UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
MESTRADO E DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

Geração de Malhas por Refinamento Adaptativo Usando GPU

Autor
Ricardo Lenz

Orientadores
Joaquim Bento Cavalcante-Neto
Creto Augusto Vidal

Dissertação de Mestrado

FORTALEZA – CEARÁ – BRASIL
2009

UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
MESTRADO E DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

Ricardo Lenz

Geração de Malhas por Refinamento Adaptativo Usando GPU

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadores: Joaquim Bento Cavalcante-Neto
Creto Augusto Vidal

FORTALEZA – CEARÁ – BRASIL
2009

Geração de Malhas por Refinamento Adaptativo Usando GPU

Ricardo Lenz César

Dissertação apresentada ao Curso de Mestrado em Ciência da Computação da Universidade Federal do Ceará, como parte dos Requisitos para a obtenção do Grau de Mestre em Ciência da Computação.

Composição da Banca Examinadora:

Prof Dr. Joaquim Bento Cavalcante Neto (Presidente) (DC/UFC

Profa. Veronica Teichrieb (UFPE)

Prof. Dr. Creto Augusto Vidal (DC/UFC

Aprovada em 24 de abril de 2009

Agradecimentos

Em primeiro lugar, ao bom e eterno Deus, que me deu ânimo quando necessário e me guiou em momentos repletos de eventualidades.

Aos meus pais, Silas Lenz Cesar e Monica F. Lenz Cesar, por serem pessoas tão boas. Ser filho deles é um privilégio do qual eu e meu irmão compartilhamos. À minha namorada, amiga e companheira, Gleice, pela compreensão, estímulo e paciência.

Aos meus orientadores, Joaquim Bento e Creto Vidal, pela confiança, apoio e conhecimento passado. Aos meus colegas e amigos que me deram um ambiente alegre e inteligente na universidade.

À UFC (Universidade Federal do Ceará), pelo ensino, e ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), pelo apoio financeiro.

E, por fim, ao meu avô, Homero Lenz Cesar, que me serve de modelo em muitas coisas.

Resumo

O alto desempenho da GPU e o crescente uso dos seus mecanismos de programação têm estimulado diversas aplicações gráficas de realidade virtual a explorar melhor o potencial desse dispositivo para alcançar níveis mais altos de realismo. Trabalhos têm surgido com um enfoque no refinamento da silhueta de malhas geométricas, buscando expressar melhor a superfície dos objetos tridimensionais sendo representados. O tipo de refinamento aplicado pode ser, por exemplo, uma suavização da malha bruta de um avatar, por meio da interpolação de uma superfície curva sobre suas faces. A ideia básica é fazer uma discretização adaptativa da malha do objeto e então gerar uma nova silhueta usando essa discretização. Métodos anteriores são analisados e são apresentadas melhorias que juntas formarão o método proposto. O desempenho obtido é superior devido a uma exploração melhor do paralelismo da GPU, e a técnica proposta funciona suficientemente bem com malhas existentes sem necessidade de se projetar novos modelos para isso.

Abstract

The GPU's high performance processing allied with the growing use of its programming mechanisms have been stimulating several virtual reality and other real time rendering applications to better explore the graphics hardware in order to achieve higher levels of realism. Papers focused on the problem of mesh silhouette refinement have been published and the overall goal in these papers is to better express the surface of three-dimensional objects being rendered. One type of mesh refinement that can be made is smoothing a coarse triangular model, like, for example, an avatar's mesh in a virtual reality application. This kind of smoothing is accomplished by means of a curved surface interpolation on the coarse mesh data. The basic ideia is to adaptively discrete the object's mesh and then generate a new silhouette using this discretization. Previous methods are analyzed and some ideas are presented to achieve a better result. The proposed method is faster because of its better use of GPU's parallelism, and the technique works well even for existing meshes without the necessity for designing new three-dimensional models.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1. Introdução

1.1. Motivação

Um dos modos tradicionais para se atingir maiores níveis de realismo nas aplicações gráficas em tempo real tem sido a exploração de texturas, usadas como imagens que são mapeadas sobre a superfície dos objetos tridimensionais (Figura 1.1).

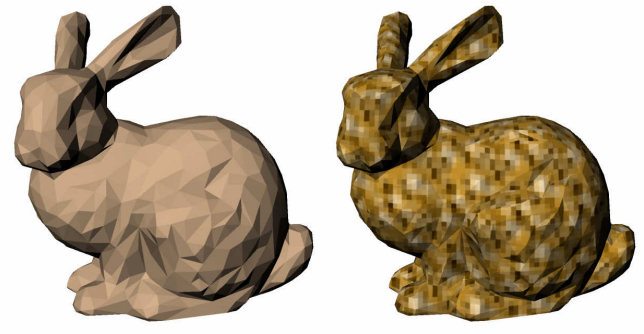


Figura 1.1. Aumento do realismo por meio de texturas.

Além de texturas, tem-se usado efeitos de iluminação calculados para cada *pixel* (Dyken et al., 2007), (Fünfzig et al., 2008), a fim de aumentar também o realismo (Figura 1.2). Os efeitos de iluminação são simplesmente formas diferentes de colorir a superfície dos objetos em função das luzes e sombras, tentando fazer com que a imagem gerada do objeto tridimensional possa parecer ser mais realística.

Figura 1.2. Aumento do realismo por meio de efeitos de iluminação.

Embora seja possível melhorar as texturas ou alguns dos efeitos aplicados, as malhas dos objetos não são nunca modificadas por esses métodos tradicionais. Por exemplo, mesmo usando uma nova textura de 512x512 *pixels* ao invés da textura de 64x64 *pixels* usada na Figura 1.1, a malha do objeto representado ainda continua a mesma (Figuras 1.3 e 1.4).

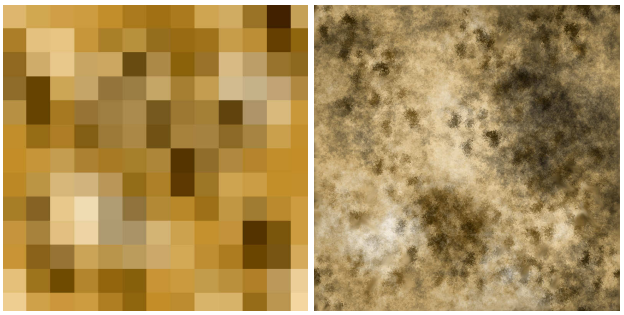


Figura 1.3. Melhorando as texturas para maior realismo.

Figura 1.4. Objeto mais realista por meio de uma textura melhor.

Pelo fato da malha permanecer inalterada nesses métodos tradicionais, as silhuetas ainda expõem as imperfeições da malha subjacente, destruindo parcialmente o realismo obtido pela aplicação desses métodos.

Um outro modo proposto para aumentar o realismo é o de explorar melhor a própria geometria do objeto, gerando-lhe uma nova silhueta a fim de aumentar sua expressividade. Esse tipo de técnica realiza, portanto, uma síntese geométrica de dados com base num modelo existente. O processo ocorre passando-se uma malha de entrada como parâmetro para uma função, que irá conduzir o devido refinamento para gerar a malha final (Figura 1.5).

Por exemplo, a função de refinamento pode ser uma interpolação de uma superfície cúbica de Bézier com base nos dados da própria malha original. Nesse caso, comumente se usa o

chamado *Curved PN Triangle* (Vlachos et al., 2001), que resultará numa malha com curvas mais suaves do que a sua versão inicial. Isso pode ser usado para expressar melhor o objeto na aplicação de realidade virtual, e o custo para isso é relativamente baixo. Dessa forma, há aplicação direta do trabalho apresentado em muitos aplicativos gráficos de tempo real, como sistemas de realidade virtual, jogos, etc.

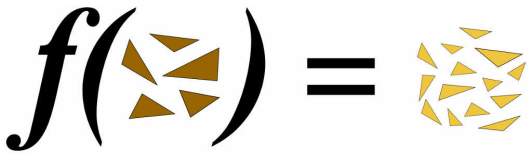


Figura 1.5. Síntese geométrica por meio de uma função que recebe uma malha existente.

Por fim, é importante mencionar que o desempenho de um método pode influenciar bastante o realismo obtido ao final. Isso ocorre quando, por exemplo, um método existente pode refinar até certo nível fixado de qualidade, num determinado espaço de tempo, e um novo método é capaz de atingir o mesmo nível fixado de qualidade num espaço de tempo muito menor. Assim, processando-se a malha no mesmo espaço de tempo gasto pelo método anterior, porém agora no novo método, pode-se obter um nível muito mais alto de realismo (Figura 1.6).

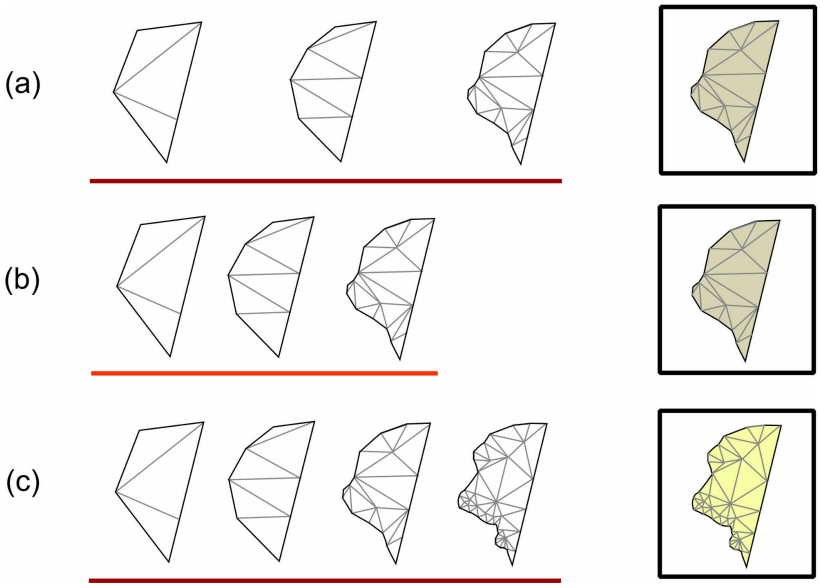


Figura 1.6. Maior desempenho e a relação com o maior realismo.

Na Figura 1.6a, o refinamento da malha por um método existente conduz a um determinado resultado gráfico num certo espaço de tempo. Na Figura 1.6b, um novo método gera o mesmo resultado gráfico da Figura 1.6a, porém usando um espaço de tempo menor. Na Figura 1.6c, o novo método trabalha no mesmo espaço de tempo que o método existente da Figura 1.6a, atingindo, assim, um resultado gráfico final superior.

1.2. Objetivo

A proposta deste trabalho é apresentar um método de refinamento de malhas triangulares que possa usufruir do alto poder computacional de uma GPU, seguindo as tendências recentes da computação gráfica de alto desempenho (Owens et al., 2007). Alguns dos principais recursos das GPUs de última geração são usados, como os relatados em (Blythe, 2006), (Brown & Lichtenbelt, 2006) e (Lichtenbelt & Brown, 2006). O desempenho é um dos principais objetivos do trabalho também, pois, conforme explicado anteriormente, ter um desempenho maior significa poder obter, também, um realismo maior.

Figura 1.7. Exemplo de aplicação do método.

O resultado é uma técnica flexível que permite realizar os mais diversos tipos de refinamento em malha. A Figura 1.7 mostra o resultado geométrico, sem o uso de texturas, de uma aplicação do método feita em tempo real.

1.3. Organização

O restante desta dissertação está organizado da seguinte forma: o Capítulo 2 trata dos trabalhos relacionados e faz uma discussão comparativa entre eles; o Capítulo 3 explica alguns dos principais conceitos envolvidos no método; o Capítulo 4 trata os detalhes da técnica proposta; o Capítulo 5 mostra os resultados, fazendo uma comparação e discutindo o desempenho geral do método; o Capítulo 6 apresenta as conclusões e trabalhos futuros.

Capítulo 2. Trabalhos Relacionados

2.1. Introdução

Nesse capítulo são apresentadas as principais técnicas de refinamento de malhas com GPU. Os trabalhos são apresentados numa ordem cronológica, com uma descrição crítica de cada um enfatizando o paradigma adotado e os problemas encontrados.

2.2. Trabalhos Iniciais

Algumas ideias sobre como se deve trabalhar com o refinamento de malhas em GPU já surgiram com o *framework* teórico proposto por Shiue e seus coautores (Shiue et al., 2003). Os trabalhos iniciais, como o de Bolz e Schröder (Bolz & Schröder, 2003), usavam técnicas de *General-Purpose computing on the GPU* (GPGPU) (Owens et al., 2007), isto é, exploravam o poderio computacional da GPU por meio de técnicas mais gerais simuladas com esse dispositivo, sem, no entanto, fazer uso das especialidades geométricas da mesma. Nesses algoritmos de GPGPU, por vezes, uma malha de vértices é representada numa forma adaptada para uma tabela de pixels, por exemplo, onde cada pixel agrega informações sobre os vértices da malha sendo processada (Figura 2.1). Então, há um processamento em nível de pixels na GPU, por meio do chamado *Pixel Shader*, onde os pixels são manipulados como se fossem meramente dados numéricos, e não intensidades de cor, e assim geram-se os resultados.

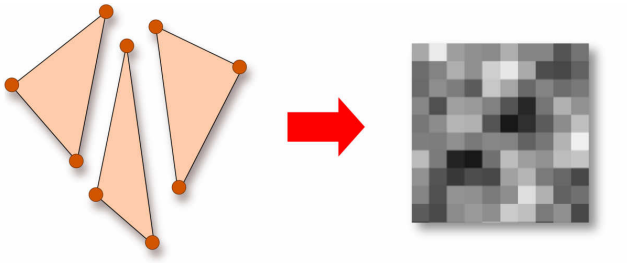


Figura 2.1. Adaptação dos dados nas técnicas de processamento de malha por GPGPU.

Há uma dificuldade inerente nesses métodos na questão de como adaptar as estruturas da malha para esse tipo de formato. O trabalho de Boubekeur e Schlick (Boubekeur et al., 2005b) contém uma lista histórica desses trabalhos iniciais.

2.3. Um Refinamento Local Simples

A tarefa de fazer um refinamento de malha usando as próprias funcionalidades geométricas da GPU, até então ainda não realizada, poderia ser mais fácil de implementar se o tipo de refinamento a ser aplicado fosse relativamente simples. Por exemplo, seria simples fazer uma suavização da malha por meio da interpolação de uma superfície que fosse local (Figura 2.2) e não exigisse nem informações de adjacências nem maiores informações do que apenas vértices e normais, que são elementos já presentes na arquitetura geométrica da GPU.

A superfície de Bézier cúbica triangular do *Curved PN Triangle*, apresentada em (Vlachos et al., 2001), supriu essa necessidade. O *Curved PN Triangle* funciona, na verdade, como uma heurística particular para a determinação de uma superfície de Bézier cúbica triangular geral. Essa heurística usa somente os dados originais de uma malha bruta, num cenário onde há escassez de maiores descrições geométricas do objeto sendo representado. Apenas três vértices { P_1, P_2, P_3 } e as normais correspondentes { N_1, N_2, N_3 } são necessários para o uso dessa superfície (Figura 2.3).

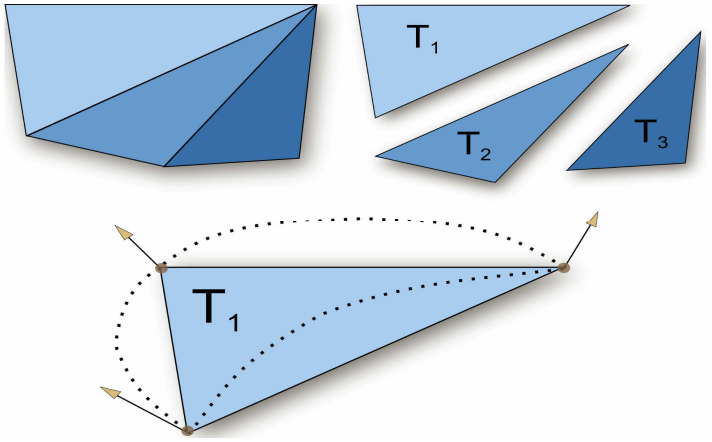


Figura 2.2. Método de refinamento local.

Uma superfície de Bézier cúbica triangular é definida por meio de 10 pontos de controle. Com as 6 informações geométricas S = { P_1, P_2, P_3, N_1, N_2, N_3 } disponíveis para cada elemento da malha triangularizada original, a técnica do *Curved PN Triangle* simplesmente formula como esses 10 pontos de controle devem ser calculados de modo que sejam dependentes somente de S. Além disso, o trabalho original sobre essa técnica especifica uma maneira de realizar um

sombreamento mais suave na malha renderizada por meio de uma interpolação quadrática de normais na superfície ao invés de uma interpolação linear, como tradicionalmente é feito. O resultado disso é uma técnica simples e de grande apelo visual.

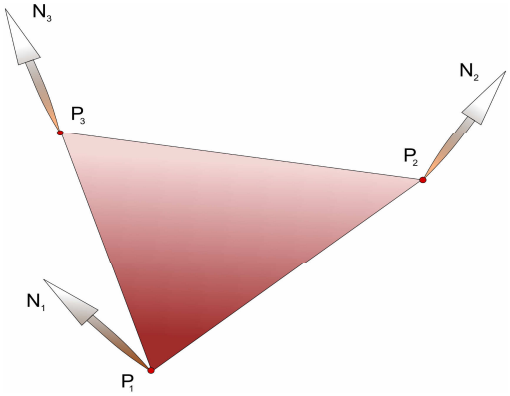


Figura 2.3. Dados básicos exigidos pelo Curved PN Triangle.

A facilidade de implementação dessa superfície em *hardware* logo foi evidenciada, com sua implementação em GPUs (Ati, 2001). Posteriormente, Boubekeur e seus coautores (Boubekeur et al., 2005a) apresentaram uma versão modificada do *Curved PN Triangle*, alterando alguns dos seus coeficientes para permitir novas manipulações da superfície final gerada. Isso foi feito por meio da introdução de três novos parâmetros escalares (*Scalar Tags*) para cada vértice: *sharpness* (σ), *tension* (θ) e *bias* (β). Seu modelo foi chamado, por causa disso, de *ST-Mesh*. Esses parâmetros escalares funcionam em conjunto com um quarto parâmetro ainda, um vetor Δ. As Figuras 2.4 e 2.5 mostram exemplos de manipulação que são possíveis de se fazer usando somente esses parâmetros.

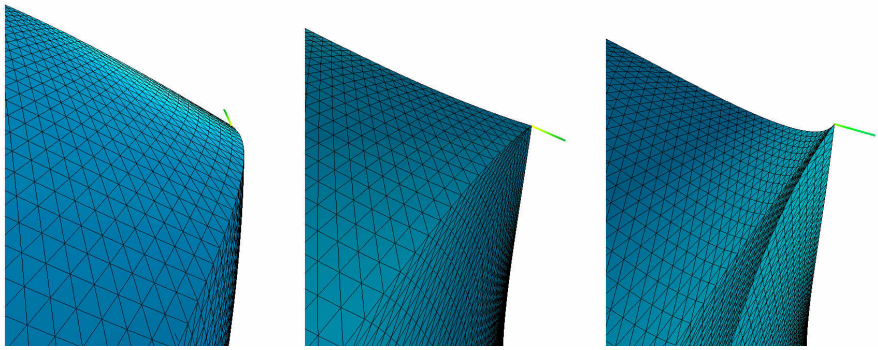


Figura 2.4. Manipulação em conjunto dos parâmetros σ e Δ para determinado vértice.

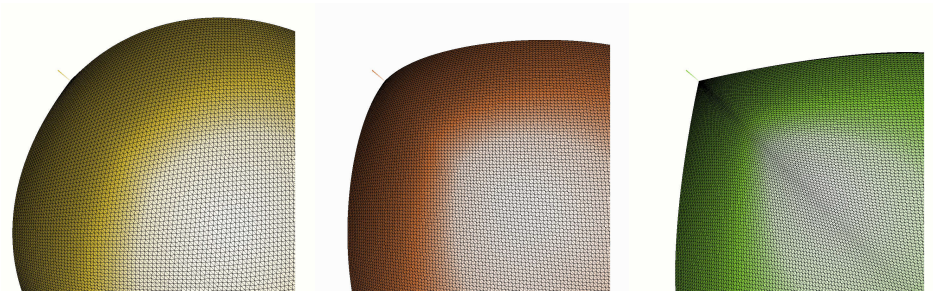


Figura 2.5. Manipulação do parâmetro θ para determinado vértice.

O refinamento local de malhas triangulares, tirando proveito das próprias funcionalidades geométricas da GPU, nasceu usando a técnica de suavização por meio da superfície do *Curved PN Triangle* com *Scalar Tags* (ST-Mesh), no método apresentado por Boubekeur e Schlick (Boubekeur et al., 2005b). Esse método usava padrões topológicos que eram uniformes (uma breve explicação desses conceitos é dada no Capítulo 3). Mais tarde, os mesmos autores apresentaram uma versão melhorada que trabalhava com padrões adaptativos ao invés de uniformes (Boubekeur & Schlick, 2007). Eles chamaram o seu novo método de *Adaptive Refinement Kernel* (ARK).

Recentemente, Lorenz e Döllner (Lorenz & Döllner, 2008) introduziram um outro método de refinamento de malhas triangulares com GPU, chamado de *Dynamic Mesh Refinement* (DMR). Esse método surgiu complementando o método ARK, atuando melhor onde o ARK atuava pior. Contudo, como os autores ressaltam, o DMR foi feito especificamente para complementar o ARK, e não para substituí-lo; sua atuação em certos casos ainda é pior do que a do ARK. Uma análise comparativa entre os métodos ARK e DMR é apresentada na próxima seção.

2.4. ARK vs DMR

O método ARK trabalha realizando uma longa lista de chamadas da CPU à GPU, processando em cada chamada um elemento da malha de entrada. Esse processamento se dá da seguinte forma: a CPU disponibiliza para a GPU os dados do elemento a ser processado e a GPU faz o refinamento local do mesmo. Em termos de implementação, esse método utiliza o *Vertex Shader* para o processamento e variáveis *uniform* para a disponibilização de informações da CPU à GPU. Para um maior aprofundamento nos conceitos básicos de programação com GPU, consultar o excelente livro de Rost (Rost, 2006). Uma rápida introdução, em todo caso, é feita no Capítulo 3.

O método ARK, portanto, funciona com intensa comunicação entre a CPU e a GPU, numa lista de chamadas proporcional ao número de elementos da malha original. Por exemplo, se a malha original tiver 500 elementos, em uma execução completa do ARK ocorrerão 500 *ping-pongs* entre a CPU e a GPU.

Como há certo custo inerente em cada chamada que a CPU faz à GPU, recomenda-se que se minimize o número de tarefas passadas para a GPU e se maximize o tamanho de cada tarefa, a fim de que o processamento realizado compense o custo da própria comunicação. O método ARK é afetado por isso de forma negativa, pois quando se passa uma tarefa muito pequena para a GPU que não compense o custo da própria comunicação, o processo fica mais dispendioso. Isso ocorre quando muitas regiões da malha são pouco discretizadas, pois cada uma dessas regiões acarretará uma tarefa muito pequena para a GPU e, portanto, dispendiosa. Tendo isso em vista, o método DMR trabalha com uma metodologia oposta, onde a CPU faz uma única chamada à GPU e deixa a GPU realizar todo o processamento necessário.

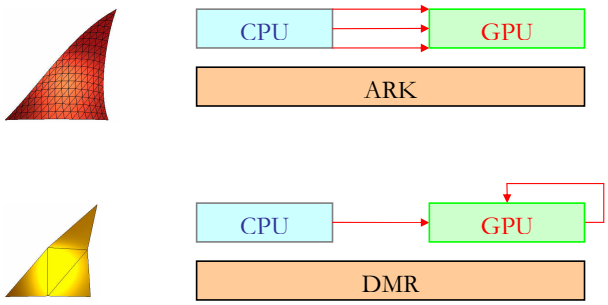


Figura 2.6. Comparação entre os métodos ARK e DMR.

O método DMR utiliza o mecanismo da GPU chamado de *Geometry Shader* (Brown & Lichtenbelt, 2006) que permite realizar uma modesta *amplificação geométrica*. Isso significa que a GPU recebe, de uma vez, uma malha completa com N elementos e gera uma outra malha com kN elementos, onde o fator k é pequeno. Dizendo de outra forma, o *Geometry Shader* permite, por exemplo, realizar um processamento do tipo "para cada triângulo T_i recebido da malha, gere os três novos triângulos T_a, T_b, T_c para a malha resultante". De fato, é precisamente isso que o método DMR faz, gerando assim uma malha resultante composta pelos próprios triângulos dos padrões topológicos.

Porém, foi observado que o desempenho do *Geometry Shader* cai significativamente numa amplificação geométrica de kN elementos quanto maior for o fator k. Isso significa que um limite de discretização da malha é imposto pela GPU; todavia, o método DMR, engenhosamente,

contorna esse problema e consegue atingir maiores níveis de discretização. A ideia é utilizar um refinamento progressivo, que pode receber uma malha previamente refinada e simplesmente adicionar alguns elementos à mesma, melhorando seu nível de refinamento. É possível também, se necessário, desrefinar uma malha, removendo alguns elementos. Além disso, o método trabalha com retro-alimentação, processando novamente as malhas que ele próprio gerou em um passo anterior. Nesse ponto, porém, reside a força e a fraqueza do método DMR. Por um lado, ele processa malhas com muitas regiões de baixa discretização mais rapidamente do que o método ARK; por outro lado, ele poderá gastar muitos passos de execução sobre toda a malha para atingir um nível de discretização mais alto, problema esse que não ocorre com o método ARK. Portanto, pode-se dizer que: o método ARK é mais rápido do que o DMR ao processar malhas com grande discretização, enquanto que, para malhas com baixo nível de discretização, o método DMR é mais rápido do que o método ARK. A Figura 2.6 mostra uma comparação entre ambos, evidenciando que em ARK há muita comunicação entre a CPU e a GPU, enquanto que em DMR há retro-alimentação e maior saturação da GPU.

2.5. SUAPT

Um outro método de refinamento de malhas triangulares com GPU, *Semi-Uniform Adaptive Patch Tessellation* (SUAPT), foi proposto por Dyken e seus coautores em (Dyken et al., 2009). O método usa o recurso de paralelismo das atuais GPUs, chamado de *Instancing*, a fim de aumentar o desempenho quando comparado aos demais métodos.

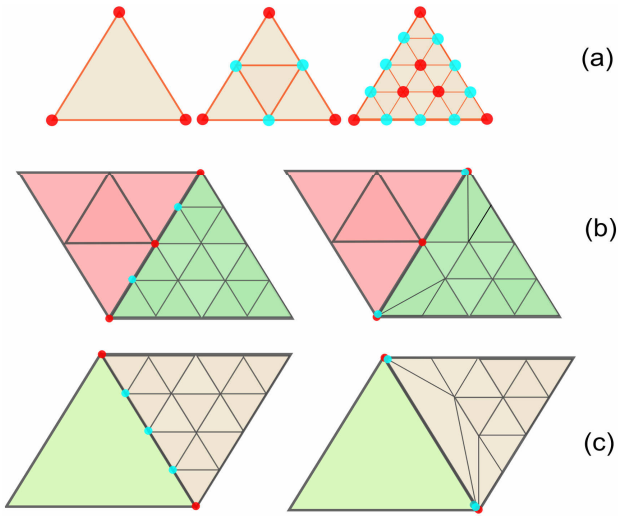


Figura 2.7. Adaptação feita pelo SUAPT na fronteira entre os padrões.

Ao invés de usar padrões adaptativos como nos métodos ARK e DMR, o método SUAPT usa padrões uniformes, mas o faz com uma rotina na GPU que realiza, em tempo de execução, a devida adaptação entre uma região e outra (Figura 2.7), o que mantém a malha conforme. Nessa figura, os pontos indicados em azul são devidamente movidos e assim ocorre a adaptação.

Essa adaptação feita na malha final garante a compatibilidade entre todas as regiões, mesmo que tenham discretizações drasticamente diferentes. A ideia é fazer um reposicionamento dos vértices da fronteira, movendo os pontos da região que tem maior discretização para se encaixar com os pontos da região de menor discretização. Nesse reposicionamento surgem triângulos que devem ser descartados. Há, porém, um outro problema mais sério, inerente à própria natureza do método: a geração de malha resultante do método SUAPT exibe, por vezes, regiões de transição muito descontínuas, como relata Moreton (Moreton, 2001). Isso ocorre porque a região que é adaptada entre uma face e outra é uma região pequena (Figura 2.8a); em comparação, a região que poderia ser adaptada caso se utilizasse padrões adaptativos previamente computados, levando isso em consideração, seria muito maior (Figura 2.8b). Em ambas as figuras, os pontos indicados em azul podem mudar de posição.

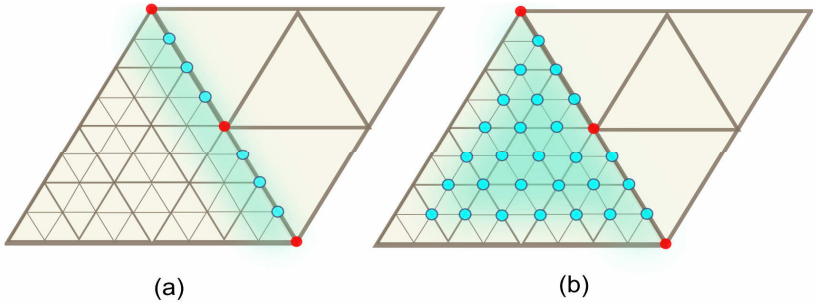


Figura 2.8. O problema das regiões de transição muito descontínuas.

A vantagem do método SUAPT, todavia, está no maior paralelismo que se obtém quando se usa *Instancing* com o pequeno número de padrões topológicos uniformes. Deve-se lembrar que, para um refinamento uniforme da malha, tem-se um número muito menor de padrões topológicos do que para um refinamento adaptativo (este permanece como N^3 nos métodos ARK e DMR). No método SUAPT, o número de chamadas da CPU à GPU é proporcional ao número de padrões topológicos. Isso ficará mais claro com as explicações do método proposto neste trabalho.

2.6. Considerações Finais

Esse capítulo apresentou uma visão geral sobre os trabalhos da área. Os principais métodos são o ARK, DMR e SUAPT. O ARK é uma evolução de um trabalho anterior, e seu paradigma se mostrou um terreno rico para exploração. O DMR surgiu depois do ARK, procurando explorar um ponto onde o ARK era fraco. O SUAPT, também, surgiu depois do ARK, e explorou melhor um recurso de paralelismo que estava disponível nas GPUs mais novas.

Nesta dissertação, também, é feita uma exploração do paradigma utilizado pelo ARK, resultando num novo método. Mas, antes de discutir sobre isso, é necessário primeiro definir alguns conceitos importantes. Isso será feito no próximo capítulo.

Capítulo 3. Conceitos Preliminares

3.1. Introdução

Este capítulo introduz os principais conceitos de base envolvidos nesta dissertação. Para a compreensão completa do método proposto, é necessário estabelecer antes os conceitos geométricos usados e os conceitos técnicos de GPU que serviram de meio para a implementação dos métodos relacionados. É apresentada uma pequena introdução à arquitetura programável da GPU e em seguida são apresentados os conceitos geométricos. A GPU é apresentada em primeiro lugar porque alguns dos conceitos geométricos apresentados posteriormente fazem referências a certos aspectos técnicos da GPU.

3.2. Mecanismos Programáveis da GPU

As GPUs possuem uma arquitetura de forte paralelismo, e foram especificamente projetadas para os processamentos típicos da computação gráfica em tempo real. Embora, no passado, as GPUs basicamente fizessem seu trabalho sobre a tradicional *pipeline* gráfica (Figura 3.1) de um modo fixo e bem estabelecido, esse antigo paradigma tem sido modificado, e já há alguns anos é possível usar instruções de GPU para escrever um programa que será executado em certos trechos da *pipeline* gráfica (Figura 3.2). O fato de que esses trechos são programáveis dá ao programador total controle sobre eles.

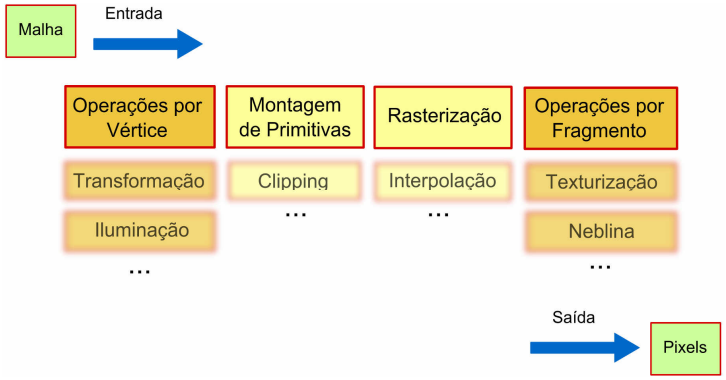


Figura 3.1. Uma pipeline gráfica tradicional.

Na pipeline gráfica moderna (Figura 3.2) esses trechos programáveis são chamados de *Vertex Shader* e de *Fragment Shader*. O programa executado no *Vertex Shader* faz uma manipulação local dos vértices da malha, e é executado isoladamente para cada vértice v_i a ser processado. Se uma malha contém N vértices, então N execuções do programa associado ao *Vertex Shader* serão realizadas para essa malha em questão. Em cada execução, o programa recebe como entrada um vértice bruto da malha e devolve esse vértice devidamente processado. Atributos, como posição, normal e cor, podem ser modificados livremente dessa forma.

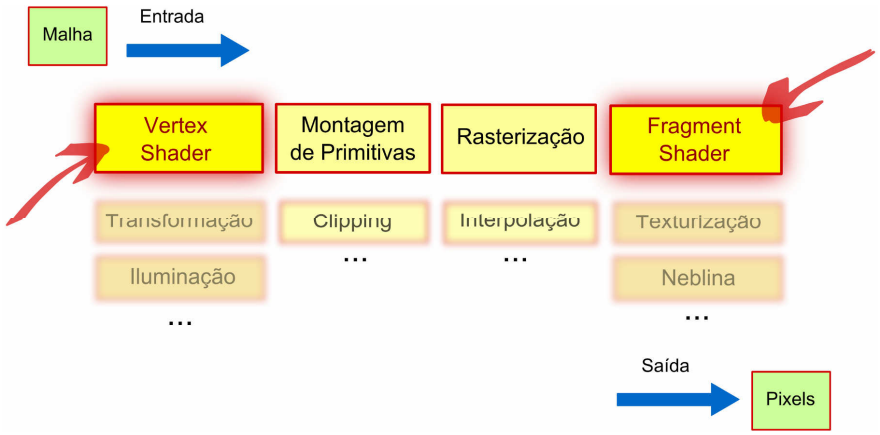


Figura 3.2. Versão moderna de uma pipeline gráfica com trechos programáveis.

O outro trecho programável da *pipeline* gráfica, o *Fragment Shader*, também conhecido como *Pixel Shader*, é acionado quando a malha já se encontra em vias finais de ser renderizada na tela. O programa associado ao *Fragment Shader* é executado isoladamente para cada pixel gerado pela renderização da malha. Se essa renderização gerar N pixels, então N execuções do programa associado ao *Fragment Shader* serão realizadas. Em cada execução, o programa recebe como entrada o pixel em questão e retorna sua cor. Alguns cálculos de iluminação e interpolação de cores geralmente ocorrem aqui.

Na Figura 3.3, são evidenciadas as diferenças entre ambos os mecanismos citados. Na Figura 3.3a, a malha original entra na GPU para ser devidamente processada. Na Figura 3.3b, a malha passa pelo *Vertex Shader*, onde cada um de seus vértices é processado de forma local e pode ter seus atributos modificados. Na Figura 3.3c, a renderização da malha é feita, pixel-a-pixel, pelo *Fragment Shader*.

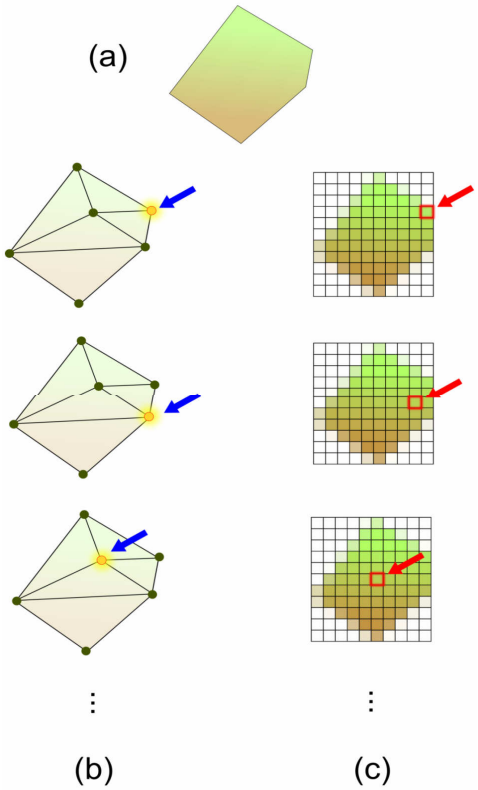


Figura 3.3. Processamento de uma malha geométrica na GPU.

3.3. Processamento de Elementos da Malha na GPU

Os mecanismos programáveis da GPU apresentados na seção anterior foram, por vários anos, os únicos meios de programação para a GPU. Porém, foi introduzido posteriormente, na chamada 4ª geração de GPUs (Blythe, 2006), um terceiro mecanismo programável, conhecido como *Geometry Shader* (Brown & Lichtenbelt, 2006). Esse mecanismo permitiria um novo tipo de processamento geométrico.

O *Geometry Shader* é introduzido na *pipeline* gráfica depois do *Vertex Shader*. Sua unidade básica de processamento é um conjunto completo de vértices que definem uma primitiva gráfica. Por causa disso, afirma-se que o *Geometry Shader* processa primitivas, como triângulos, ou elementos de uma malha triangularizada.

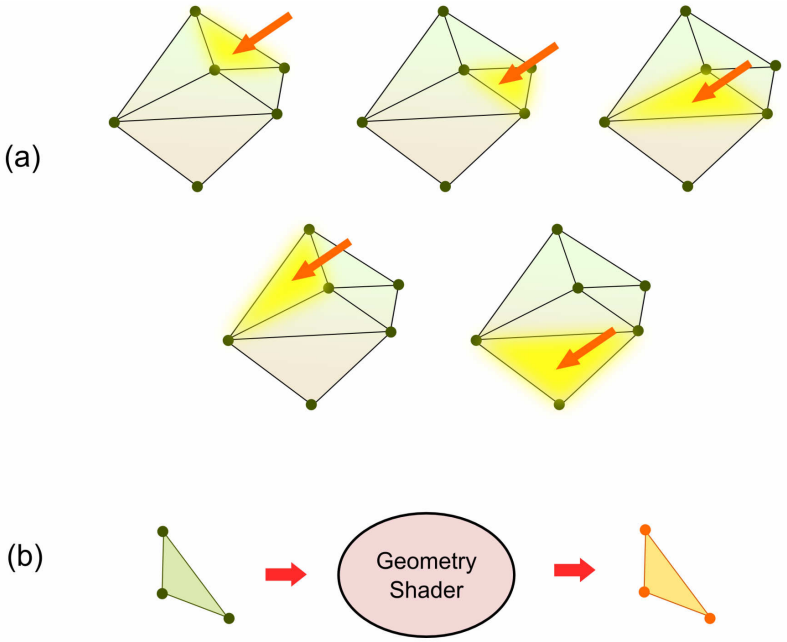


Figura 3.4. Processamento de elementos da malha por meio do Geometry Shader.

Assim, cada elemento de uma malha passa individualmente pelo *Geometry Shader* (Figura 3.4). Se uma malha contém N elementos, então N execuções do programa associado ao *Geometry Shader* serão realizadas (Figura 3.4a). Em cada execução, o programa recebe o elemento todo como entrada, e deve devolver o elemento todo na saída, possivelmente modificado (Figura 3.4b). Por exemplo, um elemento triangular é fornecido na forma de um conjunto de 3 vértices; então o programa em questão pode modificar livremente qualquer um dos três vértices. Ele o faz, contudo, tendo acesso a todos os três vértices de uma só vez (no *Vertex Shader*, por outro lado, apenas o vértice sendo processado era acessível).

A síntese ou amplificação geométrica (isto é, a operação que, dada uma malha com N vértices, retorna uma malha com N+M vértices) torna-se possível de ser feita por meio do *Geometry Shader* por causa de uma regra na especificação desse mecanismo: para cada elemento da malha sendo processado, o programa pode, na verdade, devolver zero ou vários elementos como resposta. Isso significa, por um lado, que uma malha pode ter elementos removidos ao passar pelo *Geometry Shader* (bastando, para isso, não retornar elemento algum nos casos desejados), e, por outro, significa que cada elemento processado pode ser substituído por outros K novos elementos (para isso, bastaria retornar esses novos K elementos numa dada execução).

Porém, observou-se que o *Geometry Shader* não escalona muito bem na amplificação geométrica; seu desempenho cai rapidamente para uma amplificação mais intensa (Figura 3.5). Por causa disso, o uso do *Geometry Shader* em síntese geométrica tem sido limitado às amplificações geométricas de pequena escala.

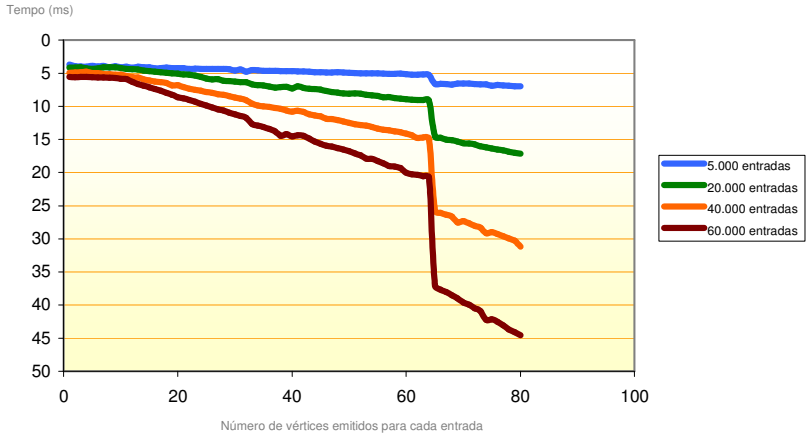


Figura 3.5. Amplificação geométrica por meio do Geometry Shader cai bruscamente.

3.4. Instancing

O *Instancing* é um recurso introduzido, também, na 4ª geração de GPUs (Blythe, 2006), motivado por um problema bastante comum. Muitas aplicações gráficas comumente têm uma mesma malha geométrica que é replicada, diversas vezes, em um vasto cenário, em posições e tamanhos distintos (Figura 3.6a). Cada replicação, ou instância, da malha geométrica em questão, envolve um determinado custo operacional para ser processada.

Na Figura 3.6b, são enumeradas todas as instâncias da malha sendo replicada. Tradicionalmente, a aplicação teria que processar, por si própria, cada instância em particular, definindo sequencialmente qual malha usar e processando as transformações geométricas particulares de cada caso. Porém, com o *Instancing*, esse processamento de replicação é automatizado e passa a ser feito em paralelo, dentro da própria GPU. Isso minimiza consideravelmente o custo operacional das replicações.

Ativando o recurso do *Instancing*, uma variável que identifica qual é a instância atual *i* pode ser usada no *Vertex Shader*. Através dessa variável, as propriedades particulares para uma determinada instância poderão ser acessadas. Por exemplo, para aplicar uma transformação

geométrica particular para a instância *i*, basta acessar a matriz de índice *i* numa lista de matrizes. Essa ideia é ilustrada na Figura 3.7.

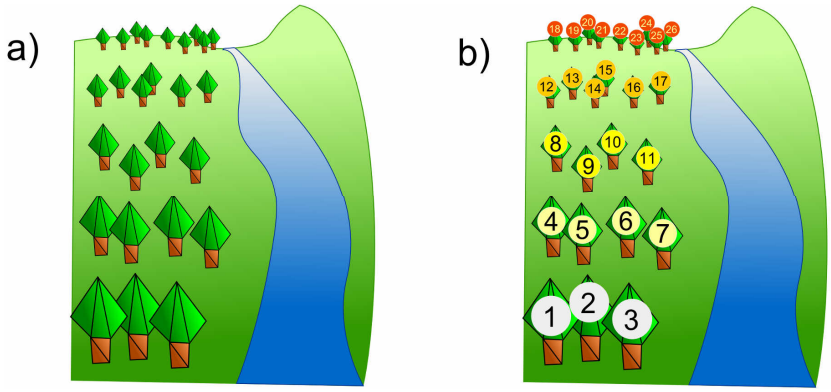


Figura 3.6. Várias instâncias de uma mesma malha geométrica sendo replicada.

Deve-ser observar que, na verdade, não há nenhum "efeito gráfico" a mais, usando *Instancing*, que não possa ser obtido sem usá-lo também. O objetivo dessa técnica é, portanto, simplesmente a otimização do processamento da malha.

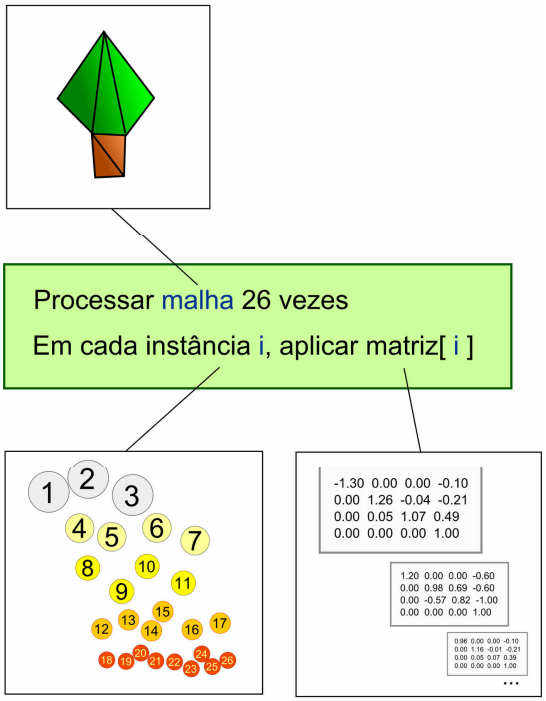


Figura 3.7. Processamento particular para cada instância sendo replicada.

Com essa discussão sobre os conceitos técnicos da GPU finalizada, os conceitos de malha e geometria relacionados com esta dissertação são agora devidamente apresentados.

3.5. Refinamento de Malhas

De acordo com a terminologia de (Shiue et al., 2003), uma malha pode ser vista como um tipo especial de grafo onde os nós carregam informações sobre os atributos dos vértices, como posição, normal, etc. Dentro dessa perspectiva, o refinamento da malha seria visto simplesmente como uma inserção de nós no grafo da malha; e o chamado ajuste da malha seria uma manipulação dos atributos que estão nos nós (em especial, o atributo da posição do vértice).

Por uma questão de praticidade, no entanto, o refinamento da malha é designado, aqui, como o conjunto completo dessas operações: a fase de inserir nós e a fase de manipular nós. O modo como os nós são manipulados, por sua vez, é uma característica do tipo de refinamento usado. Cada tipo de refinamento trabalha de um modo particular para determinar como devem ser manipulados os nós (por exemplo, usando uma função que interpole uma superfície).

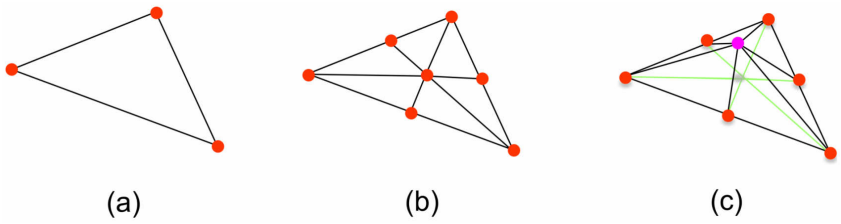


Figura 3.8. Refinamento da malha: inserção e manipulação de nós.

A Figura 3.8 apresenta o conceito de refinamento de malha discutido. Na Figura 3.8a, tem-se a malha original; na Figura 3.8b, a malha após a inserção de nós, e na Figura 3.8c, a malha após o ajuste do nó central. Na terminologia empregada aqui, o refinamento da malha é o processo ilustrado nas Figuras 3.8b e 3.8c. A função que descreve como serão manipulados os nós é determinada pelo tipo de refinamento e seria indicada na Figura 3.8c.

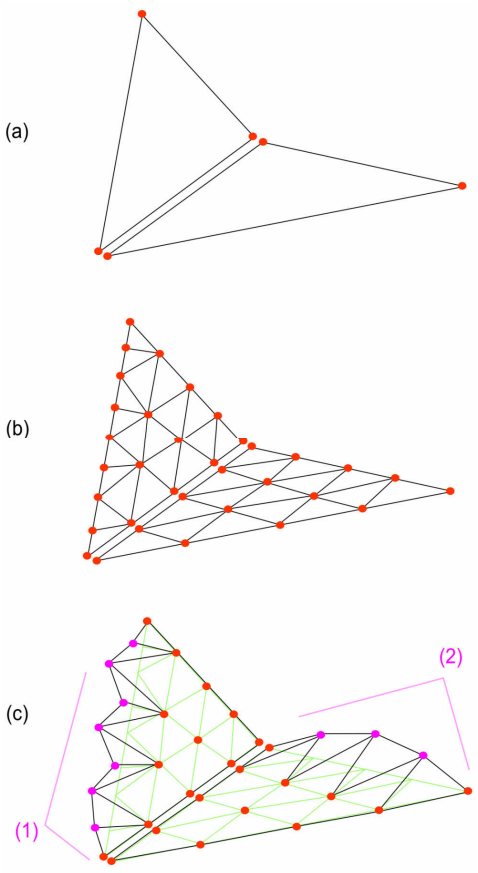


Figura 3.9. Exemplo de refinamento de malha.

Na Figura 3.9, esse conceito é aplicado numa malha ilustrativa onde há uma discretização maior. Na Figura 3.9a, tem-se a malha original. Na Figura 3.9b, essa malha é modificada pela inserção de vários nós. Essa fase de inserção de nós pode ser chamada de subdivisão ou discretização da malha, em contraste com a malha original (que ainda não foi subdividida ou discretizada). Na Figura 3.9c, vê-se a fase de manipulação dos nós da nova malha discretizada. Em especial, nessa figura, há duas manipulações sendo realizadas: a primeira aplica um ruído na posição dos vértices indicados (1) e a segunda aplica uma equação de uma curva na posição dos vértices indicados (2).

Em uma técnica de refinamento de malhas, há duas preocupações: a de como fazer a subdivisão da malha e a de como permitir que uma função de manipulação de nós trabalhe em

cima da malha discretizada, buscando montar uma infraestrutura que permita que essas operações possam ser feitas da forma mais eficiente possível. Portanto, o refinamento de malhas com GPU seria simplesmente buscar fazer isso usando bem os recursos disponíveis numa GPU.

3.6. Subdivisão da Malha na GPU

A unidade básica para trabalho geométrico na GPU é o triângulo. Considerando, portanto, que cada elemento da malha de entrada a ser trabalhada consiste num triângulo, a subdivisão de uma malha significa, essencialmente, subdividir cada triângulo da mesma. Esse processo de subdivisão de triângulos, portanto, seria a primeira tarefa com que se preocupar (Figura 3.10).

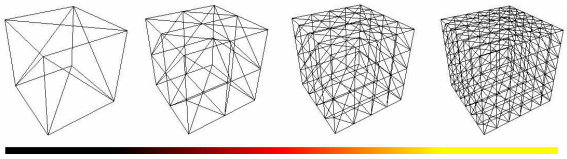


Figura 3.10. Subdivisão gradual de uma malha.

Porém, devido a um problema de restrição da arquitetura das GPUs de outrora, isso não podia ser feito diretamente. As GPUs mais antigas não tinham um recurso de programação que permitisse realizar a subdivisão de um triângulo fornecido como entrada. Os recursos programáveis de então se resumiam ao *Vertex Shader* e ao *Pixel Shader*, já apresentados na Seção 3.2. Não havia ainda, nesse paradigma de programação com a GPU, um mecanismo que permitisse realizar uma síntese geométrica (o *Geometry Shader* ainda não havia sido introduzido). Por causa disso, um outro caminho teve que ser escolhido para realizar a subdivisão das malhas.

Diante desse impasse foi adotada uma solução engenhosa que explorava bem o potencial da GPU. A ideia é que, ao invés de procurar subdividir cada elemento de uma malha existente, utilizam-se elementos já prontos e subdivididos, num formato genérico, e faz-se a devida adaptação desses elementos, em tempo real, para a malha em particular. Esses elementos genéricos, já prontos, são chamados de padrões topológicos (vide Seção 3.7).

Vale lembrar que, mesmo após a introdução do *Geometry Shader* nas novas GPUs, a síntese geométrica que ele era capaz de realizar limitava-se a exemplos de pequena escala, pois o *Geometry Shader* não escalona bem para amplificações geométricas mais intensas (vide discussão na Seção 3.3). Por causa disso, a solução engenhosa mencionada anteriormente continua sendo de grande valia.

3.7. Padrões Topológicos

Os trabalhos de refinamento de malha triangular com GPU analisados utilizam primariamente padrões topológicos de refinamento, sendo por isso chamados de métodos de refinamento orientado a padrões (*pattern-based refinement methods*).

Um padrão topológico é um triângulo subdividido, com vértices em coordenadas baricêntricas (u, v, w), como mostra a Figura 3.11. Como os vértices estão em coordenadas baricêntricas, esse triângulo subdividido funciona, essencialmente, como um padrão de topologia: a aplicação dele a um triângulo T qualquer faz com que T fique discretizado de acordo com o padrão empregado (Figura 3.12).

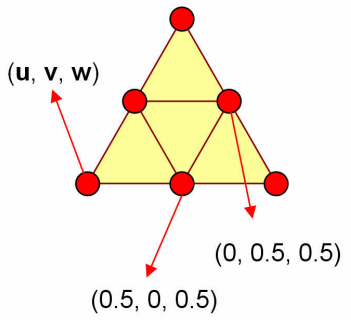


Figura 3.11. Exemplo de um padrão topológico.

O padrão topológico obedece a uma certa discretização, que pode ser especificada por um valor dado em cada aresta. Por exemplo, uma aresta com nível de discretização $d = 2$ deve ser subdividida ao meio recursivamente duas vezes. A notação $[d_1, d_2, d_3]$ pode ser usada para se referir ao padrão topológico que usa o nível de discretização d_1 para sua primeira aresta, d_2 para a segunda e d_3 para a terceira. No exemplo da Figura 3.11, o padrão topológico em questão seria o [1, 1, 1].

Um padrão topológico é dito *uniforme* quando se mantém a restrição de que os níveis de discretização em suas arestas sejam todos iguais ($d_1 = d_2 = d_3$). Quando não há essa restrição, o padrão é considerado *adaptativo*. Pode-se considerar que todo padrão uniforme seja um caso particular de um padrão adaptativo. Além disso, um método de refinamento de malhas é dito uniforme quando o mesmo trabalha somente com padrões uniformes, e é considerado adaptativo quando se permite trabalhar com padrões adaptativos.

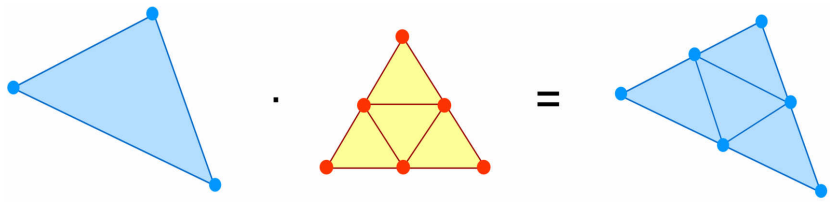


Figura 3.12. Aplicação de um padrão topológico a um triângulo qualquer.

É possível montar uma lista de padrões topológicos em sucessivas discretizações. Por exemplo, usando-se somente padrões uniformes, essa lista poderia ser montada com os padrões [0, 0, 0], [1, 1, 1], [2, 2, 2], [3, 3, 3], etc. Dessa forma, para realizar um refinamento mais intenso, se utilizaria um padrão de discretização maior, como [3, 3, 3]; para um refinamento menos intenso, um padrão de discretização menor, como [1, 1, 1].

Na verdade, a ideia de ter uma lista de padrões topológicos pronta, para toda uma gama de discretizações possíveis, é vital para os métodos de refinamento de malha com GPU. Isso ocorre porque cada padrão topológico desses pode ser devidamente alocado na memória da GPU e então reutilizado tantas vezes quanto forem necessárias.

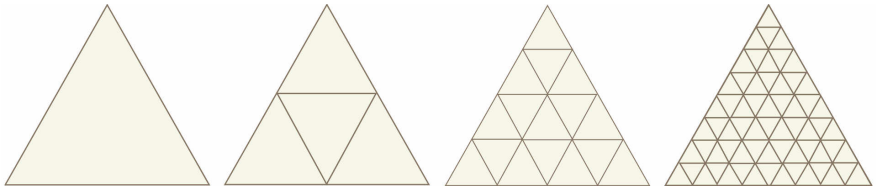


Figura 3.13. Lista de padrões topológicos uniformes.

A Figura 3.13 mostra uma lista de padrões topológicos em sucessivas discretizações, e a Figura 3.14 mostra um exemplo de manipulação de malha, onde a mesma malha inicial, com os mesmos parâmetros de manipulação da silhueta, apresenta resultados diferentes de refinamento usando um padrão de baixa discretização e um padrão de alta discretização.

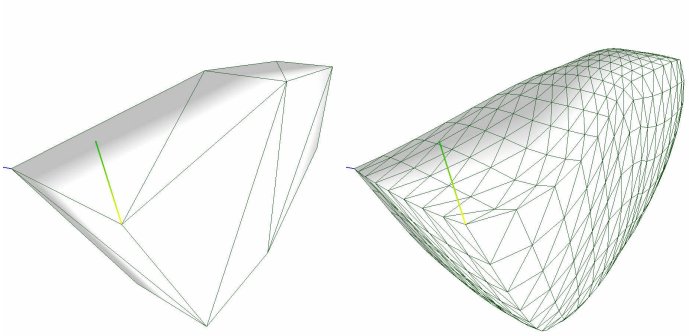


Figura 3.14. Manipulação de uma malha usando padrões de diferentes discretizações.

3.8. Seleção de Padrões

O fato de haver agora uma lista pronta de padrões topológicos à disposição na GPU significa que, para o método de refinamento de malhas, basta escolher quais padrões deverão ser usados na malha em questão. Isso significa que, para cada triângulo da malha de entrada, deve ser selecionado um padrão topológico que se enquadre na discretização desejada para aquele triângulo em particular. Esse processo é chamado de *seleção de padrões* (Figura 3.15).

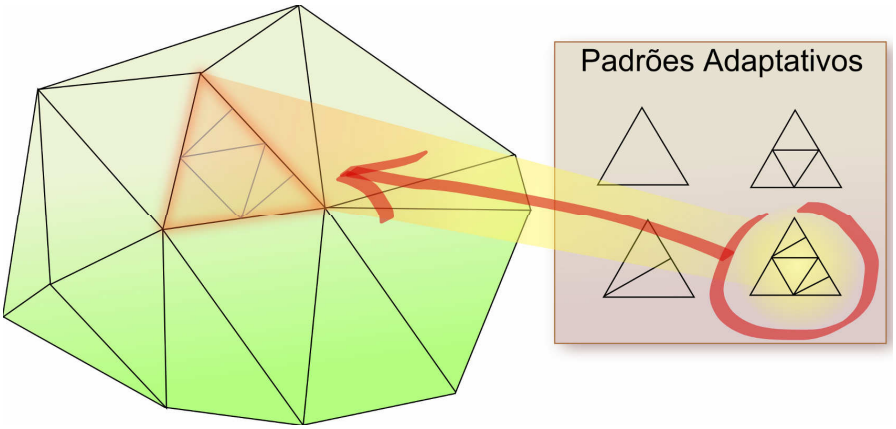


Figura 3.15. Seleção de padrões.

Lembrando que um padrão topológico é referido por uma tupla $[d_1, d_2, d_3]$, onde d_i especifica o nível de discretização para a aresta i, para determinar qual padrão selecionar para um certo elemento da malha de entrada é simples: bastaria introduzir os atributos d_1, d_2 e d_3 nas

respectivas arestas da malha de entrada e então usar esses valores para determinar qual padrão topológico escolher.

Porém, em termos de implementação nos sistemas gráficos atuais, é mais simples ter um atributo para cada vértice da malha do que para cada aresta. Isso significa que é mais fácil ter uma tupla (d_{v1}, d_{v2}, d_{v3}) que defina o nível de discretização em cada um dos vértices v_1, v_2 e v_3 do que a tupla (d_{a1}, d_{a2}, d_{a3}) que defina o nível de discretização em cada uma das arestas a_1, a_2 e a_3.

Por causa disso, um modelo simples de conversão pode ser adotado: quando um triângulo for processado, obtém-se a tupla (d_{a1}, d_{a2}, d_{a3}) em função de (d_{v1}, d_{v2}, d_{v3}) fazendo-se a média entre os vértices, de acordo com as equações 3.1, 3.2 e 3.3.

$$d_{a1} = \frac{d_{v1} + d_{v2}}{2} \quad \textbf{(3.1)}$$

$$d_{a2} = \frac{d_{v2} + d_{v3}}{2} \quad \textbf{(3.2)}$$

$$d_{a3} = \frac{d_{v3} + d_{v1}}{2} \quad \textbf{(3.3)}$$

As equações 3.1, 3.2 e 3.3 determinam o nível de discretização nas arestas de um triângulo em função de seus vértices. Com a tupla (d_{a1}, d_{a2}, d_{a3}) obtida, o padrão a ser usado é o referido justamente por [d_{a1}, d_{a2}, d_{a3}]. Esse processo de conversão é ilustrado na Figura 3.16.

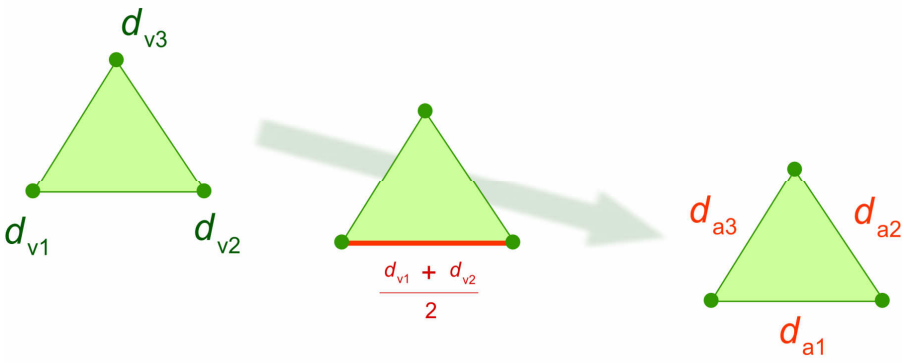


Figura 3.16. Conversão do valor de discretização dado em vértices para arestas.

Com essa conversão, tem-se a infraestrutura básica para seleção de padrões topológicos: entra-se com uma malha, onde cada vértice contém um atributo extra de discretização (chamado na literatura por *depth-tag*), faz-se a conversão mostrada acima e obtém-se a tupla final [d_1, d_2, d_3],

que é usada para selecionar o padrão topológico correspondente. A questão, agora, é como determinar esse atributo de discretização, *depth-tag*, de cada vértice.

A escolha de um nível de discretização para um dado vértice pode ser feita em função de diferentes critérios. Por exemplo, para as regiões da malha que estão mais próximas do observador, adota-se um nível de discretização maior. Outro critério seria avaliar a curvatura em cada ponto da malha para determinar o quão profundo deve ser o nível de discretização na região correspondente. Para isso, há diversos operadores de curvatura disponíveis. No caso das malhas triangulares usadas em jogos e outros sistemas gráficos interativos, um operador simples como o de curvatura Gaussiana discreta (Meyer et al., 2002) pode ser usado. Dyken e seus coautores (Dyken et al., 2007) propõem ainda um critério que discretiza mais fortemente todo elemento da malha que esteja bem próximo de sua borda visual na tela. O efeito disso é que a malha fica com um contorno visual muito bem refinado.

Em todo caso, a especificação dos *depth-tags* da malha de entrada pode ser feita de forma automática por um algoritmo. Esse algoritmo pode ser executado na própria CPU ou, se for suficientemente simples de se adaptar para a arquitetura da GPU, pode ser realizado também nela, por meio do *Vertex Shader*, por exemplo.

3.9. Conformidade da Malha

No processo de seleção de padrões, como se dá a questão de conformidade na malha final? Não se pode selecionar um padrão topológico para um elemento que tenha um número diferente de pontos amostrais na aresta em que esse mesmo elemento compartilha com um elemento vizinho (Figura 3.17). Na Figura 3.17b, há um caso de não-conformidade na malha gerada, e, na Figura 3.17a, uma versão conforme da mesma.

Esse problema, porém, não afeta o método em questão por causa do modo como o padrão topológico é selecionado. O nível de discretização *depth-tag* fornecido nos próprios vértices de um elemento da malha não é usado diretamente, mas indiretamente: uma média dos mesmos é calculada para ser usada como se fosse um atributo que especifica a discretização geral da própria aresta. Nesse caso, uma aresta compartilhada por dois triângulos liga-se aos mesmos vértices, e, por isso, terá o mesmo valor médio de discretização. Assim, os padrões topológicos são selecionados de tal forma que um elemento sempre "casa" com o vizinho. Ou seja, sempre o mesmo número de pontos amostrais será utilizado para a aresta compartilhada entre dois elementos vizinhos (Figura 3.18).

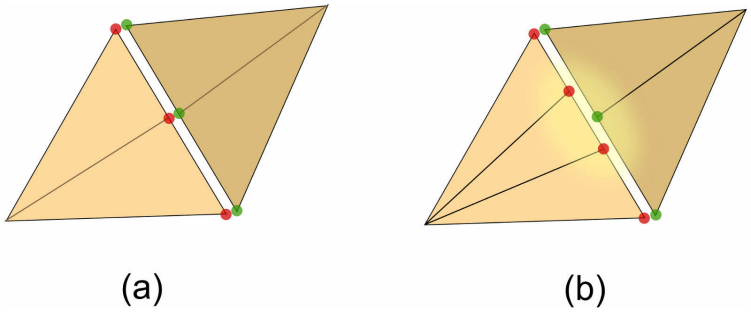


Figura 3.17. Malha conforme e malha não conforme.

Assim, na seleção de padrões, a malha resultante é mantida conforme porque a escolha dos padrões se dá, em última instância, em nível de aresta. Por exemplo, na Figura 3.18b, à direita, um triângulo que use o padrão [0, 1, 0] encaixa-se perfeitamente com um vizinho seu que usa o padrão [1, 1, 1], pois ambos têm, na aresta compartilhada, o mesmo nível $d = 1$ de discretização. Ainda na Figura 3.18, em (a), tem-se vários exemplos de padrões topológicos adaptativos. Todos eles podem ser devidamente encaixados uns nos outros quando possuem em comum uma aresta com o mesmo nível de discretização.

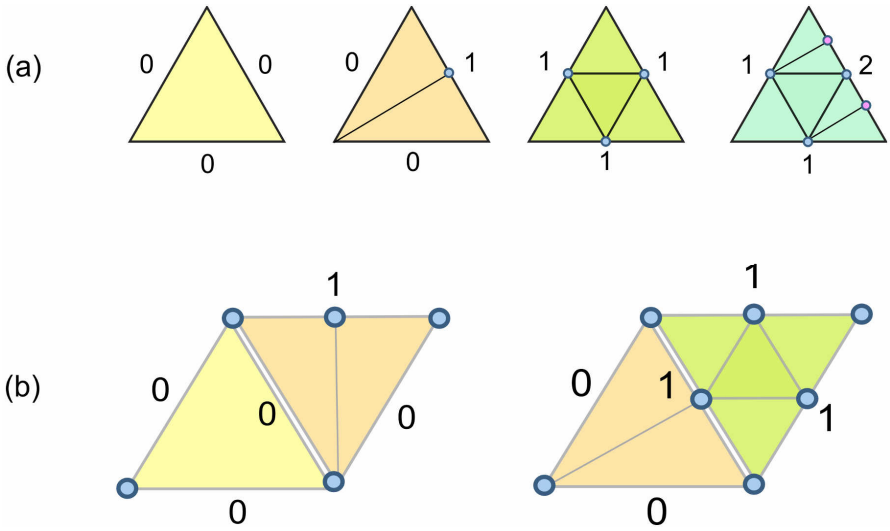


Figura 3.18. Diversos padrões adaptativos e a conformidade da malha.

3.10. Aplicação do Padrão

Após a seleção de padrões, como os vértices dos triângulos de cada padrão estão em coordenadas baricêntricas (u, v, w), o método de refinamento faz um mapeamento simples para se obter as coordenadas finais. Considerando (P_1, P_2, P_3) como as coordenadas dos vértices do triângulo original sendo processado, o mapeamento realizado pela equação 3.4, por exemplo, pode ser usado para gerar uma simples discretização para cada triângulo da malha original, de acordo com a topologia dos padrões selecionados. Essa operação está ilustrada na Figura 3.12.

$$V_{final} = P_1 u + P_2 v + P_3 w \qquad \textbf{(3.4)}$$

3.11. Considerações Finais

Esse capítulo introduziu os principais conceitos de base necessários para um entendimento do método. Foram vistos os mecanismos programáveis da GPU usados pelos métodos de refinamento da área, como o *Vertex Shader* e o *Geometry Shader*. Foi feito também um maior embasamento teórico sobre o que significa realmente o refinamento de malhas e como funciona, de maneira geral, um algoritmo de refinamento de malhas com GPU. Os padrões topológicos e a notação correspondente foram também apresentados, sendo discutido inclusive como eles são aplicados na malha. Resta, agora, a discussão técnica do próprio método proposto. É esse o material de discussão do próximo capítulo.

Capítulo 4. Método de Refinamento Adaptativo de Malhas com Instancing e menos Padrões

4.1. Introdução

O método proposto no presente trabalho refina adaptativamente malhas triangulares, usando padrões topológicos adaptativos. O número de padrões adaptativos é bastante reduzido com a introdução de um novo esquema de indexação dos padrões. A redução do número de padrões não melhora somente a questão do espaço de memória utilizado, mas permite ainda, por causa da maior reutilização de dados, que se obtenha um paralelismo mais intenso, aumentando o desempenho significativamente. A Figura 4.1 delineia uma visão geral do algoritmo.

1. Gerar os padrões topológicos adaptativos e carregá-los na GPU (isso é feito somente uma vez).

2. Realizar a seleção de padrões, associando, para cada elemento da malha original, determinado padrão topológico [i, j, k].

3. Agrupar todos os elementos da malha que referenciam o mesmo padrão topológico [i, j, k] numa lista, de modo a obter *N* listas, onde *N* é o número de padrões topológicos usados.

4. Renderizar cada agrupamento da etapa 3 usando Instancing e fazer o devido mapeamento de cada elemento em tempo de execução.

Figura 4.1. Uma visão geral do método proposto.

A geração dos padrões topológicos adaptativos (etapa 1 do algoritmo delineado) é feita *off-line*, de modo que seu tempo de execução não causa nenhum impacto sobre o desempenho do método de refinamento. Apenas os dados gerados por essa etapa, isto é, os padrões topológicos propriamente ditos, é que são importantes. Eles podem ser gerados uma única vez num programa à parte e gravados em disco para posterior utilização pelo método. Assim, toda vez que o método for inicializado, basta copiar esses dados armazenados no disco para a memória da GPU.

As próximas seções deste capítulo discutem as outras partes do algoritmo delineado, como a seleção de padrões, os agrupamentos e a renderização. Uma das ideias centrais do método está na indexação de padrões que é realizada, e, por causa disso, a matéria relacionada a essa parte é discutida primeiro. As consequências dessa indexação de padrões aparecem em seguida.

4.2. Obtenção de Novos Padrões

Um certo padrão [i, j, k] pode, facilmente, ser usado como se fosse o padrão [j, i, k], por exemplo. Isso ocorre porque a equação de mapeamento das coordenadas baricêntricas (Equação 3.4) pode ser facilmente modificada para trabalhar com esses valores numa ordem diferente. Ao invés de se associar P_1 com *u*, P_2 com *v* e P_3 com *w*, pode-se associar P_1 com *w*, P_2 com *v* e P_3 com *u*, por exemplo. O efeito disso na prática é como se o padrão [j, i, k] tivesse sido utilizado ao invés do padrão [i, j, k]. Por causa disso, ao invés de se ter as 6 permutações distintas de [i, j, k] armazenadas como padrões topológicos distintos (Figura 4.2a), pode-se armazenar um único padrão desses (Figura 4.2b) e os demais são obtidos por meio da permutação das variáveis (u, v, w) do padrão armazenado.

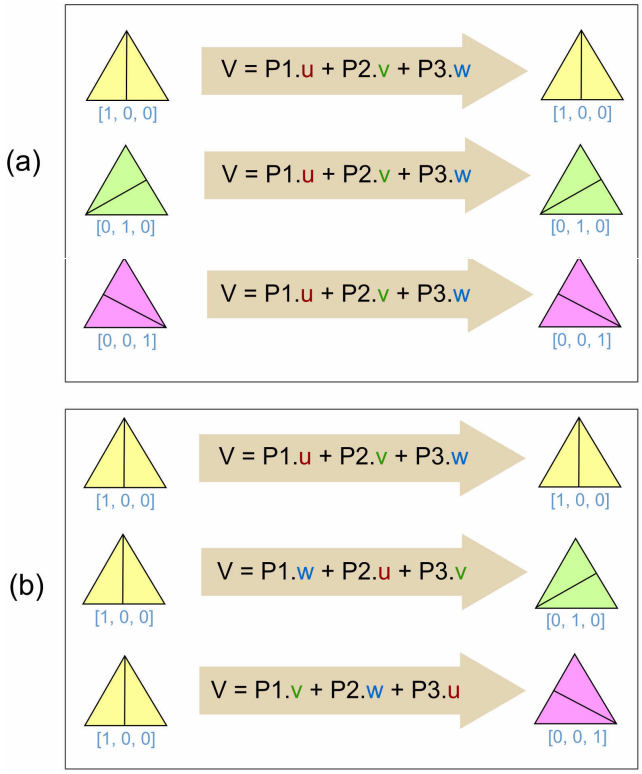


Figura 4.2. Permutação das coordenadas baricêntricas (u, v, w).

Para compreender como devem ser feitas as permutações das coordenadas baricêntricas (u, v, w) a fim de se obter os padrões desejados em função dos padrões armazenados, são ilustrados em seguida alguns exemplos.

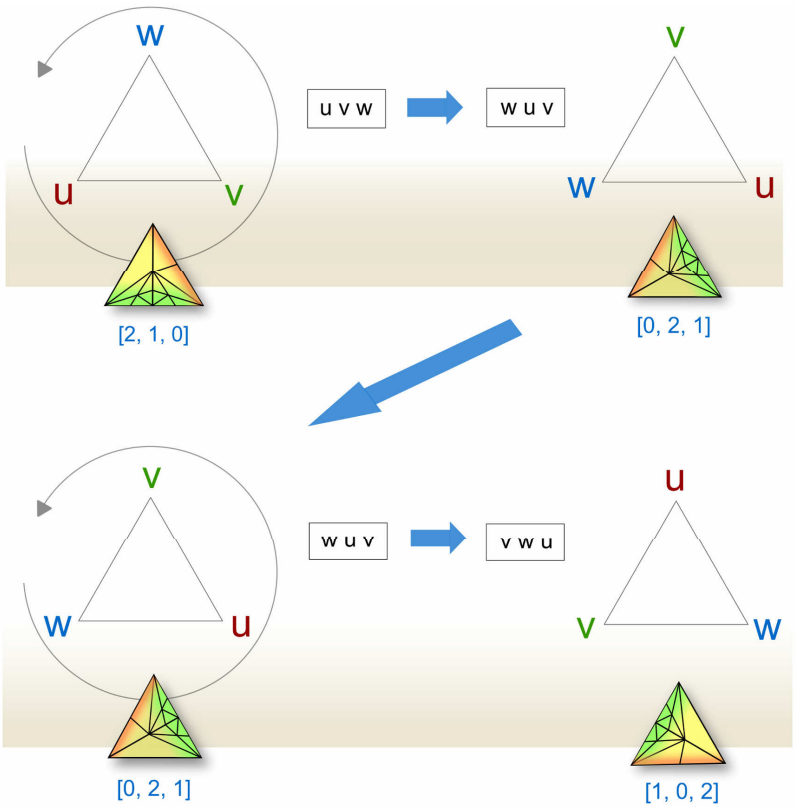


Figura 4.3. Rotações de um padrão e permutações correspondentes.

A Figura 4.3 mostra que rotacionar um padrão topológico em torno do eixo Z, no sentido anti-horário, equivale a "rotacionar" as coordenadas baricêntricas (u, v, w). Isso pode ser visto como se um triângulo de vértices *u*, *v* e *w* fosse também rotacionado junto, e a simples leitura dos vértices desse triângulo já indicaria a permutação a ser realizada para se obter o padrão desejado.

Por exemplo, no caso mostrado na parte superior da Figura 4.3, um padrão [2, 1, 0], que seria acessado normalmente pela tupla (u, v, w), é rotacionado uma vez, obtendo-se o padrão [0, 2, 1]. Esse padrão [0, 2, 1] é, na verdade, o mesmo padrão original [2, 1, 0], porém sendo acessado por meio da tupla (w, u, v) ao invés de (u, v, w).

Rotacionando-se novamente um padrão já rotacionado anteriormente, obtém-se ainda um outro padrão. Na Figura 4.3, embaixo, o padrão [0, 2, 1] obtido anteriormente é novamente

rotacionado, chegando-se dessa forma no padrão [1, 0, 2]. O triângulo representativo das coordenadas baricêntricas também é devidamente rotacionado, e assim se obtém a permutação (v, w, u), como indicada nessa figura. Caso esse último padrão fosse rotacionado mais uma vez, se obteria novamente o padrão original. Com isso, encerra-se um ciclo, e determinam-se as permutações específicas para a obtenção dos padrões [0, 2, 1] e [1, 0, 2] em função do padrão original [2, 1, 0].

Na Figura 4.4, são apresentadas mais operações sobre os padrões topológicos, dessa vez usando um "espelhamento":

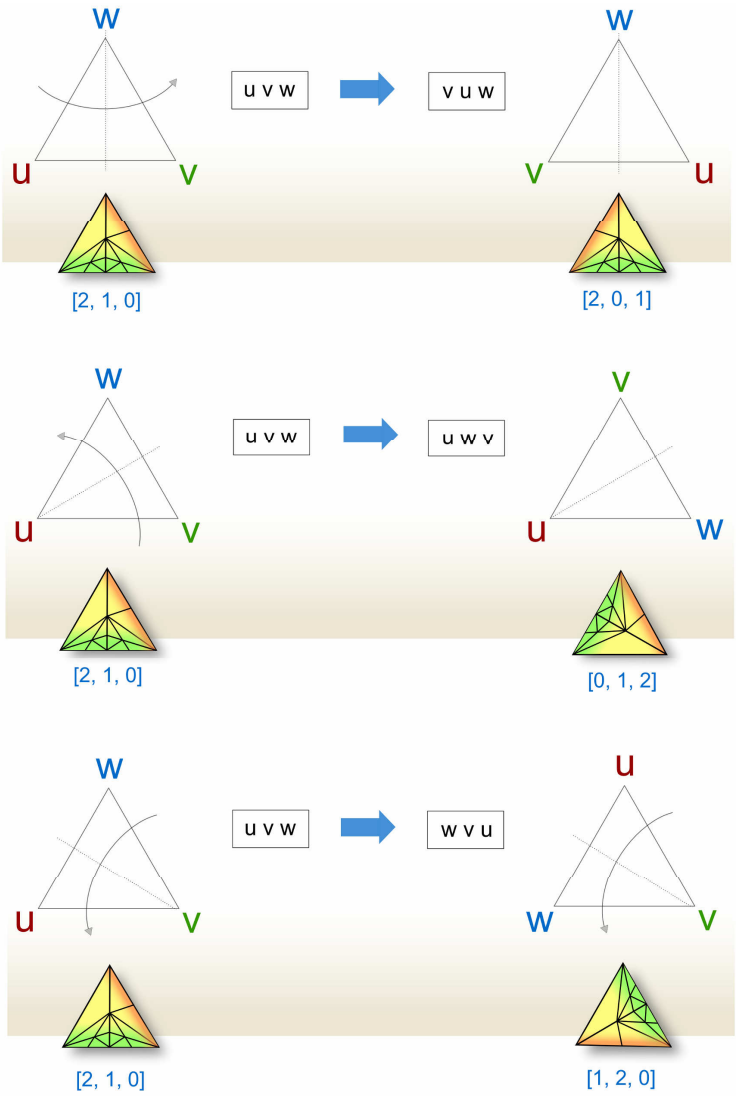


Figura 4.4. Espelhamentos do padrão e permutações correspondentes.

No espelhamento da Figura 4.4, adota-se um determinado eixo e então "gira-se" o triângulo em torno daquele eixo. O eixo adotado ocorre na direção de um dos vértices do triângulo que representa as coordenadas baricêntricas (u, v, w). Por exemplo, no primeiro caso apresentado na Figura 4.4, o eixo em questão aponta na direção do vértice *w*, e, por causa disso, o espelhamento em questão gera o triângulo (v, u, w) em função do triângulo original (u, v, w), como pode ser visto nessa figura. Isso significa que, se for usada a permutação (v, u, w) com o padrão original [2, 1, 0], será obtido o padrão [2, 0, 1] indicado na figura. De modo semelhante, os outros dois casos da Figura 4.4 obtêm os padrões [0, 1, 2] e [1, 2, 0].

Montando-se um catálogo com todos os padrões obtidos pelas operações ilustradas nas Figuras 4.3 e 4.4, obtém-se a seguinte lista de padrões:

Padrão original: **[2, 1, 0]**.
Padrões obtidos da Figura 4.3: **[0, 2, 1], [1, 0, 2]**.
Padrões obtidos da Figura 4.4: **[2, 0, 1], [0, 1, 2], [1, 2, 0]**.

Essa lista cobre todas as permutações possíveis das coordenadas baricêntricas (u, v, w). Além disso, mostra que, com apenas um padrão inicial, todos os outros cinco padrões são obtidos por meio de permutações nas coordenadas baricêntricas (u, v, w). A questão de qual deve ser o padrão inicial pode ser decidida por um critério arbitrário. No presente trabalho, o seguinte critério para escolha de qual padrão [i, j, k] armazenar foi adotado: aquele que tem os valores i, j, k ordenados na forma $i \geq j \geq k$. Assim, nos exemplos apresentados, o padrão [2, 1, 0] foi adotado como o padrão original de onde foram gerados os outros cinco. Os respectivos padrões da lista apresentada e as correspondentes permutações de (u, v, w) que geram esses padrões são mostrados na Tabela 4.1. Essa tabela contém os padrões obtidos pelas operações das Figuras 4.3 e 4.4, com as respectivas permutações indicadas.

Tabela 4.1. Obtenção de padrões em função das permutações em (u, v, w) realizadas.

[2, 1, 0]	Obtido de [2, 1, 0] como (**u**, **v**, **w**)
[2, 0, 1]	Obtido de [2, 1, 0] como (**v**, **u**, **w**)
[1, 2, 0]	Obtido de [2, 1, 0] como (**w**, **v**, **u**)
[1, 0, 2]	Obtido de [2, 1, 0] como (**v**, **w**, **u**)
[0, 2, 1]	Obtido de [2, 1, 0] como (**w**, **u**, **v**)
[0, 1, 2]	Obtido de [2, 1, 0] como (**u**, **w**, **v**)

Como, a partir de um único padrão, podem ser gerados outros cinco padrões em tempo real, por meio de uma simples troca entre os valores das variáveis baricêntricas (u, v, w), se poderia pensar que o número necessário de padrões topológicos adaptativos seria de apenas 1/6 do número total de padrões. Porém, há padrões topológicos que possuem valores repetidos de discretização, como [1, 2, 1] ou mesmo [1, 1, 1]. Para esses casos, as permutações das variáveis baricêntricas (u, v, w) não geram outros cinco padrões distintos. Assim, o número necessário de padrões topológicos é determinado pela Equação 4.1.

$$\frac{N^3 + 3N^2 + 2N}{6} \qquad \textbf{(4.1)}$$

Essa equação fornece o número total de padrões adaptativos [i, j, k], com i, j, k ≤ N, que precisam de fato ser armazenados, segundo o esquema proposto, para se obter os demais padrões em função dos padrões armazenados.

Conforme pode ser visto no gráfico da Figura 4.5, a redução no número de padrões necessários torna-se significativa quando a discretização máxima, N, é elevada. Por exemplo, para N = 10, são armazenados 1000 padrões [i, j, k] pelos métodos adaptativos tradicionais sem a otimização proposta; mas pelo esquema do presente trabalho apenas 220 padrões são armazenados, obtendo-se, assim, uma redução de 78% no número de padrões.

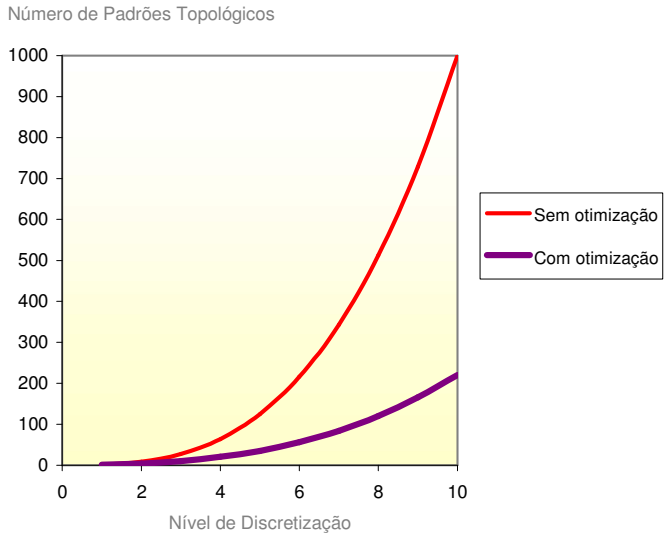


Figura 4.5. Armazenamento de padrões topológicos.

4.3. Escolha da Permutação

Na fase de seleção de padrões, cada elemento da malha original é associado com um determinado padrão [i, j, k], porém somente o padrão que obedece ao critério $i \geq j \geq k$ existe, conforme foi convencionado na Seção 4.2. Por causa disso, as variáveis (u, v, w), na fase de mapeamento de coordenadas, precisam passar por uma permutação adequada, para obtenção do padrão [i, j, k] desejado. Assim, uma vez escolhido o padrão topológico desejado para um determinado elemento da malha, deve-se usar uma das permutações apresentadas na Tabela 4.1, para obter o padrão desejado em tempo real. Contudo, é necessário que, em tempo de execução, o método de refinamento seja capaz de decidir qual permutação usar.

Por exemplo, se o padrão [0, 2, 1] for selecionado para um determinado elemento da malha, é possível verificar na Tabela 4.1 que esse padrão pode ser obtido do padrão [2, 1, 0] armazenado, através da permutação (w, u, v), como mostra a Figura 4.6. Porém, essa tabela contém apenas alguns casos de padrões; por exemplo, ela não contém nenhuma informação sobre os padrões [2, 7, 4], [3, 3, 2], [9, 3, 6], etc. Como, então, o algoritmo poderá decidir qual permutação usar para esses casos?

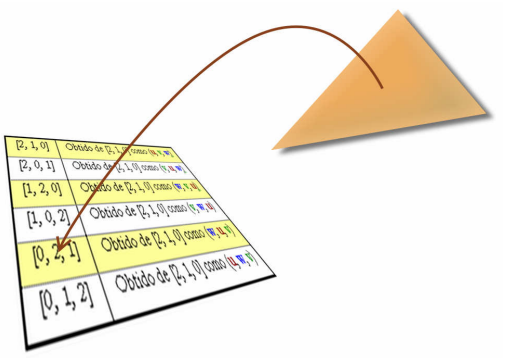


Figura 4.6. Obtendo a permutação do padrão desejado através da tabela.

Por causa disso, um algoritmo geral para a escolha da permutação se faz necessário. Isso significa que o algoritmo deve ser capaz de escolher a permutação adequada à obtenção de qualquer padrão [i, j, k] solicitado. Para os padrões listados na Tabela 4.1, o algoritmo deverá responder com as mesmas permutações indicadas nessa tabela.

Essa questão pode ser resolvida com a introdução de uma nova variável λ, para cada elemento da malha, que determina a permutação a ser feita para obter o padrão desejado para aquele elemento em função dos padrões armazenados (Figura 4.7).

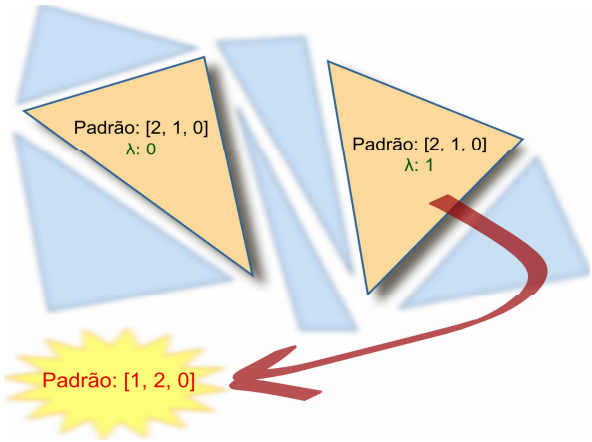


Figura 4.7. A variável λ define qual permutação fazer para obter o padrão desejado.

No exemplo ilustrado na Figura 4.7, os dois elementos da malha indicados usam permutações específicas do padrão armazenado [2, 1, 0]. Essa permutação é indicada pelo valor de λ que o próprio elemento da malha armazena. Dessa forma, enquanto um elemento usa λ = 0, indicando a permutação padrão (u, v, w), o outro elemento usa λ = 1, indicando outra permutação. O efeito disso é que o elemento com λ = 0 ficará como o padrão [2, 1, 0] sem permutação, e o elemento com λ = 1 ficará com o padrão [1, 2, 0], obtido através da permutação escolhida.

Com essa ilustração, o problema fica solucionado; agora, é necessário apenas especificar a maneira de calcular λ e como é escolhida a permutação das coordenadas baricêntricas (u, v, w) em função do valor de λ.

O modo como λ pode ser calculado é simples. Como a permutação de (u, v, w) depende da ordem de [i, j, k], então basta realizar um algoritmo que determine, de maneira única, um valor que dependa da ordenação de [i, j, k] (vide algoritmo da Figura 4.8). Depois disso, uma simples associação 1:1 de cada valor possível de λ com uma permutação específica pode ser feita.

A entrada para o algoritmo é a sequência numérica (i, j, k) que indica o padrão [i, j, k] desejado. Durante o algoritmo, a sequência recebida será devidamente ordenada para ficar na forma final, que foi estabelecida como sendo $i \geq j \geq k$ (vide Seção 4.2). Essa forma final é a que indica o padrão armazenado. Por exemplo, na Figura 4.7, o padrão armazenado é o [2, 1, 0], e o padrão desejado para um dos elementos é o [1, 2, 0]. Então a sequência numérica (1, 2, 0) é fornecida como entrada para o algoritmo da Figura 4.8, que fará a devida ordenação desses números e obterá a sequência numérica final (2, 1, 0), juntamente com o valor λ = 1.

```
1.  λ ← 0

2.  se i < j então
3.      trocar i ↔ j
4.      λ ← λ + 1

5.  se i < k então
6.      trocar i ↔ k
7.      λ ← λ + 2

8.  se j < k então
9.      trocar j ↔ k
10.     λ ← λ + 4
```

Figura 4.8. Algoritmo para calcular λ.

O algoritmo apresentado pode ser melhor entendido quando se considera que uma ordenação de uma sequência de itens numéricos pode ser feita apenas em função de determinadas trocas entre esses itens. No caso, para uma sequência com três itens, até três trocas poderão ser necessárias. Se cada troca for devidamente catalogada, então a ordenação completa pode ser apreendida apenas observando o catálogo de tais trocas. A Figura 4.9 ilustra como são nomeadas as trocas na ordenação feita pelo algoritmo, e a devida mudança que cada troca acarreta em λ.

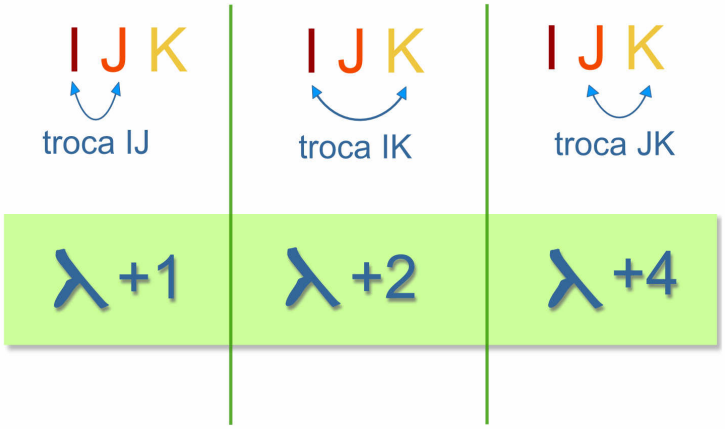


Figura 4.9. As três trocas possíveis na ordenação e as respectivas mudanças em λ.

Como são necessárias apenas três trocas, no máximo, para ordenar um conjunto numérico de três itens, e cada troca foi devidamente nomeada (Figura 4.9), uma simples máscara de bits

pode ser usada para saber quais trocas foram necessárias para a ordenação de uma sequência numérica. Da forma como o valor de λ é calculado, o número obtido já representa essa máscara de bits. A Figura 4.10 ilustra exemplos de ordenações usando o algoritmo do cálculo do λ apresentado. Nessa figura, diferentes padrões [i, j, k] são fornecidos para o algoritmo e, em cada exemplo, o algoritmo ordena os valores *i*, *j* e *k* do padrão fornecido para se chegar na forma $i \geq j \geq k$, catalogando cada troca feita e assim gerando o valor de λ.

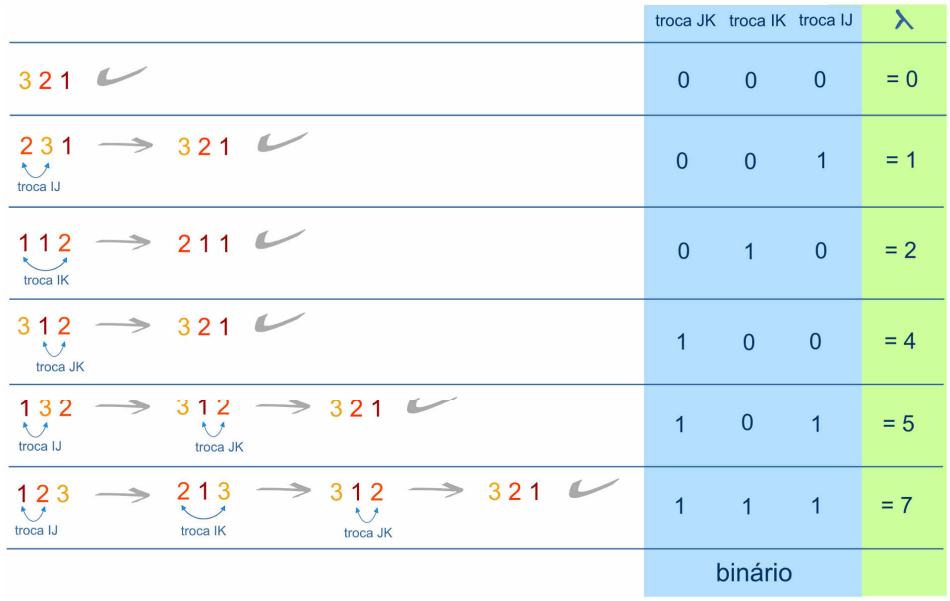


Figura 4.10. Como o λ pode ser calculado em função das trocas em uma ordenação.

Na verdade, o caso λ = 3 nunca irá ocorrer na execução do algoritmo da Figura 4.8, já que, se as condições das linhas 2 e 5 do algoritmo de ordenação da Figura 4.8 forem verdadeiras, a condição da linha 8 também será necessariamente verdadeira (se a < b e b < c, pela relação transitiva do operador *menor do que* tem-se que a < c necessariamente; isso torna o caso λ = 3 impossível de acontecer, já que nesse caso a condição a < c seria negada).

Convém lembrar, ainda, que o algoritmo apresentado funciona também para os casos onde há repetição de itens numéricos, como, por exemplo, para a sequência numérica (1, 1, 2).

Com o algoritmo que determina λ devidamente analisado, resta agora associar, para cada valor de λ obtido, a respectiva permutação de (u, v, w). Observando-se os valores da Tabela 4.1 em conjunto com os valores de λ calculados para cada caso ilustrado na Figura 4.10, pode-se encontrar essa associação entre os valores de λ e as permutações correspondentes (Tabela 4.2).

Tabela 4.2. Permutações (u, v, w) para cada valor de λ.

Padrão	Troca IJ (+1)	Troca IK (+2)	Troca JK (+4)	TOTAL (λ)	Permutação
2 1 0				0	u v w
1 2 0	**x**			1	w v u
1 1 2		**x**		2	u w v
2 0 1			**x**	4	v u w
0 2 1	**x**		**x**	5	w u v
1 0 2		**x**	**x**	6	v w u
0 1 2	**x**	**x**	**x**	7	u w v

A Tabela 4.2 mostra uma associação de cada valor λ possível de ser calculado com o algoritmo da Figura 4.8 com cada permutação da Tabela 4.1. Também são mostradas quais trocas o algoritmo de ordenação da Figura 4.8 realiza para cada caso. Em função dessas trocas, pode-se montar um esquema onde se obtém a permutação correta de (u, v, w) sem o auxílio da Tabela 4.1. Esse esquema é fornecido no algoritmo da Figura 4.11.

```
1.  permutação ← (u, v, w)
2.  para cada "x" marcado na linha do λ desejado:
3.      marcar "y" na coluna à DIREITA
4.  para cada "y" marcado na linha do λ desejado, em sentido inverso:
5.      fazer a troca indicada na coluna do "y" sobre a permutação
```

Figura 4.11. Algoritmo para calcular a permutação de (u, v, w) em função do λ.

No algoritmo proposto na Figura 4.11, a expressão "coluna da direita" significa que, para o "x" na coluna IJ, marca-se "y" na coluna IK; para o "x" na coluna IK, marca-se "y" na coluna JK; e para o "x" na coluna JK, por não haver uma coluna à direita de JK, marca-se "y" na primeira coluna IJ mesmo, formando assim um ciclo.

Na linha 4 do algoritmo apresentado na Figura 4.11, há a observação de que a iteração em questão deve ser feita no sentido inverso. Isso significa que, caso sejam marcados com "y" as colunas IK e JK, por exemplo, as trocas deverão ser feitas na ordem inversa: a troca JK em primeiro, seguida pela troca IK em segundo.

Para ilustrar o algoritmo apresentado, dois exemplos são suficientes. No primeiro exemplo, executa-se o algoritmo para λ = 1. Nesse caso, consultando-se a Tabela 4.2, pode-se verificar que apenas a coluna IJ tem um "x" marcado. Então, apenas um "y" é marcado na coluna à direita, a coluna IK. Agora, lendo as colunas que foram marcadas com "y" da direita para a esquerda, encontra-se apenas essa coluna IK; assim, realizando uma troca do tipo IK na sequência original (u, v, w), obtém-se a sequência final (w, v, u), como corretamente é indicado na Tabela 4.2.

Como segundo exemplo, o algoritmo é executado para λ = 6. Para esse caso, as colunas com um "x" marcado são as colunas de IK e JK. Marcando-se com um "y" as respectivas colunas da direita, são marcadas as colunas de JK e IJ (lembrando que, por JK ser a última coluna da direita, imagina-se uma outra coluna ainda, à direita de JK, que represente IJ). Agora, realizando primeiramente a troca IJ na sequência original (u, v, w), obtém-se (v, u, w); fazendo-se, em seguida, a troca JK nessa sequência (v, u, w), obtém-se a sequência final (v, w, u), como corretamente é indicado na Tabela 4.2.

Nesta seção, foi apresentado o procedimento para determinar as permutações que devem ser feitas de modo a obter os padrões topológicos desejados. À seguir, na Seção 4.4, discute-se como aplicar as informações apresentadas até aqui no método de refinamento.

4.4. Seleção de Padrões no Método Usando λ

Na etapa de seleção de padrões do método de refinamento, os padrões desejados para cada elemento da malha são devidamente selecionados. Como nem todos os padrões de fato existem na memória, introduz-se a variável λ para indicar como realizar uma permutação das coordenadas baricêntricas (u, v, w), na fase de mapeamento do método, a fim de obter os padrões desejados para os elementos, em função dos padrões que de fato estão armazenados na memória.

Na Seção 4.3, foi discutido como calcular o valor de λ para determinado elemento da malha em função do padrão [i, j, k] selecionado para esse elemento. Em algum momento, esse λ será usado para descobrir qual permutação deve ser feita com as coordenadas baricêntricas (u, v, w) a fim de obter o padrão desejado para o elemento. Nesta seção, pretende-se mostrar como isso se encaixa no algoritmo geral do método.

Na Figura 4.12, um algoritmo para seleção de padrões na malha é apresentado. Esse algoritmo faz uso do cálculo de λ apresentado no algoritmo da Figura 4.8.

```
1.  para cada elemento E da malha:
2.      escolher o padrão [i, j, k] desejado
3.      executar algoritmo que calcula λ
4.      armazenar a tupla (i, j, k, λ) em E
```

Figura 4.12. Algoritmo de aplicação de uma permutação de (u, v, w) em função do λ.

A linha 2 do algoritmo da Figura 4.12 deve ser personalizada para a aplicação. Conforme visto na Seção 3.8, a escolha do nível de discretização, feita para cada vértice da malha, pode ser realizada em função dos mais variados critérios, como distância ao observador, curvatura, etc. O padrão [i, j, k] escolhido pode não estar armazenado. Por causa disso, é importante a variável λ aqui, para indicar como obter esse padrão.

Para cada elemento da malha, devem ser armazenados: o padrão [i, j, k] determinado e o valor de λ. Não é necessário armazenar especificamente os valores (i, j, k) do padrão desejado [i, j, k], mas sim os valores (i, j, k) que indicam o padrão armazenado usado como base para obter o padrão desejado. Por exemplo, se o padrão [1, 2, 0] é selecionado, então, a tupla correspondente (2, 1, 0, 1) é que deve ser armazenada no elemento. Nessa tupla, ficam indicados o padrão que serve de base, [2, 1, 0], e o valor de λ que, sendo aplicado sobre esse padrão, devolverá o padrão desejado [1, 2, 0]. Nas outras fases do método, essas informações serão necessárias.

4.5. Agrupamentos de Padrões

Com a mudança de indexação proposta, a lista de padrões ficou menor e, por causa disso, mais triângulos da malha original estarão usando o mesmo padrão [i, j, k] armazenado. Se um mesmo padrão é repetidamente usado, isso significa que a técnica de *Instancing* da GPU pode ser usada para acelerar o algoritmo, paralelizando o processamento realizado para todos os elementos da malha que utilizam o mesmo padrão (Figura 4.13). Nesse tipo de paralelismo, os mesmos dados são reutilizados muitas vezes em paralelo, aumentando bastante o desempenho. A Seção 3.4 contém maiores detalhes sobre o *Instancing*.

Para usar esse recurso de paralelização é necessário, antes, agrupar todos os elementos da malha que utilizam o mesmo padrão, e então informar à GPU esses agrupamentos. Assim, a GPU irá, num único lote, processar todos os elementos que usam o mesmo padrão topológico.

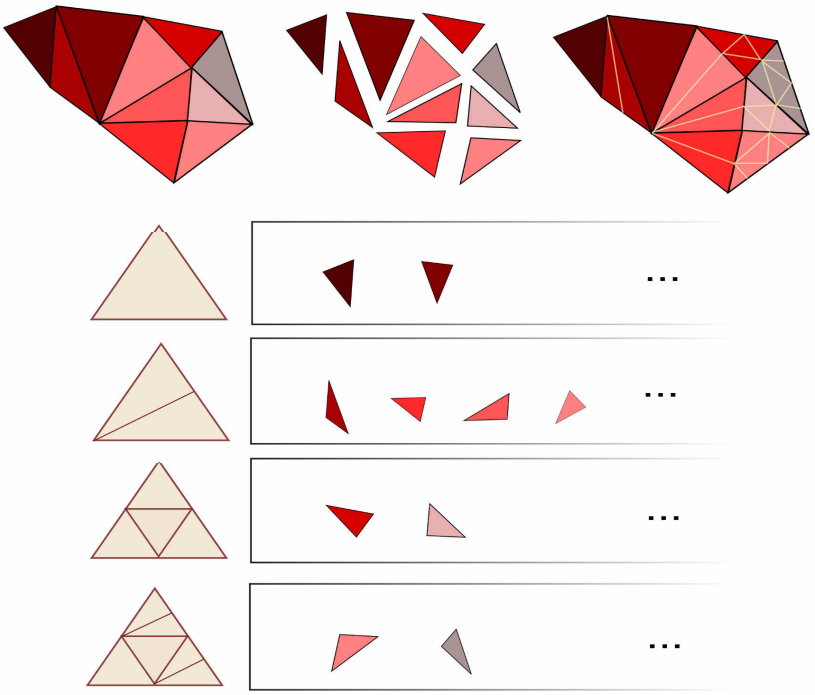


Figura 4.13. Processamento de elementos agrupados em lotes, por meio do Instancing.

Numa implementação inicial desse agrupamento de padrões, faz-se uma simples varredura dos elementos da malha e, assim, coleta-se quais padrões são usados na discretização da malha. Uma vez que essas informações tenham sido coletadas, cria-se um agrupamento para cada padrão distinto [i, j, k] sendo usado, e adiciona-se nesses agrupamentos as referências dos elementos da malha associados a tais padrões. A Figura 4.14 ilustra essa implementação.

Por exemplo, na Figura 4.14, após o algoritmo varrer todos os elementos da malha, foram obtidos os seguintes dados estatísticos: quatro padrões topológicos foram usados, que são os padrões [2, 1, 0], [2, 0, 1], [1, 1, 0], [0, 1, 1], e são usados 3, 1, 2, 1 vezes, respectivamente. Portanto, serão criados quatro agrupamentos, um para cada padrão, e em cada agrupamento serão referenciados os elementos da malha correspondentes. Quando, finalmente, o processamento do *Instancing* ocorrer na GPU, cada agrupamento será processado e, assim, todos os elementos da malha referenciados por aquele agrupamento serão devidamente processados também, num único lote e em paralelo. Para maiores explicações sobre como funciona o *Instancing*, ver a Seção 3.4.

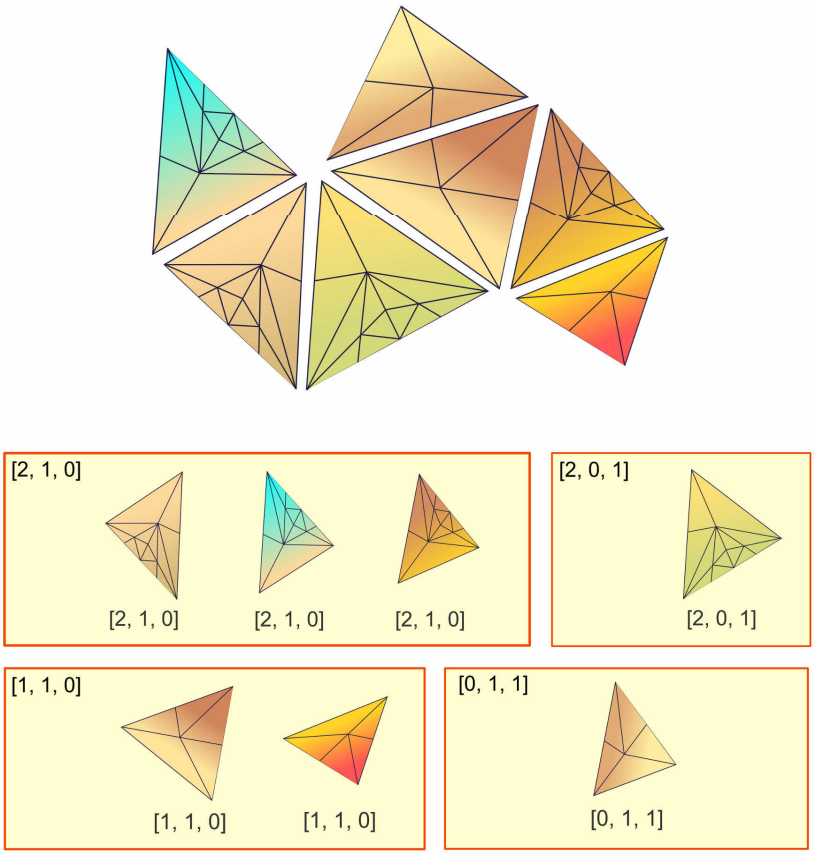


Figura 4.14. Implementação inicial do agrupamento de padrões.

Embora a implementação inicial, conforme explicada anteriormente e evidenciada na Figura 4.14, já mostre sinais promissores de desempenho, foi observado, contudo, que uma otimização mais forte ainda seria possível, dentro do método proposto. Acontece que, como cada elemento possui o seu próprio λ, pode-se agrupar todos os elementos que usam o padrão [2, 1, 0] e [2, 0, 1] num mesmo grupo, por exemplo (na Figura 4.14, isso equivaleria a fundir, por exemplo, os grupos indicados por [2, 1, 0] e [2, 0, 1], mostrados na parte superior, num único grupo). A razão pela qual isso se torna possível é que o padrão [2, 0, 1] usa, no método proposto, os mesmos dados armazenados para o padrão [2, 1, 0], já que ambos possuem o mesmo padrão base.

Assim, uma nova implementação mais otimizada é possível de ser feita: ao invés de se montar agrupamentos para cada padrão [i, j, k] sendo utilizado na discretização da malha, monta-

se os agrupamentos para cada padrão base sendo utilizado. A Figura 4.15 ilustra o resultado dessa nova implementação.

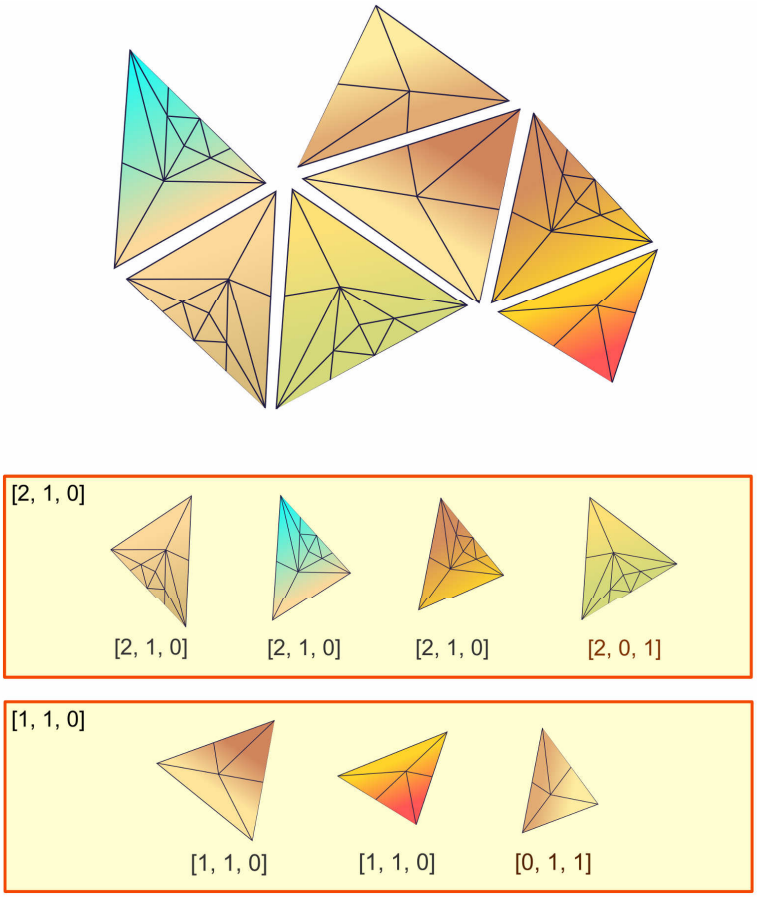


Figura 4.15. Implementação otimizada do agrupamento de padrões.

Comparando-se as Figuras 4.14 e 4.15, nota-se a diferença fundamental entre ambas as implementações: enquanto que na primeira os grupos são feitos para cada padrão, na segunda os grupos são feitos para cada padrão base. A consequência direta disso é que há mais grupos para a primeira implementação e menos grupos para a segunda. Como o número total de elementos da malha permanece o mesmo em ambos os casos, conclui-se que, na primeira implementação, haverá em média menos elementos para cada grupo, e, na segunda implementação, mais elementos para cada grupo. Portanto, a segunda implementação tem um desempenho bem superior, já que o paralelismo é intensificado quanto menor for a quantidade de agrupamentos.

Por exemplo, se 10 elementos da malha usam o mesmo padrão [2, 1, 0], então, pela primeira implementação, todos os 10 elementos seriam processados num único lote. Porém, pela segunda implementação, todos esses 10 elementos que usam o padrão [2, 1, 0] seriam processados num único lote juntamente com, por exemplo, outros 10 elementos que usam o padrão [1, 2, 0], e outros 10 elementos que usam o padrão [0, 1, 2], e outros 10 elementos que usam o padrão [0, 2, 1], e assim sucessivamente. Todos os padrões [i, j, k] que podem ser obtidos a partir de [2, 1, 0] são processados num único lote, o lote do padrão armazenado [2, 1, 0]. Portanto, a reutilização de dados, como é feita aqui, é bastante alta. Essa otimização só foi possível de se realizar por causa do esquema de obtenção de padrões por meio de permutações discutido nas Seções 4.2 e 4.3.

O algoritmo da Figura 4.16 cria os agrupamentos dos elementos conforme a discussão feita anteriormente, para a implementação otimizada. Para trocar para a implementação não-otimizada, bastaria modificar a linha 2 desse algoritmo: onde tem "padrão base", trocar por "padrão" simplesmente.

```
1.  para cada elemento E da malha:
2.      recuperar padrão base [i, j, k] indicado em E
3.      se já existir um grupo para o padrão então
4.          incluir elemento E nesse grupo do padrão
5.      senão
6.          criar grupo para o padrão
7.          incluir elemento E nesse grupo do padrão
```

Figura 4.16. Algoritmo para agrupar elementos da malha por padrão [i, j, k].

Ao final da execução do algoritmo da Figura 4.16, vários grupos terão sido criados, contendo, em cada um, referências a todos os elementos da malha que usam um mesmo padrão-base em comum.

4.6. Renderização da Malha

Boa parte do método apresentado trabalha com a malha num espaço ainda paramétrico, usando coordenadas baricêntricas. Na fase final do método a malha é *renderizada*, e nesse

momento a GPU realiza um mapeamento das coordenadas baricêntricas para as coordenadas finais desejadas.

Nesse mapeamento, muitos efeitos diferentes podem ser obtidos. Pode-se usar a técnica de *per-vertex displacement mapping* para aumentar o realismo de superfícies com uma textura de *height map* (Szirmay-Kalos & Umenhoffer, 2008) (Figura 4.17a), por exemplo, ou mesmo realizar um *fitting* de uma superfície paramétrica suave (Figura 4.17b) sobre os dados de cada face da malha de entrada. No caso, o já citado *Curved PN Triangle* é um exemplo desse tipo de superfície. Por fim, mesmo *métodos procedurais* (Figura 4.17c) são também empregados.

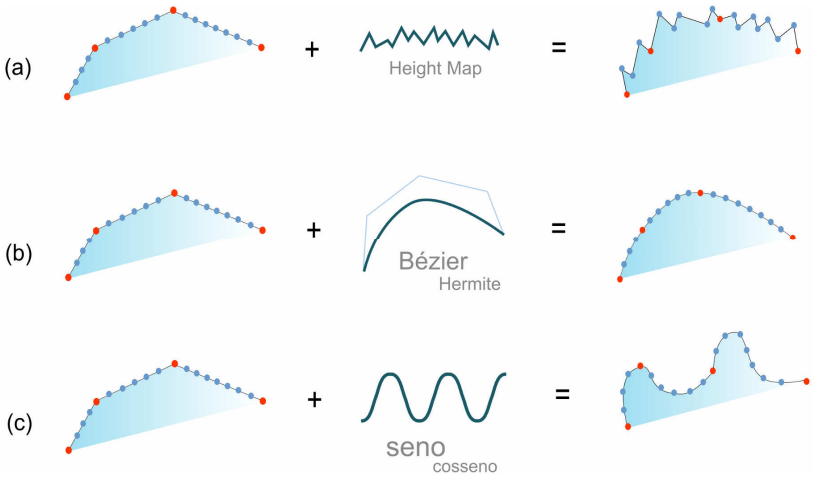


Figura 4.17. Manipulações na silhueta podem ser feitas no estágio final da renderização.

Na fase de renderização, cada agrupamento montado na fase anterior é processado da seguinte forma: manda-se a GPU *renderizar* o padrão-base [i, j, k] do agrupamento, usando *Instancing*, para N execuções, onde N é o número de elementos presentes no agrupamento em questão. Para cada execução, o devido mapeamento é realizado, em tempo real, dentro da própria GPU, por meio de um programa no *Vertex Shader*, por exemplo. Esse programa no *Vertex Shader* recebe cada vértice do padrão topológico, em coordenadas baricêntricas (u, v, w), e modifica esse vértice para ficar de acordo com a malha. Para exemplificar isso, uma simples discretização da malha, sem maiores efeitos, é ilustrada no algoritmo da Figura 4.18. Nesse algoritmo, os vértices do elemento triangular atualmente sendo processado são acessíveis por meio das variáveis P_1, P_2 e P_3, e o vértice em coordenadas baricêntricas é acessível por meio das variáveis *u*, *v* e *w*. O vértice final produzido é chamado de V_{final}. O uso do λ é considerado no algoritmo, a fim de se obter o

elemento discretizado de acordo com o padrão selecionado. As permutações de (u, v, w) usadas aqui estão de acordo com a Seção 4.3.

```
1.  para cada vértice (u, v, w) do padrão sendo usado:
2.      obter λ do elemento atualmente sendo processado
3.      para cada caso de λ:
4.          caso λ = 0: V_final ← P1 u + P2 v + P3 w
5.          caso λ = 1: V_final ← P1 w + P2 v + P3 u
6.          caso λ = 2: V_final ← P1 u + P2 w + P3 v
7.          caso λ = 4: V_final ← P1 v + P2 u + P3 w
8.          caso λ = 5: V_final ← P1 w + P2 u + P3 v
9.          caso λ = 6: V_final ← P1 v + P2 w + P3 u
10.         caso λ = 7: V_final ← P1 u + P2 w + P3 v
```

Figura 4.18. Um algoritmo de aplicação do λ em cada vértice processado no mapeamento.

4.7. Considerações Finais

Neste capítulo foi visto o método proposto neste trabalho em todos os seus pormenores. Uma visão geral foi apresentada e depois cada fase devidamente discutida. No próximo capítulo discutem-se os resultados obtidos com a aplicação deste método, com uma análise do desempenho e de alguns tipos de efeitos que podem ser aplicados sobre a silhueta da malha.

Capítulo 5. Aplicações, Resultados e Discussão

5.1. Introdução

Neste capítulo são apresentados exemplos de aplicação do método de refinamento proposto, introduzindo, quando necessário, os principais conceitos relativos às aplicações mostradas. Além dos resultados obtidos, também é apresentada uma análise comparativa do desempenho do método.

5.2. Aplicação com Curved PN Triangle

Um dos principais usos do método de refinamento proposto é o de suavizar malhas existentes. O exemplo da Figura 5.1 apresenta uma malha bruta de 1,5 mil elementos do coelho de Stanford renderizada pelos sistemas gráficos tradicionais e sua versão renderizada pela técnica proposta. Essa versão é obtida em função da malha bruta, usando a interpolação de uma superfície cúbica de Bézier conforme a especificação do *Curved PN Triangle*, e discretizada a ponto de ter 380 mil elementos. Com o ARK original, o desempenho da renderização é de 9 FPS; com o método proposto é de 59 FPS, um aumento de 556%. O visual obtido é o mesmo para ambos os métodos, uma vez que ambos foram executados para a mesma técnica geométrica do *Curved PN Triangle*. Porém, como o desempenho do método proposto é bastante superior, pode-se conduzir um refinamento com muito mais discretização ainda, de forma que o próprio resultado visual pode ser muito mais realista também.

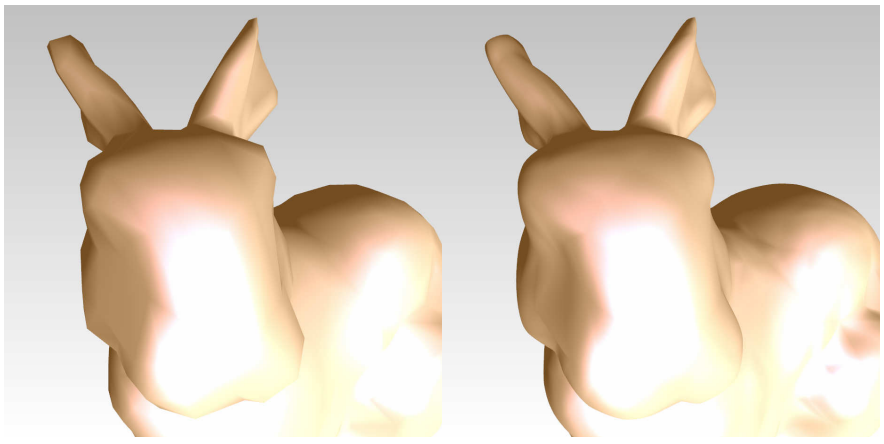


Figura 5.1. Malha do coelho de Stanford, sem suavização e com suavização.

O refinamento que realiza a suavização da malha é especialmente percebido à medida que o objeto se aproxima do observador. Sem a suavização, o usuário da aplicação gráfica rapidamente nota as imperfeições da silhueta da malha. Com a suavização, porém, a malha continua com uma silhueta bem definida, como se pode notar nos detalhes mostrados na Figura 5.2.

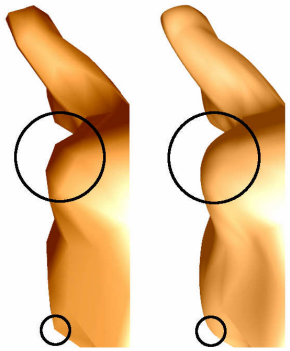


Figura 5.2. Detalhe da silhueta da malha.

A técnica de refinamento com suavização é útil mesmo para aplicações que trabalham com malhas existentes antigas e brutas. A suavização aplicada não depende de maiores descrições geométricas do objeto. Por exemplo, mesmo sem maiores detalhes sobre a xícara e contando simplesmente com os triângulos da mesma, o refinamento aplicado diretamente sobre uma versão triangularizada de baixa discretização da xícara gera um bom resultado (Figura 5.3). Para esse exemplo, a malha original tinha 1 mil elementos, enquanto a malha refinada tem 81 mil. A renderização com o ARK original é de 10 FPS, contra 228 FPS com o método proposto.

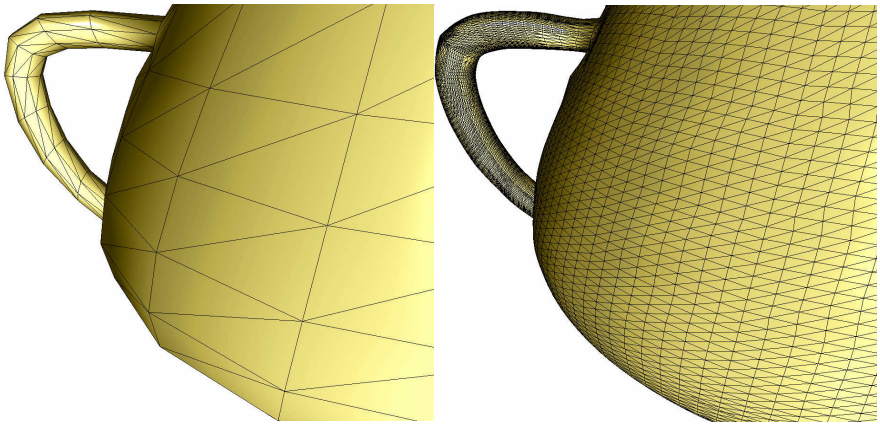


Figura 5.3. Detalhe da malha da xícara.

Em objetos mais "orgânicos" ou "arredondados", a suavização da silhueta é especialmente perceptível: a representação gráfica, para tais casos, fica bem superior. A Figura 5.4 apresenta um exemplo de objeto "arredondado" onde esse tipo de suavização é vantajoso: na esquerda, a malha original do objeto, e, na direita, a malha refinada com suavização da silhueta.

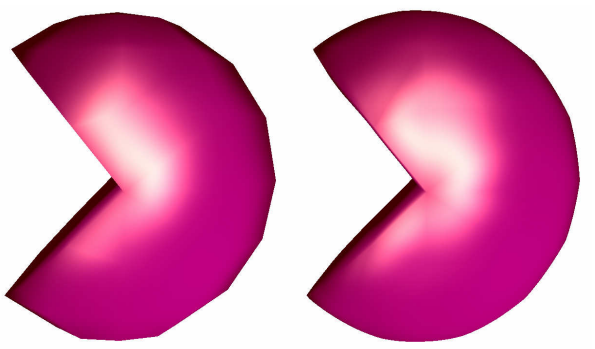


Figura 5.4. Aplicação de suavização na malha de um objeto "arredondado".

Como a suavização é aplicada no método de refinamento proposto? Isso ocorre na fase de mapeamento das coordenadas do espaço baricêntrico para o espaço geométrico da malha, durante a renderização do método (ver Seção 4.6). Um algoritmo de suavização, usando o *Curved PN Triangle*, é apresentado na Figura 5.5.

Nesse algoritmo, recebe-se como entrada as variáveis *u*, *v* e *w*, que representam as coordenadas baricêntricas do vértice sendo processado. As variáveis P_1, P_2 e P_3 são os vértices do triângulo atual, e N_1, N_2 e N_3 as respectivas normais. O que o algoritmo faz é o cálculo dos 10 pontos de controle que definem a superfície do *Curved PN Triangle* (Figura 5.6), definidos nas linhas 1-12, seguido da interpolação do ponto em questão na linha 13. A notação *dp* foi introduzida para facilitar a exibição desse código em particular, e é designada como a função produto escalar.

Na Figura 5.6, há uma ilustração da superfície paramétrica do *Curved PN Triangle*, contendo os 10 pontos de controle dessa superfície. Na verdade, o *Curved PN Triangle* funciona como uma heurística para a geração de uma superfície de Bézier cúbica triangular. Seus pontos de controle b_{300}, b_{030} e b_{003} são os próprios vértices do triângulo original; o ponto de controle b_{111} é um coeficiente central, e o restante dos pontos de controle são tangenciais. As razões pelas quais esses pontos de controle são definidos, conforme se pode ver no algoritmo da Figura 5.5, são dadas com maiores detalhes em (Vlachos et al., 2001).

1. $b_{300} \leftarrow P_1$

2. $b_{030} \leftarrow P_2$

3. $b_{003} \leftarrow P_3$

4. $b_{210} \leftarrow (2P_1 + P_2 - dp(\ dp(\ P_2 - P_1, N_1\ ), N_1\ )\ ) / 3$

5. $b_{120} \leftarrow (2P_2 + P_1 - dp(\ dp(\ P_1 - P_2, N_2\ ), N_2\ )\ ) / 3$

6. $b_{021} \leftarrow (2P_2 + P_3 - dp(\ dp(\ P_3 - P_2, N_2\ ), N_2\ )\ ) / 3$

7. $b_{012} \leftarrow (2p_3 + P_2 - dp(\ dp(\ P_2 - P_3, N_3\ ), N_3\ )\ ) / 3$

8. $b_{102} \leftarrow (2p_3 + P_1 - dp(\ dp(\ P_1 - P_3, N_3\ ), N_3\ )\ ) / 3$

9. $b_{201} \leftarrow (2P_1 + P_3 - dp(\ dp(\ P_3 - P_1, N_1\ ), N_1\ )\ ) / 3$

10. $e \leftarrow (b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201}) / 6$

11. $v \leftarrow (P_1 + P_2 + P_3) / 3$

12. $b_{111} \leftarrow e + (e - v) / 2$

13. $V_{final} \leftarrow 1b_{300}w^3 + 1b_{030}u^3 + 1b_{003}v^3 + 3b_{210}w^2u + 3b_{120}wu^2 + 3b_{201}w^2v + 3b_{021}u^2v + 3b_{102}wv^2 + 3b_{012}uv^2 + 6b_{111}wuv$

Figura 5.5. Algoritmo de mapeamento para o Curved PN Triangle.

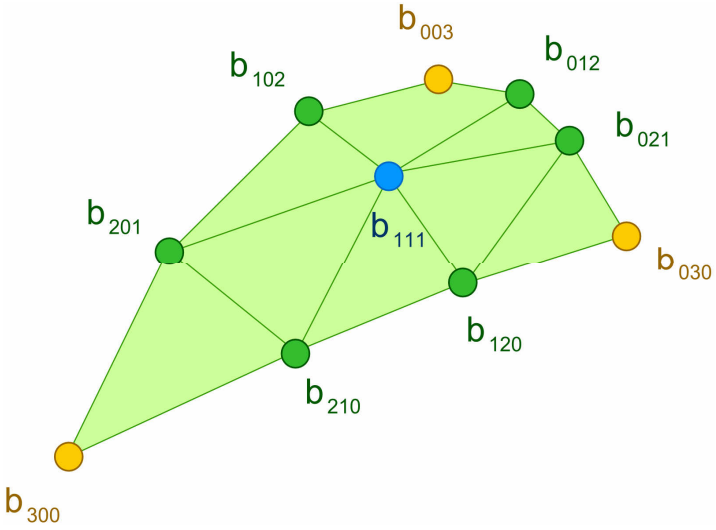


Figura 5.6. A superfície paramétrica do Curved PN Triangle e seus pontos de controle.

5.3. Aplicação com ST-Mesh

Em certas aplicações, é possível anexar valores que possam ajudar na descrição da malha existente. Nesse caso, uma técnica de suavização como a de ST-Mesh pode ser usada com o método proposto para suavizar a malha de uma forma mais personalizada.

No exemplo com ST-Mesh da Figura 5.7, a malha original contém 376 elementos, e a versão resultante, 10 mil. Para fazer a discretização dos elementos dessa malha, foi utilizado o operador de curvatura Gaussiana discreta de (Meyer et al., 2002), mencionado na Seção 3.8.

O ST-Mesh, implementado com o método ARK original, atinge 35 FPS na renderização nesse exemplo. Já com o método proposto, o refinamento com ST-Mesh atinge 291 FPS na renderização, um aumento de 731%.

Na Figura 5.7, à esquerda, observa-se a malha original usada para o refinamento; à direita, a malha resultante da técnica de refinamento feita por ST-Mesh. Nesse refinamento, foram configurados quatro vértices da malha original, a fim de guiar a geração das superfícies curvas adjacentes (Figura 5.8). Nessas configurações, são usados os *Scalar Tags* de *sharpness* (σ), *tension* (θ) e *bias* (β), além do parâmetro vetorial Δ (vide Seção 2.3).

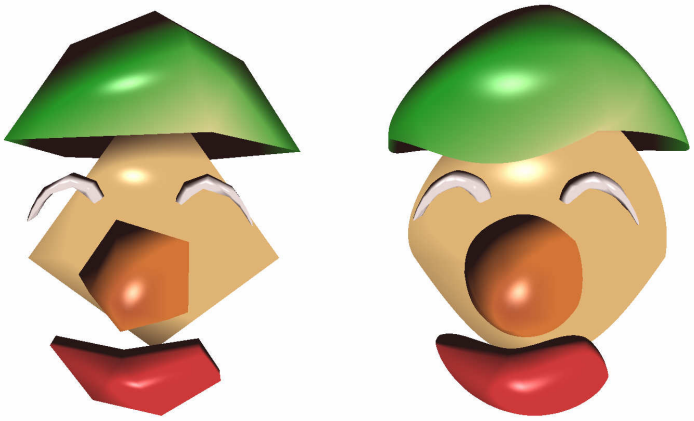


Figura 5.7. Malha da cabeça de um boneco, sem refinamento e com refinamento ST-Mesh.

Outro exemplo de refinamento de malha usando a técnica de ST-Mesh é ilustrado na Figura 5.9, onde a malha de um jato é manipulada para ficar com as asas mais curvadas. Além disso, como se pode observar numa aproximação do jato (Figura 5.10), o refinamento introduziu uma "dobra" na superfície da asa. Esse tipo de recurso é obtido por meio da geração proposital de uma descontinuidade nas normais da superfície, e mostra como a técnica de ST-Mesh pode ser flexível para isso.

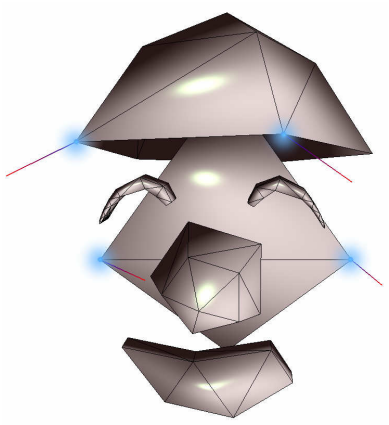


Figura 5.8. Vértices na malha onde os parâmetros de scalar tags são configurados.

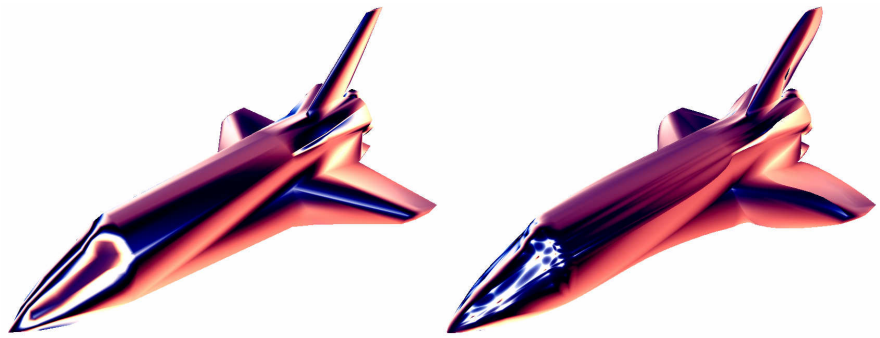


Figura 5.9. Manipulação da malha de um jato por meio do ST-Mesh.

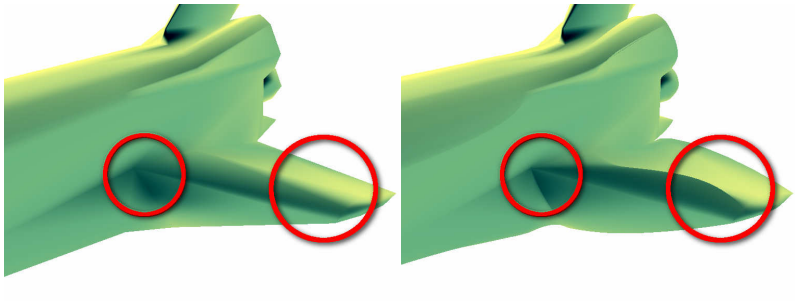


Figura 5.10. Alguns pontos de manipulação do jato no refinamento.

5.4. Aplicação com Métodos Procedurais

Também é possível usar outras técnicas de refinamento da malha, como, por exemplo, uma técnica que empregue métodos procedurais. A ideia é trabalhar com funções matemáticas básicas (como seno ou cosseno, por exemplo), para introduzir algum tipo de "pseudo-ruído" na silhueta da malha. O efeito sobre a silhueta da malha deve ser paramétrico, de modo que pequenos ajustes numéricos possam ser feitos a fim de se obter o efeito desejado.

Nesta seção, um efeito que procura simular pelos na superfície de um coelho é apresentado. O efeito tem parâmetros de amplitude ("tamanho dos pelos") e frequência. A Figura 5.11 apresenta a malha resultante dessa técnica de refinamento usando o parâmetro de amplitude como zero. O resultado gerado é de um coelho sem pelos.

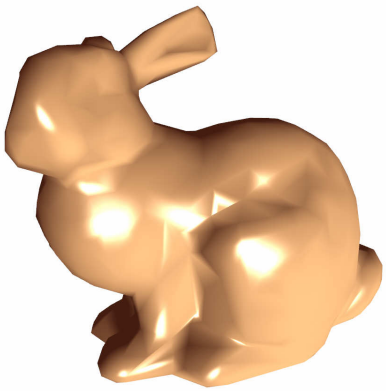


Figura 5.11. Malha resultante da aplicação de pelos, com amplitude 0.0.

Porém, aumentando a amplitude dos pelos, já é possível observar uma diferença visual na malha gerada. A Figura 5.12 exibe a malha resultante da aplicação dessa técnica de pelos com uma frequência de 1.0 e amplitudes de 0.03 (à esquerda) e 0.08 (à direita).

A Figura 5.13 exibe a malha resultante da aplicação de pelos com uma frequência de 3.0 e as mesmas amplitudes da figura anterior. A conclusão, após a observação dos resultados gráficos obtidos, é de que o método procedural empregado de fato produz um resultado diferenciado para ambos os parâmetros de entrada envolvidos (amplitude e frequência). Porém, o método de refinamento apenas gera vértices com posições diferenciadas, mas o número de elementos na malha resultante permanece o mesmo independente dos parâmetros, já que a discretização da malha não muda.

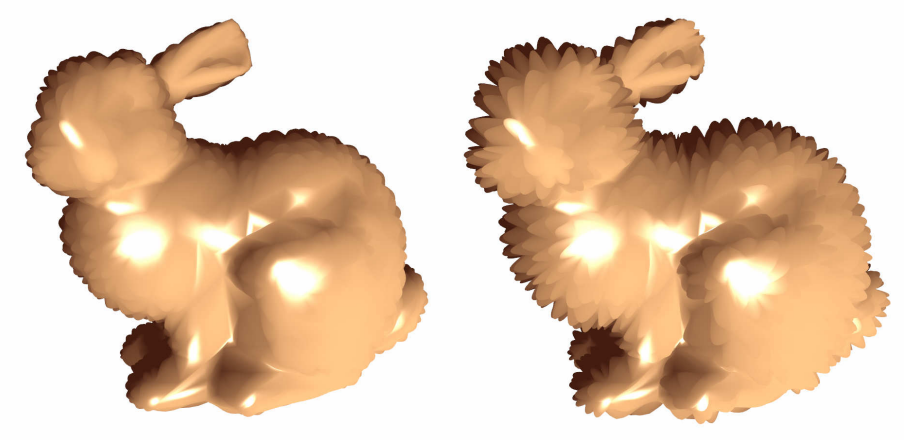


Figura 5.12. Aplicação de pelos, com frequência 1.0 e amplitudes 0.03 e 0.08.

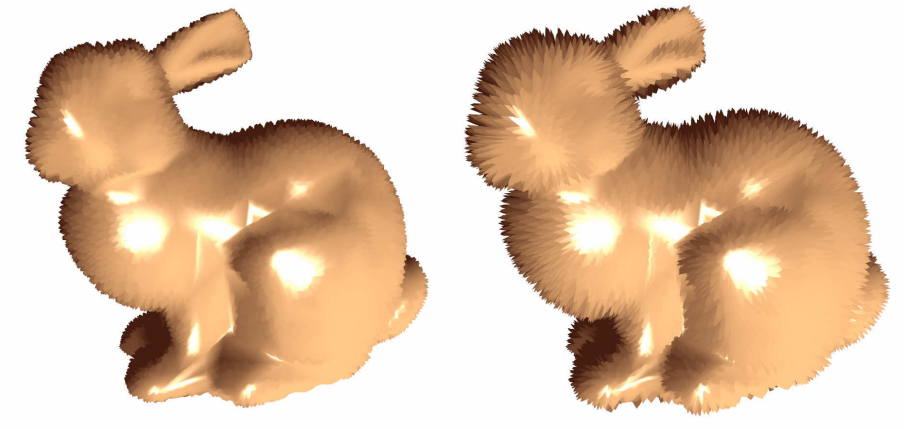


Figura 5.13. Aplicação de pelos, com frequência 3.0 e amplitudes 0.03 e 0.08.

Para gerar as imagens do coelho com pelos, foi utilizado o método de refinamento proposto nesta dissertação em conjunto com um algoritmo de mapeamento de coordenadas, definido na fase final do método. Embora possam ser produzidas muitas variações desse algoritmo, uma versão simples é apresentada na Figura 5.14.

```
1. f1 ← sen( uf_π )

2. f2 ← sen( vf_π )

3. f3 ← sen( wf_π )

4. g ← ( N1·N2 + N2·N3 + N3·N1 ) / 3

5. Δ ← agf1f2f3

6. N ← N1 u + N2 v + N3 w

7. Vfinal ← P1 u + P2 v + P3 w + ΔN
```

Figura 5.14. Algoritmo para geração de pelos.

Nesse algoritmo, a variável *f* representa o parâmetro escalar de frequência, e a variável *a* representa o parâmetro escalar de amplitude. As variáveis *u*, *v* e *w* são as coordenadas baricêntricas consideradas, e os vértices do triângulo são P_1, P_2 e P_3; as respectivas normais são N_1, N_2 e N_3. A linha 6 faz a interpolação da normal, N, com base nas normais da malha original, para simplificar. A linha 5 calcula um fator, Δ, que é usado para mover o vértice em questão na direção da normal N. É isso que fará com que apareçam, na silhueta, os picos gerados pela função seno usada nas linhas 1-3.

O fato de que os pelos são produzidos por um método procedural permite que os parâmetros de frequência e amplitude dos pelos sejam ajustados facilmente em tempo real, inclusive com animação. O resultado pode ser usado em aplicações de jogos ou de realidade virtual para enriquecer a representação gráfica.

Na Figura 5.15 são mostrados os pelos do coelho com o wireframe desativado (esquerda) e ativado (direita). Nesse exemplo, o número de elementos da malha original é de 1,5 mil, em contraste com o número de 380 mil elementos da malha resultante. No procedimento da renderização, nesse caso, obtém-se 112 FPS com o método proposto e 9 FPS com o método ARK original, devido ao seu gargalo na comunicação CPU-GPU.

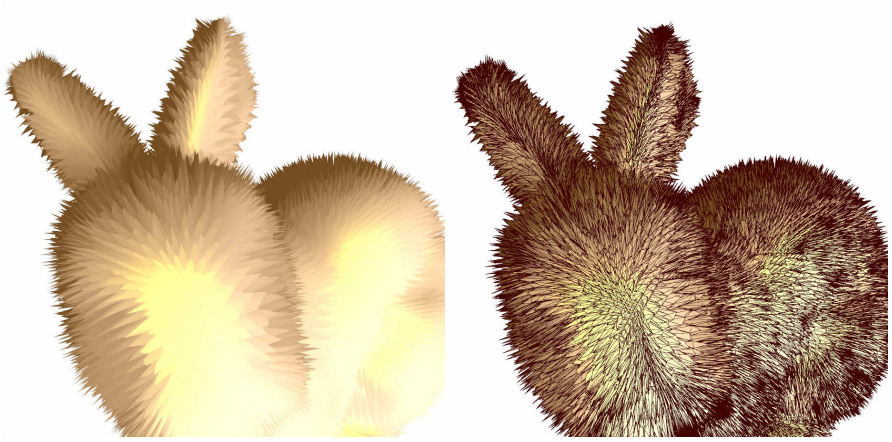


Figura 5.15. Refinamento para simular pelos na silhueta da malha do coelho.

5.5. Aplicação com Displacement Mapping

A aplicação do método proposto com *Displacement Mapping* exige antes uma breve discussão dos conceitos relacionados ao *Displacement Mapping*. Assim, a técnica geral do *Displacement Mapping* é discutida a seguir, e depois são mostrados os pormenores da aplicação dessa técnica no método proposto.

Um algoritmo de *Displacement Mapping* aplica informações geométricas de alta frequência na superfície de um modelo (Szirmay-Kalos & Umenhoffer, 2008). Na prática, o efeito disso é a adição de um relevo sobre a superfície dos objetos sendo representados (Figura 5.16).

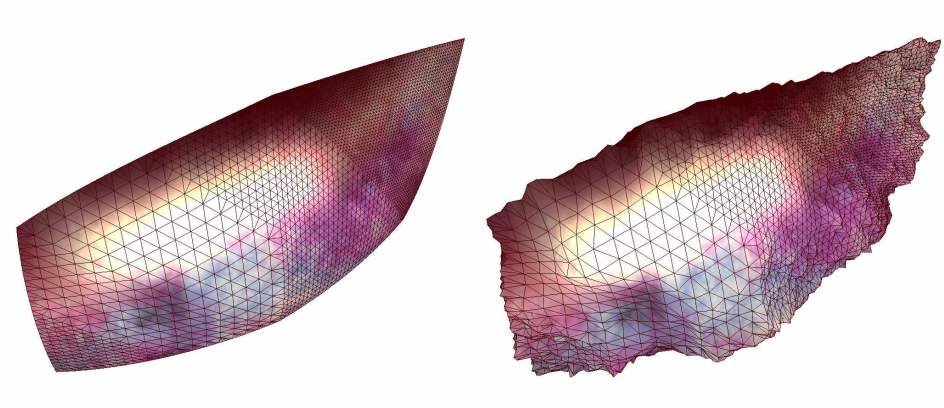


Figura 5.16. Adição de informações geométricas de relevo à malha.

Na Figura 5.16, à esquerda, uma malha representando um pedaço de coral é renderizada com a textura do coral, porém sem manipulação alguma de sua silhueta. Quando se aplica a técnica de *Displacement Mapping*, porém, a malha resultante fica muito mais rica, tornando a representação gráfica do objeto mais fiel.

O algoritmo de *Displacement Mapping* pode trabalhar modificando vértices ou modificando pixels (em especial, a visibilidade ou a coordenada de textura usada no pixel). Quando se trabalha com vértices, chama-se o algoritmo de *Per-Vertex Displacement Mapping*; quando se trabalha com pixels, o algoritmo é chamado de *Per-Pixel Displacement Mapping*. A técnica usada nesta seção se encaixa no grupo de algoritmos de *Per-Vertex Displacement Mapping*.

A aplicação da técnica de *Displacement Mapping* envolve um *height map*, que é uma textura em escala de cinzas. Nessa textura, cada ponto carrega a intensidade da altura do relevo naquela posição (Figura 5.17). O que a técnica de *Displacement Mapping* faz é utilizar as informações do relevo provenientes do *height map* para deslocar os vértices em uma superfície. A direção do deslocamento ocorre na mesma direção do vetor normal à superfície no ponto considerado, e a magnitude do deslocamento é proporcional à intensidade indicada no *height map*.

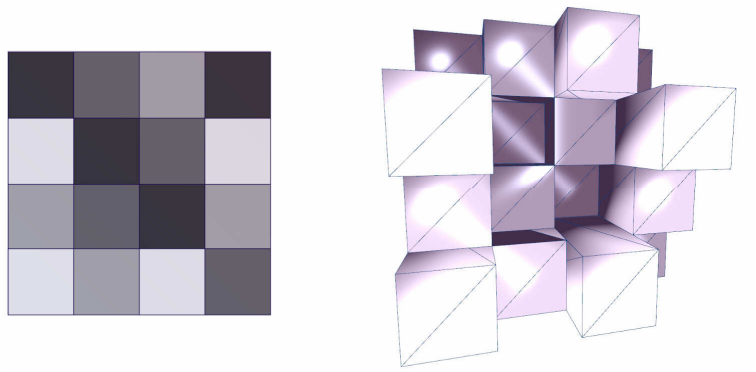


Figura 5.17. Textura do height map e as alturas correspondentes de cada ponto.

A textura do *height map* pode ser construída de várias maneiras. Na Figura 5.18, um esquema de obtenção das intensidades do *height map* é mostrado. Em (a), um raio é lançado de cima até tocar na superfície de algum objeto; e, em (b), a distância percorrida por cada raio é apresentada na forma de um tom de cinza. Assim, tem-se um "mapeamento" do relevo numa forma com tons de cinza, que é o próprio *height map*.

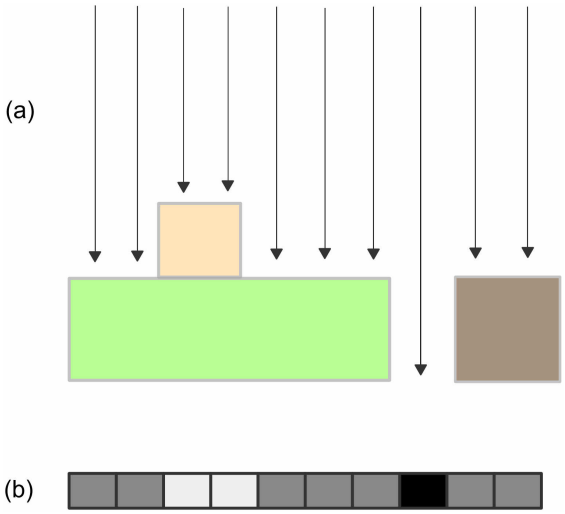


Figura 5.18. Coleta das informações de altura de um relevo.

Para usar a técnica de *Displacement Mapping* no método de refinamento de malhas, é necessário, primeiramente, especificar o nível de discretização desejado para cada parte da malha. Quando essa discretização é aplicada, a malha fica com vários vértices "sobrando" no mesmo plano de uma face. Então, um simples deslocamento da posição dos vértices é suficiente para gerar a informação do relevo. A Figura 5.19 ilustra esse processo de discretização na superfície da malha. A discretização é adaptativa, isto é, usa padrões topológicos adaptativos, o que permite que uma parte seja mais discretizada do que outra.

Na Figura 5.20, a aplicação das informações de relevo de um *height map* são aplicadas sobre a discretização da Figura 5.19. O resultado é o deslocamento dos vértices, na direção da normal, de acordo com a intensidade obtida do *height map*.

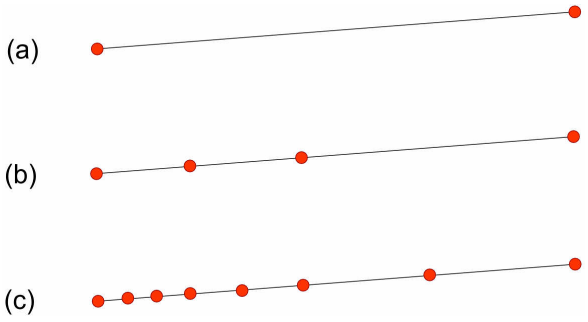


Figura 5.19. Geração da discretização da silhueta.

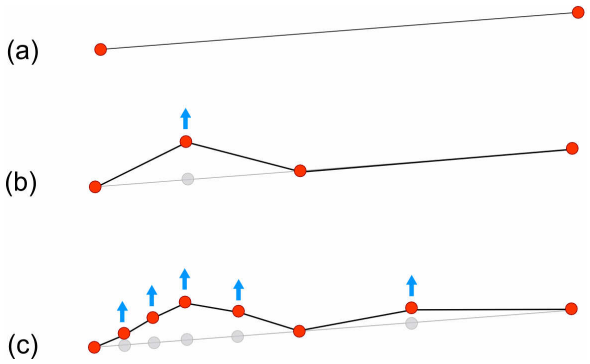


Figura 5.20. Aplicação do Displacement Mapping sobre a discretização da silhueta.

Aplicando a discretização em uma malha inicial bastante simples (Figura 5.21), e, em seguida, o deslocamento dos vértices de acordo com um *height map*, obtém-se a malha final com *Displacement Mapping* (Figura 5.22).

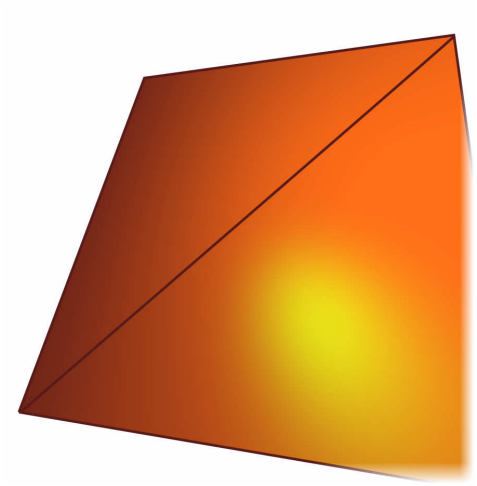


Figura 5.21. Malha inicial simples para aplicação do relevo.

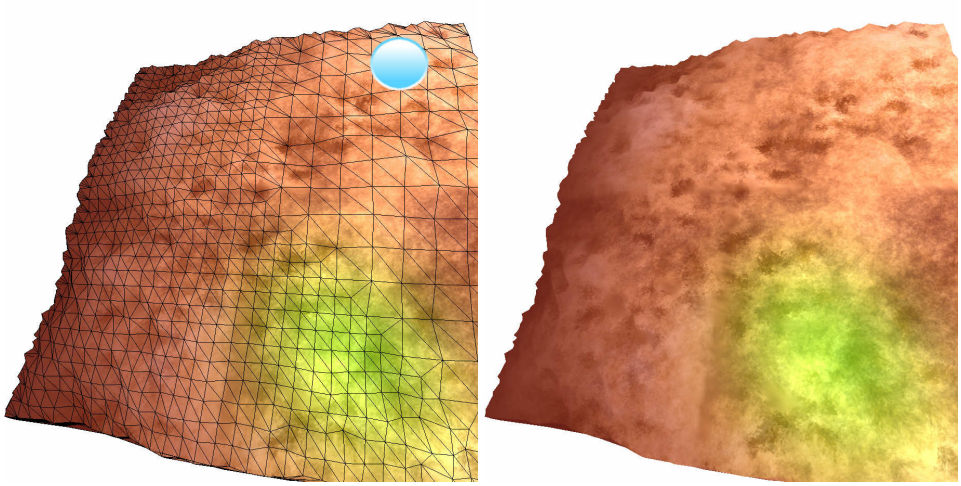


Figura 5.22. Malha obtida com o relevo aplicado numa discretização não-uniforme.

Porém, como o método de refinamento subjacente à técnica de *Displacement Mapping* sendo aplicada suporta padrões topológicos adaptativos, o relevo aplicado pode ser mais detalhado em certas regiões do que em outras. Dessa forma, na Figura 5.22, o local indicado pelo círculo azul foi propositalmente deixado com uma discretização menor, a fim de ilustrar o relevo obtido nesse cenário em contraste com o relevo da Figura 5.23, onde toda a superfície foi discretizada de forma uniforme. Nas Figuras 5.22 e 5.23, a imagem à esquerda representa a mesma malha da imagem à direita, sendo a versão da esquerda mostrada com o wireframe ativado e a da direita não.

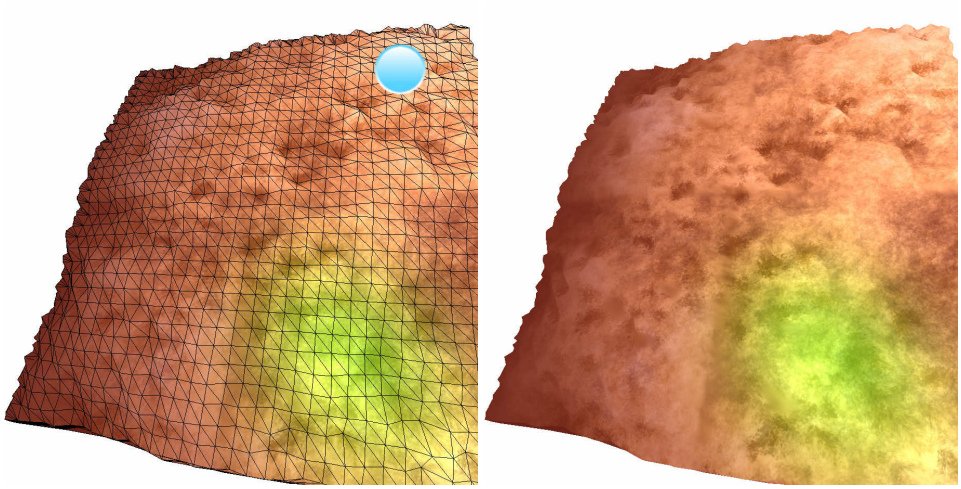


Figura 5.23. Relevo aplicado com uma discretização uniforme.

O interesse em ter uma técnica de *Displacement Mapping* implementada com um método de refinamento que trabalha com discretizações adaptativas é grande, pois permite, por exemplo, que um terreno seja melhor discretizado quando próximo do observador.

O último exemplo desta seção é o da foto por satélite de uma área urbana (Figura 5.24). Nessa figura, da foto à esquerda (retirada do *Google Maps*), é gerada uma imagem em escala de cinzas, o *height map* (à direita), que fornece as informações de relevo do terreno.

Figura 5.24. As texturas usadas: foto via satélite e imagem do relevo.

Com a foto e o *height map* obtidos, é possível gerar uma malha por *Displacement Mapping*, mostrando o terreno de forma tridimensional (Figura 5.25).

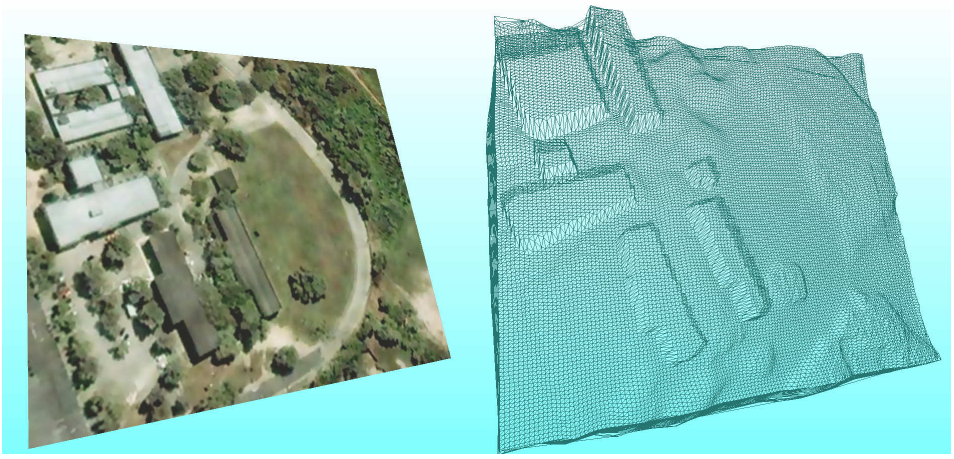


Figura 5.25. Superfície somente com a foto (esquerda) e somente com o relevo (direita).

A aplicação do *height map*, inclusive, pode ser personalizada. O deslocamento que é aplicado sobre os vértices, D, pode ser maior ou menor. Na Figura 5.26, o mesmo relevo das figuras

anteriores é mostrado com um deslocamento maior (à esquerda) e um deslocamento menor (à direita). Os prédios da foto (Figura 5.24), por exemplo, podem ter suas alturas ajustadas dessa forma.

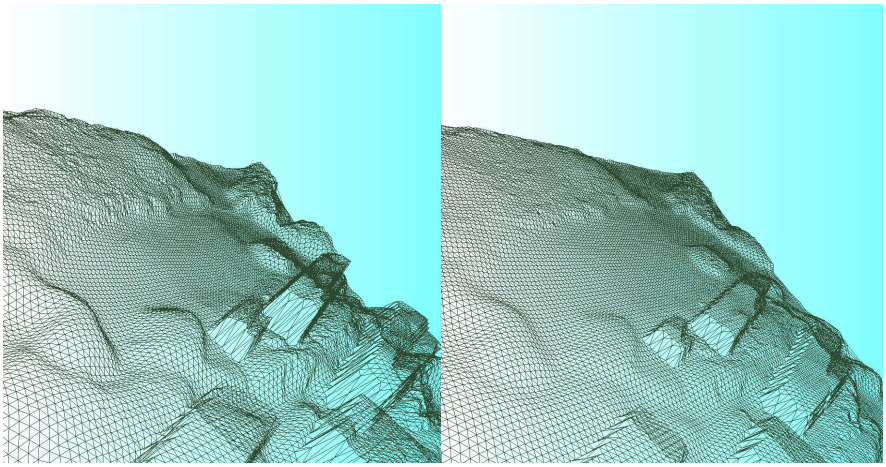


Figura 5.26. Diferentes alturas do relevo aplicado na superfície da malha.

As Figuras 5.27 e 5.28 mostram a malha refinada pelo método proposto com o *Displacement Mapping* aplicado. Nessas figuras, a imagem à esquerda mostra a malha resultante com o wireframe ativado, e, à direita, a mesma malha com o wireframe desativado.

Figura 5.27. Displacement Mapping aplicado com foto de satélite.

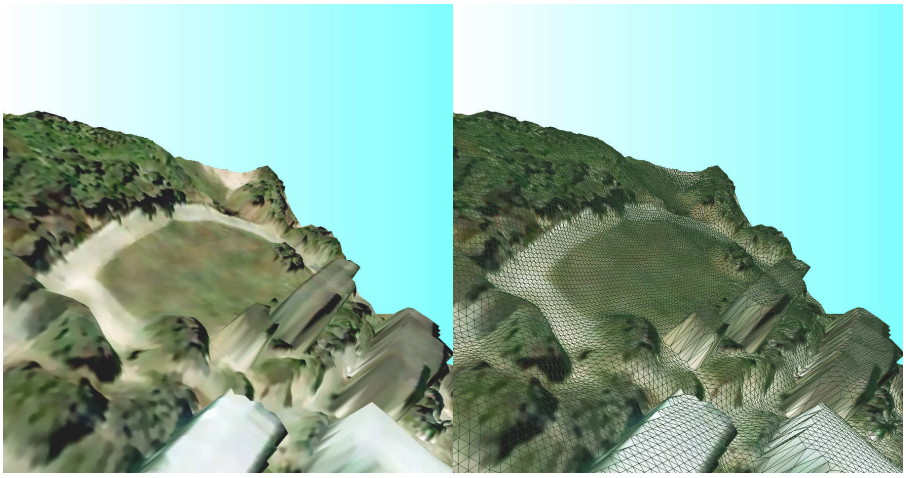


Figura 5.28. Outra visão do Displacement Mapping aplicado com foto de satélite.

5.6. Aplicação com Displacement Mapping e Curved PN Triangle

É possível implementar a união do *Displacement Mapping* com o *Curved PN Triangle*, de um modo bastante simples. Esse processo completo é ilustrado na Figura 5.29, onde: em (a), observa-se a malha original recebida pelo método; em (b), observa-se a malha discretizada após a fase de seleção de padrões; em (c), observa-se a malha resultante da aplicação da técnica do *Curved PN Triangle* somente; em (d), observa-se o resultado da aplicação da técnica do *Displacement Mapping* somente; em (e), observa-se o resultado da aplicação combinada do *Curved PN Triangle* com o *Displacement Mapping*. Na Figura 5.29f, é mostrado o *height map* usado para obtenção dos resultados ilustrados nas Figuras 5.29d e 5.29e, com os quatro valores usados no processo de deslocamento dos vértices.

A união do *Curved PN Triangle* com o *Displacement Mapping* ocorre da seguinte forma: na fase de mapeamento das coordenadas, no final do método de refinamento, faz-se o cálculo padrão do *Curved PN Triangle*, obtendo-se uma posição final V; em seguida, aplica-se o deslocamento proveniente da técnica do *Displacement Mapping*, D, sobre V; obtendo-se, assim, a equação $V_{final} = V + D$, que dá a posição do vértice proveniente das duas técnicas somadas.

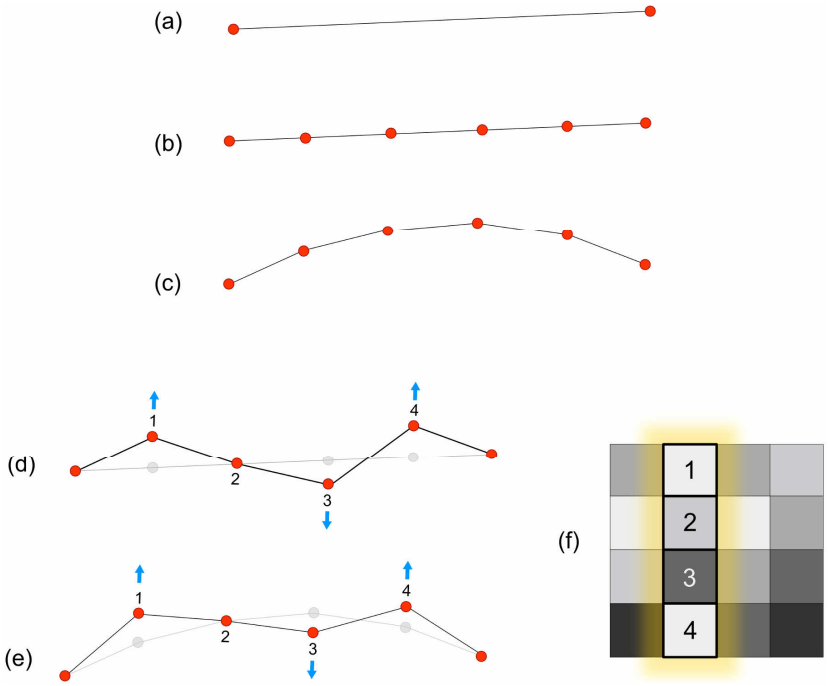


Figura 5.29. Aplicação do Displacement Mapping com Curved PN Triangle.

A Figura 5.30 ilustra um triângulo bastante discretizado submetido a ambas as técnicas por meio desse esquema. Como a geometria resultante de uma aplicação com *Curved PN Triangle* é fortemente influenciada pelas normais nos vértices, a malha pode ser interativamente manipulada, mesmo com *Displacement Mapping*, por meio de um manuseio das normais por parte do usuário.

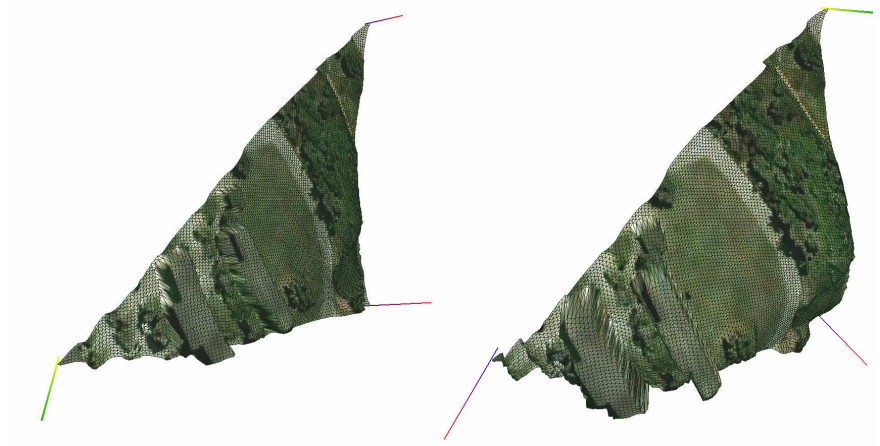


Figura 5.30. Manipulação geométrica do Displacement Mapping com Curved PN Triangle.

A Figura 5.31 ilustra um caso onde o biólogo, coletadas as amostras fotográficas da folha de uma determinada planta na forma de texturas (Figura 5.32), pode observar numa aplicação gráfica especializada a folha em toda a sua riqueza de detalhes, sem para isso precisar ir ao campo. As nervuras, sujeiras e outras características da folha podem ser devidamente reforçadas simplesmente ajustando-se o parâmetro da magnitude do deslocamento aplicado com o *Displacement Mapping*. Além disso, como a folha da planta foi toda mapeada numa malha tridimensional, suas características podem ser cuidadosamente observadas por qualquer ângulo.

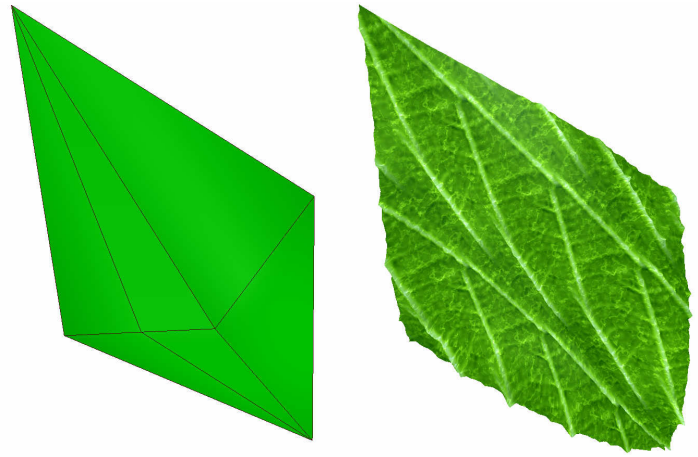


Figura 5.31. Refinamento com Displacement Mapping e Curved PN Triangle de uma folha.

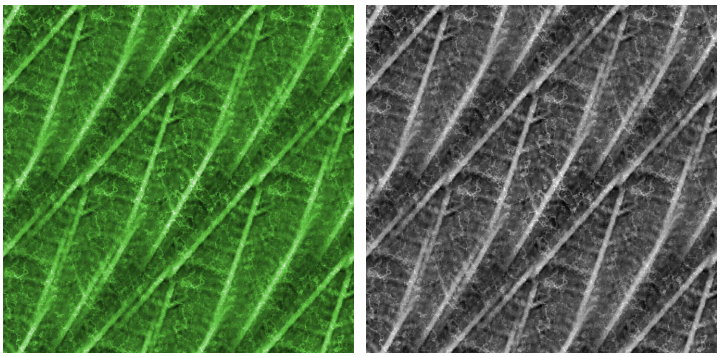


Figura 5.32. Texturas usadas no refinamento da folha.

O algoritmo de refinamento adaptativo de malhas facilita a implantação desse tipo de sistema de pesquisa, pois permite que a visualização dos dados geométricos seja otimizada e executada com maior eficiência (Figura 5.33).

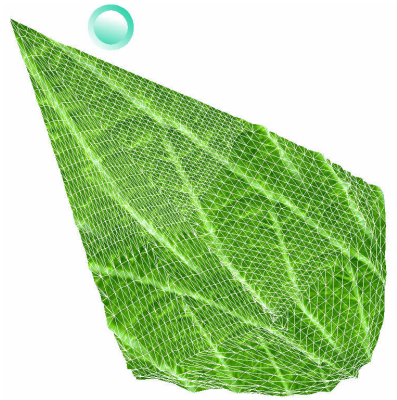


Figura 5.33. Refinamento adaptativo da folha.

Na Figura 5.33, a folha da planta é discretizada mais fortemente na região próxima do círculo indicado; dessa forma, apenas a parte onde o biólogo está observando com mais atenção e proximidade é mais refinada, sendo as demais partes da folha processadas com uma discretização menor.

5.7. Aplicação com Displacement Mapping e Métodos Procedurais

A textura usada como *height map* pode ser gerada através de um método procedural. Isso, em conjunto com uma imagem de vegetação também gerada por método procedural, caracterizaria um uso misto das técnicas de *Displacement Mapping* e métodos procedurais, como ilustrado na Figura 5.34. Embora o método procedural, propriamente dito, não seja aplicado diretamente sobre os vértices da superfície da malha, como é feito na Seção 5.4, ele é aplicado indiretamente por meio do *Displacement Mapping*.

Na Figura 5.34, a malha da direita é o resultado do refinamento aplicado sobre a malha da esquerda. Nessa figura, uma textura procedural foi aplicada tanto na forma de uma imagem mapeada sobre a superfície do objeto como na forma de um *height map* contendo informações sobre o relevo.

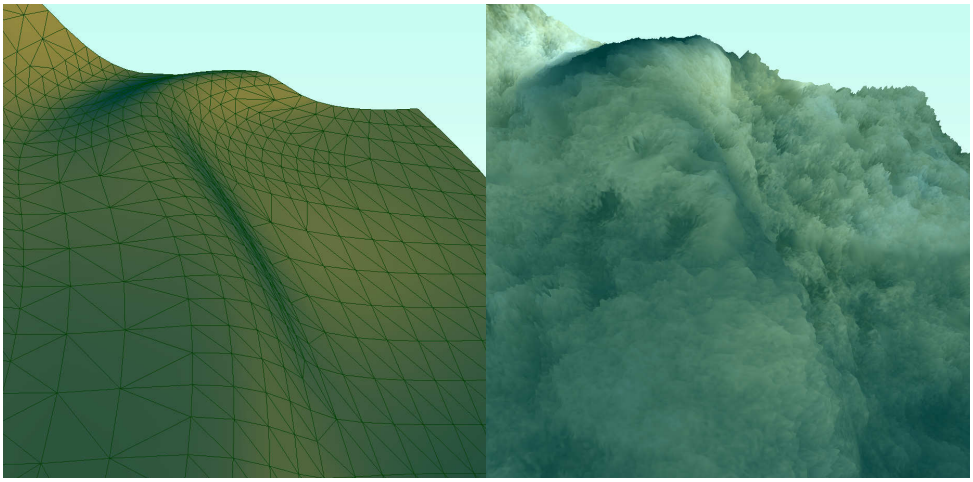


Figura 5.34. Aplicação de uma textura procedural com Displacement Mapping.

Na Figura 5.35, um outro ponto de vista é mostrado para a mesma malha refinada da figura anterior. O uso de texturas procedurais, em conjunto com a técnica de *Displacement Mapping*, ambas aplicadas pelo método de refinamento proposto, torna-se uma ferramenta muito flexível e otimizada para geração de conteúdo gráfico em simulações e jogos. Outro exemplo dessa técnica é ilustrado na Figura 5.36.

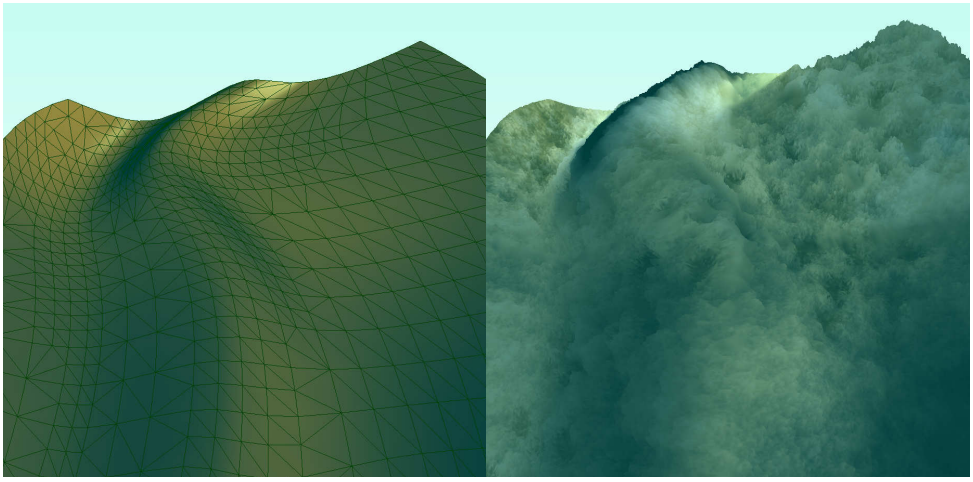


Figura 5.35. Outro ponto de vista da aplicação de textura procedural.

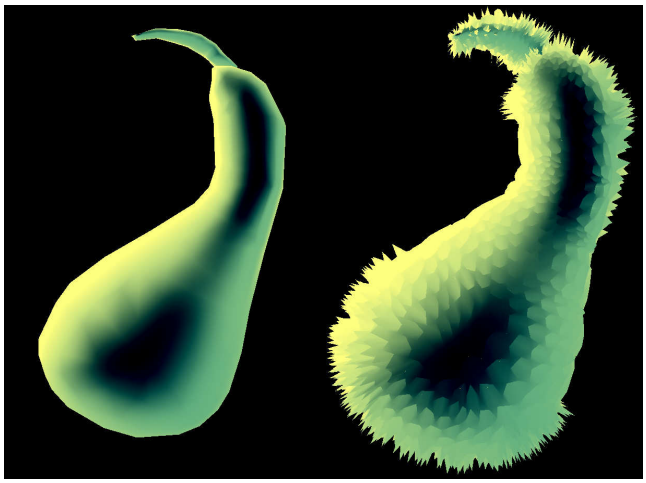


Figura 5.36. Aplicação de textura procedural na malha refinada de uma fruta.

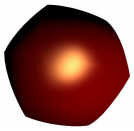
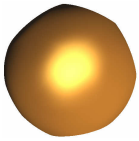
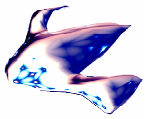
5.8. Análise Comparativa

Por uma questão de nomenclatura, o método apresentado nesta dissertação é aqui chamado de ARKFPI (ARK with Fewer Patterns and Instancing), em contraste com o método original ARK. Na Tabela 5.1, uma comparação estrita do ARK original com o ARKFPI é apresentada com outras malhas além dos exemplos já mostrados.

Cada malha testada passa por uma amplificação geométrica: o algoritmo recebe a entrada com certo número E de elementos e gera uma saída de S elementos. A relação S/E indica a amplificação realizada. O desempenho da renderização é medido em quadros por segundo (*frames per second*, FPS), e o aumento do desempenho é calculado como a razão da diferença dos desempenhos de renderização dos métodos ARKFPI e ARK. As diversas discretizações utilizadas são escolhidas de forma aleatória, porém em cada comparação entre os métodos, a mesma discretização e a mesma malha são usadas. Todos os testes foram conduzidos com um Athlon 64 X2 2.6 GHz, 4 GB RAM, GeForce 9600 GT, utilizando OpenGL.

O ARK realiza um grande número de chamadas da CPU à GPU, proporcional ao número de elementos da malha de entrada. Por causa disso, quando esse número é pequeno, o desempenho do ARK é bom, porém quando o número de elementos aumenta, o desempenho cai significativamente. Esse comportamento é explicável já que o ARK gasta mais tempo na comunicação CPU-GPU do que no trabalho gráfico realizado pela GPU. Na verdade, nos testes realizados, isso já acontece com malhas relativamente pequenas, da ordem de 200 elementos.

Tabela 5.1. Comparação do método ARK com a versão proposta neste trabalho (ARKFPI).

Malha	Elementos		S/E	FPS		Aumento do
	Entrada	Saída		ARK	ARKFPI	Desempenho
sphere1 	16	25312	1582	159	285	79.2%
	16	11006	688	168	404	140.5%
	16	8848	553	219	639	191.8%
sphere2 	48	4758	99	130	499	283.8%
	48	8856	185	123	396	222.0%
	48	16476	343	69	269	289.9%
pacman1 	110	73094	664	46	118	156.5%
	110	25154	229	52	188	261.5%
	110	14154	129	59	452	666.1%
jet 	303	1174	4	23	728	3065.2%
	303	4363	14	23	498	2065.2%
	303	10157	34	23	338	1369.6%
	303	25226	83	23	199	765.2%
pacman2 	490	1974	4	14	501	3478.6%
	490	3846	8	14	437	3021.4%
	490	8084	16	14	365	2507.1%
	490	15170	31	14	239	1607.1%
	490	98146	200	14	80	471.4%
	490	521536	1064	12	20	66.7%

Já o ARKFPI satura melhor a GPU: quando a malha a ser gerada tem um número menor de elementos, menos tempo total é gasto, e quando se tem um número maior de elementos de saída, mais tempo total é gasto. Isso significa que o peso da comunicação entre a CPU e a GPU é bem mais aliviado no ARKFPI, o que ocorre porque há um número menor de chamadas sendo feitas à GPU e mais dados sendo processados com cada chamada. A maior reutilização de dados proporcionada pelo esquema de indexação proposto contribui para isso, por meio do *Instancing*.

Visto que o ARKFPI satura bem a GPU, seu desempenho depende mais do tempo gasto no trabalho realizado pela própria GPU. Dessa forma, o desempenho aumenta quando se baixa o número de elementos de saída, mesmo para malhas grandes. Com o ARK isso não necessariamente acontece: quando a malha é grande, o desempenho permanece baixo o tempo todo. Em muitas aplicações gráficas isso se torna um peso, pois quando se faz necessário realizar um aumento do desempenho por meio de um ajuste de parâmetros, isso pode facilmente ser obtido no ARKFPI simplesmente regulando-se os níveis *d* de discretização da malha. No ARK, porém, essa operação não terá efeito algum, o que o torna mais inflexível para isso.

O melhor caso do ARKFPI em relação ao ARK ocorre justamente para malhas de entrada com maior número de elementos e com uma amplificação menor; nesse caso, o fator de melhora chega a ser de 3479%. No pior caso, porém, ainda se tem uma melhora de 67%. Pode-se afirmar, portanto, que o ARKFPI é mais rápido do que o ARK e é mais flexível para ajustes de desempenho, pelos motivos previamente explicados.

Capítulo 6. Conclusões

6.1. Principais Contribuições

Nesta dissertação, foi apresentada uma técnica para refinamento local de malhas com GPU, tomando como base malhas triangulares existentes. A técnica em questão usa um refinamento baseado em padrões topológicos adaptativos. Os principais conceitos associados ao tema deste trabalho foram introduzidos, e um conjunto de operações com padrões topológicos de refinamento foi apresentado. Foi mostrado que outros padrões podem ser obtidos de um único padrão-base, de modo que, em uma aplicação, basta armazenar o padrão-base e recuperar os padrões necessários, a partir do padrão-base, durante a execução. Essa obtenção dos padrões derivados ocorre em tempo real, sem perda alguma de desempenho.

Com o esquema introduzido, uma maior reutilização de padrões acontece, e, por causa disso, foi possível explorar melhor o refinamento de malhas com GPU através de agrupamentos de padrões, que passam a ser amplamente reutilizados. Os agrupamentos entram na GPU e são processados em paralelo por meio da técnica de *Instancing* da GPU, atingindo assim um bom desempenho em relação aos outros métodos semelhantes na área.

Vários exemplos de aplicação do método foram apresentados, como, por exemplo, a suavização de malhas existentes em tempo real, sem informações adicionais, por meio do *Curved PN Triangle*. De fato, essa aplicação do método proposto funciona suficientemente bem para as abundantes malhas triangulares brutas que existem hoje em diversos aplicativos de realidade virtual, jogos, etc., de modo que todos esses setores já podem se beneficiar dessa aplicação. Foram mostrados também outros exemplos de manipulação da silhueta, incluindo métodos procedurais e também o *Displacement Mapping*. Um dos exemplos de uso da técnica de *Displacement Mapping* aliada ao método proposto é o de prover relevo em terrenos, que passam a ser discretizados adaptativamente de forma rápida e flexível por meio do método proposto.

A flexibilidade do método de refinamento proposto é evidente pela quantidade de aplicações e exemplos dados. Para muitos programas, a motivação é imediata: vários tipos de uso do método podem ser empregados, e o desempenho obtido é rápido porque boa parte do processamento intenso na malha é realizada numa infraestrutura que permite a obtenção de amplo paralelismo na GPU. Além disso, a velocidade de processamento das GPUs aumenta

constantemente, numa evolução impressionante de números, sendo, provavelmente, o dispositivo com maior poder computacional hoje numa relação de custo/benefício (Owens et al., 2007).

6.2. Trabalhos Futuros

Para trabalhos futuros, pode-se investigar outros meios de se lidar com os padrões topológicos, de modo a diminuir seu número mais ainda, sem, no entanto, cair em outros problemas, como o caso das regiões de transição muito descontínuas.

O modo como os padrões topológicos são escolhidos, em função de níveis de discretização presentes nos vértices, também poderia ser mudado para ser definido diretamente em função das arestas, levando-se em consideração também as faces adjacentes às arestas, que passaram a ser acessíveis nas GPUs de 4ª geração.

Também se poderia verificar uma possível incorporação da superfície de PNG1 (Fünfzig et al., 2008) na técnica apresentada. Outros tipos de superfícies também poderiam ser incorporados. Por exemplo, pode-se investigar o uso da *Rational Bézier*, sobretudo na questão de seus pesos w_i e de possível correspondência a ser feita com os triângulos adjacentes.

Novos avanços nas GPUs permitem explorar melhor o tratamento geométrico de malhas, através de recursos como o *Hull Shader* e *Domain Shader* (Gee, 2008), o que pode fornecer novos meios para se trabalhar na área.

Por fim, um outro trabalho futuro, ainda, é a comparação do método proposto com o método DMR.

Referências Bibliográficas

(Ati, 2001) ATI Inc. TRUFORM white paper, 2001.

(Blythe, 2006) David Blythe. The Direct3D 10 system. Proceedings of ACM SIGGRAPH, p. 724-734, 2006.

(Bolz & Schröder, 2003) Bolz J. and Schröder P. Evaluation of subdivision surfaces on programmable graphics hardware, 2003. Disponível em: http://www.multires.caltech.edu/pubs/GPUSubD.pdf.

(Boubekeur et al., 2005a) Tamy Boubekeur, Patrick Reuter, Christophe Schlick. Scalar Tagged PN Triangles. Eurographics 2005 Short Presentations, p. 17-20, 2005.

(Boubekeur et al., 2005b) Tamy Boubekeur, Christophe Schlick. Generic Mesh Refinement on GPU. Proceedings of the ACM Siggraph/Eurographics Conference on Graphics Hardware, p. 99-104, 2005.

(Boubekeur & Schlick, 2007) Tamy Boubekeur, Christophe Schlick. A Flexible Kernel for Adaptive Mesh Refinement on GPU. Computer Graphics Forum, Volume 27 issue 1, p. 102-113, 2007.

(Brown & Lichtenbelt, 2006) Pat Brown and Barthold Lichtenbelt. EXT_geometry_shader4, 2006. Há uma versão disponível em: http://www.opengl.org/registry/specs/EXT/geometry_shader4.txt.

(Dyken et al., 2007) C. Dyken, M. Reimers, J. Seland. Real-Time GPU Silhouette Refinement using Adaptively Blended Bézier Patches. Computer Graphics Forum, Volume 27, Issue 1, p. 1-12, 2007.

(Dyken et al., 2009) C. Dyken, M. Reimers, J. Seland. Semi-uniform Adaptive Patch Tessellation. Computer Graphics Forum, 2009 (a ser publicado).

(Fünfzig et al., 2008) Christoph Fünfzig, Kerstin Müller, Dianne Hansford, Gerald Farin. PNG1 Triangles for Tangent Plane Continuous Surfaces on the GPU. ACM International Conference Proceeding Series, Volume 322, Proceedings of graphics interface, pages 219-226, 2008.

(Gee, 2008) Kevin Gee. Direct3D 11 Tessellation. Apresentação do GameFest 2008, Seattle, Washington, Julho 2008.

(Lichtenbelt & Brown, 2006) Barthold Lichtenbelt and Pat Brown. EXT_gpu_shader4, 2006. Há uma versão disponível em:
http://www.opengl.org/registry/specs/EXT/gpu_shader4.txt.

(Lorenz & Döllner, 2008) Haik Lorenz, Jürgen Döllner. Dynamic Mesh Refinement on GPU using Geometry Shaders. Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, 2008.

(Meyer et al., 2002) Mark Meyer, Mathieu Desbrun, Peter Schroeder and Al Barr. Discrete Differential Geometry Operators for Triangulated 2-Manifolds. VisMath '02 Proceedings, 2002.

(Moreton, 2001) Henry Moreton. Watertight tessellation using forward differencing. Graphics Hardware 2001 (2001), pp. 25–32.

(Owens et al., 2007) John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, Volume 26, number 1, p. 80-113, 2007.

(Szirmay-Kalos & Umenhoffer, 2008) László Szirmay-Kalos, Tamás Umenhoffer. Displacement Mapping on the GPU - State of the Art. Computer Graphics Forum, Volume 27, Number 6, pp. 1567-1592. 2008.

(Rost, 2006) Randi J. Rost. OpenGL Shading Language, Second Edition. Addison Wesley Professional, 2006.

(Shiue et al., 2003) Shiue, L.-J., Goel, V., and Peters, J. 2003. Mesh mutation in programmable graphics hardware. Proceedings of the Conference on Graphics Hardware, 15–24.

(Vlachos et al., 2001) Alex Vlachos, Jörg Peters, Chas Boyd, Jason Mitchell. Curved PN Triangles. ACM Symposium on Interactive 3D Graphics, p. 159-166, 2001.

Apêndice A. Ferramentas

Durante a pesquisa feita para esta dissertação, várias ferramentas foram desenvolvidas:

- O Crab GPU Shader Model (Figura A.1) inspeciona as propriedades da GPU instalada no sistema.

- O Crab GPU Sim (Figura A.2) é um programa educacional para aprendizado das tecnologias do *Vertex Shader* e *Pixel Shader* da GPU, por meio de uma simulação interativa da própria GPU.

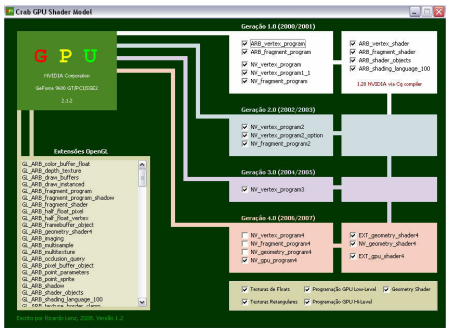


Figura A.1. Crab GPU Shader Model.

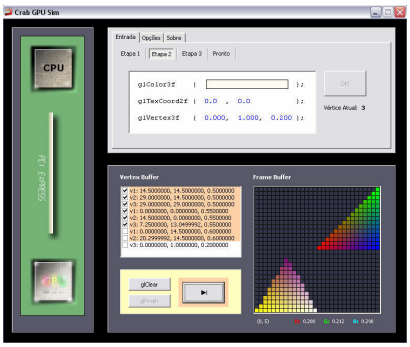


Figura A.2. Crab GPU Sim.

- O Crab GPU Streams (Figura A.3) é um programa de inspeção do desempenho da GPU, realizando operações básicas de matemática e trigonometria e relatando os tempos gastos; uma comparação, para as mesmas tarefas, é feita também com a CPU. Medições sobre a velocidade de transferência para a GPU, e da GPU, também são relatadas.

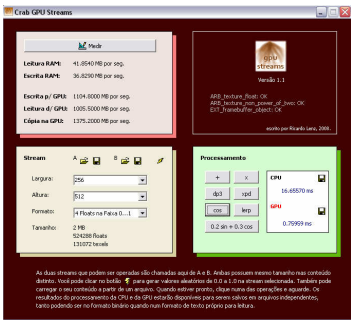


Figura A.3. Crab GPU Streams.

- O Crab GPU Studio (Figura A.4) fornece um ambiente completo para experimentação com os mecanismos programáveis da GPU, como o *Vertex Shader*, *Pixel Shader* e *Geometry Shader*, por meio da linguagem de alto nível GLSL. As mesmas opções de programação também são disponibilizadas para a linguagem de baixo nível correspondente na OpenGL. O resultado da programação é visto de modo imediato: o que o usuário digita no painel da direita, é executado em tempo real no painel da esquerda. Por causa disso, a rápida prototipagem de algoritmos com GPU é uma das atrações do programa, inclusive algoritmos que trabalhem com malhas geométricas.

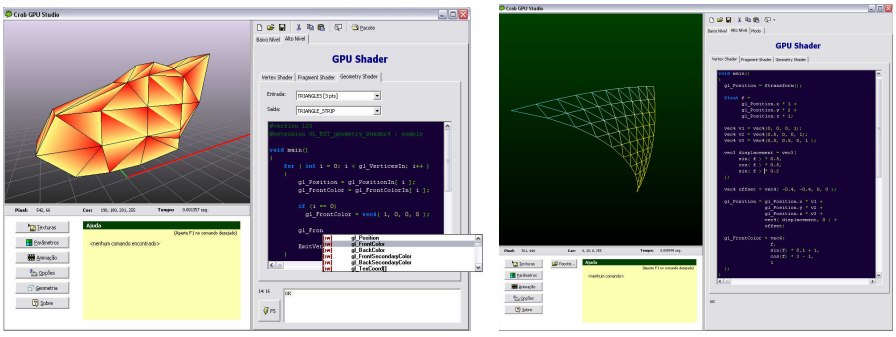


Figura A.4. Crab GPU Studio.

- O Gerador de Padrões Topológicos (Figura A.5) gera os arquivos dos padrões topológicos usados no método desta dissertação. O gerador suporta padrões adaptativos e pode gerar os arquivos em vários formatos diferentes.

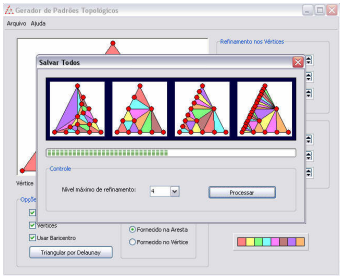


Figura A.5. Gerador de Padrões Topológicos.

- O Crab GPU Mesh Studio (Figura A.6) é o ambiente criado para a implementação do método desta dissertação. O programa contém diversos métodos de refinamento de malhas com GPU e suas variações implementados, além de alguns operadores de curvatura. É fornecido um controle total sobre a malha, a medição do desempenho, as opções de renderização e os parâmetros do refinamento.

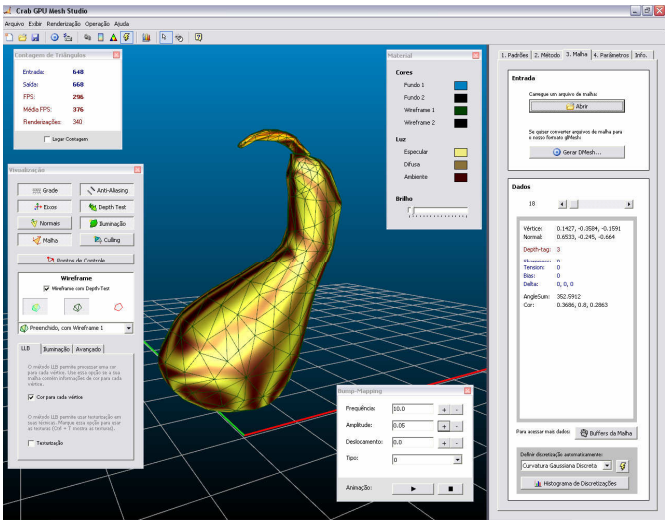


Figura A.6. Crab GPU Mesh Studio.

Apêndice B. Implementação

Neste trabalho, uma implementação foi feita usando a biblioteca OpenGL e a linguagem GLSL (usada para a programação do *Vertex Shader* e demais estágios programáveis da GPU). Na implementação realizada, diversas estruturas de dados foram utilizadas. Neste Apêndice serão comentados algumas estruturas da implementação e uma parte do código necessário para implantá-las.

Como a arquitetura gráfica da GPU trabalha com um nível de estrutura de dados bastante simples, foi optado seguir esse mesmo paradigma também na implementação deste trabalho. Dessa forma, a malha é definida, aqui, como um conjunto de vértices e um conjunto de faces.

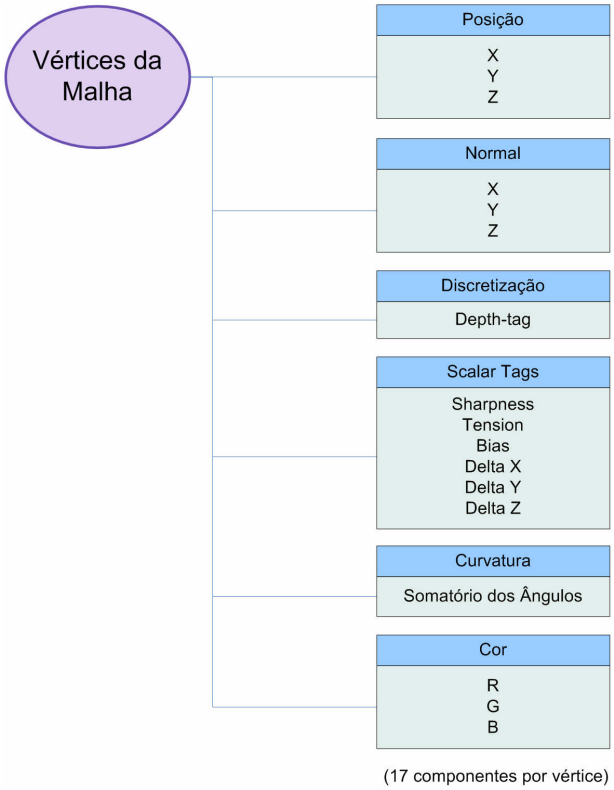


Figure B.1. Estrutura de um vértice da malha

Dessa forma, a primeira estrutura fundamental é a de um vértice da malha, descrito conforme a Figura B.1. Nessa estrutura, cada campo é um valor numérico, e, aqui, um dos

atributos mais importantes para o método reside no campo chamado de "Depth-tag". Esse campo define o nível de discretização desejado para a malha, na posição do vértice em questão. Os demais campos podem ou não ser utilizados, conforme a técnica escolhida. Por exemplo, para a técnica de *Curved PN Triangle*, os campos de "Scalar Tags" não são usados; por outro lado, esses mesmos campos são usados na técnica de *ST-Mesh*.

A segunda estrutura fundamental da implementação é a que define uma face da malha. Como a malha é triangularizada, todas as face de sua estrutura subjacente são consideradas triangulares também. A Figura B.2 descreve a estrutura de uma face.

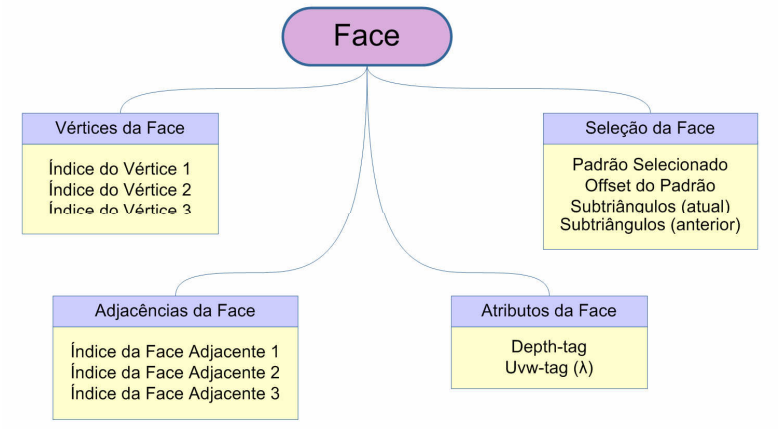


Figura B.2. Estrutura de uma face da malha.

Conforme pode ser constatado no esquema da Figura B.2, a malha é descrita de maneira indexada, o que significa que cada vértice de uma face é referenciado pelo seu respectivo índice. Dessa forma, evita-se ter que adicionar diversas instâncias de um mesmo vértice para um mesmo ponto geométrico da malha. Além disso, isso facilita a manipulação de certos parâmetros de refinamento adotados em nível de vértice.

Como as faces são sempre triangulares, isso significa, também, que há somente três faces adjacentes para cada face da malha. Assim, essas três adjacências já ficam armazenadas na descrição da malha e podem ser usadas para diversos fins, como, por exemplo, para o cálculo das curvaturas. E, por fim, um dos campos mais importantes de uma face é o chamado "Uvw-tag". Esse é o nome que foi escolhido, na implementação, para se referir ao λ (ver Seção 4.3).

À parte das estruturas fundamentais do vértice e da face, uma outra preocupação surge durante a implementação: nem todas as estruturas de dados relativas à malha poderão ser usadas ao mesmo tempo. Isso ocorre porque há um limite no número de Texturas de *Buffer* (OpenGL

Buffer Textures) que depende da arquitetura da GPU sendo utilizada. As Texturas de *Buffer* são blocos de memória que a GPU pode acessar. Embora existam outros mecanismos de acesso a dados por parte da GPU, o mecanismo das Texturas de *Buffer* é considerado, no presente momento, um dos mais rápidos e flexíveis para grandes quantidades de dados residentes num único *buffer*. Assim, o seu uso se justifica, mesmo que hajam limites no número de *buffers* que poderão ser acessados dentro de um mesmo algoritmo.

No caso da arquitetura da GPU utilizada na implementação deste trabalho, somente até 4 *Buffer Textures* poderiam ser acessados num mesmo algoritmo, dentro da GPU. Assim, o método ARKFPI, por exemplo, foi projetado de tal forma que apenas 3 *Buffer Textures* são utilizados. No DMR, porém, 4 *Buffers Textures* são usados. A Figura B.3 ilustra essa questão.

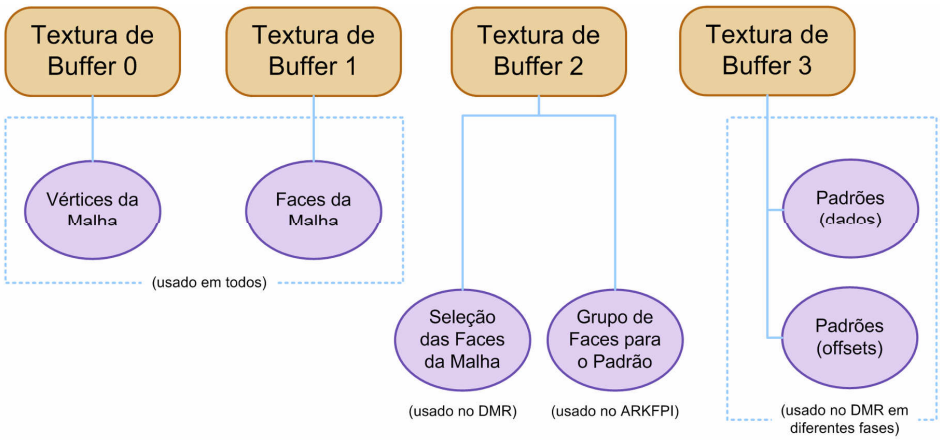


Figura B.3. Esquema geral dos buffers.

Conforme pode ser visualizado na Figura B.3, as duas primeiras unidades de Textura de *Buffer* são sempre alocadas para os *buffers* mais fundamentais da malha, como o conjunto dos vértices e o conjunto das faces. As outras duas unidades, contudo, são alocadas para *buffers* específicos de certos métodos. Por exemplo, no caso do ARKFPI, usa-se um determinado *buffer* na unidade de Textura de *Buffer* 2, enquanto que, no DMR, usa-se outro *buffer*. Esses *buffers* são determinados em tempo de execução, antes de se executar um ou o outro método. Assim, quando se executa o método ARKFPI, apenas as três primeiras unidades são utilizadas; no DMR, todas as quatro unidades são utilizadas.

Há um *buffer*, usado pelo ARKFPI, que é indicado na Figura B.3 como "Grupo de Faces para o Padrão". Esse *buffer* contém os índices das faces que compartilham um mesmo padrão base, e é preenchido no momento da execução do método. Dessa forma, para cada chamada de

renderização com *Instancing* de um determinado padrão base [i, j, k], esse *buffer* é devidamente preenchido e então usado, no algoritmo que reside na GPU, para acessar os dados da malha relativos a cada instância. Por exemplo, se um determinado padrão base [i, j, k] é instanciado 30 vezes (isto é, existem 30 faces que usam esse mesmo padrão base), a instância de número 14 saberá qual face da malha ela deverá usar por meio desse *buffer*, simplesmente acessando-se o 14º item desse mesmo *buffer*. O item acessado fornecerá um número, que é o índice da face para aquela instância. Por exemplo, se o item lido for o número 287, então a face 287 da malha é a que está sendo processada nessa 14ª instância. Com essa informação obtida, então, as faces são lidas por meio do *buffer* indicado como "Faces da Malha" na Figura B.3. Da mesma maneira, os vértices são acessados por meio do *buffer* indicado como "Vértices da Malha", na mesma figura. A Figura B.4 ilustra esse esquema de acesso aos dados da malha, conforme discutido aqui.

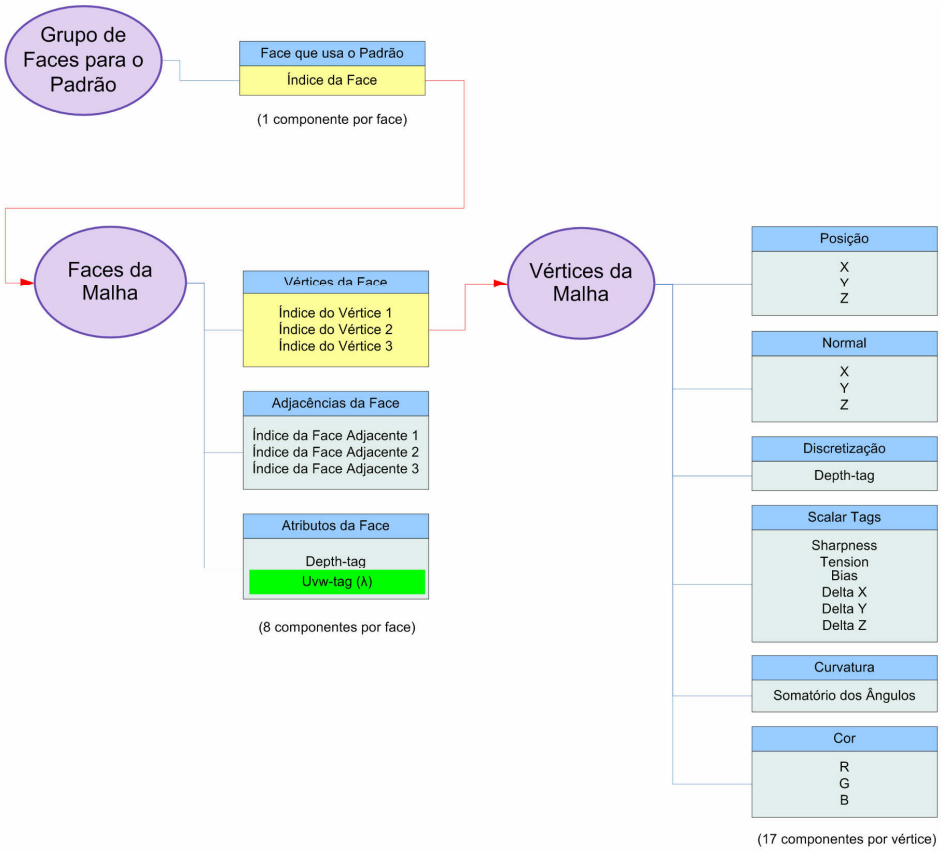


Figura B.4. Acesso aos dados da malha.

Na Figura B.4, os campos indicados em amarelo fornecem índices, que são usados para acessar outras estruturas. Dessa forma, cada item do *buffer* indicado como "Grupo de Faces para o Padrão" contém o campo "Índice da Face". Esse índice é usado para acessar um item do *buffer* indicado como "Faces da Malha"; cada item desse *buffer* é uma face da malha. Os campos "Índice do Vértice 1", "Índice do Vértice 2", etc. de uma face são usados para acessar itens individuais do *buffer* indicado como "Vértices da Malha", onde cada item, aqui, é um vértice da malha, com seus atributos típicos como "Posição" e "Normal".

O campo indicado em verde, na Figura B.4, é de importância vital no método do ARKFPI. Ele é o campo do "Uvw-tag", que define o λ da face em questão. Esse campo existe para cada face da malha, pois cada face possui o seu próprio λ.

As questões discutidas aqui sobre as estruturas de dados e como elas se relacionam umas com as outras, no âmbito da implementação com GPU, é melhor evidenciada por meio da apresentação do próprio código-fonte. Por causa disso, um exemplo de implementação em GPU é apresentado, a seguir, para uma das técnicas usadas neste trabalho, a do *Curved PN Triangle*. Essa implementação está escrita na linguagem GLSL, e descreve como o mecanismo do *Vertex Shader* deve atuar para aplicar a técnica do *Curved PN Triangle* nos moldes do método ARKFPI.

```
/*
   Vertex Shader, em GLSL, do Curved PN Triangle,
   implementado em ARKFPI, por Ricardo Lenz.
*/


#define componentsPerVertex   17
#define compPx                0
#define compPy                1
#define compPz                2
#define compNx                3
#define compNy                4
#define compNz                5
#define compCr                14
#define compCg                15
#define compCb                16

#define componentsPerFaceInfo 8
#define compIndexFaceVert0    0
#define compIndexFaceVert1    1
#define compIndexFaceVert2    2
#define compUvwTag            7


uniform samplerBuffer meshVerts;
uniform isamplerBuffer meshFaces;
uniform isamplerBuffer patternFaceIndices;
```

```
uniform bool quadratic_normals;
attribute vec3 vertex_uvw;

vec3 p1, p2, p3;
vec3 n1, n2, n3;
int uvwtag;



float Wij(vec3 Pi, vec3 Pj, vec3 Ni)
{
    return dot( Pj - Pi, Ni );
}


vec3 pn_pos(vec3 pesos)
{
    vec3 b300, b030, b003;

    b300 = p1;
    b030 = p2;
    b003 = p3;

    vec3 b210, b120, b021, b012, b102, b201, b111;
    vec3 e, vv;

    b210 = (2*p1 + p2 - Wij( p1, p2, n1 ) * n1 ) / 3;
    b120 = (2*p2 + p1 - Wij( p2, p1, n2 ) * n2 ) / 3;
    b021 = (2*p2 + p3 - Wij( p2, p3, n2 ) * n2 ) / 3;
    b012 = (2*p3 + p2 - Wij( p3, p2, n3 ) * n3 ) / 3;
    b102 = (2*p3 + p1 - Wij( p3, p1, n3 ) * n3 ) / 3;
    b201 = (2*p1 + p3 - Wij( p1, p3, n1 ) * n1 ) / 3;

    e = (b210 + b120 + b021 + b012 + b102 + b201) / 6;
    vv = (p1 + p2 + p3) / 3;
    b111 = e + (e - vv) * 0.5;

    float u = pesos.y;
    float v = pesos.z;
    float w = pesos.x;

    vec3 resp =
         (b300 * w * w * w) +
         (b030 * u * u * u) +
         (b003 * v * v * v) +
         (b210 * 3 * w * w * u) +
         (b120 * 3 * w * u * u) +
         (b201 * 3 * w * w * v) +
         (b021 * 3 * u * u * v) +
         (b102 * 3 * w * v * v) +
         (b012 * 3 * u * v * v) +
         (b111 * 6 * w * u * v);

    return resp;
}
```

```
vec3 media(vec3 pesos, vec3 v1, vec3 v2, vec3 v3)
{
    vec3 v = vec3(
        v1.x * pesos.x  +  v2.x * pesos.y  +  v3.x * pesos.z,
        v1.y * pesos.x  +  v2.y * pesos.y  +  v3.y * pesos.z,
        v1.z * pesos.x  +  v2.z * pesos.y  +  v3.z * pesos.z
    );

    return v;
}



vec3 reflection(vec3 A, vec3 B)
{
    float v = dot( B, A ) / dot( B, B );

    return A - 2 * v * B;
}



vec3 pn_normal(vec3 pesos)
{
    if ( !quadratic_normals )
         return media( pesos, n2, n3, n1 );

    // o curved pn triangle usa uma notacao invertida de (u, v, w).
    // ele usa (v, w, u). por causa disso, invertemos aqui.
    float u, v, w;
    u = pesos.y;
    v = pesos.z;
    w = pesos.x;

    vec3 n200, n020, n002, n110, n011, n101;

    n200 = n1;
    n020 = n2;
    n002 = n3;

    n110 = normalize( reflection( n1 + n2, p2 - p1 ) );
    n011 = normalize( reflection( n2 + n3, p3 - p2 ) );
    n101 = normalize( reflection( n3 + n1, p1 - p3 ) );

    vec3 resp =
         (n200 * w * w) +
         (n020 * u * u) +
         (n002 * v * v) +
         (n110 * w * u) +
         (n011 * u * v) +
         (n101 * w * v);

    return resp;
}
```

```
vec3 permuta_pesos( vec3 uvw )
{
    float u = uvw.x;
    float v = uvw.y;
    float w = uvw.z;


    if ( uvwtag == 1 ) return vec3( w, v, u );
    if ( uvwtag == 2 ) return vec3( u, w, v );
    if ( uvwtag == 3 ) return vec3( w, u, v );
    if ( uvwtag == 4 ) return vec3( v, u, w );
    if ( uvwtag == 5 ) return vec3( w, u, v );
    if ( uvwtag == 6 ) return vec3( v, w, u );
    if ( uvwtag == 7 ) return vec3( u, w, v );
    return vec3( u, v, w );
}

float get_float(int vert, int comp)
{
    return texelFetchBuffer(
         meshVerts,
         vert * componentsPerVertex + comp
    ).x;
}

vec3 get_vec(int vert, int comp1)
{
    return vec3(
         get_float( vert, comp1 ),
         get_float( vert, comp1 + 1 ),
         get_float( vert, comp1 + 2 )
    );
}

void le_dados()
{
    int face = texelFetchBuffer(
         patternFaceIndices,
         gl_InstanceID
    ).x;


    int face_v0_index = texelFetchBuffer(
               meshFaces,
               (face * componentsPerFaceInfo) + compIndexFaceVert0
    ).x;

    int face_v1_index = texelFetchBuffer(
               meshFaces,
               (face * componentsPerFaceInfo) + compIndexFaceVert1
    ).x;

    int face_v2_index = texelFetchBuffer(
               meshFaces,
               (face * componentsPerFaceInfo) + compIndexFaceVert2
    ).x;
```

```
    uvwtag = texelFetchBuffer(
         meshFaces,
         (face * componentsPerFaceInfo) + compUvwTag
    ).x;

    p1 = get_vec( face_v0_index, compPx );
    p2 = get_vec( face_v1_index, compPx );
    p3 = get_vec( face_v2_index, compPx );

    n1 = get_vec( face_v0_index, compNx );
    n2 = get_vec( face_v1_index, compNx );
    n3 = get_vec( face_v2_index, compNx );
}


void main()
{
    le_dados();

    vec3 pesos = permuta_pesos( vertex_uvw );
    vec3 pos = pn_pos( pesos );
    vec3 normal = pn_normal( pesos );

    gl_TexCoord[0] = vec4( normal, 1 );
    gl_Position = gl_ModelViewProjectionMatrix * vec4( pos, 1 );
}
```