Dissertação para obtenção do grau de mestre em matemática pelo
INSTITUTO NACIONAL DE MATEMÁTICA PURA E APLICADA

VISUALIZAÇÃO DE NUVENS DE PONTOS COM APROXIMAÇÕES QUASE PLANARES E BLENDING DE TEXTURA

por
JOSÉ LUIZ SOARES LUZ

Orientador: PAULO CEZAR P. CARVALHO
Co-Orientador: LUIZ VELHO

27 de Setembro de 2004

Agradecimentos

Gostaria de agradecer primeiramente a Deus, e depois, a todas as pessoas que tornaram possível a realização desta dissertação:
A minha mãe, sempre minha inspiração, guerreira, batalhadora, e que sempre me incentivou a correr atrás de meus sonhos.
A minha avó querida, sempre me ajudando, e me apoiando nas minhas decisões.
A minha irmã, companheira no IMPA, que sempre agüentou o irmão até quando ele não se agüentava.
Aos companheiros do Laboratrio "VISGRAF", grandes amigos, incentivadores, dispostos a discutir quando preciso, aconselhar, e criticar de forma construtiva.
A Lourena Rocha, amiga, companheira, obrigado por seu ombro amigo, incentivo, carinho, afeto...Obrigado por tudo...
Aos meus orientadores Paulo Cezar Carvalho e Luiz Velho, pela paciência, críticas e idéias ao longo do desenvolvimento deste trabalho, meu muito obrigado.

Resumo

Com os recentes avanços na área de escaneamento 3D, a quantidade de dados obtida vem aumentando consideravelmente; estes dados tipicamente são nuvens de pontos não estruturadas, sem informações de conectividade. Muitos dos trabalhos propostos na área de renderização para superfícies representadas por pontos propõem a associação ponto-surfel (elemento de superfície) com o intuito de visualizar a superfície com aproximações do plano tangente em cada ponto.

Este trabalho propõe uma alternativa ao uso das aproximações planares, associando a cada ponto um surfel com leve curvatura, permitindo perceptualmente uma melhor adaptação das aproximações à superfície, favorecida pelo uso de textura e operações de *blending*.

Keywords: Point-based Rendering, Graphics Data Structures, Texture Mapping.

Abstract

With the recent advances in the 3D scanning field, the amount of datasets which should be displayed has increased up to billions of points. Typically, we have a dense, unstructured set of points without connectivity information. Most researchers have proposed the association point-surfel to represent the surface's geometry and render it using planar aproximation for each point.

This work proposes an alternative approximation. We associate a curved surfel to each point allowing perceptually better adaptation to the surface, supported by the use of texture mapping and blending.

Keywords: Point-based Rendering, Graphics Data Structures, Texture Mapping.

Sumário

Lista de Figuras

Capítulo 1

Introdução

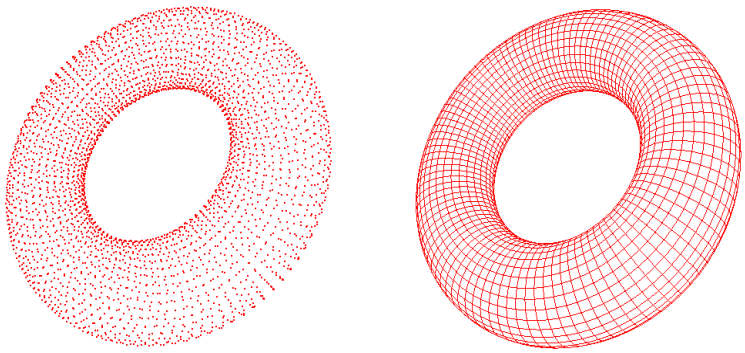
1.1 Motivação

Muitos objetos do mundo real são inerentemente suaves e modelar esses objetos é uma das principais preocupações da computação gráfica através de uma representação correta de curvas e superfícies [9]. Uma questão fundamental com relação a modelagem do mundo real é a escolha da representação mais adequada para a superfície: Qual descrição matemática deve-se escolher para representar a superfície de um objeto 3D? Poderíamos usar uma infinidade de coordenadas dos pontos que definem o objeto, como é o caso dos sistemas de escaneamento 3D; ou uma aproximação do objeto por pedaços de planos, esferas, ou outras formas que são fáceis de descrever matematicamente, necessitando-se que os pontos sobre o modelo estejam suficientemente próximos dos pontos correspondentes sobre o objeto.

Atualmente, os sistemas de escaneamento 3D estão se tornado canônicos no que se refere a obtenção de dados sobre um objeto do mundo real, e como consequência, o problema de gerar visualizações de alta qualidade dos objetos escaneados está recebendo mais atenção. De um modo geral, esses sistemas produzem uma amostragem discreta de um objeto físico, ou seja, uma amostragem de pontos; dependendo da técnica de aquisição, cada ponto carrega um certo número de atributos, tais como: cor, propriedades de material, ou medidas de confiança.

O uso de malhas poligonais geralmente é a maneira ideal de representar os resultados das medidas feitas a partir de um determinado objeto, como por exemplo uma nuvem de pontos não estruturada (Figura 1.1(a)); converter essa nuvem de pontos em um modelo poligonal consistente permite reconstruir de forma precisa ou

aproximada a superfície do objeto (Figura 1.1(b)).



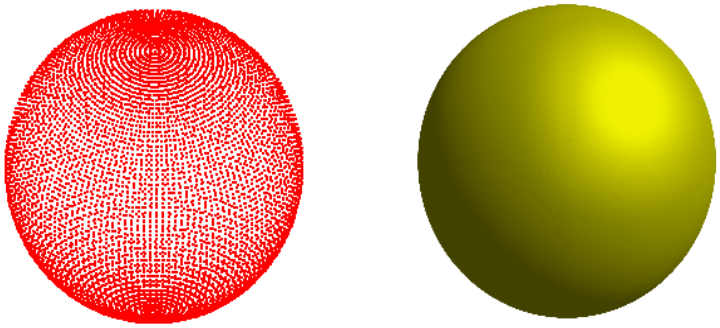
(a) Representação do toro por pontos

(b) Representação do toro através de malha poligonal

Figura 1.1: Exemplos de representação do toro por pontos e por malha poligonal

Muitas técnicas são desenvolvidas para criar uma representação regular e contínua de uma nuvem de pontos através de malhas poligonais, normalmente malhas triangulares; porém com o aumento da complexidade dos modelos geométricos, os polígonos (triângulos) estão ficando cada vez menores, e provavelmente, o processamento gasto com eles não será mais justificado, uma vez que ocuparão áreas correspondentes a sub-pixels no espaço da imagem. Uma das alternativas exploradas atualmente na comunidade científica é a utilização de primitivos mais simples, como pontos.

Dada a simplicidade de um ponto, parece natural que uma geometria baseada em pontos (Figura 1.2(a)) torne-se um importante elemento em modelagem e renderização (Figura 1.2(b)). O objetivo da renderização baseada em pontos (*Point-based Rendering*) é mostrar na tela uma representação por pontos como uma superfície contínua. A motivação para usarmos uma representação por pontos inclui sua eficiência para renderizar objetos complexos, simplicidade dos algoritmos de renderização, e como já citado anteriormente, a crescente utilização de dispositivos de escaneamento 3D.



(a) Representação da Esfera por pontos

(b) Esfera renderizada

Figura 1.2: Exemplos de representação do esfera por pontos e renderizada

1.2 Objetivos

A proposta dessa dissertação consiste na visualização de superfícies representadas por conjuntos de pontos. Através dos atributos associados a cada ponto, tais como: posição, normal, cor, e coordenadas de textura; procuramos reconstruir a superfície usando aproximações locais dos planos tangentes; mas além de usarmos superfícies planares como a maioria dos trabalhos propostos pela comunidade científica, propomos o uso de um primitivo perceptualmente mais adaptado à superfície, o que aliado ao uso de mapeamento de textura e operações de *blending* permitem a reconstrução de forma contínua dos modelos utilizados.

1.3 Esquema da Dissertação

A dissertação está dividida da seguinte forma:

- **Capítulo 2:** Fazemos uma breve discussão sobre renderização com pontos, a evolução das idéias, técnicas desenvolvidas ao longo dos anos, e o *pipeline* básico no processo de visualização.
- **Capítulo 3:** Apresentamos uma motivação para a utilização de textura em computação gráfica, incluindo definições, aplicações, e exemplos.

- **Capítulo 4:** Damos uma visão geral sobre a técnica de *splatting*, descrevendo-se de forma resumida a sua teoria, o algoritmo básico, e um pouco da teoria de filtragem envolvida.
- **Capítulo 5:** Apresentamos os pré-requisitos básicos para os modelos utilizados e a técnica proposta, explicando a construção dos primitivos utilizados no processo de renderização e os resultados obtidos.
- **Capítulo 6:** Fazemos uma revisão da dissertação, com um breve comentário sobre os trabalhos futuros.

Capítulo 2

Representação e Renderização de superfícies com pontos

Neste capítulo falaremos um pouco sobre a representação de superfícies por pontos juntamente com algumas propostas de renderização, fazendo um breve levantamento cronológico das técnicas empregadas e apresentando um *pipeline* básico para renderização de nuvens de pontos.

2.1 Introdução

Os avanços recentes com relação as tecnologias de escaneamento 3D têm resultado na obtenção de uma grande quantidade de dados. Tipicamente, obtemos um conjunto de pontos no espaço tridimensional, denso, não estruturado, e sem informação de conectividade. Progressivamente a quantidade de pontos obtida vem aumentando chegando a ordem de milhões, ou até mesmo bilhões de pontos.

A representação de superfícies por pontos permite o processamento direto dos dados de *scanners* 3D, e a simplicidade estrutural dessas representações dá suporte a uma reamostragem eficiente para deformações geométricas e mudanças topológicas; também permite algoritmos concisos que são adequados para implementação em placas gráficas.

Atualmente há uma tendência em direção a representação de objetos baseada em pontos, uma vez que o crescente aumento da complexidade dos modelos geométricos 3D, e a crescente demanda por uma modelagem avançada e funcional, vêm exigindo o desenvolvimento de algoritmos eficientes e confiáveis para o processamento das

informações geométricas. As atuais técnicas para manipulação destes dados têm evoluído ao longo dos anos principalmente como resultado das pesquisas com malhas triangulares.

O objetivo da renderização baseada em pontos (*Point-based Rendering*) consiste na visualização de uma amostragem de pontos como uma superfície contínua; então surge a questão de como obter um *pipeline* completo para a renderização, que vá desde a aquisição (escaneamento) até a visualização. Isto acarreta o desenvolvimento de diversas técnicas para: representação, processamento, e edição de superfícies baseadas em pontos. Possibilitando trabalhar com nuvens de pontos como se fossem superfícies; podendo-se realizar traçado de raios [19], suavizações, ou qualquer tipo de modificação da superfície [33]. Uma das grandes vantagens da utilização de pontos para modelagem é a facilidade para modificar a topologia do objeto.

Os algoritmos para renderização de pontos podem ser classificados de acordo com a forma como reconstroem, ou aproximam a superfície a partir das amostras. Existem pelo menos quatro diferentes propostas [26]:

- **Detecção de buracos e preenchimento no espaço de tela:** Amostras individuais são projetadas para o espaço de cena, e os pixels que não recebem amostras são detectados. A superfície é então obtida pela interpolação das amostras em uma vizinhança.
- **Geração de mais amostras:** A superfície é adaptada no espaço do objeto de modo que cada pixel recebe pelo menos uma amostra.
- ***Splatting*:** Uma amostra da superfície é projetada no espaço de tela e sua contribuição é espalhada para os pixels vizinhos de modo a garantir o recobrimento. Métodos com resultados melhores calculam a média das contribuições de todos os *splats* para um pixel.
- **Malhas:** Uma malha poligonal é usada para interpolar as amostras sobre uma superfície, um algoritmo completo para renderização de malhas poligonais é necessário.

2.2 Visão Cronológica

Em 1985, Levoy e Whitted [21] propuseram o uso de pontos como um primitivo universal para renderização de superfícies. Eles notaram que quando a complexidade

de um objeto aumenta perdemos parte da eficiência fornecida pela renderização de polígonos ou outro primitivo de mais alto nível; sendo que pontos são mais simples e capazes de representar qualquer tipo de objeto (superfície). Considerando uma superfície diferenciável, faziam uma estimativa do plano tangente e da normal para um conjunto de pontos em uma determinada vizinhança. A reconstrução da superfície era feita estimando-se para cada pixel uma cobertura calculada a partir de pesos acumulados, os quais são obtidos pelo número de pontos projetados para o espaço de tela. O *antialiasing* nas arestas pode ser feito considerando-se pixels parcialmente cobertos como semi-transparentes.

Em 1992, Szeliski e Tonnenson [29] usavam partículas orientadas para modelagem de superfícies e edição interativa. As partículas orientadas eram pontos com um sistema de coodenadas local, onde tinha-se interação entre os pontos (partículas) a partir de forças de repulsão e atração. A visualização era feita a partir de elipses e triangulação, não havendo a utilização de técnicas de *point-rendering*. As partículas orientadas eram consideradas como elementos de superfície (surfel - *surface element*).

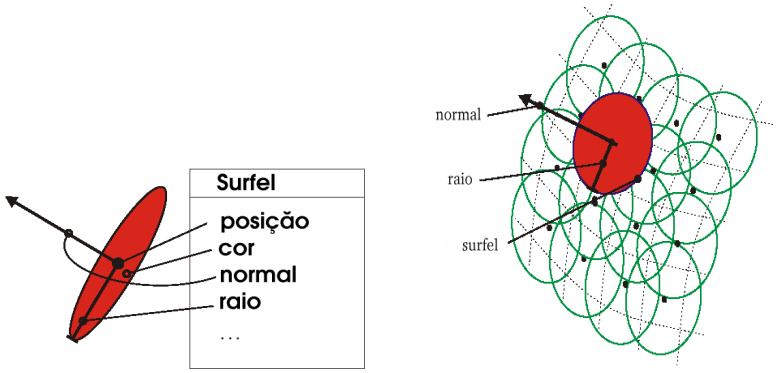
Em 1998, Grossman e Dally [15] consideraram amostragens de objetos a partir de um conjunto de vistas ortográficas. Na renderização os pontos são transformados e escritos para o *frame-buffer*, e usando uma hierarquia de *depth-buffers*, eles estabeleceram valores de confiança para um pixel, através da comparação do seu valor de profundidade com os valores de profundidade de pixels próximos. A partir destes valores de confiança estabeleceram um critério para detectar buracos, se um valor de confiança estava abaixo de um determinado limiar (*threshold*), então o pixel era considerado como um buraco, e preenchido seguindo uma determinada heurística.

Em 2000, dois trabalhos destacaram-se na de renderização de superfícies representadas por pontos: o trabalho de Pfister et al. [25] e o de Rusinkiewicz e Levoy [28].

Pfister et al. estenderam o trabalho de Grossman e Dayle com um controle hierárquico do nível de detalhes (LOD - *level of detail*) e da visibilidade. Em um pré-processamento, objetos são amostrados a partir de três direções ortogonais e as amostras são armazenadas em uma *octree* de *Layered Depth Images*[1]. Cada amostra (surfel - Figura 2.1) contém uma normal a superfície, e um conjunto de amostras de textura pré-filtradas, assim como outros atributos. Durante a renderização, as

[1]LDI - uma LDI consiste na vista de uma cena a partir de uma câmera, mas com múltiplos pixels ao longo de cada linha de visão. Os dados de uma LDI são representados em um sistema simples de coordenadas da imagem

amostras dentro dos nós visíveis da *octree* são projetadas para o espaço de cena. Após a projeção, buracos são detectados e preenchidos usando *splatting* e o *depth-buffer*, interpolando-se os atributos da superfície nos pixels que contêm as amostras.



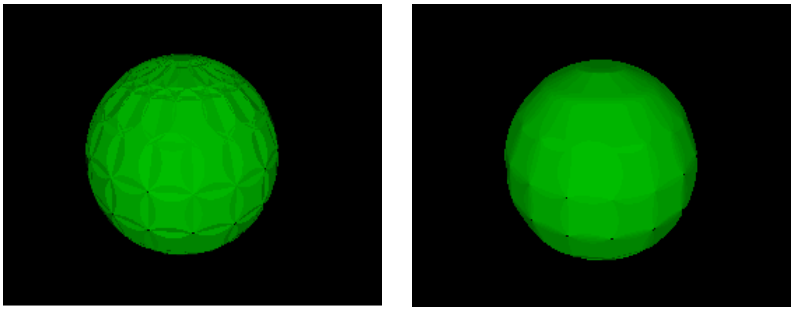
(a) (b)

Figura 2.1: Surfel: posição, normal, raio

Rusinkiewicz e Levoy desenvolveram um sistema de renderização baseado em *splatting*, que permite a visualização em tempo real de modelos formados por milhões de pontos. Eles utilizam uma estrutura hierárquica baseada no conceito de esferas envolventes para a controle do nível de detalhes, recorte (*culling*), e visibilidade. Durante a renderização a estrutura hierárquica é percorrida até um nível que depende do ângulo de visão, da resolução da tela, e dos recursos computacionais disponíveis. A amostra é então renderizada (*splatted*) sobre a tela. Os *splats* são orientados ao longo do plano de visão e renderizados de trás para frente (*back-to-front order*) como *quads* texturizados. Transparência e questões relativas a *antialiasing* não são consideradas.

Ainda em 2000, Schauffer e Jensen [19] usaram pequenos discos para renderização de pontos, enxergando esses discos como pequenos pedaços de planos tangentes à superfície em cada ponto, utilizaram traçado de raios para realizar a interpolação dos atributos associados a cada ponto, obtendo imagems de alta qualidade com iluminação global (Figura 2.2).

Em 2001, Alexa et al [1] propuseram um método para reconstrução baseado em aproximações polinomiais locais para um conjunto de pontos usando *Moving Least*



(a) Esfera sem interpolação (b) Esfera com interpolação

Figura 2.2: Interpolação com traçado de raios

Squares (MLS). Também usaram MLS para adicionar e remover pontos à superfície durante a renderização, para a obtenção de superfícies suaves de alta qualidade.

Zwicker et al. [34], utilizaram filtros EWA (*Elliptical Weighted Average*) para a reconstrução de amostras de pontos, onde os *splats* eram enviados para o interior de um *a-buffer*. A reconstrução era feita utilizando uma ponderação sobre os valores dos fragmentos dos *splats* com valores de profundidade similares para cada pixel, permitindo uma filtragem de textura de alta qualidade, transparência independente da ordem, e *shading* (cálculos de iluminação) por pixel. Problemas relacionados com *antialiasing* são resolvidos como no trabalho de Levoy e Whitted [21].

Em 2002, um tutorial do Eurographics [14] sobre computação gráfica baseada em pontos cobriu diversos tópicos sobre técnicas para renderização de superfícies baseadas em pontos.

2.3 Renderização de Pontos

Várias aproximações existem para renderizar objetos a partir de um conjunto de pontos. A diferença está na forma como interpretam-se os primitivos para a renderização, podendo ser interpretados como pontos sem dimensão, ou ainda como núcleos de reconstrução planares (*splatting*).

A maioria dos algoritmos enviam os dados (pontos) para o *pipeline* de

renderização e calculam as suas contribuições; é o que se costuma chamar de *object-order*: para cada objeto acham-se os pixels que ele ocupa. Esta forma de processar os dados é caracterizada por uma projeção dos objetos na tela (espaço do objeto), para o espaço de cena (*forward mapping*), como é o caso do *splatting*. Também pode-se utilizar o processamento em *image-order*, para cada pixel acham-se os objetos que o ocupam; é caracterizado pelo mapeamento que vai do espaço de cena para o espaço onde estão definidos os dados (*backward mapping*), como por exemplo traçado de raios e mapeamento de textura na renderização de polígonos. Muitos algoritmos realizam uma etapa da renderização em *object-order* e outra etapa em *image-order*, e um *depth-buffer* ou um *a-buffer* para armazenar os dados entre as etapas. Outras técnicas dão suporte no processo de construção da imagem, tais como: recorte (*culling*), estruturas de dados, controle de nível de detalhe, etc.

Na renderização baseada em pontos precisamos analisar quais atributos estão relacionados a cada ponto de uma determinada amostragem: uma posição, uma normal, atributos de cor para cálculos de *shading*. Se existe uma área associada com o ponto podemos caracterizar um elemento de superfície (*surface element* - surfel). Um surfel representa um pedaço da superfície ao invés de simplesmente um ponto da amostra. De um modo geral não faz-se distinção entre ponto e surfel. Dependendo da situação pode-se empregar uma ou outra designação; comumente o termo surfel está relacionado com o processo de renderização, e ponto com a modelagem. Pode-se também armazenar outros atributos como transparência e propriedades do material para cada surfel, e sua área pode ser expressa pelo valor de um raio considerando-se uma área circular no espaço do objeto. As áreas dos surfels devem garantir uma reconstrução livre de buracos.

Aceitar pontos como dados de entrada e produzir uma imagem final como saída é a parte essencial de qualquer sistema de renderização de pontos. Segundo Krivanek [20] pode-se esboçar um pipeline para a renderização de pontos conforme o esquema abaixo.

- ***Warping*:** Projeta-se cada ponto para o espaço de cena usando projeção perspectiva.
- ***Shading*:** Faz-se os cálculos de *shading* (iluminação) por ponto, geralmente após resolvido os problemas de visibilidade, ou após a interpolação dos atributos dos pontos na reconstrução da imagem (*shading* por pixel).

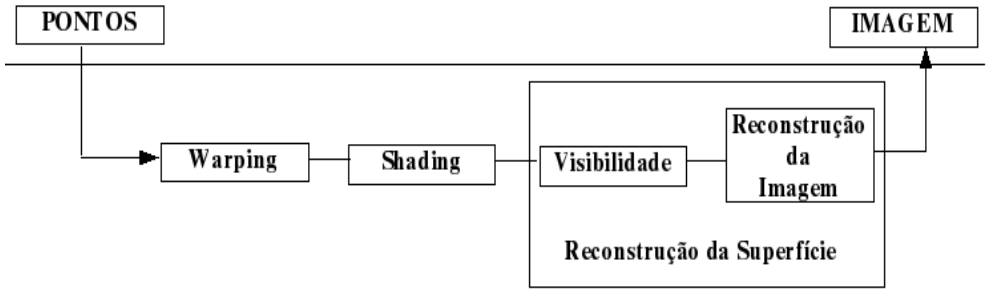


Figura 2.3: *Pipeline* para *Point-Rendering*

- **Visibilidade e Reconstrução da Imagem:** A reconstrução da superfície no espaço de cena depende da posição do observador. Essas duas técnicas envolvem dois passos: o primeiro identifica os pontos em *background*, e os deleta do *frame-buffer*, e marca os pixels correspondentes como buracos. No segundo passo, o *frame-buffer* resultante, contem somente as informações dos pixels correspondentes aos pontos em *foreground* e os buracos, então é feita a reconstrução da imagem. Ambas as técnicas podem ser realizados separademente ou em conjunto. Pode-se incluir filtragem para *antialiasing*.

O principal problema com relação a visibilidade e reconstrução da imagem final é o aparecimento de buracos. Quando analisa-se as contribuições dos pontos para um pixel diversos fatores devem ser levados em consideração: se o pixel recebe ou não alguma contribuição, pontos sofrendo oclusão, pontos em *background* com ou sem pontos em *foreground*; e nem sempre as informações de *depth-buffer* são suficientes para resolver todos os casos. Uma fator fundamental é conhecer a densidade das amostras. Uma amostragem aparentemente ideal teria uma densidade maior nas regiões com mais detalhes geométricos.

Quando a densidade das amostras é pequena, deixando buracos durante a renderização, pode-se gerar mais amostras de acordo com a posição do observador, assegurando uma densidade alta o suficiente para evitar o aparecimento de buracos. Desta forma, cada pixel recebe as informações relativas a um ponto em *foreground*. Uma desvantagem deste processo está no fato de que as amostras são geradas dinamicamente durante a renderização, o que exige mais em termos de tempo.

2.4 Visibilidade (*Visibility Culling*)

A realização de recortes para resolver questões de visibilidade não se torna eficiente quando se pensa em cada ponto separadamente. Desta forma, temos que realizar os recortes através de grupos de pontos ou utilizando uma determinada estrutura hierárquica.

Os sistemas atuais para renderização de pontos utilizam recorte, uma vez que se mostram indispensáveis para uma boa performance. Os mais comumente utilizados são: *view frustum culling*, e o *backface culling*.

- ***view frustum culling*:** recorta a geometria que não está dentro do frustum de visualização; seja qual for a forma como os pontos estejam organizados, cada grupo de pontos para ser recortado necessita ter algum tipo de caixa envolvente (*bounding box*). O teste de recorte é feito verificando-se se a caixa envolvente está dentro ou fora do frustum. O frustum é definido pelos planos *near* e *far*, e por quatro planos que se interceptam na posição do observador.
- ***backface culling*:** realiza o recorte baseando-se no sentido do vetor de visualização e na orientação da geometria (direção-sentido das normais). Desta forma para cada grupo de pontos deve-se analisar se as respectivas normais estão de acordo ou não com o sentido do vetor de visualização. O cone de normais é uma técnica padronizada para a realização de *backface culling*, onde cada grupo de pontos é armazenado juntamente com um cone de normais. O cone é representado por seu eixo (normal média) e sua largura.

2.5 Textura e *Splatting*

Muitas técnicas para a renderização de superfícies com representação por pontos baseiam-se nos fundamentos do mapeamento de textura: amostragem, filtragem, *blending*, *antialiasing*, o que permite uma renderização de alta qualidade. Uma técnica comumente usada neste sentido é o *splatting*.

Splatting é uma técnica para a reconstrução de superfícies representadas por pontos, que assume que uma área é designada para cada surfel no espaço do objeto, e a reconstrução da superfície de forma contínua é feita através da rasterização da projeção do surfel para o espaço de cena. A forma do surfel projetado é denominada

splat.

Segundo Botsch, Wiratanaya, and Kobbelt [4] a idéia básica do *splatting* seria substituir cada amostra de uma nuvem de pontos por pequenos discos tangentes à superfície, cujos raios estariam de acordo com a densidade local de uma determinada amostragem de pontos, e cuja opacidade pode ser constante ou decair radialmente a partir do seu centro. Quando um desses discos é projetado sobre o plano de cena torna-se um *splat* elíptico, cuja distribuição da intensidade é chamada de *footprint* ou *kernel* do *splat*. Se muitos *splats* sobrepõem-se sobre o mesmo pixel, então a cor do pixel é calculada baseando-se na média ponderada das intensidades das cores dos *splats*.

Como esta dissertação baseia-se no uso de mapeamento de textura e na definição acima de *splatting*, vamos discutir os aspectos gerais do mapeamento de textura no capítulo 3 e de forma resumida a técnica de visualização por *splatting* no capítulo 4.

Capítulo 3

Mapeamento de Textura

Este capítulo visa abordar os aspectos básicos de mapeamento de textura, desde a idéia perceptual até o conceito matemático. Falaremos sobre o desenvolvimento das aplicações em ordem cronológica, sobre as transformações envolvidas entre os espaços *cena-objeto-textura*; e abordaremos alguns aspectos do uso de textura para obtenção de objetos com transparência.

3.1 Introdução

Ao longo da história da computação gráfica, os pesquisadores vêm buscando melhorar o realismo das imagens geradas. No princípio as imagems das superfícies apresentavam suavidade excessiva, não mostravam textura, distorções, arranhões, sujeira, etc.

Realismo exige complexidade, ou pelo menos aparência de complexidade, e tentando encontrar a melhor forma de renderizar uma superfície, buscou-se o desenvolvimento de técnicas para a geração de imagens com maior grau de realismo. Estas técnicas podem ser divididas basicamente em *shading* e mapeamento de textura.

Shading consiste na determinação das propriedades da superfície de um objeto, tais como: cor, informação de normal, reflectância, transparência e modelo de iluminação. A cor de um pixel é calculada a partir das propriedades da superfície definidas pelo usuário e do modelo de *shading*.

O mapeamento de textura é uma técnica para variação das propriedades da superfície de um ponto para outro, com o intuito de melhorar os detalhes relativos a

sua aparência e que não estão presentes em sua geometria. O mapeamento pode ser aplicado para diversos atributos: cor, normal, especularidade, transparência, iluminação, sendo as técnicas de mapeamento essencialmente as mesmas em todos os casos. Segundo Heckbert [17], os fundamentos do mapeamento de textura podem ser divididos em dois tópicos: o mapeamento geométrico, que adapta a textura sobre uma superfície e a filtragem, que é necessária para evitar *aliasing*.

A textura consiste basicamente em uma informação visual relacionada à natureza de um determinado objeto a ser representado, servindo como parâmetro para alterar o seu aspecto final, e simulando textura no sentido usual como, por exemplo, uma nuvem, mármore, madeira (Figura 3.1).

Matematicamente Gomes e Velho [12] definem textura como uma aplicação

$$t : U \subset \mathbb{R}^m \longrightarrow \mathbb{R}^k$$

de um subconjunto U do espaço euclidiano $\mathbb{R}^m$ no espaço euclidiano $\mathbb{R}^k$. A denominação textura geralmente considera o caso em que $m = 2$ e $k = 3$, e o espaço euclidiano $\mathbb{R}^k$ é identificado com um espaço de cor. Neste caso t representa uma imagem digital.

Dada uma função

$$g : V \longrightarrow U \subset \mathbb{R}^m$$

de um subconjunto $V \subset \mathbb{R}^n$ do espaço do objeto, chamamos mapeamento de textura à composta de aplicações

$$m = t \circ g : V \longrightarrow \mathbb{R}^k,$$

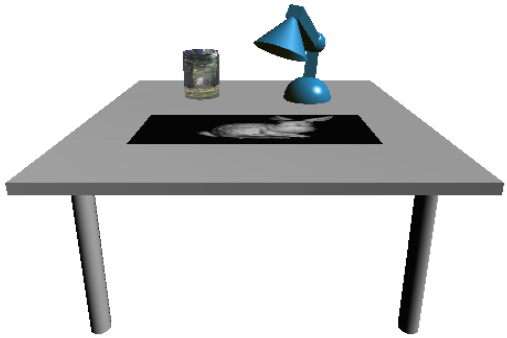
que associa a cada ponto (x, y, z) do espaço do objeto, um elemento $t(g(x, y, z))$ do espaço vetorial $\mathbb{R}^k$ (espaço de atributos).

Deste modo o mapeamento de textura envolve os espaços: $V \subset \mathbb{R}^n$ (espaço do objeto), $U \subset \mathbb{R}^m$ (espaço de textura), e $\mathbb{R}^k$ (espaço de atributos), os quais estão relacionados pelas funções:

$$\begin{array}{ll} t : U \subset \mathbb{R}^m \longrightarrow \mathbb{R}^k & \text{(função de textura)}, \\ g : V \subset \mathbb{R}^n \longrightarrow U & \text{(função de mapeamento)}. \end{array}$$

A função de textura faz a associação entre o domínio geométrico da textura e seus valores de atributos, e depende da dimensão m do espaço de textura $\mathbb{R}^m$ e da natureza do espaço de atributos $\mathbb{R}^k$. A função de mapeamento estabelece a

(a) Imagem de textura



(b) Mesa sem textura

(c) Mesa com textura

Figura 3.1: Exemplo de mapeamento de textura em uma cena

correspondência entre pontos do objeto e pontos do domínio da textura.

De acordo com a dimensão do domínio da textura ($\mathbb{R}^m$), temos três tipos de mapeamento:

- **Mapeamento 1D:** quando $m = 1$, nesse caso t é um mapa de cor;
- **Mapeamento 2D:** quando $m = 2$, nesse caso t é uma imagem digital;
- **Mapeamento 3D:** quando $m = 3$, nesse caso t é chamada de textura sólida.

Um dos usos mais populares de textura consiste no mapeamento 2D, onde temos uma imagem (espaço de textura), que é mapeada sobre uma superfície 3D (espaço do objeto), que é então mapeada para a imagem final (espaço de cena ou da imagem) através de uma projeção. Temos as coordenadas (u, v) no espaço de textura, (x, y, z) no espaço do objeto, e (s, l) no espaço de cena.

3.2 Aplicações de Mapeamento de Textura

As aplicações para o mapeamento de textura são diversas. Como mencionamos anteriormente, ele afeta vários parâmetros, tais como: reflexão, cor, transparência, rugosidade. Embora muito simples em princípio, o mapeamento de uma imagem digital envolve alguns truques de filtragem, e muitos trabalhos podem ser encontrados relativos ao uso de textura em computação gráfica (Ebert [8], Haines [16], Heckbert [17]).

Os primeiros esforços provaram que utilizando textura pode-se obter uma aparência interessante e detalhada para uma superfície, e não simples e tediosa com uma cor uniforme. Desta forma houve um salto significativo no grau de realismo das imagens geradas.

Aproximadamente ao mesmo tempo em que os primeiros modelos de reflexão especular estavam sendo formulados, Catmull [6] gerou as primeiras imagens texturizadas em computação gráfica. As superfícies de Catmull eram representadas por *patches* paramétricos. Um *patch* paramétrico é uma função de dois parâmetros (u, v); desta forma há um único ponto bi-dimensional associado a cada ponto 3D sobre a superfície, podendo-se assim facilmente estabelecer uma transformação que

associe os pixels da imagem de uma textura digital (texels), aos pontos correspondentes (u, v) no espaço de parâmetros de um *patch*, para onde a textura digital será mapeada.

Blinn e Newel [2] introduziram o *reflection mapping* (também chamado *environment mapping*) para simular reflexões de superfícies de forma semelhante a um espelho. O *reflection mapping* é uma forma simplificada de traçado de raios (*ray tracing*), calculando-se a direção de reflexão r de um raio a partir da câmera (ou observador) para o ponto sendo iluminado. Assim temos um vetor de reflexão r dado por:

$$r = 2(n \cdot v) - v$$

onde n é a normal à superfície e v aponta na direção do observador (ambos devem ser normalizados). O vetor de reflexão é então transformado para coordenadas esféricas (ρ, θ), onde θ representa a longitude variando de 0 a 2π radianos, e ρ representa a latitude variando de 0 a π radianos. O par (ρ, θ) é obtido a partir das equacões:

$$\rho = \arccos(-r_z)$$

$$\theta = \begin{cases} \arccos(r_x / \sin(\rho)) & \text{se } r_y \geq 0 \\ 2\pi - \arccos(r_x / \sin(\rho)) & \text{caso contrário} \end{cases}$$

onde $r = (r_x, r_y, r_z)$ é o vetor de reflexão normalizado. As coordenadas esféricas são então mapeadas para os valores no intervalo $[0, 1)$ e usadas como as coordenadas (u, v) no espaço de textura, produzindo a cor refletida. Se a textura de reflexão é escolhida cuidadosamente, a textura na superfície parece estar refletindo uma imagem na sua vizinhança. A ilusão é aumentada se ambas, a posição do ponto iluminado, e a direção de reflexão são usadas para calcular uma interseção de um raio com uma esfera imaginária que envolve a cena.

Blinn [3] em 1978, introduziu o *bump mapping*, que tornou possível simular a aparência de superfícies com ondulações, protuberâncias (*bumps*), sem na verdade modificar a geometria. A idéia básica consiste em modificar a normal à superfície ao invés de modificar as componentes de cor na equação de iluminação. A normal geométrica permanece a mesma, modificando-se apenas a normal usada no cálculo de iluminação. Como exemplo podemos citar duas técnicas para realizar o *bump mapping* (Figura 3.2); uma delas usa valores bu e bv em cada ponto, onde esses valores correspondem a variações ao longo dos eixos u e v no espaço de textura. A segunda usa um mapa de alturas, tomando cada valor monocromático da textura

para representar a altura, deste modo a cor branca representa uma área elevada, e a cor preta uma área baixa. O mapa de alturas é usado para calcular variações u e v de forma similar a primeira técnica; isto é feito tomando-se a diferença entre colunas (linhas) vizinhas para o cálculo das inclinações para u (v).

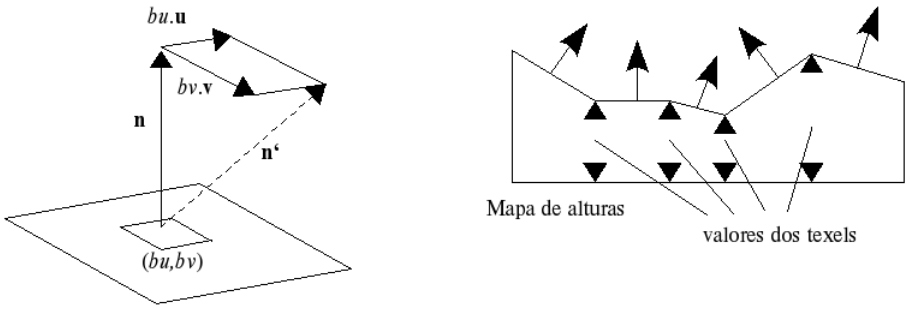


Figura 3.2: *Bump Mapping*

O *bump mapping* é extremamente convincente e oferece uma forma barata de adicionar detalhes à geometria. Dentre as suas limitações, está o fato de que ao longo das silhuetas dos objetos não percebemos o efeito de *bumping*. Além disso, sombras não são produzidas pelos *bumps*, o que pode provocar uma perda no grau de realismo do objeto renderizado.

Cook [7] descreveu uma extensão do *bump mapping* chamada *displacement mapping*, na qual as texturas são usadas para mover a superfície, o que permite adicionar maior complexidade à geometria. O método aplica-se a uma superfície base, que pode ser um *patch*, ou uma superfície de subdivisão. Uma superfície base pode ser definida como uma função vetorial de duas variáveis $P(u, v)$ que corresponde a pontos (x, y, z) sobre a superfície. Também pode-se caracterizá-la por um escalar de deslocamento (*displacement*) $d(u, v)$, e a normal sobre a superfície base por $N(u, v)$. Usando esta representação obteremos os novos pontos deslocados sobre a superfície que serão dados por:

$$P_0(u, v) = P(u, v) + d(u, v) \cdot \overline{N}(u, v) \quad \text{, onde} \quad \overline{N}(u, v) = \frac{N(u,v)}{|N(u,v)|}$$

Movendo a superfície não alteramos a direção das normais, então o *displacement mapping* frequentemente parece uma espécie de *bump mapping* exceto pelo fato de

que os *bumps* criados pelos deslocamentos estão visíveis mesmo na silhueta dos objetos.

Reeves et al. em 1987 [27] apresentaram um algoritmo para produzir sombras sem *aliasing* usando mapeamento de textura e a profundidade da imagem de uma cena renderizada a partir da posição da fonte de luz. Uma técnica de amostragem estocástica chamada *percentage closer filtering* foi aplicada para reduzir os efeitos de *aliasing*.

Peachey em 1985 [23] e Perlin em 1985 [24] descreveram as *space filling textures* (texturas sólidas) como uma alternativa à textura 2D. Gardner [1984 [10], 1985 [11]] usou textura sólida gerada pelas somas de funções senoidais para adicionar textura a modelos de árvores, terrenos e nuvens. Texturas sólidas são avaliadas baseadas nas coordenadas 3D do ponto sendo texturizado ao invés dos parâmetros 2D do ponto sobre a superfície. Conseqüentemente texturas sólidas não são afetadas pelas distorções no espaço de parâmetros da superfície, tais como os que são vistos próximos aos pólos de uma esfera. Além disso, a continuidade entre as parametrizações da superfície entre *patches* adjacentes não é um problema que cause preocupação, uma vez que a textura sólida permanecerá consistente, e tem características de tamanho constante apesar das distorções no sistema de coordenadas da superfície.

3.3 A Geometria do Mapeamento de Textura

A geometria do mapeamento de textura segundo Weinhaus [31] pode ser explicada de acordo com o esquema abaixo (Figura 3.3). Temos duas transformações envolvidas: uma transformação entre o espaço do objeto e o de textura (objeto-textura) e outra entre o espaço do objeto e o de cena (objeto-cena). O espaço do objeto é caracterizado pela equação de cada superfície do modelo 3D, ou simplesmente pelas coordenadas (x, y, z) que definem cada ponto sobre a superfície do objeto.

Quando a textura é mapeada ortogonalmente sobre uma superfície plana, a transformação objeto-textura pode ser tão simples quanto uma transformação afim ou bilinear, ou ainda pode ser uma transformação paramétrica, quando as coordenadas de textura são usadas para representar coordenadas não cartesianas, tais como coordenadas cilindrícas ou esféricas, e as texturas são deformadas para se adaptarem ao modelo 3D.

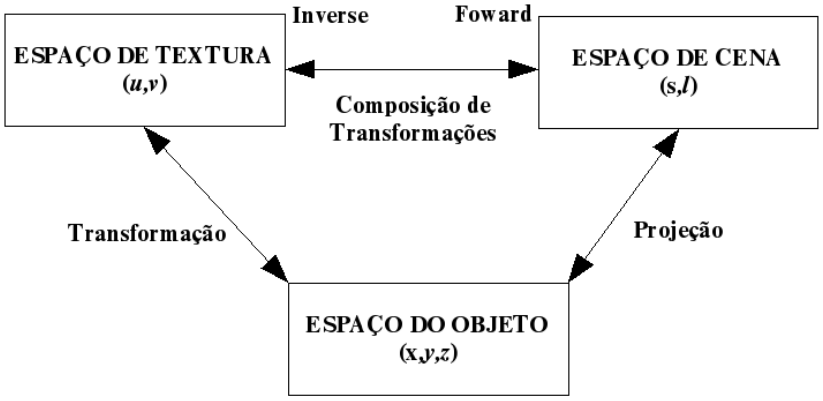


Figura 3.3: Geometria do mapeamento de textura

A transformação objeto-cena é uma projeção ortográfica ou uma projeção perspectiva. A primeira tem como principal característica o fato de que raios paralelos permanecem paralelos após transformados, e a transformação é uma combinação de translação e escalamento. A segunda é um pouco mais complexa. Neste tipo de projeção quanto mais distante um objeto está da câmera, menor ele parece, e além disso, linhas paralelas podem convergir no infinito. A transformação perspectiva imita o modo como percebemos o tamanho dos objetos no mundo real.

Frequentemente as transformações objeto-textura e objeto-cena são concatenadas com o intuito de diminuir o esforço computacional. A transformação composta resultante pode ser *forward* (textura-cena), ou *inverse* (cena-textura). Cada qual com suas vantagens e desvantagens.

A tabela abaixo apresenta algumas das equações que definem as transformações fundamentais para o mapeamento de textura tradicional. Nessas equações, conforme o sentido da transformação (Figura 3.3), temos x, y (e opcionalmente z), e X, Y (e opcionalmente Z), representando as coordenadas, ou no espaço do objeto, ou no de textura, ou ainda no de cena.

Nome	***Propriedades***	***Forma***
AFIM(ou LINEAR)	Translação, escalamento, rotação Preserva linhas retas, paralelismo e homogeneidade na escala	$x = A_0 + A_1X + A_2Y$ $y = B_0 + B_1X + B_2Y$
BILINEAR	Preserva retidão de linhas horizontais e verticais Mapeia linhas retas em curvas hiperbólicas	$x = A_0 + A_1X + A_2Y + A_3XY$ $y = B_0 + B_1X + B_2Y + B_3XY$
QUADRÁTICA	Mapeia linhas retas em curvas quadráticas	$x = A_0 + A_1X + A_2Y + A_3XY + A_4X^2 + A_5Y^2$ $y = B_0 + B_1X + B_2Y + B_3XY + B_4X^2 + B_5Y^2$
CÚBICA	Mapeia linhas retas em curvas cúbicas	$x = A_0 + A_1X + A_2Y + A_3XY + A_4X^2 + A_5Y^2 + A_6X^2Y + A_7XY^2 + A_8X^3 + A_9Y^3$ $y = B_0 + B_1X + B_2Y + B_3XY + B_4X^2 + B_5Y^2 + B_6X^2Y + B_7XY^2 + B_8X^3 + B_9Y^3$
PROJEÇÃO - Perspectiva - Ortográfica	Preserva linhas retas	$\begin{pmatrix} wx \\ wy \\ wz \\ w \end{pmatrix} = PRT \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$ w é eliminado pela divisão pela 4^a coordenada **Matrizes** P =projeção R =rotação T =translação $P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 1 \end{pmatrix}$ f =distância focal para o caso de perspectiva $1/f = 0$ e $w = 1$ para o caso ortográfico

A partir destas idéias, tal como em [16], suponhamos que queremos mapear a imagem de um muro de tijolos em um polígono simples. Então, determinamos uma posição (x, y, z) no objeto (polígono) no sistema de coordenadas do mundo, digamos $(-2.3, 7.1, 88.2)$, a partir dai projetamos esta coordenada para um espaço paramétrico, ou seja, transformamos o valor deste ponto em um par de coordenadas, cada qual variando entre 0 e 1, obtendo o par $(0.32, 0.29)$; esse valor no espaço de parâmetros (u, v) é utilizado para achar a cor da imagem nesta localização. Vamos supor que a resolução da imagem que representa o muro é de 256 por 256, então multiplicando-se cada coordenada do par (u, v) por 256, teremos $(81.29, 74.24)$, arredondando para obtermos valores inteiros, obteremos o pixel $(81, 74)$ na imagem do muro a ser mapeada, cuja cor é dada por $(0.9, 0.8, 0.7)$ (Figura 3.4). Alternativamente, poderíamos aplicar alguma forma de interpolação as cores dos quatro vizinhos de $(81.29, 74.24)$.

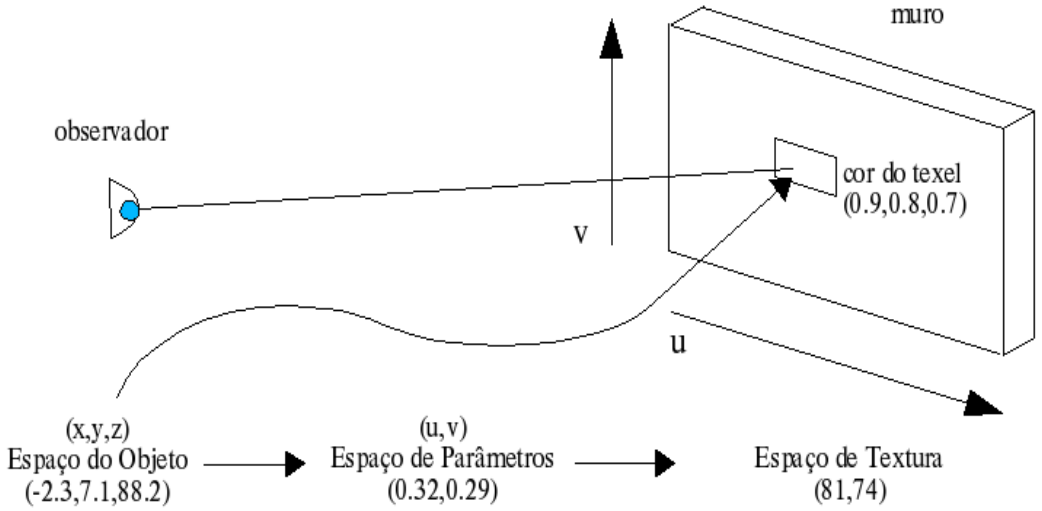


Figura 3.4: Exemplo da geometria do mapeamento de textura em uma cena

Os pixels na textura são frequentemente chamados de texels, para diferenciá-los dos pixels na tela. Quanto ao modelo de iluminação a ser utilizado, não existe um específico para o mapeamento, e pode-se considerar duas situações nesta aplicação:

1. O valor da intensidade do pixel na imagem final será exatamente o mesmo obtido da imagem usada no mapeamento.

2. O valor da intensidade do pixel da imagem final será o resultado da combinação do valor obtido da imagem de textura com o valor gerado pelo cálculo do modelo de iluminação.

Suponhamos agora que queremos mapear uma textura para um polígono regular definido sobre o plano XOY e centrado sobre a origem, cujo raio da circunferênica circunscrita seja igual a 1. Então devemos projetar as coordenadas de seus vértices sobre um domínio paramétrico uv ($[0,1] \times [0,1]$). Podemos fazer isto utilizando as seguintes equações:

$$u = \frac{1}{2} \cdot \cos\left(\frac{2k\pi}{n}\right) + \frac{1}{2}, \qquad v = \frac{1}{2} \cdot \sin\left(\frac{2k\pi}{n}\right) + \frac{1}{2},$$

onde k varia de 0 até $(n-1)$ (n: número de lados do polígono).

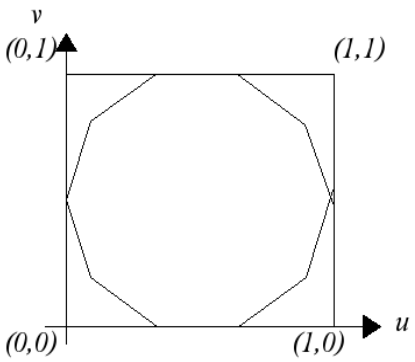
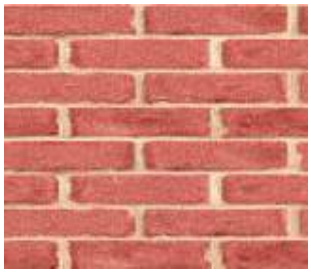
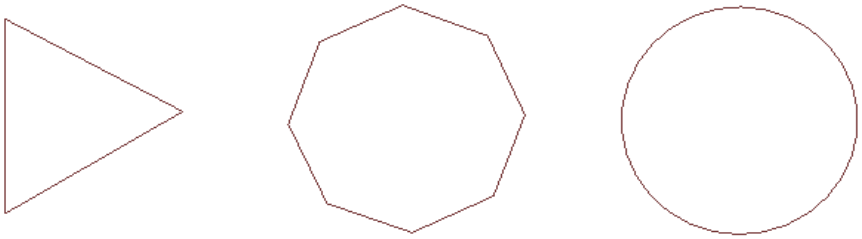


Figura 3.5: Polígono definido no espaço de textura

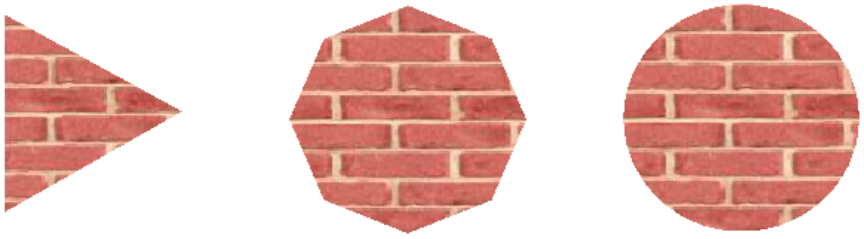
Neste caso não estaremos deformando a textura, mas simplesmente ajustando-a de forma a termos o polígono definido dentro do espaço paramétrico. Uma vez obtidas as coordenadas no domínio paramétrico, podemos fazer o mapeamento da textura, tal como foi comentado no exemplo anterior (Figura 3.6).



(a) Imagem de textura



(b) 3 lados (c) 8 lados (d) 30 lados

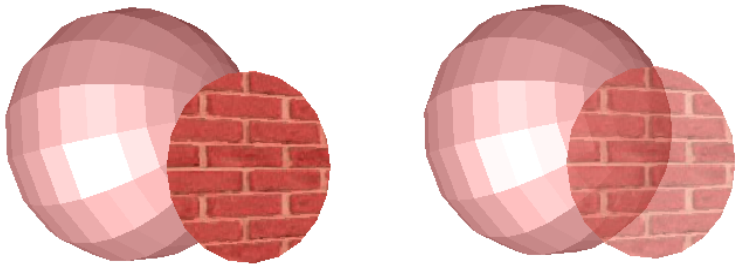


(e) (f) (g)

Figura 3.6: Polígonos com textura mapeada

3.4 Mapeamento de Textura, Transparência e Blending

Outro uso interessante de textura é o controle da transparência de uma superfície (*Transaparency mapping* - Figura 3.7), que é feito através da utilização de valores de opacidade na imagem de textura. Esta técnica é útil por exemplo para a simulação de nuvens e árvores através da rasterização de polígonos com textura sobre o fundo de uma cena, de modo que o fundo da cena é visto através das margens das nuvens ou dos ramos das árvores.



(a) (b)

Figura 3.7: Exemplo de aplicação de textura com transparência

Durante a renderização de uma cena determina-se a cor correta para cada pixel da imagem. A informação para cada pixel está localizada no *color-buffer*, que consiste em uma matriz retangular de cores (vermelho (*Red*), verde (*Green*), azul (*Blue*)) com as informações de cor dos primitivos visíveis na cena do ponto de vista da câmera. A visibilidade para a maioria dos dispositivos gráficos é determinada pelo algorítmo de *z-buffer*, também chamado *depth-buffer*, de modo que para cada pixel armazena-se um valor de profundidade z do primitvo com relação a câmera. Assim, quando um primitivo vai ser renderizado, o seu valor z de profundidade para um dado pixel é comparado com os valores no *depth-buffer*, se o valor for menor que o valor previamente armazenado para aquele pixel, o primitivo está mais proximo da câmera, então atualizam-se os valores de z e de cor para o pixel. Se o valor de profundidade z for maior que o do *depth-buffer*, os valores para z e para a cor são deixados inalterados. Alguns cuidados especiais devem ser tomados com relação a ordem (profundidade)

com que os objetos são enviados para o *pipeline* de renderização quando consideramos questões relativas a transparência.

Para efeitos mais gerais e flexíveis de transparência deve-se saber quando e como misturar, combinar (*blending*) a cor de um objeto transparente com a cor de um objeto atrás dele. Para isso tem-se o conceito de *alpha-blending*. Quando um objeto é renderizado na tela usam-se as informações de cor (RGB) e profundidade (*depth-buffer*) para cada pixel, sendo que outra componente chamada *alpha*, também pode ser armazenada. O valor *alpha* descreve o grau de opacidade de um objeto para um dado pixel. Um valor *alpha* igual a 1.0 (ou 255), significa que o objeto é opaco e cobre inteiramente a área de interesse do pixel. À medida que o valor decai para 0, a opacidade do objeto diminui até tornar-lo totalmente transparente.

Para percebermos a transparência de um objeto deve-se renderizá-lo sobre a cena com *alpha* menor do que 1.0 (ou 255), cada pixel coberto por este objeto receberá um valor para a cor (RGBA) durante a renderização. O *blending* (combinação) irá combinar a cor do objeto processado com a cor do pixel previamente armazenado no *color-buffer* usando um operador *over* dado por:

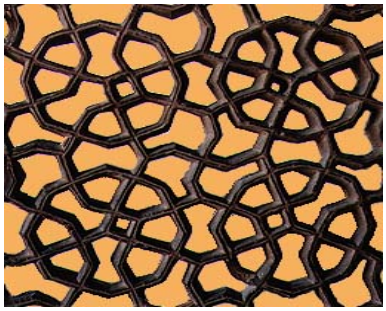
$$c_0 = \alpha_s \cdot c_s + (1 - \alpha_s) \cdot c_d \quad \text{operador } \textit{over},$$

onde c_s é a cor do objeto transparente (*source*), α_s é o *alpha* do objeto, c_d é a cor do pixel antes do *blending* (*destination*), e c_0 é a cor resultante quando coloca-se o objeto transparente sobre a cena. Caso o objeto seja opaco, ou seja $\alpha = 1.0$, então a equação resulta na simples substituição da cor do pixel pela cor do objeto.

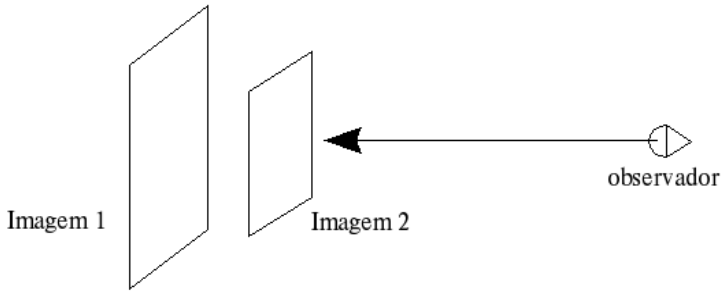
Haines e Möller [16] comentam que para renderizar objetos transparentes em uma cena, usualmente precisa-se de uma ordenação. Primeiro, os objetos opacos são renderizados, então o *blending* é feito pondo-se os objetos transparentes na cena sobre os objetos previamente renderizados, segundo o que se costuma chamar *back-to-front order*. O *blending* em uma ordem arbitrária pode causar sérios artefatos porque a equação de *blending* depende da ordem.

Como exemplo consideremos as duas imagens abaixo, uma com opacidade constante $\alpha = 1$, e outra com opacidade menor, tomemos $\alpha = 0.32$ para uma região e 1 no restante. Quando mapeadas sobre polígonos simples (quadriláteros), pode-se verificar o efeito quando alternamos a ordem de envio dos polígonos para o pipeline de renderização (Figura 3.8).

(a) Imagem com $\alpha = 1$ (imagem 1)

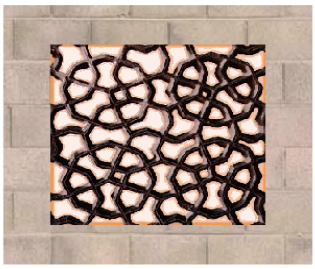


(b) Imagem com $\alpha = 0.32$ (imagem 2)



(c)

(d) Polígono com imagem 1 enviado primeiro



(e) Pólígono com imagem 2 enviado primeiro

Figura 3.8: Transparência no pipeline de renderização

Capítulo 4

Splatting

Embora toda a teoria de amostragem envolvida no renderização por *splatting* não seja aplicada explicitamente no desenvolvimento das idéias desta dissertação, falaremos resumidamente sobre a técnica de *splatting* e a teoria de amostragem envolvida, uma vez que possibilitam o desesenvolvimento de trabalhos futuros.

4.1 Introdução

Splatting é uma técnica que pode ser aplicada para renderizar tanto superfícies como volumes; há bem pouco tempo era usada principalmente para renderização de volumes [32]. Trabalhos recentes como os de Rusinkiewicz e Levoy [28], e Zwicker et al [34], dentre outros, mostraram como *splats* podem ser usados para renderizar superfícies de objetos 3D representadas por pontos.

No *splatting* a superfície de um objeto 3D é representada por uma coleção de amostras, cada amostra é representada por uma pequena superfície planar orientada ao longo da superfície do objeto. Matematicamente a superfície do objeto é representada pela superposição de funções base gaussianas, truncadas para fornecer suporte local.

Os *splats* são usados em várias técnicas de renderização como uma forma de calcular as contribuições de uma amostra para a área que ocupa no espaço da imagem. Do ponto de vista do processamento de sinais, os *splats* são filtros de reconstrução e assumem uma caráter bem geral como primitivos para renderização.

Para a utilização dos *splats* (filtros) na reconstrução de objetos 3D representados

por pontos, tem-se que levar em consideração os atributos associados a cada ponto, como: posição, normal, cor, e uma matriz de variância. A variância serve como uma alternativa para as informações de conectividade, e fornece informações quanto a densidade local (espaçamento entre as amostras), sendo usada para determinar forma e tamanho para os *splats*. Para cada ponto tem-se um filtro de reconstrução orientado, desta forma os *splats* podem ser vistos como amostras de textura espaçadas como uma orientação e tamanho individual. Os trabalho de Zwicker et al [34] e Räsänen [26] são boas referências.

A teoria de amostragem constitui a base para o *splatting*, onde as amostras (pontos) que descrevem uma superfície são reconstruídas, filtradas (filtro passa-baixa), e projetadas para o espaço de cena. Este processo tem muitos pontos em comum com mapeamento de textura. De fato, pode-se considerar mapeamento de textura e *splatting* como processos inversos. No mapeamento determina-se que amostras da textura afetam o pixel atual, enquanto que o *splatting* determina que pixels são afetados pela atual amostra de textura.

Quando utiliza-se o conceito de surfel no *splatting*, assume-se que uma área está associada para cada ponto no espaço do objeto. Esta área é projetada para o espaço de cena, realizando-se a reconstrução da superfície de forma a garantir continuidade através da rasterização de uma aproximação da projeção (elíptica). O surfel projetado será o *splat* usado na reconstrução (Figura 4.1).

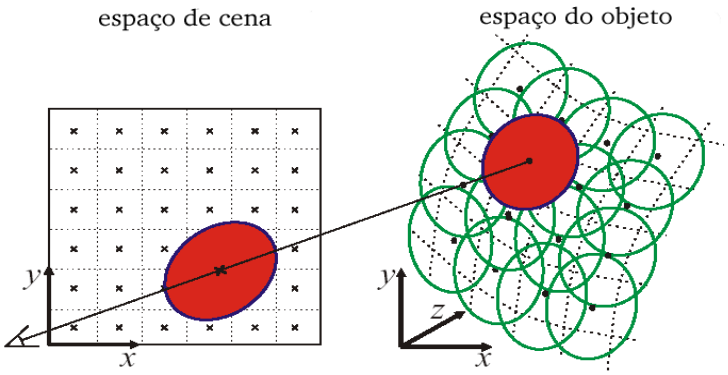
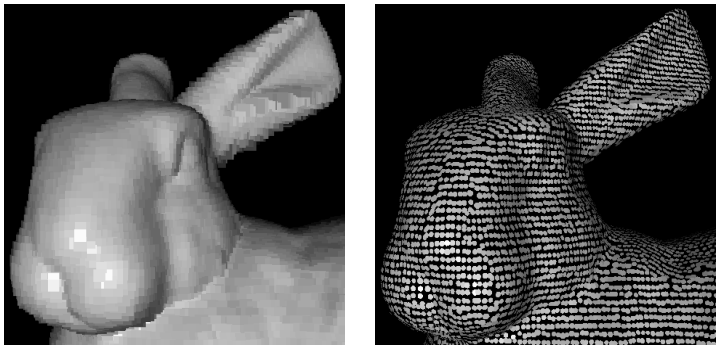


Figura 4.1: Surfel projetado para o espaço de cena

A forma do *splat* depende da rasterização do surfel projetado. Podem-se ter *quads*, que são rápidos de renderizar e com suporte por *hardware* (OpenGL), mas não

se adaptam a orientação da superfície e produzem uma imagem final com baixa qualidade (Figura 4.2(a)); ou elipses, ou ainda círculos. As elipses assemelham-se mais a projeção do surfel (plano tangente circular) e se adaptam a orientação da superfície, produzindo imagens com mais alta qualidade do que os *quads* (Figura 4.2(b)), entretanto são mais lentas de renderizar e não possuem suporte por *hardware*.



(a) *Bunny* com *quads*

(b) *Bunny* com pequenas elipses

Figura 4.2: *Splats* - *quads* e elipses

Para realizar *splatting* pode-se pensar no algoritmo dividido em duas etapas. Na primeira etapa o *splat* é transformado para o espaço de cena e rasterizado. Então os fragmentos do *splat* são armazenados em um *a-buffer* [5]. Na etapa seguinte, para cada pixel resolvem-se as questões relativas a visibilidade, e a partir dos fragmentos armazenados realiza-se uma composição para o cálculo da cor final do pixel.

Algoritmo (Pseudo-código):

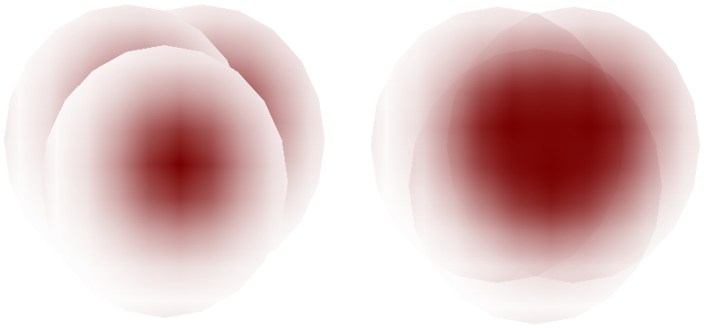
```
1ª etapa:  splatting
Para cada splat
  transformar
  realizar shading
  rasterizar
  Para cada pixel do splat
    armazenar cor, profundidade e peso
  Fim Para
Fim Para

2ª etapa:  splatting
Para cada pixel
  ordenar os fragmentos front to back
  Para cada superfície (amostra)
    achar os fragmentos formando a superfície
    realizar blending da cor da superfície com a cor do pixel
  Fim Para
Fim Para
```

Neste algoritmo o peso representa o valor do filtro de reconstrução para um pixel em particular, e a profundidade é a distância do *splat* com relação a câmera ao longo do eixo perpendicular ao plano de tela. O filtro de reconstrução deve ser amplo o suficiente para cobrir a superfície.

O *Splatting* pode ser usado em muitas aplicações interativas, uma vez que permite um alto grau de detalhes e efeitos que são difíceis de implementar usando renderização de polígonos. Suas vantagens derivam em grande parte do desenvolvimento de técnicas para modelagem e deformação de objetos baseados em pontos; da facilidade para a criação e uso de múltiplos níveis de detalhes; e implementações eficientes em hardware.

4.2 *Fuzzy Splats*

Um defeito comum que independe da forma dos *splats* (*quad*, círculo, elipse, etc) é a ausência de interpolação entre as suas cores, o que possibilita ao observador distinguir a sua forma. Para melhorar a qualidade da imagem final pode-se usar uma máscara para o valor *alpha* que define a opacidade do *splat*, com a opacidade decaindo a partir do seu centro (*fuzzy splat* - Figura 4.3), realizando-se as operaçoes de *blending* entre as cores no espaço de cena de acordo com o valor *alpha*.



(a) Sem *blending* das cores (b) Com *blending* das cores

Figura 4.3: *Fuzzy splats*

A cor c de um pixel na posição (x, y) será dada por uma média ponderada normalizada das contribuições de diferentes *splats*.

$$c(x,y) = \frac{\sum_i c_i w_i(x,y)}{\sum_i w_i(x,y)},$$

onde c_i é a cor do *splat* i e $w_i(x, y)$ o seu peso na posição (x, y). A normalização é necessária uma vez que os pesos não formam necessariamente uma partição da unidade no espaço de cena, devido a valores irregulares para as posições dos surfels e o truncamento relativo a máscara do valor *alpha*.

Quando utilizam-se *fuzzy splats*, precisa-se determinar quando o *blending* será realizado entre um novo fragmento e um fragmento já previamente armazenado no *depth-buffer*. A utilização do *depth-buffer* possibilita verificar para cada pixel se o novo fragmento está na frente, visível; ou atrás, invisível, o que geralmente permite analisar quando dois fragmentos (novo + armazenado) pertencem a uma mesma

região da superfície, aonde deverá ser feito o *blending*.

Em 2001, Zwicker [34] usou um limiar para verificar a habilitação do *blending*. Se a diferença entre as coordenadas z (profundidade com relação a câmera) de dois fragmentos era menor do que o limiar estabelecido, então realizava-se o *blending*, caso contrário efetuava-se o teste convencional de visibilidade com o *z-buffer*.

Uma outra extensão da utilização do *z-buffer* está baseada no estabelecimento de faixas de valores de z para os *splats*. A faixa de valores pode ser calculada a partir de um valor mínimo e máximo para z no espaço de câmera, sendo que a fórmula para a determinação destes valores pode ser encontrada no trabalho de Räsänen [26]. Uma forma mais simples seria considerar a faixa $(z - r, z + r)$, onde z é a coordenada do centro do *splat* no espaço de câmera, e r o seu raio. Então, se as faixas de valores do novo fragmento e do fragmento no *z-buffer* se sobrepõem no teste de profundidade, efetua-se o *blending*, e a nova faixa de valores é fixada como sendo a união das duas faixas.

Quando faz-se referência aos valores de z no espaço de câmera, está se considerando o z antes da projeção perspectiva, uma vez que após a projeção não seria tão simples determinar se dois fragmentos estão suficientemente próximos. Ambos os métodos descritos acima não têm suporte por *hardware*. No trabalho de Rusinkiewicz e Levoy [28] há uma sugestão de como simular o *z-offset* usando OpenGL através de um algoritmo de renderização em duas etapas.

4.3 Reconstrução de um objeto utilizando splats

Como mencionado anteriormente, no *splatting* a reconstrução da superfície de um objeto é feita utilizando-se filtros de reconstrução planares. A reconstrução é feita no espaço do objeto, e para obter-se a imagem em perspectiva transformam-se os filtros de reconstrução para o espaço de cena.

Pode-se pensar em linhas bem mais gerais quando considera-se a renderização com a utilização de um filtro de reconstrução. A renderização consiste em um processo de amostragem-reamostragem, e a teoria de amostragem nos permite trabalhar as questões relativas a reconstrução de sinais. Uma imagem é um sinal 2D contínuo representado por amostras discretas, onde o espaço de cena corresponde a uma grade de amostragem, tendo-se que achar uma cor para cada pixel através das amostras

obtidas em uma cena. Quando a descrição da cena é dada através de amostras como é o caso do *splatting* e do mapeamento de textura, necessitamos reamostrar os dados de entrada.

Reamostrar envolve a reconstrução de uma representação contínua dos dados de entrada com um filtro de reconstrução apropriado, transformando a representação do espaço de entrada para o espaço de saída, limitando-se a largura de banda (restringindo o suporte local), e usando um pré-filtro para evitar *aliasing*. Faz-se então a amostragem dessa representação para produzir um sinal de saída discreto. No contexto do *splatting*, o espaço de entrada é o espaço do objeto onde os *splats* são definidos, e o espaço de saída o espaço de cena.

O processo de reamostragem descrito em Heckbert [18] pode ser descrito como segue:

$\boldsymbol{U}$: conjunto de todas as posições das amostras de entrada, ou seja, posições no espaço do objeto
$\boldsymbol{X}$: conjunto de todas as posições das amostras no espaço de saída
$\boldsymbol{r}$: filtro de reconstrução
$\boldsymbol{f_c}$: função que descreve a cor para uma dada amostra de entrada
$\boldsymbol{h}$: pré-filtro
$\boldsymbol{M}$: representa uma transformação geral, onde $x = M(u)$.
Amostras de entrada discretas: $f(u_k), u_k \in U$
Entrada reconstruída: $f_c(u) = f(u_k) \otimes r_k(u) = \sum_{u_k} f(u_k) r(u - u_k)$
Mapeamento: $g_c(x) = f_c(M^{-1}(x))$
Saída contínua: $g_c{}'(x) = g_c(x) \otimes h(x) = \int_{\mathbb{R}^n} g_c(t) \cdot h(x - t)\mathrm{d}t$
Amostras de saída discretas: $g(x_0) = g_c{}'(x), x_0 \in X$

Agora se expandirmos a fórmula para $g(x_0)$ com $x_0 \in X$, obteremos:

$$\begin{aligned} g(x_0) &= g_c{}'(x_0) = \int_{\mathbb{R}^n} f_c(M^{-1}(t)) \cdot h(x_0 - t)\mathrm{d}t \\ &= \int_{\mathbb{R}^n} h(x_0 - t) \cdot \sum_{u_k} f(u_k) r(M^{-1}(t) - u_k)\mathrm{d}t \\ &= \sum_{u_k} f(u_k) \cdot \rho(x_0, u_k) \end{aligned}$$

Onde $\rho(x_0, u_k) = \int_{\mathbb{R}^n} h(x_0 - t) \cdot r(M^{-1}(t) - u_k)\mathrm{d}t$, é o filtro de reamostragem.

A forma do filtro de reamostragem em geral depende de x_0 e u_k. A fórmula para $g(x_0)$ afirma que a cor para cada posição x no espaço de saída é uma soma das cores das amostras de entrada $f(u_k)$ ponderada pelo filtro de reamostragem. Para o mapeamento de textura isso significa que com o intuito de reamostrar uma cor para cada pixel, temos que somar as amostras de textura reconstruídas sob a área do pré-filtro. Para o *splatting*, isto significa que pode-se acumular sobre a cena as contribuições de todas as amostras de entrada, onde a área de influência de cada amostra é a área do filtro de reamostragem na posição da amostra.

O filtro de reconstrução sozinho pode não ser suficiente para os pontos de uma amostra. No caso do *splatting* isto significa que um filtro de reconstrução pequeno poderia não ser completamente desenhado (rasterizado), resultando em *aliasing*. Com a combinação (convolução) do filtro de reconstrução com um pré-filtro garante-se um tamanho mínimo para o filtro de reamostragem, de modo que seja largo o suficiente para ser desenhado (rasterizado). A convolução do filtro de reconstrução e do pré-filtro pode ser interpretada como uma varredura do filtro de reconstrução sobre a área do pré-filtro. Assim cada amostra é reconstruída com um filtro de reconstrução combinado com um pré-filtro para formar o filtro de reamostragem. Os *splats* são os núcleos dos filtros de reamostragem.

Para reconstruir a superfície, cada amostra é projetada para o espaço de cena, colocando-se um *splat* (orientado) na posição correspondente, e combinando-o com um pré-filtro para formar o filtro de reamostragem, que é então rasterizado. Os valores das amostras rasterizadas são acumulados dentro do *frame-buffer*. Depois que todas as amostras são acumuladas, começa-se a reconstrução da superfície em cada pixel analisando-se as questões relativas a visibilidade. Para o *splatting* geralmente são utilizados núcleos de reconstrução planares, desta forma a curvatura local próxima a um ponto da amostra deve ter um valor preferencialmente baixo. Além disso, como mencionado anteriormente, a projeção de um surfel para o espaço de cena gera melhores resultados quando a forma da projeção assemelha-se a uma elipse, por isso, com o intuito de produzir imagens de mais alta qualidade como no trabalho de Zwicker [34], geralmente usam-se núcleos de filtros elípticos (Heckbert´s EWA filter) [18] na reconstrução da superfície de um objeto.

Para um objeto representado por um conjunto de pontos p_k pode se definir uma função de textura, associando-se um filtro r_k para cada ponto e coeficientes w_k^r, w_k^g, w_k^b para os canais de cor. Sem perda de generalidade consideremos um simples canal w_k para a discussão do EWA. Uma parametrização local da superfície é suficiente

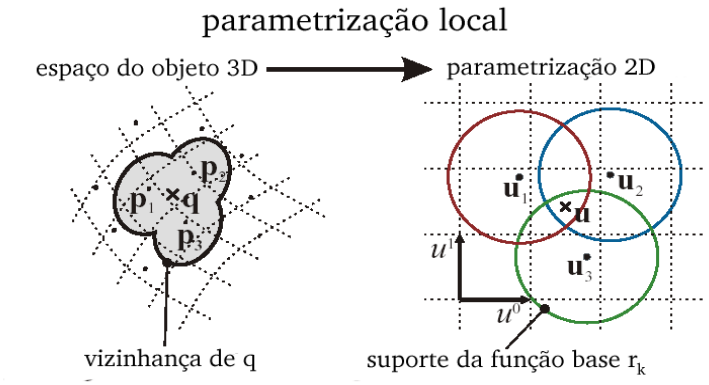


Figura 4.4: Parametrização local para um ponto q da amostra

para definir uma função de textura, uma vez que o suporte (pontos nos quais os valores da função são não-nulos) para as funções r_k é local (gaussianas truncadas). Então dado um ponto q sobre a superfície com coordenada local u (coordenadas na parametrização local) (Figura 4.4), o valor da função de textura contínua é expresso por:

$$f_c(u) = \sum_{k \in \aleph} w_k \cdot r_k(u - u_k),$$

onde u_k é a coordenada local do ponto p_k, e o valor $f_c(u)$ dá o valor da cor do ponto q.

Desta forma para renderizar-se um objeto representado por pontos de acordo com a equação acima, a função de textura f_c tem que ser mapeada para o espaço de cena. Segundo Heckbert [17] podemos seguir os seguintes passos:

1. Primeiro a função de textura f_c no espaço do objeto é reconstruída a partir das amostras usando a equação acima.
2. A função f_c é então mapeada para o espaço de cena usando uma aproximação afim para a transformação espaço do objeto-espaço de cena, então a função mapeada f_c é multiplicada (convolução no espaço de cena) por um pré-filtro h, resultando na saída em uma função com largura de banda limitada $g_c(x)$.
3. A função $g_c(x)$ é então amostrada para gerar as cores dos pixels sem *aliasing*.

A partir dos 3 passos acima teremos:

$$g_c(x) = \sum_{k \in \aleph} w_k \cdot \rho_k(x), \quad \text{sendo} \quad \rho_k(x) = (r_k{}' \otimes h)(x - m_{u_k}(u_k)),$$

onde $r_k{}'$ é a função r_k mapeada, h é o pré-filtro, m_{u_k} é a aproximação afim do espaço do objeto para o espaço de cena em torno do ponto u_k e $\otimes$ denota convolução. As coordenadas x são usadas para designar as coordenadas no espaço de cena. A função ρ_k é a função r_k filtrada e mapeada, e é chamada de núcleo de reamostragem. A primeira das duas equações acima indica que a função de textura com largura de banda limitada no espaço de cena, pode ser renderizada primeiro mapeando e limitando a largura de banda da função r_k individualmente, e depois somando-as no espaço de cena.

A função r_k mapeada, no contexto do EWA, representa a projeção do surfel para o espaço de cena. A convolução com o pré-filtro (limitação da largura de banda) torna o *splat* maior, portanto o núcleo de reamostragem é basicamente um *splat* aumentado.

A estrutura do EWA usa gaussianas elípticas para os filtros de reconstrução r e o pré-filtro h. Com gaussianas é possível expressar o núcleo de reamostragem (*splat*) em uma forma fechada representada por uma simples gaussiana elíptica. Uma gaussiana elíptica 2D com matriz de variância V é definida como:

$$G_V(x) = \frac{1}{2\pi\sqrt{|V|}} e^{\frac{-1}{2}x^T V^{-1} x},$$

onde $|V|$ é o determinante de V. A matriz V^{-1} é chamada matriz cônica e $x^T V^{-1} x = const$ são os isocontornos da gaussiana G_V, os quais são elipses se, e somente se, V é definida positiva.

As matrizes de covariânca para os filtros r_k e o pré-filtro h são denotadas por V_k^r, V^h respectivamente (geralmente $V^h = I$); com gaussianas teremos:

$$\rho_k(x) = \frac{1}{\left|J_k^{-1}\right|} G_{J_k V_k^r J_k^T + I}(x - m(u_k))$$

Nesta formulação $\rho_k(x)$ é uma gaussiana, denominada núcleo de reamostragem EWA. J_k denota o Jacobiano do mapeamento m em u_k do espaço do objeto para o espaço de cena. Para um surfel circular com raio r, a matriz de variância será dada por:

$$\boldsymbol{V_k^r} = \begin{pmatrix} 1/r^2 & 0 \\ 0 & 1/r^2 \end{pmatrix}$$

Anteriormente falou-se sobre o fato do filtro de reconstrução sozinho não ser suficiente para a renderização do *splat*, necessitando-se da combinação (convolução) com um pré-filtro. O EWA fornece uma solução satisfatória. Quando o filtro de reconstrução mapeado é pequeno, o pré-filtro h é dominante e o núcleo de reamostragem resultante (*splat*) é aproximadamente circular (*splats* circulares), caso contrário, o pré-filtro é pequeno quando comparado com o filtro mapeado e assim não afeta de forma significativa sua forma elíptica (*splats* elípticos) (Figura 4.5).

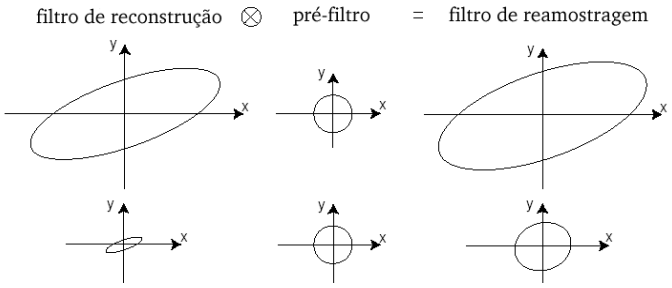


Figura 4.5: convolução do filtro mapeado com pré-filtro

Capítulo 5

Visualização de Nuvens de Pontos com Blending de Textura

Neste capítulo propomos uma técnica para visualização de superfícies representadas por nuvens de pontos baseada principalmente nas idéias de Schauffer e Jensen [19], Pfister e Zwicker [25], Rusinkiewicz e Levoy [28]. A principal contribuição desse trabalho é a utilização de novos surfels para representar a superfície a ser visualizada na vizinhança de cada ponto.

5.1 Introdução

Dada uma nuvem de pontos $P = \{p_i \in \mathbb{R}^3\}$, deseja-se definir uma aproximação para a superfície S definida pelos pontos $p_i \in P$. No contexto dessa dissertação fazemos as suposições a seguir:

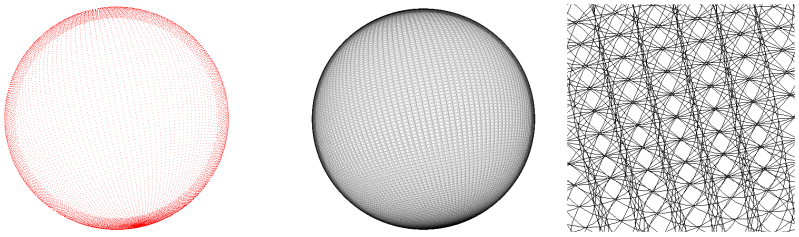
Densidade: Os modelos possuem alta densidade, tal como as obtidas a partir de sistemas de escaneamento 3D. A distribuição é aproximadamente uniforme, de modo que o espaço entre os pontos é suficiente para garantir uma reconstrução contínua da superfície sem informações explícitas de conectividade.

Suavidade: Os pontos estão sobre uma superfície S suficientemente suave, sendo que nas regiões de alta curvatura a alta densidade permite a reconstrução com um bom grau de aproximação.

Normais: Uma normal é designada para cada ponto.

Coordenadas de textura: Quando possível usamos coordenadas de textura para os pontos do modelo.

A partir da definição de Kobbelt [4] para *splatting*, e do paradigma dos elementos de superfície (surfels) de Pfister e Zwicker [25], considera-se cada ponto como um surfel (Figura 5.1), e por meio dessa associação faz-se uma aproximação do plano tangete à superfície em cada ponto. Cada aproximação tem como objetivo representar uma região da superfície que corresponde a área associada com cada surfel.



(a) (b) (c)

Figura 5.1: (a) pontos sobre a esfera. (b) surfels sobre a esfera. (c) sobreposição dos surfels.

5.2 Escolha dos Surfels

A maioria dos pesquisadores propõem o uso de surfels planos (discos circulares) para a representação dos pontos sobre a superfície de um objeto 3D. Nesta seção descrevemos uma proposta alternativa baseada no uso de um surfel com curvatura para a aproximação do plano tangente em cada ponto.

Segundo Gomes e Velho [30] uma superfície topológica é um subconjunto S do espaço euclidiano $\mathbb{R}^3$ que é localmente homeomorfo ao plano euclidiano $\mathbb{R}^2$, ou seja, para cada ponto $p \in S$ existe uma vizinhança esférica $B^3_\varepsilon \subset \mathbb{R}^3$ com centro em p, tal que o conjunto $B^3_\varepsilon \cap S$ é homeomorfo ao disco aberto unitário

$$B^2_1 = \left\{(x, y) \in \mathbb{R}^2; x^2 + y^2 < 1\right\}$$

do plano euclidiano (Figura 5.2). Intuitivamente esta definição diz que uma superfície é obtida colando pedaços deformados do plano.

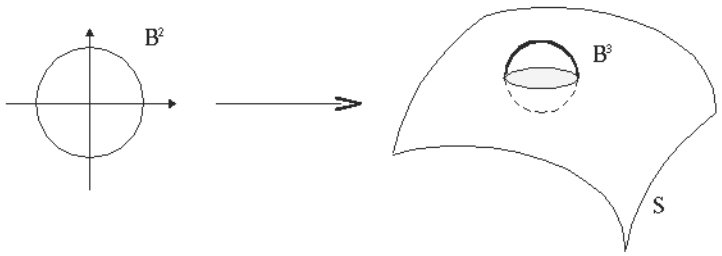
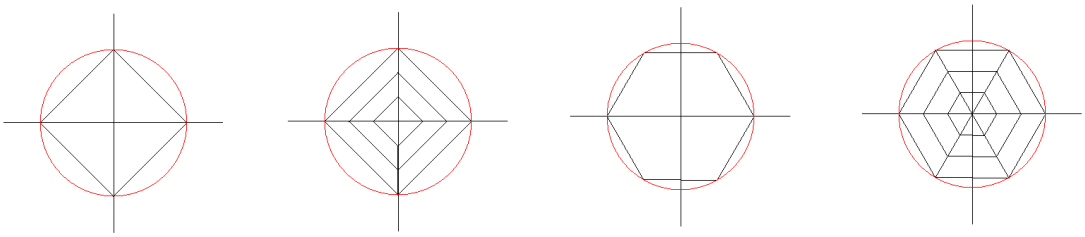


Figura 5.2: Definição de uma superfície

Considerando o plano euclidiano $\mathbb{R}^2$, podemos definir polígonos regulares inscritos no disco unitário (Figura 5.3).



(a) polígono com 4 lados

(b) polígono-malha com 4 lados

(c) polígono com 6 lados

(d) polígono-malha com 6 lados

Figura 5.3:

Dados os polígonos regulares inscritos no disco unitário, podemos usá-los para representar intuitivamente o homeomorfismo do conjunto $B^3_\varepsilon \cap S$ e o disco aberto unitário. Dependendo do ε adotado para a vizinhança esférica, a interseção pode ser pensada como uma aproximação do plano tangente no ponto p (surfel).

Vamos definir cada polígono pela função $f(\varepsilon, \theta) = (\varepsilon \cdot \cos\theta, \varepsilon \cdot \sin\theta, \varphi(\varepsilon))$ ($\theta \in [0, 2\pi]$), onde a função φ é dada por:

$$\varphi(\varepsilon) = \begin{cases} 0 & \text{, para uma aproximação planar ou} \\ 1 - e^{-\left(\frac{\varepsilon^2}{c^2}\right)^2} & \text{, para uma aproximação quase planar} \end{cases}$$

Quando empregamos o termo "**quase planar**"estamos fazendo referência a uma superfície levemente curva, sendo que tal curvatura definida a partir do valor c não impede a sua utilização para a aproximação local da superfície. O uso deste primitivo permite intuir uma melhor adaptação do surfel à superfície do objeto (Figura 5.4), e por uma questão de praticidade vamos chamá-lo c-surfel.

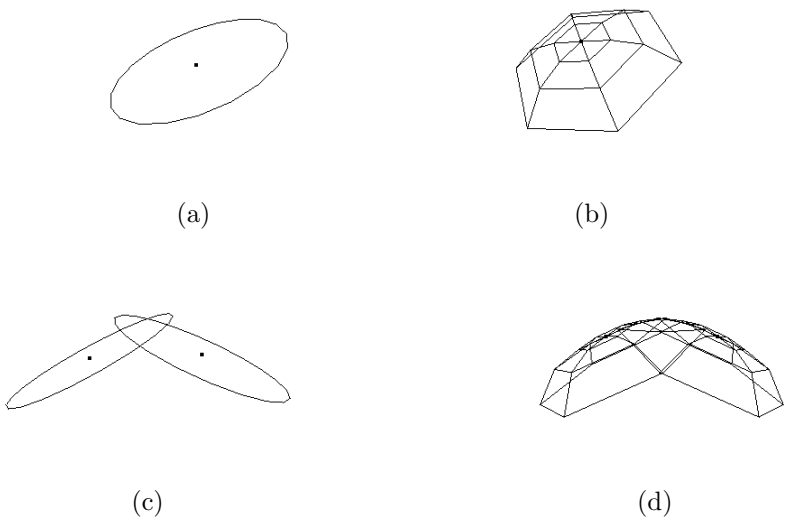


Figura 5.4: (a) surfel plano. (b) surfel com curvatura (c-surfel). (c) interseção de surfels planos. (d) interseção dos c-surfels.

De acordo com o paradigma dos quatro universos definido por Gomes e Velho [30], para o estudo de um determinado fenômeno ou objeto do mundo real no computador, associamos ao mesmo um modelo matemático, e em seguida procuramos uma representação finita desse modelo. Qual seria então uma boa representação finita para o conceito de aberto? Utilizou-se aqui o conceito de *fuzzy splats* tal como definido no capítulo anterior, o que nos leva à aplicação de textura e operações de *blending*.

5.3 Mapeamento de Textura e *Blending*

O mapeamento de textura permite modificar as propriedades ao longo da superfície de um objeto sem a necessidade de modificações em sua geometria, podendo afetar atributos tais como: cor, especularidade, transparência, normais para cálculos de iluminação e etc. O uso mais comum é o mapeamento de uma imagem para uma

superfície 3D, que é então mapeada para o espaço de cena.

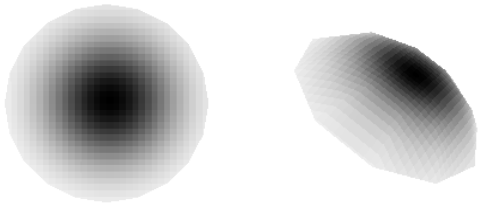
Para cada surfel utilzado na reconstrução, mapeia-se uma textura composta de uma única cor, e opacidade decaindo radialmente de acordo com uma aproximação gaussiana. O valor *alpha* (α) correspondente a um pixel (i, j) na imagem de textura é dado por $\alpha = \alpha \cdot f(i, j)$ e:

$$f(i,j) = e^{-\left(\frac{(i-x_0)^2+(j-y_0)^2}{h^2}\right)}$$

onde,

(i, j)	- posição na textura
(x_0, y_0)	- centro da imagem de textura
h	- fator de decaimento

No capítulo 3 foi dado um exemplo do mapeamento de textura para um polígono regular sobre o plano XOY centrado na origem, que corresponde exatamente ao polígono regular inscrito no disco unitário mencionado anteriormente, podendo-se usar as mesmas equações para determinar as coordenadas de textura uv ($[0, 1]\times[0, 1]$). Desta forma o centro do polígono coincide com o centro da textura, obtendo-se a transparência na borda do surfel (ou do c-surfel) (Figura 5.5).



(a) (b)

Figura 5.5: (a) surfel plano com textura. (b) c-surfel com textura

Quando os surfels (c-surfels) se sobrepõem, realizamos o *blending* entre as texturas mapeadas tal como descrito na capítulo 3, ou seja, considerando que a cor final será dada por:

$$c_0 = \alpha \cdot c_s + (1 - \alpha) \cdot c_d$$

onde,

c_s - cor do objeto transparente;
c_d - cor do pixel antes do *blending*;
c_s - cor resultante quando coloca-se o objeto transparente na cena.

5.4 Tamanho dos surfels

O objetivo deste trabalho é a reconstrução aproximada de uma superfície baseada em pontos, portanto necessita-se determinar um tamanho adequado para os surfels utilizados na renderização. Se considerarmos a função $f(\varepsilon, \theta) = (\varepsilon \cdot \cos\theta, \varepsilon \cdot \sin\theta, \varphi(\varepsilon))$ que define os surfels, então temos que calcular um valor ε que garanta uma reconstrução livre de buracos.

Rusinkiewicz e Levoy [28] no programa Qsplat usam como dado de entrada uma malha inicial de triângulos e determinam o tamanho de cada *splat* que representa um vértice (ponto), o qual corresponde ao tamanho da esfera envolvente de todos os triângulos que tocam aquele vértice. Partindo desta idéia, porém sem uma malha inicial e usando só uma nuvem de pontos (+normais) $P = \{p_i \in \mathbb{R}^3\}$ com uma distribuição aproximadamente uniforme, podemos calcular um tamanho para os nossos surfels estimando propriedades locais da superfície, através da Análise de Componentes Principais (Pauly [13]).

Dada uma nuvem de pontos P, a matriz de covariância para a vizinhança de um ponto p será dada por:

$$\mathbf{C} = \begin{bmatrix} p_{i_1} - \overline{p} \\ \cdots \\ p_{i_k} - \overline{p} \end{bmatrix}^T \cdot \begin{bmatrix} p_{i_1} - \overline{p} \\ \cdots \\ p_{i_k} - \overline{p} \end{bmatrix} \quad , i_j \in N_p,$$

onde $\overline{p}$ é o centróide dos vizinhos p_{i_j} do ponto p, e N_p é o conjunto de índices dos k vizinhos mais próximos de p. As componentes principais são as soluções para o problema de autovetores (autovalores):

$$C \cdot v_l = \lambda_l \cdot v_l \quad , l \in \{0, 1, 2\}\,.$$

A matriz C é simétrica definida positiva e portanto seus autovalores serão números reais com os autovetores correspondentes ortogonais dois-a-dois, ou seja, tem-se um sistema de coordenadas local para a vizinhança N_p, além disso cada autovalor λ_l

representa a variação dos pontos p_{i_j} ao longo da direção do respectivo autovetor v_l.

Dada uma vizinhança de tamanho n do ponto p, e assumindo que os autovalores para esta vizinhança são dados por $\lambda_0 \leq \lambda_1 \leq \lambda_2$, tem-se que o autovalor λ_0 descreve a variação ao longo da normal, o que permite a definição de **variação da superfície no ponto** p, que é dada por:

$$\sigma_n(p) = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2}$$

Se $\sigma_n(p)$ é aproxidamente zero podemos considerar que os pontos na vizinhança estão praticamente sobre o mesmo plano. Usando esta informação podemos realizar um particionamento do espaço com uma estrutura de dados conveniente, e realizar um agrupamento (*clustering*) hierárquico dos pontos sobre a superfície.

5.4.1 Agrupamento Hierárquico (*BSP - Binary Space Partition*)

O agrupamento hierárquico divide a nuvem de pontos em subregiões menores, de modo que obtemos um conjunto $\{C_i\}$ de agrupamentos (*clusters*). Podemos calcular este conjunto dividindo a nuvem de pontos recursivamente usando uma BSP. A nuvem de pontos $P = \{p_i \in \mathbb{R}^3\}$ é dividida se a variação $\sigma_n(p)$ está acima de uma valor pré-estabelecido (próximo de zero).

O plano de divisão para a BSP é definido pelo centróide de P e o autovetor v_2 da matriz de covariância que corresponde ao maior autovalor, deste modo a nuvem de pontos sempre é dividida ao longo da direção de maior variação. Se os critérios de divisão não são satisfeitos a nuvem de pontos P torna-se um *cluster*.

Considerando apenas os *clusters* com 3 pontos não-colineares, determina-se o diâmetro da esfera envolvente do triângulo definido por estes pontos. Como estamos assumindo que a distribuição é aproximadamente uniforme, os triângulos formados na sua maioria terão praticamente a mesma área (Figura 5.6). Assim faz-se uma média aritmética simples para os valores (diâmetro da esfera) dos *clusters* considerados, e adota-se esse valor como o tamanho dos surfels.

Figura 5.6: Triângulos sobre a superfície obtidos a partir do agrupamento (BSP)

Se calcularmos os raios de acordo com a densidade dos pontos nos *clusters* [22], podemos ter problemas com a transparência dos surfels (Figura 5.7). Nesta dissertação não estamos tratando este caso; possivelmente será necessário um ajuste no fator de decaimento da opacidade de cada surfel baseado na densidade.

(a) raio-densidade (esfera 16.471 pontos). (b) raio-fixo (esfera 16.471 pontos).

Figura 5.7:

5.5 Visibilidade

Como mencionado anteriormente, dado um conjunto de polígonos com transparência, a ordem com que são enviados para o *pipeline* de renderização afeta o *blending* e a oclusão (Figura 5.8). Então, antes de rasterizarmos os surfels ordenamos os pontos, utilizamos uma renderização em dois passos como proposto por Rusinkiewicz e Levoy.

Rusinkiewicz e Levoy em 2000 [28] propuseram uma renderização em dois passos usando OpenGL. Para assegurar que tanto o *blending* quanto a oclusão sejam

Figura 5.8: Toro com erro de oclusão para os cálculos de *blending*

realizados corretamente, primeiro renderizamos no *depth-buffer* os surfels com um deslocamento z_0 com relação ao observador (Figura 5.9); depois sem o deslocamento, mas permitindo o teste de profundidade sem atualização do *depth-buffer* e escrevendo para o *color-buffer*. Estes passos permitem realizar o *blending* entre os surfels que estão dentro de uma faixa de valores z_0 da superfície.

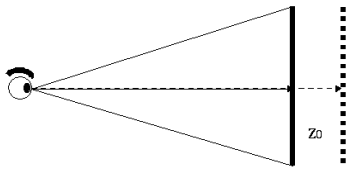
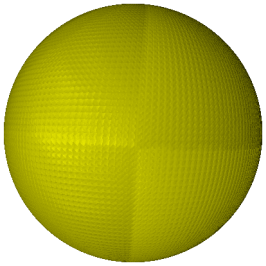


Figura 5.9: deslocamento z_0 do observador

5.6 Reconstruindo com os surfels

Para cada ponto temos um surfel associado, os quais são orientados, transladados e redimensionados de acordo com os atributos dos pontos. Alguns fatores contribuem para a obtenção de melhores resultados quando utilizamos os c-surfels, dentre os quais podemos destacar:

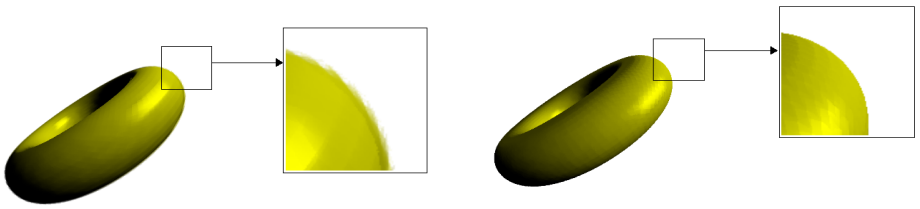
- A interseção entre os c-surfels permite uma melhor aproximação local para os pontos sobre a superfície (Figura 5.4(d)), considerando que o espaço entre os pontos permite perceptualmente visualizar a interseção dos c-surfels como que aproximadamente sobre a superfície;



normais variando ao longo do c-surfel

Figura 5.10:

- Como os c-surfels tem uma malha base, a interseção entre eles permite visualizar uma determinada região com mais detalhes;
- Como todas as normais dos vértices da malha base apontam na mesma direção (direção da normal no ponto), os cálculos de iluminação juntamente com as operações de *blending* permitem a visualização de forma contínua da superfície. As normais variando ao longo do c-surfel causam artefatos que prejudicam a continuidade (Figura 5.10);
- Se a quantidade de pontos não é suficientemente densa, ainda assim podemos obter bons resultados, especialmente com relação a silhueta quando optamos pelo uso exclusivo dos c-surfels (Figura 5.11).



(a) (b)

Figura 5.11: (a) surfels planos na silhueta (6.232 pontos). (b) c-surfels na silhueta (6.232 pontos).

Uma desvantagem com relação ao uso dos c-surfels provém do maior número de polígonos que o compõe, portanto o custo computacional durante a sua utilização é

maior do que o de um polígono simples.

Uma das preocupações deste trabalho foi melhorar a silhueta dos objetos renderizados. Embora seja possível utilizar somente os surfels planos, ou só c-surfels, procurou-se aliar as vantagens de ambos: a simplicidade de um polígno simples, e o maior nível de detalhes e adaptação da malha base dos c-surfels. Assim, usam-se os c-surfels na silhueta dos objetos e os planos no restante. Quando o ângulo entre o vetor de visualização (observador-ponto) e a normal no ponto está em um intervalo $[90^o - \varepsilon, 90^o + \varepsilon]$ ($\varepsilon \in [0^o, 20^o]$), utilizamos um c-surfel neste ponto. Caso contrário, utilizamos um surfel plano caso contrário (Figura 5.12).

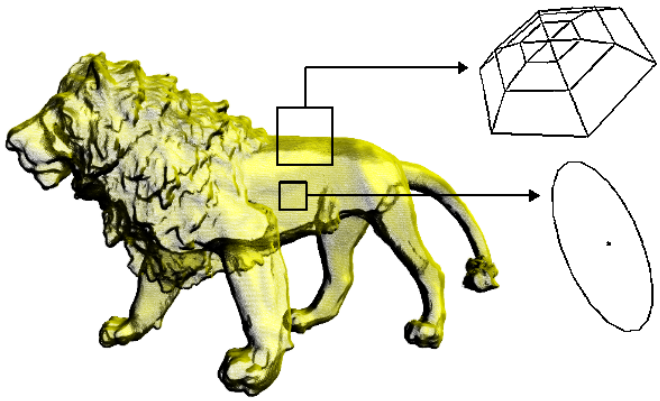


Figura 5.12: Uso de surfels e c-surfels

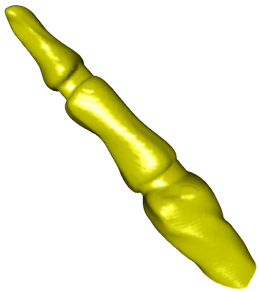
O tamanho calculado para os surfels mostrou-se satisfatório para todos os modelos testados (Figura 5.13) de acordo com os critérios mencionados no início do capítulo, permitindo uma reconstrução contínua da superfície sem a presença de buracos.

5.7 Mapeando textura com surfels

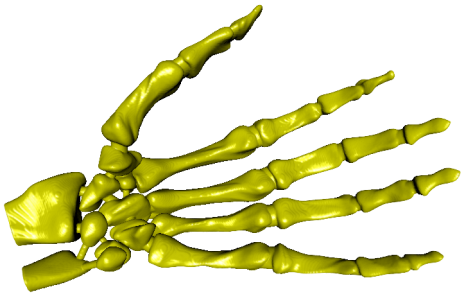
Quando dispomos das coordenadas de textura para os pontos da superfície do objeto 3D, podemos mapear uma imagem de textura utilizando os surfels, juntamente com o *blending* entre as cores da imagem a ser mapeada. Dadas as coordenadas de textura u e v no intervalo $[0, 1]$, determinamos a cor correspondente ao texel (pixel na imagem de textura) na posição (i, j), sendo:

$$i = (\text{largura da imagem}) * u,$$
$$j = (\text{altura da imagem}) * v.$$

Esta cor é mapeada para o surfel correspondente ao ponto com coordenadas (u, v), com o fator *alpha* decaindo do centro para a borda, como descrito anteriormente. Na figura 5.14 mostramos alguns exemplos de mapeamento de textura para o toro e para a esfera.



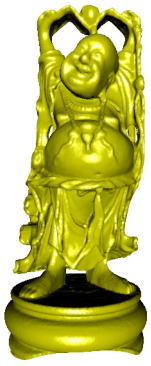
(a) dedo (24.095 pontos)



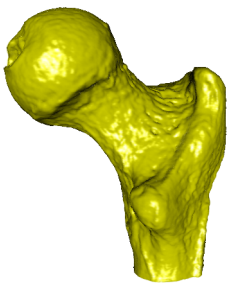
(b) mao (327.323 pontos)

(c) igea (134.345 pontos)

(d) dragon (406.687 pontos)

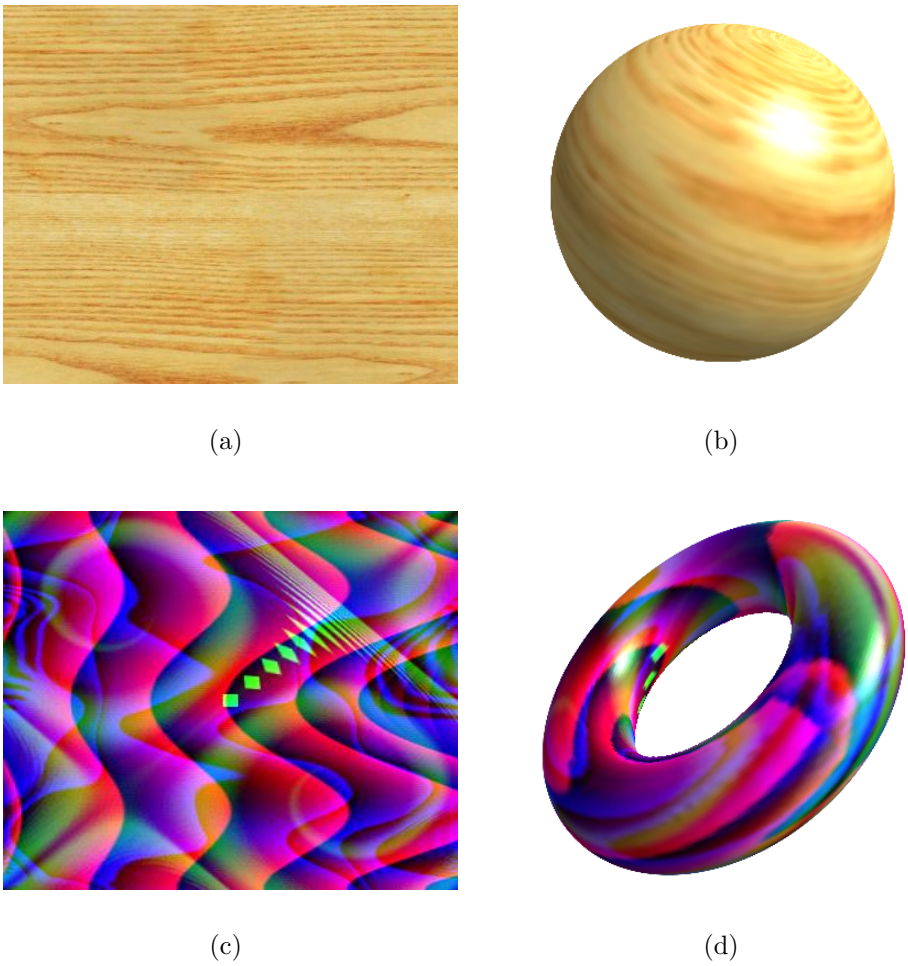


(e) budha (389.347 pontos)



(f) balljoint (137.059 pontos)

Figura 5.13:



(a) (b)

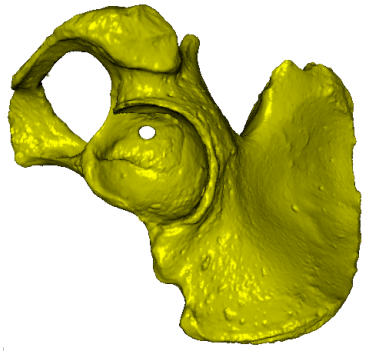
(c) (d)

Figura 5.14: (a),(c) imagens de textura. (b) esfera com textura mapeada. (d) toro com textura mapeada.

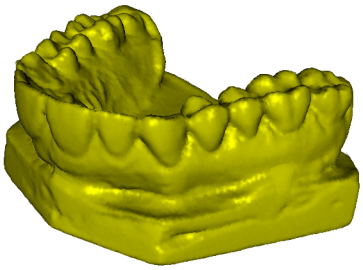
5.8 Usando o Qsplat

Com o intuito de visualizar modelos da ordem de milhões de pontos com um alto grau de interatividade, parte da implementação feita para definição, construção, e posicionamento dos surfels usada nesta dissertação foi incorporada ao código do programa Qsplat de Rusinkiewicz e Levoy [28], permitindo usar as otimizações de sua estrutura hierárquica de esferas envolventes.

O tamanho dos surfels é o mesmo calculado para os *splats* pelo programa (Qsplat) tal como descrito no início da seção 5.4 com o uso de uma malha inicial. A reconstrução não apresentou buracos. Alguns resultados são mostrados abaixo.



(a)



(b)

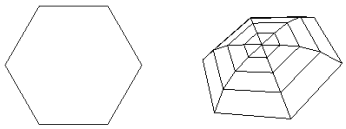
Figura 5.15: (a) osso do quadril (530.168 pontos). (b) dentes (116.604 pontos).

5.9 Desempenho

Mostra-se na tabela abaixo algumas medidas de tempo para os modelos renderizados. Os tempos foram calculados para a renderização com os surfels planos; com os c-surfels, e usando os dois tipos de surfels sem nenhuma otimização a não ser *backface culling*.

Modelo	surfel plano	c-surfel	surfel plano + c-surfel
Igea (134.345 pontos)	$1.22fps$	$1.06fps$	$1.18fps$
Balljoint (137.059 pontos)	$1.08fps$	$0.96fps$	$1.03fps$
Budha (389.347 pontos)	$0.35fps$	$0.3fps$	$0.33fps$

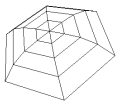
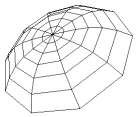
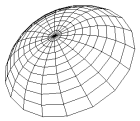
Pelos valores calculados percebe-se que embora os c-surfels sejam formados por mais de um polígono, o custo computacional não aumentou consideravelmente quando comparado com o uso de polígonos simples. Os primitivos usados foram os da figura abaixo (Fig 5.16).



(a) (b)

Figura 5.16: (a) surfel plano (6 lados). (b) c-surfel (6 lados).

Entretanto, se aumentarmos a quantidade de polígonos que formam os c-surfels percebe-se claramente o aumento do custo no processo de visualização. A tabela baixo mostra algumas medidas de tempo usando somente os c-surfels com diferentes resoluções.

c-surfel	 24 polígonos	 60 polígonos	 200 polígonos
Igea (134.345 pontos)	$1.06 fps$	$1.00 fps$	$0.41 fps$

Quando usamos os dois tipos de primitivos, com os c-surfels só na silhueta dos objetos, o tempo calculado não aumentou consideravelmente quando comparado com o uso exclusivo de polígonos simples. Assim os c-surfels são uma boa opção para melhorar as silhuetas dos objetos renderizados. Novamente destacando que a resolução dos c-surfels influencia no custo final da renderização.

De um modo geral, o mapeamento de textura e a renderização em dois passos tornam a renderização mais lenta, uma vez que o objeto precisa ser renderizado duas vezes; uma no *depth-buffer* e posteriormente no *color-buffer*.

Capítulo 6

Conclusões e Trabalhos futuros

A implementação das idéias apresentadas ao longo deste trabalho foi feita usando linguagem C e a biblioteca gráfica OpenGL no ambiente LINUX, em um Pentium IV, com 512 MB de memória RAM. As únicas otimizações feitas para a visualização foram com relação ao uso de *backface culling* e *frustum culling*. Claramente tais otimizações não são suficientes para uma visualização rápida. Como alternativa usamos parte da implementação das idéias da dissertação no programa Qsplat obtendo maior velocidade no processo de renderização, porém sem utilizar a métrica proposta para o cálculo do tamanho dos surfels, deixando esta implementação como um trabalho futuro.

O principal foco desta dissertação foi a reconstrução aproximada de superfícies representadas por pontos. Nosso intuito era obter uma forma simples de renderização sem informações de conectividade, permitindo através de aproximações do plano tangente em cada ponto uma reconstrução suave e contínua da superfície. Para isso:

- Utilizamos o paradigma dos elementos de superfície (surfels), propondo o uso de um surfel com curvatura visando melhorar a adaptação e interpolação dos surfels sobre a superfície, assim como a qualidade da silhueta de objetos suaves com poucos pontos.
- Usamos Análise de Componentes Principais para efetuar o particionamento do espaço (BSP) e calcular o tamanho dos surfels usados na renderização.
- Usando texturas com uma única cor e opacidade decaindo radialmente, definimos cor e transparência para cada surfel, e com operações de *blending* do OpenGL, efetuamos a interpolação dos surfels que se sobrepõem.

- Usando as coordenadas de textura para cada ponto (surfel) podemos mapear imagens de textura para as superfícies, através da determinação das cores correspondentes para os surfels.

A simplicidade de um ponto como primitivo para renderização torna-o muito útil em operações de modelagem, entretanto uma superfície representada por pontos precisa atender a certos critérios, principalmente com relação a densidade. A reconstrução correta de pontos adjacentes muito afastados pode ser impensável sem informações explícitas de conectividade. Além disso, por mais que pontos sejam mais simples para renderização, a maioria do *hardware* gráfico é otimizada para o uso de triângulos, assim a renderização baseada em triângulos ainda é melhor quando se compara a qualidade das imagens.

6.1 Trabalhos Futuros

A utilização da textura e a renderização em dois passos tornam o processo de renderização mais lento. No futuro queremos:

- pesquisar a combinação de estruturas de dados espacias que permitam acelerar o processo de visualização resolvendo o problema *blending*+oclusão;
- verificar a possibilidade de usar um operador de projeção para os vértices dos surfels, para uma melhor aproximação no processo de reconstrução;
- guardar informações de curvatura para cada ponto a partir de sua vizinhança local, para adaptar a curvatura do c-surfel.

Muitas questões relativas ao uso de pontos ainda podem ser trabalhadas. Esperamos no futuro estender as idéias aqui apresentadas, e apresentar novas técnicas para a renderização de objetos com representação por pontos.

Referências Bibliográficas

[1] Marc Alexa, Johannes Behr, Daniel Cohen-or, Shachar Fleisman, David Levin, and Claudio Silva. Point set surfaces. *IEEE Visualization 2001*, pages 21–28, 2001.

[2] J. F. Blinn and M. E. Newell. Texture and reflection for computer generated images. *Communications of the ACM*, 19:542–547, 1976.

[3] James F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics (SIGGRAPH 78 Proceedings)*, 12, no 3:286–292, 1978.

[4] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. *Proceedings of the 13th Eurographics workshop on Rendering*, pages 53 – 64, 2002.

[5] Loren Carpenter. The a-buffer, an antialised hidden surface method. *Computer Graphics (SIGGRAPH 84 Proceedings)*, 18(3):103–108, 1984.

[6] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Department of Computer Science, University of Utah, 1974.

[7] R. L. Cook. Shade trees. in h. christiansen, ed. *Computer Graphics (SIGGRAPH 84 Proceedings)*, 18:223–231, 1984.

[8] David S. Ebert and et al. *Texturing and Modeling: A Procedural Approach*. Academic Press, 1998.

[9] J. D. Foley, A. Vam Dam, S. K. Feiner, and J. H. Hughes. *Computer Graphics - Principles and Practice*. Addison-Wesley, Massachusetts, 1990.

[10] G. Gardner. Simulation of natural scenes using textured quadric surfaces. *Computer Graphics (SIGGRAPH 84 Proceedings)*, 18:11–20, 1984.

[11] G. Gardner. Visual simulation of clouds. *Computer Graphics (SIGGRAPH 85 Proceedings)*, 19(3):297–304, 1985.

[12] Jonas Gomes and Luiz Velho. *Sistemas Gráficos 3D*. IMPA, Rio de Janeiro, 2001.

[13] Markus Gross, Mark Pauly, and Leif P. Kobbelt. Efficient simplification of point-sampled surface. *In Proceedings IEEE Visualization 2002*, pages 163–170, 2002.

[14] Markus Gross, Hanspeter Pfister, Marc Alexa, Mark Pauly, Marc Staminger, and Matthias Zwicker. *Point-based computer graphics*. Eurographics '02 Tutorial, 2002.

[15] J. P. Grossman and William J. Dally. Point sample rendering. *Eurographics Rendering Workshop 1998*, pages 181–192, 1998.

[16] Eric Haines and Tomas Möller. *Real-Time Rendering*. A K Peters, Ltd., Massachusetts, 1999.

[17] Paul Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6:56–67, 1986.

[18] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California, 1989.

[19] Henrik Wann Jensen and Gernot Schauffer. Ray tracing point sampled geometry. In Springer-Verlag, editor, *Rendering Techniques 2000*, pages 319–328. Eds. Peroche and Rushmeier, 2000.

[20] Jaroslav Kriványek. Representing and rendering surfaces with points. Postgraduate Study Report DC-PSR-2003-3, Departament of Computer Science and Engineering - Czech Technical University, February 2003.

[21] Marc Levoy and Whitted Turner. The use of points as a display primitive. Technical Report 85-022, University of North Carolina at Chapel Hill, 1985.

[22] M. Pauly. *Point Primitives for Interactive Modeling and Processing of 3D Geometry*. PhD thesis, Federal Institute of Technology (ETH) of Zurich, 2003.

[23] D. R. Peachey. Solid texturing of complex surfaces. *Computer Graphics (SIGGRAPH 85 Proceedings)*, 19:279–286, 1985.

[24] K. Perlin. An image synthesizer. *Computer Graphics (SIGGRAPH 85 Proceedings)*, 19(3):287–296, 1985.

[25] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Barr, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH 2000*, pages 335–342. ACM Press/ ACM SIGGRAPH/ Addison Wesley Longman, 2000.

[26] Jussi Räsänen. Surface splatting: Theory, extensions and implementation. Master's thesis, Dept. of Computer Science, Helsinki University of Technology, 2002.

[27] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering shadows with depth maps. *Computer Graphics (SIGGRAPH 87 Proceedings)*, 21:283–291, 1987.

[28] Szymon Rusinkiewicz and Mark Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343 – 352. ACM Press/Addison-Wesley Publishing Co, 2000.

[29] Richard Szeliski and Christophe Schlick. Surface modeling with oriented particle system. *Computer Graphics (SIGGRAPH 92 Proceedings)*, pages 185–194, 1992.

[30] L. Velho and J. Gomes. *Fundamentos de Computação Gráfica*. Impa, Rio de janeiro, 2003.

[31] M. Frederick Weinhaus and Venkat Devarajan. Texture mapping 3d models of real-world scenes. *ACM Computing Surveys*, 29(4):325–365, 1997.

[32] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics (SIGGRAPH 90 Proceedings)*, 24(4):367–376, 1990.

[33] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3d: An interative system for point-based surface editing. *Computer Graphics (SIGGRAPH 2002 Proceedings)*, pages 322–329, 2002.

[34] Matthias Zwicker, Hanspeter Pfister, Jeroen van Barr, and Markus Gross. Surface splatting. *Computer Graphics (SIGGRAPH Proceedings 2001)*, pages 371–378, 2001.