meshes are not developable or quasi-developable, which causes distortion in the mapping between the input mesh and the base mesh, thus reducing accuracy. Distortion error is controlled by dividing the input mesh into regions where the unfolding distortion does not pass a given threshold.

- **Improved performance**: Memory footprint is reduced, which leads to smaller computational cost for calculating distances.

- **Generic solution**: Algorithms for distance computation can be adapted to use our simplified mesh, such as Dijkstra(DIJKSTRA, 1959), Sethian (SETHIAN, 1999), Surazhsky et al. (SURAZHSKY et al., 2005) and Martinez et al. (MOREIRA; VELHO; CARVALHO, 2005).

5.1 Related Work

In this section we review the related work on distance computation on triangular meshes - which has a good survey in (BALASUBRAMANIAN; POLIMENI; SCHWARTZ, 2009) - and quasi-developable mesh segmentation.

5.1.1 Distance Computation Algorithms

The well-known Dijkstra algorithm (DIJKSTRA, 1959), and its variations, have been extensively used to compute distances on meshes. Instead of computing geodesic paths, Disjkstra-like algorithms compute paths over the edges of the mesh, resulting, in most cases, in poor approximations. Another algorithm that gives approximate results, with attractive performance, is the Fast Marching (FM) (SETHIAN, 1999). FM uses wave propagations on the mesh surface to generate a distance field. As in most distance field-based algorithms, the shortest path on the mesh surface connecting two points can be found by gradient descent on the distance field.

Martinez et al. (MOREIRA; VELHO; CARVALHO, 2005) computes shortest paths between two points, but its computational cost is higher than Dijkstra and FM. They compute the exact solution based on an initial guess, computed using FM. Performance depends on the accuracy of the initial guess, which motivated the use of FM instead of the faster Dijkstra algorithm. Mitchell et al. (MITCHELL; MOUNT; PAPADIMITRIOU, 1987) proposal was extended by Surazhsky et al. (SURAZHSKY et al., 2005) to calculate exact distances from a given point on the mesh surface to any other point on the mesh. They showed that their method has similar performance, however, improved precision if compared to FM. Recently, Balasubramanian et al. (BALASUBRAMANIAN; POLIMENI; SCHWARTZ, 2009) proposed an algorithm based on the iterative growth of triangle strips. The goal is to find the triangle strip that has the shortest path connecting two points on the mesh surface. According to the authors, their method has approximately half of the computational cost of (SURAZHSKY et al., 2005).

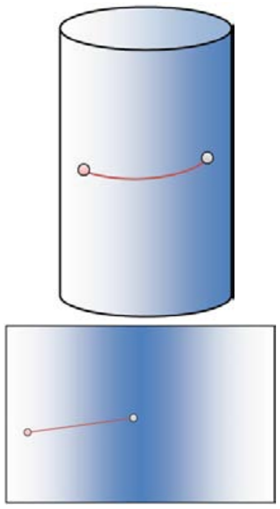
5.1.2 Quasi-Developable Segmentation

The segmentation of non-developable meshes into quasi-developable charts has attracted considerable attention, due to the practical applications and difficulty of the task. The work of Julius et al. (JULIUS; KRAEVOY; SHEFFER, 2005) has accomplished quasi-developable mesh segmentation by computing the angle between each triangle normal and a common axis and grouping triangles with constant angle variation. Alternatively, Yamaguchi et al. (YAMAUCHI et al., 2005) introduced a segmentation based on Gauss area, where the segmentation can be accomplished by grouping regions of the mesh with zero Gauss area. Recent work of Wang (WANG, 2008) obtained quasi-developable charts by merging charts generated by the method of Cohen-Steiner et al. (COHEN-STEINER; ALLIEZ; DESBRUN, 2004). The idea is to identify several small quasi-developable charts and test if two charts can be merged without generating too much distortion.

5.2 Triangular Mesh Simplification for on-Surface Distance Computation

5.2.1 Distance Computation on Developable Surfaces

Developable surfaces (such as cylinders and conical surfaces) are surfaces with zero Gaussian curvature everywhere. Such surfaces admit isometric parameterization to the plane (see section 3.2). The distance-preserving property of such parameterizations is the characteristic that motivates our approach. Consider, for instance, the computation of the shortest path between two points on a cylinder. The shortest path curve over the cylinder becomes a line when the cylinder is unfolded. The resulting path is a line, because the distance variation is constant inside unfolded developable surfaces. However, the shortest path connecting both points is not the only possible path connecting them. Finding the shortest path among all possible paths is the goal of most distance computation algorithms.



A common way to compute distances on the surface of triangular meshes is using wave propagation. To determine the shortest path, between two points, wave propagation from an origin point is started. Consider the example show in figure 5.2, where color represents the distance to each point inside the wave to the origin point. When the wave collides with itself, the front closer to the origin pushes the other front until both fronts have the same distance to the origin. The process continues until the entire surface is covered. Finding the shortest path is a matter of descending the gradient of the distance field from destination to origin.

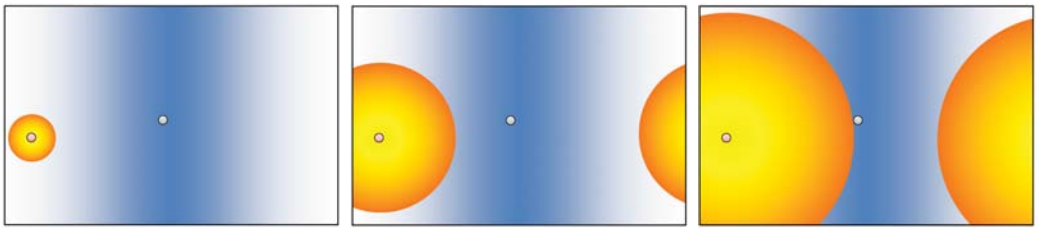


Figure 5.2: Wave propagation on developable meshes.

Wave propagation, on triangular meshes, is performed by walking over the triangles in steps equal to the triangles adjacent to the wave front (see figure 5.3), thus the number of triangles greatly affects the performance of this process. If the distance varies constantly and isotropically over the unfolded triangular mesh surface, the number of triangles does not alter the resulting distance field (i. e., the size of the propagation step does not change the result). In this sense, for computing distances on a developable mesh, we can find a simplified triangulation (base mesh) that does not alter the borders of the unfolded mesh (input mesh) and compute distances on this simplified triangulation, thus optimizing the process.

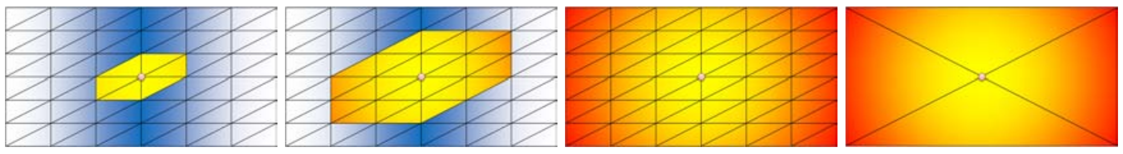

(a) 1 Step (b) 2 Steps (c) 6 Steps (d) 1 Step

Figure 5.3: Wave propagation on two triangular meshes with same area. (a)-(c) represents the propagation on an unfolded mesh (input mesh), while (d) represents the propagation on a simplified version of the mesh (base mesh). Resulting distance fields are the same but more efficiently computed on the simpler mesh (base mesh).

Figure 5.4 illustrates a path computed on the base mesh and mapped to the input mesh. We can map a point from one mesh into the other because the base mesh has the same area and borders of the unfolded input mesh. The dashed rectangle, in figure 5.4, illustrates the border of the input mesh, which is the same on the base mesh.

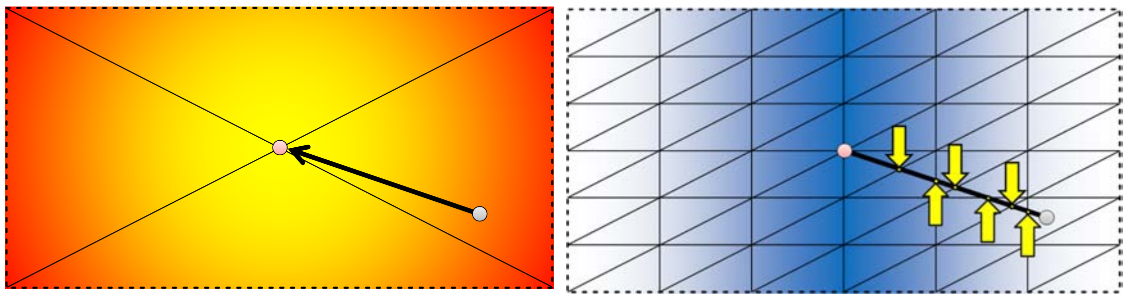
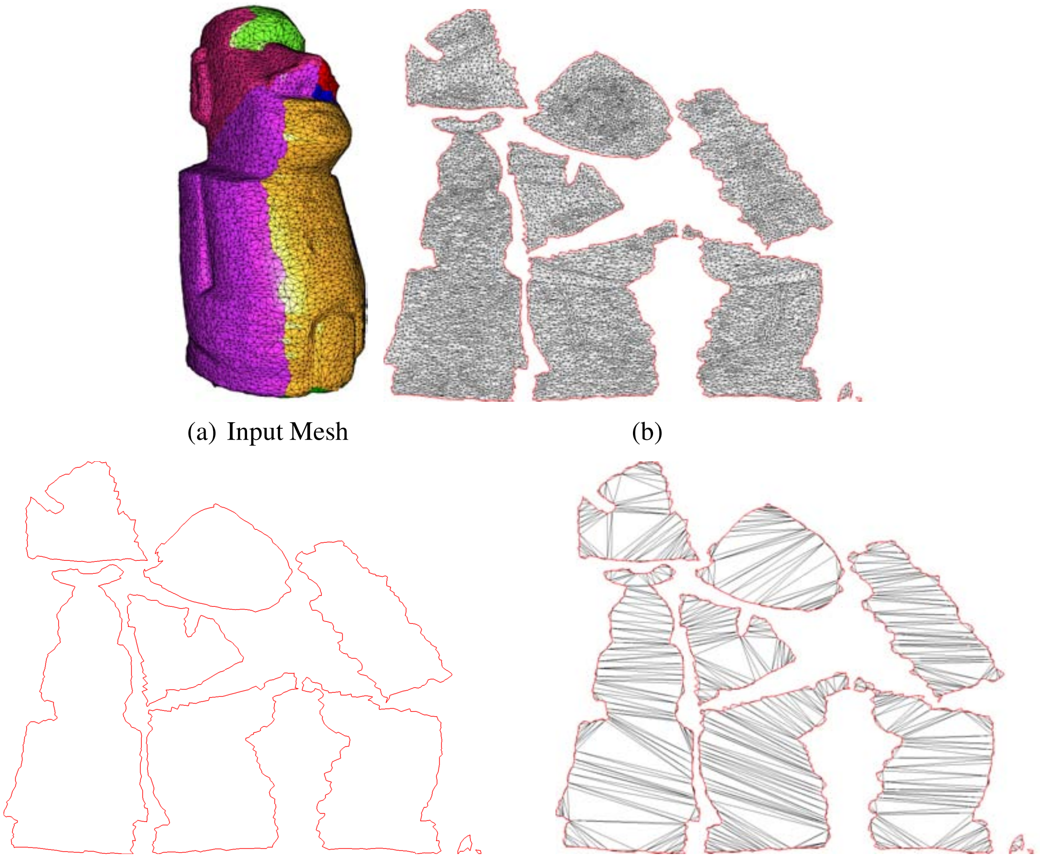

(a) Base Mesh + Distance Field (b) Unfolded Input Mesh

Figure 5.4: Shortest path. Path computed on the base mesh is mapped to the input mesh, and reconstructed by computing intersections against the triangles (yellow arrows).

This method applied on developable meshes computes distances without loss of precision or generality. However, on non-developable meshes, a global unfolding may result in high amounts of distortion, which leads to loss of accuracy on distance computation. To overcome this problem, we first segment the input mesh in charts that can be unfolded with limited and controlled distortion. Next, we simplify each chart independently, and the collection of all simplified charts results in our base mesh.

5.2.2 Quasi-Developable Mesh Segmentation

The mesh segmentation technique we use is based on the work of Wang (WANG, 2008), which merges the charts generated using the Cohen-Steiner et al. (COHEN-STEINER; ALLIEZ; DESBRUN, 2004) algorithm. This choice was based on the features of the method: distortion control and reduced number of charts. We can control the distortion by defining the initial number of charts generated with (COHEN-STEINER; ALLIEZ; DESBRUN, 2004). The merging of charts, as presented by Wang (WANG,


(a) Input Mesh (b)

(c) (d) Base Mesh

Figure 5.5: Base mesh construction pipeline. Given an input mesh, the first step is to identify quasi-developable regions on the mesh (a); followed by the unfolding of those regions (b). Chart simplification removes interior vertices without changing the boundary (simple polygon) (c). Finally, charts are re-triangulated (d). In this example, the input mesh was formed by 20,180 triangles, while the base mesh is composed by 1,148 triangles. Notice the reduction in number of triangles from (b) to (d).

2008), reduces the final number of charts, controlled by a pre-defined threshold on the maximum distortion allowed. Figure 5.5(a) illustrates a mesh segmentation. The parameterization method used is the one described in Sheffer et al. (SHEFFER et al., 2005), and an example of charts that we obtain can be seen on figure 5.5(b). Although the method of Sheffer et al. (SHEFFER et al., 2005) results on parameterizations which low distortions it isn't the state-of-the-art according to Ligang Liu (LIU et al., 2008), which appears to be the best method for quasi-developable mesh parameterization in terms of performance and quality. Due to the limitations of time we used the work of Sheffer et al. (SHEFFER et al., 2005) which provides an implementation of the method.

5.2.3 Chart Simplification

After the mesh segmentation and parameterization we have the regions of the input mesh that are quasi-developable. We simplify each chart, first by removing all interior vertices. The result is a simple polygon: the chart border. Figure 5.5(c) depicts the region of the charts that are not modified during the simplification process. We do not modify the vertices and edges in the borders to enforce the connection between neighboring charts, allowing us to navigate between charts. In other words, we have a bijective mapping between the borders of neighboring charts.

We assume that the distance variation inside the red polygons in figure 5.5 is constant in any direction, as a result of the quasi-developable mesh segmentation. In order to use existing distance computation algorithms we need a triangular mesh. In that sense, we re-triangulate the interior of the polygons. Figure 5.5(d) illustrates the charts re-triangulation, showing a considerable reduction in the number of triangles to the original shown in 5.5(b).

The re-triangulation method used is based on the constrained delaunay triangulations presented by Chew (CHEW, 1987), however any other polygon triangulation method can be used. Our choose was intended to minimize numerical problems resulting from handling almost degenerated triangles which may appear on simple triangulation techniques.

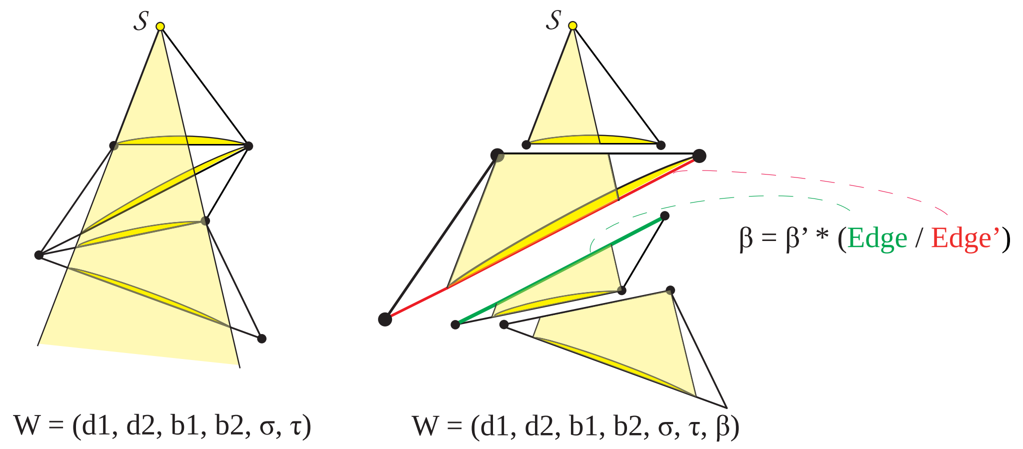
After each chart has been simplified, we have the base mesh. We use this flat mesh as input for distance computation algorithms. Note that the charts are logically connected by shared edges (red polygons). However, as a result of the independent chart parameterization shared edges do not share the same length; in other words, the connection between charts is implicit and not explicit. This logical connection and the difference in length between shared edges guide the necessary modifications on distance computation algorithms to apply them to the base mesh. The next section describes how one can adapt a distance computation algorithm designed for usual triangular meshes.

5.2.4 Adapting a Distance Computation Algorithm

For calculating distances on the base mesh, we adapted the algorithm from Surazhsky et al. (SURAZHSKY et al., 2005), which calculates exact geodesic distances on triangular meshes. We generated our results with this algorithm to avoid contaminating the approximated distances obtained through our approach with any other source of inaccuracies. We used the robust implementation from Liu et al. (LIU; ZHOU; HU, 2007), with the adaptation described in this subsection.

In the algorithm of Surazhsky et al., the source point, from which geodesic distances are calculated, casts $windows$ (figure 5.6(a)) that are successively propagated across adjacent triangles. $Windows$ represent regions of the mesh for which a shortest path candidate to the source point is already defined. Therefore, $windows$ shapes must not be changed when they are propagated across triangles; however, in the base mesh, a $window$ may be propagated across the frontier between two charts, where the shared edges (red polygons on figure 5.5) usually do not have the same length on both charts, due to the parameterization non-uniform distortion. For handling this peculiarity, with minimum changes on the original algorithm, we added to the $window$ representation a scale factor β, which changes when the $window$ is propagated across the frontier. The original parameters for the $window$ representation are depicted in figure 5.6(a).

The scale factor β is initially set to one for every created $window$. When a $window$ is propagated across the frontier between two charts, its width may change, as shown in figure 5.6(b) due to the difference in edge length between both sides of the frontier. In this transition, the shape of the $window$ can be kept unchanged by scaling $b1$, $b2$, $d1$, $d2$ and σ by the same scale factor that changed the $window$ width. This scale factor is the ratio between the length of the frontier edges on each chart (see figure 5.6(b)). The scale factor is also accumulated in β at successive $window$ propagations. This way, the scale of the $window$ may be removed by dividing $b1, b2, d1, d2$ and σ by β. Removing the $window$ scaling is necessary when the actual length of the shortest path candidate encoded by the



(a) Window as in Surazhsky et al. (b) Window adapted to include a scale factor

Figure 5.6: Modified $window$ representation. The scale factor β is the ratio between the frontier edges crossed by the $window$. In this case, each triangle in (b) is a chart.

$window$ must be known. Finding shortest paths and resolving $windows$ superpositions are examples of such situation.

Note that this modification is related to the distance computation algorithm presented by Surazhsky et al. (SURAZHSKY et al., 2005). Another solution may be required for a different distance computation algorithm.

5.3 Results

The algorithm was tested on an Intel® Core^{TM}2 1.86GHz PC with 4GB RAM. We used, as segmentation method, the approach presented by Wang (WANG, 2008), while the following unfolding of the segments was done using the method of Sheffer et al. (SHEFFER et al., 2005). Both mesh segmentation and parameterization can be accomplished with several different methods, see (SHAMIR, 2008) for a survey about mesh segmentation and (SHEFFER; PRAUN; ROSE, 2006) for a survey about mesh parameterization. The constraints which must be considered during the construction of the base mesh are the parameterization distortion and the number of charts. Distortion controls the accuracy of the computed distances, and the number of charts controls the mesh simplification. We use the metric presented by Sander et al. (SANDER et al., 2001) to measure the parametric distortion.

Table 5.1 presents the results for the models depicted in figures 5.1, 5.7 and 5.10. Figure 5.1 illustrates the distance field computed for the Moai model, using the segmentation from figure 5.5(a), parameterization from 5.5(b) and base mesh from 5.5(d). Distance computation in the base mesh compared to using the original input mesh leads to a speedup of 28 times, with an average error of 0.017. Using the input mesh, the method of Surazhsky et al. (SURAZHSKY et al., 2005) requires 299000 $windows$, while on the base mesh it requires only 13096 - a reduction of 22 times. Using the original $window$ description ($b1, b2, d1, d2, \sigma, \tau$) and double precision, we need 11679 KB to store the $windows$ necessary to compute the distance field represented in figure 5.1(a) and (c). Including the scale factor β, into the $window$ representation, and computing the distances on the base mesh, we need 613 KB to store the $windows$ necessary to generate the distance field of figure 5.1(b) and 5.1(c). Figure 5.1(e) presents the error histogram of the distance field computed using our method.

The distance field and isolines for the models Ramesses and Dancer (genus non-zero meshes) are presented in figure 5.7 and figure 5.8. Two results are presented for the Dancer, each using different base mesh resolutions (see Table 5.1). Note that in our mesh simplification scheme higher distortion thresholds imply lower base mesh resolutions, which allow us to indirectly control distance computation accuracy and performance.

Model	Mesh Faces		L^∞		L^2		Error		Dynamic Mem. (KB)		Time (sec)	
	Input	Base	Avg.	Max	Avg.	Max	Avg.	Max	Input	Base	Input	Base
Moai	20,180	1,148	1.021	7.084	1.008	5.009	0.017	0.131	11,679	613	13	0.4
Dancer L.1	50,004	5,288	1.006	1.627	1.002	1.609	0.006	0.02	39,505	5,117	39	4
Dancer L.2	50,004	4,038	1.016	2.258	1.011	2.076	0.009	0.033	39,505	3,724	39	2.8
Kilimangiaro	100,000	587	1.011	3.022	1.005	2.137	0.002	0.01	92,870	93	70	0.03
Ramesses	386,500	16,852	1.006	4.944	1.003	4.648	0.006	0.035	319,536	24,920	263	23

Table 5.1: Performance and accuracy of computing distances on the base mesh (our method) and computing on the input mesh. The relative error is given by the difference between the distance computed on the base mesh and on the input mesh, normalized by the farthest point found on both meshes; $|D_{BaseMesh}(M(V_n)) - D_{InputMesh}(V_n)|/Max(Max(D_{InputMesh}(V), Max(D_{BaseMesh}(M(V))))$, where M maps a point in 3-D (input mesh) to 2-D (base mesh), $V_n \in V$ and V is the set of vertices of the input mesh.

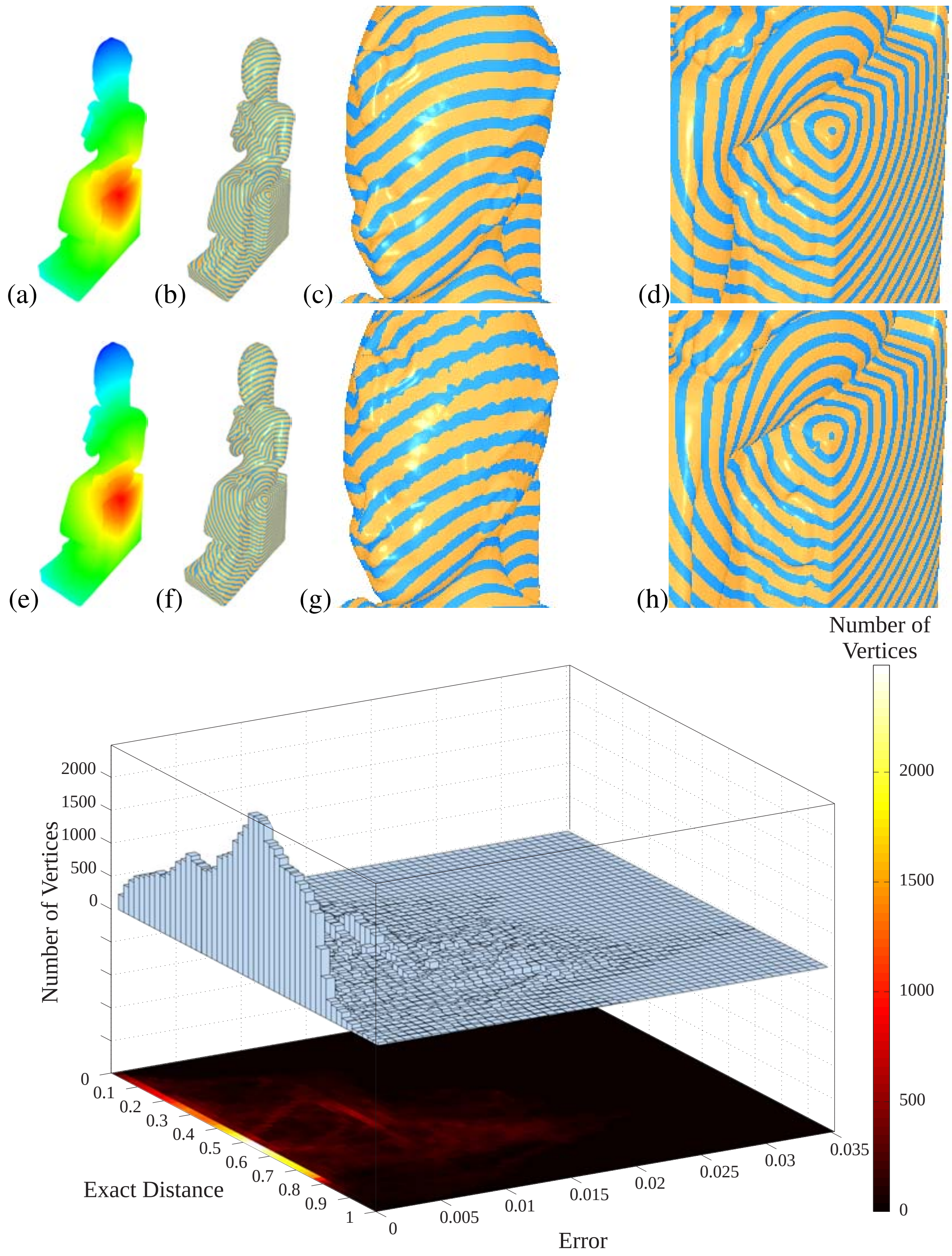
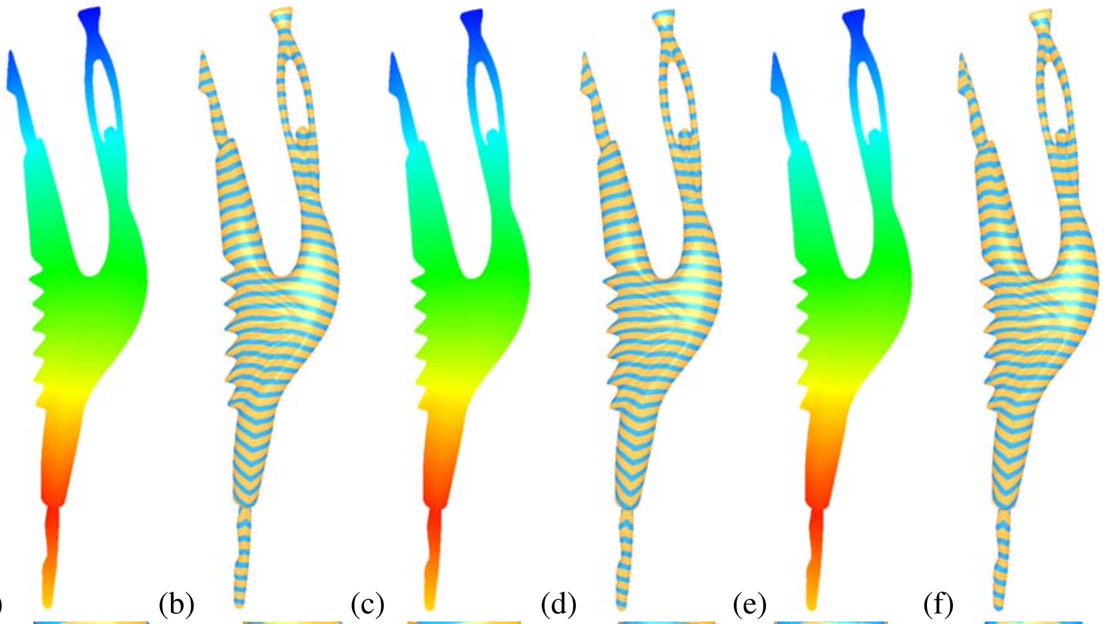
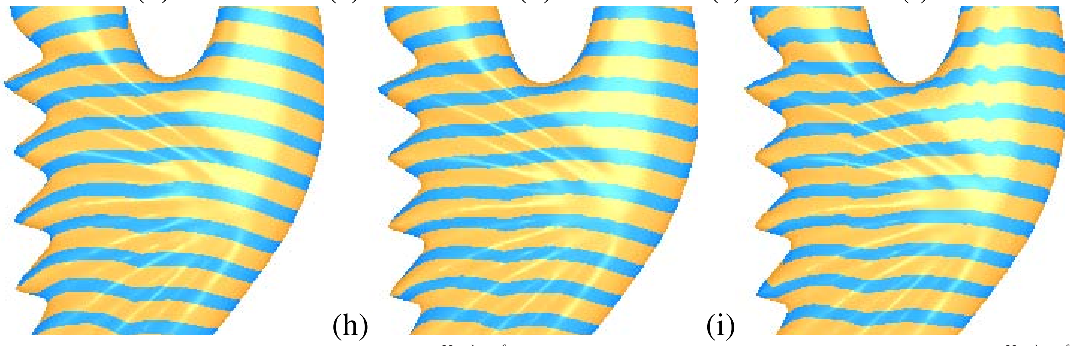
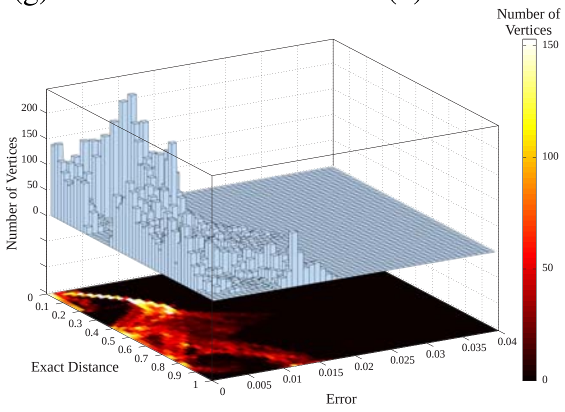


Figure 5.7: Distance field, isolines and zoomed details for the Ramesses model computed on the input mesh (a-d) and on the base mesh (e-h). Following we have the error histogram which shows small distance error for nearly all vertices (observe near zero concentration on the Error axis). The average L^∞ and L^2 distortion is of 1.006 and 1.003, respectively. Due to the low average distortion, error is also low. In highly distorted regions (max L^∞ of 4.944) some error is introduced, which is not localized (as in the parametrization distortion) since the error is propagated.

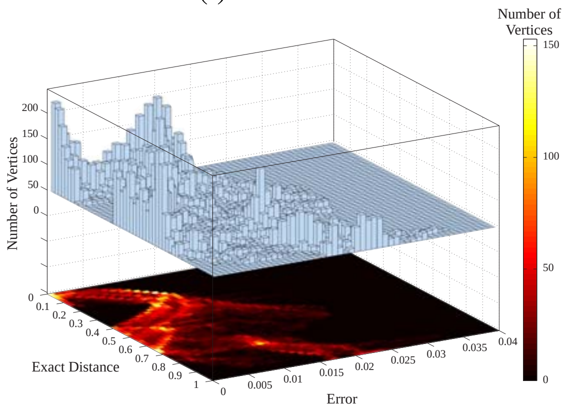
Figure 5.9 illustrates how our method can be used on meshes that are far from developable. This mesh was created applying random noise to a planar mesh, where the amplitude of the noise is low at the lower-left corner of the mesh and increases towards


(a) (b) (c) (d) (e) (f)


(g) (h) (i)



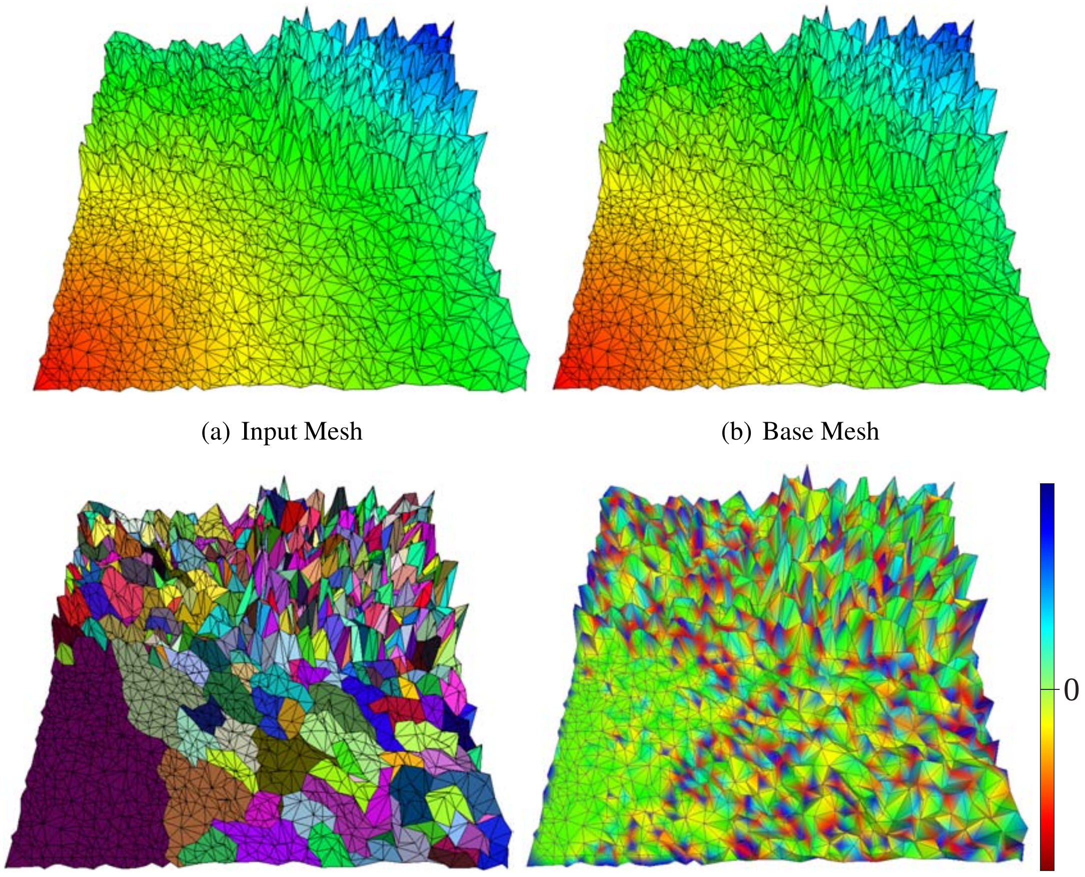
(a) Dancer Mesh Level 1



(b) Dancer Mesh Level 2

Figure 5.8: This figure show the results for the Dancer: input mesh (a-b) and from base meshes with different resolutions (c-d and e-f). Zoomed versions appear in the middle row: input mesh (g), base mesh resolution 1 (h) and resolution 2 (i), note the difference in accuracy. The last row shows the error histograms for the base meshes. Both histograms have the same upper bound error, but only level 2 reaches this limit, illustrating the tradeoff between accuracy and mesh simplification.

the upper-right corner. Note the self-intersections in the region with high noise amplitude. The Gaussian curvature of that noisy mesh computed as presented in Meyer et al. (MEYER et al., 2002) ranges from -4640.19 to 20994.02, while the plane, which is the surface where the input mesh is parameterized, has Gaussian curvature equal to 0. The



(a) Input Mesh

(b) Base Mesh

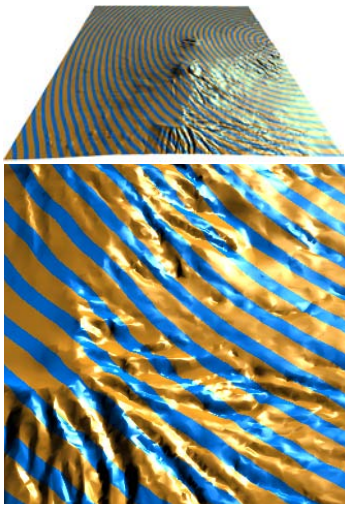
(c) Segmented Input Mesh

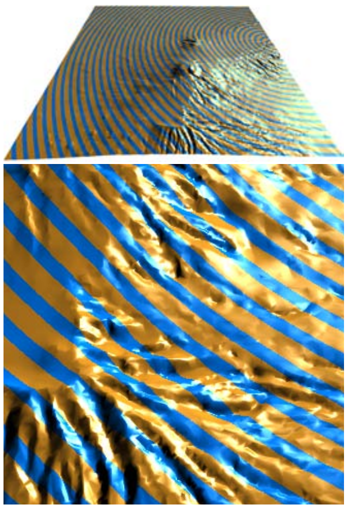
(d) Gaussian Curvature

Figure 5.9: Distance computation on meshes with high Gaussian curvatures. Our method can handle meshes which are far from developable, however, the amount of simplification becomes limited. Note, on item (c), the concentration of charts in the regions where the Gaussian curvature frequency is higher, see (d). However, in low frequency regions the charts are larger.

colormap in figure 5.9(d) illustrates the Gaussian curvature of the input mesh. To declutter the figure we clamped the curvature range, removing the 5% higher and lower vertices to generate the colormap. Note that in regions with higher peaks the number of charts is larger, compared to flatter regions (colored in green shades - closer to 0 Gaussian curvature in the color scale). Regions with higher peaks are farther away from a plane (and thus not as developable), and require more and smaller charts, which reduces the amount of simplification. Those small charts, after the chart re-triangulation, remain with the same number of triangles, as in the input mesh. However, in most cases, meshes are not formed only by regions with high Gaussian curvature, in that sense, most cases present some room for simplification. Therefore, we reached a maximum error of 0.05 and a speedup of 1.65 times by computing distances in a base mesh resulting from a 43% simplification of the input mesh.

Figure 5.10 illustrates the isolines computed on a terrain model. This particular model is an example of surface where distance computation is frequently required. For example, in games where performance has priority over precision, our method can be used to compute paths for agents with high performance as we present in chapter 6. Figure 5.11 illustrates the effect of introducing noise in the input mesh before simplification.


(a) Input Mesh


(b) Base Mesh

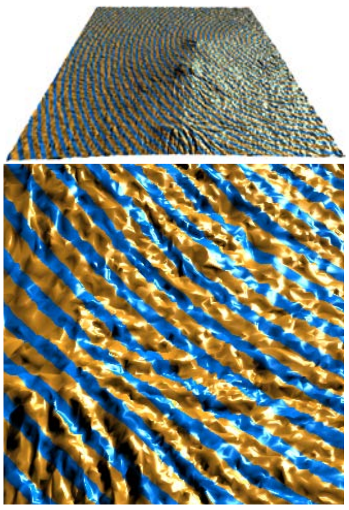
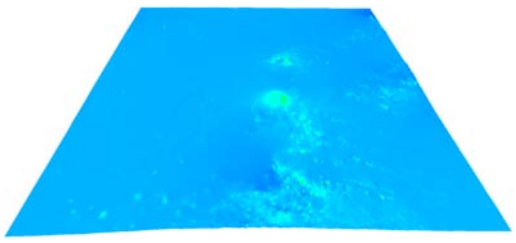
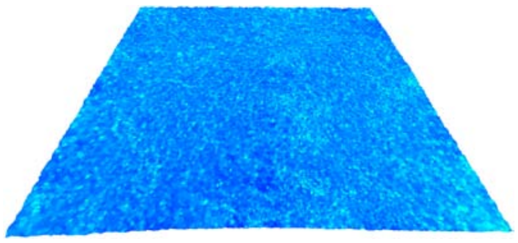
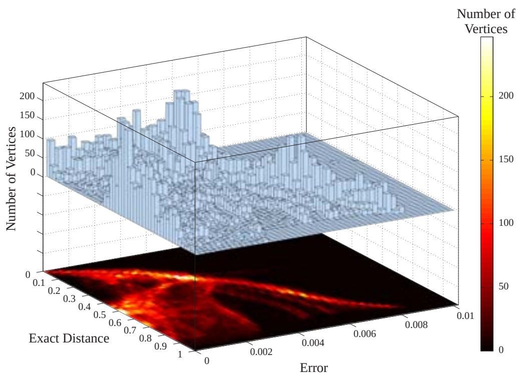

(c) Base Mesh (Input Mesh + Noise)

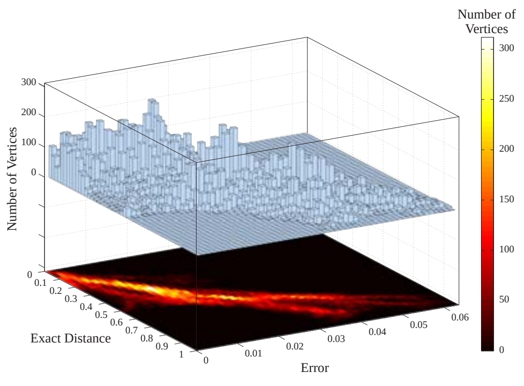
Figure 5.10: Kilimangiaro terrain. Item (a) represents isolines of the distance field computed on the input mesh. Item (b) illustrates the isolines computed on the base mesh. This base mesh was constructed from a global parameterization of the input mesh. While the base mesh, from item (c), was too constructed from a global parameterization, noise was introduced on the input mesh (only the result computed on the base mesh is illustrated). See figure 5.11 for the histograms of item (b) and (c).


(a) L^2 from figure 5.10(b)


(b) L^2 from figure 5.10(c)



(c) Kilimangiaro Mesh

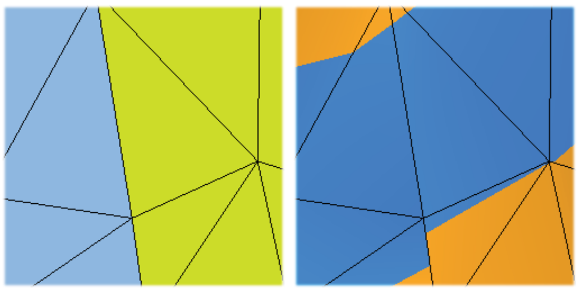


(d) Kilimangiaro Mesh + Noise

Figure 5.11: Relation between parametric distortion and error on the distance computation. (a) and (b) depict the L^2 distortion present in the parameterizations that constructed the base meshes used on (b) and (c) from figure 5.10, respectively. The uniform distribution of distortion, due to the introduction of noise, results in an almost linear increase of error, see (d). This is the result of the wave front continually passing over distortion. As a result, the error, created due the distortion, is continuously added to the distance that the wave front encode. Note that the upper bound error in (c) and (d) is different.

5.4 Discussion

Our approach is based on quasi-developable mesh segmentation and parameterization. In that sense, meshes with high Gaussian curvature require small charts to avoid distortion, limiting the amount of simplification. While parameterizing the mesh in charts reduces global distortion, this approach tends to concentrate distortion in the boundary between charts. As a result, chart boundaries can present discontinuities, see figure 5.12. Note the discontinuity in the isoline that is crossing a frontier between charts. This may not be suitable for applications where the distance field must be smooth. A possible solution is to use a global smooth parameterization, as in the work of Khodakovsky et al. (KHODAKOVSKY; LITKE; SCHRöDER, 2003). The use of such approach would allow us to remove the scale factor used to transmit the $windows$ between charts frontiers. This would further reduce the memory footprint, of our method, and would allow the direct use of distance computation algorithms developed for triangular meshes, without modifications such the one presented in section 5.2.4. However, such global parameterizations tend to have a larger total distortion, compared to independent charts.



(a) Chart boundary (b) Isoline

Figure 5.12: Chart boundary between two charts (a) and isoline visualization (b). Note the discontinuity in (b) due to edges having different sizes in each chart.

A limitation of our method is related to the base mesh construction, which cannot be done in real-time, limiting its use to static meshes. Considering memory footprint, our method requires more static memory, due to requiring two meshes (input mesh and base mesh) and the mapping between them. However, only the static memory footprint is increased, which is not the bottleneck of state-of-the-art distance computation algorithms. On the other hand, the same parameterization, used in the mapping between both meshes, can be used to map textures on the input mesh. Mapping textures is required in several applications, and reusing the parameterization for this task can be explored, for example, to improve the performance of procedural texture techniques.

5.5 Conclusion and Future Work

In this chapter we presented a mesh simplification method for accelerating on-surface distance computation. The results show that our method can improve the performance of distance computation algorithms with controllable accuracy. The method is particularly useful for real-time applications, such as games, where several agents must navigate on the surface of meshes, as we show on the next chapter. A possible modification to further improve the efficiency of our method is to compute the distances on the GPU.

6 MULTI-AGENT NAVIGATION GUIDED BY DISTANCES COMPUTED ON THE PARAMETRIC DOMAIN

In this chapter we explore the gains of performance of computing on-surface distances in the parametric version of the mesh. Due to the gains of performance of such approach we present in this chapter a new path planning algorithm intended to navigate agents in the surface of arbitrary meshes.

Path planning for multiple agents is an open research topic that has been increasingly explored in the literature, especially in the game development community. Issues like navigation and interaction between multiple agents, as well as scalability with respect to scene size and agent number, have been recently addressed to provide increased realism in dynamic and complex scenes.

Recent research efforts can be divided into two main approaches: those based on autonomous interaction between agents and those dealing with directing whole crowds to a given destination. Algorithms that address both approaches are, as a rule, computationally expensive and unsuitable for real-time simulations.

Furthermore, most solutions to date limit the movement of agents to the Euclidean plane. This restriction is unsuitable for scenes where agents need to move over curved surfaces. In this chapter, we propose a method for navigation of multiple individuals on non-planar surfaces, allowing multiple destinations and collision avoidance between agents. Our technique imposes no limitations on the nature of the underlying mesh, which can have any genus (see Figure 6.1), as well as arbitrary shape. To the best of our knowledge, the work (TORCHELSEN et al., 2010) is the first to propose a completely general approach to non-planar path planning.

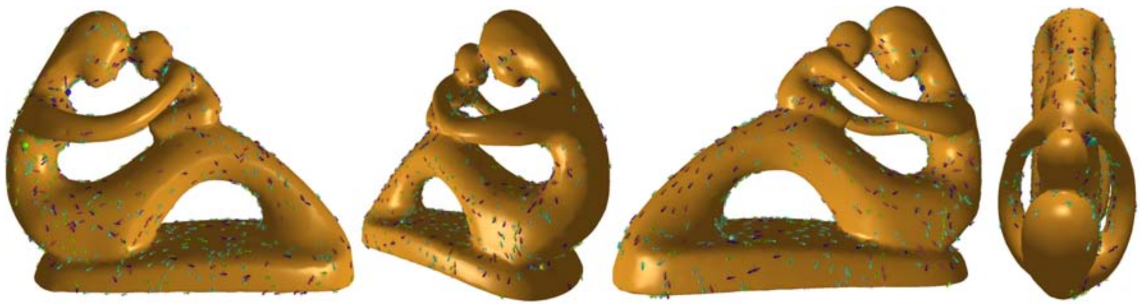


Figure 6.1: Four different views of a scene where agents navigate on the surface of a complex triangular mesh. Agents are color-coded by their different objectives. The system supports path planning of multiple agents on non-planar surfaces, and imposes no limitation on the domain mesh, such as this mesh with more than one genus.

The main contributions presented in this chapter can be summarized as follows:

- A method for multi-agent navigation on non-planar triangular meshes using geodesics
- A multi-resolution technique that allows new destinations for agents to be defined at runtime, while maintaining interactive frame rates
- An efficient GPU-based approach for collision avoidance between agents with minimal synchronization, which maximizes performance by exploring the massively parallel architecture of modern GPUs
- A hybrid CPU/GPU path planning pipeline that explores the best qualities of each processor

This chapter is structured as follows. Section 6.1 discuss previous work relevant to the contributions presented in this chapter. We address the issue of global navigation in Section 6.2, presenting our approach for distance computation and our parallel pipeline, used to obtain interactive frame rates. In Section 6.3, we focus on the local movement of agents and collision avoidance. Section 6.4 presents results obtained with our method, as well as the implementation and performance data. Finally, in Sections 6.5 and 6.6, we discuss limitations, conclusions and avenues for future work.

6.1 Related Work

Path planning of multiple agents can be summarized in terms of two main concerns: global and local navigation. Global navigation simply deals with the path an agent must follow to its destination, while local navigation ensures that agents will avoid obstacles and other agents, while minimizing deviation from the original path.

The proposed path planning method computes geodesic distances over a mesh to define the global path, while using a novel local navigation method. This new approach is a parallel algorithm with very few synchronization points, which makes it particularly suitable for GPU implementation. In what follows, we discuss previous work concerning global and local navigation.

6.1.1 Global Navigation: Distance Computation

Agents moving in a virtual environment typically have to navigate towards a set of objectives. The global path that each agent follows to its goal can be defined by several techniques. The agents can walk directly towards their targets (BERG; LIN; MANOCHA, 2008) or they can move following precomputed roadmaps, as is the case of van den Berg et al. (BERG et al., 2008b). Other approaches for defining the global path of an agent also avoid sharp turns due to obstacles along the way. Thus, the path chosen is a compromise between the shortest possible path and a solution that smoothly dodges static obstacles. This approach is discussed in the work of Silveira et al. (SILVEIRA; PRESTES; NEDEL, 2008).

These techniques, however, are restricted to planar domains. More general approaches that can deal with non-planar surfaces usually involve computing distances on the surface of a triangular mesh. This can be done using several algorithms, such as Dijkstra's algorithm (DIJKSTRA, 1959) (and its variations), Fast Marching (FM) Methods (SETHIAN, 1999) and Window Propagation (SURAZHSKY et al., 2005). Dijkstra's algorithm and the

FM methods give approximations of the geodesic distance on a mesh, while the Window Propagation technique gives exact results.

Because of its relatively low cost most applications that require interactivity use Dijkstra's algorithm to compute approximations to geodesics. However, our distance computation method is more accurate than Dijkstra's methods and is still capable of maintaining interactive frame rates. Furthermore, Moreira et al. (MOREIRA; VELHO; CARVALHO, 2005) demonstrate that using Dijkstra's algorithm to approximate geodesic paths can lead to poor results when compared to FM methods.

6.1.2 Local Navigation: Collision Avoidance

Collision between agents becomes an important concern when agents are moving in a sufficiently crowded environment, as demonstrated by Thalmann and Musse (THALMANN; MUSSE, 2007). Local navigation techniques exist to avoid collision while minimizing deviation from the globally optimal path. Local navigation methods have been proposed in the literature, as in van den Berg et al. (BERG; LIN; MANOCHA, 2008) and Guy et al. (GUY et al., 2009), who extended the concept of Velocity Objects (FIORINI; SHILLER, 1998) from robotics to scenarios densely populated with agents. Van den Berg et al. assume similar collision avoidance reasoning for every agent, which results in collision-free and oscillation-free motion.

Fulgenzi et al. (FULGENZI; SPALANZANI; LAUGIER, 2007) apply the concept of Probabilistic Velocity Obstacles (PVO) (KLUGE; PRASSLER, 2004) to a dynamic occupancy grid, to estimate the probability of collision when there is uncertainty in the position, shape and velocity of the obstacles. Lamarche and Donikian (LAMARCHE; DONIKIAN, 2004) propose a method for real-time navigation on complex scenarios using topological structuring on the geometry of the environment. Finally, Paris et al. (PARIS; PETTRE; DONIKIAN, 2007) present a predictive agent-based approach where the agents react to possible collisions by sampling their surroundings and extrapolating their trajectories.

Some authors also focus on crowd behavior. Yeh et al. (YEH et al., 2008) introduce the concept of Composite Agents to model complex interaction among individuals. Their technique is based on emergent behavior among agents, such as aggression and priority. Foudil and Noureddine (FOUDIL; NOUREDDINE, 2006) address collision avoidance by introducing priority rules and social behavior to the individuals. Other authors, such as Silveira et al. (SILVEIRA; PRESTES; NEDEL, 2008), propose hybrid algorithms that combine global paths and collision avoidance in their techniques.

It is important to note that, in most prior work, local navigation relies heavily on distance measurements between agents and obstacles. This makes such techniques unsuitable for use in non-planar surfaces, where computing distances is typically an expensive operation. Our work addresses this problem and enables a fast and reliable local navigation technique over non-planar surfaces.

In the next section, we discuss how the method performs global navigation. We demonstrate how to compute a hierarchical distance field over the domain mesh, and how the path planning pipeline employs separate, parallel threads to construct this distance field efficiently while maintaining interactive frame-rates.

6.2 Global Navigation

In our path planning technique, each agent moves toward one of n different objectives on the mesh. Each of these objectives is defined by a point p_i over the surface $\mathcal{S}$ of the domain, and for each p_i we construct a distance field $f_i : \mathcal{S} \rightarrow \mathbb{R}$. As can be seen in Figure 6.2, given any point on the mesh, the field f_i determines the shortest distance to the objective p_i. The gradient ∇f_i of a distance field gives the direction of the shortest path (also known as the *direction of steepest descent*) from any point on the domain to its corresponding objective. Therefore it is this direction that agents follow in our method. Using distance field gradients as the direction of the agents' movement has the advantage that any single individual deviating from the optimal route to avoid obstacles can easily return to the shortest path once it has moved around the obstruction. Also, this technique causes fewer collisions than roadmaps, since roadmaps tend to group agents close together on the same path.

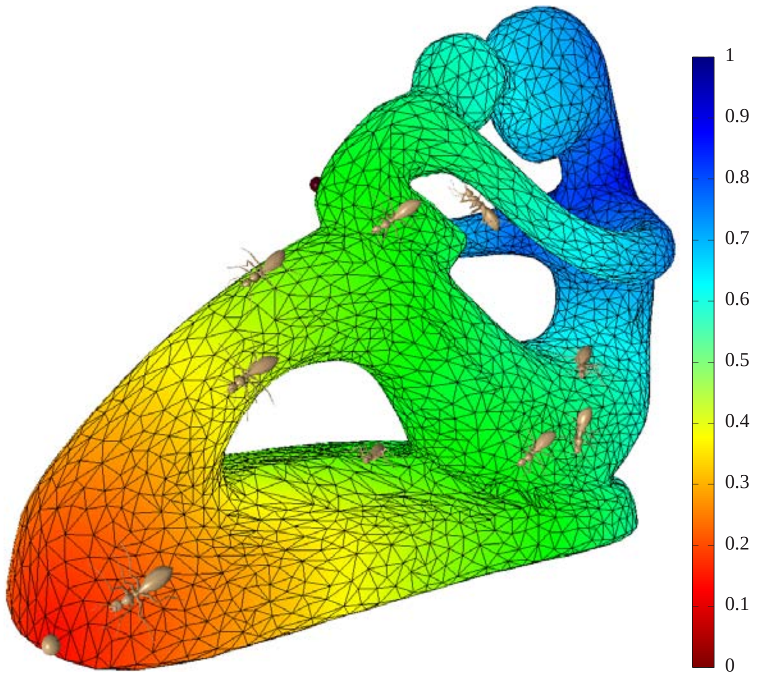


Figure 6.2: **Distance Field** Each objective p_i defines a distance field f_i over the domain mesh. In this figure, the surface is colored by distance, and all values are normalized to the interval $[0, 1]$. Agents follow paths of steepest descent along ∇f_i to reach their destination.

Most path planning techniques to date have used Dijkstra's algorithm or its variations to compute distance fields over a mesh due to its low computational overhead. However, distance fields created in this way are very rough approximations of the correct, geodesic distance field, and their paths of steepest descent are not very smooth. Our distance computation method (chapter 5) addresses the computational cost of algorithms that compute distance fields over irregular triangular meshes.

Irregular triangular meshes are the most common form of surface representation in computer graphics, due to the irregular structure being optimal for multi-resolution; for example, we can have more vertices in regions that need more information, but avoid wasting memory and computation in flat regions. We can see an example of an irregular mesh in figure 6.3.

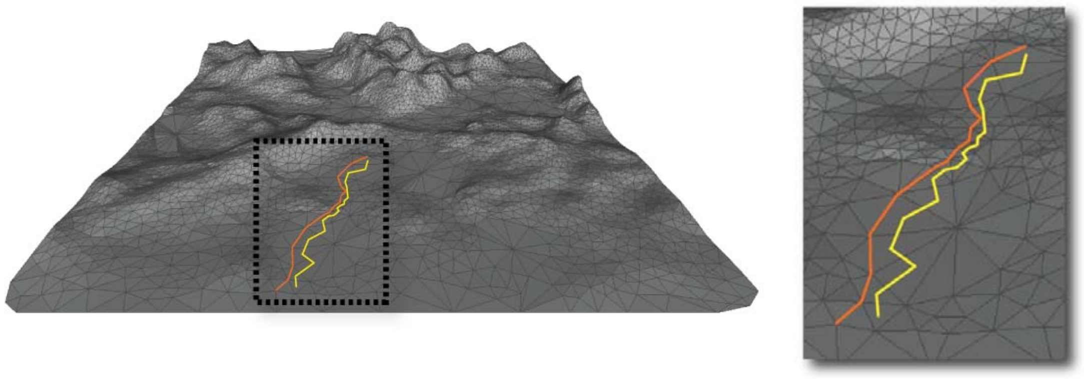

(a) (b)

Figure 6.3: Illustration of an irregular mesh and two paths computed over it. The orange and yellow paths were computed on a search space defined as an irregular graph. The graph of the orange path was formed by the vertices of the mesh as nodes, while the graph of the yellow path was formed by triangles as nodes.

Most graphic applications use irregular meshes as the method to represent surfaces; however, irregular meshes are not optimized for surface navigation, mainly due to the irregular neighborhood between vertices and its irregular size of triangles.

The A* (HART; NILSSON; RAPHAEL, 1968) is one of the most well know search algorithms used when the search space is an irregular mesh. The A* is based on the Dijkstra's graph-search algorithm, which is intended to identify the shortest path between two nodes of a graph (*single-source shortest problem*). The mesh connectivity between vertices or triangles can be interpreted as a graph, and a path on the surface can then be computed as a graph-search problem. Figure 6.3 illustrates two paths computed using a graph search algorithm.

The orange path in figure 6.3 was computed in a search space defined as a graph formed by vertices' connectivity. This means we can only navigate on mesh edges. We can cite two problems with this type of search space: irregular neighborhood and irregular edge lengths. The irregular neighborhood implies that we cannot predict the neighborhood of vertices and we must explicitly store the vertex connections. In addition, the irregular length of edges makes it hard to define a minimal geodesic path. Furthermore, it is difficult to define a smooth path on a multi-resolution mesh as the one from figure 6.3.

The yellow path from figure 6.3 illustrates the use of another search space, which is also defined as a graph. In this case, we use the mesh triangles as nodes, and the connectivity is defined by triangles sharing an edge. The main issue here is the irregular size of the mesh triangles; in general, the path is defined by connecting the center of the triangles, which, again, turns a minimal geodesic path difficult to establish.

Eventhough our distance computation algorithm can reach high performances on distance field computation in general, simultaneously computing many different distance fields over a mesh is still a costly operation. Since this is a requirement for a system that supports multiple goals, we have designed a hybrid CPU/GPU pipeline that can sustain interactive frame-rates by scheduling different threads to gradually compute distance fields in increasing resolutions.

6.2.1 Hierarchical Computation of Geodesics

The first step in our path planning pipeline involves computing the distance fields over the mesh on the CPU, as can be seen in Figure 6.4. We use the CPU for this task due to the nature of distance field computation. Distance field algorithms typically employ a variable advancing front over the mesh, which is hard to implement on classical GPU programming models.

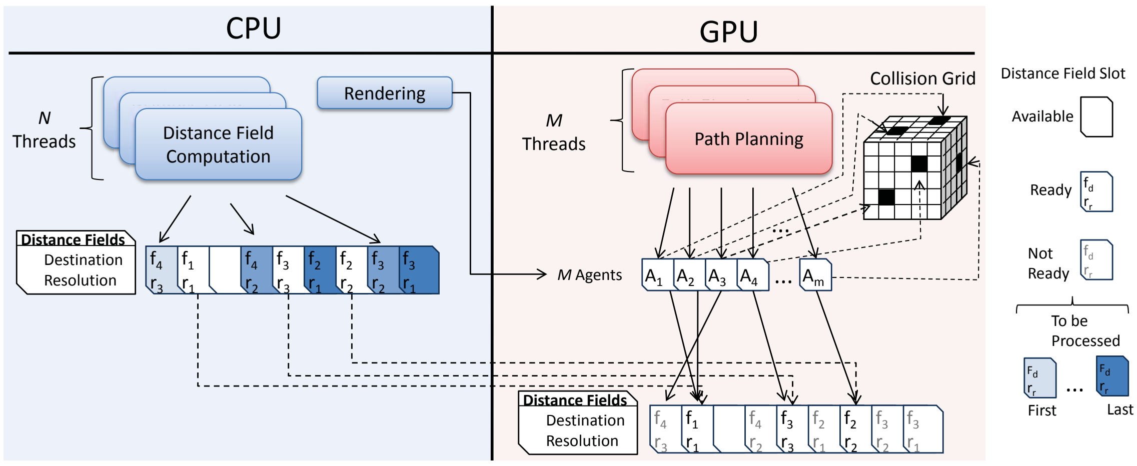


Figure 6.4: **Path planning pipeline.** Path planning system architecture. On the left, the CPU side is responsible for computing the hierarchical geodesic distance fields. The GPU, on the other hand, reads the available distance fields to guide the agents, and implements our grid-based collision avoidance technique. Note that since our system computes distance field of progressively finer resolution, there may be different versions of the same field in the pipeline.

Naturally, an agent bound to objective p_i requires that ∇f_i, and hence f_i, be computed before it can start moving. In order to maximize interactivity, we compute a hierarchy of distance fields for each objective. This computation proceeds from a coarse version of the mesh all the way to the original domain, and is done in parallel threads that do not block the simulation. Thus, agents can start moving almost immediately, and will follow a progressively more accurate path as higher-resolution distance fields become available. Whenever an agent receives a new goal p_i it immediately sends a message to the system requesting ∇f_i. If no version of ∇f_i is available, this request is added to a queue of distance fields that need to be computed. This queue is always traversed from lowest to highest resolution, to ensure that agents can start moving as soon as possible.

Distance fields and their gradients are transferred to the GPU as soon as they are completed by the CPU threads, and the system tells agents to update their distance fields every time higher resolutions become available for use. Due to a limitation on the architecture of current GPUs, we must preallocate memory for all versions of the distance fields at start-up. This is due to the fact that current GPUs do not allow dynamic memory allocation during code execution. The system limit to the number of distance fields, usually, is the number of agents because the worst case, in terms of computational cost, is each agent walking toward a unique goal which requires a distance field for each agent. However, the limitation on the amount of memory, usually, is the limiting factor on the number of distance fields available per instant.

Although this system efficiently provides smooth shortest-paths for agents to traverse, it does not in any way avoid collisions between agents and obstacles. To address this, we propose a simple grid-based obstacle detection and avoidance approach, which we describe in the next section.

6.3 Local Navigation

Even though steepest descent lines along a single distance field will never cross, when dealing with multiple objectives in a single scene it is entirely plausible that the path of different agents will intersect, potentially leading to collision between them. We propose a simple grid-based method that deals with collision avoidance while minimizing deviation from the optimal path. In the following sections, we describe the collision avoidance grid and the procedure to determine a new moving direction for agents that must round obstacles.

6.3.1 Collision Grid

Agents need to know, with some anticipation, whether there are obstacles in their intended path, so they may change their direction and avoid collisions. Our method uses a static 3-D grid to store the current position of each agent in the scene. Each cell in the grid holds a binary flag indicating whether or not there is an agent in that cell. We chose this simple structure because it minimizes the need for synchronization between parallel threads controlling multiple agents, turning it ideal for GPU implementation. Bleiweiss (BLEIWEISS, 2009) uses a hash table for their GPU-based path planning algorithm, but due to the peculiarities of GPU programming this hash table is implemented on the CPU. We chose a simpler grid data structure precisely to avoid such complications. Another advantage of using a grid is that it obviates the need for nearest neighbor queries over the scene domain. While these queries are usually simple for planar domains, nearest neighbor computation on meshes of arbitrary shape can be a very costly operation.

Our local-navigation scheme begins with each agent tagging, as occupied, the grid cell that encloses its center of mass. Following, each agent tests its preferred path for collision against the collision grid. If the path is collision free, the agent can follow it; otherwise, a new path is defined and again tested for collision. The process is repeated, until a free path is found.

Since agents are controlled by parallel threads, the grid cell tagging must be done as an atomic write. It is, however, the only synchronized operation in our algorithm. Even if agents are larger than the grid cell size they only mark a single cell. This improves performance by minimizing the number of critical sessions involved. Our grid sampling strategy ensures that agents will be able to detect one another even when they are larger than the cell they occupy.

When attempting to move forward agents first sample the grid in a neighborhood of their current position and intended path, searching for tagged cells. If they find any obstacles they randomly choose a small amount of time to stop and then select a new moving direction. The stop is intended to give passage to other agents and reduce the chance of a deadlock. We explain the process of choosing a new direction in further detail in section 6.3.2. Figure 6.5 illustrates the collision test process.

In our algorithm, local navigation is affected by three main parameters: the size of

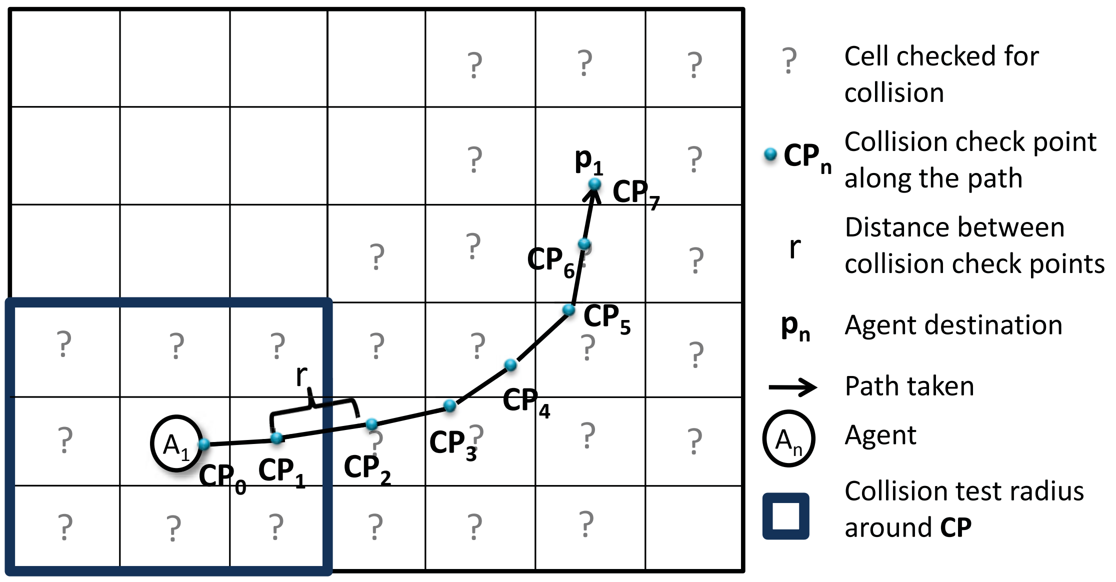

?
Cell checked for collision
CPn
Collision check point along the path
r
Distance between collision check points
pn
Agent destination
Path taken
An
Agent
Collision test radius around CP
p1
CP0
CP1
CP2
CP3
CP4
CP5
CP6
CP7
A1
r

Figure 6.5: An agent A_1 moves to its target p_1. At each step through the global path, the agent checks the grid in its local neighborhood for occupation. The sampling radius around the agent depends on the ratio between the agent's radius and the grid resolution.

the sampling neighborhood, the grid resolution and the discrete step r taken by agents for each time step. The radius of the sampling neighborhood is defined by the ratio between the agent's radius and the grid resolution. Thus, if the agent is much larger than a grid cell, it will accordingly check more cells around its neighborhood. This ensures that agents will not collide even though they only tag a single cell in the grid. Nevertheless, the grid resolution should be close to the agent size, striking a balance between memory usage and accuracy of the results. Finally, r is a discretization parameter that is usually set as the radius of the agent.

One drawback of our collision avoidance method is that agents only avoid one another when they are relatively close. A more refined approach could be achieved by making agents modify their distance fields, for example by adding a Gaussian bump to their local neighborhood. This would make other agents smoothly avoid contact, but at an increased cost, since the fields would have to be modified dynamically. This would leave few computational resources available for other parts of an interactive application, such as rendering and, in the context of games, AI and sound. Our approach is a simple compromise with small computational overhead.

Another advantage of using a 3-D grid, as opposed to a structure defined directly on $\mathcal{S}$, is the ability to naturally support external obstacles not on the mesh. Moreover, two agents can be close enough to collide even though the distance between them on $\mathcal{S}$ might be arbitrarily large. Figure 6.6 depicts an example of these cases. Only a true 3-D data structure can naturally deal with these situations.

6.3.2 Finding an Unobstructed Direction

In order to simplify our collision avoidance scheme, the result of sampling the grid is a binary answer. Agents have no knowledge as to where along their path the collision occurred. Therefore, our collision avoidance mechanism searches for new paths around the original one, and repeats the grid sampling in these new directions.

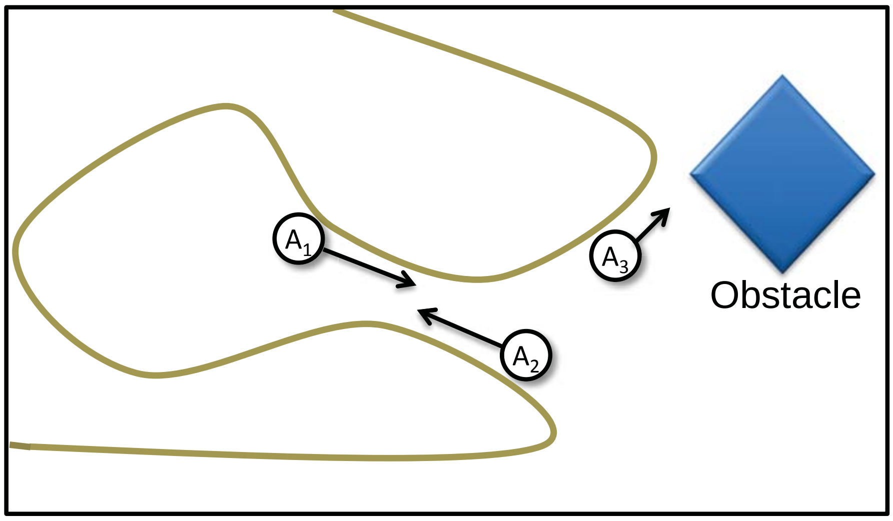


Figure 6.6: **Spatial collisions.** Our grid-based collision avoidance system is capable of detecting collisions between agents that are far apart over the surface of the mesh, even though they are close together in terms of simple Euclidean distance (observe agents A_1 and A_2). Also, it allows agents such as A_3 to avoid obstacles not necessarily on the surface of the mesh.

The modified path through which an agent will move is defined by following a new vector constructed by *rotating* ∇f_i. This makes the agent follow a logarithmic spiral that converges on the objective, as we illustrate in Figure 6.7. By traveling on this spiral, instead of the original path, agents can smoothly avoid obstacles along the way.

To determine the rotation angle for the spiral, agents progressively test wider rotations around the original path, as depicted in Figure 6.8. For each candidate rotation, the agent repeats the collision test described earlier. This process terminates either when the agent finds a free path, or when the rotation angle reaches a user-defined upper limit. This upper limit, β, exists to ensure that agents will not curve too sharply. If the agent cannot find a rotation smaller than β, it stops for a small random amount of time, waiting for its neighboring agents to move out of the way. Note that a small β may result in an agent holding forever in front of a static obstacle. However, if the obstacle is static it can be considered in the gradient generation, that way, the global navigation would give a collision free route. This can be done by removing, from the surface mesh, the intersection of the obstacle with the surface.

The agents gradually blend the rotated vector with the original direction. This results in a smooth transition between the path of steepest descent and the collision-aware route. A drawback of this approach is that some collisions may still occur in very specific conditions where agents can't dodge one another quickly enough. In our experiments, however, this was not the dominant situation.

After an agent has rounded an obstacle, it can safely return to the path of steepest descent (see Figure 6.9). Therefore, we reset the rotation angle and repeat the collision test at every time step.

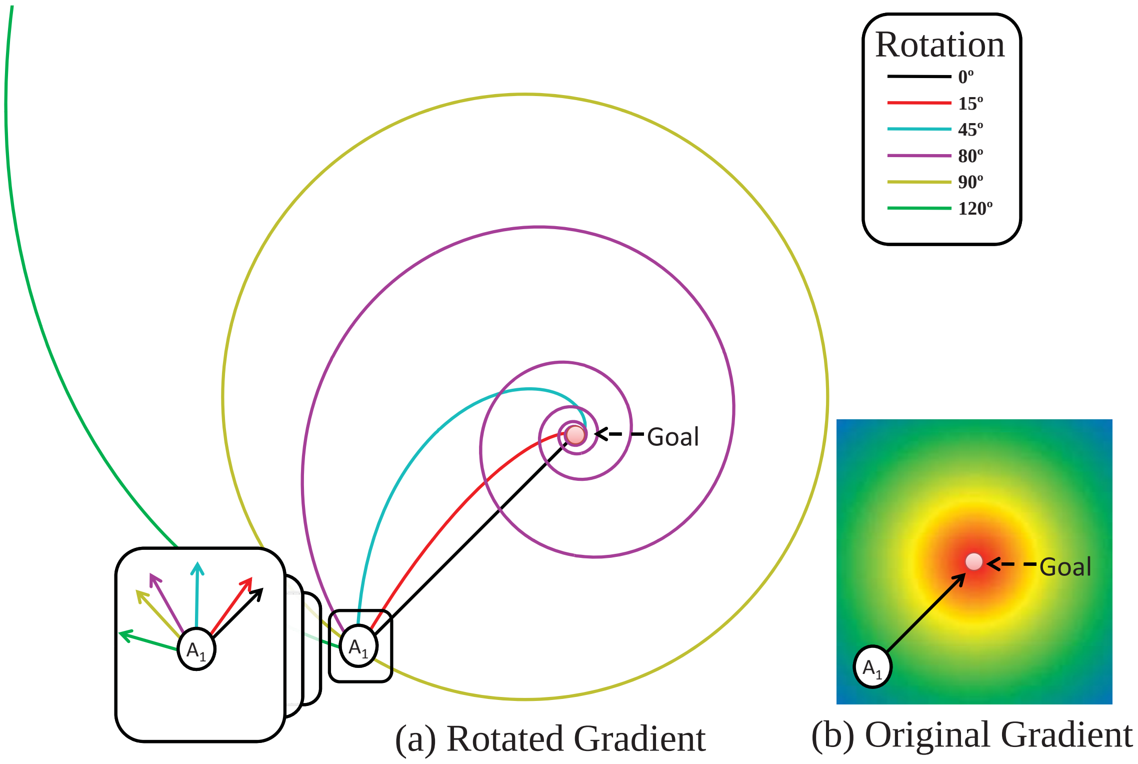


Figure 6.7: **Rotating the Gradient.** Rotating the global path by a constant angle results in progressively wider logarithmic spirals. As long as the rotation angle is less than 90°, however, the spirals are guaranteed to converge on the objective, as can be seen in this illustration. When the angle is exactly 90°, the path is a circle around the objective, and if the angle is more than that, the path diverges to infinity. Note that item A is the result over the simple gradient of item B. However, in a concave surface, instead of a rectangular as in this case, the resulting paths would be different because the resulting gradient would be different from item B.

6.4 Results

We have conducted our experiments on an Intel Core 2 Quad running at 2.83GHz, with 3GB of RAM, and an NVIDIA GeForce GTX260 graphics card. We have tested our system with three different meshes, the Stanford Bunny, the Fertile dataset and the Moai Statue. Table 6.1 shows the size of these datasets.

Model Name	Vertex Count	Face Count
Fertility	4994	10000
Moai Statue	10092	20180
Stanford Bunny	34834	69664

Table 6.1: ***Test Datasets.*** This table shows the size of the datasets we used as surfaces for our path planning algorithm.

As can be seen in Figure 6.10, our method performs at interactive frame rates even with a large number of agents. It is interesting to observe that the performance of the system is practically constant when the number of agents is less than 512. This is due to the fact that, as long as this number is less than the amount of threads supported in the

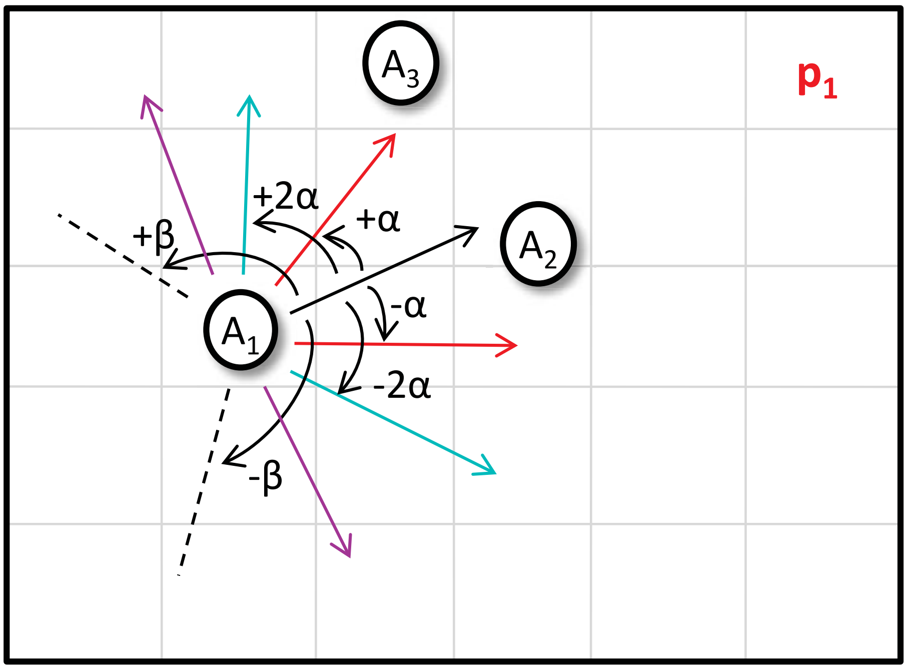


Figure 6.8: **Collision avoidance.** When the steepest descent path is blocked by an obstacle, agents progressively search for wider rotation angles around the original route. This is done by adding clockwise and counter-clockwise increments $k\alpha$ to the original path, where k is an integer factor.

graphics card, all the work is parallelizable. Once the number of agents goes beyond this value, the card must make more than one serial pass to generate the results. Therefore, it is expected that the performance will decrease gradually as the number of agents increases by increments of 512. Also, we have included a graph showing the performance without rendering, to better highlight the computational cost of our path planning technique.

Although Figure 6.10 shows results for only a single mesh (the Fertility model), we have observed that all meshes have a similar performance behavior, since the query for the gradient over the mesh can be done in constant time. The only performance figure that varies with mesh size is the time it takes to compute the distance fields and send them to the GPU. These were 1.5 seconds for the Stanford Bunny, 0.3 seconds for the Moai Statue and 0.12 seconds for the Fertility model. We can observe a strong correlation between these times and the number of triangles of the models.

It is interesting to note that the performance of our local navigation system does not depend on the size of the datasets. This is due to the fact that our implementation allows agents to fetch ∇f in constant time. Also, the performance loss that we observe when increasing the resolution of the grid is probably due to caching behavior on the graphics card, since the grid is stored in random access memory.

Figures 6.11 and 6.12 show our path planning algorithm running on two different datasets. In both cases, the agents are color-coded by their objective, which is also rendered as a solid sphere.

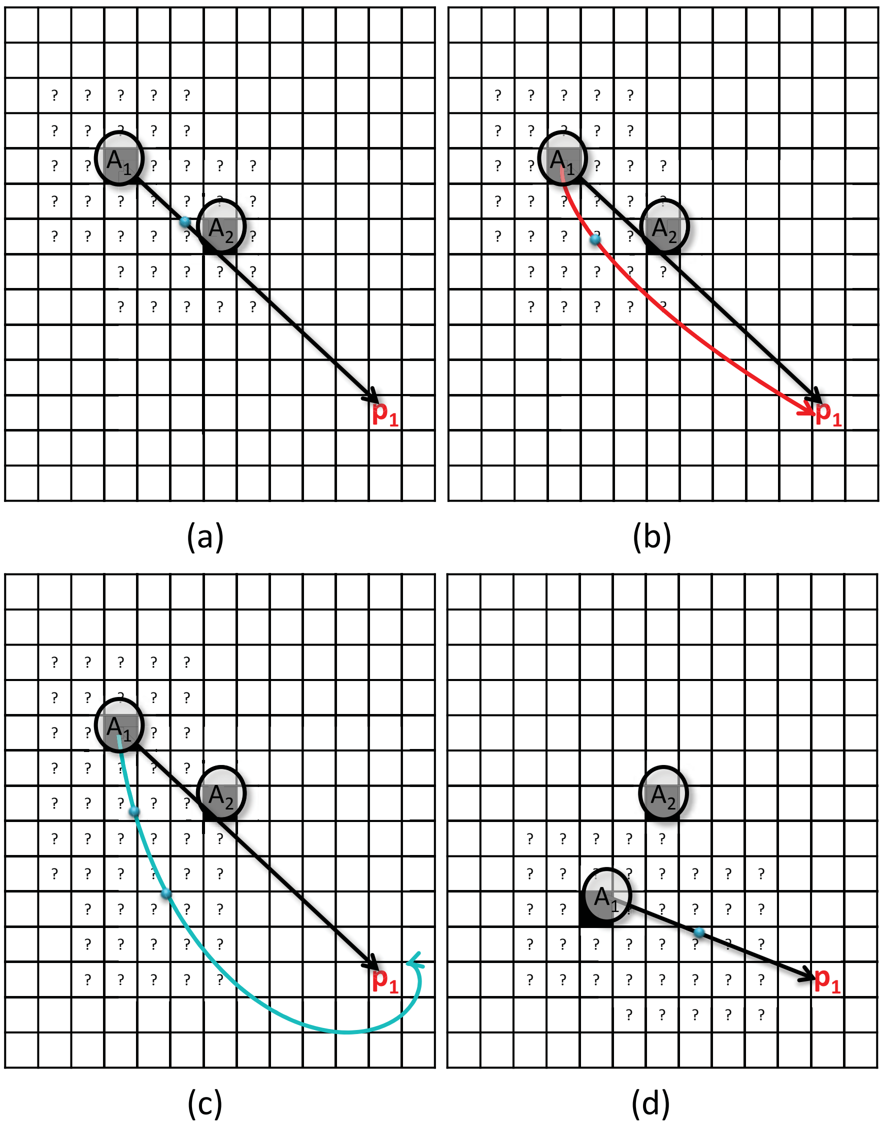


Figure 6.9: **Finding a free route.** An agent A_1 must find a collision-free path to the goal. The path of steepest descent causes a direct collision between A_1 and A_2 (a). A_1 then rotates its original direction in search of a free route (b), until it finds a rotation angle wide enough to avoid A_2 (c). Finally, after avoiding A_2, the agent again follow the path of steepest descent to the goal (d).

6.5 Discussion and Limitations

Our global navigation approach is similar to existing techniques in the literature, where we define agent movement by the gradient of a distance field. However, we found no existing mechanism of local navigation that could easily be extended to arbitrary surfaces. Therefore, we designed a simple grid-based collision avoidance system that strikes a compromise between collision-free navigation and smooth movement. While our solution does not, in principle, guarantee that no collisions will occur, we have conducted experiments and concluded that collisions are avoided in most of the situations, which is acceptable for games given the ability to navigate on arbitrary meshes. One of the main advantages of tagging only one cell of the grid is the reduced concurrent writing. However, it assumes a sphere or cube bounding box around the agent. For handling

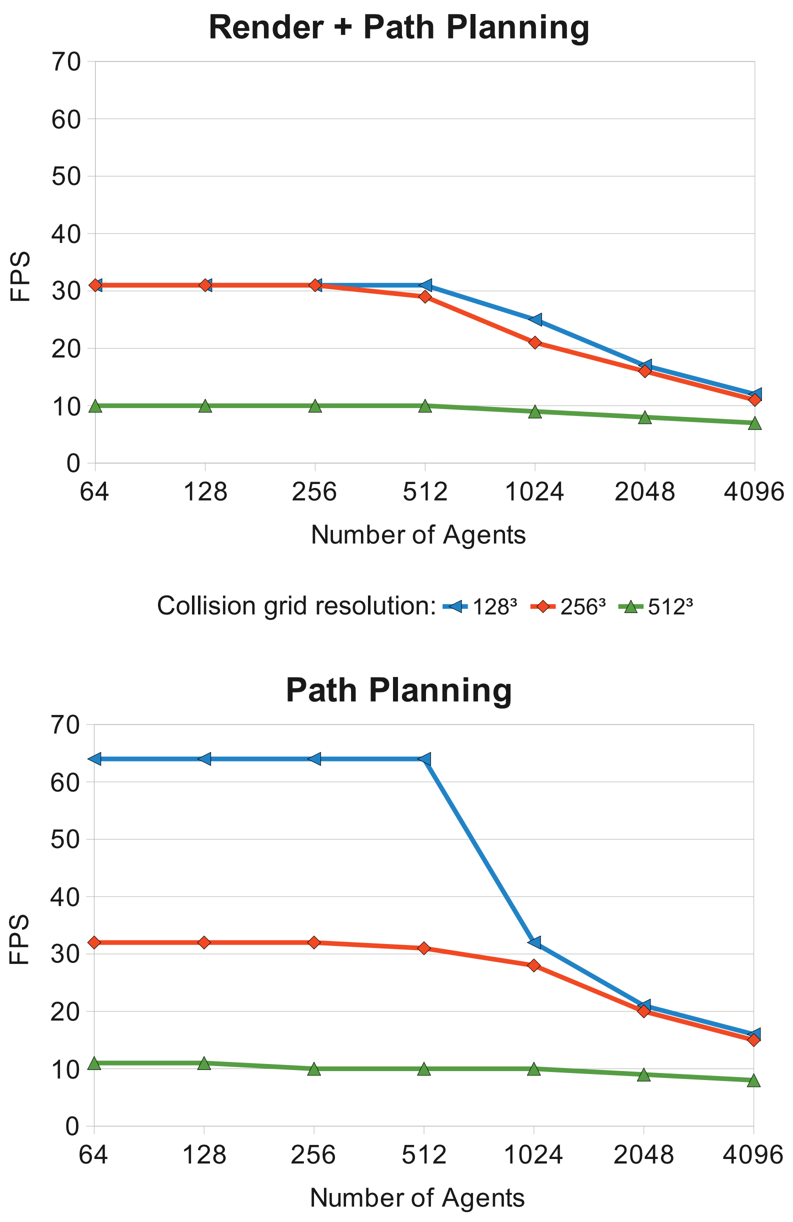


Figure 6.10: **Performance Graphs.** Depicted here are performance results of our algorithm. We measured FPS in an interactive system while varying the number of agents and the resolution of the collision grid. On the left, we illustrate the results of running our path planning algorithm with rendering enabled, while on the right we show the performance of the path planning pipeline running without rendering.

less-regular-shaped agents, we could store shape information, along with the tagging in the collision grid. The collision test would also have to be enhanced for using irregular shapes.

We have designed and implemented a solution meant to be *practical*. Although the 3-D grid has limitations, such as forcing a compromise between memory usage and simulation accuracy, the simplicity of this data structure makes it very well suited for implementation on the GPU. The massively parallel nature of this processor is very attractive to regular data structures, which need no dynamic updates and have a predictable access pattern. This access pattern naturally improves the performance of our algorithm, due to

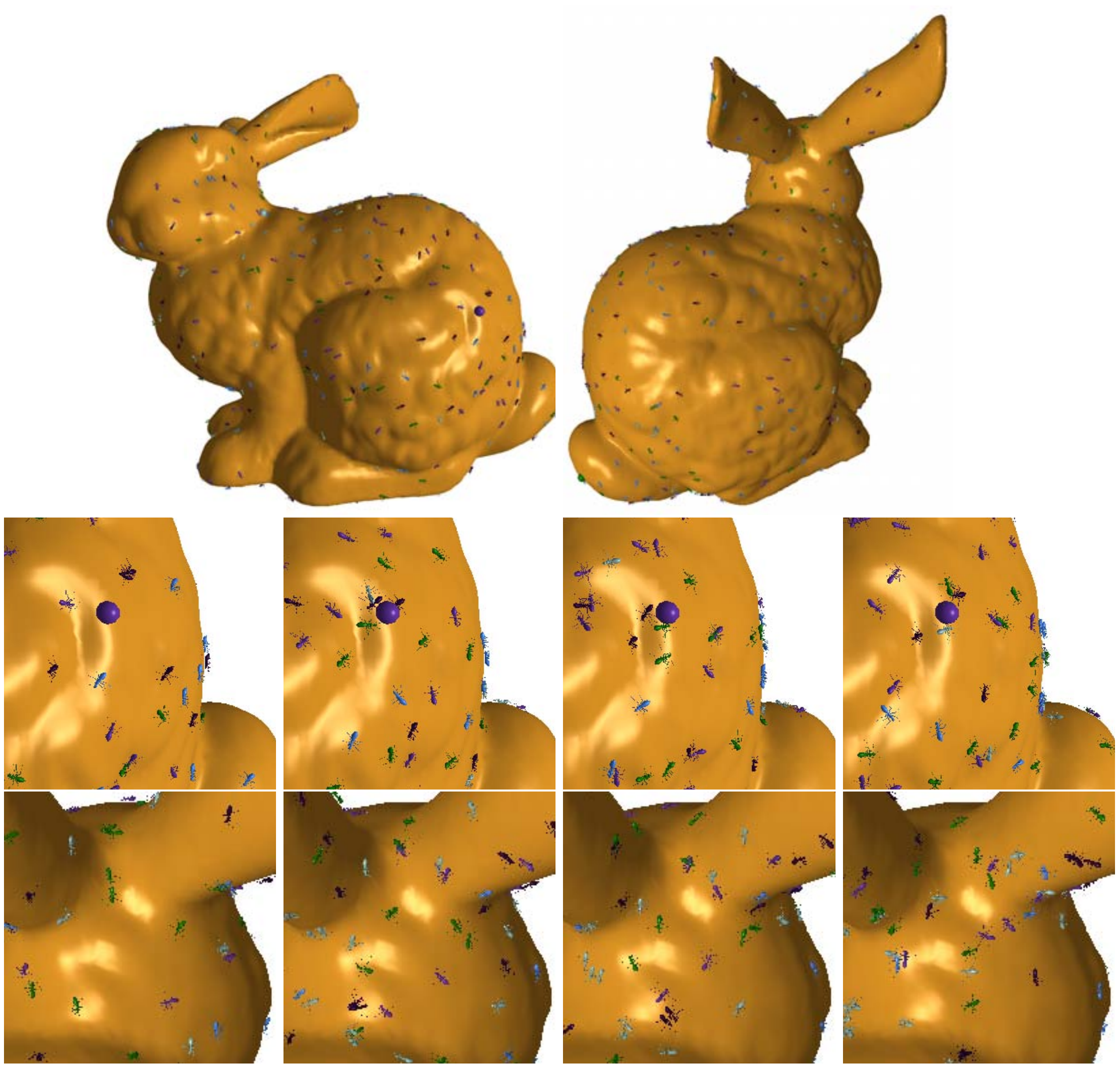


Figure 6.11: **The Stanford Bunny.** Four time steps of a multi-agent simulation running on the surface of the Stanford Bunny dataset, observed from two different camera positions. The agents are colored according to their objectives, which are also rendered as solid spheres. This simulation was executed with 512 agents over the mesh.

efficient memory caches. Also, our technique imposes an affordable computational overhead, and can therefore be easily integrated into larger applications, such as games and VR environments.

Figure 6.6 depicts a situation where two agents that are close together will collide even though they are far apart on the surface. This is correct and expected behavior, but our solution might fail in the opposite case, that is, when two agents are on opposite sides of a positively curved surface, such as a sharp bump on the mesh. This can happen mainly due to an excessively coarse grid, resulting in an overly conservative collision avoidance test.

Our approach aimed at avoiding collisions without trying to mimic particular behavior of the agents (ants, humans, etc.). The motivation for our approach was performance, which usually limits the design of most path planning algorithms in games. In general, games use the same path planning algorithm for all kinds of agents, only changing their size and speed, due to the computational cost for handling more complex agents.

6.6 Conclusion and Future Work

This chapter presented a novel approach for multi-agent path planning on arbitrary surfaces. To our knowledge, the work (TORCHELSEN et al., 2010) was the first work in the literature to address this problem. Our solution makes no assumptions about the domain mesh, and can handle surfaces of arbitrary shape and genus. We use a hierarchical computation of geodesic distances on the domain to define a scalar field whose gradient smoothly guides the agents to their goals. In order to avoid collision between agents we designed a grid-based local navigation algorithm that can also handle obstacles not on the mesh. Our collision avoidance technique is approximate, but we have conducted many experiments and found that few collisions occur. We have also integrated these techniques into a hybrid CPU/GPU pipeline carefully designed to exploit both processors particular strengths. We have implemented the hierarchical distance field computation on the CPU, and the massively parallel collision avoidance algorithm entirely on the GPU, using CUDA.

Currently, our method requires that the domain mesh be static. To allow dynamic domains it would be necessary to modify the hierarchical distance field computation to update its results every time the mesh changes, as well as update the agents position to ensure that they are still properly on the mesh. Another challenge for future work is to design and implement a more refined data structure to deal with collision avoidance. Hierarchical structures such as kD-trees or Octrees present themselves as attractive alternatives.

Another direction of research is to compute the local navigation entirely in the parametric domain, as we do with the distance computation (see chapter 5). This would allow the use of existing local navigation techniques, because the parameterization is defined in 2-D. However, the local navigation would only consider collisions on the surface of the mesh and performance might become an issue for real-time applications.

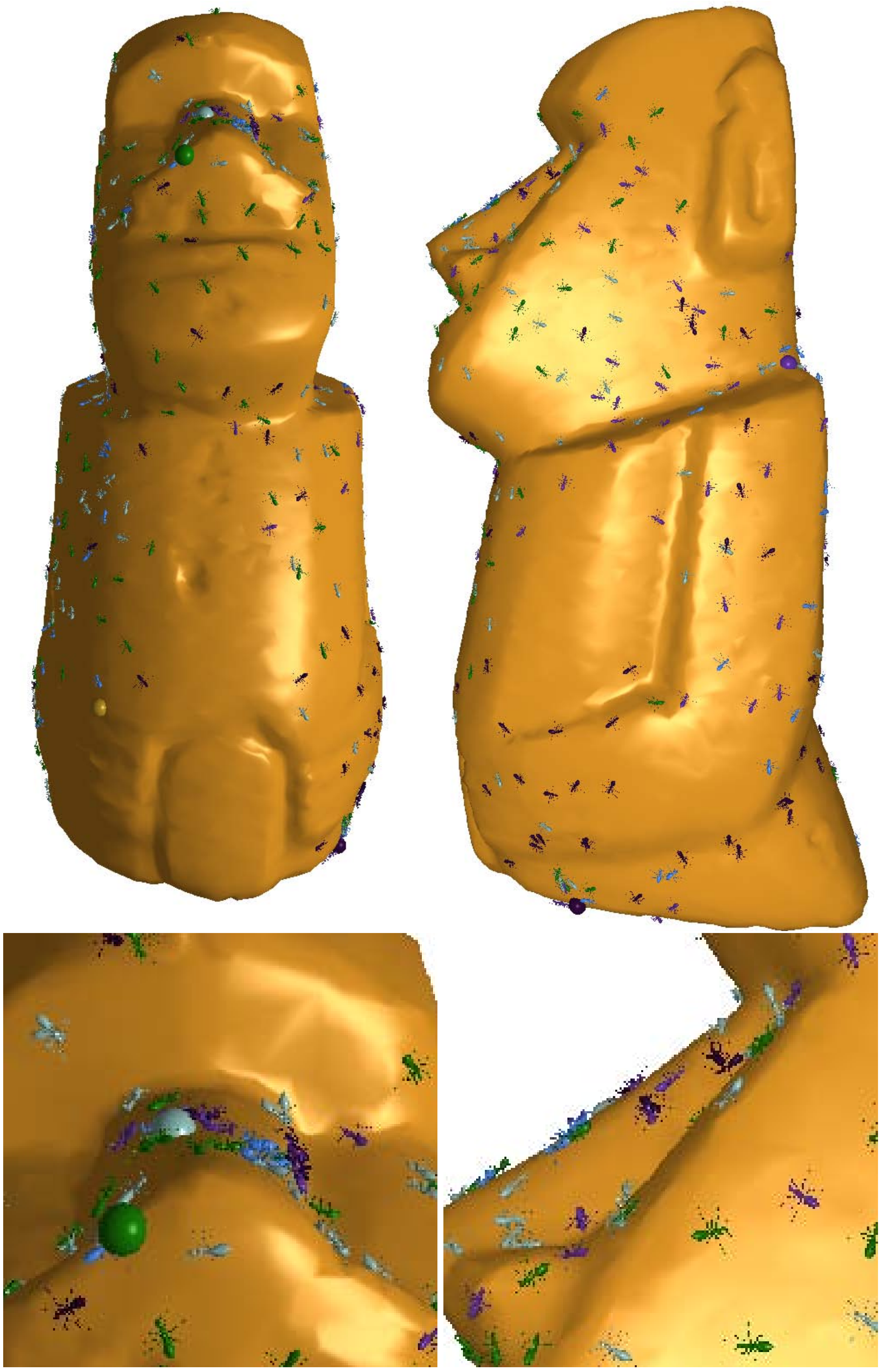


Figure 6.12: **The Moai Statue.** Time step of the algorithm viewed from two different positions, simulating 512 agents on top of the Moai Statue. In this image we can observe that some collision cases occur, as can be seen on the nose of the model. This is rare, however, and happened in this case only because there are two objectives very close together (the green and blue-gray objectives).

7 CONCLUSION

This work's contribution to computer graphics comes from using unfolded meshes to improve the efficiency and visual quality of several techniques, such as displacement mapping, on-surface distance computation and agent path planning.

The contribution on efficient rendering and reconstruction of surfaces encoded on displacement maps allows the use of more detailed surfaces in virtual environments. The visual appeal of such surfaces, particularly terrains, plays a major role in user immersion in the virtual environment. This contribution allows the use of more detailed surfaces particularly in games. The contributions to displacement mapping are also discussed in: (TORCHELSEN; BASTOS; COMBA, 2008) and (TORCHELSEN et al., 2009).

Another important contribution is the efficient computation of on-surface distances, which is a process used on several applications, including: procedural textures, surface labeling, remeshing, mesh segmentation, agent path planning, etc. The use of quasi-developable charts to compute on-surface distances has increased the computational efficiency, in some cases, by more them an order of magnitude.

On-surface distance computation in parametric space, to the best of our knowledge, was first presented on (TORCHELSEN et al., 2010). The use of parametric space is a general method, in the sense that it can be used on most existing distance computation techniques, not being restricted to any specific algorithm. This part of the work is also discussed in (TORCHELSEN et al., 2009).

The agent path planning algorithm presented in this thesis is an example of a new research line created by the presented on-surface distance computation method. Until now, the movement of agents in real-time applications was limited to almost-planar surfaces. The possibility of computing geodesics in real-time allowed for the development of a method to navigate agents on more generic surfaces. Even though there was a significant improvement in performance on the distance computation side of the agent path planning equation, the computational cost of simultaneously calculating several on-surface distances is prohibitive for real-time applications using only the CPU or the GPU. A path planning algorithm was thus developed to distribute the computational cost to both processors.

An important part of this work is designing the methods to combine the processing of the CPU and GPU. Using a hybrid approach the agent path planning method is capable to navigate 512 agents in real-time on the surface of complex meshes. This technique is also discussed in (TORCHELSEN et al., 2010).

The applications of this work are many; furthermore, new research lines are possible, including:

- **Distance computation algorithm**: The use of quasi-developable charts to compute distances can be further improved. Currently our method re-triangulates the interior of the charts to use existing distance computation algorithms. Those algorithms are designed for triangular meshes. The re-triangulation may be unnecessary, because the boundary of the charts already defining the region that is quasi-developable. An algorithm to compute distance fields in the interior of simple polygons, besides triangles, can be more efficient due to the reduced number of geometric primitives.
- **Real-Time Procedural Textures**: Procedural coloring of surfaces, as in Walter et al (WALTER; FOURNIER; MENEVAUX, 2001), requires on-surface distance computations, which can now be computed more efficiently. Such techniques can now be extended for interactive simulations.
- **Agent path planning in the parametric space**: The presented agent path planning technique uses distances computed on the parametric space, however, the collision avoidance is computed in object space. Computing the entire process in parametric space can improve the performance by eliminating one dimension from the space used in the collision avoidance step.

REFERENCES

AMOR, M.; BOO, M.; STRASSER, W.; HIRCHE, J.; DOGGETT, M. A Meshing Scheme for Efficient Hardware Implementation of Butterfly Subdivision Using Displacement Mapping. **IEEE Comput. Graph. Appl.**, Los Alamitos, CA, USA, v.25, n.2, p.46–59, 2005.

ASIRVATHAM, A.; HOPPE, H. **Terrain Rendering Using GPU-Based Geometry Clipmaps**. [S.l.]: Addison Wesley, 2005. n.2, p.27–45. (GPU Gems).

BALASUBRAMANIAN, M.; POLIMENI, J. R.; SCHWARTZ, E. L. Exact Geodesics and Shortest Paths on Polyhedral Surfaces. **IEEE TPAMI**, [S.l.], v.31, n.6, 2009.

BERG, J. van den; LIN, M.; MANOCHA, D. Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation. In: IEEE INT. CONF. ON ROBOTICS AND AUTOMATION - ICRA'08, 2008. **Proceedings...** [S.l.: s.n.], 2008.

BERG, J. van den; PATIL, S.; SEWALL, J.; MANOCHA, D.; LIN, M. Interactive navigation of multiple agents in crowded environments. In: SI3D '08: PROCEEDINGS OF THE 2008 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.139–147.

BERG, J. van den; PATIL, S.; SEWALL, J.; MANOCHA, D.; LIN, M. Interactive navigation of multiple agents in crowded environments. In: I3D '08: PROCEEDINGS OF THE 2008 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.139–147.

BLEIWEISS, A. Multi Agent Navigation on the GPU. In: GDC'09: GAME DEVELOPERS CONFERENCE 2009, 2009. **Anais...** [S.l.: s.n.], 2009.

BOTSCH, M.; PAULY, M.; ROSSL, C.; BISCHOFF, S.; KOBBELT, L. Geometric modeling based on triangle meshes. In: SIGGRAPH '06: ACM SIGGRAPH 2006 COURSES, 2006, New York, NY, USA. **Anais...** ACM, 2006. p.1.

CHEW, L. P. Constrained Delaunay triangulations. In: SCG '87: PROCEEDINGS OF THE THIRD ANNUAL SYMPOSIUM ON COMPUTATIONAL GEOMETRY, 1987, New York, NY, USA. **Anais...** ACM, 1987. p.215–222.

CIPRIANO, G.; GLEICHER, M. Text Scaffolds for Effective Surface Labeling. **IEEE TVCG**, Piscataway, NJ, USA, v.14, n.6, p.1675–1682, 2008.

CLASEN, M.; HEGE, H.-C. Terrain rendering using spherical clipmaps. In: EUROVIS 2006 - EUROGRAPHICS / IEEE VGTC SYMPOSIUM ON VISUALIZATION, 2006. **Anais...** [S.l.: s.n.], 2006. p.91–98.

COHEN-STEINER, D.; ALLIEZ, P.; DESBRUN, M. Variational shape approximation. **ACM Trans. Graph.**, New York, NY, USA, v.23, n.3, p.905–914, 2004.

COOK, R. L. Shade trees. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.18, n.3, p.223–231, 1984.

DIJKSTRA, E. W. A Note on Two Problems in Connexion with Graphs. **Numerische Mathematik**, [S.l.], v.1, p.269–271, 1959.

DOGGETT, M.; HIRCHE, J. Adaptive view dependent tessellation of displacement maps. In: HWWS '00: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS WORKSHOP ON GRAPHICS HARDWARE, 2000, New York, NY, USA. **Anais...** ACM, 2000. p.59–66.

DUCHAINEAU, M.; WOLINSKY, M.; SIGETI, D.; MILLER, M.; ALDRICH, C.; MINEEV-WEINSTEIN, M. ROAMing terrain: real-time optimally adapting meshes. In: IEEE VISUALIZATION, 1997. **Anais...** [S.l.: s.n.], 1997. p.81–88.

DYKE, C.; REIMERS, M.; SELAND, J. Real-Time GPU Silhouette Refinement using Adaptively Blended B&eacute;zier Patches. **Computer Graphics Forum**, [S.l.], v.27, n.1, p.1–12, 2008.

FIORINI, P.; SHILLER, Z. Motion Planning in Dynamic Environments using Velocity Obstacles. **International Journal of Robotics Research**, [S.l.], v.17, n.7, p.760–772, 1998.

FLOATER, M. S.; HORMANN, K. Surface Parameterization: a tutorial and survey. In: ADVANCES IN MULTIRESOLUTION FOR GEOMETRIC MODELLING, 2005. **Anais...** Springer Verlag, 2005. p.157–186.

FOUDIL, C.; NOUREDDINE, D. Collision Avoidance in Crowd Simulation with Priority Rules. **European Journal of Scientific Research**, [S.l.], v.15, n.1, October 2006.

FULGENZI, C.; SPALANZANI, A.; LAUGIER, C. Dynamic Obstacle Avoidance in uncertain environment combining PVOs and Occupancy Grid. In: IEEE INT. CONF. ON ROBOTICS AND AUTOMATION, 2007. **Proceedings...** [S.l.: s.n.], 2007.

FUNKHOUSER, T.; KAZHDAN, M.; SHILANE, P.; MIN, P.; KIEFER, W.; TAL, A.; RUSINKIEWICZ, S.; DOBKIN, D. Modeling by example. **ACM TOG**, New York, NY, USA, v.23, n.3, p.652–663, 2004.

GRIMM, C.; ZORIN, D. Surface modeling and parameterization with manifolds. In: SIGGRAPH '05: ACM SIGGRAPH 2005 COURSES, 2005, New York, NY, USA. **Anais...** ACM, 2005. p.1.

GUY, S. J.; CHHUGANI, J.; KIM, C.; SATISH, N.; LIN, M.; MANOCHA, D.; DUBEY, P. ClearPath: highly parallel collision avoidance for multi-agent simulation. In: SCA '09: PROCEEDINGS OF THE 2009 ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON COMPUTER ANIMATION, 2009, New York, NY, USA. **Anais...** ACM, 2009. p.177–187.

HART, P.; NILSSON, N.; RAPHAEL, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. **Systems Science and Cybernetics, IEEE Transactions on**, [S.l.], v.4, n.2, p.100–107, July 1968.

HOPPE, H. Smooth view-dependent level-of-detail control and its application to terrain rendering. In: IEEE VISUALIZATION, 1998. **Anais...** IEEE Computer Society, 1998. p.35–42.

HORMANN, K.; LéVY, B.; SHEFFER, A. Mesh parameterization: theory and practice. In: SIGGRAPH '07: ACM SIGGRAPH 2007 COURSES, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.1.

JULIUS, D.; KRAEVOY, V.; SHEFFER, A. D-Charts: quasi-developable mesh segmentation. In: COMPUTER GRAPHICS FORUM, PROCEEDINGS OF EUROGRAPHICS 2005, 2005, Dublin, Ireland. **Anais...** Blackwell, 2005. v.24, n.3, p.581–590.

JULIUS, D. N. **Developable Surface Processing Methods for Three-Dimensional Meshes**. 2006. Dissertação (Mestrado em Ciência da Computação) — The University Of British Columbia.

KHODAKOVSKY, A.; LITKE, N.; SCHRöDER, P. Globally smooth parameterizations with low distortion. In: ACM SIGGRAPH, 2003. **Anais...** [S.l.: s.n.], 2003. p.350–357.

KLUGE, B.; PRASSLER, E. Reflective Navigation: individual behaviors and group behaviors. In: ICRA, 2004. **Anais...** [S.l.: s.n.], 2004. p.4172–4177.

LAMARCHE, F.; DONIKIAN, S. Crowd of Virtual Humans: a new approach for real time navigation in complex and structured environments. **Computer Graphics Forum**, [S.l.], v.23, p.509–518, 2004.

LéVY, B.; PETITJEAN, S.; RAY, N.; MAILLOT, J. Least squares conformal maps for automatic texture atlas generation. **ACM Trans. Graph.**, New York, NY, USA, v.21, n.3, p.362–371, 2002.

LINDSTROM, P.; PASCUCCI, V. Visualization of large terrains made easy. In: VIS '01: PROCEEDINGS OF THE CONFERENCE ON VISUALIZATION '01, 2001, Washington, DC, USA. **Anais...** IEEE Computer Society, 2001. p.363–371.

LIU, L.; ZHANG, L.; XU, Y.; GOTSMAN, C.; GORTLER, S. J. A Local/Global Approach to Mesh Parameterization. In: COMPUTER GRAPHICS FORUM, PROCEEDINGS OF EUROGRAPHICS SYMPOSIUM ON GEOMETRY PROCESSING 2008 (SGP 2008), 2008. **Anais...** Blackwell, 2008. v.27, n.5, p.1495–1504.

LIU, Y.-J.; ZHOU, Q.-Y.; HU, S.-M. Handling degenerate cases in exact geodesic computation on triangle meshes. **Visual Comput.**, Secaucus, NJ, USA, v.23, n.9, p.661–668, 2007.

LIVNY, Y.; SOKOLOVSKY, N.; GRINSHPOUN, T.; EL-SANA, J. A GPU persistent grid mapping for terrain rendering. **The Visual Computer**, Secaucus, NJ, USA, v.24, n.2, p.139–153, 2008.

LOSASSO, F.; HOPPE, H. Geometry clipmaps: terrain rendering using nested regular grids. **ACM Trans. Graph.**, New York, NY, USA, v.23, n.3, p.769–776, 2004.

MEYER, M.; DESBRUN, M.; SCHROEDER, P.; BARR, A. Discrete Differential Geometry Operators for Triangulated 2-Manifolds. In: VISMATH '02 PROCEEDINGS, 2002. **Anais...** [S.l.: s.n.], 2002.

MITCHELL, J. S. B.; MOUNT, D. M.; PAPADIMITRIOU, C. H. The discrete geodesic problem. **SIAM J. Comput.**, Philadelphia, PA, USA, v.16, n.4, p.647–668, 1987.

MOREIRA, D. M.; VELHO, L.; CARVALHO, P. C. Geodesic Paths on Triangular Meshes. **Computers & Graphics Journal**, Washington, DC, USA, v.29, n.5, p.667–675, 2005.

MOULE, K.; MCCOOL, M. D. Efficient Bounded Adaptive Tessellation of Displacement Maps. In: IN GRAPHICS INTERFACE, 2002. **Anais...** [S.l.: s.n.], 2002. p.171–180.

NVIDIA. **NVIDIA CUDA**. 2009.

PAJAROLA, R.; GOBBETTI, E. Survey on Semi-Regular Multiresolution Models for Interactive Terrain Rendering. In: THE VISUAL COMPUTER, 2007. **Anais...** [S.l.: s.n.], 2007. v.23(8), p.583–605.

PARIS, S.; PETTRE, J.; DONIKIAN, S. Pedestrian Reactive Navigation for Crowd Simulation: a predictive approach. **Computer Graphics Forum**, [S.l.], v.26, p.665–674, 2007.

PEYRé, G.; COHEN, L. Geodesic computations for fast and accurate surface remeshing and parameterization. **Progress in Nonlinear Differential Equations and Their Applications**, [S.l.], v.63, p.157–171, 2005.

SANDER, P. V.; SNYDER, J.; GORTLER, S. J.; HOPPE, H. Texture mapping progressive meshes. In: SIGGRAPH '01: PROCEEDINGS OF THE 28TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2001, New York, NY, USA. **Anais...** ACM, 2001. p.409–416.

SETHIAN, J. **Level Set Methods and Fast Marching Methods**. [S.l.]: Cambridge University Press, 1999.

SHAMIR, A. A survey on Mesh Segmentation Techniques. **Comput. Graph. Forum**, [S.l.], v.27, n.6, p.1539–1556, 2008.

SHEFFER, A.; HORMANN, K.; LEVY, B.; DESBRUN, M.; ZHOU, K. Mesh Parameterization: theory and practice. In: SIGGRAPH 2007 (COURSE NOTES), 2007. **Anais...** ACM, 2007.

SHEFFER, A.; LéVY, B.; MOGILNITSKY, M.; BOGOMYAKOV, A. ABF++: fast and robust angle based flattening. **ACM TOG**, New York, NY, USA, v.24, n.2, p.311–330, 2005.

SHEFFER, A.; PRAUN, E.; ROSE, K. Mesh parameterization methods and their applications. **Found. Trends. Comput. Graph. Vis.**, Hanover, MA, USA, v.2, n.2, p.105–171, 2006.

SILVEIRA, R.; PRESTES, E.; NEDEL, L. Managing Coherent Groups. **Computer Animation and Virtual Worlds**, [S.l.], 2008.

SUD, A.; ANDERSEN, E.; CURTIS, S.; LIN, M.; MANOCHA, D. Real-time Path Planning for Virtual Agents in Dynamic Environments Using Multi-agent Navigation Graphs. **vr**, Los Alamitos, CA, USA, v.0, p.91–98, 2007.

SUD, A.; GAYLE, R.; ANDERSEN, E.; GUY, S.; LIN, M.; MANOCHA, D. Real-time navigation of independent agents using adaptive roadmaps. In: VRST '07: PROCEEDINGS OF THE 2007 ACM SYMPOSIUM ON VIRTUAL REALITY SOFTWARE AND TECHNOLOGY, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.99–106.

SURAZHSKY, V.; SURAZHSKY, T.; KIRSANOV, D.; GORTLER, S. J.; HOPPE, H. Fast exact and approximate geodesics on meshes. **ACM TOG**, New York, NY, USA, v.24(3), p.553–560, 2005.

SZIRMAY-KALOS, L.; UMENHOFFER, T. Displacement Mapping on the GPU - State of the Art. **Computer Graphics Forum**, [S.l.], v.27, n.1, 2008.

TEWARI, G.; SNYDER, J.; SANDER, P. V.; GORTLER, S. J.; HOPPE, H. Signal-specialized parameterization for piecewise linear reconstruction. In: SGP '04: PROCEEDINGS OF THE 2004 EUROGRAPHICS/ACM SIGGRAPH SYMPOSIUM ON GEOMETRY PROCESSING, 2004, New York, NY, USA. **Anais...** ACM, 2004. p.55–64.

THALMANN, D.; MUSSE, S. R. **Crowd Simulation**. [S.l.]: Springer, 2007.

TORCHELSEN, R. P.; BASTOS, R.; COMBA, J. L. D. Practical Geometry Clipmaps for Rendering Terrains in Computer Games. **ShaderX 6: Advanced Rendering Techniques**, [S.l.], p.103–114, 2008.

TORCHELSEN, R. P.; DIETRICH, C.; SILVA, L. F. M. S.; BASTOS, R.; COMBA, J. L. D. Adaptive Re-Meshing for Displacement Mapping. **ShaderX 7: Advanced Rendering Techniques**, [S.l.], p.107–114, 2009.

TORCHELSEN, R. P.; PINTO, F.; BASTOS, R.; COMBA, J. L. D. Approximate on-Surface Distance Computation using Quasi-Developable Charts. **Computer Graphics Forum**, [S.l.], v.28, n.7, p.1781–1789, 2009.

TORCHELSEN, R. P.; SCHEIDEGGER, L. F.; OLIVEIRA, G. N.; BASTOS, R.; COMBA, J. L. D. Real-Time Multi-Agent Path Planning on Arbitrary Surfaces. **ACM Symposium on Interactive 3D Graphics and Games**, [S.l.], 2010.

WALTER, M.; FOURNIER, A.; MENEVAUX, D. Integrating shape and pattern in mammalian models. In: SIGGRAPH '01: PROCEEDINGS OF THE 28TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2001, New York, NY, USA. **Anais...** ACM, 2001. p.317–326.

WANG, C. Computing Length-Preserved Free Boundary for Quasi-Developable Mesh Segmentation. **IEEE TVCG**, Los Alamitos, CA, USA, v.14, n.1, p.25–36, 2008.

YAMAUCHI, H.; GUMHOLD, S.; ZAYER, R.; SEIDEL, H.-P. Mesh segmentation driven by Gaussian curvature. **Visual Comput.**, [S.l.], v.21, n.8-10, p.659–668, 2005.

YEH, H.; CURTIS, S.; PATIL, S.; BERG, J. van den; MANOCHA, D.; LIN, M. Composite Agents. **Symposium on Computer Animation - SCA'08**, [S.l.], 2008.