RENDERIZAÇÃO EM TEMPO REAL DE PELOS APLICADOS SOBRE MALHAS TRIDIMENSIONAIS ARBITRÁRIAS OBTIDAS DE OBJETOS REAIS

BRUNO PEREIRA EVANGELISTA
Orientador: Renato Antônio Celso Ferreira

RENDERIZAÇÃO EM TEMPO REAL DE PELOS APLICADOS SOBRE MALHAS TRIDIMENSIONAIS ARBITRÁRIAS OBTIDAS DE OBJETOS REAIS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte
Agosto de 2009

© 2009, Bruno Pereira Evangelista.
Todos os direitos reservados.

Evangelista, Bruno Pereira
E92r Renderização em tempo real de pelos aplicados sobre malhas tridimensionais arbitrárias obtidas de objetos reais / Bruno Pereira Evangelista. — Belo Horizonte, 2009
xxii, 112 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Renato Antônio Celso Ferreira

1. Computação gráfica - Teses. 2. Renderização em tempo real - Teses. 3. Processamento eletrônico de dados em tempo real - Teses. I. Título.

CDU 519.6*83(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Renderização Em Tempo Real de Pêlos Aplicados Sobre Malhas Tridimensionais Arbitrárias Obtidas de Objetos Reais

BRUNO PEREIRA EVANGELISTA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. RENATO ANTÔNIO CELSO FERREIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. CLAUDIO ESPERANÇA
Universidade Federal do Rio de Janeiro-UFRJ

PROF. LUIZ CHAIMOWICZ
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 25 de setembro de 2009.

Agradecimentos

Dois anos se passaram desde que eu comecei a trabalhar nesta dissertação, e durante todo esse tempo inúmeras mudanças aconteceram em minha vida. Por todo o apoio, cumplicidade e compreensão nestes dois anos eu gostaria de agrader:

- Deus, que sempre me mostrou o caminho, me deu saúde e permitiu que eu chegasse até aqui.
- Aos meus pais Kathia e Gledson, e ao meu padrasto Cláudio e minha madrasta Célida, que sempre me incentivaram a ir mais longe e nunca me deixaram desistir.
- A minha companheira Helenice, por toda a compreensão e cumplicidade.
- Ao meu orientador Renato, que mesmo estando longe me ajudou a tomar decisões importantes, e foi compreensível com os atrasos que esse trabalho sofreu.
- Ao CEO da Tectoy Digital André Penha, pelas inúmeras coisas que aprendi na Tectoy, e por ter possibilitado que eu tivesse tempo para conciliar estudo e trabalho.
- Ao Alessandro Ribeiro e Wallace Lages, por sempre terem tido um tempinho no MSN para ler coisas sobre o meu trabalho.
- E por último, mas não menos importante, as inúmeras pessoas que me inspiram e me fazem querer ir SEMPRE mais longe.

"De fato, que aproveitará ao homem ganhar o mundo inteiro mas perder sua alma?"
(Mateus 16, 26)

Resumo

Em ambientes virtuais a presença de cabelos e pelos é fundamental para a criação de humanos e criaturas virtuais realistas. Este trabalho apresenta um algoritmo para criação e aplicação automática de pelos sobre malhas arbitrárias obtidas a partir de objetos do mundo real sem a necessidade de intervenção humana. O algoritmo proposto pode ser aplicado sobre malhas tridimensionais de topologia arbitrária, construídas como arranjos de faces triangulares. Para resolver os problemas relacionados ao desempenho na renderização de pelos, este trabalho também apresenta um algoritmo de pré-processamento, o qual permite a redução e otimização dos dados de malhas arbitrárias aumentando o desempenho da renderização em processadores gráficos modernos.

Para avaliar a eficácia do algoritmo proposto, o mesmo foi testado com um conjunto de malhas de alta complexidade geométrica, obtidas a partir de quatro diferentes objetos reais através do uso de *scanners* tridimensionais, e obtidos através do repositório de *scanning* 3D da universidade de Stanford.

Aplicativos de computação gráfica, em especial jogos e simuladores, têm buscado criar ambientes virtuais com níveis de realismo tão altos que não possam ser diferenciados do mundo real. Nestes ambientes, a presença de cabelos e pelos é fundamental para a criação de humanos e criaturas virtuais realistas.

Este trabalho foca a criação, aplicação e renderização de pelos sobre malhas arbitrárias obtidas a partir de objetos reais. Mais especificamente, este trabalho foca a resolução de um problema pouco abordado pelos trabalhos científicos da área: a criação e aplicação automática de pelos sobre malhas arbitrárias, sem necessidade de intervenção humana. Normalmente o processo de criação e aplicação de pelos é realizado por artistas através do uso de ferramentas de modelagem tridimensional, e pode demandar horas ou dias para ser finalizado. Este trabalho também foca a renderização de malhas contêndo pelos em tempo real, o que permite o uso de pelos em aplicativos de tempo real como jogos e simuladores.

O algoritmo proposto pode ser aplicado sobre malhas tridimensionais de topologia arbitrária, construídas como arranjos de faces triangulares. Para resolver os problemas relacionados ao desempenho na renderização de pelos, este trabalho apresenta um

algoritmo de pré-processamento, o qual permite a redução e otimização dos dados da malha para aumentar o desempenho da renderização em processadores gráficos modernos.

Para avaliar a eficácia do algoritmo proposto, o mesmo foi testado com um conjunto de malhas de alta complexidade geométrica, obtidas a partir de quatro diferentes objetos reais através do uso de *scanners* tridimensionais, e obtidos através do repositório de *scanning* 3D da universidade de Stanford. O algoritmo proposto tambem foi testado com versões simplificadas das malhas dos objetos reais, as quais foram geradas a partir do uso de algoritmos de LOD [Luebke et al., 2002]. A partir dos testes realizados com essa malhas é possível demonstrar que o algoritmo proposto é robusto e pode ser aplicado a malhas arbitrárias, incluindo malhas com alta e baixa complexidade geométrica.

Para avaliar a qualidade visual e o desempenho do algoritmo proposto foi criado um ambiente virtual que permite a visualização em tempo real das malhas onde os pelos foram aplicados, e a modificação em tempo real de câmeras, luzes e os parâmetros utilizados na criação, aplicação e renderização dos pelos.

Abstract

In this work we present an algorithm to create and apply fur over arbitrary meshes extracted from real world objects. We focus on the solution of the problem of automatic apply fur over meshes without human intervention. Our algorithm was tested with four different high complexity meshes, which were obtained from the Stanford scanning repository. We also tested our algorithm using three differents levels of detail generated from each of the original meshes evaluated. Based on our tests we can prove that our approach is robust and can be applied to arbitrary meshes with high or low geometry complexity.

The environment created for fur rendering allows the modification of any parameter related to the fur generation (such as: height, thickness, density or color) or fur rendering (such as: camera, light, number of layers) in real time.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Aplicativos de computação gráfica, em especial jogos e simuladores, têm buscado criar ambientes virtuais com níveis de realismo tão altos que não possam ser diferenciados do mundo real. Nestes ambientes, a presença de cabelos e pelos é fundamental para a criação de humanos e criaturas virtuais realísticas. No entanto, a renderização de cabelos e pelos ainda apresenta vários desafios devido ao grande volume de dados, espessura das superfícies e espaçamento entre as mesmas [Yang et al., 2008].

O cabelo de um humano possui cerca de 100.000 fios [Ward et al., 2007], enquanto o pelo de uma lontra chega a ter 200.000 fios por centímetro quadrado [Wikipedia, 2009]. Esse alto nível de detalhe presente em criaturas do mundo real também pode ser visto em ambientes virtuais, como por exemplo, no cômico herói Scrat do filme "A Era do Gelo 2: O degelo" apresenta uma malha contendo mais de 1.3 milhões de pelos modelados como curvas B-splines [Swaaij, 2006]. A Figura 1.1 apresenta uma renderização do personagem virtual Scrat.

Os desafios presentes no uso de cabelos e pelos em criaturas virtuais não estão relacionados apenas à sua renderização, mas também à sua modelagem, estilização, simulação e iluminação [Kim e Neumann, 2002, Nguyen e Donnelly, 2005, Cani e Bertails, 2006, Ward et al., 2007, Sintorn e Assarsson, 2008].

Este trabalho foca a criação, aplicação e renderização de pelos sobre malhas arbitrárias obtidas a partir de objetos reais. Mais especificamente, este trabalho foca a resolução de um problema pouco abordado pelos trabalhos científicos da área: a criação e aplicação automática de pelos sobre malhas arbitrárias, sem necessidade de intervenção humana. Normalmente o processo de criação e aplicação de pelos é realizado por artistas através do uso de ferramentas de modelagem tridimensional, e pode demandar horas ou dias para ser finalizado. Este trabalho também foca a renderização de malhas contendo pelos em tempo real, o que permite o uso de pelos em aplicativos de tempo real como jogos e simuladores.

Figura 1.1. Scrat. Herói cômico do filme "A Era do Gelo 2: O degelo" [Swaaij, 2006]. Malha contendo mais de 1.3 milhões de pelos.

1.1 Renderização de pelos

Algoritmos de rasterização e traçado de raios (*ray-tracing*) [Akenine-Möller et al., 2008] são utilizados para geração de uma imagem bidimensional a partir da descrição de uma cena tridimensional, processo denominado renderização. No entanto, algoritmos tradicionais de rasterização e traçado de raios apresentam problemas na renderização de superfícies muito pequenas e de baixa espessura, como pelos. Os problemas apresentados por esses algoritmos tornam-se mais evidentes quando a área da superfície renderizada é inferior a área de um *pixel* da imagem gerada. Devido a esses problemas, as imagens geradas tendem a apresentar alto nível de serrilhamento e baixa qualidade.

Os problemas apresentados na renderização de geometrias muito pequenas, podem ser parcialmente resolvidos com o uso de técnicas de *anti-aliasing* [Beaudoin e Poulin, 2004]. Utilizando a técnica de superamostragem, por exemplo, pode-se renderizar imagens com dimensões superiores às desejadas (forçando a geração de um número maior de *pixels*), o que permite uma melhor amostragem das geometrias presentes em uma cena. A principal desvantagem das técnicas de *anti-aliasing* se deve às mesmas apresentarem um alto custo computacional e não conseguirem resolver completamente os problemas de serrilhamento presentes em uma cena.

Devido aos problemas encontrados na renderização de pequenas geometrias, novas abordagens foram propostas para esse tipo de renderização. As técnicas de renderização de pelos estudadas neste trabalho não se baseiam na renderização direta da geometria dos pelos, mas na renderização de volumes contendo dados pré-processados dos pelos [Kajiya e Kay, 1989, Neyret, 1995, Neyret, 1996, Neyret, 1998, Porumbescu et al., 2005, Swaaij, 2006, Jeschke et al., 2007], e

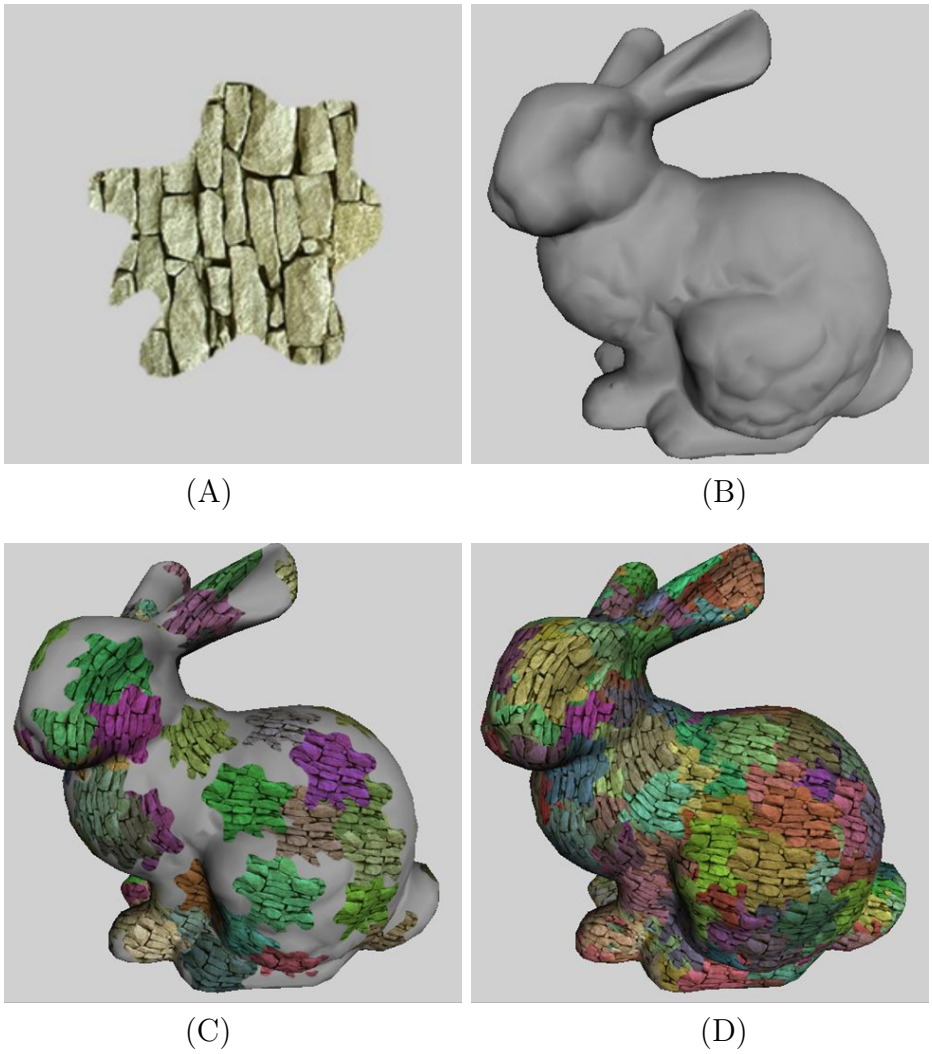
na renderização de superfícies paralelas contendo dados discretizados de um volume [Lacroute, 1995, Meyer e Neyret, 1998, Lengyel e Praun, 2001, Isidoro e Mitchell, 2002, Decaudin e Neyret, 2004, Sheppard, 2004, Wyatt, 2007, Yang et al., 2008].

1.2 Aplicação de pelos a Malhas Arbitrárias

Em 2001 Lengyel [Lengyel e Praun, 2001] apresentou um algoritmo para aplicação de pelos a malhas arbitrárias, sem nenhum tipo de restrição topológica. O algoritmo proposto por Lengyel divide a malha arbitrária em um conjunto de superfícies parametrizadas e, para cada superfície parametrizada, um trecho da textura (ou volume) onde os pelos estão armazenadas é aplicado sobre a mesma. Este processo é repetido recursivamente até que toda a malha tenha sido coberta por pelos. A Figura 1.2 ilustra o processo de parametrização e aplicação de um trecho de textura sobre uma malha arbitrária.

Após os pelos terem sido aplicados sobre toda malha, os mesmos são renderizados como múltiplas camadas paralelas da malha, dando a impressão visual da renderização de um volume. O algoritmo proposto por Lengyel é uma versão modificada do algoritmo de *Lapped Textures* [Praun et al., 2000], pois não permite que trechos parametrizados da malha tenham triângulos sobrepostos, o que é desejado no algoritmo original de *Lapped Textures* pois reduz as distorções entre junções de superfícies parametrizadas.

O algoritmo proposto por Lengyel consegue gerar imagens com boa qualidade visual mas apresenta várias limitações. Inicialmente, é preciso definir o formato do trecho da textura (ou volume) de pelos a ser repetido sobre as parametrizações da malha criadas, e em seguida, é necessário definir vetores de crescimento das parametrizações sobre a malha arbitrária. Esses dois passos necessitam da intervenção humana, sendo geralmente realizados por artistas. Por último, as parametrizações precisam ser geradas para cada camada da malha a ser renderizada. Desta maneira, para a renderização de pelos utilizando 16 camadas, por exemplo, é necessário a parametrização de 32 malhas diferentes. Note que o algoritmo de Lengyel permite que o trecho da textura de pelos seja aplicado automaticamente sobre a malha caso os pelos sejam homogêneos, o que acontece no caso dos pelos serem estreitos. Outra desvantagem dessa técnica se deve à divisão da malha arbitrária em um conjunto de parametrizações descontínuas, onde os dados dos vértices adjacentes entre duas parametrizações são diferentes, não permitindo a remoção de dados duplicados da malha e impossibilitando outros tipos de otimização na renderização da malha.



(A) (B)

(C) (D)

Figura 1.2. Parametrização e aplicação de um trecho de textura sobre uma malha arbitrária. (A) trecho de textura, (B) malha arbitrária, (C) conjunto intermediário de parametrizações e (D) resultado final.

1.3 Objetivos

O principal objetivo deste trabalho é a criação e aplicação automática de pelos sobre malhas arbitrárias obtidas a partir de objetos do mundo real, e a renderização dessas malhas em tempo real.

1.4 Metodologia

Para resolver o problema de criação e aplicação automática de pelos sobre malhas arbitrárias, este trabalho propõe um algoritmo procedural para a geração dos pêlos, e uma extensão do algoritmo de mapeamento de ambiente [Akenine-Möller et al., 2008] para o mapeamento dos pelos em tempo real. A renderização dos pelos é feita a partir

da renderização de múltiplas camadas paralelas da malha onde os mesmo são aplicados, o que simula a renderização de um volume composto por várias camadas discretas da malha. Esta técnica é apresentada em detalhes na Seção [**?**].

O algoritmo proposto pode ser aplicado sobre malhas tridimensionais de topologia arbitrária, construídas como arranjos de faces triangulares. Para resolver os problemas relacionados ao desempenho na renderização de pelos, este trabalho apresenta um algoritmo de pré-processamento, o qual permite a redução e otimização dos dados da malha para aumentar o desempenho da renderização em processadores gráficos modernos.

Para avaliar a eficácia do algoritmo proposto, o mesmo foi testado com um conjunto de malhas de alta complexidade geométrica, obtidas a partir de quatro diferentes objetos reais através do uso de *scanners* tridimensionais, e obtidos através do repositório de *scanning* 3D da universidade de Stanford. O algoritmo proposto tambem foi testado com versões simplificadas das malhas dos objetos reais, as quais foram geradas a partir do uso de algoritmos de LOD [Luebke et al., 2002]. A partir dos testes realizados com essas malhas é possível demonstrar que o algoritmo proposto é robusto e pode ser aplicado a malhas arbitrárias, incluindo malhas com alta e baixa complexidade geométrica.

Para avaliar a qualidade visual e o desempenho do algoritmo proposto foi criado um ambiente virtual que permite a visualização em tempo real das malhas onde os pelos foram aplicados, e a modificação em tempo real de câmeras, luzes e os parâmetros utilizados na criação, aplicação e renderização dos pelos.

1.5 Motivação

Os trabalhos de renderização de pelos estudados e referenciados neste trabalho, com excessão do trabalho de Lengyel [Lengyel e Praun, 2001], não consideram a criação e aplicação automática de pelos sobre malhas arbitrárias. Esses trabalhos se baseiam na renderização de pelos a partir de malhas tridimensionais, formadas por um conjunto de triângulos ou a partir de superfícies paramétricas [Kajiya e Kay, 1989], as quais geralmente precisam atender a vários requisitos, como: ser contínua (não havendo presença de buracos), possuir vetores normais contínuos entre faces adjacentes, possuir coordenadas de mapeamento de textura, dentre outros. Todos esses requisitos fazem com que não seja possível a aplicação automática de pelos as malhas, sendo necessário um prévio processamento da malha por um artista utilizando uma ferramenta de modelagem tridimensional.

Outra limitação observada em várias das técnicas de renderização de pelos estuda-

das, se deve as mesmas não levarem em conta o *hardware* dos processadores gráficos modernos. Desta maneira, a construção e renderização dos pelos não leva em conta fatores como: tamanho dos dados de cada vértice da malha, indexação dos dados, e reordenação dos dados para maximizar uso do *cache* dos processadores gráficos. Neste trabalho, os algoritmos propostos visam tirar proveito do *hardware* dos processadores gráficos modernos, e com isso obter um maior desempenho.

1.6 Organização do Trabalho

A Seção 2 apresenta uma revisão bibliográfica dos trabalho relacionados a renderização de pelos através de volumes e camadas discretas, e os trabalhos relacionados a métodos otimizados para renderização de malhas. A Seção 3 apresenta as malhas arbitrárias obtidas a partir de objetos reais utilizadas neste trabalho, o processo de reconstrução dessas malhas e o processo de geração dos níveis de detalhe das malhas. A Seção 4 apresenta o pré-processamento aplicado sobre as malhas de entrada, o qual torna as mesmas aptas a renderização de pelos. A Seção 5 apresenta o algoritmo proposto para a geração procedural dos pelos, e o armazenamento dos pelos através de mapas de ambiente no formato de cubo e esfera. A Seção 6 apresenta o ambiente criado para visualização, edição e renderização dos pelos, e os algoritmos utilizados na renderização dos pelos. Finalmente, a Seção 7 apresenta os resultados obtidos e a conclusão do trabalho.

Capítulo 2

Trabalhos Relacionados

As técnicas de renderização de pelos estudadas neste trabalho não se baseiam na renderização direta de geometrias, mas na renderização de volumes contendo dados pré-processados dos pelos, e na renderização de camadas discretas e paralelas simulando a renderização de um volume. Esta seção apresenta uma revisão bibliográfica dos principais trabalhos relacionados a renderização de pelos através do uso de volumes, e camadas discretas.

2.1 Renderização de Volumes

Em 1989, Kajiya e Kay [Kajiya e Kay, 1989] apresentaram um algoritmo para renderização de pelos a partir de texturas tridimensionais. Neste algoritmo, os detalhes de micro escala presentes na superfície de um objeto são discretizados e armazenados na forma de um volume, onde cada *voxel* (elemento discreto do volume) armazena a densidade, orientação e *BRDF* (*Bidirectional Reflectance Distribution Function* - Função de distribuição de reflectância bidirecional) [Cook e Torrance, 1982] da superfície, ou combinações de superfícies, incidentes naquele ponto. Computacionalmente este volume é armazenado na forma de uma textura tridimensional, onde cada *texel* (elemente discreto da textura) armazena os dados de um *voxel*. A Figura 2.1 ilustra um padrão de pelos criado por Kajiya e Kay e armazenado na forma de um volume.

O volume de dados criado armazena apenas os detalhes da superfície de um objeto, e portanto, precisa ser mapeado em uma superfície base antes de poder ser renderizado. No trabalho de Kajiya e Kay a superfície base utilizada para renderização é construída a partir de *patches* bilineares, os quais não possuem coordenadas de mapeamento de textura. Desta maneira, a textura tridimensional contendo os detalhes da superfície é repetida para cada *patch* (ou quádrica) da superfície, sendo deformada para se ajustar

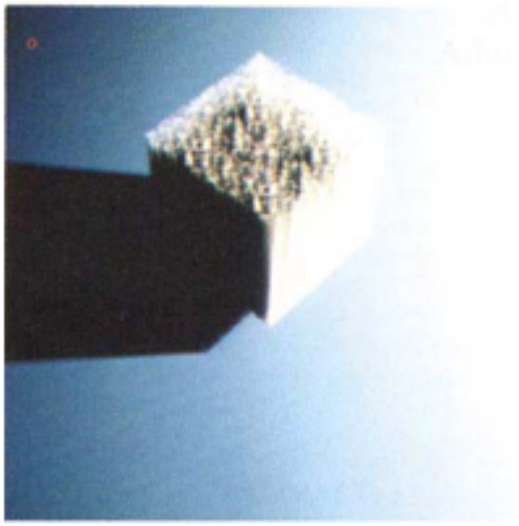


Figura 2.1. Padrão de pelos criado por Kajiya e Kay e armazenado na forma de um volume [Kajiya e Kay, 1989].

à dimensão e forma de cada quádrica. A Figura 2.2 ilustra o mapeamento de um volume sobre uma superfície criada utilizando quádricas.

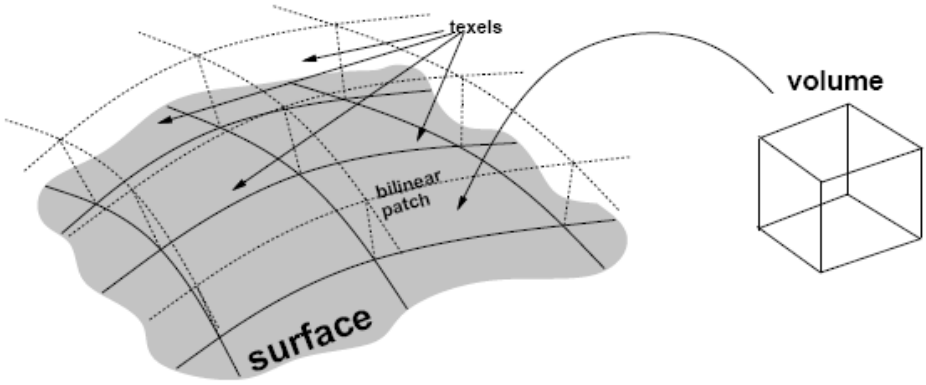


Figura 2.2. Mapeamento de um volume sobre uma superfícies criada utilizando quádricas [Neyret, 1998].

A repetição da textura tridimensional sobre cada quádrica da superfície base cria uma nova superfície paralela à superfície base. No entanto, a nova superfície criada pode apresentar buracos devido à curvatura da superfície base, como ilustrado na Figura 2.3. Para resolver esse problema, cada quádrica da superfície base possui quatro vetores de altura perpendiculares a superfície. Esses vetores são utilizados para deformar a textura tridimensional de acordo com sua altura, eliminando buracos e áreas sobrepostas. A Figura 2.3 ilustra o uso dos vetores de altura para corrigir buracos e áreas sobrepostas.

A renderização do volume contendo detalhes da superfície é feita utilizando um

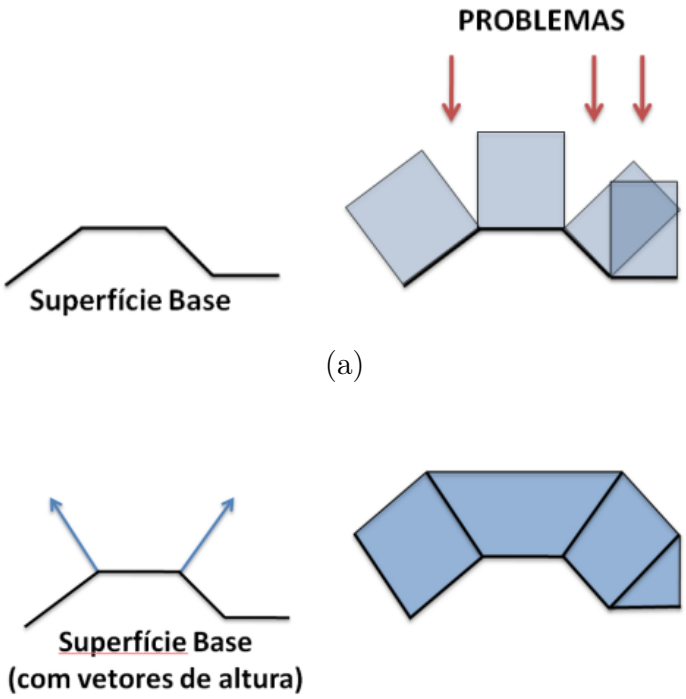


Figura 2.3. Repetição de uma textura tridimensional sobre uma superfície com curvatura. (a) Presença de buracos e sobreposição de volumes. (b) Uso de vetores de altura permite a criação de um volume contínuo sobre a superfície base.

algoritmo de *ray-tracing* [Akenine-Möller et al., 2008]. Para cada *pixel* da imagem gerado é lançado um raio, e para cada raio lançado são feitos testes de interseção contra todos os volumes mapeados sobre a superficie base. É importante notar que após um volume ser mapeado sobre uma quádrica, cada uma de suas faces pode ser representada por uma nova quádrica. Desta maneira, os testes de interseção se tornam mais simples, sendo feitos apenas entre raios e quádricas.

Para cada volume intersectado por um raio é calculado o ponto de entrada e saída do raio neste volume no espaço do mundo. As posições de entrada e saída do raio são posteriormente mapeadas do espaço de mundo para o espaço do volume. Em seguida, o interior do volume intersectado é percorrido linearmente, onde a luz refletida por cada *voxel* intersectado é calculada, utilizando a densidade, orientação e BRDF armazenados no mesmo. Finalmente, a luz total refletida é calculada somando-se a luz refletida por cada *voxel*. A Figura 2.4 ilustra o resultado do mapeamento do volume contendo um padrão pelos apresentado na Figura 2.1 para a malha tridimensional de um urso.

A abordagem utilizada por Kajiya e Kay, apesar de simples, possui diversas limitações. Devido a textura tridimensional que armazena os detalhes da superfície ser mapeada diretamente sobre cada *patch* da superfície base gerada, a mesma sofre gran-

Figura 2.4. Renderização de pelos aplicados a malha tridimensional de um urso modelado a partir de *patches* [Kajiya e Kay, 1989].

des deformações. Isso pode ser observado na Figura 2.4, onde partes diferentes do modelo apresentam diferentes densidades de pelos. Além disso, a superfície base precisa ser construída utilizando quádricas para que a textura tridimensional possa ser replicada sobre a mesma. Outra limitação é que a textura tridimensional criada não permite a representação de vários tipos de superfícies, como pelos anelados ou ondulados, limitando a técnica ao uso de cabelos estreitos e lisos. Por último, o algoritmo de *ray-tracing* utilizado requer que para cada raio traçado sejam feitos testes de interseção contra todos os volumes da cena, e dentro de cada volume contra todos os seus *voxels*. Isso demanda um grande tempo de processamento e pode restringir o uso desta técnica em aplicativos de tempo real.

A principal vantagem da técnica de Kajiya e Kay é permitir a renderização de detalhes de pequena escala, e a geração de imagens com baixo aliasing e maior qualidade.

2.1.1 Distribuição de Normais e Nível de Detalhe

Neyret [Neyret, 1995, Neyret, 1996, Neyret, 1998] apresentou várias extensões ao método de renderização de pelos proposto por Kajiya [Kajiya e Kay, 1989]. A técnica proposta por Neyret permite que diferentes tipos de superfícies, como folhas e árvores, sejam armazenadas na forma de texturas tridimensionais. Neyret também propõe que

coordenadas de mapeamento de textura sejam utilizadas para o mapeamento de texturas tridimensionais sobre a superfície base, permitindo maior controle na aplicação da textura e reduzindo (ou eliminando) possíveis distorções. Outra extensão proposta por Neyret é o uso de um esquema de nível de detalhe similar ao mip-mapping, permitindo otimizar o algoritmo de *ray-tracing*. Neyret propõe ainda três abordagens diferentes para animação dos detalhes armazenados em texturas tridimensionais. A Figura 2.5 ilustra diferentes geometrias armazenadas no modelo de textura tridimensional proposta por Neyret.

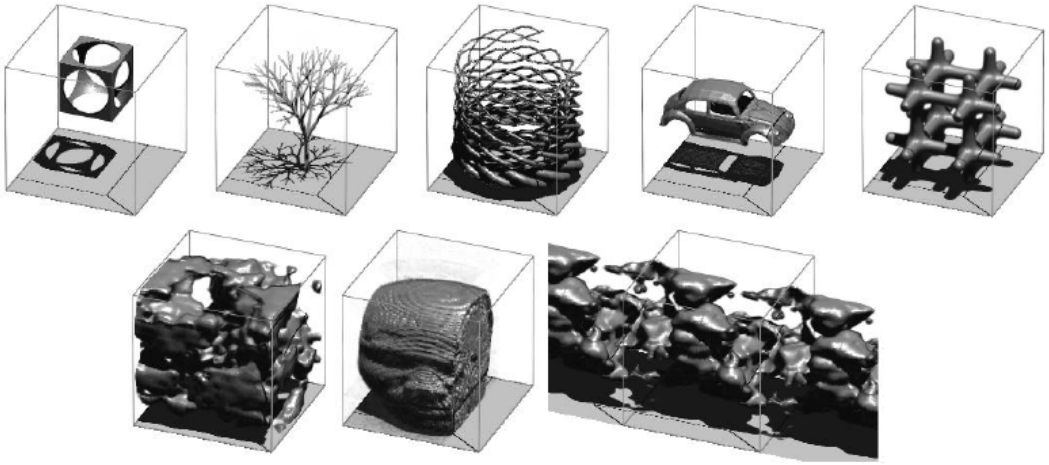


Figura 2.5. Diferentes geometrias armazenas em texturas tridimensionais. [Neyret, 1998].

Quando uma superfície é armazenada dentro de um volume discreto, várias partes da geometria da superfície podem incidir sobre um mesmo *voxel* do volume. Essas pequenas geometrias contidas dentro de cada *voxel*, podem ser representadas através do BRDF armazenado no voxel [Westin et al., 1992]. Para isso seria necessário armazenar um BRDF diferente para cada voxel, onde cada BRDF requer o armazenamento de uma tabela 4D, o que geraria um grande volume de dados.

O modelo apresentado por Neyret utiliza uma FDN (função de distribuição de normais) para representar as microgeometrias incidentes em cada voxel do volume, sendo que a FDN pode ser integrada a qualquer momento para calcular a intensidade de luz refletida em uma determinada direção (analogamente ao BRDF). A principal vantagem da FDN é que a mesma permite uma representação mais compacta que o BRDF. Neyret propõe que a FDN seja armazenada na forma de um elipsóide, o que torna sua representação ainda mais compacta, mas limita a distribuição de normais a ter apenas uma direção principal de reflectância.

Para acelerar a renderização, Neyret propõe que as microgeometrias sejam armazenadas em volumes com diferentes níveis de resoluções. Desta maneira, quando um

volume é renderizado é possível escolher qual a melhor resolução a ser utilizada, aumentando o desempenho da aplicação e também a qualidade das imagens geradas. A resolução do volume a ser utilizada pode ser escolhida, por exemplo, de acordo com a a distância do volume para a câmera. A Figura 2.6 ilustra a geometria de folhas sendo armazenadas em volumes com diferentes resoluções.

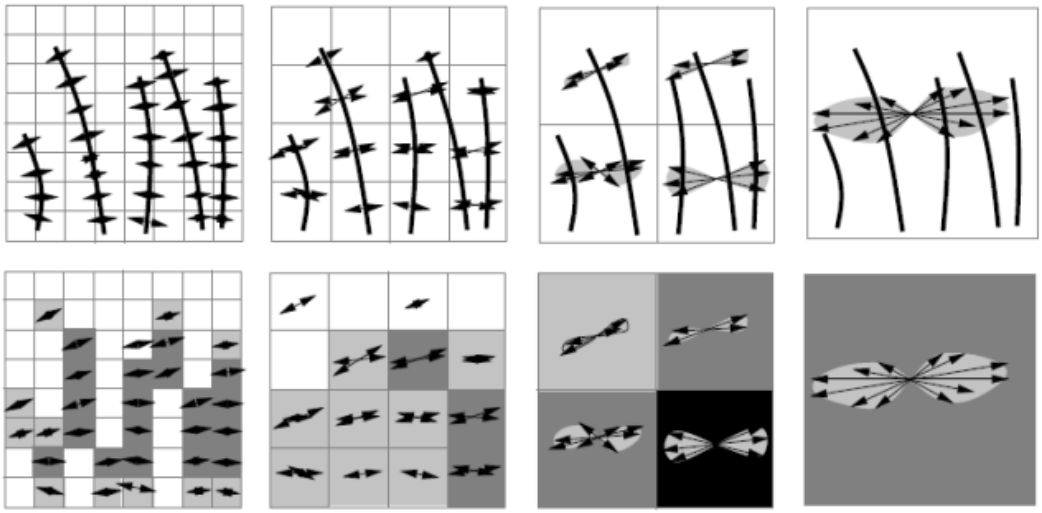


Figura 2.6. Geometria de folhas armazenada em diferentes níveis de resolução utilizando esquema similar ao mip-mapping [Neyret, 1998].

Finalmente, as texturas tridimensionais são mapeadas para superfícies regulares. O mapeamento proposto por Neyret utiliza coordenadas UVW e vetores de altura armazenados na superfície base, o que torna o mapeamento independente da subdivisão da superfície. Além disso, são aplicadas pequenas perturbações às coordenadas UVW e vetores de altura de modo a deixar o mapeamento menos repetitivo. Outras contribuições do trabalho de Neyret incluem ainda três abordagens diferentes para animação dos detalhes armazenados em texturas tridimensionais.

2.1.2 Shell Mapping

Em 2004, Hirche apresentou uma abordagem mais genérica para o mapeamento de detalhes sobre superfícies de malhas arbitrárias construídas como listas de triângulos [Hirche et al., 2004]. A abordagem proposta se baseia na aplicação de uma extrusão sobre uma malha utilizada como base para o mapeamento dos detalhes. A extrusão é utilizada para gerar um sólido sobre cada face da malha, o qual envolve o espaço sobre a malha no qual é possível realizar o mapeamento de detalhes. Devido as faces da malha base serem triângulares, os sólidos gerados são prismas, os quais são armazenados como uma lista de oito faces triângulares.

Os detalhes que serão mapeados para a malha base são armazenados em um volume tridimensional, o qual está contido no espaço de coordenadas da textura. Para que os detalhes possam ser mapeados sobre os prismas gerados é necessário mapear as coordenadas dos detalhes do espaço da textura para as coordenadas dos prismas no espaço do mundo. Os prismas gerados são enviados ao *pipeline* de renderização, e durante sua renderização um *shader* de *pixels* é utilizado para fazer o mapeamento dos detalhes do espaço da textura para o espaço do mundo. Desta maneira, todo o processamento necessário para o mapeamento e renderização dos detalhes é feito dentro do processador gráfico a partir da renderização dos prismas, apresentando um alto desempenho.

Quando os prismas gerados sobre a superfície base são retos, o mapeamento dos detalhes entre o espaço da textura e do mundo é feito a partir de um algoritmo de traçado de raios que caminha em linha reta dentro do volume que contém os detalhes. No entanto, a extrusão de uma malha base fechada (por exemplo, um sólido) não gera prismas retos e, dessa maneira o caminhamento do algoritmo de traçado de raios se torna complexo, sendo dependente da inclinação das faces do prisma. Além disso, não é possível caminhar com o raio no espaço do mundo, pois não existe uma solução para a transformação de coordenadas no espaço de mundo para o espaço da textura [Hirche et al., 2004]. Para resolver esse problema Hirche propõe a decomposição dos prismas em três tetraedros. O uso de tetraedros permite uma aproximação linear do mapeamento entre o espaço da textura e mundo, desta maneira, o mapeamento não linear do prisma passa a ser aproximado por três mapeamentos lineares (um para cada tetraedro). Note que o uso de tetraedros aumenta mais a complexidade dos sólidos renderizados, fazendo com que seja necessária a renderização de três tetraedros para cada face da malha original renderizada.

Em 2005, Porumbescu utilizou a técnica proposta por Hirche para o mapeamento de diversos tipos de detalhes sobre uma malha base [Porumbescu et al., 2005]. A técnica para mapeamento de detalhes sobre um volume criado sobre uma malha passa a ser referenciada como *Shell Mapping*, nome que referencia o volume criado sobre a malha base, chamado de *Shell Space*.

O algoritmo proposto por Porumbescu não tira proveito dos processadores gráficos, o que permite uma maior liberdade no mapeamento de detalhes sobre as malhas. Desta maneira, os detalhes mapeados para o volume sobre a superfície de uma malhas não precisam ser armazenados na forma de volumes, sendo possível, por exemplo, fazer o mapeamento direto de uma geometria sobre uma das faces da malha base. A Figura 2.7 apresenta o uso da técnica de *Shell Mapping* na renderização de um vidro, onde várias cópias da malha do vidro são mapeadas sobre o mesmo.

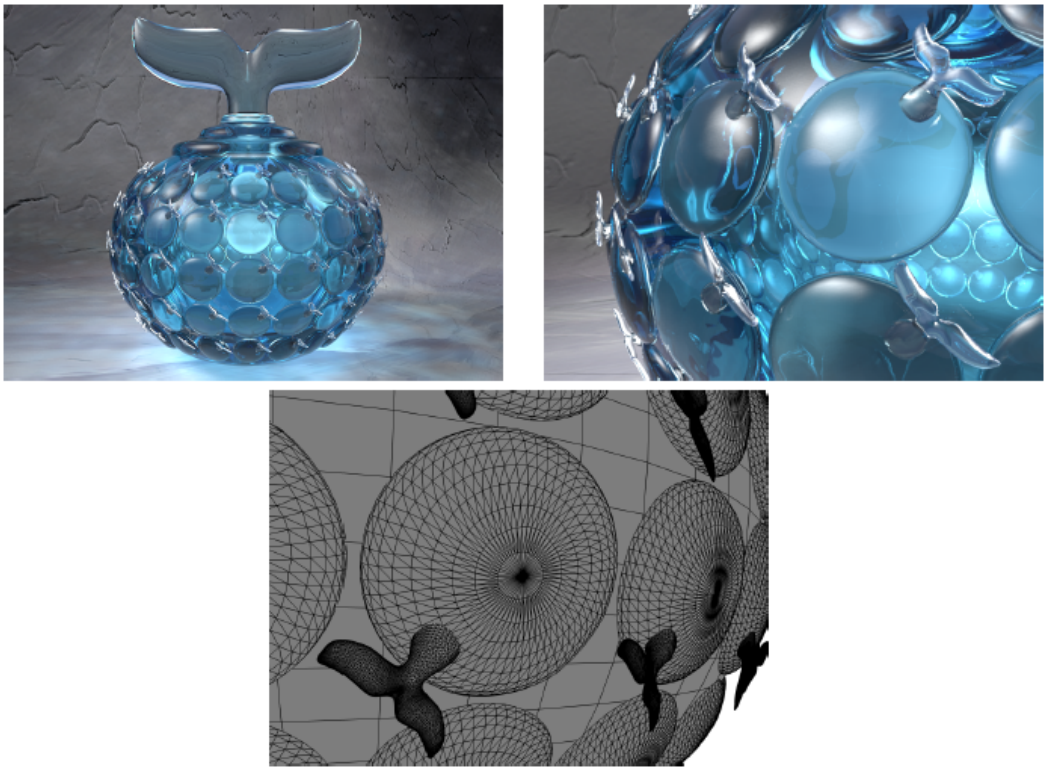


Figura 2.7. Resultados da renderização de um vidro, onde várias instâncias da malha do vidro são mapeadas sobre o mesmo utilizando *Shell Mapping* [Porumbescu et al., 2005].

Em 2006, Swaaij apresentou os problemas encontrados no uso de pelos e cabelos nas criaturas virtuais do filme "Era do gelo: o degelo"[Swaaij, 2006], e mostra como a renderização de volumes (ao invés da renderização direta de geometrias) foi utilizada para resolver os problemas encontrados.

A geometria de algumas criaturas virtuais presentes no filme, como do personagem Scrat, possui um número superior a um milhão de pelos modelados como curvas. O armazenamento e a renderização direta da geometria dos pelos não era viável no *pipeline* de renderização utilizado no filme, devido principalmente à necessidade de todos os dados de uma cena estarem na memória principal do computador no momento da renderização. Além disso, a renderização direta de um grande número de curvas, as quais apresentam pequena espessura e espaçamento, apresenta uma qualidade visual ruim.

A solução encontrada foi a geração de um volume contendo os dados dos pelos pré-processados. O armazenamento dos pelos através de um volume discreto permite que os mesmos sejam pré-filtrados, armazenando para cada posição discreta do volume, o resultado da possível interseção e interação de diferentes fios de pelos. Outra vantagem do uso de volumes é a redução no volume de dados dos pelos, sendo que o tamanho

do volume de dados ainda pode ser controlado aumentando ou reduzindo o número de camadas do volume.

Por último, em 2007, Jeschke apresentou uma técnica para a renderização de *Shell Maps* através do uso de prismas e tirando proveito dos processadores gráficos [Jeschke et al., 2007]. Jeschke demonstra que as abordagens anteriores para a renderização de *Shell Maps* através da decomposição de prismas em tetraedros apresentam problemas severos no mapeamento, além de aumentar a complexidade da geometria utilizada. A decomposição de prismas em tetradros é uma aproximação linear do mapeamento entre o espaço de textura e o espaço de mundo, no entanto na renderização de geometrias de baixa complexidade essa aproximação não é suficiente para produzir bons resultados visuais, como ilustrado na Figura 2.8.

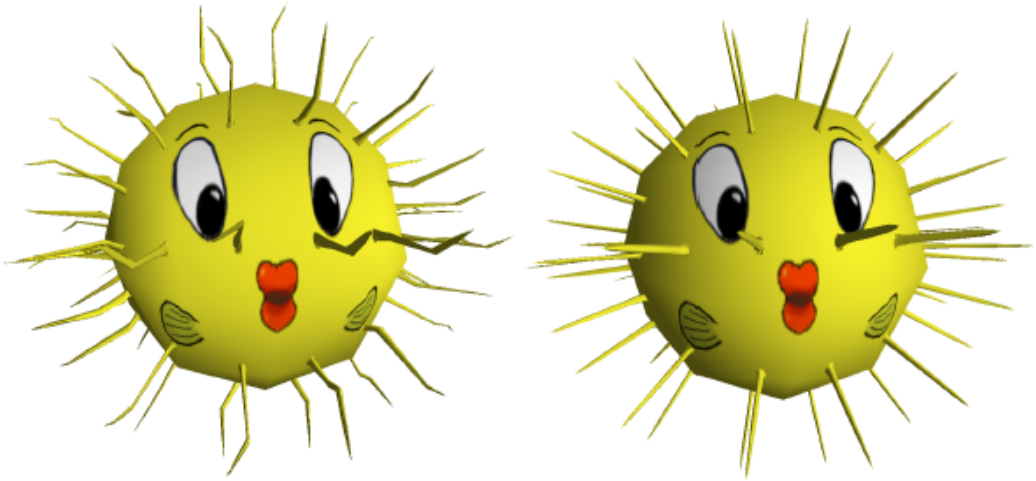


Figura 2.8. Comparação entre a renderização de detalhes utilizando a técnica de *Shell Mapping*. (Esquerda) aproximação do mapeamento entre espaço da textura e espaço do mundo utilizando a subdivisão de prismas em tetraedros. (Direita) mapeamento utilizando o algoritmo proposto por Jeschke [Jeschke et al., 2007].

Para resolver o problema de mapeamento, Jeschke propõe que o mapeamento entre o espaço da textura e do mundo seja aproximado linearmente dentro de cada prisma, e que os erros no mapeamento sejam corrigidos em tempo real. Os erros no mapeamento são calculados durante a execução do algoritmo de traçado de raios dentro do volume da textura de detalhes, o qual é executado através de um *shader* de *pixels* para cada ponto da superfície do prisma rasterizado. Para cada valor mapeado entre o espaço da textura e o espaço do mundo é calculado um erro de mapeamento (devido ao mesmo ser aproximado linearmente), e quando o erro é superior a um valor piso definido o mapeamente é corrigido.

A correção do mapeamento é feita executando o algoritmo de traçado de raios no espaço de mundo, verificando qual seria a posição correta para a posição mapeada

entre o espaço de textura e o espaço de mundo, e gerando um vetor de correção para o mapeamento. Desta maneira, o algoritmo de traçado de raios é executado no espaço da textura utilizando uma aproximação linear, mas a cada passo da execução do algoritmo é possível aplicar uma correção no mapeamento fazendo com que o resultado possa ser igual ao mapeamento não linear original. A Figura 2.9 apresenta a correção no mapeamento proposta por Jeschke.

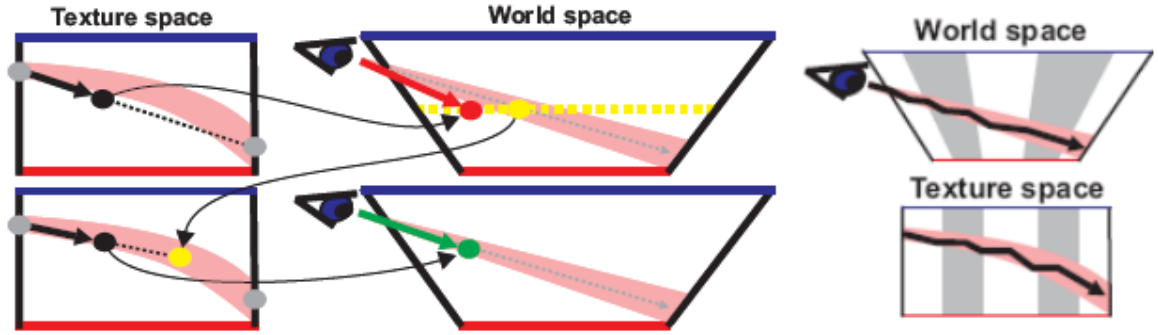


Figura 2.9. Correção no mapeamento entre espaço da textura e espaço do mundo [Jeschke et al., 2007].

A técnica proposta por Jeschke apresenta uma alta qualidade visual, mas demanda um alto custo computacional devido à necessidade da correção do mapeamento entre o espaço de textura e o espaço de mundo.

2.2 Renderização de Camadas de Volumes

Em 1995, Lacroute [Lacroute, 1995] propos um algoritmo para renderização interativa de volumes. O algoritmo proposto consiste em discretizar cada volume presente em uma cena em um arranjo de planos paralelos, onde os dados de cada plano são armazenados na forma de uma textura bidimensional. Este processo é chamado pelo autor de *shear-warp factorization*. A Figura 2.10 ilustra a discretização de um volume contendo uma superfície em um arranjo de planos.

Utilizando esse algoritmo a renderização de um volume pode ser feita através da renderização de um arranjo de texturas bidimensionais aplicadas a planos paralelos à superfície de uma malha base. A renderização de cada plano é feita em *software* a partir de rasterização de primitivas utilizando um algoritmo de *scan line*. A Figura 2.11 ilustra o processo de renderização utilizando texturas bidimensionais. A principal motivação do trabalho de Lacroute se deve ao fato de mesmo os algoritmos mais eficientes para renderização de volumes, como os que utilizam estruturas espaciais para acelerar a renderização, não conseguirem renderizar volumes em tempo real.

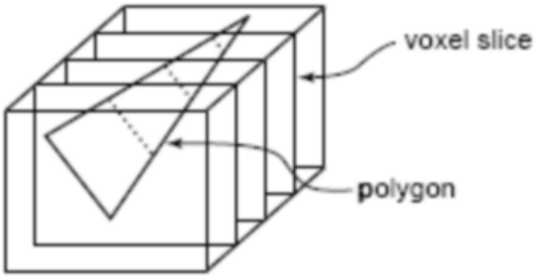


Figura 2.10. Discretização de um volume contendo uma superfície em um arranjo de planos paralelos [Lacroute, 1995].

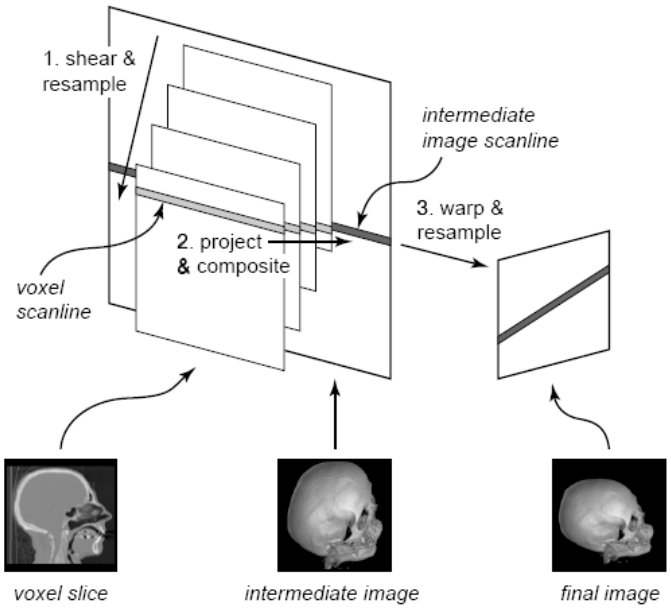


Figura 2.11. Discretização de um volume em texturas bidimensionais e renderização [Lacroute, 1995].

Em 1998, Meyer e Neyret [Meyer e Neyret, 1998] apresentaram uma extensão do algoritmo proposto por Lacroute [Lacroute, 1995] para renderização de cenas complexas e repetitivas, como paisagens e pelos, tirando proveito dos processadores gráficos. A renderização de volumes proposta por Lacroute é feita a partir da rasterização de superfícies texturizadas e com transparência, operações que podem ser realizadas completamente em hardware nos processadores gráficos que utilizam *Z-Buffer* [Akenine-Möller et al., 2008]. Desta maneira, Meyer e Neyret propõem a renderização de volumes a partir da renderização de superfícies utilizando o processador gráfico.

Alguns dos volumes utilizados neste trabalho são os mesmos utilizados em trabalhos prévios dos autores [Neyret, 1998], onde a renderização é feita utilizando *ray-tracing*. O principal objetivo dos autores é renderizar cenas complexas, gerando imagens de

alta qualidade visual (similares às obtidas com a renderização de volumes utilizando *ray-tracing*) em tempo real. Os volumes utilizados na renderização são representados utilizando 64 camadas, que são armazenadas na forma de texturas. Desta meneira, o volume de dados armazenado para cada objeto é pequeno comparado ao volume original, o que permite renderizar um maior número de objetos numa mesma cena.

Entretanto, existem algumas limitações na renderização das camadas do volume, devido à mesma ser realizada através de rasterização utilizando o algoritmo de *Z-Buffer*. Nos processadores gráficos não programáveis, não é possível calcular a iluminação ou sombreamento de cada ponto, em cada camada do volume. Sendo possível apenas pré-calcular esses valores e aplicar os mesmos utilizando outras texturas. Devido a iluminação das camadas ser calculada localmente para cada camada, tambem não existem sombras nas imagens geradas.

Outra limitação se deve à necessidade de desenhar camadas do volume contendo partes semi-transparentes. Para que superfícies semi-transparentes sejam desenhadas corretamente utilizando o algoritmo de *Z-Buffer*, é necessário que as mesmas sejam ordenadas de acordo com sua profundidade. Desta maneira, camadas mais distantes do observador devem ser desenhadas antes de camadas mais próximas. Devido ao alto custo de ordenar as superfícies antes da renderização, os autores permitem que as superfícies tenham apenas partes opacas, ou completamente transparentes. A Figura 2.12 ilustra o problema no uso de objetos semi-transparentes com o algoritmo de *Z-Buffer*.

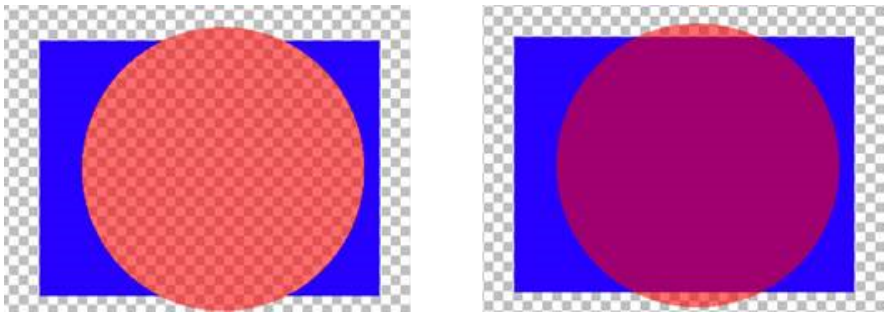


Figura 2.12. Renderização de um círculo semi-transparente sobre um retângulo opaco utilizando o algoritmo de *Z-Buffer*. Esquerda, desenho fora de ordem gerando resultado incorreto. Direita, desenho em ordem gerando resultado correto.

Finalmente, quando o vetor de visão é paralelo às camadas do volume a noção de continuidade do volume é perdida, sendo possível visualizar cada camada do mesmo em separado. Para tratar esse problema os autores propõem gerar as camadas dos volumes em três direções diferentes, sendo que a direção utilizada na renderização é escolhida para cada volume de acordo com o vetor de visão. A Figura 2.13 ilustra a

criação das camadas do volume nas três direções utilizadas, e a Figura 2.14 ilustra os resultados obtidos na renderização de folhas.

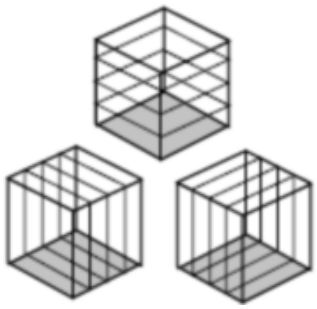


Figura 2.13. Geração das camadas de um volume em três direções diferentes. As camadas a serem utilizadas são escolhidas de acordo com o vetor de visão.

Figura 2.14. Geração das camadas de um volume em três direções diferentes. As camadas a serem utilizadas são escolhidas de acordo com o vetor de visão.

Em 2002, Isidoro e Mitchell [Isidoro e Mitchell, 2002] apresentaram um algoritmo para renderização de pelos tirando proveito dos processadores gráficos programáveis. O algoritmo proposto aplica uma extrusão em tempo real sobre uma malha base, gerando novas camadas paralelas e perpendiculares sobre a malha onde os pelos serão aplicados. Essas novas camadas são geradas em tempo real a partir de um *shader* de vértices. A criação de camadas perpendiculares à malha base, de acordo com o vetor de visão pelo qual a malha é observada, ajuda a aumentar o realismo na renderização dos pelos cobrindo pequenos buracos que poderiam ser observados entre camadas paralelas da malha. Devido às camadas paralelas e perpendiculares da malha serem geradas em tempo real, é possível modificar o número de camadas geradas, ou a distância entre as mesmas em tempo real. Neste trabalho, as camadas paralelas e perpendiculares da malha geradas, são geradas a partir da extrusão da malha base utilizando o vetor

normal armazenado nos vértices. Desta maneira, os autores propõem a deformação dos vetores normais nos vértices, para simular a movimentação dos pelos. A Figura 2.15 ilustra a criação de camadas paralelas e perpendiculares para a aplicação e renderização de pelos em um torus. As camadas paralelas são chamadas de *Shells* e as camadas perpendiculares de *Fins*.

Outra extensão importante apresentada por Isodoro e Mitchel é o uso de texturas bidimensionais para a representação da direção de crescimento dos fios do pelo em cada uma das camadas de pelos. Esta abordagem reduz a homogeneidade dos pelos e permite representar tipos mais complexos de pelos, como pelos ondulados.

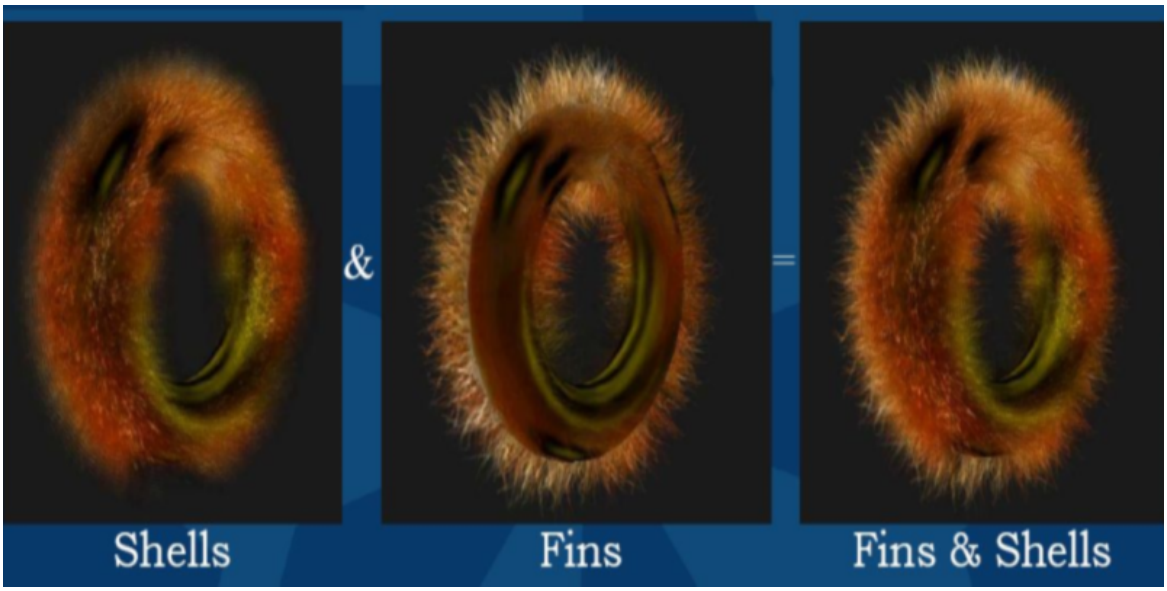


Figura 2.15. Geração de camadas paralelas e perpendiculares para a aplicação e renderização de pelos. (Esquerda) camadas paralelas, (Centro) camadas perpendiculares e (Direita) combinação de camadas paralelas e perpendiculares [Isidoro e Mitchell, 2002].

Em 2004, Sheppard apresentou um resumo de vários trabalhos relacionados a modelagem e renderização de pelos [Sheppard, 2004], e propoem um algoritmo para renderização de pelos em tempo real a partir da geração de camadas paralelas e perpendiculares de uma malha base (chamadas respectivamente de *shells* e *fins*). O resumo da literatura apresentado cobre os principais tipos de modelagem dos pelos (utilizando geometrias, volumes e camadas paralelas), os principais modelos utilizados para geração de pelos (utilizando sistema de partículas, ou *noise*) e os principais algoritmos utilizados para iluminação e sombreamento dos pelos.

O algoritmo para renderização de pelos proposto pelo autor combina a abordagem proposta por Isidoro [Isidoro e Mitchell, 2002] para renderização de pelos utilizando *shells* e *fins*, com a abordagem proposta por Lengyel [Lengyel e Praun, 2001] para

aplicação de pelos em superfícies arbitrárias, e o modelo de iluminação de pelos utilizado é o proposto por Kajiya e Kay [Kajiya e Kay, 1989]. Nesse trabalho, toda a renderização dos pelos é feita utilizado *shaders* de vértice e *pixel*.

Em especial, o autor apresenta uma ferramenta para a criação de pelos baseado em parâmetros de espessura, densidade e curvatura. Os pelos criados pela ferramenta são comparados com imagens extraídas de pelos reais de animais e humanos, sendo possível observar quais tipos de pelos podem ser representados com maior fidelidade pela abordagem proposta. A Figura 2.16 apresenta a comparação de dois pelos reais com pelos gerados pela ferramenta apresentada por Sheppard.

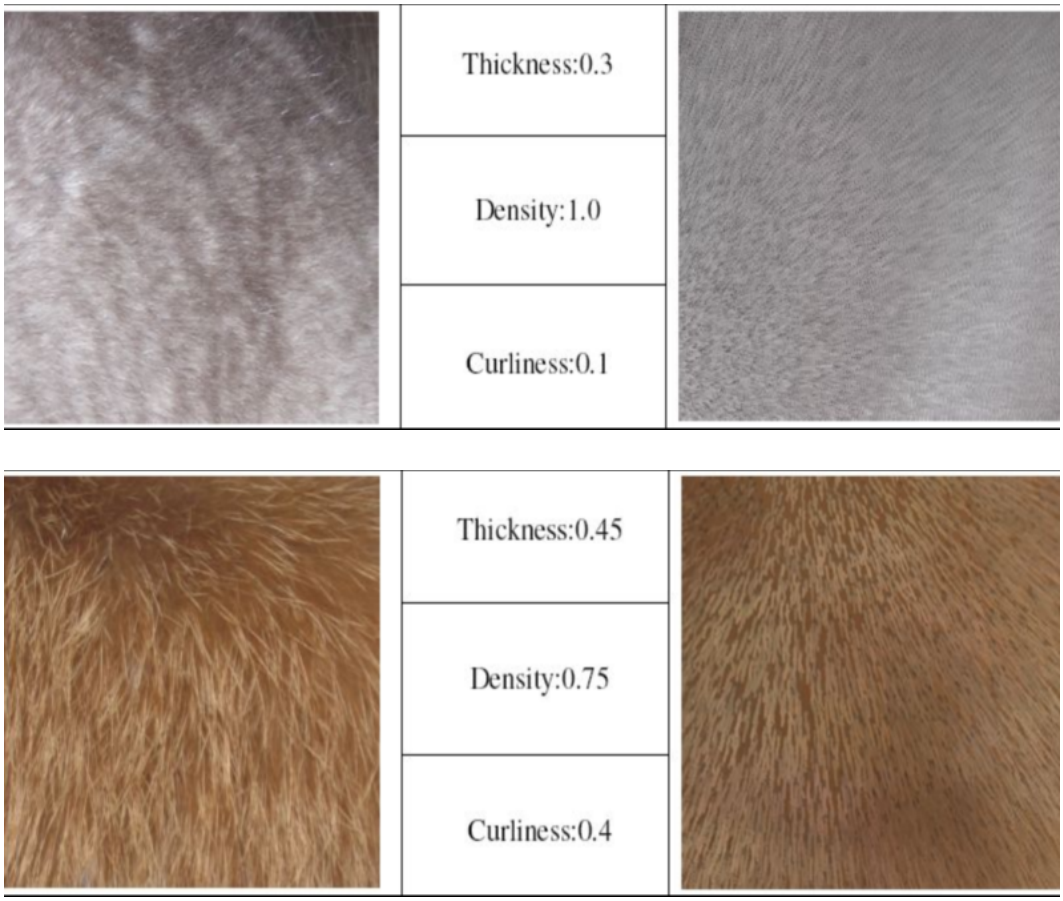


Figura 2.16. Comparação entre imagens de pelos extraídas do mundo real (esquerda) e imagens geradas pela ferramenta FurMak (direita) proposta por Sheppard [Sheppard, 2004]. Imagem superior apresenta o pelo de um chinchila, e imagem inferior apresenta o pelo de um coelho.

Em 2007, Wyatt apresentou o algoritmo para renderização de pelos utilizado no jogo *Ratchet & Clank* para o console Playstation 3, produzido pela empresa Insomniac [Wyatt, 2007]. O algoritmo utilizado por Wyatt se baseia apenas na renderização de camadas paralelas da malha para renderização dos pelos, não realizando a renderização de *fins* (camadas perpendiculares). Desta maneira, a renderização dos pelos é feita

utilizando 16 ou 20 camadas paralelas da malha. O autor cita que um dos maiores custos dessa abordagem é o grande volume de dados transferidos devido a mistura de cores nos *pixels* dos pelos renderizados, o que ocorre devido a todos os pelos terem algum nível de transparência. Este trabalho é de grande importância pois demonstra que a renderização de pelos a partir de camadas paralelas pode ser facilmente integrado a um ambiente interativo com diversas restrições, como é o caso de um console. A Figura 2.17 ilustra a renderização da face do personagem Ratchet do jogo Ratchet and Clank.

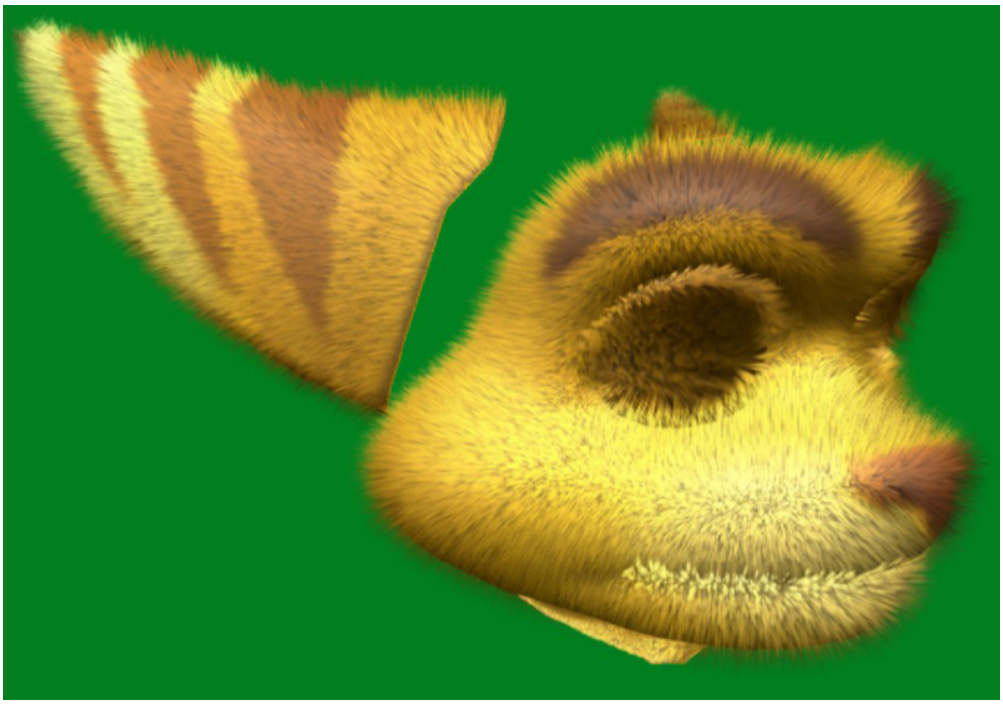


Figura 2.17. Renderização de pelos sobre a face do personagem Ratchet, do jogo Ratchet and Clank para o Playstation 3 [Wyatt, 2007].

Em 2008, Yang observou que diferentes tipos de pelos precisam de um diferente número de camadas paralelas para serem renderizados com qualidade [Yang et al., 2008]. Em especial, é observado que fatores como distância de observação da malha, ângulo pelo qual a malha é observada, altura dos pelos, e grau de curvatura dos pelos interferem no número de camadas necessárias para a renderização dos pelos. Desta maneira, o autor propõe a geração de camadas paralelas não uniformes para a renderização dos pelos, onde o número de camadas geradas para cada trecho da malha varia de acordo com vários fatores. A Figura 2.18 ilustra um número variável de camadas para diferentes trechos da malha de acordo com a posição da câmera pelo qual a mesma é observada.

No algoritmo proposto por Yang o número de camadas varia entre 16, 31 e 61.

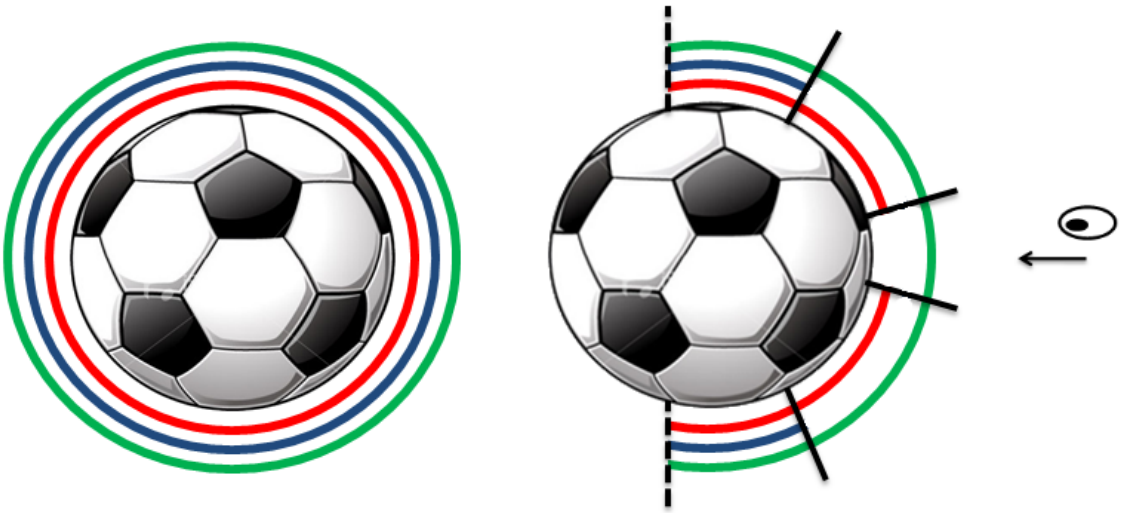


Figura 2.18. Renderização utilizando camadas uniformes e não uniformes. (Esquerda) várias camadas uniformes criadas ao redor de uma bola, (Direita) camadas não uniformes criadas ao redor de uma bola onde partes que precisão de maior detalhe recebem um maior número de camadas.

Desta maneira é possível fazer uma transição suave entre o número variável de camadas adicionando ou removendo uma camada entre cada par de camadas existentes. A troca entre os números de camadas existentes é feita a partir da classificação de trechos da malha, a qual pode consumir um tempo considerável mas garante a redução no número de camadas. O autor demonstra que utilizando a renderização de camadas não uniformes é possível obter resultados visuais comparáveis a renderização de camadas uniformes mas obtendo um desempenho até duas vezes superior. No entanto, não é apresentado no trabalho o tempo necessário para classificar as diferentes partes da malha, atribuindo um número de camadas diferente para cada trecho. Dependendo da complexidade utilizada para classificação da malha esse custo pode tornar o uso do algoritmo inviável em aplicações de tempo real, onde a câmera pode ser mover livremente no espaço. No entanto, se a posição da câmera for fixa no espaço, ou se a mesma não for considerada na classificação de trechos da malha, esse algoritmo pode proporcionar um grande aumento de desempenho.

O trabalho de Yang também apresenta um esquema para deformação e animação dos pelos modificando como as camadas da malha são geradas. Também é apresentado um esquema para geração de sombras, incluindo sobras que os pelos geram sobre eles mesmos utilizando *shadow maps*.

2.3 Triângulos Indexados e *Strips* de Triângulos

A malha de um objeto geralmente é armazenada na forma de uma lista de faces triângulares, onde três vértices são utilizados para armazenar os dados de cada face. Em uma malha fechada (sem a existência de buracos), armazenada como uma lista de faces triângulares, cada vértice é compartilhado na média por 6 faces adjacentes.

Uma maneira comum de reduzir os dados de uma malha e aumentar o desempenho na renderização da mesma, é transformá-la em um conjunto de *strips* de triângulos [Akenine-Möller et al., 2008]. Em uma *strip* de triângulos, os três primeiros vértices são utilizados para definir uma face inicial, em seguida, cada vértice do *strip* define uma nova face ligando o vértice atual com os dois últimos vértices do *strip*. Desta maneira, o número de vértices utilizados para armazenar uma face da malha em um *strip* de triângulos tende a um. Esta otimização também aumenta o desempenho da renderização das malhas, pois faz com que seja necessário o processamento de apenas um novo vértice para cada novo triângulo. *Strips* de triângulos são baratos de serem implementados em processadores gráficos, necessitando de apenas três registradores [Hoppe, 1999].

A transformação ótima de uma lista de triângulos em *strips* de triângulos é um problema NP-completo [Arkin et al., 1994], sendo necessário o uso de heurísticas para resolver o problema. Além disso, muitas vezes é impossível gerar uma única *strip* de triângulos para um objeto, como é o caso de um cubo, que requer a criação de pelo menos duas *strips*. No pior caso, a transformação de uma malha (armazenada como uma lista de triângulos) em *strips* de triângulos, pode gerar uma *strip* para cada triângulo da lista. Note que é possível juntar várias strips de triângulos utilizando triângulos degenerados. O uso de *strips* de triângulos tende a ser muito vantajoso em processadores gráficos antigos, ou de baixo custo, os quais não possuem cache de vértices. Em processadores gráficos modernos, com cache de vértices, o uso de triângulos indexados tende a ser mais vantajoso [Bogomjakov e Gotsman, 2001].

Como citado anteriormente, em uma malha fechada cada vértice é compartilhado na média por 6 faces adjacentes. Se os vértices da malha forem indexados, no melhor caso, é necessário o armazenamento de metade dos dados de um vértice para cada face da malha. Essa representação é duas vezes mais compacta que a representação utilizando *strips* de triângulos, e da mesma maneira, tende a ter um desempenho duas vezes superior. Note que também é possível indexar os vértices em um strip de triângulos, no entanto, como cada novo vértice adicionado ao strip deve ser ligado ao dois últimos vértices do strip muitas vezes não é possível fazer uma boa escolha de novo vértice que maximize o uso de cache [Sander et al., 2007, Hoppe, 1999]. Neste trabalho, explora-se o uso de triângulos indexados para aumentar o desempenho na renderização de malhas

arbitrárias contendo pelos.

2.4 Algoritmo Proposto e Contribuições

O algoritmo de renderização de pelos apresentado neste trabalho se baseia na renderização de múltiplas camadas de uma malha base. Desta maneira, a renderização dos pelos é feita através da rasterização e sombreamento de múltiplas malhas, operações que são suportadas por um grande número de processadores gráficos, o que permite o uso dessa técnica em várias plataformas. O algoritmo de renderização dos pêlos é apresentado com detalhes na Seção 6.

Este trabalho também apresenta um algoritmo de pré-processamento de malhas, o qual se baseia no uso de triângulos indexados para realizar a remoção de dados duplicados das malhas, e a reordenação dos índices para otimizar a localidade espacial dos dados. O algoritmo de pré-processamento das malhas proposto é apresentado com detalhes na Seção 4.

A principal contribuição deste trabalho está relacionada ao mapeamento automático dos pêlos criados sobre malhas arbitrárias em tempo real. Este mapeamento é realizado em tempo real utilizando uma extensão do algoritmo tradicional de mapeamento de ambiente [Akenine-Möller et al., 2008]. Devido ao mapeamento ser calculado em tempo real, não é necessário a pré-geração e armazenamento de coordenadas de mapeamento de textura na malha. Outra vantagem desta abordagem, é que a mesma permite que o número de camadas paralelas da malha que são renderizadas seja modificado em tempo real, o que é possivel devido ao mapeamento dos pelos sobre a nova camada criada ser calculado em tempo real.

2.5 Sumário

Este capítulo apresentou um resumo dos trabalhos relacionados a renderização de pelos, e um breve resumo dos trabalhos relacionados a otimização da representação dos dados de uma malha. Os trabalhos relacionados a renderização dos pêlos são divididos em dois grupos, o primeiro baseado na renderização de volumes construídos sobre uma malha, e o segundo baseado na renderização de múltiplas camadas de uma malha. O próximo capítulo apresenta a metodologia utilizada para a obtenção e reconstrução das malhas obtidas de objetos reais, assim como a metodologia utilizada para geração de níveis de detalhe para essas malhas.

Capítulo 3

Malhas Arbitrárias Obtidas de Objetos Reais

As malhas utilizadas neste trabalho foram obtidas a partir de objetos do mundo real através do uso de *scanners* tridimensionais. Este trabalho não foca o processo de *scanning* e coleta de dados de objetos reais, sendo que todas as malhas utilizadas neste trabalho foram obtidas a partir do repositório de *scanning* 3D da universidade de Stanford [Stanford, 2009]. O repositório da universidade de Stanford é uma excelente opção para pesquisadores que não possuem acesso a equipamentos de *scanning* 3D. O repositório disponibiliza uma grande variedade de modelos contendo malhas extremamente detalhadas, onde algumas malhas chegam a conter milhões de triângulos.

As malhas tridimensionais disponibilizadas no repositório de Stanford são utilizadas em inúmeros trabalhos da área de computação gráfica, desta maneira, o uso dessas malhas permitirá a comparação dos resultados visuais obtidos neste trabalho com demais trabalhos da área. Neste trabalho foram utilizadas malhas de quatro objetos diferentes do repositório de Stanford: *Armadillo*, *Buddha*, *Bunny* e *Dragon*. A Figura 3.1 ilustra as malhas dos objetos utilizados neste trabalho.

3.1 Reconstrução das Malhas

O processo de *scanning* de um objeto real, geralmente precisa ser realizado a partir de diferentes poses do objeto. Para cada pose do objeto, o *scanner* tridimensional coleta um conjunto de pontos pertencentes a superfície do objeto naquela pose. Após um número suficiente de poses do objeto ter sido coletada, é possível reconstruir a superfície do mesmo. Para isso, as poses precisam ser alinhadas de acordo com a orientação da câmera utilizada na coleta dos dados de cada pose. Em seguida, o conjunto de pontos pertencentes a superfície do objeto precisa ser processado para gerar

(A) (B)

(C) (D)

Figura 3.1. Malhas obtidas de objetos reais a partir do uso de um *scanner* tridimensional, e disponibilizados pela universidade de Stanford. (A) Bunny, (B) Dragon, (C) Buddha e (D) Armadillo.

uma malha (arranjo de triângulos), processo denominado reconstrução de superfície. A Figura 3.2 apresenta o diagrama do processo de *scanning* e reconstrução da malha de um modelo real, assim como a geração de níveis de detalhe para o mesmo (tópico abordado na Seção 3.2).

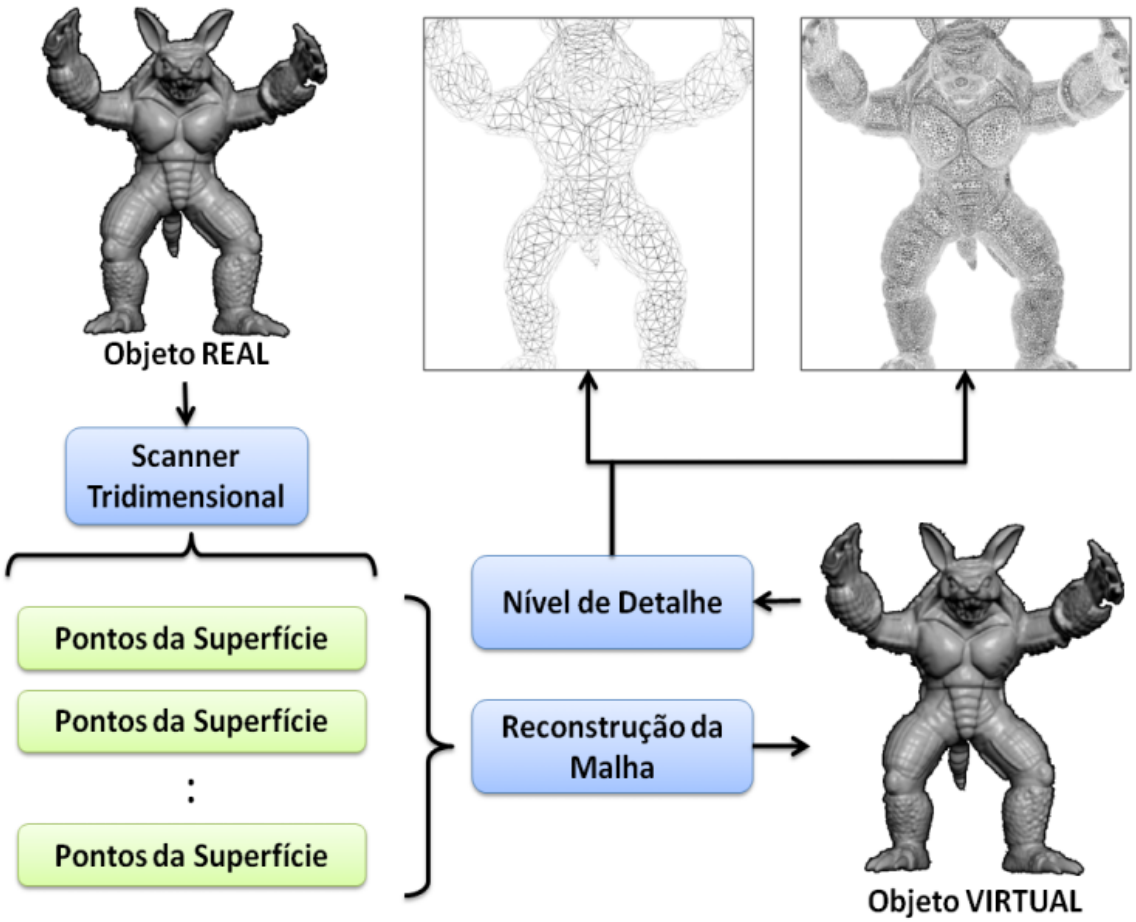
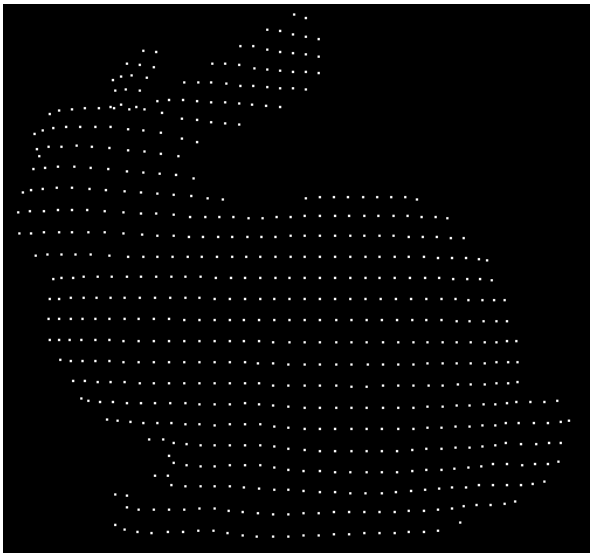


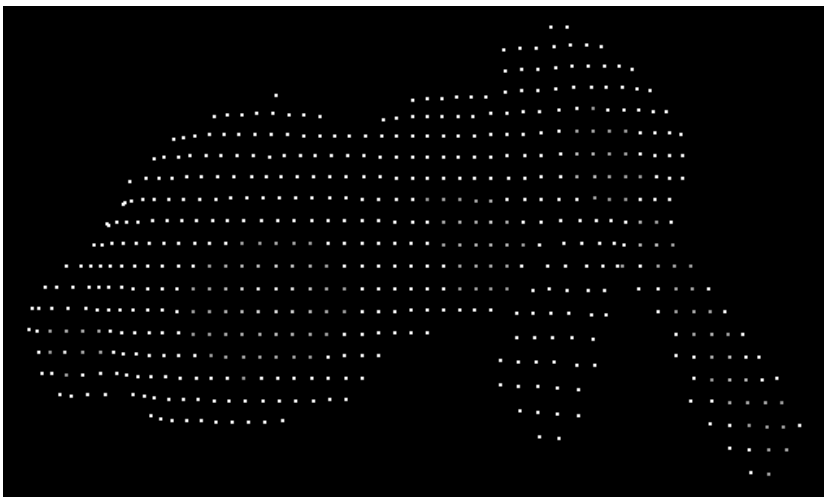
Figura 3.2. Diagrama do processo de *scanning*, reconstrução da malha e geração de níveis de detalhe para um objeto real.

Neste trabalho, os algoritmos de *Zippering* [Turk e Levoy, 1994] e *Volumetric Merging* [Curless e Levoy, 1996] foram utilizados para reconstrução da malha dos objetos. Implementações de código aberto dos algoritmos de *Zippering* e *Volumetric Merging* são disponibilizados pelos autores nas ferramentas ZipPack [Turk e Levoy, 2007] e VripPack [Curless e Levoy, 2007]. A Figura 3.3 ilustra a coleta de pontos na superfície do objeto *Bunny* (ilustrado na Figura 3.1) em duas diferentes poses. Esta imagem foi gerada utilizando o *software* de código aberto Scanalyze [Pulli e Ginzton, 2007].

O algoritmo de *Volumetric Merging* possui algumas vantagens sobre o algoritmo de *Zippering*, como a capacidade de preencher buracos nos objetos usando informações das poses coletadas no processo de *scanning*. Além disso, em malhas contendo detalhes



(A)



(B)

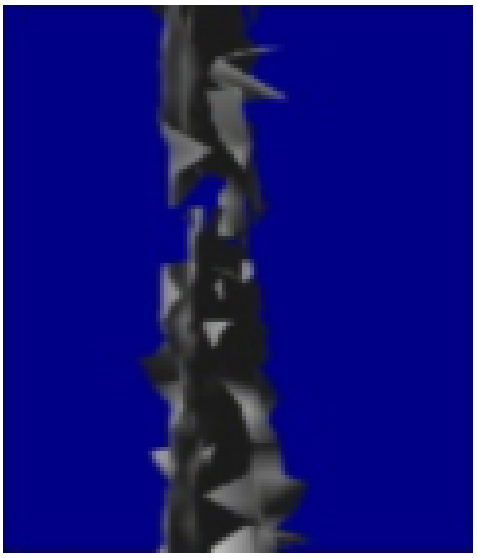
Figura 3.3. Pontos coletados da superfície do objeto *Bunny* (ilustrado na Figura 3.1) a partir de um *scanner* tridimensional. (A) pontos extraídos da parte frontal do objeto, (B) pontos extraídos da parte superior do objeto. Imagens geradas utilizando o *software* Scanalyze [Pulli e Ginzton, 2007]

de alta frequência o algoritmo de *Volumetric Merging* pode apresentar um resultado significativamente superior ao algoritmo de *Zippering*, como ilustrado na Figura 3.4.

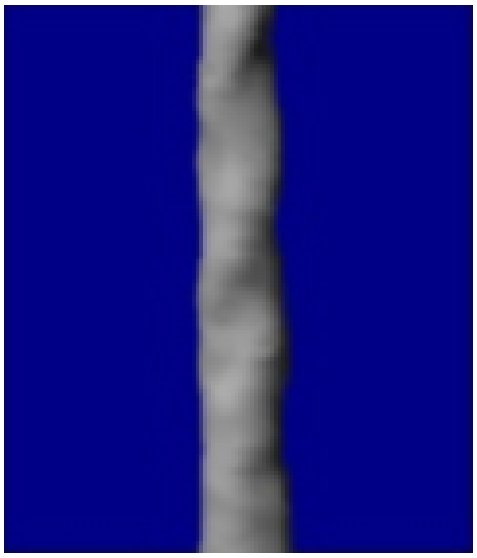
3.2 Nível de Detalhe

Malhas obtidas de objetos reais possuem uma superfície extremamente detalhada, podendo chegar a complexidade de milhões de triângulos. Neste trabalho, optou-se por utilizar as malhas originais obtidas dos objetos, assim como versões reduzidas dessas

(A)



(B)



(C)

Figura 3.4. Comparação entre os algoritmos *Zippering* [Turk e Levoy, 1994] e *Volumetric Merging* [Curless e Levoy, 1996]. (A) imagem do objeto real a ser reconstruído, (B) reconstrução utilizando algoritmo *Zippering* e (C) reconstrução utilizando *Volumetric Merging*.

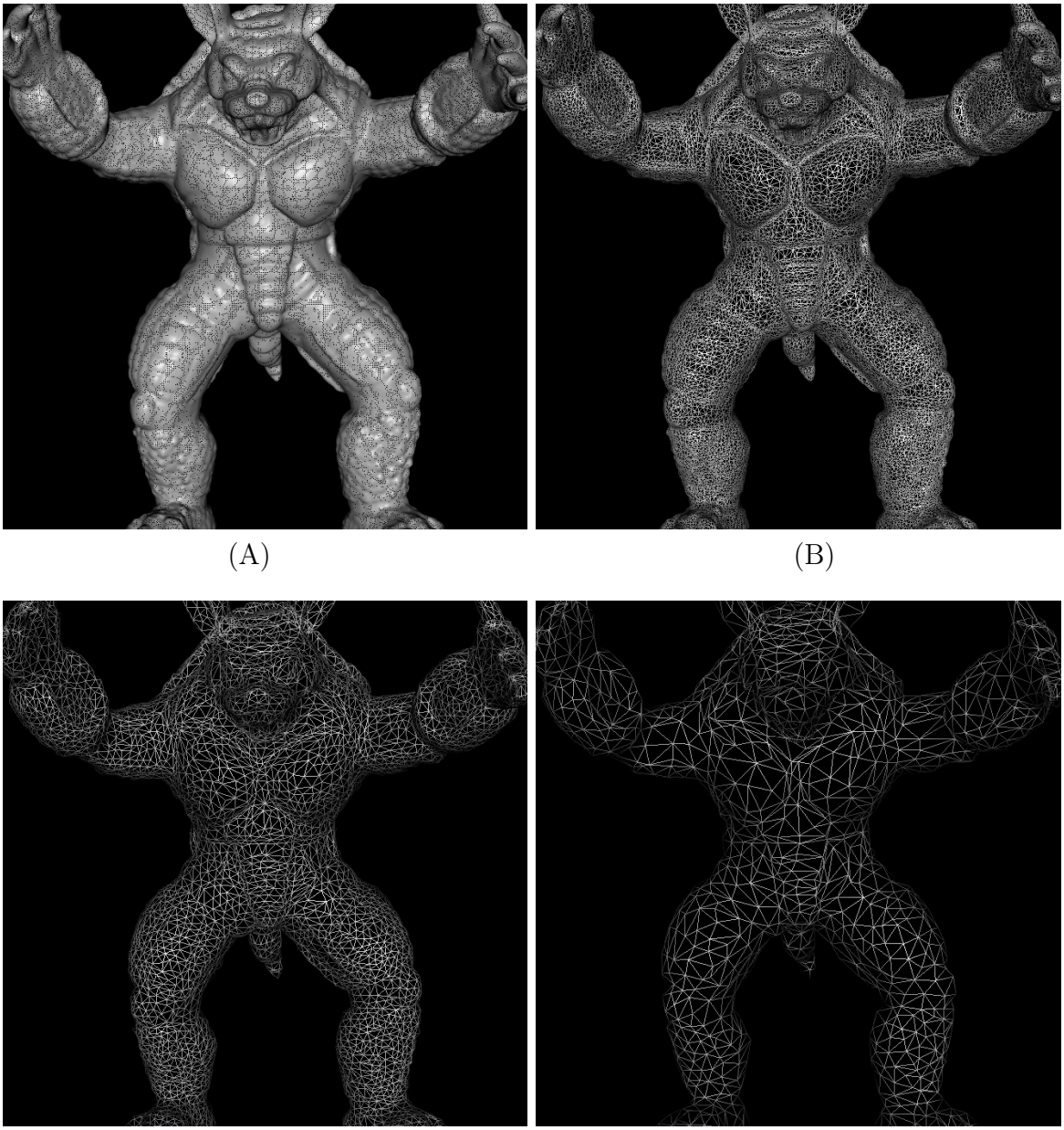
malhas, geradas a partir do uso de algoritmos de simplificação de malhas. As versões simplificadas das malhas utilizadas nesse trabalho foram geradas utilizando o *software* de código aberto *MeshLab* [Lab, 2009]. O *MeshLab* permite a simplificação de malhas através das técnicas de clusterização e colisão de arestas adjacentes. A Figura 3.2 ilustra o processo de geração de níveis de detalhe para a malha obtida de um objeto.

Neste trabalho, optou-se pela redução das malhas utilizando o algoritmo de colisão de arestas (*edge collapse*) [Luebke et al., 2002]. O algoritmo de colisão de arestas remove faces da malha de um objeto a partir da colisão e agrupamento de arestas de faces adjacentes, em cada passo do algoritmo as arestas cuja remoção implica no menor erro sobre a malha (calculado como o somatório das distorções causadas pela colisão

das arestas) são escolhidas para a colisão. A colisão de arestas foi escolhida porque permite definir com precisão o número de triângulos da malha simplificada a ser gerada. O algoritmo de colisão de arestas adjacentes utilizado pelo *MeshLab* é uma versão melhorada do algoritmo proposto por Garland [Garland e Heckbert, 1997]. A Figura 3.5 apresenta as malhas simplificadas geradas para o objeto |*Armadillo* (ilustrado na Figura 3.1).

3.3 Sumário

Este capítulo apresentou a metodologia utilizada para a obtenção e reconstrução das malhas obtidas de objetos reais, assim como a metodologia utilizada para geração de níveis de detalhe para essas malhas. O próximo capítulo apresenta os algoritmos utilizados para pré-processamento das malhas arbitrárias obtidas que têm como objetivo tornar a malha arbitrária de entrada apta à renderização de pelos, e otimizar os dados da mesma para proporcionar uma renderização eficiente da malha.



(A) (B)

(C) (D)

Figura 3.5. Malhas simplificadas do objeto Armadillo (ilustrado na Figura 3.1) geradas utilizando *MeshLab*. (A) reconstrução original do objeto contendo 345.944 triângulos, (B) primeira simplificação contendo 83.026 triângulos, (C) segunda simplificação contendo 19.926 triângulos e (D) terceira simplificação contendo 4.782 triângulos.

Capítulo 4

Pré-Processamento da Malha para Aplicação de pelos

Antes que a malha de um objeto, ou a malha de algum dos níveis de detalhe de um objeto, possa ser utilizada para aplicação e renderização de pelos é necessário que a mesma passe por um pré-processamento. O objetivo do estágio de pré-processamento é modificar a malha arbitrária utilizada como entrada, fazendo com que essa malha atenda aos requisitos impostos pelo algoritmo de aplicação e renderização de pelos. Por exemplo, o algoritmo de renderização dos pelos exige que cada vértice da malha renderizada possua um vetor normal. Desta maneira, o estágio de pré-processamento deve garantir que os vértices da malha de entrada possuam um vetor normal, e quando isso não ocorrer, o vetor normal de cada vértice deve ser gerado. Além de garantir que a malha de entrada atenda aos requisitos necessários para a renderização dos pelos, o estágio de pré-processamento também é utilizado para aplicar otimizações sobre a malha de entrada. As otimizações aplicadas sobre a malha são utilizadas para reduzir os dados da malha e aumentar o desempenho na renderização da mesma. A Figura 4.1 apresenta o diagrama do estágio de pré-processamento, aplicado sobre as malhas de entrada.

No estágio de pré-processamento proposto neste trabalho são aplicados quatro diferentes algoritmos sobre a malha de entrada, sendo eles: centralização da posição dos vértices, geração de normais contínuas, detecção de dados duplicados e indexação e reordenação de índices para *cache* de GPU. Esses algoritmos são apresentados respectivamente nas Seções 4.1, 4.2, 4.3 e 4.4.

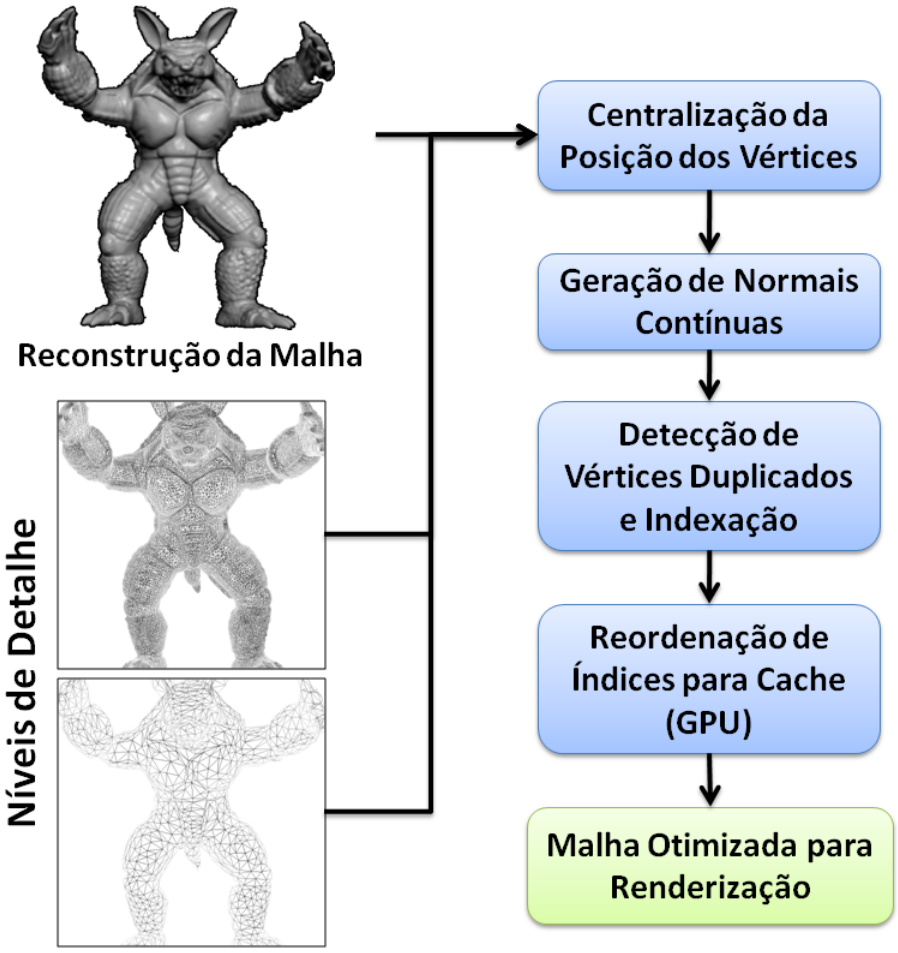


Figura 4.1. Diagrama do estágio de pré-processamento das malhas. Transforma a malha arbitrária utilizada como entrada em uma nova malha que atenda a todos os requisitos impostos pelo algoritmo de aplicação e renderização de pelos.

4.1 Centralização da Posição dos Vértices

Inicialmente, a malha de entrada pode ter seus vértices localizados em qualquer posição do espaço, fazendo com que o centro da malha não coincida com o centro do mundo (localizado na posição [0,0,0] do espaço tridimensional). Quando o centro da malha não coincide com o centro do mundo, a aplicação de transformações geométricas sobre a malha (como rotação), assim como a aplicação de outros algoritmos podem apresentar um resultado incorreto.A Figura 4.2 ilustra o resultado da transformação de rotação aplicada a dois objetos, onde o primeiro está centralizado no mundo e o segundo está posicionado em uma posição arbitrária. Note que o resultado da mesma transformação é diferente em cada um dos casos.

Neste trabalho, a centralização da malha no centro do mundo é importante por dois motivos: permite o uso da posição de cada vértice como um vetor de acesso ao mapa de ambiente, o qual é utilizado durante a renderização dos pelos (visto em detalhes

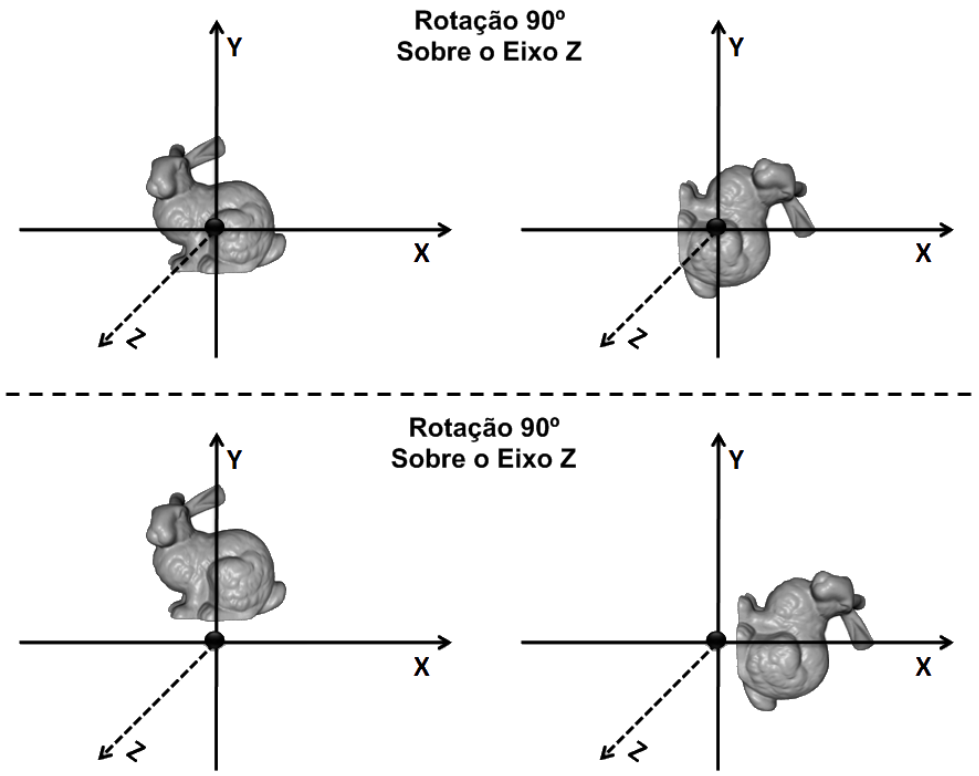


Figura 4.2. Rotação de 90 graus aplicada sobre o eixo Z do mundo. (Imagem superior) Orientação do objeto é alterada mas posição do centro do objeto não é modificada. (Imagem inferior) Orientação do objeto e posição do centro do mesmo são alteradas.

na Seção 6.3.1), e permite modificar a orientação do modelo em tempo real utilizando apenas transformações de rotação.

O algoritmo de posicionamento da malha no centro do mundo também calcula o raio da menor esfera que envolve completamente a malha. O raio da esfera é útil na definição da distância em que a câmera deve ser posicionada da malha para que a mesma possa ser vista completamente, e na definição do campo de visão da câmera. Uma porcentagem do raio da esfera tambem é utilizado como constante na aplicação de diversas operações, como translação da malha no espaço, ou aplicação de zoom na câmera. Isso torna a navegação no ambiente criado independente do tamanho da malha visualizada. O algoritmo para centralização da malha no mundo é apresentado no Algoritmo 1.

Na primeira etapa do algoritmo, o valor mínimo e máximo da posição de cada vértice da malha nos eixos X, Y e Z do mundo são calculados, e a partir dos valores de mínimo e máximo o centro da malha é calculado. Em seguida, se a posição do centro da malha não coincidir com o centro do mundo, cada um dos vértices da malha

Algorithm 1 Algoritmo para posicionamento da malha no centro do mundo e cálculo do raio da menor esfera que envolve a malha.

$minPosition(x, y, z) \leftarrow MaxInteger(x, y, z)$
$maxPosition(x, y, z) \leftarrow MinInteger(x, y, z)$
for each $vertexPosition(x, y, z) \in Mesh$ **do**
  $minPosition \leftarrow min(minPosition, vertexPosition)$
  $maxPosition \leftarrow max(maxPosition, vertexPosition)$
end for

$meshCenterPosition \leftarrow (minPosition + maxPosition) * 0.5$
if $meshCenterPosition \neq (0, 0, 0)$ **then**
  for each $vertexPosition(x, y, z) \in Mesh$ **do**
    $vertexPosition \leftarrow vertexPosition - meshCenterPosition$
  end for
end if

$sphereVertex \leftarrow meshCenterPosition - minPosition$
$sphereRadius \leftarrow VectorLength(sphereVertex)$

é transladado. Neste caso, o valor de translação usado é a diferença entre o centro da malha e o centro do mundo. Por último, o raio da menor esfera que envolve toda a malha é calculado como a distância entre o centro da malha e o vértice formado pelas maiores distâncias nos eixos X, Y e Z do centro.

4.2 Geração de Normais Contínuas

Cada vértice da malha de um objeto possui necessariamente uma posição espacial. No entanto, o armazenamento de um vetor normal em cada vértice é opcional. Os vetores normais possuem diversas aplicações, podendo ser usados para definir a direção principal de reflexão da luz sobre uma superfície, a orientação das faces de uma malha, dentre outros usos.

Neste trabalho, para que os pelos possam ser aplicados e renderizados sobre a malha, cada vértice da mesma deve necessáriamente armazenar um vetor normal. Os vetores normais são usados durante três etapas da renderização dos pelos: a geração de camadas paralelas da malha a partir da extrusão de uma malha base (apresentado na Seção 6.4.1), o endereçamento do mapa de ambiente utilizado para armazenar os dados dos pelos (apresentada na Seção 6.3.1.1) e a iluminação dos pelos (apresentada na Seção 6.3.2). Para que essas etapas da renderização produzam resultados corretos, os vetores normais armazenados nos vértices da malha devem necessáriamente ser contínuos entre faces adjacentes. Desta maneira, vértices de duas ou mais faces adjacentes que incidam

em uma mesma posição espacial devem possuir o mesmo vetor normal. Note que a malha é armazenada como uma lista de faces, não havendo compartilhamento de informações dos vértices entre faces adjacentes.

Na etapa de geração de camadas paralelas da malha, as novas camadas são geradas a partir da extrusão de uma malha base. A extrusão da malha é feita aplicando uma translação sobre cada um dos seus vértices, onde a translação aplicada é igual ao vetor normal armazenado no vértice multiplicado por uma constante de extrusão. Caso as normais dos vértices não sejam contínuas, a extrusão da malha irá gerar uma nova malha contendo buracos, como ilustrado na Figura 4.3.

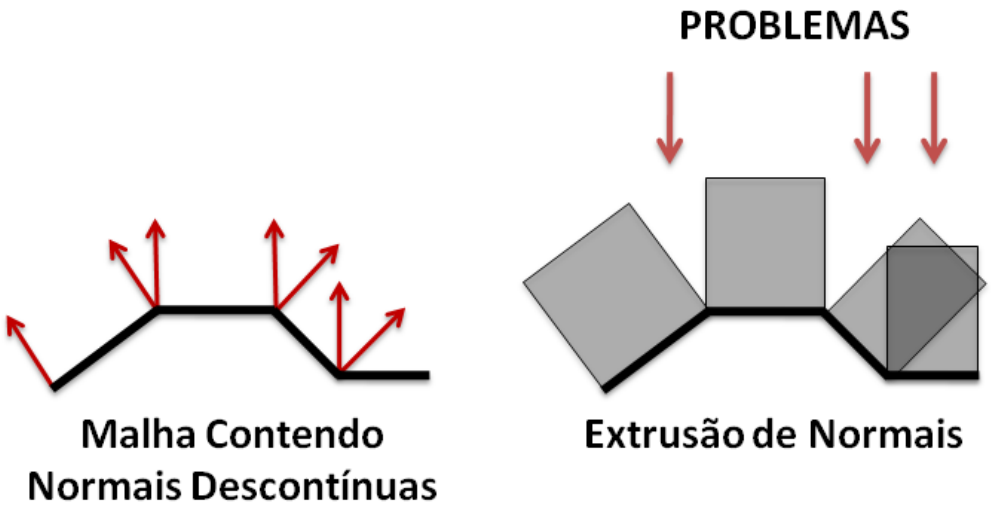
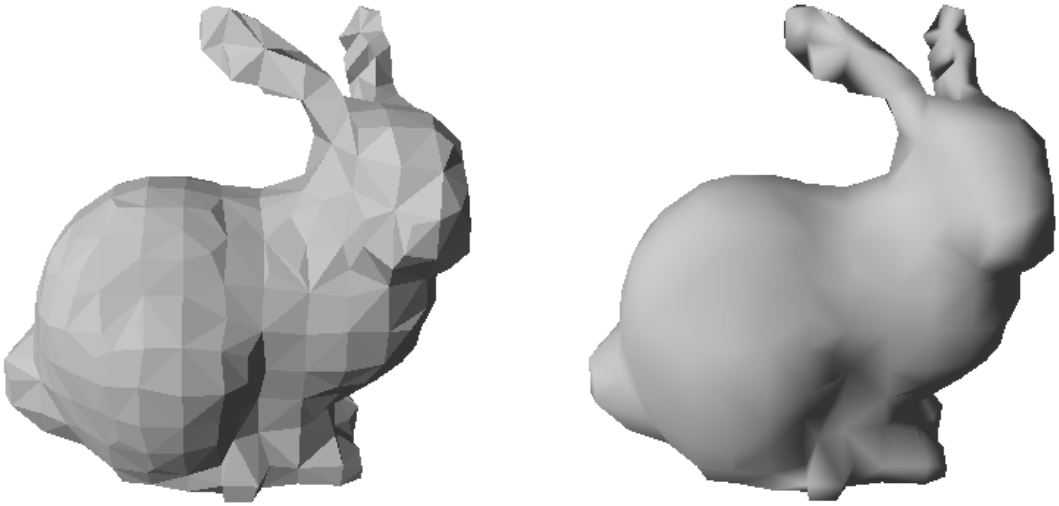


Figura 4.3. Extrusão de uma malha base contendo normais descontínuas. (Esquerda) malha base onde vértices incidentes em uma mesma posição possuem diferentes normais. (Direita) problema causado pela extrusão da malha base utilizando os vetores normais. Note os buracos e as faces sobrepostas presentes na imagem da direita.

A descontinuidade das normais também gera problemas na qualidade visual da imagem renderizada, gerando por exemplo, efeitos indesejados na iluminação com sombreamentos bruscos e chapados. A Figura 4.4 ilustra os problemas de sombreamento causados na renderização de uma malha com normais descontínuas.

Para resolver a descontinuidade de normais entre faces adjacentes são propostos dois algoritmos. Os algoritmos propostos calculam uma nova e única normal para cada posição espacial de vértice da malha, e posteriormente, o vetor normal de todos os vértices incidentes naquela posição são substituidos pelo novo vetor normal calculado. No primeiro algoritmo, a nova normal é gerada como o somatório do vetor normal de todos os vértices incidentes na mesma posição espacial. A normal gerada é normalizada, ga-



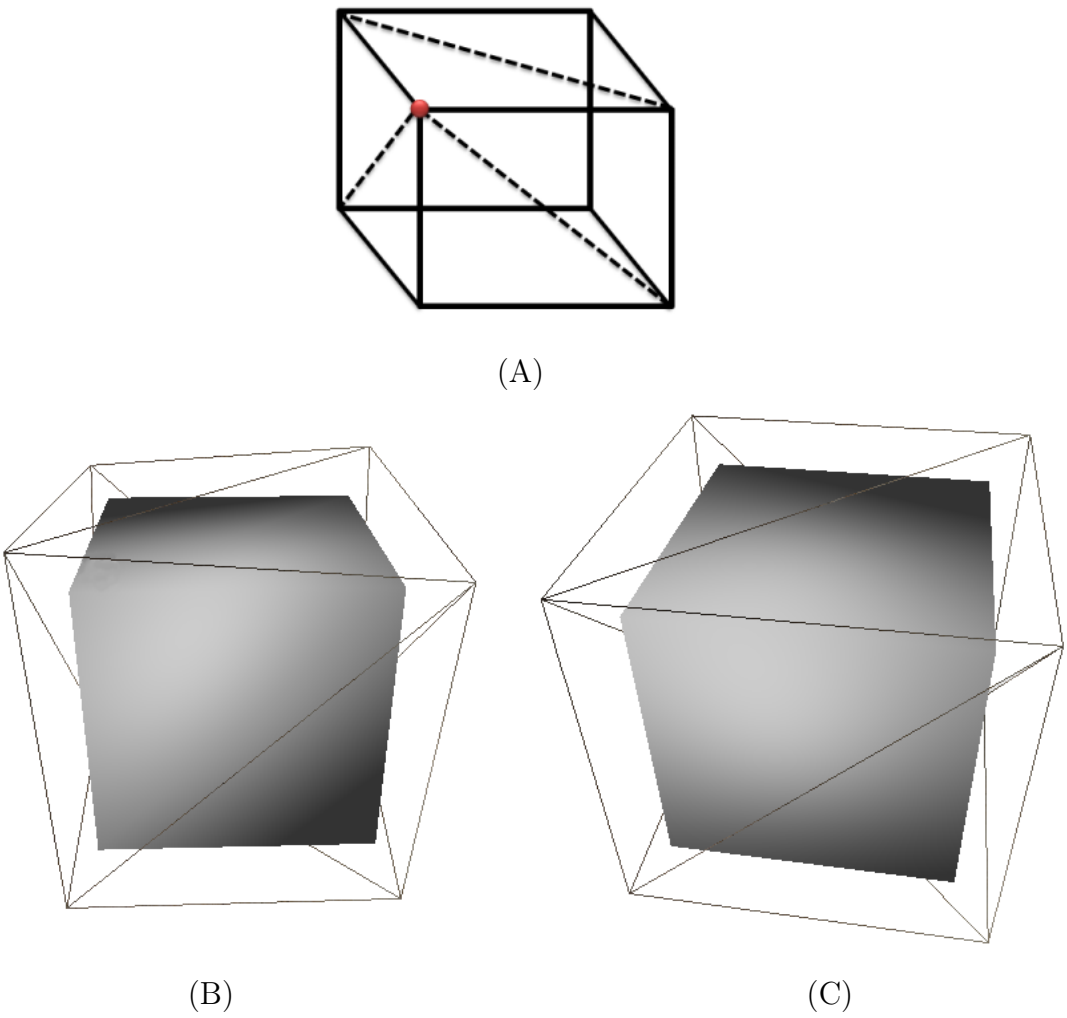
(A) (B)

Figura 4.4. Iluminação e sombreamento de *Phong* [Akenine-Möller et al., 2008] aplicados sobre duas malhas. (A) Normais descontínuas. (B) Geração de normais contínuas a partir da soma e normalização de todas as normais incidentes em uma mesma posição.

rantindo que a mesma tenha tamanho unitário. Esse algoritmo, apesar de garantir que todos os vértices incidentes em uma mesma posição possuam a mesma normal (gerando normais contínuas entre faces), permite que normais idênticas (armazenadas em faces adjacentes) sejam utilizadas mais de uma vez no somatório de normais, gerando um resultado indesejado. A Figura 4.5 ilustra um resultado indesejado causado pelo uso do primeiro algoritmo.

Na Figura 4.5, o vértice em destaque no cubo é comparilhado por 5 diferentes faces, no entanto, as duas faces laterais possuem a mesma normal, assim como as duas faces frontais que também possuem a mesma normal. Isso faz com que as normais da face frontal e lateral tenham um peso superior à normal da face superior, o que gera distorções. Para resolver esse problema, o segundo algoritmo proposto não permite que normais com valores idênticos sejam utilizadas no cálculo de uma nova normal. O segundo algoritmo proposto para geração de normais contínuas é apresentado no Algoritmo 2.

No Algoritmo 2, inicialmente é criado um *hash* que mapeia para cada posição de vértice da malha, todas as normais existentes nessa mesma posição. Em seguida, para cada posição de vértice da malha é calculado uma nova normal, a partir do somatório de todas as normais existentes naquela posição. Uma lista de normais usadas em cada posição é utilizada para garantir que normais idênticas não sejam consideradas mais de uma vez.



(A)

(B) (C)

Figura 4.5. Problema causado pelo uso do primeiro algoritmo no cálculo de um novo vetor normal para cada vértice. (A) Geometria de um cubo descrita como uma lista de faces triângulares. Vértice em destaque é compartilhado por uma face do topo do cubo, duas faces da parte frontal do cubo e duas faces da lateral do cubo. (B) Renderização do cubo apresentado em (A) com uma camada de extrusão (renderizada em *wireframe*) gerada utilizando o vetor normal gerado pelo primeiro algoritmo apresentado. Note as distorções presentes no *wireframe* da extrusão, fazendo com que a face frontal do cubo não tenha uma aparência reta. (C) Renderização do cubo e de uma camada de extrusão (em *wireframe*) gerada utilizando o vetor normal gerado pelo segundo algoritmo. Note a ausência de distorções.

4.3 Detecção de Vértices Duplicados e Indexação

Após a malha de entrada ter sido processada pelos algoritmos de centralização de malha e geração de normais contínuas, a mesma já está pronta para ser usada para aplicação e

Algorithm 2 Algoritmo para geração de normais contínuas para cada vértice da malha, sem permitir o uso de normais repetidas.

```
Hash positionToNormalMap ← ∅
for each vertex ∈ Mesh do
  positionToNormalMap[vertex.position] insert vertex.normal
end for

for each vertex ∈ Mesh do
  newNormal ← 0
  List usedNormals ← ∅

  for each normal ∈ positionToNormalMap[vertex.position] do
    if usedNormals not contains normal then
      newNormal = newNormal + normal
      usedNormals ← normal
    end if
  end for

  vertex.normal = newNormal
end for
```

renderização de pelos. Neste trabalho, optou-se pela adição de dois algoritmos ao pré-processamento da malha que têm como objetivo reduzir o volume de dados da malha e reordenar os dados da malha para tirar maior proveito da *cache* dos processadores gráficos

A malha de entrada é armazenada na forma de uma lista de faces triângulares, onde três vértices são utilizados para armazenar cada face. Esta representação pode parecer ineficiente a princípio, pois requer o armazenamento de três vértices para cada face. No entanto, os vértices utilizados em cada face podem ser indexados, o que permite o compartilhamento de vértices entre faces adjacentes.

Em uma malha fechada (sólida) armazenada como uma lista de faces triângulares contendo vértices indexados, cada vértice da malha é compartilhado na média por seis diferentes faces [Bogomjakov e Gotsman, 2001]. Desta maneira, o custo de armazenamento de uma face da malha é equivalente a metade do custo de armazenamento de um vértice [Sander et al., 2007]. Se a malha de entrada fosse armazenada como *strips* de triângulos [Akenine-Möller et al., 2008], outro tipo de representação compacta existente, no melhor caso o custo de armazenamento de uma face da malha seria equivalente ao custo de armazenamento de um vértice. Note que geralmente é necessário um grande número de *strips* de triângulos para armazenar os dados de uma malha, apesar de ser possível a criação de uma única *strip* de triângulos unindo diferentes *strips* a partir da inserção de triângulos degenerados [Akenine-Möller et al., 2008]. Desta maneira, o

uso de triângulos indexados permite uma representação de uma malha duas vezes mais compacta que utilizando *strips* de triângulos.

Para que os vértices da malha possam ser indexados, é necessário primeiramente identificar todos os vértices únicos existentes na malha. Os vértices únicos são aqueles que possuem mesma posição espacial, e mesmos atributos (como vetores normais, coordenadas de mapeamento de textura, etc). Note que a geração de normais contínuas é extremamente importante para a indexação da malha, pois faz com que todos os vértices incidentes em uma mesma posição espacial possuam o mesmo vetor normal, podendo ser mapeados para um único vértice. Após os vértices únicos da malha terem sido identificados, os mesmos são salvos em uma lista. Em seguida, é necessário gerar uma lista contendo os índices dos vértices utilizados por cada face da malha. Esta lista é construida percorrendo os vértices de todas as faces da malha, onde para cada vértice de uma face é feita uma busca na lista de vértices únicos pelo seu índice. Após a lista de índices ser gerada todos os vértices duplicados da malha podem ser removidos, restando apenas os vértices únicos. O algoritmo de detecção de dados duplicados e indexação é apresentado no Algoritmo 3.

Algorithm 3 Algoritmo de detecção de dados duplicados e indexação.

```
List uniqueVertexList ← ∅
for each vertex ∈ Mesh do
  if uniqueVertexList not contains vertex then
    uniqueVertexList ← vertex
  end if
end for

List indexList ← ∅
for each face ∈ Mesh do
  for each vertex ∈ face do
    indexList ← index of vertex in uniqueVertexList
  end for
end for

vertexBuffer ← uniqueVertexList
indexBuffer ← indexList
```

Note que o custo de armazenamento de um índice é muito inferior ao custo de armazenamento de um vértice. Neste trabalho, os índices foram armazenados utilizando 16 ou 32 *bits*, de acordo com o número de vértices únicos existentes na malha. O custo do armazenamento de um vértice contendo apenas posição e vetor normal (armazenados com ponto flutuante), por exemplo, é de 192 *bits*. Os resultados da indexação das malhas são apresentados na Seção 7.1.

4.4 Reordenação de Índices para Cache

O custo computacional da renderização de uma face de uma malha está diretamente relacionado ao custo da execução de três estágios do *pipeline* de renderização sobre está face: processamento de vértices, rasterização e processamento de fragmentos [Akenine-Möller et al., 2008]. Processadores gráficos modernos dispõem de um *cache* para armazenar os dados dos vértices que já passaram pelo estágio de processamento de vértices no *pipeline* de renderização, os quais são chamados de vértices pós-processados. Os vértices pós-processados já estão prontos para ser enviados ao próximo estágio do *pipeline* de renderização, o estágio de rasterização. Desta maneira, o uso de um cache de vértices pós-processados permite acelerar a primeira fase do pipeline de renderização. Note que os dados dos vértices armazenados neste cache geralmente são diferentes dos dados dos vértices de entrada. Por exemplo, um vértice de entrada que contém posição e vetor normal, após ser processado, pode gerar um vértice que contém posição (em coordenadas homogêneas) e coordenada de mapeamento de textura.

Para que o processador gráfico possa tirar proveito do cache de vértices pós-processados os vértices da malha renderizada devem necessáriamente estar indexados. No entanto, devido à desordenação dos vértices processados (enviados de acordo com a lista de índices) o uso do cache tende a ser ruim, não aumentando o desempenho na renderização da malha. Lin [Lin e Yu, 2006] demonstra que em um conjunto de 30 malhas diferentes avaliadas, a porcentagem de uso do *cache* quando os dados estão desordenados é na média 1.22%. Note que como um vértice é compartilhado na média por seis diferentes faces, o aproveitamente máximo do cache seria de 5/6 (ou 83.3%).

Neste trabalho, optou-se pelo uso de dois diferentes algoritmos de reordenação de índices visando tirar maior proveito do cache do processador gráfico e aumentar o desempenho na renderização de malhas arbitrárias.

O primeiro algoritmo de reordenação de índices utilizado foi o disponível através da biblioteca de extensões gráficas do DirectX, chamada D3DX (*Direct3D extensions*). O *Direct3D extensions* está presente em todas as versões recentes do DirectX, como no DirectX 9, 10 e 11. O D3DX disponibiliza vários recursos para otimização de malhas, como reordenação de índices, remoção de vértices não utilizados, geração de strips de triângulos, dentre outros. Em especial, o algoritmo de reordenação de índices permite que a reordenação seja feita para processadores gráficos modernos que possuem um cache maior, ou para processadores gráficos medianos que possuem cache de menor tamanho. Acredita-se que o algoritmo utilizado para reordenação dos índices no D3DX seja uma extensão do algoritmo proposto por Hoppe [Hoppe, 1999], devido ao mesmo ser um dos principais pesquisadores na área de computação gráfica da Micro-

soft. Note que o pré-processamento de uma malha utilizando o D3DX não implica na necessidade da renderização da mesma utilizando o DirectX. Neste trabalho, por exemplo, as malhas pré-processadas são salvas em arquivos binários, que podem ser posteriormente carregados e renderizados utilizando qualquer biblioteca gráfica. Note também que o D3DX pode ser usado em diferentes sistemas operacionais, como Linux e Mac, através de implementações da biblioteca do DirectX pelos programas Cedega (da Transgaming Technologies [Transgaming, 2009]), e CrossOver (da Codeweavers [Codeweavers, 2009]).

O segundo algoritmo de reordenação de índices utilizado foi uma implementação do algoritmo proposto por Lin [Lin e Yu, 2006]. O trabalho de Lin apresenta dois algoritmos para reordenação dos índices, onde o primeiro algoritmo considera um cache controlável (onde dados podem ser removidos de qualquer posição do cache), e o segundo considera um cache FIFO (*First In First Out*). O algoritmo implementado neste trabalho considera um cache do tipo FIFO, que é o tipo de cache encontrado nos processadores gráficos domésticos. O algoritmo proposto por Lin é um dos mais recentes existentes na literatura, e dentre os algoritmos avaliados é o que apresenta o melhor aproveitamento do cache [Sander et al., 2007].

No algoritmo proposto por Lin, o custo de renderização de cada vértice da malha é calculado, e o vértice com menor custo é adicionado ao cache. Em seguida, todas as faces que compartilham o vértice são renderizadas, o que pode fazer com que novos vértices sejam adicionados e removidos do cache. Este processo é repetido até que todos os vértices tenham sido processados. A eficiência deste algoritmo está relacionada à escolha do melhor vértice em cada passo da execução, sendo que essa escolha é feita baseada em três custos: número de vértices que deverão ser inseridos no cache, número de faces que ainda não foram renderizadas e utilizam o vértice avaliado, e posição do vértice avaliado dentro do cache FIFO quando o mesmo não é mais necessário (o que ocorre quando todas as faces que utilizam o vértice são renderizadas). O custo final do processamento de cada vértice é o somatório ponderado desses custos, sendo que Lin propõe a atribuição de pesos empíricos a cada um dos custos. A ordem de complexidade desse algoritmo, como demonstrado por Lin, é O (V * F), onde V é o número de vértices da malha e F o número de faces.

Lin apresenta em seu trabalho uma implementação alto nível do algoritmo para reordenação de índices, onde diversas omissões são feitas e alguns passos do algoritmo são apresentados superficialmente. Além disso, a implementação apresentada possui problemas se aplicada diretamente a caches do tipo FIFO. Desta maneira, o algoritmo implementado neste trabalho, o qual baseia-se no algoritmo proposto por Lin, pode apresentar pequenas diferenças se comparado ao algoritmo original. Note que em seu

site o autor é não disponibiliza o código fonto do seu algoritmo, mas é disponibilizada uma versão executável do mesmo. Nos testes realizados o programa executável disponibilizado por Lin apresentou erro ao ser executado no sistema operacional *Windows Vista*. O resultado da reordenação dos índices utilizando os algoritmos do *DirectX extensions* e Lin [Lin e Yu, 2006] são apresentados na Seção 7.1.

4.5 Sumário

Este capítulo apresentou os algoritmos utilizados no pré-processamento das malhas arbitrárias utilizadas como entrada, os quais têm como objetivo tornar a malha apta à renderização de pelos, e otimizar os dados da mesma para proporcionar uma renderização eficiente da malha. Os algoritmos apresentados neste capítulo são utilizados para: posicionar o centro da malha no centro do mundo, gerar vetores normais para os vértices da malha, detectar vértices com dados duplicados e indexar vértice únicos da malha, e por último, reordenar o uso dos vértices da malha para melhorar a localidade espacial no acesso aos dados e tirar maior proveito do *cache* dos processadores gráficos. Os resultados obtidos a partir da aplicação desses algoritmos são apresentados na Seção 7. O próximo capítulo apresenta o algoritmo utilizado para a geração dos mapas de pelos, os quais serão posteriormente aplicados sobre as malhas que passaram pelo estágio de pré-processamento.

Capítulo 5

Geração dos pelos

Este trabalho trata da representação de pelos estreitos, onde os fios do pelo gerado são paralelos e não sofrem variações de acordo com a sua altura. Os pelos estreitos utilizados neste trabalho são gerados proceduralmente e armazenados em uma única textura bidimensional. Tipos de pelos que sofrem variações de acordo com sua altura, como pelos ondulados ou enrolados, também poderiam ser gerados proceduralmente, mas seria necessário o armazenamento de várias texturas bidimensionais ou de uma textura volumétrica.

O armazenamento dos pelos na forma de uma textura bidimensional seria suficiente para sua aplicação sobre uma malha caso a mesma possuísse coordenadas de mapeamento de textura. Como não é possível garantir a presença de coordenadas de mapeamento de textura na malha, a textura dos pelos é transformada em um mapa de ambiente, o qual é utilizado para mapear os pelos sobre a malha em tempo real. *APIs* gráficas modernas, como *DirectX* e *OpenGL*, disponibilizam funções para acesso aos mapas de ambiente que são geralmente implementadas no *hardware* dos processadores gráficos, possibilitando um alto desempenho. A Figura 5.1 apresenta o diagrama do algoritmo para geração dos mapas de ambiente dos pelos.

Os pelos estreitos são armazenados em uma textura no formato *RGBA* (*red*, *green*, *blue* e *alpha*), onde os canais *RGB* da textura são utilizados para armazenar a cor de cada posição da textura, e o canal *Alpha* a transparência de cada posição. Neste trabalho, os canais *RGB* da textura são utilizados para armazenar a cor padrão dos pelos, e o canal *Alpha* é utilizado para marcar as posições da textura que contêm pelos.

A cor padrão utilizada para armazenar posições da textura com pelos é o branco opaco, representado pelo RGBA(255, 255, 255, 255), e a cor utilizada para armazenar posições sem pelos é o branco transparente, representado pelo RGBA(255, 255, 255, 0). Note que como a cor branca opaca é utilizada como a cor padrão dos pelos, o que permite que a cor dos pelos seja posteriormente modulada a outras cores, utili-

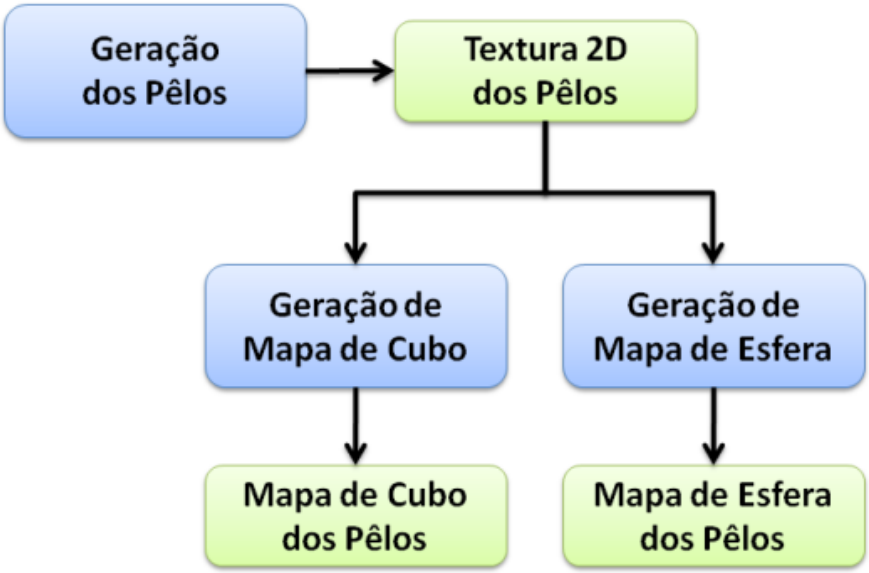


Figura 5.1. Diagrama de geração do mapa de ambiente dos pelos. Mapas de ambiente no formato de cubo e esfera são gerados a partir da textura bidimensional gerada para armazenar os pelos.

zando por exemplo, uma imagem obtida a partir de pelos reais. Também é importante notar que a cor armazenada nos canais *RGB* deve ser igual em todas as posições da textura, sendo que apenas o valor do canal *Alpha* deve ser alterado para marcar posições que possuem, ou não possuem, pelos. Isso é necessário para evitar alterações na cor padrão do pelo quando a textura é filtrada linearmente, ou bilinearmente (o que ocorre quando *mip-maps* são gerados para a mesma [Akenine-Möller et al., 2008]). Por exemplo, imagine que as posições da textura que não contêm pelos são armazenadas utilizando o valor RGBA(0, 0, 0, 0), e que o tipo de amostragem de textura utilizado é linear (amostragem padrão nas bibliotecas gráficas DirectX e OpenGL). Neste caso, a amostragem da textura é feita utilizando uma máscara 2x2, e poderia gerar como resultado o valor RGBA(128, 128, 128, 128), representando um pelo cinza escuro com 50% de transparência. Outra solução possível seria desabilitar a filtragem de texturas, no entanto, isso pode diminuir a qualidade visual da imagem gerada. O algoritmo de geração dos pelos estreitos é apresentado no Algoritmo 4.

O algoritmo para geração dos pelos tem como entrada as dimensões da textura onde os pelos serão armazenados, a densidade de pelos na textura, o tipo de vizinhança desejado na criação dos pelos e um valor semente. O tipo de vizinhança dos pelos é útilizado para impedir o crescimento de novos pelos em posições vizinhas de pelos já existentes, permitindo obter uma melhor distribuição dos pelos, assim como criar

Algorithm 4 Algoritmo de geração de pelos estreitos.

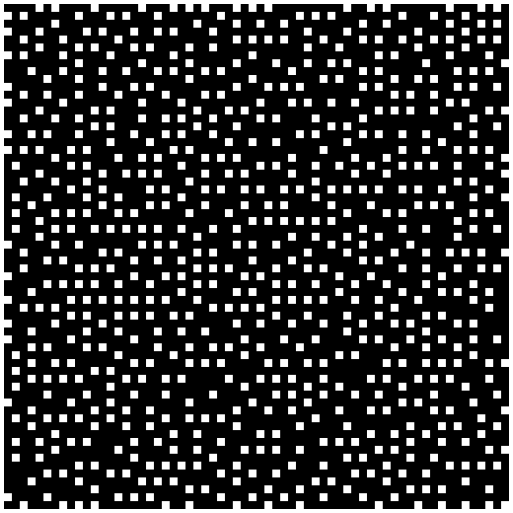
```
Texture furTexture ← Texture(width, height)
for each texel ∈ furTexture do
  texel ← RGBA(255, 255, 255, 0)
end for
hasFurStrand ← Bool(width, height)
for each value ∈ hasFurStrand do
  value ← false
end for

furStrandCount ← 0
maxFurStrands ← furDensity × furTexture.width × furTexture.height
while furStrandCount < maxFurStrands do
  furPosition ← noiseFunction(seed)
  if hasFurStrand[furPosition] = false then
    hasFurStrand[furPosition] ← true
    furTexture[furPosition] ← RGBA(255, 255, 255, 255)
    furStrandCount ← furStrandCount + 1

    setNeighborsToTrue(hasFurStrand, furPosition, neighborhoodType)
  end if
end while
```

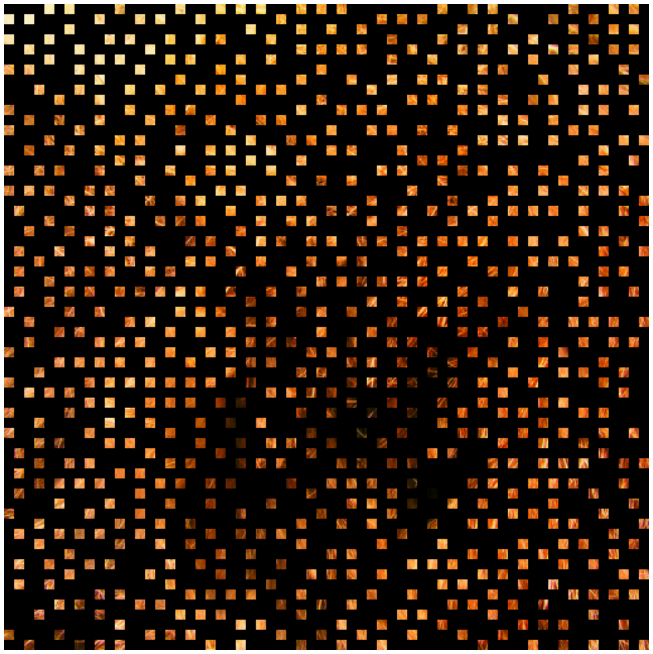
padrões no posicionamento dos pelos.

Inicialmente todos os *texels* da textura são preenchidos com a cor branco transparente, RGBA(255, 255, 255, 0). Um arranjo bidimensional do tamanho da textura é utilizado para marcar posições da textura disponíveis para criação de novos pelos. A geração dos pelos é feita utilizando uma função de ruído, que gera posições aleatórias para o crescimento de pelos a partir do valor semente utilizado na entrada. Para cada nova posição gerada pela função ruído é necessário verificar se a mesma está disponível para a criação de um novo pelo. Se a posição estiver disponível, o pelo é criado com a cor branco opaco e sua posição, assim como a dos seus vizinhos (de acordo com o tipo de vizinhança), é marcada como indisponível. Este processo é repetido até que a densidade desejada da textura de pelos tenha sido obtida. A Figura 5.2 apresenta uma textura bidimensional gerada pelo Algoritmo 4. Como citado anteriormente, a cor dos pelos pode ser modulada utilizando uma textura de coloração auxiliar, contendo por exemplo, uma imagem real de pelos. O resultado da modulação da textura de pelos com uma textura de coloração é apresentada na Figura 5.2.



(A)

(B)



(C)

Figura 5.2. Textura de pelos gerada e resultado da modulação da textura de pelos com uma textura de coloração. (A) textura de pelos gerada pelo Algoritmo 4, (B) textura de pelos reais utilizada para coloração e (C) resultado da modulação das texturas apresentadas (A) e (B). Note que em (A) e (C) a cor preta é usada para representar partes transparentes da imagem.

5.1 Geração de Mapas de Ambiente para os pelos

Mapas de ambiente apresentam uma solução em tempo real para adicionar aos objetos de uma cena contribuições do ambiente ao seu redor. Dentre as inúmeras aplicações para os mapas de ambiente pode-se citar: iluminação indireta da cena, reflexão difusa

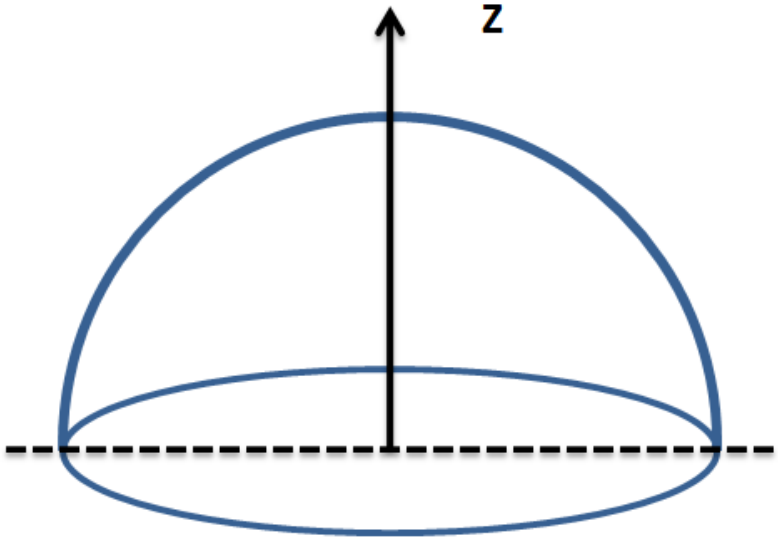
e especular, dentre outras [Akenine-Möller et al., 2008]. Neste trabalho, o mapa de ambiente é utilizado para mapear os pelos sobre malhas arbitrárias em uma cena.

Os mapas de ambiente podem ser armazenados de várias formas, como por exemplo, no formato de um cubo, cilindro, esfera ou parábola, onde cada tipo de representação do mapa de ambiente apresenta diferentes vantagens e desvantagens. Neste trabalho, optou-se pelo armazenamento dos mapas de ambiente utilizando os formatos de cubo e esfera. Os formatos de cubo e esfera são os mais utilizados por aplicativos gráficos para armazenar mapas de ambiente, sendo que as bibliotecas gráficas do DirectX e OpenGL possuem suporte nativo ao formato de cubo, e o OpenGL ao formato de esfera.

5.1.1 Mapa de Ambiente no Formato de Esfera

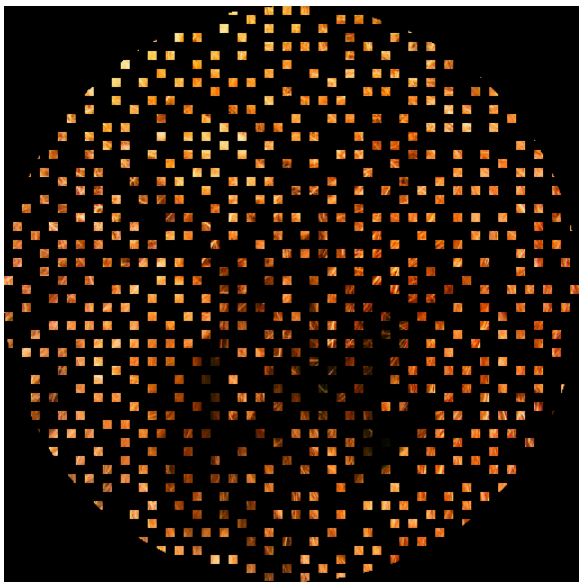
O mapa de ambiente no formato de esfera possui uma representação compacta, sendo armazenado como o interior de uma hemi-esfera, projetado paralelamente em um plano. Desta maneira, o mapa de ambiente de esfera pode ser armazenado como uma única imagem (ou textura) bidimensional. O mapa de esfera quando obtido a partir de uma esfera refletora [Debevec, 2009] não apresenta distorções, no entanto, o mesmo pode apresentar distorções quando gerado a partir de uma imagem bidimensional. Neste trabalho, o mapa de esfera é gerado a partir da textura bidimensional dos pelos, como por exemplo a apresentada na Figura 5.2. Para evitar problemas com distorções, ao invés da textura dos pelos ser mapeada em uma hemi-esfera e posteriormente projetada em um plano, optou-se pelo recorte da textura no formato de um círculo. Note que a projeção paralela da parte interior da hemi-esfera em um plano gera a imagem bidimensional de um círculo. A Figura 5.3 ilustra a representação tridimensional do mapa de ambiente no formato de esfera e apresenta dois mapas armazenados como texturas bidimensionais: o primeiro mapa obtido a partir de uma esfera refletora [Debevec, 2009] e o segundo gerado a partir da textura de pelos da Figura 5.2.

Apesar do mapa de ambiente de esfera possuir uma representação compacta, o mesmo armazena os dados do ambiente para uma única e específica direção da cena, como ilustrado na Figura 5.4. Esta restrição não afeta cenários que são observadas a partir de uma única direção fixa, pois nesses casos é possível alinhar a orientação do mapa de esfera com a direção pela qual a cena é observado. Em cenários onde a câmera pode ser orientada livremente no espaço, um único mapa no formato de esfera não é suficiente para armazenar todos os dados do ambiente. Nesses casos, pode-se utilizar um número maior de mapas de ambiente de esfera, onde cada mapa é orientado de maneira diferente no espaço. Outra possibilidade é definir uma orientação diferente, mas fixa, para o mapa de ambiente de esfera em diferentes partes de uma cena. Note que também seria possível mover a orientação do mapa de esfera de acordo com a



(A)

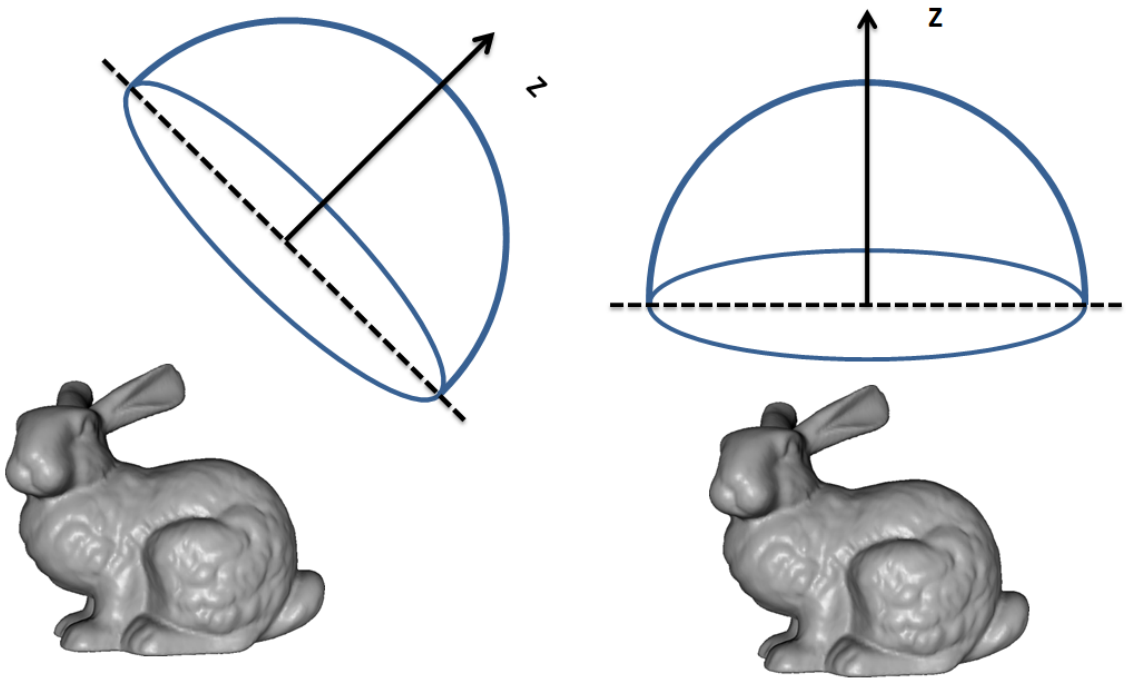
(B)



(C)

Figura 5.3. Mapas de ambiente no format de esfera. (A) representação tridimensional do mapa de ambiente no formato de esfera, (B) mapa de ambiente obtido a partir de uma esfera refletora [Debevec, 2009], (C) mapa de esfera gerado para a textura de pelos da Figura 5.2 (gerado a partir de um recorte circular da imagem original).

orientação da câmera, mas isso faria com que os detalhes mapeados sobre os objetos da cena mudassem de acordo com a câmera, um efeito muitas vezes indesejado.



(A) (B)

Figura 5.4. Orientações do mapa de ambiente de esfera. (A) Mapa orientado na diagonal direita do modelo, (B) mapa orientado sobre o modelo.

5.1.2 Mapa de Ambiente no Formato de Cubo

O mapa de ambiente no formato de cubo armazena os dados do ambiente nas seis diferentes faces de um cubo, onde cada face do cubo pode ser utilizada para armazenar diferentes dados do ambiente, como ilustrado na Figura 5.5. Este tipo de mapa possui uma representação do ambiente independente de orientação, sendo que qualquer orientação em uma cena pode ser mapeada para uma posição no mapa de ambiente, como ilustrado na Figura 5.6. Desta maneira, ele apresenta uma solução para o principal problema presente no mapa de ambiente de esfera: o armazenamento dos dados de ambiente para uma única orientação da cena. No entanto, o formato de cubo apresenta um alto custo de armazenamento, geralmente equivalente a seis vezes o custo de armazenamento do mapa de esfera, mas apresentando uma resolução seis vezes maior.

O mapa de cubo é armazenado como um arranjo de seis texturas bidimensionais, onde cada uma das texturas possui uma orientação específica dentro do cubo. Formatos convencionais utilizados para armazenamento de imagens (como BMP, PNG, JPEG e outros) não são capazes de armazenar o mapa de cubo, desta maneira é necessário o uso de um formato específico que suporte o armazenamento do mapa de cubo. Neste trabalho, optou-se pelo armazenamento do mapa de cubo no formato DDS (*Micro-*

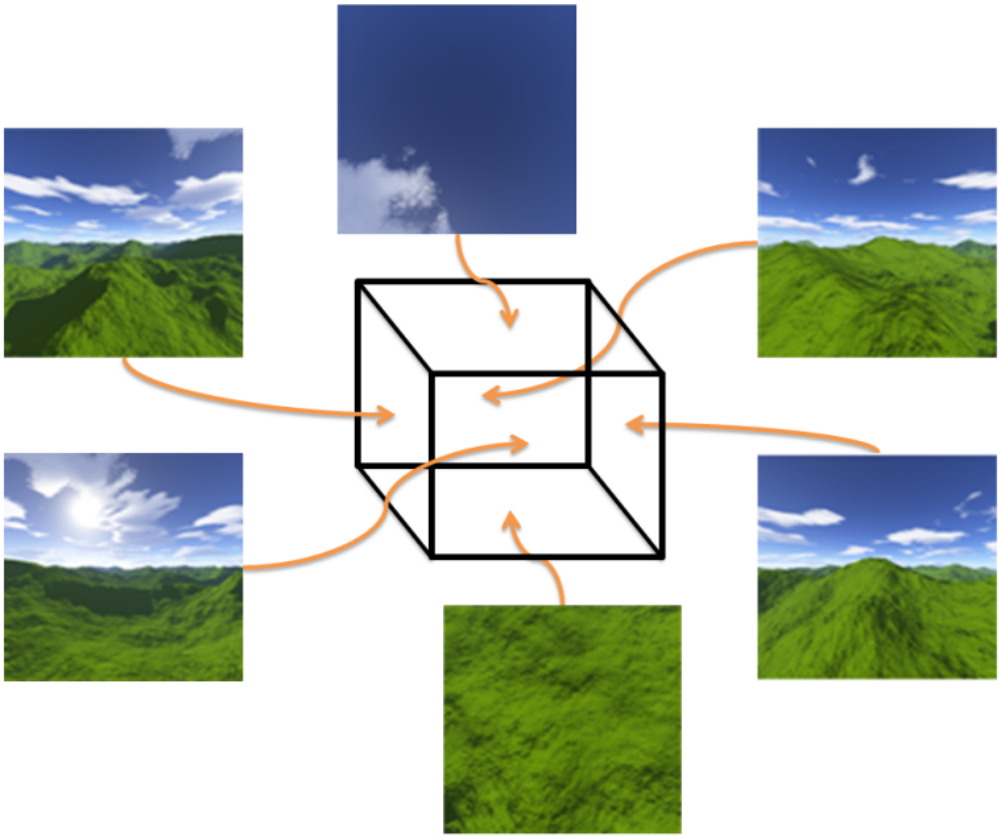


Figura 5.5. Mapa de ambiente no formato de cubo. Seis imagens diferentes são utilizadas para representar o ambiente.

soft DirectX Surface). O formato DDS possui várias vantagens sobre outros formatos, como o suporte ao armazenamento do mapa de ambiente no formato de cubo, suporte ao armazenamento de *mip-maps* [Akenine-Möller et al., 2008] e suporte a praticamente todos os tipos de compressão de dados (DX *Texture Compression*, ATI *Texture Compression*, etc) suportados pelos processadores gráficos. Note que neste trabalho também optou-se pelo armazenamento do mapa de ambiente de esfera no formato DDS.

A biblioteca gráfica do *DirectX* disponibiliza métodos para a geração de mapas de ambiente de cubos a partir de texturas bidimensionais, assim como métodos para a leitura e gravação de arquivos no formato DDS. A nVidia também disponibiliza soluções para a geração dos mapas de cubo, assim como a gravação e leitura de arquivos DDS, sendo eles o *NVIDIA Texture Tools* [nVidia, 2009], que provê uma biblioteca para tratamento de imagens com código fonte aberto, e o *Adobe Photoshop Normal Map and DDS Authoring Plug-ins* [nVidia, 2009], que provê plugins para a ferramenta de edição de imagens *Photoshop*.

De maneira geral, a renderização de um cenário tridimensional utilizando o mapa de ambiente de cubo apresenta uma melhor qualidade visual, comparado ao mapa de

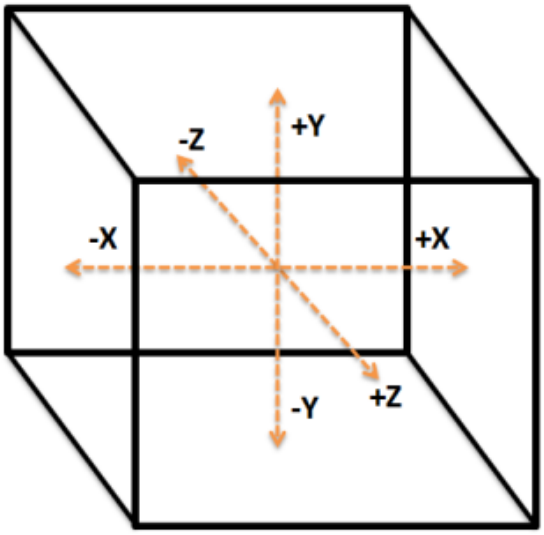


Figura 5.6. Orientações dentro do mapa de ambiente no formato de cubo. Diferente do mapa de esfera o mapa de cubo possui uma representação para qualquer posição do ambiente no espaço.

esfera, e a liberdade de orientar livremente a câmera em um cenário tridimensional. O uso de vários mapas de esfera também permite que a câmera seja orientada livremente, no entanto demanda um custo computacional superior ao uso do mapa de cubo.

5.2 Sumário

Este capítulo apresentou os algoritmos utilizados para geração de pelos estreitos, e armazenamento dos mesmos em mapas de ambiente no formato de cubo e esfera. Também foi apresentado como imagens reais de pelos podem ser utilizadas como mapas de coloração para os pelos gerados. Após os pelos serem gerados e armazenados em mapas de ambientes, os mesmos já estão prontos para serem aplicados sobre as malhas arbitrárias que passaram pelo estágio de pré-processamento. O próximo capítulo apresenta os algoritmos utilizados para geração de camadas paralelas da malha (utilizadas para aplicação dos pelos), mapeamento dos pelos gerados sobre cada camada da malha gerada, e cálculo da iluminação e sombreamento dos pelos. Os resultados visuais e o desempenho obtido na renderização dos pelos são apresentados na Seção 7.

Capítulo 6

Renderização dos pelos

Nas Seções 3, 4 e 5 foram apresentados os algoritmos utilizados para extração e reconstrução de uma malha arbitrária a partir de um objeto do mundo real, pré-processamento e otimização da malha para aplicação de pelos, e finalmente, geração dos mapas de pelos. Após a aplicação desses algoritmos a malha extraída de um objeto real já esta pronta para ser renderizada, e os pelos gerados já estão prontos para serem aplicados sobre a mesma. Neste capítulo será apresentado o algoritmo proposto para renderização dos pelos aplicados as malhas, e o ambiente desenvolvido para visualização, modificação e renderização das malhas em tempo real.

6.1 Algoritmo de Renderização dos pelos

Devido às características especiais dos pelos, apresentadas na Seção 1.1, os principais algoritmos existentes para renderização de pelos se baseiam na renderização de um volume, ou camadas discretas simulando a renderização de um volume. De maneira geral, a renderização de pelos através de um volume apresenta melhor qualidade visual, mas necessita do armazenamento de um grande volume de dados, além de um alto custo computacional e requer suporte de processadores gráficos programáveis modernos, como apresentado na Seção 2.1. Também de maneira geral, a renderização dos pelos através de camadas discretas é mais simples de ser implementado, geralmente não requer o armazenamento de dados adicionais, e pode ser utilizada em processadores gráficos não programáveis, como apresentado na Seção 2.2.

Neste trabalho, optou-se pela renderização dos pelos através da renderização de camadas discretas, sendo que os principais motivos para a escolha da renderização através de camadas foram:

1. Maior desempenho em troca de uma menor qualidade visual. Em aplicativos

interativos, como jogos e simuladores, a resposta em tempo real é geralmente mais importante do que a qualidade visual das imagens geradas.

2. Maior controle sobre o desempenho. O custo computacional da renderização de pelos utilizando camadas discretas está diretamente relacionado ao número de camadas renderizadas (que pode variar entre 1 e N) e o número de *pixels* na imagem gerada ocupados pela camada renderizada.

   Na renderização através de volumes, os mesmos são decompostos em prismas ou tetraedros, o que requer a renderização de um número mínimo de faces equivalente a 8 vezes o número de faces da malha original [Jeschke et al., 2007]. Além disso, o algoritmo de *ray-tracing* utilizado na renderização de volumes requer a execução de um número mínimo de passos para garantir seu funcionamento.

3. Redução dos dados e maior aproveitamento do cache. A renderização de camadas a partir da extrusão de uma malha base, requer que cada vértice da malha armazene sua posição espacial e vetor normal contínuo. Dados comumente armazenados para cálculo de iluminação dinâmica.

   Na renderização através de volumes geralmente é necessário armazenar informações de adjacência em cada vértice da malha, o que pode aumentar consideravelmente o espaço necessário para armazenar cada vértice e reduzir o uso do cache.

4. Redução dos dados dos pelos. Na renderização de camadas os dados dos pelos são armazenados em mapas de ambiente no formato de cubo, ou esfera.

   Na renderização de volumes é necessário utilizar uma textura volumétrica (contendo altura, largura e profundidade) para armazenar os pelos.

Utilizando como base a renderização de camadas discretas, o algoritmo de renderização dos pelos pode ser divido em dois estágios: o primeiro responsável em gerar as camadas discretas e paralelas da malha (as quais devem conter todas as informações necessárias para execução do segundo estágio), e o segundo responsável em calcular a cor final de cada ponto sobre a malha.

Neste trabalho, optou-se pela implementação do primeiro e segundo estágio da renderização dos pelos utilizando *shaders*, onde o primeiro estágio é implementado através de um *shader* de vértices, e o segundo através de um *shader* de *pixels*. O uso de *shaders* permite que a renderização dos pelos tire proveito dos processadores gráficos programáveis.

O segundo estágio da renderização dos pelos é apresentado em detalhes na Seção 6.3, e o primeiro estágio na Seção 6.4. Note que o primeiro estágio da renderização

dos pelos é apresentado após o segundo estágio da renderização. Isso ocorre devido ao primeiro estágio possuir dependências com o segundo estágio, sendo que todas as informações utilizadas como entrada no segundo estágio devem necessáriamente ser geradas pelo primeiro estágio.

6.2 Ambiente Proposto

O ambiente criado para visualização, edição e renderização de malhas foi desenvolvido utilizando a biblioteca gráfica DirectX 10, a linguagem de *shaders* HLSL (*High Level Shading Language*) e o *Shader Model* 4.0. A Figura 6.1 ilustra o ambiente desenvolvido, assim como alguns dos controles presentes no mesmo. Dentre os vários recursos presentes no ambiente criado, os principais são listados abaixo.

1. Suporte a malhas tridimensionais no formato OBJ, e MyFurModel (formato simplificado utilizado neste trabalho para armazenar as malhas pré-processadas). Permite escolher o modelo e nível de detalhe para aplicação e renderização dos pelos.

2. Navegação de câmera livre no espaço e adaptativa ao tamanho do modelo visualizado. Possui controles de translação, *arc ball rotation* e zoom.

3. Controle de luz direcional utilizando *arc ball rotation*.

4. Modificação dos parâmetros utilizados no pré-processamento da malha. Permite modificar o algoritmo de geração de normais, habilitar ou desabilitar a geração de índices, habilitar ou desabilitar a reordenação de índices e selecionar o algoritmo de reordenação a ser utilizado.

5. Modificação de parâmetros utilizados na geração do mapa de pelos. Permite modificar o tamanho do mapa de pelos, a densidade dos pelos e o mapa de coloração dos pelos utilizado.

6. Modificação de parâmetros utilizados na renderização dos pelos. Permite aumentar e diminuir o número de camadas de pelos renderizadas, modificar o espaçamento entre as camadas e selecionar o *shader* utilizado na renderização das camadas, e o *shader* utilizado na renderização da malha base.

7. Pré-processamento em lote de malhas. Permite gerar todas as combinações de pré-processamento existentes para cada malha e armazenar os mesmos. Este recurso permite visualizar como os diferentes parâmetros de pré-processamento afetam a malha, sem a necessidade de pré-processar a mesma em tempo real.

Figura 6.1. Ambiente desenvolvido para visualização, edição e renderização de malhas contendo pelos. (Centro) renderização de uma malha contendo pelos. (Direita) alguns dos controles presentes no ambiente.

6.3 Segundo Estágio da Renderização dos pelos

O segundo estágio da renderização é executado para cada uma das camadas paralelas da malha geradas no primeiro estágio da renderização. As malhas geradas no primeiro estágio são enviadas para o segundo estágio em ordem crescente de nível de extrusão. Desta maneira, malhas com menor nível de extrusão são processadas antes de malhas com maior nível de extrusão. A Figura 6.2 apresenta o diagrama do segundo estágio da renderização dos pelos.

Neste estágio, o mapa de ambiente contendo pelos e o mapa de coloração dos pelos são mapeados sobre cada camada processada. Em seguida, a iluminação é calculada para cada ponto visível sobre a camada processada. Os algoritmos utilizados para mapeamento de ambiente e iluminação são apresentados em detalhes nas próximas seções.

6.3.1 Mapeamento de Ambiente

Os pelos que serão aplicados sobre as camadas da malha, os quais foram gerados proceduralmente na Seção 5, assim como o mapa de coloração dos pelos, estão armazenados

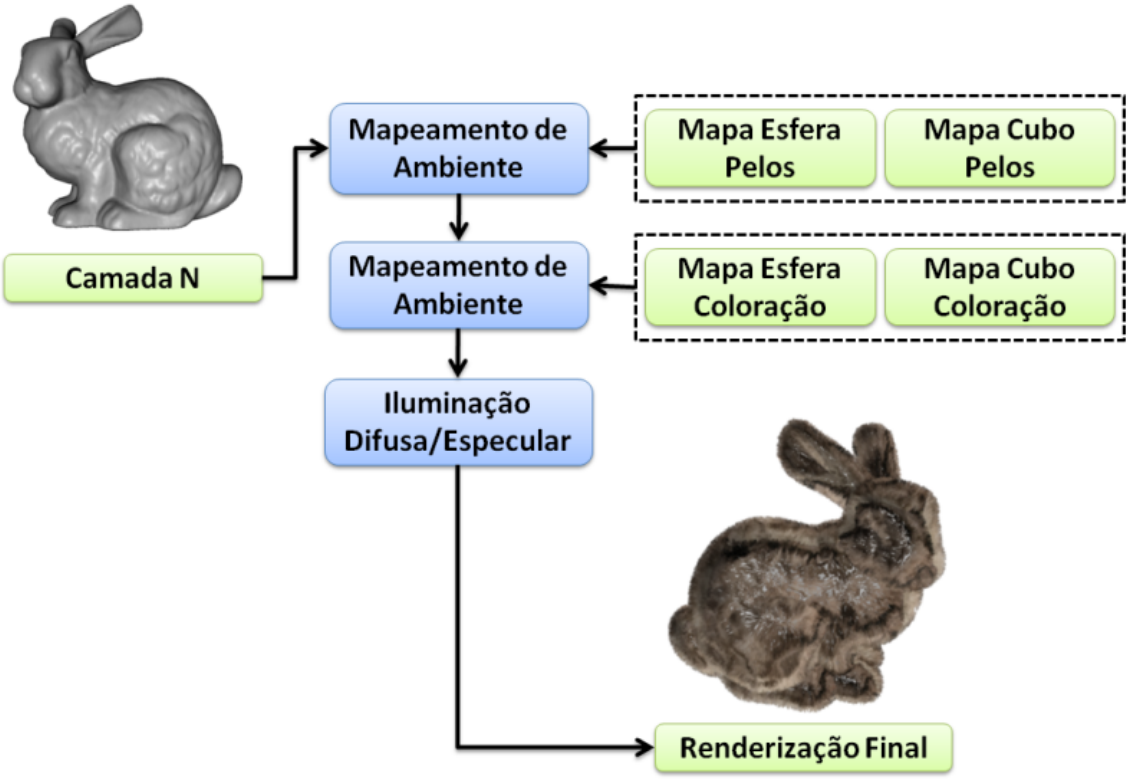


Figura 6.2. Segundo estágio da renderização dos pelos. Mapeamento dos mapas de pelos e coloração de pelos sobre a malha e iluminação para cada ponto da malha.

na forma de mapas de ambiente. Os mapas de ambiente podem estar no formato de cubo, ou no formato de esfera, sendo que em ambos os casos o mapa de ambiente é acessado através de um vetor tridimensional. Como o propósito do mapa de ambiente é armazenar contribuições do ambiente ao redor de um objeto, o vetor tridimensional utilizado para acessar o mapa de ambiente representa a direção na cena tridimensional em que o mapa deve ser amostrado. Note que em algumas bibiotecas gráficas os mapas de ambiente são acessados diretamente através das coordenadas de um vetor tridimensional, enquanto em outras bibliotecas gráficas pode ser necessário converter as coordenadas do vetor tridimensional para coordenadas reais de acesso ao mapa (as quais variam de acordo com a forma de armazenamento do mapa). A Seção 6.3.1.1 mostra como o vetor tridimensional de acesso ao mapa de ambiente é transformado para coordenadas reais de acesso aos mapas no formato de cubo e esfera, e a Seção 6.3.1.2 mostra como o vetor tridimensional de acesso (ou endereçamento) dos mapas de ambiente é calculado.

6.3.1.1 Amostragem dos Mapas de Cubo e Esfera

A partir do vetor tridimensional utilizado para amostrar o mapa de ambiente pode ser necessário gerar a coordenada real de acesso aos dados dos mapas de cubo e esfera, a qual é dependente da forma como os mapas são armazenados. O mapa de ambiente no formato de cubo é armazenado como seis imagens bidimensionais (uma para cada face do cubo), como citado na Seção 5.1.2. Desta maneira, é necessário definir qual face do cubo deve ser amostrada, e em seguida, gerar um vetor bidimensional para amostrar a imagem mapeada na face selecionada do cubo. O mapa de ambiente no formato de esfera é armazenado como uma imagem bidimensional, como citado na Seção 5.1.1. Desta maneira, o vetor tridimensional usado na amostragem do mapa de esfera deve ser transformado em um vetor bidimensional utilizado para amostrar uma imagem.

O Algoritmo 5 apresenta a transformação do vetor tridimensional de acesso ao mapa no formato de cubo em coordenadas reais de acesso ao mapa. Neste algoritmo, *vec* representa o vetor tridimensional de entrada utilizado para amostrar o mapa de cubo, $cubeFace$ representa a face que será amostrada do cubo, e $mapU$ e $mapV$ as coordenadas de amostragem da imagem armazenada na face selecionada do cubo. A linguagem de *shaders* HLSL, utilizada neste trabalho, provê métodos para amostragem direta do mapa de cubo a partir de um vetor tridimensional, não sendo necessária a aplicação do Algoritmo 5. Além disso, espera-se que o acesso ao mapa de cubos seja o mais otimizado possível, devido ao mesmo ser implementado diretamente na linguagem de *shader*.

O Algoritmo 6 apresenta a transformação do vetor tridimensional de acesso ao mapa no formato de esfera em coordenadas reais de acesso ao mapa. Neste algoritmo, *vec* representa o vetor tridimensional de entrada utilizado para amostrar o mapa de esfera, e $mapU$ e $mapV$ as coordenadas de amostragem da imagem armazenada no mapa de esfera. Note que esse algoritmo considera um único mapa de esfera, o qual está com sua orientação alinhada com ao vetor (0, 0, 1). A linguagem de *shaders* HLSL não provê métodos para amostragem direta do mapa de esfera, sendo necessário a implementação do Algoritmo 6 no *shader* criado.

6.3.1.2 Vetor Tridimensional de Amostragem do Mapa de Ambiente

O vetor tridimensional utilizado para acessar o mapa de ambiente geralmente é calculado como sendo a reflexão do vetor de visão sobre cada ponto da superfície da malha renderizada. Isto faz com que a malha funcione como um espelho, refletindo especularmente as contribuições do mapa de ambiente [Akenine-Möller et al., 2008]. Esta abordagem, apesar de produzir bons resultados visuais, faz com que o acesso ao mapa de ambiente seja dependente de fatores externos, como o vetor de visão. Desta ma-

Algorithm 5 Algoritmo de transformação do vetor tridimensional de acesso ao mapa no formato de cubo em coordenadas reais de acesso a uma das faces do cubo.

$\vec{vec} \leftarrow$ Vetor tridimensional de amostragem do mapa de ambiente
$cubeFace \leftarrow 0$
$mapU \leftarrow 0$
$mapV \leftarrow 0$

if $|\vec{vec}_x| \geq |\vec{vec}_y|$ and $|\vec{vec}_x| \geq |\vec{vec}_z|$ **then**
  if $\vec{vec}_x > 0$ **then**
    $cubeFace \leftarrow$ CUBEMAP_POSITIVE_X_FACE
    $mapU \leftarrow -\vec{vec}_z/|\vec{vec}_x|$
    $mapV \leftarrow -\vec{vec}_y/|\vec{vec}_x|$
  else
    $cubeFace \leftarrow$ CUBEMAP_NEGATIVE_X_FACE
    $mapU \leftarrow \vec{vec}_z/|\vec{vec}_x|$
    $mapV \leftarrow -\vec{vec}_y/|\vec{vec}_x|$
  end if
else if $|\vec{vec}_y| \geq |\vec{vec}_x|$ and $|\vec{vec}_y| \geq |\vec{vec}_z|$ **then**
  if $\vec{vec}_y > 0$ **then**
    $cubeFace \leftarrow$ CUBEMAP_POSITIVE_Y_FACE
    $mapU \leftarrow \vec{vec}_x/|\vec{vec}_y|$
    $mapV \leftarrow \vec{vec}_z/|\vec{vec}_y|$
  else
    $cubeFace \leftarrow$ CUBEMAP_NEGATIVE_Y_FACE
    $mapU \leftarrow \vec{vec}_x/|\vec{vec}_y|$
    $mapV \leftarrow -\vec{vec}_z/|\vec{vec}_y|$
  end if
else if $|\vec{vec}_z| \geq |\vec{vec}_x|$ and $|\vec{vec}_z| \geq |\vec{vec}_y|$ **then**
  if $\vec{vec}_z > 0$ **then**
    $cubeFace \leftarrow$ CUBEMAP_POSITIVE_Z_FACE
    $mapU \leftarrow \vec{vec}_x/|\vec{vec}_z|$
    $mapV \leftarrow -\vec{vec}_y/|\vec{vec}_z|$
  else
    $cubeFace \leftarrow$ CUBEMAP_NEGATIVE_Z_FACE
    $mapU \leftarrow -\vec{vec}_x/|\vec{vec}_z|$
    $mapV \leftarrow -\vec{vec}_y/|\vec{vec}_z|$
  end if
end if

$mapU \leftarrow (mapU + 1)/2$
$mapV \leftarrow (mapV + 1)/2$

neira, uma posição na superfície da malha pode amostrar diferentes posições no mapa de ambiente, de acordo com o vetor de visão, e da posição e orientação da malha no espaço. No mapeamento dos pelos sobre a malha é desejado que:

Algorithm 6 Algoritmo de transformação do vetor tridimensional de acesso ao mapa no formato de esfera em coordenadas reais de acesso ao mapa.

$\vec{vec} \leftarrow$ Vetor tridimensional de amostragem do mapa de ambiente
$mapU \leftarrow 0$
$mapV \leftarrow 0$

$length \leftarrow \sqrt{\vec{vec}_x^2 + \vec{vec}_y^2 + (\vec{vec}_z + 1)^2}$
$mapU \leftarrow \vec{vec}_x/(2 * length) + 0.5$
$mapV \leftarrow \vec{vec}_y/(2 * length) + 0.5$

1. O acesso ao mapa de ambiente seja independente de fatores externos, fazendo com que uma posição sobre a malha sempre amostre uma mesma posição do mapa de ambiente.

2. Dois vértices pertencentes a uma mesma face da malha não amostrem a mesma posição do mapa de ambiente, evitando assim a criação de *aliasings*.

3. O número de diferentes amostragens realizadas sobre o mapa de ambiente por cada face da malha seja idealmente proporcional ao tamanho da face, fazendo com que cada ponto da malha amostre um diferente valor do mapa de ambiente.

Para atender aos requisitos desejados na amostragem do mapa de ambiente, foram estudadas três abordagens diferentes. Na primeira abordagem, o mapa de ambiente foi endereçado utilizando o vetor normal de entrada dos vértices (sem a aplicação de transformações, como as de mundo e visão). Esta abordagem atende ao primeiro requisito do mapeamento, e também tende a atender o segundo requisito em malhas que apresentam grande variação de curvatura. Na segunda abordagem, o mapa de ambiente foi endereçado utilizando um vetor unitário representando a direção de cada ponto sobre a malha (sem a aplicação de transformações). Esta abordagem atende ao primeiro requisito do mapeamento, e também tende a atender ao terceiro requisito, pois quanto maior a distância entre vértices de uma face maior é a variação em suas posições. Por último, na terceira abordagem o mapa de ambiente foi endereçado utilizando a combinação dos endereços gerados na primeira e segunda abordagem. O resultado da aplicação dessas três diferentes abordagens para a amostragem do mapa de ambiente é apresentado na Seção 7.2.1.

6.3.2 Iluminação dos pelos

A iluminação é um dos fatores mais importantes utilizados para adicionar realismo a renderização dos pelos. Dentre os vários modelos de iluminação presentes na literatura,

um dos mais populares é o modelo empírico de Phong [Akenine-Möller et al., 2008]. No modelo de Phong, a intensidade de luz que é refletida a partir de uma posição da superfície em uma determinada direção é calculada como a soma de dois componentes: reflexão difusa e reflexão especular. O modelo de Phong apesar de ser simples de ser implementado e apresentar um alto desempenho não consegue representar com realismo alguns materiais complexos, como pelos, pele humana e outros.

Os modelos reais existentes para a iluminação de cabelos (ou pêlos longos) utilizam *sub-surface scattering* [Akenine-Möller et al., 2008], fazendo com que o cálculo das constribuições de iluminação dos cabelos seja feito considerando várias camadas de reflexão de luz, como no trabalho proposto por Moon [Moon et al., 2008]. Esse modelo apresenta uma alta qualidade visual, mas ainda não pode ser aplicado em tempo real.

Kajiya e Kay [Kajiya e Kay, 1989] propuseram um modelo para iluminação de pelos baseado no modelo empírico de Phong, onde cilindros são utilizados no cálculo da intensidade da reflexão difusa e especular da luz. Neste trabalho, optou-se pelo uso do modelo de iluminação proposto por Kajiya e Kay, devido ao mesmo apresentar uma boa qualidade visual e alto desempenho.

No modelo proposto por Kajiya e Kay, a componente de reflexão difusa é calculada de acordo com as Equações 6.1 e 6.2. O Algoritmo 7 apresenta a implementação otimizada das Equações 6.1 e 6.2 utilizando *shaders*.

$$angleTL = angleBetweenVectors(tangent, -lightVector) \tag{6.1}$$

$$diffuseIntensity = sin(angleTL) \tag{6.2}$$

Algorithm 7 Algoritmo para cálculo da intensidade de reflexão difusa da luz.

$tangentVector \leftarrow input.normalWorld$
$tangentDotLight \leftarrow tangentVector \cdot -lightVector$
$sinTL \leftarrow \sqrt{1 - tangentDotLight * tangentDotLight}$
$diffuseLightIntensity \leftarrow sinTL$

Note que nas Equações 6.1 e 6.2, e no Algoritmo 7, a normal de cada ponto de cada camada da malha é utilizada como tangente dos pelos, o que ocorre devido aos pelos serem estreitos, sempre tendo a mesma orientação da normal da malha. No Algoritmo 7, $input.normalWorld$ representa o vetor normal de uma posição da malha, $tangentVector$ represente a tangente dos pelos, que para pelos estreitos é igual à normal do ponto e $lightVector$ representa o vetor direção da luz.

No modelo proposto por Kajiya e Kay, a componente de reflexão especular é calculada de acordo com as Equações 6.3, 6.4 e 6.5. O Algoritmo 8 apresenta a implemen-

tação otimizada das Equações 6.3, 6.4 e 6.5 utilizando *shaders*.

$$angleTL = angleBetweenVectors(tangent, -lightVector) \tag{6.3}$$

$$angleTE = angleBetweenVectors(tangent, eyeVector) \tag{6.4}$$

$$specularIntensity = |cos(angleTL) * cos(angleTE) + sin(angleTL) * sin(angleTE)| \tag{6.5}$$

Algorithm 8 Algoritmo para cálculo da intensidade de reflexão specular da luz.

$tangentVector \leftarrow input.normalWorld$
$tangentDotLight \leftarrow tangentVector \cdot -lightVector)$
$sinTL \leftarrow \sqrt{1 - tangentDotLight * tangentDotLight}$

$tangentDotEye \leftarrow tangentVector \cdot input.eyeVector)$
$sinTE \leftarrow \sqrt{1 - tangentDotEye * tangentDotEye}$
$specularLightIntensity \leftarrow |tangentDotLight * tangentDotEye + sinTL * sinTE|$

No Algoritmo 8, *input.eyeVector* representa o vetor de visão que tem sua origem na posição da câmera e aponta para a posição da malha onde o cálculo de iluminação está sendo realizado.

A Figura 6.3 apresenta os vetores utilizados no cálculo das componentes de reflexão difusa e especular. Neste figura, T representa o vetor tangente, N representa uma das normais do cilindro, V representa o vetor de visão e H representa o vetor mediano entre T e N (o qual não foi utilizado no cálculo).

6.3.3 Shader de Pixels

O Algoritmo 9 apresenta o *shader* de *pixels* criado e utilizado no segundo estágio da renderização, o qual utiliza o mapa de ambiente no formato de cubo. Note que o código do *shader* é apresentado utilizando uma notação de alto nível que nao é dependente da linguagem de *shaders* HLSL.

No Algoritmo 9, a cor final de cada ponto na superfície da malha renderizada é calculada. Esse algoritmo combina os algoritmos 5, 7 e 8 apresentados nas seções anteriores.

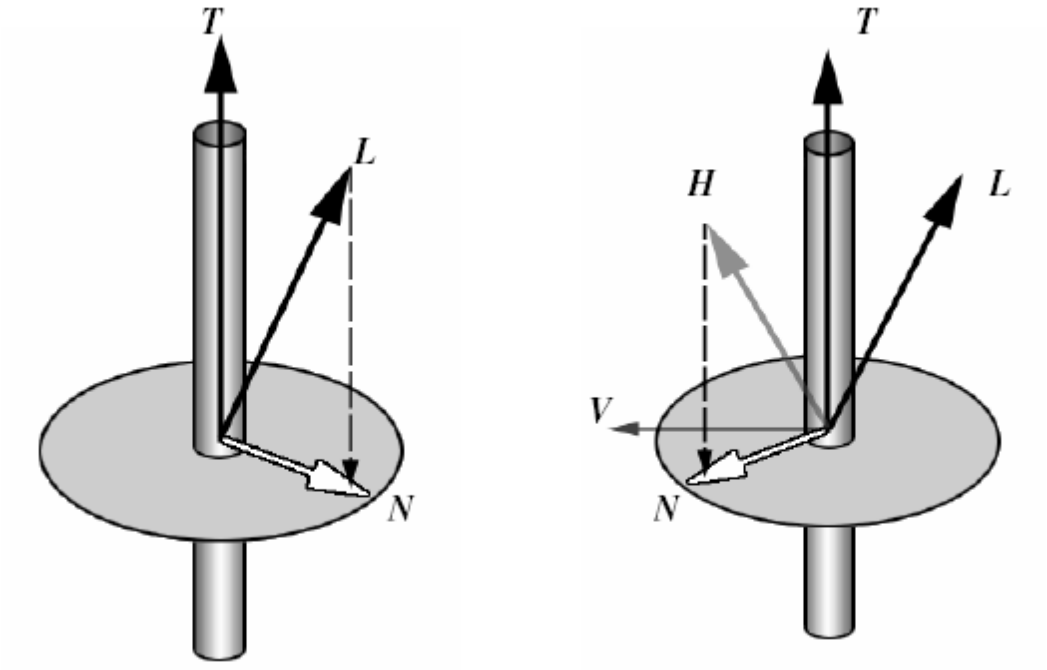


Figura 6.3. Vetores utilizados no cálculo das componentes de reflexão difusa e especular [Sheppard, 2004].

6.4 Primeiro Estágio da Renderização dos pelos

No primeiro estágio da renderização dos pelos, diferentes níveis de extrusão são aplicados sobre uma malha base, onde para cada nível de extrusão aplicado uma nova camada discreta e paralela da malha é gerada. Em seguida, todas as informações adicionais da malha necessárias para a execução do segundo estágio da renderização são geradas. A Figura 6.4 apresenta o diagrama do primeiro estágio da renderização dos pelos.

6.4.1 Extrusão da Malha

A extrusão da malha é realizada em tempo real durante o primeiro estágio da renderização dos pelos, e tem como objetivo gerar novas camadas paralelas da malha, as quais serão utilizadas para aplicação e renderização dos pelos. Devido à extrusão da malha ser realizada em tempo real, não é necessário a prévia geração e armazenamento das camadas da malha. Além disso, também é possível definir o número de camadas a serem geradas em tempo real, de acordo com a qualidade visual e desempenho desejados.

A extrusão da malha é realizada aplicando uma translação sobre a posição espacial de cada um dos vértices da malha. Neste trabalho, optou-se pela extrusão da malha utilizando os vetores normais armazenados em cada um dos seus vértices. Desta

Algorithm 10 Algoritmo para cálculo da intensidade de reflexão specular da luz.

// Normaliza os dados de entrada
$position \leftarrow normalize(input.position)$
$normalVector \leftarrow normalize(input.normal)$
$normalWorldVector \leftarrow normalize(input.normalWorld)$
$eyeVector \leftarrow normalize(input.eyeVector)$

// Utilizando mapa de ambiente no formato de cubo
// Amostra o mapa de pelos e o mapa de coloração dos pelos
$environmentCoord \leftarrow normalize(normalVector + normalize(position))$
$furSample \leftarrow sample(environmentCubeFur, environmentCoord)$
$furColorSample \leftarrow sample(environmentCubeFurColor, environmentCoord)$

// Calcula os coeficientes de reflexão difusa e especular
$tangentVector \leftarrow normalWorldVector$
$tangentDotLight \leftarrow tangentVector \cdot -lightVector)$
$sinTL \leftarrow \sqrt{1 - tangentDotLight * tangentDotLight}$
$tangentDotEye \leftarrow tangentVector \cdot input.eyeVector)$
$sinTE \leftarrow \sqrt{1 - tangentDotEye * tangentDotEye}$
$diffuseLightIntensity \leftarrow sinTL$
$specularLightIntensity \leftarrow |tangentDotLight * tangentDotEye + sinTL * sinTE|$
$ambientLightIntensity \leftarrow 0.1$

// Calcular a cor de saída
$outputColor.rgb \leftarrow furSample * furColorSample * (ambientLightIntensity + diffuseLightIntensity) + specularLightIntensity^{32}$
$outputColor.a \leftarrow 1.0$

maneira, a translação aplicada sobre cada vértice da malha possui a direção do vetor normal armazenado naquele vértice. A magnitude da translação de cada vértice é definida arbitrariamente e modulada de acordo com o raio da malha. Para que a extrusão da malha a partir das normais armazenadas em seus vértices seja capaz de gerar uma nova malha contínua, os vetores normais devem necessáriamente ser contínuos entre faces adjacentes, como citado na Seção 4.2. É importante notar que a extrusão aplicada sobre a malha modifica apenas a posição espacial dos seus vértices, sendo que outros atributos armazenados nos vértices, como vetor normal e coordenadas de mapeamento de textura não são modificados.

Neste trabalho, optou-se pela implementação do algoritmo de extrusão da malha através de um *shader* de vértices. O processamento de vértices é um dos primeiros estágios do *pipeline* de renderização, e é executado uma única vez para cada vértice da malha renderizada [Akenine-Möller et al., 2008]. Como a extrusão é realizada a partir do processamento de vértices, a malha de entrada deve ser renderizada uma

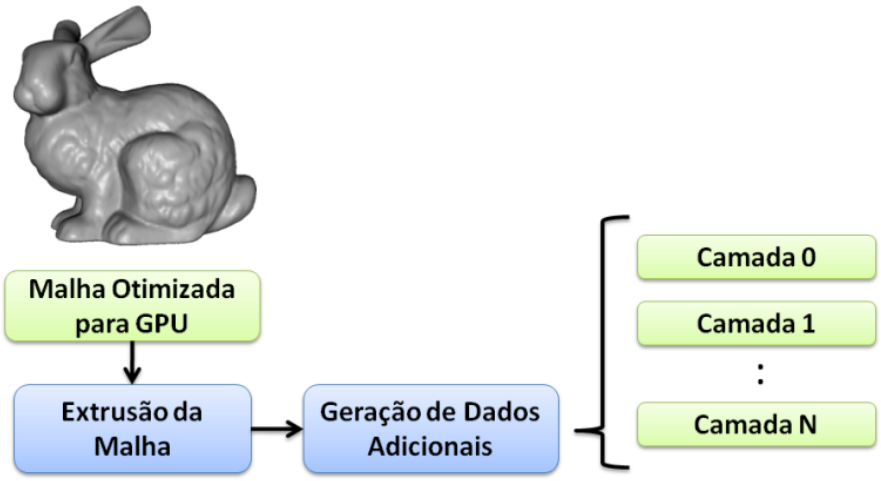


Figura 6.4. Primeiro estágio da renderização dos pelos. Diferentes níveis de extrusão são aplicados sobre a malha base gerando novas camadas discretas da malha. Para cada camada gerada todas as informações adicionais da malha (necessárias para a execução do segundo estágio da renderização) são geradas. Note que apenas uma camada é gerada para cada execução do algoritmo.

vez para cada camada a ser gerada. Note que a extrusão da malha também pode ser implementada na *CPU*, onde seria necessário a modificação do *vertex buffer* da malha antes que o mesmo fosse enviado ao *pipeline* de renderização.

Para que a extrusão possa ser realizada a partir do *shader* de vértices, o mesmo deve receber como entrada a posição espacial e o vetor normal de cada vértice da malha. Além disso, o processamento de vértice também deve ter como entrada a magnitude da translação aplicada sobre os vértices (que também pode ser interpretado como o espaçamento existente entre as camadas geradas) e o número da camada atual renderizada. Note que a magnitude da translação aplicada sobre os vértices é constante para todas as camadas da malha geradas, gerando camadas com espaçamento uniforme. No processamento de vértices, a nova posição de cada vertice da malha é calculada de acordo com a equação a seguir:

$$P' = P + \vec{normal} * translationMagnitude * layerNumber \tag{6.6}$$

Na Equação 6.6, P' é a posição final do vértice, P a posição inicial do vértice, $normal$ o vetor normal do vértice, $translationMagnitude$ a magnitude da translação aplicada (equivalente à distância entre cada camada) e $layerNumber$ o número da camada. Note que o número da camada gerada deve variar entre 1 e N, devido a camada de número zero representar a malha base, onde nenhuma extrusão é aplicada.

O espaçamento entre as camadas de pelos, representado pela variável $translationMagnitude$, não pode ser um valor constante para todas as malhas, pois é

desejado que a altura dos pelos renderizados seja proporcional ao tamanho da malha renderizada. Desta maneira, o espaçamento entre as camadas precisa ser calculado de acordo com o raio da malha renderizada. O espaçamento entre as camadas é calculado de acordo com a equação a seguir:

$$translationMagnitude = K_1 * meshRadius * translationScale \tag{6.7}$$

Na Equação 6.7, K_1 é uma constante que define a porcentagem do raio da malha utilizado como magnitude da tranlação, $meshRadius$ é o raio da malha renderizada e $translationScale$ é um valor de escala definido pelo usuário em tempo real. Finalmente, a posição final de cada vértice no espaço homogêneo é calculada multiplicando a posição do mesmo pelas matrizes de mundo, visão e projeção. Esta posição é utilizada como saída no processamento de vértices, e é utilizada pelo estágio de rasterização do pipeline de processamento gráfico [Akenine-Möller et al., 2008].

6.4.2 Geração Dados Adicionais

Além de aplicar uma extrusão sobre cada vértice da malha, o processamento de vértices também é responsável em gerar todas as informações relativas aos vértices que serão necessárias durantes as etapas seguintes da renderização. Desta maneira, o processamento de vértices é responsável em gerar as informações que serão necessárias durante o segundo estágio da renderização dos pelos, apresentado na Figura 6.4.

Durante o segundo estágio da renderização dos pelos são executados os algoritmos de mapeamento de ambiente e iluminação, os quais são apresentados em detalhes respectivamente nas Seções 6.3.1 e 6.3.2. O algoritmo de mapeamento de ambiente requer como entrada a posição inicial e a normal de cada vértice, onde ambos devem estar no espaço de coordenadas do objeto. Note que a posição inicial do vértice é diferente da posição final gerada pelo algoritmo de extrusão, e além disso, a posição final é transformada para o espaço de coordenadas homogêneo. O algoritmo de iluminação requer como entrada o vetor normal e o vetor visão de cada vértice, além do vetor de direção da luz que é constante para todos os vértices. Os vetores utilizados pela iluminação devem ser transformados pela matriz de mundo, para que os cálculos de iluminação sejam realizados corretamente. Desta maneira, os dados relacionados aos vértices que precisam ser enviados ao próximo estágio do pipeline de renderização são: posição inicial, vetor normal, vetor normal transformado pela matriz de mundo e vetor de visão.

6.4.3 Shader de Vértices

O Algoritmo 11 apresenta o *shader* de vértices criado e utilizado no primeiro estágio da renderização. Note que o código do *shader* é apresentado utilizando uma notação de alto nível que não é dependente da linguagem de *shaders* HLSL.

Algorithm 11 Algoritmo de processamento de vértices. Algoritmo utilizado para gerar as novas camadas paralelas da malha, e as informações adicionais utilizadas nos estágios seguintes da renderização.

$input \leftarrow$ input vertex
$output \leftarrow$ output vertex sent to the next stage
$matrixWV \leftarrow matrixWorld * matrixView$
$matrixWVP \leftarrow matrixWorld * matrixView * matrixProjection$

// Calculate the final vertex coordinate in homogeneous space
$magnitude \leftarrow translationMagnitude * layerNumber$
$finalPosition \leftarrow input.position + input.normalVector * magnitude$
$output.homogeneousPosition \leftarrow finalPosition * matrixWVP$

// Output data needed by the environment mapping algorithm
$output.position \leftarrow input.position$
$output.normalVector \leftarrow input.normalVector$

// Output data needed by the illumination algorithm
$output.eyeVector \leftarrow input.position * matrixWV$
$output.normalWorldVector \leftarrow input.normal * matrixW$

No Algoritmo 11, a posição final de cada vértice é gerada pela Equação 6.6. A posição final do vértice é transformada pelas matrizes de mundo, visão e projeção e enviada para saída. Em seguida, a posição de entrada do vértice, assim como seu vetor normal são enviados para a saída. Por último, o vetor de visão é calculado multiplicando a posição inicial do vértice pelas matrizes de mundo e visão, o vetor normal de entrada é transformado pela matriz de mundo, e ambos são enviados para a saída.

É importante notar que todos os dados dos vértices enviados para a saída são utilizados no estágio de rasterização do *pipeline* de renderização [Akenine-Möller et al., 2008]. No estágio de rasterização os dados dos vértices são interpolados linearmente e enviados ao estágio de processamento de *pixels*, onde ocorre a segunda parte da renderização dos pelos (a qual é apresentada na Seção 6.3).

6.5 Sumário

Este capítulo apresentou o ambiente virtual criado para visualização, edição e renderização das malhas arbitrárias contendo pelos em tempo real. Este capítulo também apresentou os algoritmos utilizados na extrusão e geração de camadas paralelas da malha, amostragem dos mapas de ambiente dos pelos e de coloração dos pelos, e cálculo da iluminação e sombreamento dos pelos. Os resultados visuais e o desempenho obtido na renderização dos pelos são apresentados no próximo capítulo.

Capítulo 7

Resultados

Este capítulo apresenta os resultados obtidos no pré-processamento das malhas arbitrárias, na geração do mapa de ambiente dos pelos, e finalmente, na aplicação e renderização dos pelos.

Os testes realizados neste trabalho foram feitos utilizando malhas extraídas de quatro diferentes objetos reais, obtidos a partir do repositório de *scanning* da universidade de Stanford. Para cada uma das quatro malhas utilizadas, foram geradas três novas versões simplificadas de suas malhas originais utilizando o *software* MeshLab [Lab, 2009]. Em alguns casos, devido ao grande volume de dados da malha original a mesma não pode ser avaliada, e nesses casos apenas suas versões simplificadas foram avaliadas. Além das malhas reais utilizadas, os testes também foram realizados sobre a malha de um cubo e de um toro. Desta maneira, neste trabalho avaliamos 18 diferentes malhas, sendo 4 extraídas de objetos reais, 8 níveis de detalhes das malhas dos objetos reais e duas malhas de sólidos.

O computador utilizado para os testes foi um *laptop*, contendo um processador de dois núcleos com frequência de 2.1 Ghz e 3MB de cache L2, 3GB memória RAM e um processador gráfico GeForce 8600GT TurboCache com 256MB de memória RAM. O sistema operacional utilizado durante os testes foi o *Windows Vista*.

A Tabela 7.1 apresenta uma listagem das malhas avaliadas contendo: o nome da malha, o seu nível de detalhe, o raio da menor esfera que envolve a malha (em unidades adimensionais), o número de vértices, o número de faces e o tamanho dos dados da malha em *kilobytes*. Note que como os dados da malha não estão previamente indexados o número de vértices é igual a três vezes o número de faces.

A partir da Tabela 7.1 é possível observar que o primeiro nível de detalhe da malha dos objetos *Dragon*, *Armadillo* e *Buddha* possuem uma alta complexidade geométrica, possuindo mais de um milhão de vértices cada.

Tabela 7.1. Lista das malhas avaliadas para aplicação e renderização dos pelos.

Nome	Nível de Detalhe	Raio	Faces	Vértices	Tamanho (KBytes)
Cubo	0	8.66	12	36	0.844
Torus	0	3.88	384	1152	27
Bunny	0	0.125	69,451	208,353	4,883.273
Bunny	1	0.124	16,301	48,903	1,146.164
Bunny	2	0.124	3,851	11,553	270.773
Bunny	3	0.120	948	2,844	66.656
Dragon	0	0.133	871,414	2,614,242	61,271.297
Dragon	1	0.133	202,520	607,560	14,239.00
Dragon	2	0.132	47,794	143,382	3,360.516
Dragon	3	0.132	11,102	33,306	780.609
Armadillo	0	114.401	345,944	1,037,832	24,324.188
Armadillo	1	114.382	83,026	249,078	5,837.766
Armadillo	2	114.402	19,926	59,778	1,401.047
Armadillo	3	114.259	4,782	14,346	336.234
Buddha	0	0.114	1,087,716	3,263,148	76,480.031
Buddha	1	0.114	293,232	879,696	20,617.875
Buddha	2	0.114	67,240	201,720	4,727.813
Buddha	3	0.114	15,536	46,608	1,092.375

7.1 Pré-Processamento da Malha

O pré-processamento da malha é dividido em 4 diferentes etapas: centralização da posição dos vértices no mundo, geração de normais contínuas, detecção de dados duplicados e indexação, e reordenação de índices. Além disso, antes que a malha possa ser processada a mesma precisa ser carregada para a memória da aplicação. A Tabela 7.2 apresenta o tempo médio gasto em segundos para a carga dos dados da malha, e a execução dos dois primeiros estágios de pré-processamento, e a Tabela 7.3 apresenta o tempo gasto em segundos para a execução dos dois últimos estágios de pré-processamento. Note que a precisão máxima do contador de tempo utilizado é de milisegundos, desta maneira o tempo é apresentado em segundos contendo três casas decimais de precisão.

A partir dos dados das Tabelas 7.2 e 7.3 é possível observar que os estágios de pré-processamento que demandam maior tempo para execução são os estágios de geração de normais contínuas e de reordenação de índices para cache utilizando o algoritmo proposto por Lin [Lin e Yu, 2006].

O algoritmo de reordenação de índices proposto por Lin possui ordem de complexidade quadrática, fazendo com que a reordenação de índices seja o estágio de pré-processamento com maior tempo de execução. Note que devido ao alto tempo necessário para a execução do algoritmo de Lin, o mesmo não pode ser executado para o primeiro nível de detalhe dos objetos *Dragon* e *Buddha*. Em ambos os casos

Tabela 7.2. Tempo gasto na carga da malha, e na execução dos dois primeiros estágios do pré-processamento da malha.

Nome	Nível de Detalhe	Carga Malha	Centralização Vert.	Geração Normais
Cubo	0	0.000	0.000	0.000
Torus	0	0.002	0.000	0.025
Bunny	0	0.341	0.002	2.727
Bunny	1	0.080	0.001	1.252
Bunny	2	0.019	0.000	0.540
Bunny	3	0.005	0.000	0.130
Dragon	0	4.544	0.037	86.396
Dragon	1	0.969	0.007	13.350
Dragon	2	0.227	0.001	7.345
Dragon	3	0.052	0.000	0.444
Armadillo	0	1.777	0.013	26.328
Armadillo	1	0.411	0.002	8.925
Armadillo	2	0.099	0.000	1.822
Armadillo	3	0.024	0.000	2.357
Buddha	0	5.657	0.041	91.336
Buddha	1	1.435	0.011	20.096
Buddha	2	0.315	0.002	9.860
Buddha	3	0.072	0.000	2.519

a reordenação de índices não foi completada após 14 horas de execução do algoritmo. O algoritmo de reordenação de índices do *DirectX extensions* apresenta um tempo de execução muito inferior ao algoritmo de Lin, no entanto, em alguns casos a reordenação de índices do D3DX aparenta entrar em um ciclo infinito, não retornando após várias horas de execução. Os casos em que a execução do D3DX não retornou estão marcados como "ERRO"na Tabela 7.3.

7.1.1 Detecção de Dados Duplicados e Indexação

A Tabela 7.4 apresenta o número de vértices e tamanho (em *kilobytes*) de cada uma das malhas avaliadas antes e após a execução do algoritmo de indexação.

Na Tabela 7.4 pode-se observar que a indexação da malha conseguiu no melhor caso reduzir os dados da malha original em 81.8%, sendo que a média de redução de dados nas malhas avaliadas foi de 73%. Desta maneira, a indexação dos dados apresenta um papel importante na redução dos dados, além de tornar possível a reordenação dos índices para maximizar o uso do *cache* dos processadores gráficos. Note que o tamanho da malha indexada é igual a soma do tamanho dos seus vértices e índices, enquanto na malha original o tamanho da malha é igual ao tamanho dos seus vértices. Nos casos em que o número de vértices da malha indexada é inferior a 65,536 vértices os índices

Tabela 7.3. Tempo gasto na execução dos dois últimos estágios de pré-processamento da malha. Dois algoritmos de reordenação de índices são avaliados, o disponibilizado pelo *DirectX Extensions* e o proposto por Lin [Lin e Yu, 2006].

Nome	Nível de Detalhe	Indexação	Reordenação D3DX	Reordenação Lin
Cubo	0	0.000	0.001	0.001
Torus	0	0.003	0.002	0.033
Bunny	0	2.743	ERRO	323.148
Bunny	1	0.614	0.025	19.239
Bunny	2	0.151	0.009	1.197
Bunny	3	0.002	0.004	0.092
Dragon	0	38.501	1.829	***
Dragon	1	9.740	0.346	2,451.987
Dragon	2	1.908	0.069	136.997
Dragon	3	0.470	0.017	8.341
Armadillo	0	17.857	1.090	7,770.470
Armadillo	1	3.776	ERRO	489.817
Armadillo	2	0.636	0.030	28.766
Armadillo	3	0.025	0.009	1.694
Buddha	0	49.789	2.304	***
Buddha	1	14.360	0.524	5,132.605
Buddha	2	2.732	ERRO	303.066
Buddha	3	0.857	0.026	16.840

gerados são armazenados como inteiros de dois *bytes*, e quando o número de vértices é igual ou superior a 65,536 os índices são armazenados como inteiros de 4 *bytes*. Desta maneira é possível observar que modelos em que a malha indexada possui mais de 65,536 vértices apresentam uma menor redução nos dados.

Sempre que uma malha é renderizada, os dados da mesma precisam ser enviados da memória principal do sistema para a memória do processador gráfico. Desta maneira, nos casos em que a transferência dos dados da malha for o gargalo na execução do *pipeline* de renderização, o que ocorre na renderização de malhas com grande volume de dados, apenas a indexação dos dados da malha pode ser suficiente para aumentar o desempenho da renderização. A Tabela 7.5 apresenta uma comparação entre o desempenho na renderização da malha original, e da malha com os dados indexados.

7.1.2 Reordenação de Índices

A Tabela 7.5 apresenta o desempenho obtido na renderização das malhas avaliadas, onde foi comparado o desempenho da malha original, da malha indexada, e da malha com os índices reordenados pelos algoritmos do D3DX e Lin. Em todos os testes realizados as etapas de centralização dos vértices da malha, e geração de nor-

Tabela 7.4. Detecção de dados duplicados e indexação dos dados das malhas.

		Malha Original		Malha Indexada		
Nome	N. Detalhe	Vértices	Tamanho (KB)	Vértices	Tamanho (KB)	Redução
Cubo	0	36	0.844	8	0.258	69.4%
Torus	0	1152	27	192	6.750	75%
Bunny	0	208,353	4,883.273	34,834	1,223.361	74.9%
Bunny	1	48,903	1,146.164	8,146	208.436	81.8%
Bunny	2	11,553	270.773	1,887	66.791	75.3%
Bunny	3	2,844	66.656	453	16.172	75.7%
Dragon	0	2,614,242	61,271.297	434,856	20,403.820	66.7%
Dragon	1	607,560	14,239.00	100,207	4,721.883	66.8%
Dragon	2	143,382	3,360.516	22,982	818.684	75.6%
Dragon	3	33,306	780.609	5,203	186.996	76%
Armadillo	0	1,037,832	24,324.188	172,974	8,108.109	66.7%
Armadillo	1	249,078	5,837.766	41,515	1,459.488	74.9%
Armadillo	2	59,778	1,401.047	9,965	350.309	74.9%
Armadillo	3	14,346	336.234	2,393	84.105	74.9%
Buddha	0	3,263,148	76,480.031	542,612	25,464.141	66.7%
Buddha	1	879,696	20,617.875	144,628	6,826.031	66.8%
Buddha	2	201,720	4,727.813	32,316	1,151.391	75.6%
Buddha	3	46,608	1,092.375	7,102	257.484	76.4%

mais contínuas foram executados. Note que o desempenho medido é relativo a renderização de uma instância da malha utilizando o modelo de iluminação de Phong [Akenine-Möller et al., 2008]. O desempenho na renderização das malhas é medido em quadros por segundo, onde um maior número de quadros por segundo representa um maior desempenho. Por último, a imagem final gerada na renderização possui a resolução de 1280x800.

O desempenho apresentado na Tabela 7.5 pode ser avaliado melhor levando em conta as malhas com maior complexidade geométrica, onde é possível observar que o uso de índices contriu para aumentar o desempenho, assim como a reordenação dos índices. No entanto, na renderização de malhas com menor complexidade a indexação em alguns casos reduz o desempenho da renderização e a reordenação de índices não apresenta nenhum efeito sobre o desempenho final. Uma comparação mais precisa sobre o desempenho requer que um maior número de instâncias de cada malha seja renderizada, o que ocorre na renderização dos pelos.

Comparando os algoritmos de reordenação de índices, é possivel perceber que de maneira geral o algoritmo do D3DX apresenta um ganho de desempenho superior ao algoritmo de Lin, onde em alguns casos o desempenho do D3DX é superior a 2.4 vezes o desempenho do algoritmo de Lin. Devido ao algoritmo do D3DX apresentar problema no processamento de algumas malhas, e devido a não ter sido possivel aplicar

Tabela 7.5. Comparação de desempenho na renderização das malhas utilizando o modelo de iluminação de Phong [Akenine-Möller et al., 2008].

Nome	N. Detalhe	Original	Indexada	Reord. D3DX	Reord. Lin
Cubo	0	596	593	585	584
Torus	0	594	594	585	589
Bunny	0	400	376	ERROR	541
Bunny	1	555	543	556	557
Bunny	2	561	563	565	566
Bunny	3	572	570	574	573
Dragon	0	84	156	161	***
Dragon	1	242	330	393	355
Dragon	2	453	486	572	567
Dragon	3	576	574	566	572
Armadillo	0	170	61	304	125
Armadillo	1	381	298	ERRO	518
Armadillo	2	555	514	560	558
Armadillo	3	568	566	572	570
Buddha	0	69	135	135	***
Buddha	1	196	274	329	256
Buddha	2	410	450	ERRO	552
Buddha	3	559	557	564	570

o algoritmo de Lin no primeiro nível de detalhe da malha dos objetos *Dragon* e *Buddah*, não foi possível fazer uma comparação mais completa entre os algoritmos.

7.2 Renderização dos pelos

Esta seção apresenta os resultados relacionados a aplicação e renderização dos pelos.

7.2.1 Amostragem do Mapa de Ambiente

Para que a amostragem do mapa de ambiente produza bons resultados, o endereçamento do mapa de ambiente deve seguir os requisitos apresentados na Seção 6.3.1.2, os quais são listados a seguir:

1. O endereço acesso ao mapa de ambiente deve ser constante para cada posição da malha.

2. Dois vértices pertencentes a uma mesma face da malha não devem amostrar a mesma posição do mapa de ambiente.

3. O número de diferente amostragens realizadas sobre o mapa de ambiente por cada face da malha deve ser idealmente proporcional ao tamanho da face.

Para atender aos requisitos desejados na amostragem do mapa de ambiente, foram utilizadas três abordagens diferentes. Na primeira abordagem, o mapa de ambiente foi endereçado utilizando o vetor normal de entrada dos vértices (sem a aplicação de transformações). O vetor normal de entrada dos objetos é contínuo entre faces adjacentes, mas dois vértices pertencentes a uma mesma face podem conter o mesmo vetor normal. Espera-se uma boa distribuição dos vetores normais, devido a malha ser um sólido possuindo normais em direções arbitrárias. Um problema dessa abordagem é que o tamanho das faces não possui nenhuma relação com as normais utilizadas na amostragem, desta maneira o número de diferentes amostras realizadas não é proporcional ao tamanho da face. A Figura 7.1 apresenta o resultado da amostragem do mapa de ambiente a partir do vetor normal para a malha dos objetos *Cubo* e *Torus*, e para a malha do segundo nível de detalhe da malha dos objetos *Bunny*, *Dragon*, *Armadillo* e *Buddah*.

Na segunda abordagem, o mapa de ambiente foi endereçado utilizando um vetor unitário contendo a direção de cada ponto sobre a malha (sem a aplicação de transformações). As posições sobre a superfície da malha são contínuas, e devido a malha ser centralizada no centro do mundo cada face da malha geralmente possui três diferentes vetores unitários de posição. Malhas com alto nível de detalhe apresentam um grande número de pequenas faces, onde a diferença entre a posição das faces é pequena, fazendo com que apenas uma pequena parte do mapa de ambiente seja amostrado. Malhas com baixo nível de detalhe apresetam faces maiores, onde a diferença entre a posição das faces é maior, fazendo com que uma maior parte do mapa de ambiente seja amostrado. A Figura 7.2 apresenta o resultado da amostragem do mapa de ambiente a partir do vetor unitário com a direção de cada ponto processado sobre a malha.

Na terceira abordagem avaliada, os endereçamentos utilizados na primeira e na segunda abordagem foram combinados visando criar uma melhor distribuição dos dados amostrados do mapa de ambiente. A Figura 7.3 apresenta o resultado da amostragem do mapa de ambiente a partir da combinação do vetor normal e do vetor unitário com a direção de cada ponto sobre a malha.

Nas imagens apresentadas nas Figura 7.1, 7.2 e 7.3 o pelo aplicado as malhas é renderizado com uma cor chapada, não sendo utilizada o mapa de coloração dos pelos. É possível observar que a terceira abordagem utilizada produz os melhores resultados visuais, onde os problemas presentes na primeira e na segunda abordagem são quase completamente eliminados. Note que o mapa de ambiente utilizado nestes testes foi o mapa de ambiente no formato de cubo.

A coordenada de acesso ao mapa de ambiente dos pelos é a mesma utilizada para acessar o mapa de ambiente de coloração dos pelos. Os valores amostrados no mapa

dos pelos e no mapa de coloração dos pelos são modulados produzindo o resultado final que é aplicado a cada ponto da malha. O resultado da amostragem do mapa de pelos e do mapa de coloração de pelos utilizando a terceira abordagem de acesso ao mapa de ambiente é apresentado nas Figuras 7.4, 7.5 e 7.6. As Figuras 7.4 e 7.5 apresentam o resultado para o mapa de ambiente no formato de cubo, utilizando respectivamente as resoluções de *128x128* texels e 256x256 *texels*. A Figura 7.6 apresenta o resultado o mapa de ambiente no formato de esfera utilizando a resolução de 1024x1024 *texels*.

A partir das Figuras 7.4 e 7.5 é possível observar que a espessura dos pelos é inversamente proporcional a resolução do mapa de pelos. Desta maneira, para obter-se pelos de pequena espessura é necessário o uso de mapas de pelos com alta resolução. Comparando os resultados obtidos nas Figuras 7.5 e 7.6 é possível observar que a amostragem do mapa de esfera gera pelos com espessura maior que a amostragem do mapa de cubo. Note que no mapa de cubo da primeira imagem são utilizadas seis imagens com resolução 256x256 *texels*, enquanto no mapa de esfera é utilizada uma imagem com resolução de 1024x1024 *texels*. Deste modo seria necessário uma resolução maior no mapa de esfera para conseguir o mesmo resultado obtido do mapa de cubo.

7.2.2 Resultados Finais

Os resultados finais obtidos na aplicação e renderização dos pelos são dependentes dos parâmetros utilizados no pré-processamento das malhas, na geração dos mapas de pelos e na renderização dos pelos. Para reduzir o número de combinações de resultados possíveis os parâmetros do pré-processamento da malha são fixados no melhor caso obtido, de acordo com os resultados apresentados na Seção 7.1. Desta maneira, o pré-processamento da malha utiliza o segundo algoritmo de geração de normais apresentado e o algoritmo de reordenação de índices do *DirectX extensions*, ou o algoritmo de reordenação de índices de Lin [Lin e Yu, 2006] nos casos em que algoritmo do D3DX não pode ser utilizado (apresentados na Tabela 7.3). Na geração do mapa de pelos o formato do mapa de ambiente utilizado será o formato de cubo, o qual requer maior consumo de memória mas apresenta uma melhor qualidade visual no mapeamento dos pelos (apresentando pelos de menor espessura) como pode ser visto na Seção 7.2.1. A resolução do mapa de ambiente de cubo utilizado foi de 256x256 *texels*, e o parâmetro avaliado na geração do mapa de pelos foi a densidade dos pelos no mapa. Na renderização dos pelos, os parâmetros avaliados foram: a altura dos pelos sobre a superfície da malha, e o número de camadas da malha onde os pelos foram aplicados e renderizados.

Nos testes realizados, a renderização dos pelos foi avaliada com o número de camadas variando entre 8 e 16 camadas, e com a altura dos pelos variando entre 2% e 5%

do raio da menor esfera que envolve a malha. Na geração do mapa de ambiente dos pelos, a densidade dos pelos foi avaliada variando entre 10% e 20% da área do mapa.

A Tabela 7.6 apresenta as combinações dos parâmetros avaliados, os quais foram utilizados na renderização do segundo nível de detalhe da malha dos objetos: cubo, toro, *Bunny*, *Dragon*, *Armadillo* e *Buddah*. Os resultados visuais obtidos nos testes realizados são apresentados respectivamente nas Figuras 7.7, 7.8, 7.9, 7.10, 7.11, 7.12, 7.13 e 7.14. O desempenho obtido na renderização das imagens finais é apresentado na Tabela 7.7. Note que não existe diferença no desempenho de acordo com a variação da densidade do mapa de pelos, desta maneira são apresentados apenas quatro diferentes desempenho na Tabela 7.7.

Tabela 7.6. Configurações utilizadas para avaliação da qualidade visual e desempenho na renderização de pelos.

Configuração	Num. Camadas	Altura pelos	Densidade pelos
1	8	2%	10%
2	8	2%	20%
3	8	5%	10%
4	8	5%	20%
5	16	2%	10%
6	16	2%	20%
7	16	5%	10%
8	16	5%	20%

É possível observar que o número de camadas necessárias para obter uma boa qualidade visual na renderização dos pelos é dependente da altura dos pelos e da densidade dos mesmos. Quando maior a altura dos pelos, maior se torna a distância entre as camadas da malha onde os mesmos são aplicados, o que gera serrilhados e buracos entre os pelos. De uma maneira similar, quanto maior a densidade dos pelos menor a área sobre a malha em que é possivel observar as descontinuidades entre as camadas. Portanto, aumentando a densidade dos pelos é possivel esconder alguns defeitos encontrados nos pelos, gerando imagens com maior qualidade visual.

Nas Figuras 7.7 e 7.7 é possível observar que utilizando apenas 8 camadas consegue-se representar pelos curtos com uma boa qualidade visual. As Figuras 7.13 e 7.14 demonstram que aumentando o nível de camadas da malha é possível aumentar o tamanho dos pelos mantendo uma boa qualidade visual. Nas Figuras 7.9 e 7.10 é possível perceber que apenas aumentando a densidade dos pelos na malha é possível reduzir o serrilhado na imagem final e gerar imagens com maior qualidade visual.

Na Tabela 7.7 é possível observar que a renderização de pelos apresenta desempenho interativo para malhas com mais de 200,000 triângulos. No pior caso avaliado, a malha do segundo nível de detalhe do objeto *Buddha* contendo 293,000 faces foi renderizada

Tabela 7.7. Desempenho final obtido na renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddha* para as configurações apresentadas na Tabela 7.6. Note que não existe diferença de desempenho na variação da densidade do mapa de pelos.

Configurações	Nome	Reordenação Índices	Quadros/Segundo
1 e 2	Cubo	D3DX	205
1 e 2	Torus	D3DX	189
1 e 2	Bunny	D3DX	133
1 e 2	Dragon	D3DX	43
1 e 2	Armadillo	Lin	90
1 e 2	Buddha	D3DX	37
3 e 4	Cubo	D3DX	181
3 e 4	Torus	D3DX	178
3 e 4	Bunny	D3DX	126
3 e 4	Dragon	D3DX	42
3 e 4	Armadillo	Lin	76
3 e 4	Buddha	D3DX	37
5 e 6	Cubo	D3DX	128
5 e 6	Torus	D3DX	115
5 e 6	Bunny	D3DX	78
5 e 6	Dragon	D3DX	23
5 e 6	Armadillo	Lin	50
5 e 6	Buddha	D3DX	20
7 e 8	Cubo	D3DX	108
7 e 8	Torus	D3DX	110
7 e 8	Bunny	D3DX	74
7 e 8	Dragon	D3DX	23
7 e 8	Armadillo	Lin	47
7 e 8	Buddha	D3DX	20

como 16 camadas contendo pelos, sobre a renderização de mais uma camada base. Nesse teste o desempenho final obtido foi de 20 quadros por segundo, o que pode ser considerado um desempenho interativo.

Figura 7.1. Resultado da amostragem do mapa de ambiente utilizando o vetor normal de cada ponto da superfície da malha dos objetos.

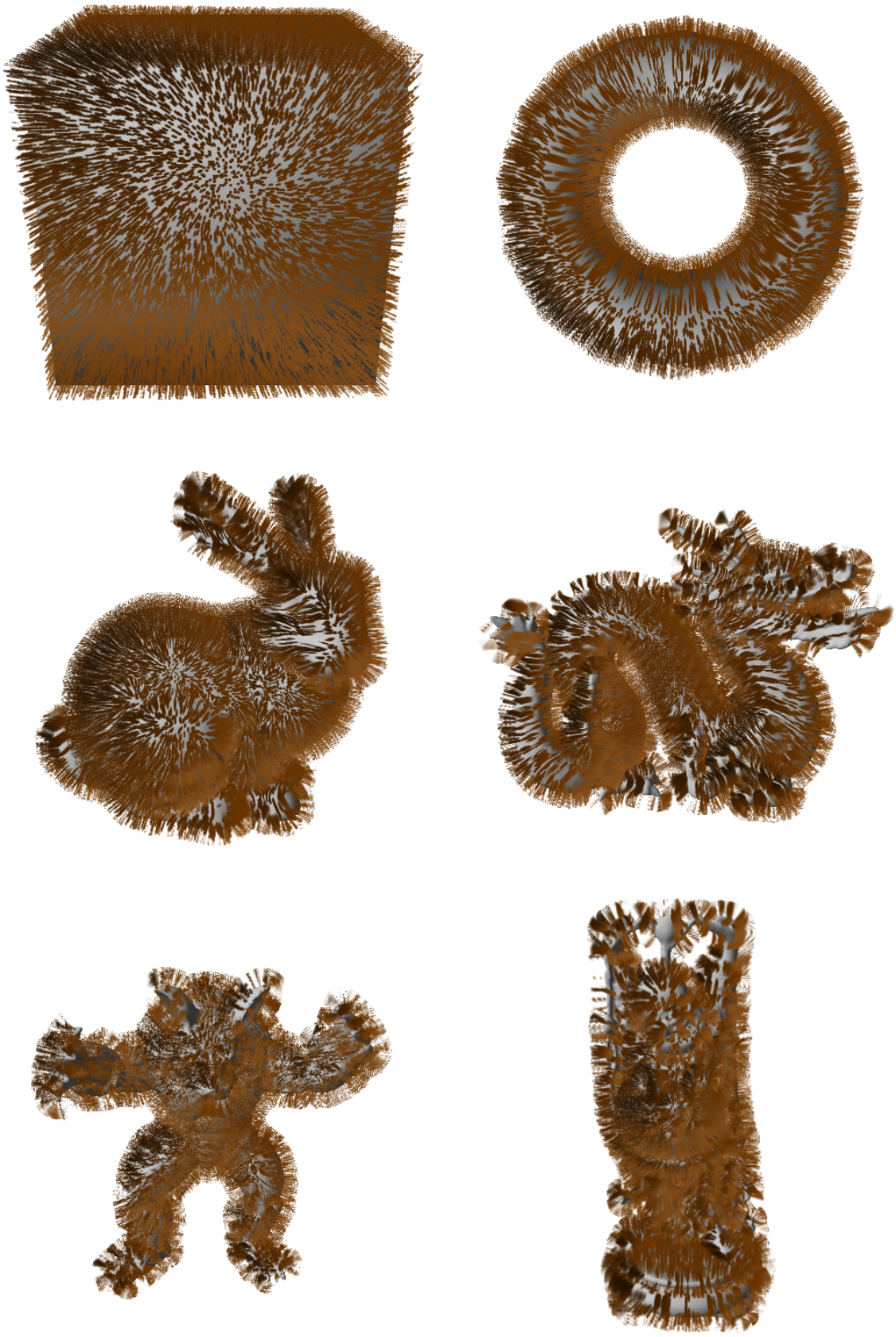


Figura 7.2. Resultado da amostragem do mapa de ambiente utilizando o vetor unitário de direção de cada cada ponto da superfície da malha dos objetos.

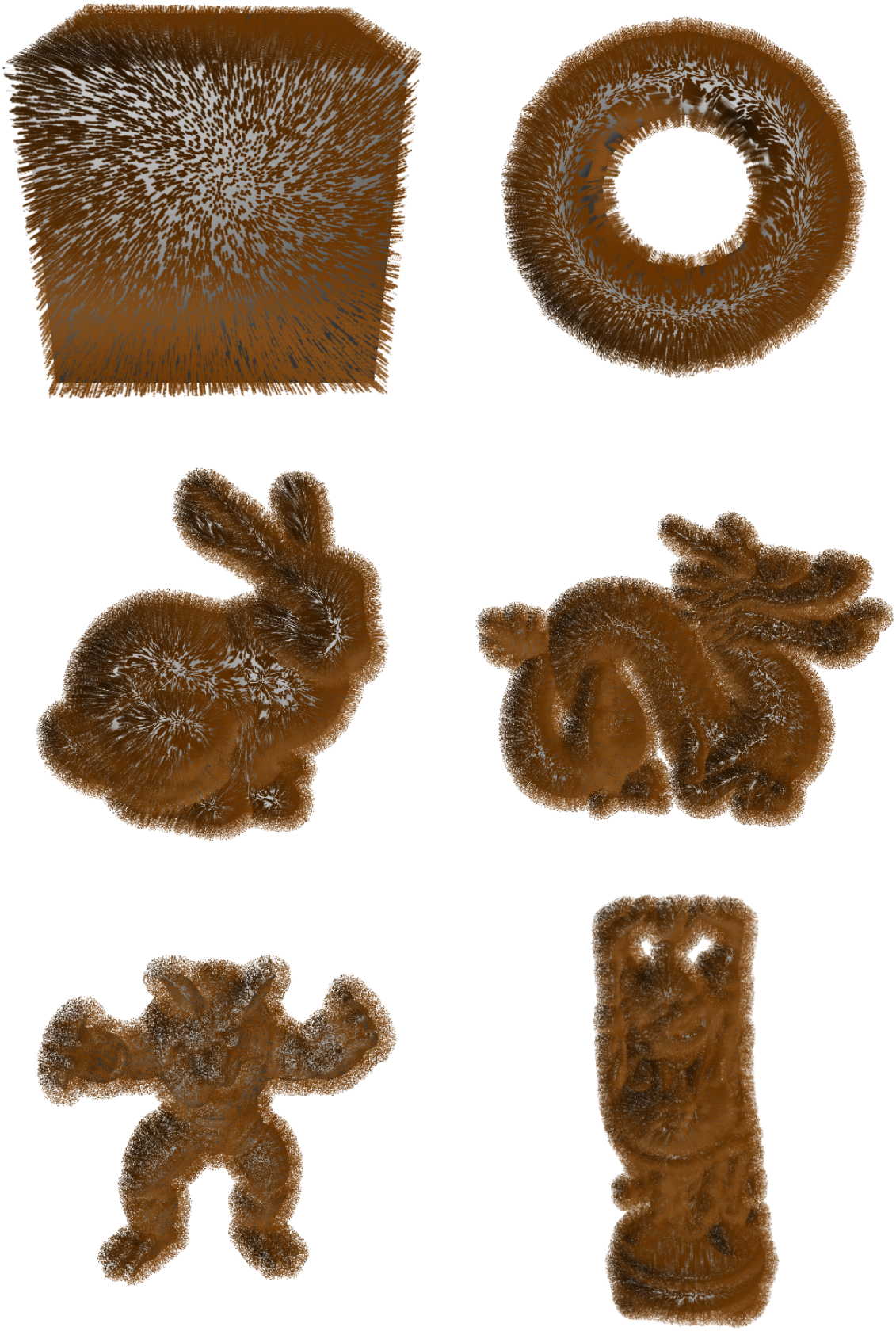


Figura 7.3. Resultado da amostragem do mapa de ambiente combinando o vetor normal, com o vetor de direção de cada ponto da superfície da malha dos objetos.

Figura 7.4. Resultado final da amostragem do mapa de ambiente dos pelos e coloração dos pelos no formato de cubo com resolução de 128x128 *texels*.

Figura 7.5. Resultado final da amostragem do mapa de ambiente dos pelos e coloração dos pelos no formato de cubo com resolução de 256x256 *texels*.

Figura 7.6. Resultado final da amostragem do mapa de ambiente dos pelos e coloração dos pelos no formato de esfera com resolução de 1024x1024 *texels*.

Figura 7.7. Resultados visuais obtidos a partir da renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddah*, utilizando a configuração 1 apresentada na Tabela 7.6.

Figura 7.8. Resultados visuais obtidos a partir da renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddah*, utilizando a configuração 2 apresentada na Tabela 7.6.

Figura 7.9. Resultados visuais obtidos a partir da renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddah*, utilizando a configuração 3 apresentada na Tabela 7.6.

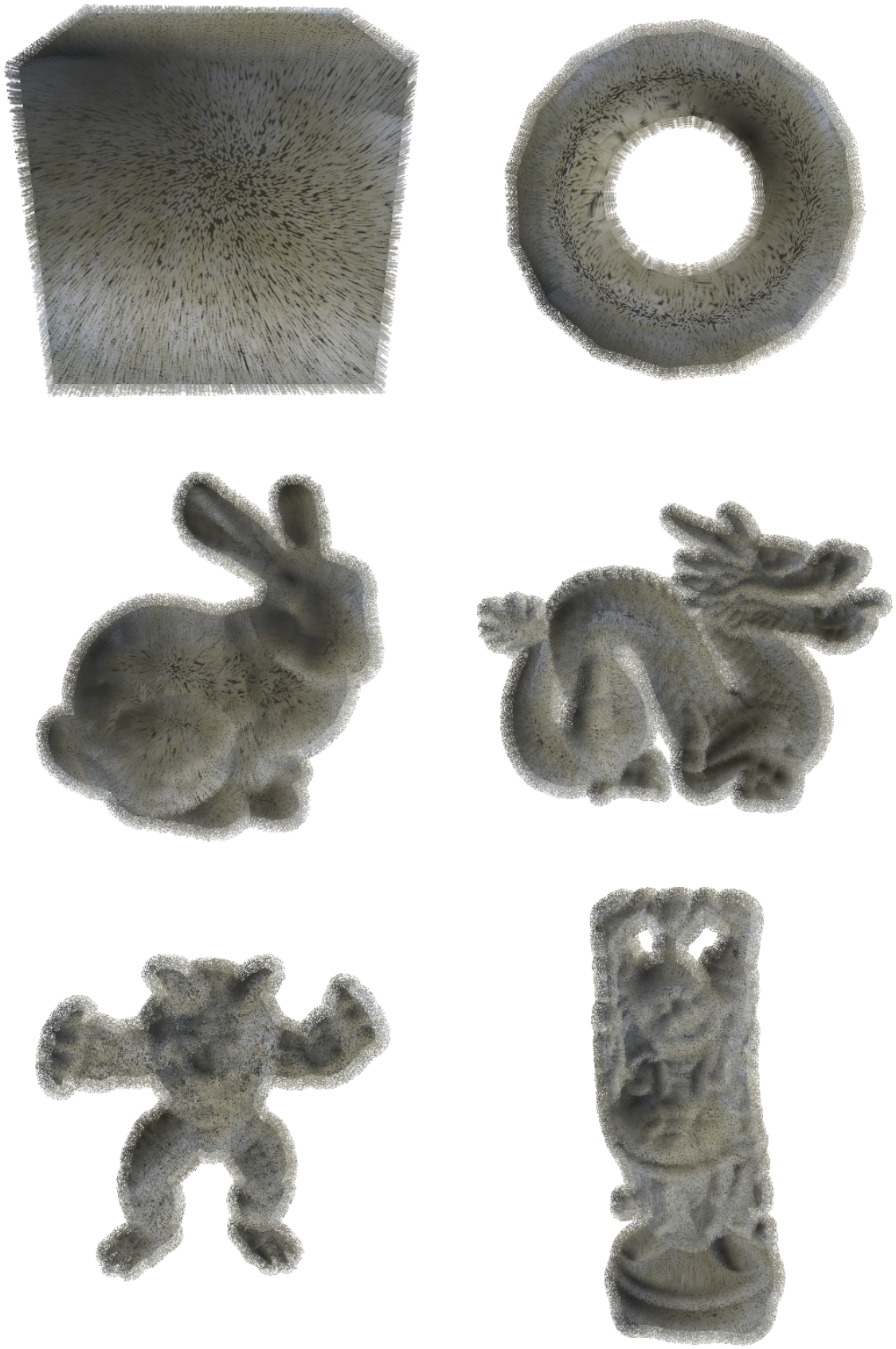


Figura 7.10. Resultados visuais obtidos a partir da renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddah*, utilizando a configuração 4 apresentada na Tabela 7.6.

Figura 7.11. Resultados visuais obtidos a partir da renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddah*, utilizando a configuração 5 apresentada na Tabela 7.6.

Figura 7.12. Resultados visuais obtidos a partir da renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddah*, utilizando a configuração 6 apresentada na Tabela 7.6.

Figura 7.13. Resultados visuais obtidos a partir da renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddah*, utilizando a configuração 7 apresentada na Tabela 7.6.

Figura 7.14. Resultados visuais obtidos a partir da renderização dos objetos Cubo, Torus, *Bunny*, *Dragon*, *Armadillo* e *Buddah*, utilizando a configuração 8 apresentada na Tabela 7.6.

Capítulo 8

Conclusao

Neste trabalho foi apresentado um algoritmo para criação e aplicação automática de pelos a malhas arbitrárias, e renderização dessas malhas em tempo real. O algoritmo proposto se mostrou robusto e capaz de adicionar pelos a malhas arbitrárias extraídas de objetos reais com alta ou baixa complexidade geométrica.

Na primeira etapa do algoritmo proposto, a etapa de pré-processamento da malha, foram apresentados algoritmos para a geração e modificação de dados em malhas arbitrárias, tornando as mesmas aptas a renderização de pelos. Nesta mesma etapa, também foram apresentados algoritmos para otimização dos dados da malha, os quais permitiram a redução dos dados das malhas em até 81%, e aumento no desempenho da sua renderização de até 95%. Na segunda etapa do algoritmo proposto, a etapa de geração de pelos, foram apresentados algoritmos para a geração de pelos estreitos, e o armazenamento desses pelos como mapas de ambiente no formato de cubo e esfera. Na última etapa do algoritmo proposto, a etapa de renderização dos pelos, foi apresentado o ambiente criado para navegação, visualização e edição das malhas com pelos, o algoritmo para geração das camadas paralelas da malha (onde os pelos são aplicados), e os algoritmos para mapeamento dos pelos sobre a malha, modulação dos pelos com um mapa de coloração e iluminação dos pelos. Por fim, a qualidade visual e o desempenho do algoritmo proposto foram avaliados utilizando diferentes cenários, e os resultados obtidos mostraram que o método proposto é capaz de gerar imagens com boa qualidade visual em tempo real.

O algoritmo proposto neste trabalho apresenta uma alternativa ao algoritmo proposto por Lengyel [Lengyel e Praun, 2001], onde o algoritmo proposto permite a aplicação de pelos de maneira automática sem a necessidade de intervenção humana em nenhuma das etapas. Além disso, a renderização de cada camada da malha como uma unidade atómica e as otimizações aplicadas sobre a mesma, apresentam uma abordagem mais eficiente para renderização das camadas dos pelos, comparado a abordagem

proposta por Lengyel.

8.1 Trabalhos Futuros

O algoritmo apresentado neste trabalho trata da aplicação de pelos estreitos sobre malhas arbitrárias, sendo que tipos de pelos mais complexos como pelos ondulados ou encaracolados não são abordados. Na geração dos mapas de pelos tambem é considerado que os pelos possuem altura uniforme, uma assertiva que pode ser verdadeira, mas que restringe os tipos de pelos que podem ser representados. Por último, a aplicação dos pelos é feita uniformemente sobre toda a malha de entrada, não sendo possível definir partes da malha onde não existem pelos. Desta maneira, o algoritmo de geração e aplicação de pelos poderia ser extendido das seguintes maneiras:

1. Suporte a tipos de pelos não estreitos.
   Para suportar esse recurso seria necessário modificar a geração de pelos para que a mesma funcionasse baseada em um sistema de partículas, onde seria gerado e armazenado um diferente mapa de posições e normais para cada camada dos pelos. Desta maneira, os pelos seriam armazenados em dois volumes, um contendo camadas de posições e outro contendo camadas de normais.

2. Suporte a altura não uniforme dos pelos.
   Para suportar esse recurso seria necessário armazenar todos os tipos de pelos, incluindo pelos estreitos, em um volume de dados. Outra solução seria pintar em cada vértice da malha o fator de extrusão utilizado naquele vértice, gerando assim, partes da malha com diferentes níveis de extrusão (e diferentes alturas).

3. Remoção de pelos em partes da malha.
   Para suportar esse recurso sobre malhas arbitrárias seria necessário pintar nos vértices da malha os trechos de geometria onde os pelos não seriam aplicados. Desta maneira, no *shader* de *pixels* esses trechos poderiam ser descartados. Uma maneira mais precisa seria utilizar uma textura, ou mapa, sobre o modelo para demarcar no nível de *pixels* os trechos onde os pelos não seriam aplicados. Mas neste caso existe o problema de mapeamento deste mapa sobre a malha.

Para melhor a qualidade visual dos resultados também seria necessário investigar o uso de técnicas de *antialiasing* na renderização dos pelos, para reduzir a presença de serrilhados que ocorre na renderização de camadas discretas. É importante citar que devido as camadas de pelos serem desenhadas utilizando *alpha blending* (mistura

baseada no canal de cor *alpha*) alguns algoritmos de *antialiasing* não seriam capazes de apresentar resultado correto.

Apêndice A

Código do *Shader* de pelos

A listagem abaixo apresenta o código completo do shader utilizado na processamento de vértices e *pixels* das malhas, o qual foi escrito utilizando a linguagem de *shaders* HLSL.

```
RasterizerState LayerRaster
{
CullMode = None;
FillMode = Solid;
FrontCounterClockWise = true;
};

DepthStencilState EnableDepthLess
{
    DepthEnable = TRUE;
    DepthWriteMask = ALL;
    DepthFunc = LESS_EQUAL;
};

BlendState SrcAlphaBlending
{
AlphaToCoverageEnable = FALSE;
BlendEnable[0] = TRUE;
SrcBlendAlpha = SRC_ALPHA;
DestBlendAlpha = INV_SRC_ALPHA;
SrcBlend = SRC_ALPHA;
DestBlend = INV_SRC_ALPHA;
```

```
BlendOpAlpha = ADD;
};


TextureCube environmentCubeFurTexture;
Texture2D environmentSphereFurTexture;
TextureCube environmentColorCubeFurTexture;
Texture2D environmentColorSphereFurTexture;

SamplerState linearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Clamp;
    AddressV = Clamp;
};

cbuffer cbChangesEveryFrame
{
    matrix matWorld;
    matrix matView;
matrix matViewInverse;
    matrix matProjection;

matrix matWorldView;
matrix matWorldViewProjection;
};

struct LAYER_VS_INPUT
{
    float3 position     : POSITION0;
    float3 normal       : NORMAL0;
};

struct LAYER_PS_INPUT
{
    float4 hposition   : SV_POSITION;
float3 position    : TEXCOORD0;
```

```
    float3 normal      : TEXCOORD1;
float3 normalWorld : TEXCOORD2;
float3 eyeVector   : TEXCOORD3;
};


LAYER_PS_INPUT LAYER_VS( LAYER_VS_INPUT input)
{
LAYER_PS_INPUT output = (LAYER_PS_INPUT)0;
float4 position = float4(input.position, 1.0f);
float3 positionDisplacement = input.normal * LayeredLayerHeight *
LayeredCurrentLayer;
position.xyz += positionDisplacement;

output.eyeVector = mul(position, matWorldView).xyz;
output.normal = input.normal;
output.normalWorld = mul(input.normal, matWorld);

output.position = input.position;
output.hposition = mul(position, matWorldViewProjection);

    return output;
}


float4 LAYER_PS( LAYER_PS_INPUT input) : SV_Target
{
float3 position = normalize(input.position);
float3 normal = normalize(input.normal);
float3 normalWorld = normalize(input.normalWorld);
float3 eyeVector = normalize(input.eyeVector);
float3 lightVector = normalize(LightDirection);

// Reflect vector
// -------------------------------------------------
//float3 reflectedView = normalize(normal);
//float3 reflectedView = normalize(position);
```

```
float3 reflectedView = normalize(normal + normalize(position));

float4 outputColor;
float3 furColor;

// Cube Mapping
// ------------------------------------------------
outputColor = environmentCubeFurTexture.SampleLevel(linearSampler,
reflectedView, 0);
furColor = environmentColorCubeFurTexture.SampleLevel(linearSampler,
reflectedView, 0);
outputColor.rgb *= furColor;

// Sphere Mapping
// ------------------------------------------------
if (false)
{
// MODE1
float2 texCoord = float2(reflectedView.x * 0.5f + 0.5f,
reflectedView.y * 0.5f + 0.5f);
outputColor = environmentSphereFurTexture.SampleLevel(
linearSampler, texCoord, 0);

// MODE2
//// Calculate the sphere map texture coordinate
//float2 texCoord = float2(0,0);
//float3 absoluteReflectedView = abs(reflectedView);
//if (absoluteReflectedView.z >= absoluteReflectedView.x &&
//  absoluteReflectedView.z >= absoluteReflectedView.y)
//{
// if (reflectedView.z > 0)
// reflectedView.z += 1;
// else
// reflectedView.z -= 1;
//
// float invDoubleLength = 1.0f / (2 * length(reflectedView));
// texCoord = float2(reflectedView.x * invDoubleLength + 0.5f,
```

```
// reflectedView.y * invDoubleLength + 0.5f);
//}
//else if (absoluteReflectedView.x >= absoluteReflectedView.y &&
// absoluteReflectedView.x >= absoluteReflectedView.z)
//{
// if (reflectedView.x > 0)
// reflectedView.x += 1;
// else
// reflectedView.x -= 1;
//
// float invDoubleLength = 1.0f / (2 * length(reflectedView));
// texCoord = float2(reflectedView.z * invDoubleLength + 0.5f,
// reflectedView.y * invDoubleLength + 0.5f);
//}
//else if (absoluteReflectedView.y >= absoluteReflectedView.x &&
// absoluteReflectedView.y >= absoluteReflectedView.z)
//{
// if (reflectedView.y > 0)
// reflectedView.y += 1;
// else
// reflectedView.y -= 1;
//
// float invDoubleLength = 1.0f / (2 * length(reflectedView));
// texCoord = float2(reflectedView.x * invDoubleLength + 0.5f,
// reflectedView.z * invDoubleLength + 0.5f);
//}
//
//outputColor = environmentSphereFurTexture.SampleLevel(linearSampler,
// texCoord, 0);
//furColor = environmentSphereFurTexture.SampleLevel(linearSampler,
// texCoord, 0);
//outputColor.rgb *= furColor;
}

// Kajiya & Kay Lighting
// -------------------------------------------------
float3 tangentVector = normalWorld;
```

```
float tangentDotLight = dot(tangentVector, lightVector);
float tangentDotEye = dot(tangentVector, eyeVector);
float sinTL = sqrt(1 - tangentDotLight*tangentDotLight);
float sinTE = sqrt(1 - tangentDotEye*tangentDotEye);
float diffuseLightIntensity = sinTL;
float specularLightIntensity = abs(tangentDotLight * tangentDotEye +
sinTL * sinTE);

outputColor.rgb *= ambientMaterial + diffuseMaterial * diffuseLightIntensity;
outputColor.rgb += specularMaterial * pow(specularLightIntensity,
specularMaterialPower);

    return outputColor;
}


technique10 LayerFur
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, LAYER_VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, LAYER_PS() ) );

        SetRasterizerState( LayerRaster );
        SetDepthStencilState( EnableDepthLess, 0 );
        SetBlendState( SrcAlphaBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ),
         0xFFFFFFF );
    }
}
```

Referências Bibliográficas

[Akenine-Möller et al., 2008] Akenine-Möller, T.; Haines, E. e Hoffman, N. (2008). *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

[Arkin et al., 1994] Arkin, E. M.; Held, M.; Mitchell, J. S. B. e Skiena, S. (1994). Hamilton triangulations for fast rendering. In *ESA '94: Proceedings of the Second Annual European Symposium on Algorithms*, pp. 36–47, London, UK. Springer-Verlag.

[Beaudoin e Poulin, 2004] Beaudoin, P. e Poulin, P. (2004). Compressed multisampling for efficient hardware edge antialiasing. In *Graphics Interface 2004*, pp. 169–176.

[Bogomjakov e Gotsman, 2001] Bogomjakov, A. e Gotsman, C. (2001). Universal rendering sequences for transparent vertex caching of progressive meshes. In *GRIN'01: No description on Graphics interface 2001*, pp. 81–90, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

[Cani e Bertails, 2006] Cani, M.-P. e Bertails, F. (2006). Hair interactions. In *Eurographics 2006, Tutorial session Collision Handling and its applications, September, 2006*, Vienna, Autriche. The Eurographics association.

[Codeweavers, 2009] Codeweavers (2009). Crossover. `http://www.codeweavers.com`. Disponível em Janeiro 2009.

[Cook e Torrance, 1982] Cook, R. L. e Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24.

[Curless e Levoy, 1996] Curless, B. e Levoy, M. (1996). A volumetric method for building complex models from range images. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 303–312, New York, NY, USA. ACM.

[Curless e Levoy, 2007] Curless, B. e Levoy, M. (2007). Vrippack volumetric range image processing package. `http://www-graphics.stanford.edu/software/vrip/`. Disponível em Janeiro 2009.

[Debevec, 2009] Debevec, P. (2009). Creating a light probe. `http://gl.ict.usc.edu/HDRShop/tutorial/tutorial5.html#part2`. Disponível em Janeiro 2009.

[Decaudin e Neyret, 2004] Decaudin, P. e Neyret, F. (2004). Rendering forest scenes in real-time. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)*, pp. 93–102.

[Garland e Heckbert, 1997] Garland, M. e Heckbert, P. S. (1997). Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Hirche et al., 2004] Hirche, J.; Ehlert, A.; Guthe, S. e Doggett, M. (2004). Hardware accelerated per-pixel displacement mapping. In *GI '04: Proceedings of Graphics Interface 2004*, pp. 153–158, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada. Canadian Human-Computer Communications Society.

[Hoppe, 1999] Hoppe, H. (1999). Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 269–276, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Isidoro e Mitchell, 2002] Isidoro, J. e Mitchell, J. L. (2002). User customizable real-time fur. In *SIGGRAPH '02: ACM SIGGRAPH 2002 conference abstracts and applications*, pp. 273–273, New York, NY, USA. ACM.

[Jeschke et al., 2007] Jeschke, S.; Mantler, S. e Wimmer, M. (2007). Interactive smooth and curved shell mapping.

[Kajiya e Kay, 1989] Kajiya, J. T. e Kay, T. L. (1989). Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.*, 23(3):271–280.

[Kim e Neumann, 2002] Kim, T.-Y. e Neumann, U. (2002). Interactive multiresolution hair modeling and editing. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 620–629, New York, NY, USA. ACM.

[Lab, 2009] Lab, V. C. (2009). Meshlab. `http://meshlab.sourceforge.net/`. Disponível em Janeiro 2009.

[Lacroute, 1995] Lacroute, P. (1995). *Fast volume rendering using a shear-warp factorization of the viewing transformation*. PhD thesis, Stanford University.

[Lengyel e Praun, 2001] Lengyel, J. e Praun, E. (2001). Real-time fur over arbitrary surfaces. In *In Symposium on Interactive 3D Graphics*, pp. 227–232. ACM Press.

[Lin e Yu, 2006] Lin, G. e Yu, T. P. Y. (2006). An improved vertex caching scheme for 3d mesh rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):640–648.

[Luebke et al., 2002] Luebke, D.; Watson, B.; Cohen, J. D.; Reddy, M. e Varshney, A. (2002). *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA.

[Meyer e Neyret, 1998] Meyer, A. e Neyret, F. (1998). Interactive volumetric texures. In *Eurographics Rendering Workshop*, p. 0. Eurographics Association.

[Moon et al., 2008] Moon, J. T.; Walter, B. e Marschner, S. (2008). Efficient multiple scattering in hair using spherical harmonics. *ACM Trans. Graph.*, 27(3):1–7.

[Neyret, 1995] Neyret, F. (1995). A general and multiscale model for volumetric textures. In Davis, W. A. e Prusinkiewicz, P., editores, *Graphics Interface '95*, pp. 83–91. Canadian Information Processing Society, Canadian Human-Computer Communications Society. ISBN 0-9695338-4-5.

[Neyret, 1996] Neyret, F. (1996). *Textures Volumiques pour la Synthèse d'images*. PhD thesis, Université Paris-XI - INRIA. http://www-evasion.imag.fr/Membres/Fabrice.Neyret/publis/thesefabrice-eng.html.

[Neyret, 1998] Neyret, F. (1998). Modeling animating and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70. ISSN 1077-2626.

[Nguyen e Donnelly, 2005] Nguyen, H. e Donnelly, W. (2005). *Hair Animation and Rendering in the Nalu Demo*, chapter 23. Addison-Wesley Professional.

[nVidia, 2009] nVidia (2009). nvidia developer tools. `http://www.developer.nvidia.com/page/home.html`. Disponível em Janeiro 2009.

[Porumbescu et al., 2005] Porumbescu, S. D.; Budge, B.; Feng, L. e Joy, K. I. (2005). Shell maps. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pp. 626–633, New York, NY, USA. ACM.

[Praun et al., 2000] Praun, E.; Finkelstein, A. e Hoppe, H. (2000). Lapped textures. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 465–470, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Pulli e Ginzton, 2007] Pulli, K. e Ginzton, M. (2007). Scanalyze a system for aligning and merging range data. `http://www-graphics.stanford.edu/software/scanalyze`. Disponível em Janeiro 2009.

[Sander et al., 2007] Sander, P. V.; Nehab, D. e Barczak, J. (2007). Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3).

[Sheppard, 2004] Sheppard, G. (2004). Real-time rendering of fur. Undergraduate thesis. University of Sheffield.

[Sintorn e Assarsson, 2008] Sintorn, E. e Assarsson, U. (2008). Real-time approximate sorting for self shadowing and transparency in hair rendering. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pp. 157–162, New York, NY, USA. ACM.

[Stanford, 2009] Stanford (2009). The stanford 3d scanning repository. `http://www-graphics.stanford.edu/data/3Dscanrep/`. Disponível em Janeiro 2009.

[Swaaij, 2006] Swaaij, M. V. (2006). Ray-tracing fur for ice age: the melt down. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, p. 44, New York, NY, USA. ACM.

[Transgaming, 2009] Transgaming (2009). Cedega. `http://www.cedega.com`. Disponível em Janeiro 2009.

[Turk e Levoy, 1994] Turk, G. e Levoy, M. (1994). Zippered polygon meshes from range images. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 311–318, New York, NY, USA. ACM.

[Turk e Levoy, 2007] Turk, G. e Levoy, M. (2007). Zippack polygon mesh zippering package. `http://www-graphics.stanford.edu/software/zippack/`. Disponível em Janeiro 2009.

[Ward et al., 2007] Ward, K.; Bertails, F.; Kim, T.-Y.; Marschner, S. R.; Cani, M.-P. e Lin, M. (2007). A survey on hair modeling: Styling, simulation, and rendering. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 13(2):213–34. To appear.

[Westin et al., 1992] Westin, S. H.; Arvo, J. R. e Torrance, K. E. (1992). Predicting reflectance functions from complex surfaces. *SIGGRAPH Comput. Graph.*, 26(2):255–264.

[Wikipedia, 2009] Wikipedia (2009). Otter. `http://en.wikipedia.org/wiki/Otter`. Disponível em Janeiro 2009.

[Wyatt, 2007] Wyatt, R. (2007). Ig custom shaders and effects. Technical report, Insomniac Games.

[Yang et al., 2008] Yang, G.; Sun, H.; Wu, E. e Wang, L. (2008). Interactive fur shaping and rendering using nonuniform-layered textures. *IEEE Comput. Graph. Appl.*, 28(4):85–93.