COPPE/UFRJ

SIMPLIFICAÇÃO E MULTIRESOLUÇÃO DE MODELOS DE PONTOS

Felipe Carvalho

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Antonio Oliveira
Ricardo Marroquim

Rio de Janeiro
Dezembro de 2009

SIMPLIFICAÇÃO E MULTIRESOLUÇÃO DE MODELOS DE PONTOS

Felipe Carvalho

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Antonio Oliveira, D.Sc.

Prof. Ricardo Marroquim, D.Sc.

Prof. ..., D.Sc.

Prof. ..., Ph.D.

Prof. ..., D.Sc.

RIO DE JANEIRO, RJ – BRASIL

DEZEMBRO DE 2009

Carvalho, Felipe

Simplificação e Multiresolução de Modelos de Pontos/Felipe Carvalho. – Rio de Janeiro: UFRJ/COPPE, 2009.

XIII, 20 p.: il.; 29, 7cm.

Orientadores: Antonio Oliveira

Ricardo Marroquim

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2009.

Referências Bibliográficas: p. 18 – 19.

1. Nível de Detalhes. 2. Simplificação. 3. Multiresolução. I. Oliveira, Antonio *et al.*. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

(. . .).

Agradecimentos

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

SIMPLIFICAÇÃO E MULTIRESOLUÇÃO DE MODELOS DE PONTOS

Felipe Carvalho

Dezembro/2009

Orientadores: Antonio Oliveira
Ricardo Marroquim

Programa: Engenharia de Sistemas e Computação

(. . .)

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

SIMPLIFICATION AND MULTRESOLUTION OF POINT-SAMPLED SURFACE

Felipe Carvalho

December/2009

Advisors: Antonio Oliveira
Ricardo Marroquim

Department: Systems Engineering and Computer Science

(. . .).

Sumário

Lista de Algoríthmos

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

GPU *Graphic Processor Unit*, p. 3

SPT *Sequential Point Trees*, p. 3

Capítulo 1

Introdução

1.1 Motivação

Renderização Baseada em pontos vem se tornando popular no últimos anos devido a evolução do *hardware* gráfico 3D, e.g (*Graphic Processor Unit*) e devido ao fato, que modelos extremamente grandes produzidos pelos *Scanners* $3D$ estarem disponíveis [3, 4]. Ao contrário do método tradicional de renderização em que modelos $3D$ são aproximados por um conjunto de polígonos (malhas poligonais) e iluminados usando *Gouraund* ou *Phong* durante a projeção, técnicas de renderização baseada em pontos usa nuvem de pontos sem conectividade explicita.

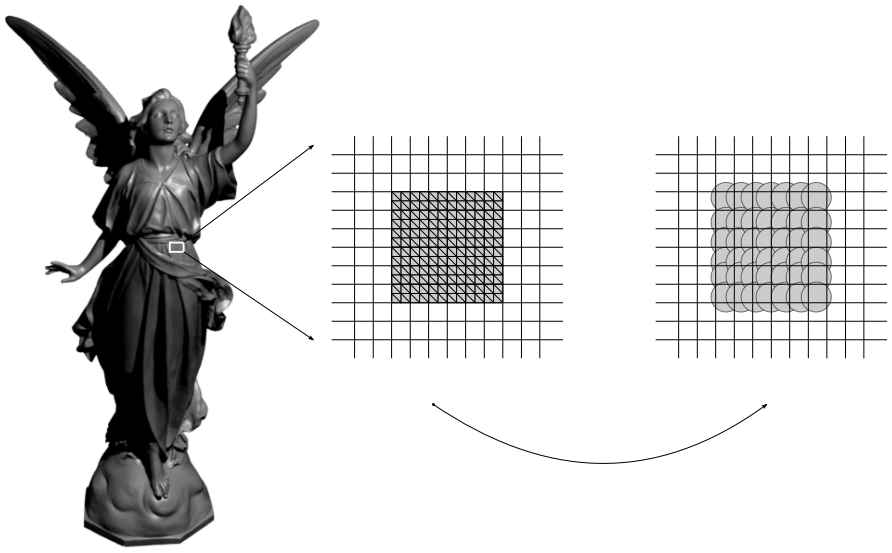


Figura 1.1: Lucy figure

Há várias razões para o sucesso da renderização baseada em pontos. Primeiro,

com o aumento na quantidade de memória das GPUs modernas tornou-se possível tratar cenas com quantidade enormes de polígonos e como consequência a quantidade de polígonos que são mapeados em menos de um *pixel* é grande, perdendo-se muito tempo no processo de rasterização(explicação) como ilustrado na Figura 1.1. Como as GPUs são altamente otimizadas para renderizar polígonos, muitos modelos eram convertidos em um conjunto de polígonos usando métodos como o algoritmo *marching cubes* [5]. No entanto as GPUs modernas agora possuem partes de seu *pipeline* gráfico programáveis, permitindo a criação de algoritmos de renderização alternativos utilizando-se todo o poder de processamento das GPUs [6, 7, 8]. Segundo, representar superfícies curvas usando polígonos planares requer informação de conectividade e ainda permanecem como uma aproximação da superfície curva. Pontos no entanto não mantém esse tipo de informação e operações como nível de detalhes, deformação geométrica e modificações na topologia são fáceis de implementar.

1.2 Objetivo

1.3 Organização da Dissertação

Capítulo 2

Superfícies Baseada em Pontos

Superfícies baseada em pontos recentetimente tem ganhado atenção da comunidade de computação gráfica [sites]. Comumente são obtidos de $3D$ *scanner* [sites], o resultado é uma nuvem de pontos P, que recobre a superfície do objeto. Operações de processamento e filtragem[Ver Tese tamy Capitulo 2] são definidos para reduzir o ruído e preencher regiões com buracos [cites]. Nessa dissertação iremos assumir que o conjunto de pontos P é livre de ruiídos e pois amostras adquadamente distribuídas na superfície. Neste capítulo, iremos mostrar uma definição de superficie de pontos, *splats*. *Splats* são uma extensão das superfícies puramente baseadas em pontos assossiando normal e um extensão aos pontos, formando-se assim, um disco orientado.

2.1 Malhas Poligonais vs Superfícies Baseadas em Pontos

2.2 *Splat*

Nessa sessão discutiremos *Splats*, que são extensão da representação pura de pontos assossiandos-se uma extensão (circular ou elíptica) e uma normal aos pontos. Na sessão 2.2.1 discutiremos como computar a normal para superfície de baseada em pontos, na sessão 2.2.2 difiniremos *splats* elípticos. Concluimos a capítulo com discurssão sobre algorítmos de renderização de superfícies baseadas em *splats*.

2.2.1 Normal

Assume-se o conjuntos de pontos como uma coleção de posições $P = x_i$ no R^3, $i \in \{1, \ldots, N\}$. Usaremos os k vizinhos mais próximos para definir a normal na superfície. O k vizinhos mais próximos de um ponto é definido como um subconjunto de k pontos com menor distância Euclidiana para o ponto dado. No capítulo 3 será apresentada uma estrutura eficiente para obter os k vizinhos de uma nuvem de pontos. Dado os k vizinhos $N_p^k = \{x_1, \ldots, x_k\}$ de um ponto x_i, a normal na superficie pode ser obtida usando a análise da matriz de covariância. A matriz de covariância é definida como:

$$C = \sum_{j}^{k} (x_j - x)(x_j - x)^T \tag{2.1}$$

com $x = \frac{1}{k} \sum_i^k x_j$ a média de todos os vizinhos. A matriz $3x3$ é semi-definida positiva e simétrica, possuindo todos os seus auto-valores reais. O auto-vetor correspondente ao menor auto-valor é uma estimativa da direção da normal na superfície no ponto x_i.

2.2.2 *Splat* Ellíptico

2.3 *Surface Splatting*

Capítulo 3

Estruturas de Dados para Pontos

3.1 Estrutura de Partição do Espaço

3.1.1 Estruturas não Hierarquicas

3.1.2 Estruturas Hieraquicas

Esquemas de partição do espaço são comuns(?) em computação gráfica, em particular quando se deseja processar geometria adquirida é essêncial: simplificação, reconstrução, compressão, visibilidade, e muita outras operações são beseadas em nesse tipo de estrutura. Sua simplicidade a tornou muito popular: o espaço inicial, frequentimente uma caixa envolvente do modelo, é recursivamente subdividida até que uma celula satisfaça um dado critério. As estutras de partição de espaço mais populares são, *octree* e *kD-Trees* (um caso especial de *BSP-Tree*). Primeiramente, *octree* será discutida na sessão 3.1.2, que é obtida particionando recursivamente a caixa envolvente do modelo em oito octantes. Na proxima sessão 3.1.2 apresentaremos *K-d Tree* que também particona o espaço, mas dividindo o modelo em cada nó de acordo com uma dimensão. *K-d Tree* será usada como estrutura eficente de busca dos k vizinhos de um dado ponto no modelo. Finalmente, hieraruia de esferas envolventes será discutida (3.1.2).

Octrees

Octree é uma das estruturas de partição espacial mais usadas para tratar grandes modelos de pontos em especial, quando se quer renderizá-los de forma interativa

[cites]. A estrutura é usada para particionar a caixa envolvente $3D$ que engloba todas as amostras no espaço. Cada célula que contenha amostras é recursivamente subdivida em oito octantes.A recursão termina quando em uma célula há um valor mínimo de amostras que passa a ser um contêiner de amostras desta célula (seriam as folhas de uma estrutura em árvore).

Octree sobre um conjuntos P de n pontos $p_{1...n}$ é construída de forma eficiente com complexidade de $O(n \log n)$

kd-Tree

Uma *kd-tree* é em geral , um árvore multidimensional de busca em k dimensões. Para dados em $3D$, a árvore correspondente é geralmente chamada de *kd-tree* tridimensional ao invés de $3d$*-tree*. Elas são um caso especial de arvores partição binária espacial . *Kd-tree* usa planos de cortes que são perpendicular a um dos eixos coordenados (também chamado hiperplanos), que é uma especialização de uma PBE, em que planos de cortes arbitrários podem ser usados. Um conjunto de pontos em uma *kd-tree* é subdividido em caixas alinhas aos eixos e que não se interceptam. O algoritmo de construção é descrito como o exemplificado:

Na raiz, o conjunto de pontos é dividido em dos subconjuntos com o mesmo tamanho por um hiperplano perpendicular ao eixo dos x. Os filhos da raiz (profundidade 1), a partição é baseada na coordenada y e o nó de profundidade 2, no próximo level, na coordenada z. Então o algoritmo começa ordenando o conjunto de pontos na coordenada x. A recursão para quando um determinada quantidade de pontos em um nó é alcançada (no caso um ponto, veja 3.1), o qual é armazenados nas folhas.

Hierarquia de Volumes Envolventes

Hierarquia de Volumes Envolventes (BVH) tem sido usado para renderização desde Clark and Rubin and Whitted [cite], uso-as como suporte para consultas como *visibilty culling* e intersecção de raio-objeto. Enquanto método de partição de dados para indexação no espaço, como *Octrees* e *Kd-Trees* descritas anteriormente, BVH não precisam ser uma partição do espaço. Sendo assim, BVH remove qualquer restrição em relação a partição do espaço e permite a construção de uma hierar-

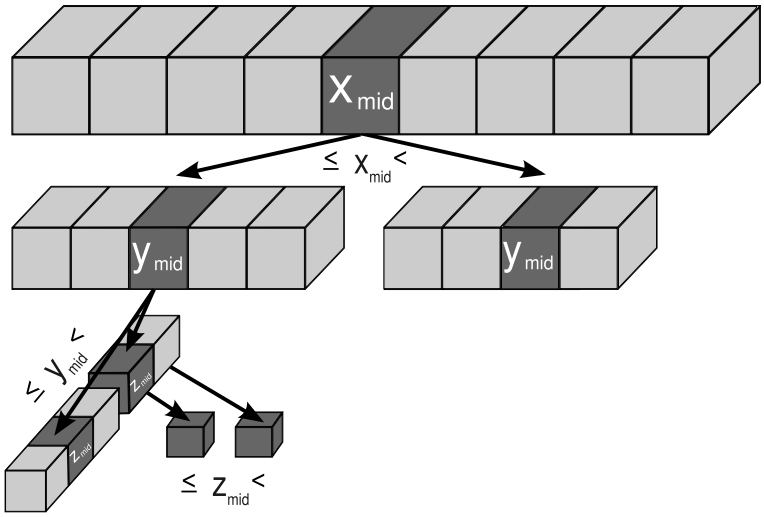


Figura 3.1: Kd-Tree

quia espacial mais genérica. De fato, ela permite qualquer hierarquia que agrupe os elementos, sem entretanto provê uma seleção espacial que é de fundamental importância para qualquer esquema de indexação espacial. O único requisito em uma BVH é que cada nó o volume envolvente (i.e., uma caixa ou esfera) engloba todos os elementos da sua subárvore. Obviamente que uma estrutura de partição espacial pode ser estendida para uma BVH, gerando o volume envolvente atribuído a cada nó como mostrado anteriormente.

3.2 Multiresolução e Nível de Detalhe

Neste sessão iremos apresentar alguns trabalhos que inspiraram esta dissertação, com ênfase em dois deles em particular. O primeiro é *QSplat* [9], um sistema para renderização de pontos baseado em uma hierarquia de esferas envolventes. O Segundo é o *Sequential Point Tress* (SPT) [2], uma estrutura de dados que permite-nos renderizar em Nível de Detalhe e diretamente na GPU modelos de pontos. E por último serão apresentados outro trabalhos que seguem esta mesma linha mas de forma resumida e com suas principais contribuições.

3.3 *QSplat*

Nesta seção iremos descrever o *QSplat*, um sistema para representação e renderização progressiva, de modelos grandes com amostras de 100 milhões há 1 bilhões de pontos como os produzidos no Projeto Michelangelo Digital [3]. *QSplat* combina uma hierarquia de esfera envolventes com renderização baseado em pontos. Os nós internos da hierarquia armazenam atributos (posição, normal, cor) que são estimados pelos seus nós filhos. O algoritmo de renderização percorre a hierarquia até que o tamanho da projeção da esfera envolvente seja menor que um valor pré-determinado (geralmente um *pixel*). Então o nó é renderizado e seus filhos podem ser descartados. O sistema será descrito com mais detalhes nas seções seguintes.

3.3.1 *QSplat* Estrutura de Dados

QSplat usa uma hierarquia de esfera envolventes que também é usada para controle de nível de detalhe, *view frustum cullling* e *back facing culling* [9]. Cada nó da hierarquia contem o centro e o raio da esfera envolvente, uma normal, o ângulo do cone de normais e uma cor (opcional). A hierarquia é criada em um pré-processamento e guardada em disco. Na Figura 3.2 temos uma esquema de como seria a hierarquia.

A estrutura de cada nó na hierarquia de esferas é mostrada na Figura 3.3. Um nó contem a posição e o tamanho da esfera relativa a seus parentes, normal, cone de normais e uma cor (opcional) e poucos *bits* que representam a estrutura da árvore.

Posição e Raio: A posição e o raio de cada esfera é codificada relativamente aos seus parentes na hierarquia de esferas envolventes. A fim de economizar memória, seus valores são quantizados em 13 valores. Então o raio de um esfera varia de 113 a 1313 do raio de seus parentes e seu centro relativo ao centro de seus parentes (em cada um dos suas coordenadas X, Y e Z) é algum múltiplo de 113. Para garantir que a processo de quantização não introduza nenhum buraco durante a renderização, todos ou valores são arredondados para o maior valor que englobe seus parentes

Normal A normal é codificada em 14 *bits*. Sua quantização usa um grade de $52x52$ em cada uma das 6 faces do cubo. Um tabela é usada para decodificar a normal

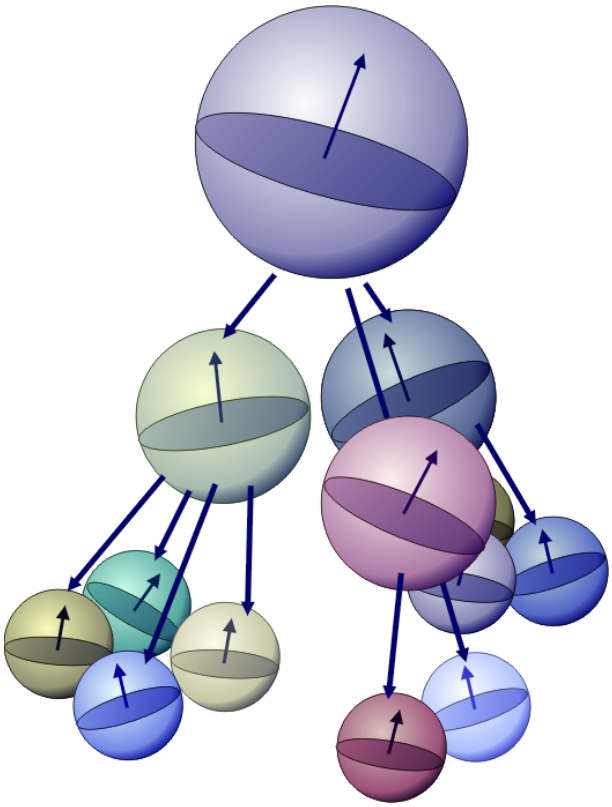


Figura 3.2: Figura esquemática da hierarquia de esferas de *QSplat* (Retirada de [1])

representada. Na prática são usados $52 \cdot 52 \cdot 6 = 16224$ diferentes valores de normal

Color O anglo do cone de normais é codificado em apenas 2 *bits*. Os quatros valores possíveis que representam o metade do anglo de abertura do cone são 116, 416, 916 e 1616

Cor Dúvida ...

3.3.2 Renderização

O processo de renderização é simples, como mostrado na Figura 3.4. Os estágios do algoritmo serão mostrados a seguir.

Visible Culling Como é usado um hierarquia de esferas envolventes, nós que não são visíveis são eliminados durante o percurso. *Frustum Culling* é feito, testando cada esfera contra os planos do tronco de pirâmide que representa o campo de visão. Se a esfera está fora, ela e sua sub-árvore são eliminados. Se

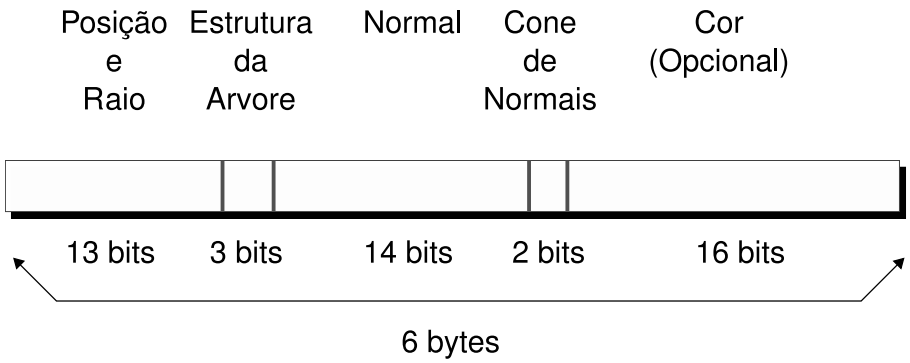


Figura 3.3: Estrutura de um Nó

ela está dentro do campo de visão, ela e seus filhos estão visíveis e não precisam mais passar pelo teste. *Backface Culling* também é realizados durante o processo de renderização, usando o ângulo do cone de normais.

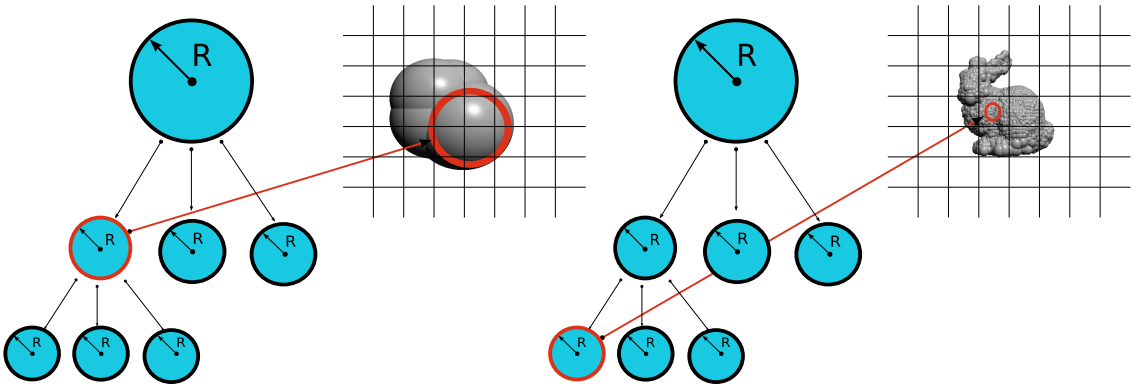
Heurística de Renderização A heurística usando é o tamanho da imagem projetada na tela, ou seja, área da esfera projetada na tela exceder um determinado valor (geralmente um *pixel*).

Renderizando um *Splat* Quando se atinge um nó desejado, de acordo com os critérios mencionados anteriormente, o *splat* é renderizado representando a esfera corrente. O tamanho do *splat* é baseado no diâmetro da projeção da esfera corrente, e sua cor é obtida usando cálculo de iluminação baseada na normal e cor da mesma.

3.3.3 Discussão

QSplat possui um processo bem simples, mas infelizmente ele não usa todo o potencial gráfico que as GPUs oferecem. Dada granularidade na sua determinação de nível de detalhes, um modelo chega e ser renderizado ponto por ponto. Como consequência, esse "modo imediato" de renderizar torna a GPU pouco utilizada, pois está sempre esperando por novos dados para renderizar. Levando em conta que não é apenas a coordenada de um ponto que está sendo utilizada, mas todos os seus outros atributos, como cor e normal.

No entanto, está simplicidade torna o *Qsplat* um algoritmo que pode ser usado em outras aplicações. Sua hierarquia de esfera é uma boa estrutura para *Ray Tracing*.



(a) (b)

Figura 3.4: Esquema do algoritmo de renderização: *(a)* Tamanho da imagem projetada do nó é maior que um *pixel*. Continua o percurso nas sub-árvores; *(b)* Tamanho da imagem projetada do nó é menor que um *pixel*. Renderiza o *splat*.

Outra são aplicações em rede como a do *Stream QSplat*[10] que permite visualizar modelos $3D$ de forma progressiva e remotamente.

3.4 *Sequential Point Trees*

QSplat possui um processamento de dados muito simples e de fácil implementação, mas infelizmente sua estrutura hierárquica recursiva é difícil de ser implementada em GPU. Os pontos renderizados não são armazenados de forma contínua, portanto não são processados sequencialmente.A *CPU* (*Central Processor Unit*) percorre a árvore e faz chamadas independentes para renderizar cada nó. Isso causa um "gargalo" entra a CPU e a GPU, sendo que esta última fica muito tempo ociosa esperando por dados da CPU. *Sequential Point Trees* propõe o uso da estrutura de *QSplat*, só que de uma forma sequencial que é facilmente tratada em GPU. Sendo assim, transferindo mais trabalho para GPU e diminuindo este "gargalo". Nas seções que seguem , SPT será apresentado com mais detalhes.

3.4.1 Hierarquia de Pontos

Inicialmente SPT possui um hierarquia de pontos representada por um *Octree* [**?**]. Cada nó da hierarquia representa parte do objeto. Ela armazena um centro **c** e

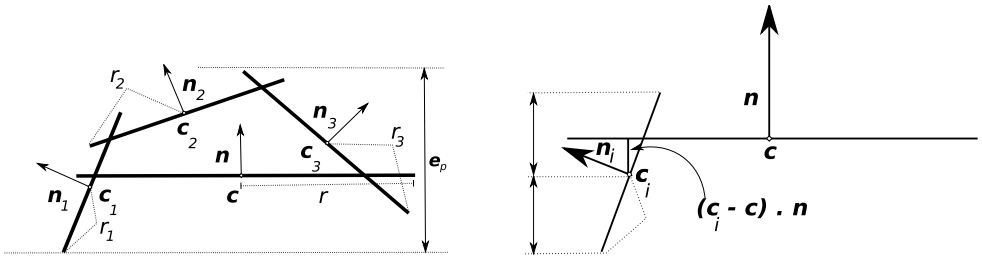


Figura 3.5: Como erro perpendicular, usa-se a distância entre dois plano paralelo ao disco que engloba todos os filhos (Retirada de [2]).

uma estimativa da normal **n**. Um nó armazena também um diâmetro **d** da esfera envolvente centrada em **c**. O nó interior representa a união de seus nós filhos, então o diâmetro cresce a medida que sobe na hierarquia. Os nós folhas possuem pontos que são distribuídos uniformemente no objeto, então possuem diâmetro aproximadamente iguais.

3.4.2 Métricas de Erro

Cada nó na hierarquia pode ser aproximada por um disco com o mesmo centro, normal e diâmetro do nó. O erro dessa aproximação é descrito por dois valores : o erro perpendicular e_p e o erro tangencial e_t.

Erro Perpendicular: O erro perpendicular e_p é a distância mínima entre dois planos paralelo ao disco que engloba todos os filhos. Este erro mede variância e pode ser calculado como:

$$e_p = max\{((c_i - c) \cdot n) + d_i\} - min\{((c_i - c) \cdot n) - d_i\} \quad (3.1)$$

$$\text{with} \quad d_i = r_i\sqrt{1 - (n_i \cdot n)^2} \quad (3.2)$$

Durante a renderização, o erro perpendicular é projetado na imagem, resultando em um erro $\tilde{e}_p$. $\tilde{e}_p$ é proporcional ao seno do ângulo entre o vetor de visão v e a normal do disco n e diminui com $\frac{1}{r}$ e $r = |v| \cdot \tilde{e}_p$ captura erros ao longo das silhuetas:

$$\tilde{e}_p = e_p \frac{\sin(\alpha)}{r} \quad \text{sendo} \quad \alpha = \angle(n, v) \quad (3.3)$$

Erro Tangencial: O erro tangencial e_t, analisa a projeção dos discos dos filhos no disco do pai como mostrado na Figura 3.6. e_t mede se o disco pai cobre um

grande área desnecessária. O erro é medido usando várias retas ao redor dos filhos projetados. e_t é portanto o menor diâmetro do disco pai menos o tamanho do menor intervalo entre retas. (dúvida em relação a escrita). e_t negativos são setados em zero. e_t é projetado no espaço de imagem como:

$$\tilde{e}_t = e_t \frac{\cos(\alpha)}{r} \tag{3.4}$$

Erro Geométrico: O erro perpendicular e tangencial podem ser combinados em um único erro geométrico: Agora o erro no espaço de imagem $\tilde{e}_g$ depende apenas de r, e não mais do ângulo do visão: $\tilde{e}_g = \dfrac{e_g}{r}$

$$\tilde{e}_g = \max\{e_p \sin\alpha + e_t \cos\alpha\} = \sqrt{e_p^2 + e_t^2} \tag{3.5}$$

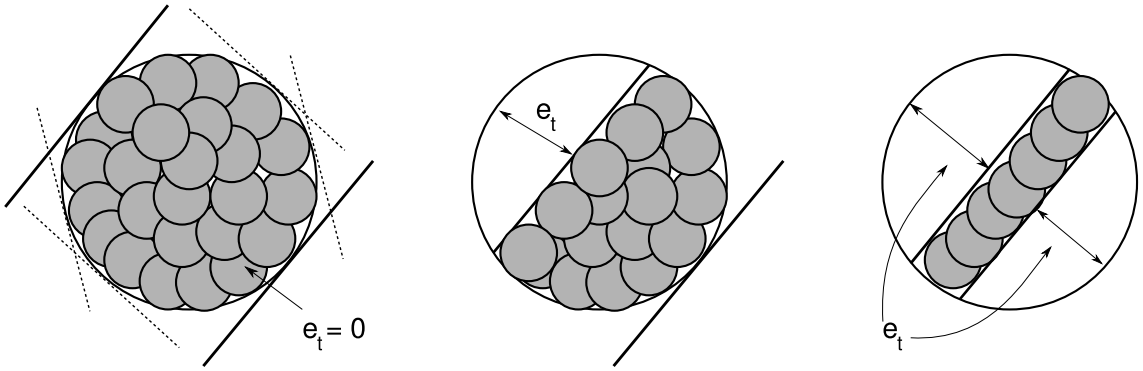
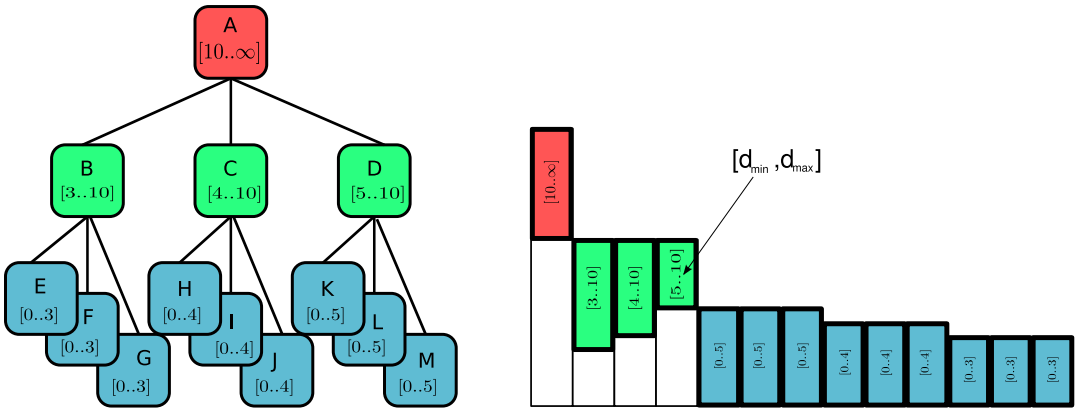


Figura 3.6: Erro tangencial, mede o quão aproximado é o disco pai em relação ao filho no plano tangente (Retirada de [2]) .

3.4.3 Renderização Recursiva

Um objeto é renderizado na hierarquia de pontos usando um percurso em profundidade. Para cada nó um erro de imagem $\tilde{e}_g$ é calculado. Se $\tilde{e}_g$ está abaixo de um limite de erro estabelecido ϵ e o nó não é uma nó folha , seus filhos são percorridos recursivamente. Por outro lado, um *splat* de tamanho $\tilde{d} = d/r$ é renderizado. Note que esta hierarquia de pontos não se adapta apenas a distância do observador r, mas também para propriedades da superfície. Grandes áreas planas são detecta-das com pequenos erro geométrico $\tilde{e}_g$ e podem ser renderizados com splats grandes.



(a) (b)

Figura 3.7: Como erro perpendicular, usa-se a distância entre dois planos paralelos ao disco que engloba todos os filhos (adaptada de [2]).

3.4.4 Arranjamento

O procedimento de renderização da seção anterior é recursivo e não se adapta ao processamento rápido e sequêncial da GPU. Assim, há um arranjamento da estrutura em árvore para um estrutura em lista e o teste recursivo é substituído por um percurso sequêncial sobre a lista de pontos.

Para isso, o erro simplificado $\tilde{e}_g$ é substituído por um que sejam mais intuitivo. Assume-se que ϵ é constante. O teste recursivo é $\tilde{e}_g = e_g/r < \epsilon$ e ao invés de e_g, é armazenado um distância mínima $r_{\min} = e_g/\epsilon$ que simplifica o teste recursivo para $r > r_{\min}$. Entretanto, quando os nós da árvore são processados seqüencialmente sem informação hierárquica, há necessidade de um teste não recursivo. Para esse fim, é adicionado um parâmetro $r_{\max}$, que é simplesmente um $r_{\min}$ do seu nó pai, em cada nó e usa-se $r \in [r_{\min}, r_{\max}]$ como um teste não recursivo. Desta maneira pode-se guiar o algoritmo de renderização com esse teste *intercalar* para cada nó.

Depois de substituir o teste recursivo por um simples teste intervalar, a hierarquia de pontos e transformada em um lista, que é processada sequencialmente. Neste estágio, $[r_{\min}, r_{\max}]$ é usada para ordenar a lista de forma decrescente a partir de $r_{\max}]$ como ilustrado na Figura 3.7.

Um exemplo de como o algoritmo de renderização funciona é mostrado na Figura [**?**]. Onde para diferentes valores de r um porção da lista é selecionada enquanto outras são descartadas.

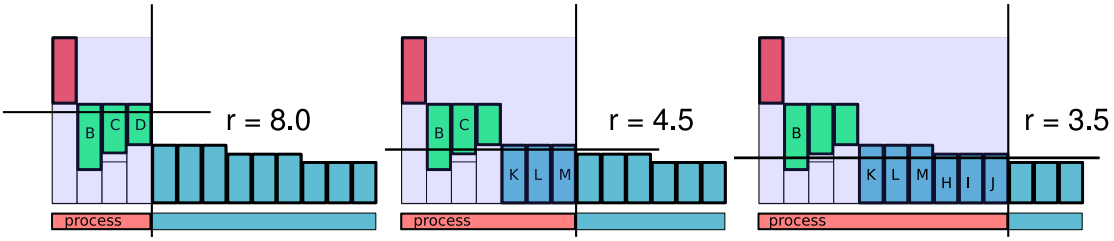


Figura 3.8: (Retirada de [2]) .

3.4.5 Discussão

SPT é simples, de fácil implementação e provê um renderização continua usando nível de detalhes.

O autor da ênfase no fato de que grande parte do trabalho é movido para GPU, deixando a CPU livre para outras tarefas. No entanto, SPT só é eficiente se o modelo estiver na memória de vídeo, o que nem sempre é possível.

SPT renderiza pontos em baixa qualidade, já que a GPU não suporta renderizar *splat* com qualidade em tempo satisfatório. Outra desvantagem é que SPT não realiza *frustum culling*, perdendo o pouco em eficiência.

Capítulo 4

Simplificação de Superfícies de Pontos

4.1 Trabalho Relacionados

4.2 Método Proposto

Capítulo 5

Conclusão e Trabalhos Futuros

Referências Bibliográficas

[1] WAND, M., *Point-Based Multi-Resolution Rendering*, Ph.D. Thesis, Department of computer science and cognitive science, University of Tübingen, 2004.

[2] DACHSBACHER, C., VOGELGSANG, C., STAMMINGER, M., "Sequential point trees", *ACM Trans. Graph.*, v. 22, n. 3, pp. 657–662, 2003.

[3] LEVOY, M., PULLI, K., CURLESS, S., etal., "The digital Michelangelo project: 3D scanning of large statues". In: *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 131–144, New York, NY, USA, ACM Press/Addison-Wesley Publishing Co., 2000.

[4] CORRÊA, W. T., FLEISHMAN, S., SILVA, C. T., "Towards Point-Based Acquisition and Rendering of Large Real-World Environments". In: *In Proceedings of the 15th Brazilian Symposium on Computer Graphics and Image Processing*, 2002.

[5] LORENSEN, W. E., CLINE, H. E., "Marching cubes: A high resolution 3D surface construction algorithm", *SIGGRAPH Comput. Graph.*, v. 21, n. 4, pp. 163–169, 1987.

[6] REN, L., PFISTER, H., ZWICKER, M., "Object Space EWA Surface Splatting : A Hardware Accelerated Approach to High Quality Point Rendering", *Computer Graphics Forum*, v. 21, n. 3, pp. 461–470, 2002.

[7] BOTSCH, M., KOBBELT, L., "High-Quality Point-Based Rendering on Modern GPUs". In: *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, p. 335, Washington, DC, USA, IEEE Computer Society, 2003.

[8] MARROQUIM, R., KRAUS, M., CAVALCANTI, P. R., "Efficient Point-Based Rendering Using Image Reconstruction". In: *Symposium on Point-Based Graphics 2007, Prague-Czech Republic*, September 2007.

[9] RUSINKIEWICZ, S., LEVOY, M., "QSplat: a multiresolution point rendering system for large meshes". In: *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 343–352, New York, NY, USA, ACM Press/Addison-Wesley Publishing Co., 2000.

[10] RUSINKIEWICZ, S., LEVOY, M., "Streaming QSplat: a viewer for networked visualization of large, dense models". In: *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 63–68, New York, NY, USA, ACM, 2001.

[11] SAMET, H., *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2005.

[12] ROSSIGNAC, J., BORREL, "Multi-Resolution 3D Approximations for Rendering Complex Scenes". In: *Modeling in Computer Graphics: Methods and Applications*, pp. 455–465, New York, NY, USA, Springer-Verlag, 1993.

Apêndice A

Análise das Componentes Principais

Apêndice B

Visibilty Culling