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

DENISON LINUS DA MOTTA TAVARES

Aproximação E ciente de Visibilidade para Nuvem de Pontos utilizando a GPU

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. João D. Comba
Orientador

Porto Alegre, April de 2009

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Tavares, Denison Linus da Motta

Aproximação Eficiente de Visibilidade
para Nuvem de Pontos utilizando a GPU / Denison Linus da Motta Tavares. – Porto Alegre: PPGC da UFRGS, 2009.

73 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2009. Orientador: João D. Comba.

1. Computação gráfica baseada em pontos. 2. Visibilidade. 3. Visualização de nuvem de pontos. I. Comba, João D.. II. Title.

“And remember my friends, future events such as these will affect you in the future.”

— CRISWELL, PLAN 9 FROM OUTER SPACE

AGRADECIMENTOS

Primeiramente agradeço ao Programa de Pós-Graduação em Computação (PPGC), do Instituto de Informática da UFRGS por proporcionar um excelente ambiente de pesquisa devido aos qualificados professores e a sua ótima infra-estrutura. Agradeço especialmente ao Grupo de Computação Gráfica, Processamento de Imagem e Interação ao qual faço parte, pois durante esses últimos dois anos ele proporcionou um ótimo lugar para o meu desenvolvimento pessoal e profissional.

Agradeço, sobretudo, ao Prof. João D. Comba, pela oportunidade de ser seu orientando, pela liberdade para desenvolver minhas ideias, por suas contribuições e sugestões nos trabalhos desenvolvidos durante o decorrer do mestrado.

De maneira especial sou grato a minha família pelo apoio e incentivo durante os períodos críticos da vida de um mestrando. Por isso este trabalho é dedicado ao meu pai Marcondes, minha mãe Eva e minha irmã Aline. Agradeço a todos os amigos e pessoas interessantes que tive o privilégio de conhecer, conviver e compartilhar um pouco da minha vida durante esses dois últimos anos.

Obrigado pela cortesia do repositório de *Scanning 3D* de *Stanford* (STANFORD, 2008) dos modelos utilizados neste trabalho *Bunny*, *Armadillo* e *Dragon*. Também os modelos *David* são cortesia do Projeto Michelangelo Digital (LEVOY et al., 2000). Agradeço pela disponibilidade dos modelos *Dinosaur*, *Santa*, *Igea*, *Sphere*, *Male*, *Face* e *Ball-joint* no *website* do *software Pointshop3D* (ETH ZURICH, 2007). Agradeço a *Bart Adams* por disponibilizar os modelos *Manhead* e *Checker* na sua página pessoal (ADAMS, 2007). Também sou grato ao repositório de *Laser Splatting* da *Universität Stuttgart* (PHOTOGRAMMETRIE, 2007) pela a disponibilização dos modelos *Farm House* e *Byzantine Basilica on the island of Crete*.

E por fim, agraço ao CNPq pelo suporte financeiro que possibilitou a realização deste trabalho durante os últimos dois anos.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

Nos últimos anos a utilização de pontos como primitiva gráfica básica vem mostrando-se uma poderosa e versátil ferramenta para a computação gráfica. Considerável esforço de pesquisa vem sendo dedicado para encontrar formas eficientes de aquisição, representação, processamento, renderização e animação para conjuntos de pontos. As representações baseadas em pontos têm-se destacado como uma estratégia eficiente em computação desde que se tornou comum extrair modelos geométricos a partir de *Scanners 3D*, os quais geram grandes quantidades de pontos que aproximam a geometria do objeto. Este trabalho apresenta um conjunto de métodos para tratar a visibilidade aproximada para nuvens de pontos sem informação de conectividade e topologia.

Primeiramente é proposto uma abordagem baseada em *clusters* para acelerar o operador de remoção de pontos proposto por Katz et al. A principal motivação para esta otimização é a possibilidade de conseguir um equilíbrio entre a velocidade e a qualidade do resultado.

Também é apresentado uma técnica de renderização baseada em pontos acelerada por *hardware* chamada *Surface Splatting*. Esta abordagem utiliza mapeamento de textura com *alpha blending* para aproximar um filtro de reamostragem *Elliptical Weighted Average* no espaço de objeto. Juntamente com o *Geometry Shader* das modernas placas gráficas, produz de forma eficiente imagens de alta qualidade de superfícies amostradas por *surfels*.

Por último é proposto um novo operador de remoção de pontos ocultos acelerado por *hardware* baseados na técnica de *splatting* juntamente com um operador morfológico de erosão modificado para reduzir o efeito de silhuetas no resultado final do operador. A motivação para a criação deste novo operador é a baixa eficiência demonstrada pelos métodos existentes para a utilização em aplicações em tempo real onde as nuvens de pontos são muito densas.

Todas as técnicas apresentadas neste trabalho podem ser utilizadas em visualização científica com taxas interativas, em particular na visualização direta de geometria baseada em pontos.

Palavras-chave: Computação gráfica baseada em pontos, visibilidade, visualização de nuvem de pontos.

E cient Approximate Visibility of Point Sets On The GPU

ABSTRACT

In recent years the use of points as a fundamental graphics primitive has proved to be a powerful and versatile tool for computer graphics. Considerable research has been devoted to the efficient representation, modeling, processing, rendering and animation of point-sampled geometry. The point-based representation has gained increasing attention in computer graphics because 3D scanning systems easily extract large information from real-world objects. On the other hand, point sets are more flexible when compared to triangle meshes, because they are not required to maintain consistent topological information. This work presents a set of tools to determine the visibility and also to render a point-based geometry efficiently.

Firstly, a cluster-based approach is proposed to speed up the hidden point removal operator proposed by Katz et al. The main idea of this study is to trade-off speed and quality in dynamic scenes of moving or deforming point clouds.

After that, a hardware based point rendering technique called Surface Splatting is introduced. This approach uses the texture mapping with alpha blending and the Geometry Shader to approximate the Elliptical Weighted Average filter in object space. This efficient technique produces high quality images as surfel-based geometry.

Finally, a new hidden point removal operator is presented. This operator, based on the splatting technique and also hardware accelerated, applies a morphological erosion operation in the depth buffer to reduce the silhouette effect in the final image. The motivation to develop a new operator is the low efficiency demonstrated by existing hidden point removal methods in real time applications, where the point cloud is very dense.

All the techniques introduced in this work can be used in scientific visualization with interactive frame rates, particularly when visualizing point-based geometry sets.

Keywords: Point-based computer graphics, visibility, visualizing point clouds.

1 INTRODUÇÃO

Modelar geometria e aparência de objetos complexos do mundo real é um problema central em geometria computacional (ZWICKER et al., 2001). Enquanto muitas soluções nesta área são bem conhecidas e descritas, existe muita pesquisa sendo feita para desenvolver formas mais eficientes de representação computacional destes objetos (SOLENTHALER; ZHANG; PAJAROLA, 2007; ADAMS, 2006; BARBARA SOLENTHALER; PAJAROLA., 2007; JACOBSSON, 2007).

Nos últimos anos a utilização de pontos como primitiva gráfica básica vem mostrando-se uma poderosa e versátil ferramenta para a computação gráfica. Considerável esforço de pesquisa vem sendo dedicado para encontrar formas eficientes de aquisição, representação, processamento, renderização e animação para conjunto de pontos (ALEXA et al., 2004).

Existem dois principais motivos para este crescente interesse por computação gráfica baseada em pontos. Primeiramente, com o aumento na complexidade de modelos poligonais em computação gráfica, o custo de processar e manipular informação de conectividade para malhas poligonais é muito grande, tornando questionável a utilidade de polígonos como primitiva gráfica fundamental (GROSS; PFISTER, 2007). Um segundo motivo é relacionado ao fato que modernos sistemas de escaneamento adquirem simultaneamente grandes volumes de amostras de geometria e aparência de objetos complexos do mundo real (nuvem de pontos) (ZWICKER et al., 2001). Nuvens de pontos são conjuntos de posições tridimensionais, que podem ser associadas a informações do modelo, como por exemplo, cor e normal da superfície. Provavelmente a representação baseada em pontos não irá substituir completamente as atuais primitivas de visualização, mas irá prover um complemento a elas.

A conversão destas nuvens de pontos geradas pelos *Scanners 3D* em modelos poligonais convencionais está se tornando demasiadamente cara ou desnecessária, portanto algumas alternativas vêm sendo discutidas na literatura para trabalhar diretamente com estas nuvens de pontos, e isto inclui métodos de representação, modelagem e visualização.

Este trabalho aborda o problema de extrair diretamente a visibilidade de uma nuvem de pontos, o qual é muito importante para o *rendering*. Determinação de visibilidade é um problema clássico em computação gráfica 3D, também conhecido como problema de remoção de superfícies ocultas (*HSR - Hidden Surface Removal*) ou ainda problema de determinação de superfícies visíveis (*VSD - Surface Determination*), e foi estudado desde o início desta área em computação nos anos 70. Entretanto, a maioria dos algoritmos visam determinar a visibilidade correta entre objetos poligonais (CATMULL, 1974; SUTHERLAND; SPROULL; SCHUMACKER, 1974; FUCHS; KEDEM; NAYLOR, 1980; WARNOCK, 1969). Extrair visibilidade de uma nuvem de pontos pode ser

feito através da reconstrução da superfície do objeto (HOPPE et al., 1992), e renderizar o objeto resultante utilizando os algoritmos já mencionados.

Alguns algoritmos baseados em espaço de imagem definem visibilidade em termos de *pixels*, tais como *Ray Tracing* e *Z-Buffer*. A ordem em que os raios disparados e as primitivas da cena se interceptam definem a ordem de visibilidade no *Ray Tracing* (WALD; SEIDEL, 2005). O algoritmo de *Z-Buffer*, o qual é implementado na maioria das placas gráficas, pode ser usado para determinar a visibilidade de uma nuvem de pontos, desde de que este contenha informação de posição e os vetores normais, utilizando técnicas de *Surface Splatting* (ZWICKER et al., 2001; SAINZ; PAJAROLA, 2004; CARSTEN DACHSBACHER; STAMMINGER, 2003).

Uma abordagem alternativa foi proposta por Katz et al. (KATZ; TAL; BASRI, 2007), o qual introduziu o primeiro algoritmo de visibilidade baseada em pontos que determina diretamente a visibilidade de um conjunto de pontos sem reconstruir a superfície, e sem utilizar informações extras da nuvem de pontos, tais como as normais dos pontos. Sabendo que pontos não sobrepõem um ao outro (exceto em casos especiais), e que os pontos representam uma superfície implícita, um procedimento para determinar a visibilidade destes pontos foi definido em seu método por um operador chamado *Hidden Point Removal Operator* (*HPR* Operator).

Uma visualização direta de duas nuvens de pontos mostrada Figura 1.1 (Modelo *David* e *Bunny*) não permite dizer se os objetos mostrados na cena estão de frente ou de costas para câmera. Mas como a aplicação do operador nas duas nuvens de pontos podemos ver claramente na Figura 1.2 que os objetos estão de costas para a câmera.

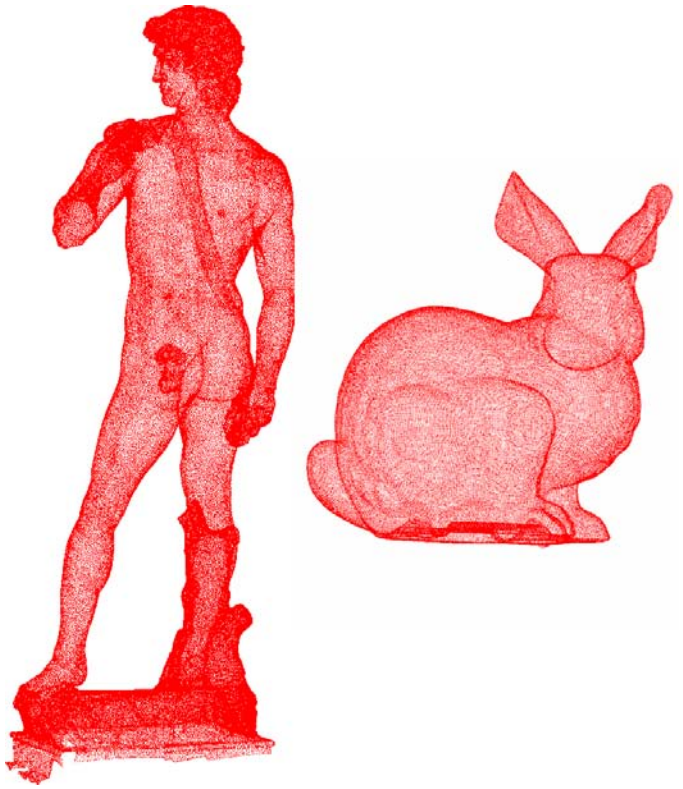


Figura 1.1: Exemplo de visualização direta de nuvem de pontos. Imagem extraída de (KATZ; TAL; BASRI, 2007).

Sua proposta é muito interessante, simples de implementar e produz resultados excelentes, mas com um custo computacional alto, pois ela possui uma etapa onde um cálculo

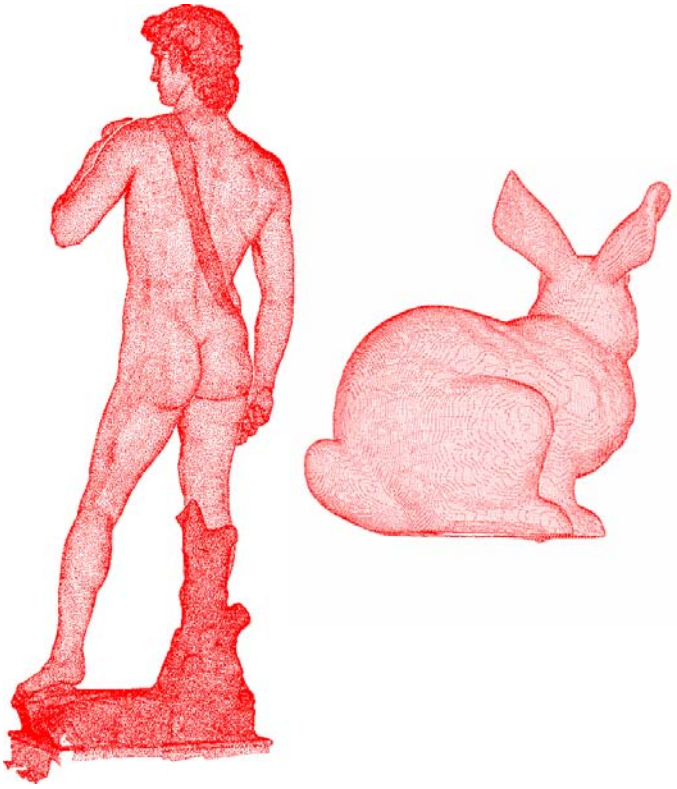


Figura 1.2: Exemplo de visualização de nuvem de pontos após a aplicação do operador *HPR*. Imagem extraída de (KATZ; TAL; BASRI, 2007).

de envoltória convexa é necessário, o qual frequentemente requer diversos segundos de processamento para renderizar nuvens de pontos de tamanhos moderados ou grandes.

Este trabalho revisita a proposta de Katz et al. propondo uma abordagem para acelerar o algoritmo, bem como um novo método para o cálculo de visibilidade de nuvem de pontos utilizando o *hardware* gráfico. Deste modo o método pode ser estendido para a utilização em cenas dinâmicas, onde as nuvens de pontos estão movendo-se ou sofrendo deformação da geometria.

A motivação por trás da abordagem para acelerar o *HPR Operator* original é a possibilidade de conseguir um equilíbrio entre a velocidade e a qualidade do resultado. Esta característica é frequentemente útil em aplicações de visualização interativa, como por exemplo, em algoritmos de *LOD* (*Level of Detail*), onde a qualidade dos objetos é reduzida para diminuir o tempo de rendering dos mesmos. A principal contribuição introduzida nesta proposta é uma extensão do *HPR Operator* baseado em *clusters* o qual proporciona um ganho significativo na performance e uma aproximação com baixa perda de qualidade na informação de visibilidade.

A motivação para a criação de um novo método, para o cálculo de visibilidade para uma nuvem de pontos, é a baixa eficiência demonstrada pelos métodos existentes para a utilização em aplicações em tempo real onde as nuvens de pontos são muito densas.

Este novo operador computa a informação de visibilidade para uma nuvem de pontos utilizando um algoritmo de quatro passos de renderização na *GPU* (*Graphics Processing Unit*), onde a superfície da nuvem de pontos é eficientemente aproximada renderizando dentro do *Z-Buffer impostores* orientados na direção da visão da câmera. Estes *impostores* têm como finalidade gerar um *depth buffer* completamente preenchido, sem buracos, para remover os pontos não visíveis dentro da superfície implícita no próximo passo do

algoritmo.

Este processo difere do passo de visibilidade de um algoritmo de *Surface Splatting* pela aplicação de um operador morfológico de erosão no *Z-Buffer* para suavizar as silhuetas geradas pelos *impostores* de tamanhos impróprios. Como as nuvens de pontos não possuem informação de conectividade, sem as normais dos pontos, não é possível obter noção de curvatura da superfície de forma eficiente. Logo, é difícil a implementação de algoritmos adaptativos que utilizem *impostores* de diferentes tamanhos, dependendo da curvatura da superfície em relação à câmera. A principal contribuição introduzida neste novo método é a aplicação de uma operação morfológica de erosão no *Z-Buffer*, simulando de forma eficiente o efeito de adaptatividade nos tamanhos *impostores*.

As contribuições desta dissertação são apresentadas a seguir:

- Aceleração do operador *HPR* original utilizando estruturas de subdivisão espacial baseadas em *clusters*.
- Aplicação do operador baseado em *clusters* para a visualização direta de nuvem de pontos dinâmicas.
- Desenvolvimento de um novo operador acelerado por *hardware* para remoção de pontos ocultos baseado na técnica de *Surface Splatting*.
- Aplicação do operador baseado em *splats* para visualização direta de nuvens de pontos.

1.1 Estrutura da Dissertação

O restante deste trabalho está organizando na seguinte estrutura. A Seção 2 sintetiza os trabalhos relacionados mais importantes na área de determinação de visibilidade, desde métodos de *Culling*, passando por métodos de remoção de superfícies ocultas, e também determinação de visibilidade para nuvem de pontos. A Seção 3 introduz uma nova abordagem baseada em clusterização espacial para acelerar o operador *HPR* original. A Seção 4 apresenta alguns fundamentos na renderização baseada em pontos, dando ênfase na técnica de *Surface Splatting*. A Seção 5 apresenta um novo método para o calculo de visibilidade de nuvens de pontos baseado na idéia do idéia do algoritmo de *Surface Splatting* na *GPU*. E por último, a Seção 6 sumariza este trabalho e lista algumas possibilidades para futuras investigações.

2 ALGORITMOS DE VISIBILIDADE

O conceito de visibilidade é uma abstração matemática da noção de visibilidade da vida real. Dado um conjunto de obstáculos no espaço Euclideano, dois pontos neste espaço são ditos visíveis um ao outro se o segmento de linha que liga os dois pontos não intercepta obstáculos. Um ser humano pode identificar sem esforço partes visíveis de muitos objetos em um ambiente. Entretanto, essa computação de visibilidade para uma cena com muitos objetos é uma tarefa difícil para computadores (GHOSH, 2007). Cálculo de visibilidade é um dos problemas clássicos em computação gráfica, e muitos algoritmos foram projetados para resolver de forma eficiente este problema.

Soluções para o problema visibilidade são importantes para diferentes áreas, incluindo computação gráfica, arquitetura, visão computacional, robótica e telecomunicações. Em computação gráfica *3D*, a determinação de superfícies ocultas é necessária para renderizar corretamente uma imagem, de modo que objetos ocultos por outros objetos não possam ser visualizados diretamente, por exemplo. Este processo é frequentemente ligado a um método de renderização baseado em rasterização, também conhecido como *pipeline* gráfico. Tipicamente o *pipeline* gráfico aceita representações geométricas vetoriais de cenas tridimensionais como entrada, e como saída imagens *2D* rasterizadas.

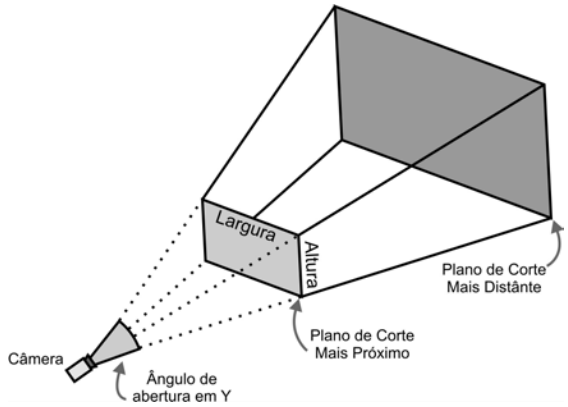
A geometria de uma cena virtual passa por etapas bem definidas dentro do *pipeline* gráfico antes de ser rasterizada e apresentada como imagem *2D*. As operações básicas presentes em um *pipeline* gráfico são as seguintes: transformação de modelagem, transformação de visualização, transformação de projeção, recorte e rasterização.

Na transformação de modelagem, a geometria *3D* dos objetos da cena virtual passa do sistema de coordenadas do objeto para o sistema de coordenadas do mundo. Isto pode incluir transformações locais, tais como translação e rotação, no espaço do objeto das primitivas geométricas dos objetos da cena virtual.

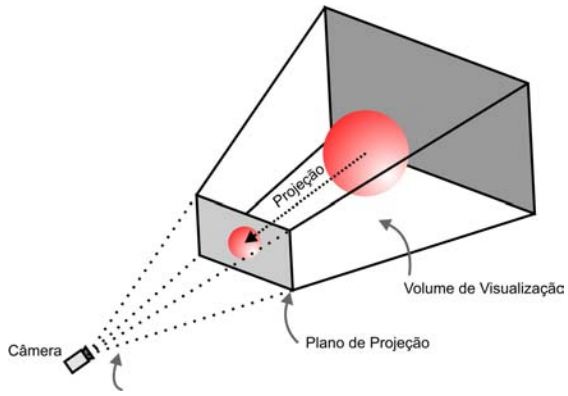
A transformação de visualização converte a geometria dos objetos da cena virtual do sistema de coordenadas do mundo para um sistema de coordenadas baseado na posição e orientação da câmera virtual. Este processo resulta em uma cena *3D* vista a partir do ponto de visão da câmera. O novo sistema de coordenadas produzido também é conhecido como espaço da câmera, ou espaço do olho.

Na transformação de projeção, a geometria dos objetos é transformada do espaço da câmera para o espaço de imagem, mapeando a cena *3D* em um plano como visto a partir da câmera virtual. Além da posição e orientação da câmera, este processo depende de parâmetros extras, tais como o ângulo do campo de visão da câmera na direção *Y* (*Field of View in Y)*, a relação entre a altura e a largura do plano de visualização (*Aspect Ratio*), a distância do plano de corte do volume de visualização (*Viewing Frustum*) mais próximo da câmera virtual (*Z Near*) e por último a distância do plano de corte do volume de visualização mais distante da câmera virtual (*Z Far*). Uma representação do volume

de visualização e o processo de projeção pode ser vista na figura 2.1.



(a) Parâmetros que definem o volume de visualização.



(b) Mapeamento da geometria da cena *3D* no plano de projeção *2D*.

Figura 2.1: (a) Representação do volume de visualização. (b) Transformação de projeção para um cena virtual.

Quando primitivas geométricas da cena virtual encontram-se parcialmente, ou totalmente fora do volume de visualização elas não são visíveis a partir da câmera virtual definida, logo devem ser descartadas ou ainda recortadas utilizando os planos do volume de visualização como polígonos de corte. A etapa de recorte não é necessária para a geração da imagem final, embora acelere o processo de rendering, eliminando primitivas desnecessárias que nunca serão visíveis a partir da câmera virtual definida.

Após a geometria da cena virtual ser transformada, projetada e recortada, ela deve ser rasterizada. Este processo converte a representação vetorial da cena virtual no espaço de imagem para *pixels* visíveis, resultando em uma imagem *2D*. Em um *pipeline* gráfico típico, é nessa etapa que é resolvido o problema de visibilidade, utilizando o algoritmo de *Z-Buffering*.

2.1 Métodos de *Culling*

Uma classe de métodos relacionados a testes de visibilidade são os algoritmos de *culling*. Estes algoritmos geralmente processam grupos de primitivas gráficas aceitando-as ou rejeitando-as antes do envio para o *pipeline* de visualização (BITTNER; WONKA, 2003). As vantagens desse pré-processamento é que grupos de primitivas que não são

visíveis, para o determinado ponto de visão, não serão enviadas para serem transformadas e rasterizadas no *pipeline* gráfico. Alguns tipos de algoritmos de *culling* são apresentados a seguir.

2.1.1 *Viewing Frustum Culling*

O *Viewing Frustum* (Ver Figura 2.1) é uma representação geométrica do volume visível da câmera virtual. O algoritmo de *Viewing Frustum Culling* remove objetos (grupos de primitivas gráficas) que localizam-se fora deste volume de visualização, pois não serão visíveis na imagem final. Frequentemente, estes objetos podem estar parcialmente dentro do volume de visualização, logo o método de *Viewing Frustum Culling* pode optar por não descartar o objeto, ou ainda, aplicar um algoritmo de recorte, passando para o *pipeline* gráfico apenas as primitivas do objeto que estão dentro do volume de visualização.

2.1.2 *Backface Culling*

Este tipo de *culling* é feito no nível de primitiva gráfica (polígono). O método de *Backface Culling* determina se um polígono de um objeto gráfico é visível, dependendo da posição da câmera virtual e o vetor normal do polígono (Figura 2.2). Ou seja, este método elimina polígonos que contenham faces *"de costas"* para o observador. Atualmente, o processo de *Backface Culling* é implementado em *hardware*, sendo necessário apenas habilitá-lo, garantindo que a enumeração dos vértices da face estejam no sentido correto (normalmente em sentido anti-horário).

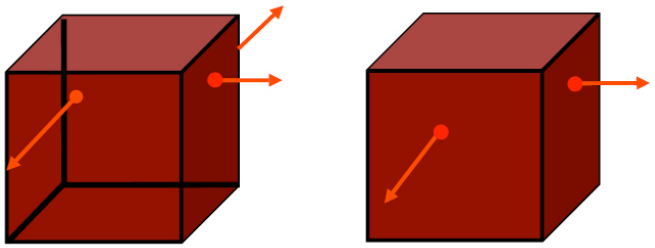


Figura 2.2: Remoção de faces traseiras ao observador (*Backface Culling*).

2.1.3 *Occlusion Culling*

O *Occlusion Culling* é uma variação mais sofisticada de método de *culling* onde objetos que estão inteiramente encobertos por outros objetos opacos são eliminados do *pipeline* gráfico (LEYVAND; SORKINE; COHEN-OR, 2003). Este é um mecanismo muito popular para otimizar a renderização de cenas 3D complexas. Por exemplo: seja uma cena com um observador em frente a um muro atrás do qual existem diversos objetos complexos. Uma grande quantidade de processamento computacional pode ser poupada descartando-se todos os objetos que se encontram atrás do muro, já que estes não podem ser visualizados pelo observador.

Existem algumas variações nos algoritmos de renderização baseada em *Occlusion Culling* (COHEN-OR; CHRYSANTHOU, 2003):

1. Renderização utilizando *Potentially Visible Set* (*PVS*): dividem a cena em regiões e fazem uma pré-processamento de visibilidade das mesmas. Estes conjuntos de visualização pré-computados são indexados de forma a acelerar a renderização;

2. Renderização utilizando Portais: dividem a cena em células (*rooms*) e portais (*doors*), computam quais células são visíveis a partir de cada um dos portais.

2.2 Métodos de Remoção de Superfície Oculta

Remoção de superfícies ocultas é o processo usado para remover as superfícies ou partes de superfícies de uma cena que não são visíveis a partir de um determinado ponto de visão. Existem muitos métodos para a determinação de superfícies ocultas. Quase todas as técnicas podem ser atribuídas a um processo de classificação, variando geralmente na ordem em que a classificação é feita e como o problema é subdividido.

No trabalho de *Ivan Sutherland et al.* (SUTHERLAND; SPROULL; SCHUMACKER, 1974) o problema de visibilidade é apresentado como um processo de ordenação, tanto utilizando métodos de ordenação baseados em *scattering* como *gathering*. Em computação gráfica, métodos de ordenação que utilizem a estratégia de ordenação baseada em *scattering* são utilizados em algoritmos de remoção de superfície oculta baseado em espaço de objeto. A classificação no espaço de objeto para grandes quantidades de primitivas gráficas é geralmente feita utilizando uma abordagem de divisão e conquista. Em contraste a esta estratégia, métodos de ordenação que utilizem *gathering* são utilizados em algoritmos de remoção de superfície oculta baseado em espaço de imagem. Estes algoritmos são projetados geralmente para produzir imagens com resolução fixa em tempo real. Atualmente esses algoritmos são processados eficientemente por *hardware* dedicado.

2.2.1 *Z-Buffering*

O Algoritmo de *Z-Buffering*, introduzido por Catmull (CATMULL, 1974), gerencia durante a rasterização as coordenadas de profundidade de cada fragmento. O valor de profundidade de cada fragmento rasterizado é verificado contra o valor de profundidade existente no *buffer* de profundidade (*Z-Buffer*, Figura 2.3) (GREENE; KASS; MILLER, 1993). Se o fragmento testado está atrás do fragmento já existente no *Z-Buffer*, então ele é rejeitado, caso contrário, o valor de profundidade do fragmento é escrito no *Z-Buffer*. Atualmente, o método de *Z-Buffering* é implementado em *hardware*, sendo necessário apenas habilitá-lo. O *Z-Buffer* pode gerar erros de precisão, também conhecido como *Z-Fighting*, devido a baixa quantidade de *bits* para representar o valor de profundidade do fragmento no *hardware*.

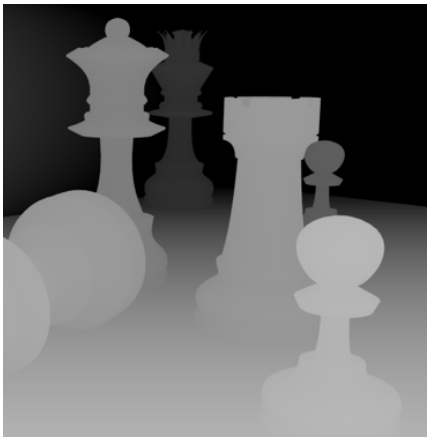


Figura 2.3: Representação do *buffer* de profundidade (*Z-Buffer*). Quanto mais escuro o *pixel*, maior a coordenada *Z*.

2.2.2 Algoritmo do Pintor

O Algoritmo do Pintor é uma simples solução para o problema de visibilidade em computação gráfica 3D. Quando a cena 3D é projetada no plano de visualização, em algum ponto do *pipeline* gráfico é necessário decidir quais polígonos são visíveis e quais são ocultos. O nome Algoritmo do Pintor refere-se ao método como um artista pinta uma obra (Figura 2.4), desenhando os cenários mais afastados do ponto de vista da cena primeiramente, cobrindo esse cenário de fundo com as partes mais próximas ao ponto de vista.

Figura 2.4: Representação do *Algoritmo do Pintor*. As montanhas distantes são pintadas por primeiro, seguido pelo terreno e finalmente pelos objetos mais próximos do observador, as árvores.

Este método ordena os polígonos da cena pelos seus centróides e então rasteriza-os nesta ordem. A complexidade assintótica deste algoritmo está no passo de ordenação. Ele pode falhar em certos casos, como por exemplo na Figura 2.5, onde os polígonos *A*, *B* e *C* sobrepõem-se um ao outro, não sendo possível decidir qual polígono está na frente dos demais. Desta forma, é criado um ciclo na ordem de visibilidade, tornado impossível determinar a profundidade de cada polígono corretamente.

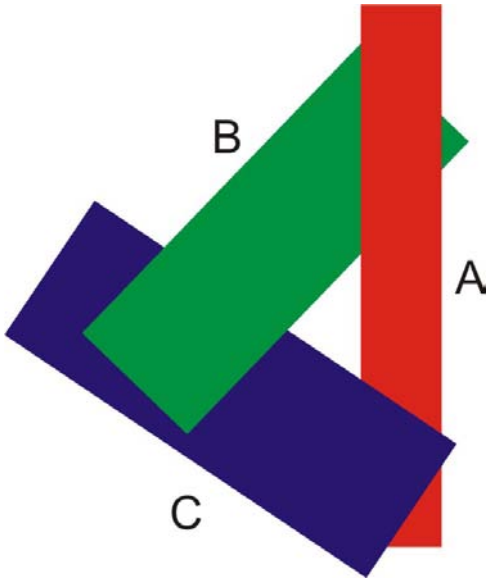


Figura 2.5: A sobreposição dos polígonos *A*, *B* e *C* pode causar erros no Algoritmo do Pintor.

2.2.3 *Binary Space Partitioning*

Binary Space Partitioning é um método de subdivisão recursiva, que divide o espaço em regiões convexas utilizando hiperplanos. Esta subdivisão espacial gera uma estrutura de dados para representação do espaço conhecida como árvore *BSP* (FUCHS; KEDEM; NAYLOR, 1980).

Esta subdivisão do espaço é construída de tal maneira que uma travessia na árvore *BSP* a partir de um ponto de visão provê uma forma de ordenação por profundidade dos polígonos. As desvantagens na utilização desta estrutura de dados para resolver o problema de visibilidade são o custo de pré-processamento para construir a árvore, e a limitação na utilização de cenas dinâmicas.

A Figura 2.6 ilustra o processo recursivo de particionamento arbitrário de uma cena virtual utilizando os polígonos *2D* da cena como linha de corte. A árvore *BSP* vai sendo construída a cada etapa de particionamento da cena virtual. Em cada etapa é adicionado um novo nodo na árvore, e cada nodo da árvore possui dois outros nodos filhos. O filho da esquerda armazena os polígonos da cena que estão na frente da linha de corte, e no filho da direita é armazenado os polígonos que estão atrás da linha de corte. Quando ocorre uma intersecção entre a linha e um polígono da cena, o polígono é recortado pela linha de corte em dois novos, que são adicionados na árvore nos seus respectivos nodos.

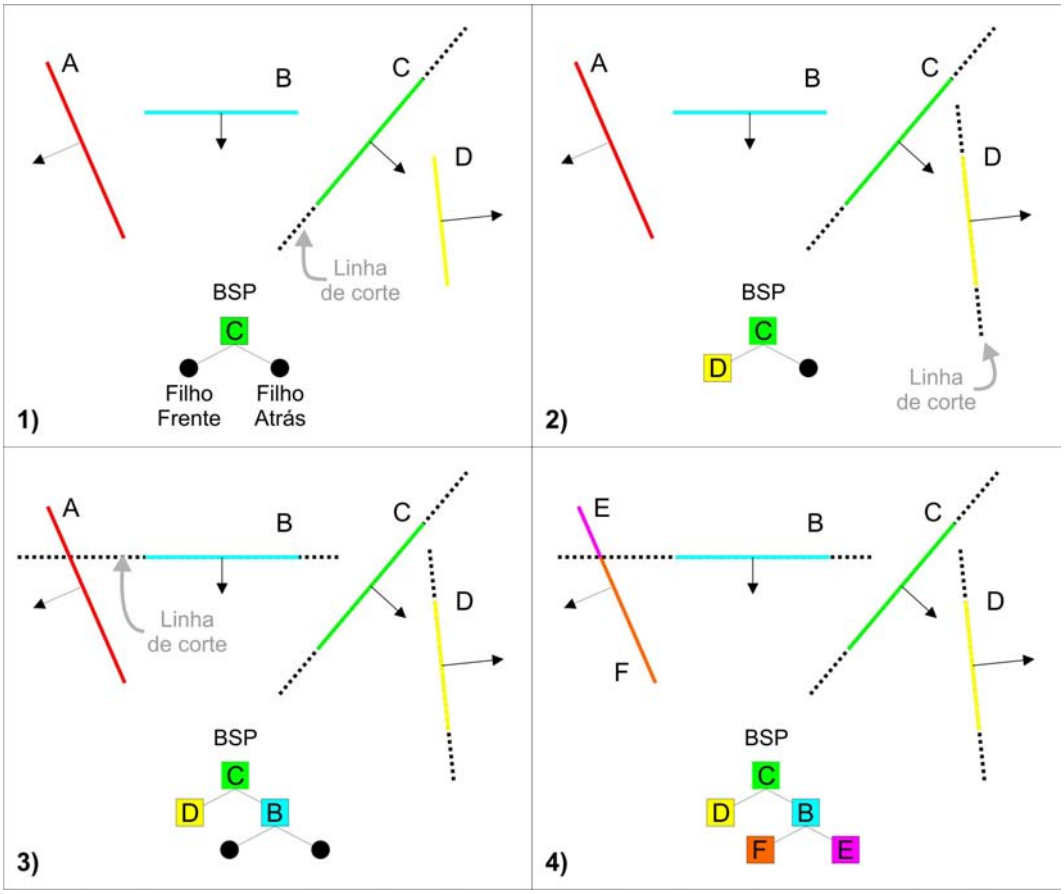


Figura 2.6: Construção de uma árvore *BSP* em 2D. 1) *C* é a raiz da árvore. 2) *D* é atribuído ao nodo esquerdo de *C* pois encontra-se na frente da linha de corte. 3) *B* é atribuído ao nodo direto de *C* pois encontra-se atrás da linha de corte. 4) A linha de corte em *B* recorta o polígono *A* em dois novos polígonos *E* e *F*.

2.2.4 *Ray Tracing*

O *Ray Tracing* é um método muito utilizado em computação gráfica para produzir imagens foto-realísticas. O primeiro algoritmo que utilizou a ideia de traçar raios a partir do plano de projeção foi o método de *Ray Casting* apresentado em 1968 por Arthur Appel (APPEL, 1968). Este algoritmo disparava um raio por *pixel* a partir do plano de projeção, a fim de encontrar o primeiro objeto que obstrua o raio lançado. Utilizando as propriedades do material do objeto interceptado e as propriedades das luzes contidas na cena, este algoritmo consegue determinar o *shading* do objeto.

O algoritmo de *Ray Tracing*, apresentado por *Turner Whitted* in 1980 (WHITTED, 2005) produz imagens mais realísticas que o método de *Ray Casting*, pois os raios traçados a partir de cada *pixel* do plano de projeção interceptam os objetos da cena, acumulando informação da contribuição de cada luz na cena, e recursivamente disparam outros raios de reflexão, refração e sombreamento a partir da última intersecção. Esta recursão continua até que um valor máximo de recursão é atingido, ou a contribuição torna-se insignificativa, gerando imagens como a ilustrada na Figura 2.7.

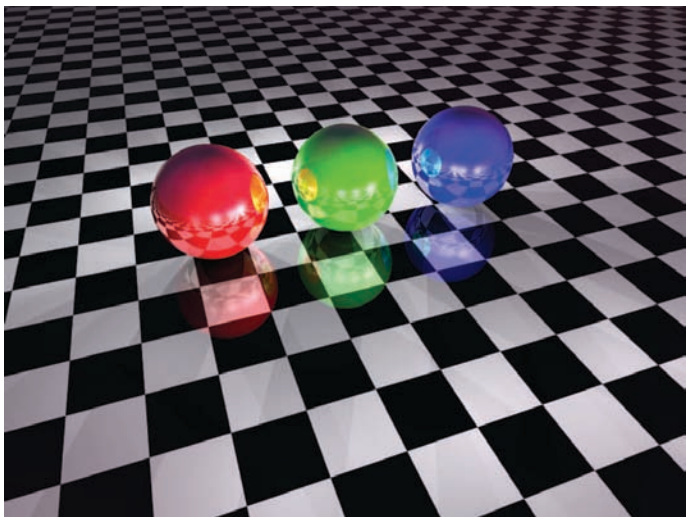


Figura 2.7: Imagem gerada com algoritmo de *Ray Tracing* utilizando o *software* POV-Ray.

Este processo de traçar raios a partir do ponto de visão faz com que o método de *Ray Tracing* resolva implicitamente o problema de remoção de superfície oculta.

2.2.5 *Algoritmo de Warnock*

O Algoritmo de *Warnock* é um método de remoção de superfície oculta (WARNOCK, 1969) que utiliza subdivisão de área recursivamente. O plano de projeção é divido em pequenas áreas as quais são feitas uma classificação dos polígonos contidos nelas. Se houver uma ambiguidade de profundidade entre os polígonos dentro de uma área, uma nova subdivisão é feita recursivamente. Esta subdivisão é feita até que a determinada área contenha um polígono simples sem intersecção com outros polígonos. A cada iteração a área de interesse é subdividida em quatro áreas iguais. Cada polígono é comparado para cada área e classificado em uma das quatro possibilidades:

- **Polígonos separados**: completamente fora da área de interesse;
- **Polígonos de intersecção**: cruzam a área de interesse;

- **Polígonos envolvidos**: contêm completamente a área de interesse;

- **Polígonos contidos**: completamente dentro da área de interesse.

Na figura 2.8 as quatro possibilidades de classificação dos polígonos são ilustradas.

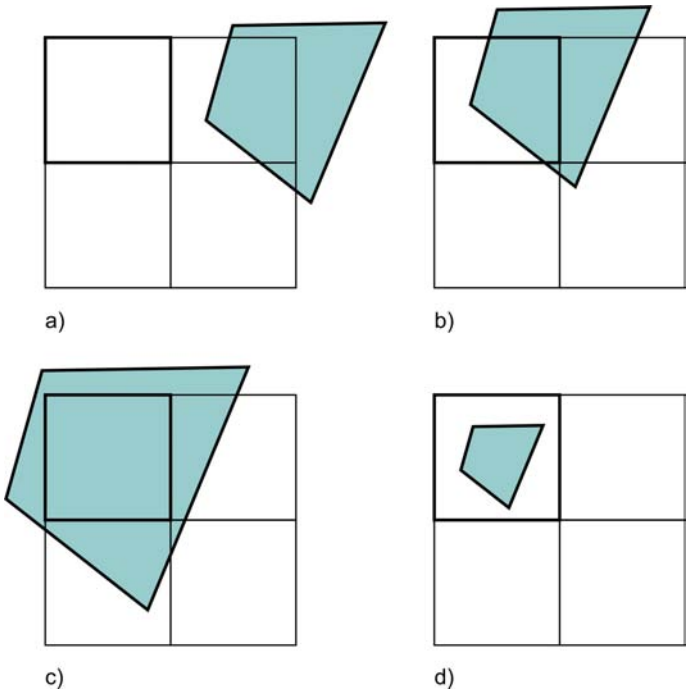


Figura 2.8: Classificação dos polígonos em relação a área de interesse. a) Polígonos separados. b) Polígonos de intersecção. c) Polígonos envolvidos. d) Polígonos contidos.

Para uma determinada área de interesse:

1. Se todos os polígonos são separados então a área é preenchida com a cor de fundo.

2. Se existe um único polígono contido ou um polígono de intersecção então a área é preenchida com a cor de fundo, após isto, a parte do polígono dentro da área é preenchido com a cor deste polígono.

3. Se existe um único polígono envolvido e não existem polígonos de intersecção ou polígonos contidos então a área é preenchida com a cor do polígono envolvido.

4. Se existe um polígono envolvido na frente de qualquer polígono envolvido, polígono de intersecção ou polígono contido então a área é preenchida com a cor do polígono envolvido que está na frente.

Caso contrário a área de interesse é dividida em quatro partes iguais e o procedimento apresentado é aplicado recursivamente a cada uma das novas áreas geradas.

A complexidade assintótica deste algoritmo é $O(np)$, onde n é o número de polígonos e p é o número de *pixels* do *viewport*.

2.3 Determinação de Visibilidade para Nuvem de Pontos

Normalmente a visibilidade de pontos é calculada durante o *ray tracing* (WALD; SEIDEL, 2005) ou pela reconstrução da superfície (HOPPE et al., 1992). Também pode ser feita com o uso de técnicas de *Voronoi/Delaunay*, representações por funções implícitas ou *Moving Least Squares* (ALEXA et al., 2003). Alguns métodos trabalham com o algoritmo de *Z-Buffer*, utilizando rasterização de pontos baseado em *Surface Splatting* (MIGUEL SAINZ; LARIO, 2004; CARSTEN DACHSBACHER; STAMMINGER, 2003). Alguns deles possuem uma fundamentação teórica forte, enquanto outros são otimizados para obter melhor desempenho. Particionamento espacial é comumente usado em computação gráfica para prover um eficiente método para detecção de colisões (GOTTSCHALK; LIN; MANOCHA, 1996; LUQUE; COMBA; FREITAS, 2005) e para indexar objetos em uma cena virtual (WANG; YANG; MUNTZ, 1997), mas também é utilizado para determinar a visibilidade entre polígonos (FUCHS; KEDEM; NAYLOR, 1980).

A maioria dos algoritmos para remoção de superfícies ocultas determina a visibilidade entre polígonos ou fragmentos, mas existem algumas abordagens baseadas em pontos, que dado um ponto de visão, processam a visibilidade correta de uma nuvem de pontos. Como pontos não podem ocultar-se diretamente (a não ser em casos degenerados), pode-se imaginar como se a superfície amostrada, representada pela nuvem de pontos, existisse continuamente, escondendo determinados pontos do ponto de visão.

Ainda que calcular a visibilidade correta seja útil em várias aplicações, em computação gráfica baseada em pontos é utilizada basicamente para a renderização. Muitos métodos de determinação de visibilidade para nuvens de pontos assumem que o conjunto de pontos satisfaz uma taxa de amostragem suficientemente boa, associação com as normais da superfície, ou ainda, com alguma informação extra que permita estimar as normais da superfície. A abordagem usada no *HPR - Hidden Point Removal Operator* em (KATZ; TAL; BASRI, 2007) resolve o problema de visibilidade sem o uso de técnicas de *rendering* e sem informações adicionais sobre a nuvem de pontos, e com garantias teóricas sobre este cálculo.

2.3.1 Operador de Remoção de Pontos *HPR*

Extrair informação de visibilidade diretamente das coordenadas espaciais de uma nuvem de pontos não deve gerar resultados conclusivos, pois pontos normalmente não ocultam um ao outro. A ideia principal por trás do operador proposto por Katz et al. (KATZ; TAL; BASRI, 2007) é considerar que os pontos estão sobre uma superfície contínua que não está explicitamente construída. Portanto, o operador *HPR* não requer o passo de reconstrução de superfície. Os detalhes sobre o operador são discutidos em (KATZ; TAL; BASRI, 2007) e são brevemente revistos aqui. Dado $P = \{p_i \mid 1 \leq i \leq n\} \subset \Re^D$, um conjunto de n pontos representando uma amostragem de uma superfície contínua S, o objetivo do operador *HPR* é determinar $\forall p_i \in P$ se p_i é ou não visível a partir de uma dada posição de câmera C. Considere uma região de interesse definida por uma esfera D-dimensional de raio R centrada na origem C. O operador *HPR* primeiramente aplica uma transformação inversa chamada *Spherical Flipping* (KATZ; LEIFMAN; TAL, 2005), o qual inverte todos pontos dentro da esfera ao longo de um raio que parte de C e chega em p_i (Figura 2.9).

A operação de *Spherical Flipping* é definida como se segue:

$$f(p_i) = p_i + 2(R - \|p_i\|)\frac{p_i}{\|p_i\|} \tag{2.1}$$

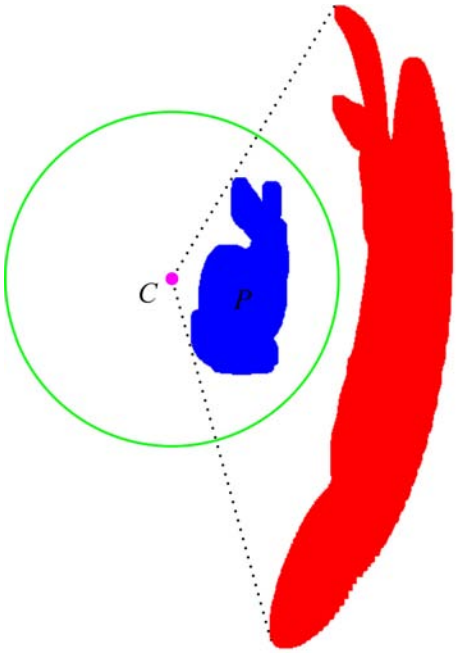


Figura 2.9: Intuição da Operação de *Spherical Flipping*. Pontos dentro de uma dada esfera são invertidos ao longo de um raio que inicia em C e passa através de p_i.

Pode-se mostrar que os pontos contidos na envoltória convexa dos pontos transformados pela operação de *Spherical Flipping* definem uma ordem de visibilidade válida para um determinado ponto de visão de uma nuvem de pontos. Este método é possível implementar em poucas linhas de código *Matlab*, e sua complexidade assintótica é dada pelo cálculo da envoltória convexa ($O(n \log n)$). O único parâmetro que o operador recebe de entrada é usado para computar o raio R do *Spherical Flipping*. O operador pode ser usado em várias aplicações, tais como visualização direta de conjunto de pontos *3D*, reconstrução de superfície dependente do ponto de vista e *Shadow Casting* (KATZ; TAL; BASRI, 2007). A Figura 2.10 apresenta um exemplo de visualização direta de um conjunto de dados baseado em pontos, mostrando o resultado obtido quando renderizado diretamente (esquerda) e com o uso operador (direita). A Figura 2.10 possui uma escala de cor, onde os pontos em vermelho estão mais próximos da câmera e os pontos em azul estão mais distantes.

2.3.2 Implementação do *HPR*

A implementação do operador de remoção de pontos *HPR* consiste em seis passos distintos, sendo que o último passo é que define a complexidade assimtótica do algoritmo $O(n \log n)$ (*Convex Hull*). O algoritmo *Hidden Point Removal Operator* apresentado no Algoritmo 1 tem como parâmetros de entrada a nuvem de pontos *Points*, o ponto de visão *Camera*, e *Param*, que serve como ajuste para o cálculo manual do raio do *Spherical Flipping*. Ele tem como saída um *array* contendo os pontos que são visíveis a partir do ponto de visão *Camera*.

Os seis passos do algoritmo são os seguintes:

1. **Passo 1**: Transforma os pontos da nuvem de pontos para que o ponto *Camera* torne-se a origem do sistema de referência;

2. **Passo 2**: Calcula a distância de cada ponto da nuvem em relação a origem, atri-

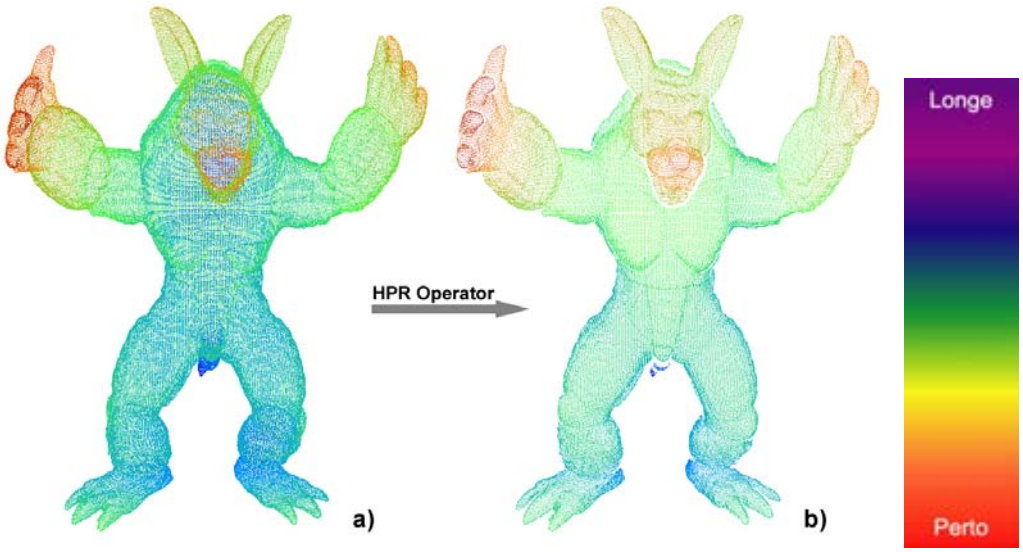


Figura 2.10: *Dataset Armadillo* renderizado com: (a) a nuvem de pontos original e (b) o operador *HPR* com R $(log(R) = 3.1)$. Os pontos estão em uma escala de cor (à direita) baseada na profundidade da cena.

buindo esses valores ao vetor *normp*;

3. **Passo 3**: Calcula o raio da esfera que será utilizada no *Spherical Flipping*;

4. **Passo 4**: Transforma os pontos da nuvem de pontos utilizando um *Spherical Flipping*, atribuindo esses novos pontos transformados no vetor *tmp*;

5. **Passo 5**: Processa a envoltória convexa da nuvem de pontos transformada pelo processo de *Spherical Flipping*, o resultado é atribuído ao vetor de números inteiros *index*, que é o índice dos pontos em *Points* que fazem parte do *Convex Hull*;

6. **Passo 6**: Atribui ao vetor de pontos *out* os pontos processados pela envoltória convexa e indexados em *index*. Os pontos contidos em *out* correspondem aos pontos da nuvem de pontos *Points* que são visíveis a partir de *Camera*.

2.4 Sumário

Este capítulo apresentou uma visão geral dos principais algoritmos para resolver o problema de visibilidade em computação gráfica. Primeiro são brevemente descritos os mais utilizados métodos baseados em *Culling*, os quais geralmente processam grupos de primitivas gráficas, aceitando-as ou rejeitando-as, previamente ao envio para o *pipeline* de visualização. Também são apresentados métodos para remoção de superfícies ocultas, os quais removem as primitivas gráficas, ou parte dessas, que não são visíveis a partir de um determinado ponto de visão. O processo utilizado nestes métodos pode ser atribuído a um processo de ordenação, que leva em conta a distância das primitivas gráficas em relação à posição da câmera. Por último, são brevemente apontados métodos que determinam a visibilidade baseado em pontos, não utilizando mais polígonos ou fragmentos como primitiva gráfica básica. Além disso, é apresentado uma breve descrição do operador *Hidden Point Removal* (KATZ; TAL; BASRI, 2007), o qual é considerado o primeiro

método de remoção de superfície oculta que computa diretamente a visibilidade para nuvem de pontos, sem o uso de reconstrução de superfície ou informações extras na nuvem de pontos.

Algorithm 1 *Hidden Point Removal Operator*

Require: $Array < Point > \ Points$ {Nuvem de Pontos}
Require: $Point\ Camera$ {Ponto de Visão}
Require: $Real\ Param$ {Parâmetro para o cálculo do raio do *Spherical Flipping*}
1: {**Passo 1**}
2: **for** $i \leftarrow 0;\ i < Points.Size();\ i++$ **do**
3: $\quad Points[i] \leftarrow Points[i] - Camera$
4: **end for**
5: {**Passo 2**}
6: $Array < Real > normp$
7: **for** $i \leftarrow 0;\ i < Points.Size();\ i++$ **do**
8: $\quad normp.Append(Points[i].Length())$
9: **end for**
10: {**Passo 3**}
11: $Real\ radius \leftarrow normp.GetMax() * pow(10 * Param)$
12: {**Passo 4**}
13: $Array < Point > tmp$
14: **for** $i \leftarrow 0;\ i < Points.Size();\ i++$ **do**
15: $\quad Point\ p \leftarrow Points[i] + 2 * (radius - normp[i]) * \frac{Points[i]}{normp[i]}$
16: $\quad tmp.Append(p)$
17: **end for**
18: {**Passo 5**}
19: $Array < Integer > \ index \leftarrow ConvexHull(tmp)$
20: {**Passo 6**}
21: $Array < Point > out$
22: **for** $i \leftarrow 0;\ i < index.Size();\ i++$ **do**
23: $\quad out.Append(Points[index[i]])$
24: **end for**
25: **return** out

3 *HPR* BASEADO EM *CLUSTER*

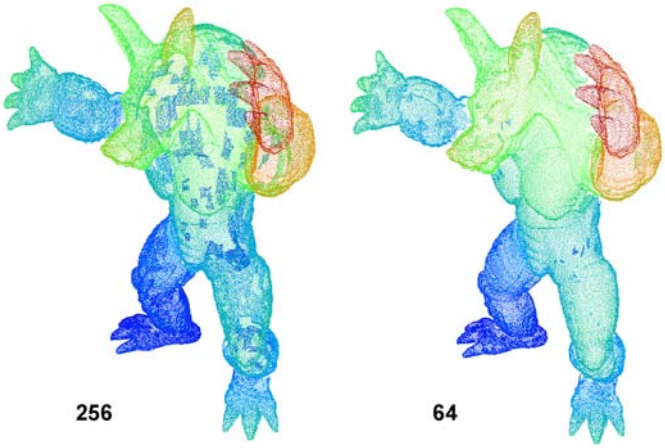
Neste capítulo o método apresentado por Katz et al. em (KATZ; TAL; BASRI, 2007) é reformulado em dois novos métodos, de forma a aumentar a performance do algoritmo original. Deste modo, o operador pode ser aplicado de maneira mais eficiente em cenas dinâmicas, onde os objetos estão se movendo ou deformando sua geometria. A motivação por trás da nova proposta é a possibilidade de obter uma troca entre desempenho e qualidade visual do resultado do operador. Este processo é semelhante aos algoritmos de níveis de detalhes (*LOD - Level of Detail*), os quais reduzem a qualidade dos modelos a fim de diminuir o tempo de *rendering*, ou aumentar a performance em sistemas de detecção de colisões. A contribuição introduzida aqui é uma extensão do operador *HPR* baseada em *clusters*, a qual, comparada com o operador original, provê melhor performance e gera uma informação de visibilidade aproximada com baixa perda de qualidade para a nuvem de pontos. Duas variações do operador original foram implementadas, a primeira usa uma *Grade Regular* e a segunda usa uma *Octree* para reduzir o gargalo do operador *HPR*. Também é demonstrado neste capítulo o uso destes novos operadores aplicados à visualização direta de nuvens de pontos dinâmicas.

Figura 3.1: Cinco quadros da animação ilustrando o resultado da aplicação do operador operador *HPR* baseado em *Octree* para uma nuvem de pontos dinâmica do modelo *Dragon*. Este método é mais eficiente que o operador original e gera imagens com baixo percentual de erro.

A Figura 3.1 ilustra o resultado do algoritmo baseado em *Octree* em uma cena dinâmica, onde o processo de remover os pontos ocultos pela superfície implícita do objeto *3D* é crucial para entender o movimento e as mudanças das formas do modelo. Neste exemplo, a nuvem de pontos do modelo *Dragon* está sofrendo uma deformação elástica, a qual pode ser vista em cada quadro da animação transformando sua geometria. Os resultados mostrados na Figura 3.1 produziram um ganho na velocidade de 4.16 vezes, e a perda da qualidade visual de 14.85%, comparado com o operador original.

O operador *HPR* foi projetado para nuvens de pontos estáticas. Estender seu uso para nuvens de pontos dinâmicas pode ser feito aplicando o operador separadamente para cada quadro da animação. Esta abordagem mostrou-se ineficiente, pois o cálculo da envoltória convexa em cada quadro da animação consome muito tempo de computação. Conseguir reduzir o tempo consumido por este estágio do operador é crucial para estendê-lo a cenas dinâmicas.

(a) operador *HPR* baseado em *Grade Regular* com diferentes resoluções do *grid*.



(b) operador *HPR* baseado em *Octree* com diferentes números máximos de pontos por *cluster*.

Figura 3.2: Resultados do operador *HPR* baseado em *Grade Regular* ou em *Octree* com diferentes estratégias de refinamento aplicado ao modelo *Armadillo* original. As versões com menor quantidade de *clusters* introduzem mais erro no operador *HPR*.

Uma técnica, que pode ser utilizada neste caso, é a redução do número de pontos usados na computação da envoltória convexa. Um possibilidade é escolher pontos representativos dentro do conjunto de dados do modelo *3D*, e esta é a essência da proposta apresentada aqui. Primeiramente é definido um método de subdivisão espacial (*Grade Regular* ou *Octree*) para armazenar as informações da nuvem de pontos em um dado quadro de animação. Em outras palavras, é definido um algoritmo de clusterização para um conjunto de pontos pela associação de cada ponto a uma determinada célula da divisão espacial. Cada célula (*cluster*) contém uma coleção de pontos e um ponto representativo para este conjunto. Para a eleição do ponto representativo, o método computa o centro de

massa do conjunto de pontos contidos na célula e guarda dentro da estrutura da mesma. Quando todos os pontos representativos estão computados, o algoritmo de envoltória convexa é aplicado somente sobre os pontos representativos. Em outras palavras, é assumido que o centro de massa do conjunto de pontos de cada célula dá uma boa estimativa da visibilidade dos pontos associados à célula.

A construção da subdivisão espacial é crucial para a eficácia desta abordagem, e reflete diretamente como uma escolha entre a velocidade e a precisão do resultado do método. Fixar um valor baixo para o número máximo de pontos por célula acarretará em um aumento do número de células na subdivisão. Para aumentar a precisão do método, é necessário reduzir o desempenho do mesmo. No caso onde existe um ponto por *cluster*, a performance é inferior ao operador original, dado que existe uma sobrecarga na construção da subdivisão espacial. As diferentes versões do operador, baseado em *Grade Regular* e *Octree* são detalhadas nos próximos tópicos e uma ilustração do impacto da escolha de diferentes estratégias no controle da subdivisão espacial pode ser vista na Figura 3.2.

3.1 *HPR* Baseado em Grade Regular

O operador *HPR* baseado em *Grade Regular* define um recorte no espaço em forma de caixas retangulares sobre a nuvem de pontos. O espaço é subdividido em uma série de *clusters* contínuos associados a identificadores únicos (i, j, k). Para cada quadro da animação da nuvem de pontos dinâmica, este operador cria um conjunto de *clusters* alinhados nos eixos com tamanho (d_x, d_y, d_z). Cada *cluster* contém os pontos que estão dentro desta região, e também o centro de massa destes pontos, o qual é usado como ponto representativo dos mesmos. Note que a resolução do *Grade Regular* e o tamanho do conjunto dos pontos representativos são diretamente proporcionais. Em outras palavras, quanto maior a resolução do *grid*, maior será o número de pontos representativos, portanto mais pontos devem ser processados na etapa do cálculo da envoltória convexa do operador *HPR*.

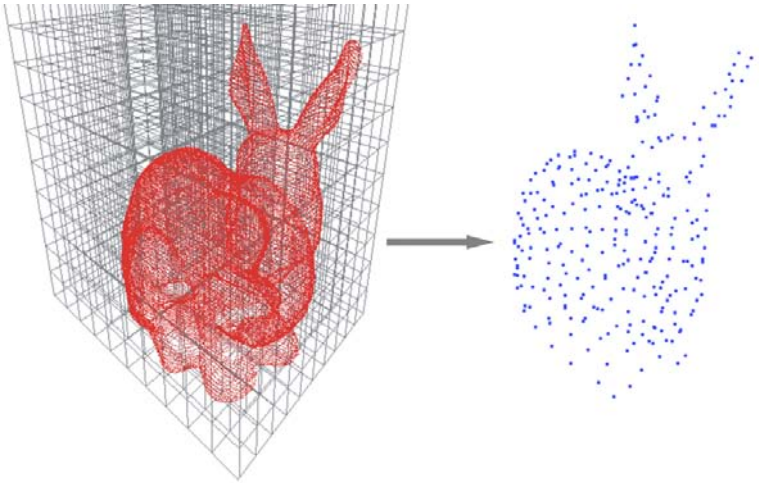
Apesar desta estrutura de dados ser construída muito rapidamente e de fácil implementação, não fornece a maneira mais eficiente de subdividir o espaço ao redor do modelo *3D*. A Figura 3.3 (a) mostra o resultado obtido usando um *Grade Regular* para calcular os centros de massas dos pontos dentro dos *clusters*. Os pontos apresentados são os pontos representativos utilizados no lugar dos pontos originais da nuvem de pontos para processar a envoltória convexa, acelerando assim o operador *HPR* original.

3.2 *HPR* Baseado em Octree

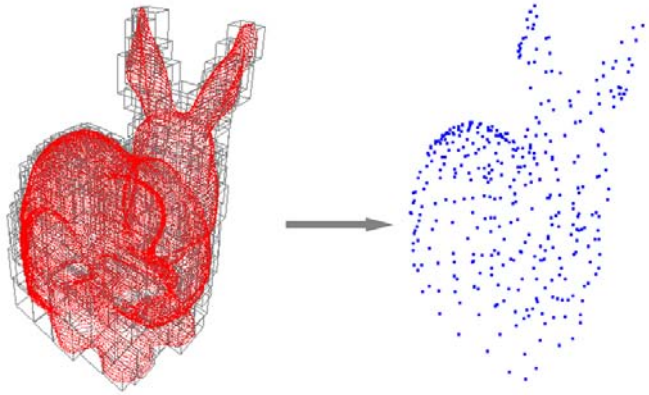
O Operador *HPR* baseado em *Octree* cria uma subdivisão espacial hierárquica de caixas alinhadas nos eixos para a nuvem de pontos. Esta estrutura de dados é criada utilizando um processo recursivo *top-down*. Para cada quadro da animação um nodo raiz é criado juntamente com uma caixa envoltória alinhada no eixo que contém os pontos da nuvem de pontos. O *cluster* raiz é recursivamente dividido em oito *clusters* chamados *octantes* e então a mesma regra de construção é aplicada a todos os oito *clusters*. Este processa recursivo acaba quando o tamanho do conjunto de pontos em cada *cluster* é menor ou igual ao número máximo de pontos permitidos em cada *cluster*. A estrutura de dados somente armazena o conjunto de pontos para cada *cluster* e o centro de massa dos mesmos quando o *cluster* é um *nodo* folha.

O número máximo de pontos permitidos por *cluster* e o tamanho do conjunto de pon-

tos representativos são inversamente proporcionais. Se a quantidade máxima de pontos por *cluster* é pequena, então o conjunto de pontos representativos é grande, resultando em um número grande de pontos que deverão ser processados na etapa do cálculo da envoltória convexa do operador *HPR*. A Figura 3.3 (b) ilustra uma *Octree* obtida a partir de uma nuvem de pontos, e os pontos representativos para cada *cluster* da nuvem de pontos.



(a) Centros de massa dos *clusters* gerado com operador *HPR* baseado em *Grade Regular*.



(b) Centros de massa dos *clusters* gerado com operador *HPR* baseado em *Octree*.

Figura 3.3: Pontos representativos gerados utilizando o operador *HPR* baseado em *Grade Regular* e *Octree* para *dataset Bunny*.

3.3 Implementação e Resultados

O operador *HPR* e os as duas variantes baseadas em *cluster* propostas foram implementados utilizando a linguagem *C++ ISO/IEC 14882:2003*, juntamente com a o pacote de cálculo de envoltória convexa *3D* da *Computational Geometry Algorithms Library* (*CGAL-3.3.1*) (BOARD, 2007) e *OpenGL* para renderização gráfica 3D. Foi utilizado um método baseado em física de *Soft Body* da biblioteca *AGEIA PhysX SDK 2.7* (INC., 2008) para animar a superfície dos objetos e criar uma nuvem de pontos dinâmica. A partir de uma superfície contínua, a deformação baseada em *Soft Bodies* muda as posições relativas dos pontos utilizando uma deformação elástica, sem causar fraturas ou

fusão (derretimento) da superfície. Esta deformação é armazenada como um conjunto de quadros de animação $Q = \{S_t | 1 \leq t \leq N\}$, onde para cada passo de tempo t existe uma superfície representada por um conjunto de pontos $S_t = \{p_i | 1 \leq i \leq n\} \subset \Re^3$, o qual é uma amostragem de uma superfície contínua. Como as operações baseadas em *cluster* geram informações de visibilidade aproximadas, torna-se necessário utilizar uma métrica de comparação baseada em imagem para estimar a perda de informação visual. Para realizar esta tarefa foi utilizada uma ferramenta para avaliação perceptual chamada *Perceptual Diff* (UTILITY., 2008), o qual simula o sistema visual humano para perceber diferenças existentes entre imagens. Este método mostra a diferença nas áreas com o maior percentual de erro, enquanto ignora regiões com diferenças aceitáveis (Figura 3.4). A diferença relativa entre imagens é calculada como se segue. Primeiro é contado o número de *pixels* visíveis para todas as imagens geradas pelo operador original. Então é computado o número de *pixels* percentualmente diferentes comparado com todas as imagens geradas pelo operador original e suas imagens correspondentes geradas pelas abordagens baseadas em *cluster*. A divisão entre este número e o número de *pixels* visíveis corresponde à diferença relativa.

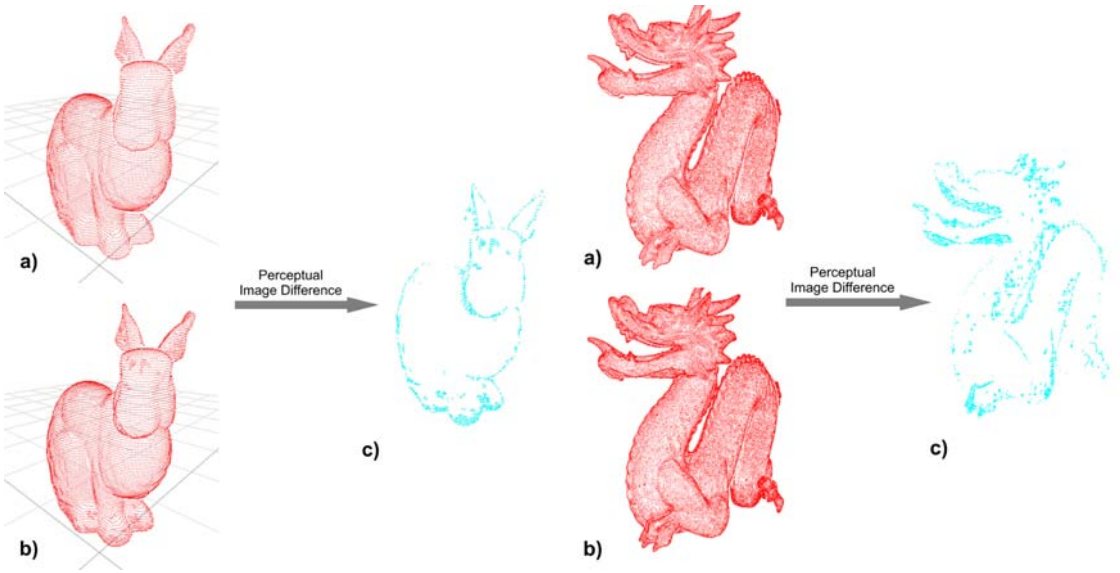


Figura 3.4: Processo de diferença perceptual de imagens. a) Resultado do operador *HPR* original. b) Resultado do operador *HPR* baseado em *Octree*. c) A diferença perceptual entre a imagem **a** e a imagem **b**.

Todos resultados de performance e imagens foram gerados em uma máquina com processador *QuadCore Intel Core 2 Quad Q6600*, *2400 MHz*, *4.0 GB RAM* de memória e uma placa gráfica *NVIDIA GeForce 8600 GT* (*256 MB*). Para gerar os resultados primeiramente foi utilizado um algoritmo de cálculo de envoltória convexa com construção estática (*CGAL::convex_hull_3*) (HERT; SCHIRRA, 2007) e o cálculo utilizando ponto flutuante de precisão simples. A função para o cálculo da envoltória convexa da biblioteca *CGAL* usa como base o algoritmo *Quickhull* (BARBER; DOBKIN; HUHDANPAA, 1996), o qual, para *datasets* dinâmicos com muitos pontos, se tornou muito instável e freqüentemente parava de funcionar ou não respondia conforme as suas especificações, entrando em laços infinitos. Para resolver este problema, foi utilizando um algoritmo incremental para o cálculo da envoltória convexa (HERT; SEEL, 2007), pois este método é mais estável do que o anterior.

Para analisar adequadamente as comparações perceptuais de imagens e de desempenho dos operadores baseados em *cluster* propostos foram usadas três diferentes nuvens de pontos dinâmicas. Os *datasets Bunny*, *Armadillo* e *Dragon* foram utilizados como modelos de base para criar *datasets* de nuvens de pontos dinâmicas em um mundo virtual baseado em física com o comportamento de *Soft Bodies* de acordo com as leis da mecânica clássica (*AGEIA PhysX SDK 2.7* (INC., 2008)). Estas animações baseadas em pontos foram criadas em uma fase de pré-processamento e armazenadas em arquivos de animação baseada em pontos. Cada quadro da animação é renderizado com o operador original e outros dois operadores baseados em *clusters*. A aplicação dos métodos baseados em *clusters* em visualização direta para nuvens de pontos dinâmicas pode ser vista na Figura 3.5.

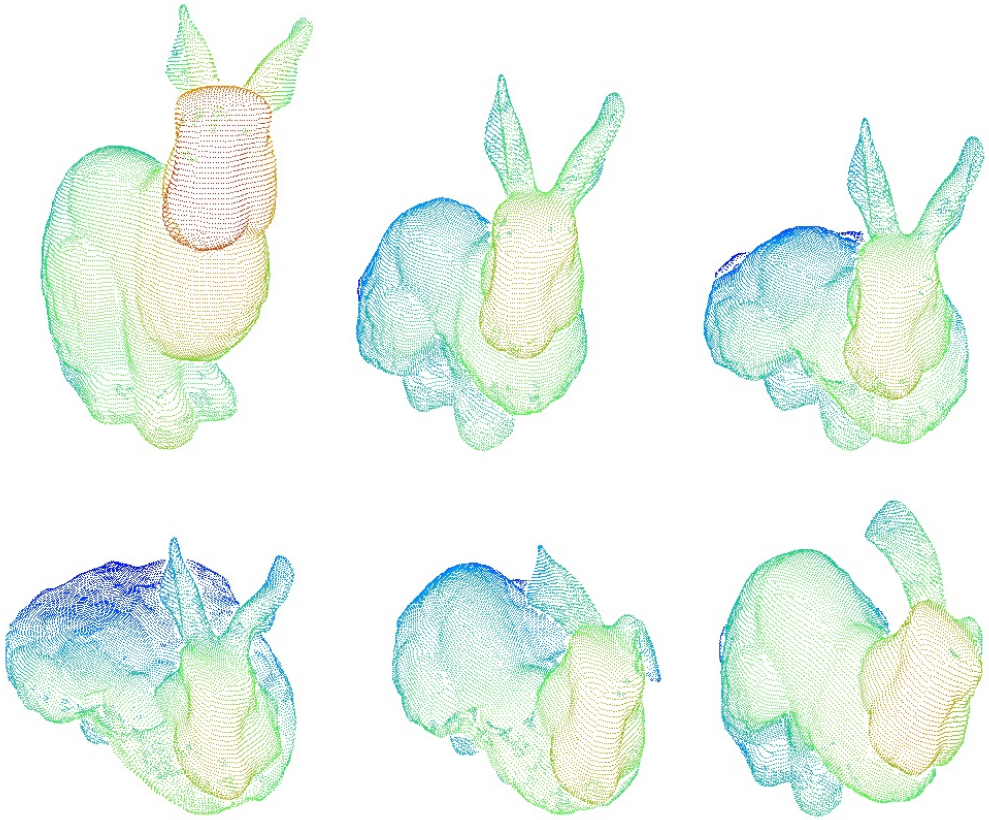


Figura 3.5: Visualização direta para a nuvem de pontos dinâmica do *dataset Bunny* utilizando o operador *HPR* baseado em *Octree*.

Todas as nuvens de pontos dinâmicas geradas contêm 150 quadros. As animações utilizando os modelos *Bunny*, *Armadillo* e *Dragon* têm respectivamente 34834, 172974 e 435545 pontos por quadro. As diferenças perceptuais foram computadas na resolução 1024×768 *pixels* para o *dataset Dragon* e *Armadillo*, e 640×480 para o *dataset Bunny*.

As tabelas A.1, A.2 e A.3 do anexo A descrevem os parâmetros de configuração utilizados para todos os operadores *HPR* implementados (ponto de visão, logaritmo do raio R usado no cálculo do *Spherical Flipping* e o critério de geração dos pontos representativos - tamanho do *Grade* ou o número máximo de pontos por *nodo* da *Octree*), estatísticas das nuvens de pontos dinâmicas (número de pontos removidos e visíveis utilizando os operadores, e porcentagem de pontos visíveis), erro perceptual (número de *pixels* diferentes e porcentagem relativa ao tamanho da janela) e medidas de performance em SPQ (segundos

por quadro).

O operador *HPR* baseado em *Grade Regular* obteve, em média, 3.7 mais desempenho e uma média de erro perceptual de 15.04% comparado ao operador original. O operador *HPR* baseado em *Octree* obteve uma média de ganho de velocidade de 4.35 vezes e uma média de erro perceptual de 13.96% comparado ao operador original.

3.4 Sumário

Este capítulo apresentou duas maneiras de tornar o operador *HPR* original mais eficiente na fase do cálculo da envoltória convexa do algoritmo. As duas abordagens de otimização utilizam subdivisão espacial para escolher pontos mais representativos dentro do conjunto de pontos do modelo 3D. Dentro de cada *cluster* gerado pelos métodos de subdivisão espacial *Grade Regular* e *Octree* é computado o centro de massa dos pontos contidos neste *cluster*, o qual é assumido como uma boa estimativa da visibilidade dos pontos associados a este *cluster*. Desta forma, as duas abordagens implementadas a partir do operador original geram menos pontos a serem tratados pela etapa de cálculo de envoltória convexa. Este procedimento gera uma troca entre desempenho e precisão do resultado gerado pelo algoritmo. O algoritmo pode tratar de forma mais eficiente nuvens de pontos dinâmicas, e ser aplicado à visualização direta das mesmas. O operador *HPR* baseado em *Octree* obteve melhores resultados comparado com o operador *HPR* baseado em *Grade Regular*, tanto em desempenho como em perda de informação de visibilidade.

4 RENDERIZAÇÃO BASEADA EM PONTOS

A utilização de pontos como principal primitiva gráfica vem sendo apontada como uma consistente ferramenta para a computação gráfica. Muita pesquisa vem sendo desenvolvida para encontrar formas eficientes para adquirir, representar, processar, renderizar e animar conjuntos de pontos (GROSS; PFISTER, 2007), devido a sua simplicidade conceitual e flexibilidade superior (BOTSCH et al., 2005).

Pode-se citar pelo menos dois importantes motivos para este interesse por computação gráfica baseada em pontos. Primeiro, é o grande aumento na complexidade de modelos poligonais em computação gráfica. A maioria dos algoritmos que trabalham com malhas de triângulos necessitam da consistência topológica de *2-manifold* (superfícies) (KOBBELT; BOTSCH, 2004). Como consequência, a manipulação e processamento de informação de conectividade em modelos poligonais grandes tornam-se caros. Por este motivo, autores como *Gross* e *P ster* vêm questionando a utilidade de polígonos como primitiva gráfica fundamental (GROSS; PFISTER, 2007).

Por último, os modernos sistemas de *Scanner 3D* coletam grandes volumes de amostras de geometria e aparência de objetos complexos. Estes volumes de informação são chamados de nuvens de pontos (ZWICKER et al., 2001). Nuvens de pontos são nada mais do que conjuntos de posições espaciais, que podem ser conectadas a outras informações do modelo *3D*, como cor e normal da superfície do objeto, por exemplo. As nuvens de pontos são uma amostragem da superfície contínua do objeto real digitalizado.

Muitos algoritmos de renderização baseada em pontos foram propostos, mas o uso de pontos como primitiva gráfica foi primeiramente proposto pelo trabalho pioneiro de *Levoy* e *Whitted* (MARC LEVOY, 1985). *Grossman* em 1998 (GROSSMAN, 1998) apresentou algoritmos para renderização de conjunto de pontos. Devido ao aumento da complexidade geométrica dos modelos, essas ideias vêm recentemente ganhando mais interesse.

Em 2000 um sistema chamado *QSplat* foi projetado (SZYMON RUSINKIEWICZ, 2000) para renderização baseada em pontos para grandes conjuntos de dados produzidos pelos modernos dispositivos de *Scanner 3D* do Projeto Michelangelo Digital. Este sistema utiliza *OpenGL* para aceleração por *hardware*, conseguindo um desempenho de 2.7 milhões de pontos por segundo (CMU et al., 2002). Em (DRETTAKIS, 2001), *Drettakis*, utilizou a primitiva gráfica *GL_POINTS* do *OpenGL* para renderizar geometria baseada em pontos gerada proceduralmente. Este sistema tinha um desempenho de cerca de 5 milhões de pontos por segundo. Ambos os métodos, (SZYMON RUSINKIEWICZ, 2000) e (DRETTAKIS, 2001), não trabalhavam com modelos texturizados.

Surface Splatting utilizando filtragem *elliptical weighted average* (*EWA*), que permite filtragem de textura anisotrópica, foi introduzido em (ZWICKER et al., 2001). Seu sistema de renderização foi melhorado por (CMU et al., 2002) com uma formulação de *EWA* no espaço de objeto e aceleração por *hardware*.

Splats poligonais foram utilizados em (CMU et al., 2002) e (PAJAROLA; SAINZ; GUIDOTTI, 2003). Esta abordagem rasteriza os *splats* mapeando uma textura de alfa elíptica sobre um triângulo e habilita o teste de alfa para descartar todos fragmentos que não pertencem ao interior dos *splats*. O problema desta técnica é que o número de vértices de triângulos que devem ser armazenados e processados é três vezes maior que o numero de *splats* do modelo.

Este capítulo apresenta uma técnica de renderização baseada em pontos chamada *Surface Splatting* usando abordagem de espaço de objeto com *Geometry Shader*. O método pode diretamente renderizar superfícies, em tempo real, de nuvens de pontos sem informação de conectividade. Para uma discussão mais detalhada sobre outros métodos de renderização baseada em pontos pode-se consultar os trabalhos (KOBBELT; BOTSCH, 2004) e (SAINZ; PAJAROLA, 2004).

4.1 *Surface Splatting*

Surface Splatting é uma técnica simples e eficiente para renderizar imagens de alta qualidade de superfícies amostradas por nuvens de pontos. Esta técnica utiliza o algoritmo de *Z-Buffer* para resolver a visibilidade da superfície. Ela também pode processar eficientemente conjuntos de pontos desordenados sem nenhuma estrutura adicional de aceleração, tais como estruturas de hierarquia espacial.

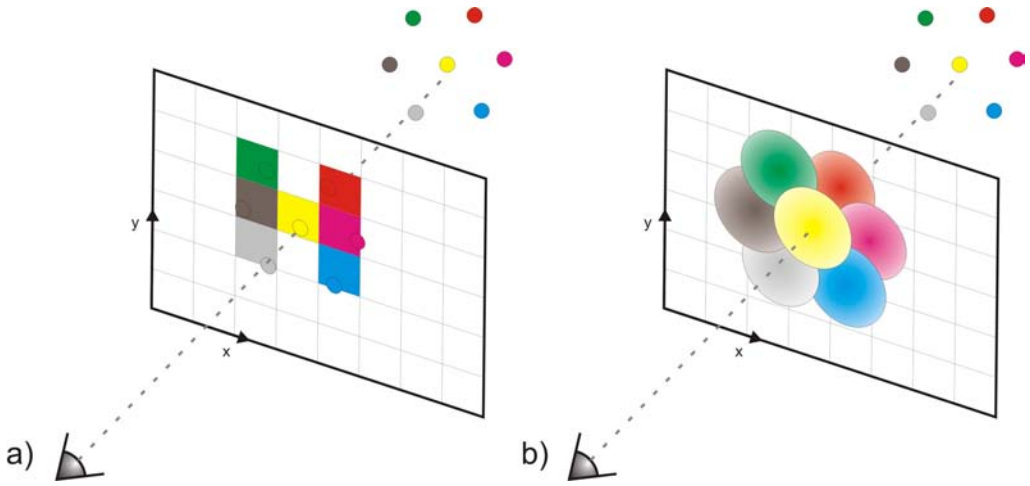


Figura 4.1: Renderização Baseada em Pontos. (a) Simples projeção e renderização de amostras de pontos. (b) *Surface Splatting* distribui a contribuição de cor aos *pixels* vizinhos.

A idéia básica do *Surface Splatting* é ilustrada na Figura 4.1. Uma abordagem simples para renderização baseada em pontos é projetar perspectivamente cada ponto 3D no plano de imagem como na Figura 4.1 a), e atribuir a cor do ponto para aos respectivos *pixels*. Evidentemente, isso produz buracos na imagem final se a superfície não for amostrada com uma densidade suficiente. Por outro lado, se mais de um ponto for projetado no mesmo *pixel*, o resultado de renderização depende da ordem em que os pontos foram projetados (GROSS; PFISTER, 2007).

O método *Surface Splatting* resolve esses problemas, como pode ser visto na Figura 4.1 b), cada ponto projetado distribui sua cor ao longo dos seus *pixels* vizinhos. Cada ponto está associado a uma função de *footprint* que calcula os pesos para a contribuição

de cor para os *pixels* vizinhos. Essas funções de *footprint* são geralmente suaves, com um decaimento crescendo rapidamente a partir do centro, como indicado na Figura 4.1b.

4.1.1 Implementação e Resultados

O algoritmo de *Surface Splatting* reconstrói uma superfície com amostragem de pontos não uniforme. Isto significa que estes métodos podem ser vistos como uma maneira de preencher os espaços vazios entre as amostras da superfície. Comumente os algoritmos de *Surface Splatting* são divididos em dois ou mais passos de renderização. Todos os passos trabalham com um tipo de primitiva baseada em ponto, sem conectividade explícita, chamada de *surfel* (***surf**ace **el**ement*). Os dois passos mais comuns na implementação destes algoritmos são comumente nomeados como *Visibility Splatting* e *Surface Reconstruction* (JACOBSSON, 2007).

Visibility Splatting decide quais *surfels* pertencentes a superfície estarão visíveis a partir do ponto de visão atual. *Surface Reconstruction* produz uma superfície com *surfels* suavizados com a ajuda da saída do primeiro passo.

Isto se procede da seguinte forma:

- Carregar o arquivo com os *surfels* do modelo;
- Utiliza o *Z-Buffer* para resolver o problema de visibilidade: *Visibility Splatting Pass*;
- Utiliza o *alpha blending* com uma textura de alfa com distribuição *Gaussiana* para construir a superfície: *Surface Reconstruction Pass*.

Pode-se dizer que os principais processos para uma implementação eficiente e renderização interativa são o uso do *Geometry Shader* para construir a geometria da superfície somente baseada nos *surfels* e a utilização de mapeamento de textura com *alpha blending* para aproximar um filtro de reamostragem *Elliptical Weighted Average* (*EWA*), no espaço de objeto. A textura com canal de alfa foi gerada utilizando uma função de distribuição *Gaussiana* bidimensional. Esta ideia é ilustrada na Figura 4.2.

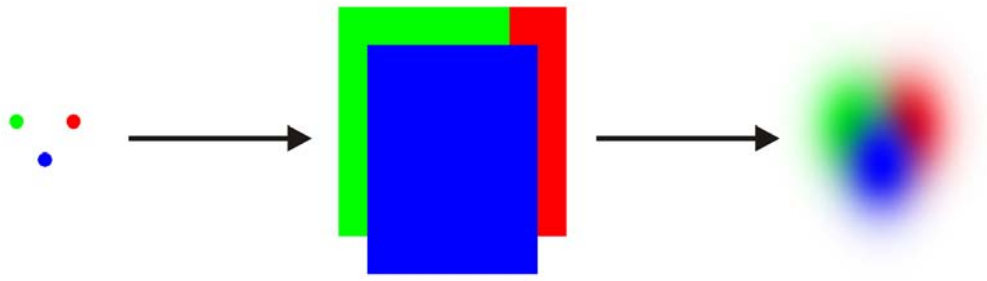


Figura 4.2: *Geometry Shader* para construir a geometria da superfície e mapeamento de textura com *alpha blending* para aproximar um filtro de reamostragem *EWA* no espaço de objeto.

O método foi implementado usando linguagem *C++ ISO/IEC 14882:2003*, *OpenGL* e *OpenGL Shading Language* (*GLSL*). Os resultados experimentais foram obtidos em uma máquina com processador *AMD Athlon^{TM}64 3500+, 2.21 GHz* com uma placa aceleradora *NVIDIA GeForce 8800 GTX 768 MB* e *2.0 GB RAM* de memória.

4.1.1.1 Surfels e Aquivo de Entrada

Além de posicionamento espacial, *surfels* podem possuir atributos tais como: profundidade, cor, vetor normal para orientação e outros (PFISTER et al., 2000). Neste trabalho um *surfel* contém os atributos de posição, normal, cor e raio de influência. A representação de um *surfel* pode ser vista na Figura 4.3.

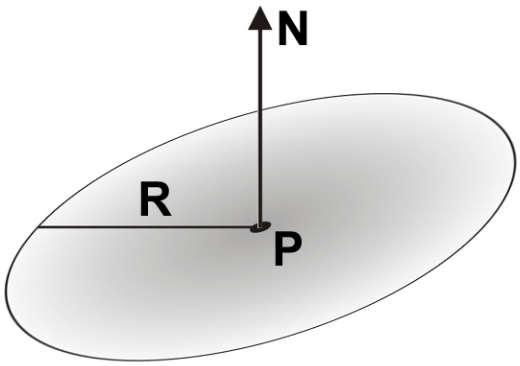


Figura 4.3: Representação de um *surfel* com atributos: P é a posição, N é a normal e R é o raio.

Os arquivos de dados de entrada dos modelos utilizados neste trabalho estão em formato *ASCII*, e contêm a descrição dos *surfels*. O primeiro número inteiro lido do arquivo é o número de *surfels* do modelo. Cada *surfel* é descrito pelos seguintes parâmetros: *x y z r g b nx ny nz radius* com (x, y, z) sendo a posição do *surfel*, (r, g, b) a cor do *surfel* (r, g e b devem estar no intervalo entre 0 e 1) , (nx, ny, nz) a normal já normalizada do *surfel* e *radius* o raio do disco utilizado para renderizar o *surfel*. Esta descrição corresponde ao formato de arquivo utilizado por *Bart Adams* no projeto *Surfel Viewer* (ADAMS, 2007).

Um exemplo de arquivo de entrada válido pode ser:

2
1 2 3 1 0 0 1 0 0 0.3
3 2 1 0 1 0 0 1 0 0.3

O exemplo acima contém dois *surfels*. O primeiro está na posição $(1, 2, 3)$, com cor vermelha e normal paralela ao eixo x. O segundo está na posição $(3, 2, 1)$ com cor verde e normal paralela ao eixo y. Os dois *surfels* possuem raio de tamanho 0.3.

4.1.1.2 Surfels e Geometry Shader

A partir da *API Direct3D 10* (BLYTHE, 2006) tornou-se possível emitir ou apagar vértices passados para a *GPU*. Existe um novo módulo, chamado *Geometry Shader*, entre o *Vertex Shader* e o *Fragment Shader*, que pode modificar a geometria passada do primeiro para o segundo módulo.

Com este novo *pipeline*, pode-se passar para a *GPU* somente vetores com a descrição dos *surfels*, com os atributos de posição, normal, cor e raio para construir a superfície dos modelos, criando em tempo real no *Geometry Shader* primitivas *quads*, que representarão os *surfels*.

No *Geometry Shader*, estes *quads* são gerados por dois triângulos conectados, compartilhando vértices (*triangle strip*). Na Figura 4.4 este processo pode ser observado. Dado o ponto central, a normal e raio do *surfel* no espaço de objeto, vindo do *Vertex*

Shader, pode-se criar e orientar mais quatro vértices no *Geometry Shader* e emitir uma primitiva de *triangle strip* para representar o *surfel*.

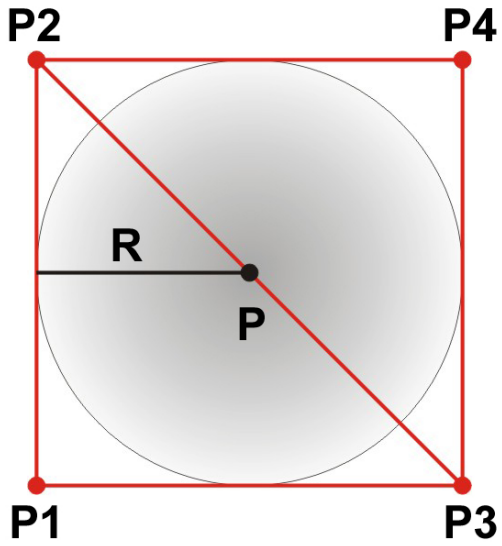


Figura 4.4: Primitiva *triangle strip* gerada no *Geometry Shader*. P é a posição central e R é o raio do *surfel*. $P1$, $P2$, $P3$ e $P4$ são os novos vértices criados.

4.1.1.3 Passo de Visibilidade

O *Visibility Splatting* é usado para remover os *surfels* da face frontal que não pertencem a superfície visível através do respectivo ponto de visão (JACOBSSON, 2007). Primeiramente, o *Backface Culling* precisa ser habilitado. Então, os *surfels* são preenchidos com um material opaco e mandados através do *pipeline* para serem escritos no *Depth Buffer* modificado através de um deslocamento positivo em z. Este deslocamento irá reduzir o fenômeno de *Z-Fighting* para o próximo passo de renderização (Ver em Figura 4.5 **a)**).

Este processo gera uma imagem de profundidade sem buracos da superfície para que no próximo passo de renderização os fragmentos de *surfels* que estão cobertos por outros *surfels* sejam removidos.

4.1.1.4 Passo de Reconstrução de Superfície

Neste passo do algoritmo (Figura 4.5 **b)**), todos *surfels* são enviados para o *pipeline* de renderização mais uma vez com *Backface Culling* habilitado. Mas, agora a renderização não é feita no *Depth Buffer*, e sim no *Color Buffer*. Deste modo todos *surfels* (ou partes de *surfels*) que não fazem parte da face frontal da superfície visível não serão renderizados no *Color Buffer*.

Para conseguir um preenchimento suavizado dos *surfels* é necessário renderizar no *Color Buffer* discos que são transparentes nas arestas e opacos no ponto central. Com este tipo de preenchimento e a sobreposição dos *surfels*, será possível produzir um mistura de cores suavizada entre os *surfels* vizinhos na superfície utilizando o *alpha blending*.

Esta suavização é produzida utilizando um filtro *Gaussiano* como *kernel* de suavização para cada *surfel*. Cada *kernel* é pré-calculado e colocado em uma textura com canal de alfa, que são mapeados no *surfel*.

Também neste passo de renderização podemos simular diferentes efeitos de ilumina-ção através da superfície. Para testar este conceito foi implementado um algoritmo de iluminação local *Gouraud Shading*.

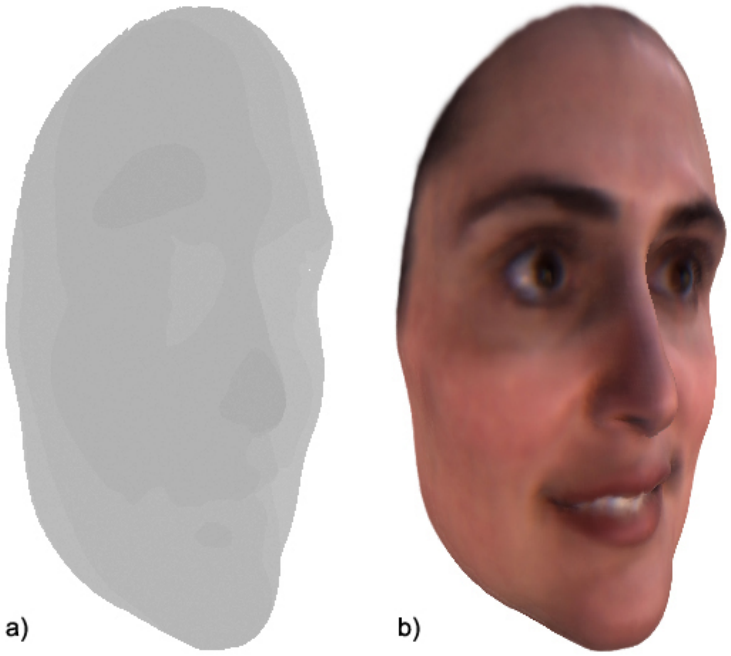


Figura 4.5: **a)** Passo de Visibilidade: *Depth Buffer* com um deslocamento positivo em z. **b)** Passo de Reconstrução de Superfície: *Color Buffer* com a superfície suavizada.

4.1.1.5 Resultados

Surface Splatting utilizando *Geometry Shader* provou ser uma abordagem viável para geração de imagens de alta qualidade em tempo interativo para renderização baseada em pontos. Com este método foi possível renderizar uma média de 15 milhões de *surfels* por segundo em um *Frame Buffer* de resolução de *512 x 512 pixels*.

Na Figura 4.6 pode-se observar o processo de gerar a superfície contínua do modelo *3D* amostrado por *surfels* através do uso do *Geometry Shader*, o qual permite criar geometria dentro do *pipeline* de renderização da *GPU*. As Figuras B.1, B.2, B.3, B.4, B.5, B.6 e B.7 do anexo B mostram alguns resultados finais obtidos com o método.

Figura 4.6: Modelo Face. a) nuvem de pontos. b) geometria dos *surfels*. c) imagem final.

4.2 Sumário

Este capítulo apresenta uma técnica de renderização baseada em pontos acelerada por *hardware* chamada *Surface Splatting*, que renderiza superfícies em tempo real a partir de uma nuvem de pontos sem informação de conectividade.

Esta é uma técnica de renderização de dois passos, onde o *Geometry Shader* é usado para construir a geometria da superfície e mapeamento de textura com *alpha blending* para aproximar um filtro de reamostragem *EWA* no espaço de objeto. A qualidade visual deste método é comparável com os sistemas de renderização baseados em pontos já existentes.

5 *HPR* BASEADO EM *SPLATTING*

O uso de *GPUs* programáveis em computação gráfica baseada pontos foi explorada por diversos autores nos últimos anos, entretanto a maioria dos trabalhos empregam com variações do método de *Surface Splatting* para *rendering* (SZYMON RUSINKIEWICZ, 2000; CMU et al., 2002; DRETTAKIS, 2001; CMU et al., 2002).

Uma maneira para calcular a visibilidade de uma nuvem de pontos é reconstruir a superfície implícita na nuvem utilizando uma malha de triângulos (HOPPE et al., 1992). Porém, isto gera problemas adicionais, pois são necessários as informações sobre os vetores normais da superfície para a construção da malha de triângulos. Também este processo é muito custoso para a sua utilização em aplicações em tempo real. Deseja-se em um operador de remoção de pontos ocultos é que seja eficiente que utilize a menor quantidade possível de parâmetros de entrada (Figura 5.1).

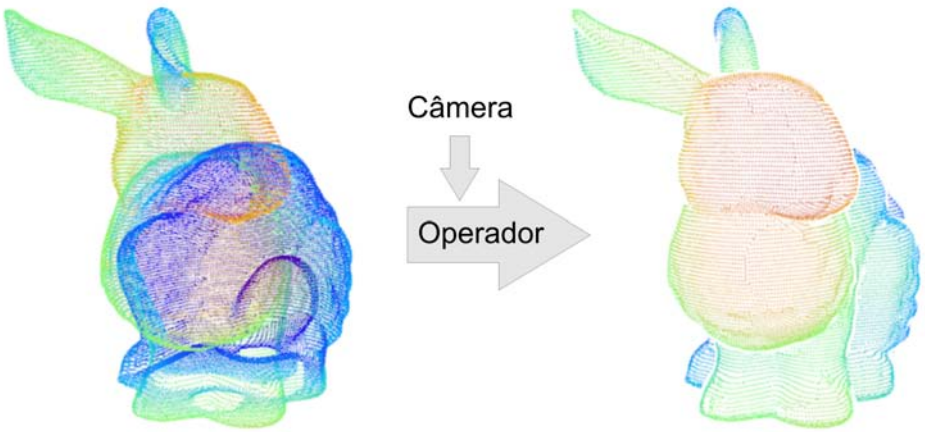


Figura 5.1: Entrada do operador: nuvem de pontos sem informação de conectividade e a posição da câmera. Saída do operador: pontos visíveis a partir da posição da câmera.

Este capítulo investiga a visibilidade de nuvem de pontos utilizando aceleração por *hardware*. É apresentado um método aproximativo eficiente para remoção de pontos ocultos em uma nuvem de pontos sem informação de conectividade. A principal motivação para o desenvolvimento deste novo método é a baixa eficiência demonstrada pelos métodos existentes para o cálculo de visibilidade de nuvens de pontos em aplicações de tempo real com nuvens muito densas. Os objetivos específicos deste trabalho são: (1) implementar, de forma eficiente, um método de visualização em tempo real para superfícies amostradas por nuvens de pontos sem informação de conectividade, e (2) um operador para remoção de pontos ocultos pela superfície implícita, sem a construção de estruturas topológicas ou de hierarquia espacial.

O operador introduzido neste capítulo resolve o problema de visibilidade para uma nuvem de pontos utilizando etapas de *rendering* e trabalhando no espaço de imagem. Logo, o algoritmo é dependente de parâmetros de projeção e da resolução da janela de renderização. A idéia básica deste método é a utilização do algoritmo de *Z-Buffer* dentro do passo de visibilidade do *Surface Splatting* para remover pontos ocultos a partir do ponto de observação. Para isto é reconstruída uma aproximação da nuvem no *depth buffer* utilizando cada ponto da superfície como base para criar impostores alinhados com a direção da visão da câmera.

Os impostores devem preencher o *depth buffer* de forma que este contenha a menor quantidade possível de buracos e sobreposições de impostores. Buracos são criados quando impostores são gerados com um raio menor que o raio ótimo. No *depth buffer* os buracos produzirão falsos positivos, ou seja, pontos da nuvem marcados como visíveis que deveriam ter sido marcados como ocultos. Sobreposições são geradas quando impostores possuem um raio maior que o raio ótimo. No *depth buffer* eles produzem falso negativos, ou seja, pontos marcados como ocultos que deveriam ter sido marcados como visíveis. Este processo de sobreposição gera um efeito de silhueta na imagem final. O problema de silhuetas foi minimizado com a aplicação de um operador morfológico de erosão no *depth buffer*.

Com os valores do *depth buffer* definidos, representando a porção visível da superfície implícita, é possível descartar pontos ocultos pela superfície representada utilizando uma *GPU* programável, comparando o valor de profundidade dos pontos projetados com o valor no *depth buffer*. Para obter os pontos visíveis, marcados pela *GPU*, é necessário desprojetá-los e trazê-los para a *CPU*. Este processo é feito utilizando mais um passo no algoritmo, cujas etapas serão apresentadas nas próximas seções.

5.1 Operador *HPR* Acelerado por *Hardware*

Dado $P = \{p_i \mid 1 \leq i \leq n\} \subset \Re^3$, um conjunto de n pontos representando uma amostragem de uma superfície contínua S, o objetivo do operador *HPR* acelerado por *hardware* é determinar, $\forall p_i \in P$, se p_i é ou não visível a partir de uma posição de câmera C.

Abordagens diretas para resolver o problema de visibilidade em nuvens pontos são falhas. Calcular a intersecção da linha gerada entre C e p_i não é muito útil, pois, exceto em casos especiais, a linha não irá interceptar outros pontos. Caso estivesse disponível a informação de vizinhança para cada ponto, seria possível estimar o vetor normal n_i e o raio r_i de separação média entre o ponto e os vizinhos do ponto amostrado. Com estas informações extras, para cada ponto p_i, seria possível criar um disco circular de raio r_i com orientação n_i no espaço do objeto, representando um *surfel*. Com o conjunto de *surfels* calculado pode-se produzir uma superfície sem buracos no *depth buffer*, sendo assim possível remover corretamente os pontos ocultos da nuvem de pontos utilizando o algoritmo de *Z-Buffer*. Este processo é ilustrado em *2D* na Figura 5.2.

Obviamente a visibilidade correta da nuvem de pontos neste método está diretamente relacionada com a densidade da amostragem da nuvem de pontos. Supondo que P é uma amostragem de densidade ρ da superfície S, se cada ponto $p_i \in P$ for substituído por um *surfel* de raio ρ e vetor normal n_i, a união de todos os *surfels* pode ser considerada uma discretização da superfície S, contendo todos os pontos P. Então, um ponto p_i pode ser considerado visível a partir de C se e somente se, não existe intersecção do segmento de reta ligando C e p_i com qualquer *surfel* gerado por p_j onde $j \neq i$.

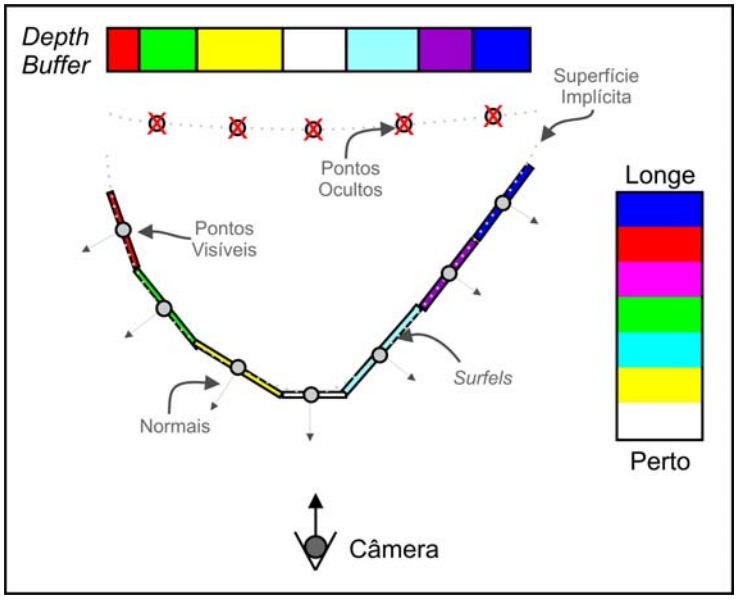


Figura 5.2: Utilização de *surfels* para reconstruir, no *depth buffer*, a superfície implícita, para remoção dos pontos ocultos com a utilização do algoritmo de *Z-Buffer*.

Como o vetor normal e raio do *surfel* não estão disponíveis, outra alternativa para gerar a superfície visível implícita na nuvem de pontos teve de ser desenvolvida. Aproximar a projeção dos *surfels* no *depth buffer* utilizando impostores alinhados com a direção da visão da câmera mostrou-se uma solução simples e eficiente.

Se fosse possível computar eficientemente o raio de cada impostor independentemente, de forma que nenhum dos impostores se sobrepusessem durante sua projeção no *depth buffer*, e que nenhum dos impostores deixasse buracos durante sua projeção no *depth buffer*, poderia-se construir um *depth buffer* bastante próximo aquele gerado com *surfels*. Buracos e sobreposições no *depth buffer* geram falhas na inferência de visibilidade de pontos, os quais são exemplificados em *2D* na Figura 5.3.

A determinação do raio ótimo de cada impostor não é possível de forma eficiente. A escolha de um raio R único para todos os impostores deve ser feita de maneira que R minimize o erro na superfície construída no *depth buffer*, gerando menos falsos positivos e falsos negativos na inferência de visibilidade. Uma estratégia é escolher um tamanho de raio R de forma que os impostores preencham completamente o *depth buffer*, sem deixar buracos.

O algoritmo de *Z-Buffer* consegue, com o *depth buffer* gerado, remover a maior parte dos pontos que não deveriam estar visíveis. Porém, devido à falta de adaptabilidade no tamanho dos raios dos impostores, as partes do modelo implícito com curvatura acentuada, ou na periferia da projeção do modelo no plano de imagem ficam incompletas por causa dos pontos erroneamente descartados, produzindo assim um efeito de silhuetas na imagem final.

Para reduzir esse efeito de silhuetas foi projetado um passo a mais no operador que reduz as bordas internas na imagem projetada no *depth buffer*. A aplicação de um operador morfológico de erosão 3 x 3 modificado no *depth buffer* mostrou-se eficaz, gerando bons resultados. Esse passo extra do operador reduziu seu erro, produzindo uma classificação dos pontos mais próximas do ideal.

Como o operador trabalha no espaço de imagem juntamente com o algoritmo de *Z-*

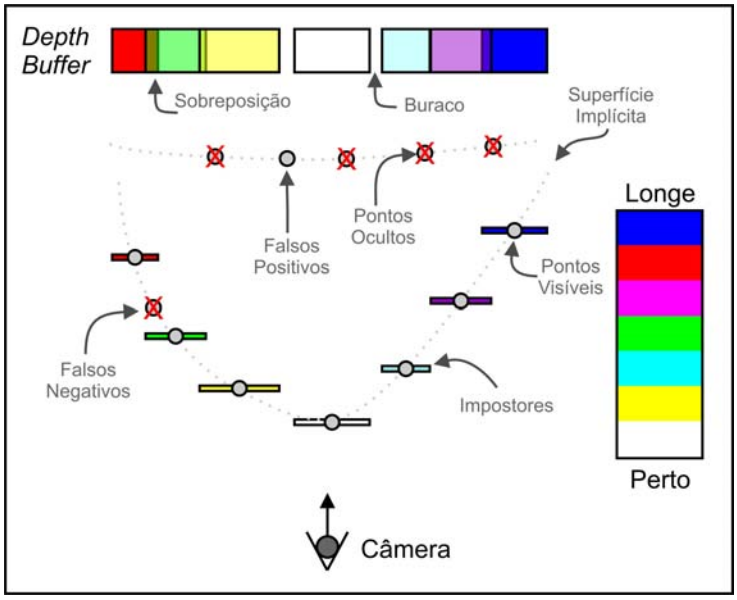


Figura 5.3: A utilização de impostores para reconstruir a superfície implícita dentro do *depth buffer* gera alguns problemas. Quando o raio de cada impostor é menor que o raio ótimo buracos são produzidos. Quando o raio de cada impostor é maior que o raio ótimo ocorre sobreposição.

Buffer, ele é dependente também dos parâmetros da matriz de projeção perspectiva M:

$$M = \begin{pmatrix} \frac{f}{aspect\_ratio} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & \frac{zfar+znear}{znear-zfar} & \frac{2*zfar*znear}{znear-zfar} & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}$$

Onde $f = \cot\left(\frac{fov\_y}{2}\right)$, fov_y é o ângulo do campo de visão na direção y em graus, $aspect\_ratio$ é a relação entre largura e altura, $znear$ é a distância do plano de corte mais próximo, $zfar$ a distância do plano de corte mais distante.

O resultado do método também é afetado pela precisão do *depth buffer*. Quanto maior a razão $\frac{zfar}{znear}$, menos eficaz será a distinção da profundidade entre superfícies que estão próximas uma das outras. Este efeito é conhecido como *Z-Fighting* e é comumente visto quando primitivas rasterizadas possuem valores similares no *depth buffer*. Este fenômeno faz com que *pixels* pseudo-randômicos (artefatos) sejam renderizados com a cor de uma primitiva ou de outra primitiva, de forma não determinística (Figura 5.4).

O operador *HPR* acelerado por *hardware* também é dependente da largura ($width$) e altura ($height$) da janela de renderização. Quanto maior a área de renderização, menor o percentual de erro do operador. Cada primitiva de ponto renderizada num *frame buffer* é sempre rasterizada em um *pixel*, logo um *frame buffer* maior irá produzir menos sobreposição de fragmentos quando uma nuvem de pontos é muito densa.

Algumas propriedades são desejáveis para o operador de remoção de pontos ocultos acelerado por *hardware*:

- Gerar o resultado mais próximo possível do correto.
- Evitar pré-computação, ou cálculo dos vetores normais localmente.

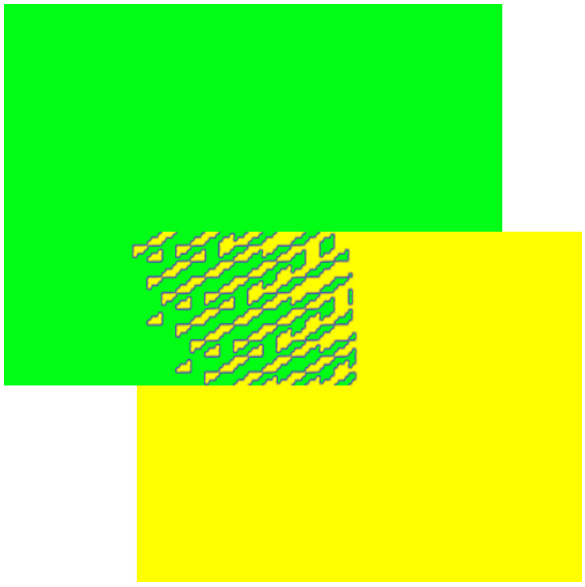


Figura 5.4: Exemplo do efeito de *Z-Fighting* gerado por duas primitivas coplanares.

- Reduzir o efeito de silhuetas.
- Ser eficiente ao ponto de poder ser utilizado em aplicações de visualização em tempo real.

Estas propriedades foram constatadas no operador implementado neste trabalho. O operador pode ser resumidamente descrito por um algoritmo de quatro passos:

1. Reconstruir a parte da superfície visível representada pelos pontos dentro do *depth buffer* utilizando impostores;
2. Reduzir o efeito de silhueta aplicando um operador morfológico modificado de erosão no *depth buffer*;
3. Remover pontos ocultos utilizando o algoritmo de *Z-Buffer* e o *depth buffer* gerado;
4. Desprojetar pontos e copiá-los da *GPU* para a *CPU*.

O operador foi implementado utilizando a linguagem de programação *C++ ISO/IEC 14882:2003*, a biblioteca *OpenGl* e a linguagem de *shader GLSL*. Todos os passos do algoritmo serão aprofundados nas próximas seções.

5.1.1 Construção dos Impostores

Como visto anteriormente, o algoritmo de *Z-Buffering* é utilizado para remover os pontos invisíveis a partir da posição da câmera. O *depth buffer* deve ser preenchido de forma que não contenha buracos. Para cada ponto da nuvem de pontos, um impostor alinhado com o vetor direção da câmera é gerado. Os impostores podem ser gerados de diversas formas, já bastante investigadas em trabalhos de *Surface Splatting* baseados em *GPU* (GROSS; PFISTER, 2007).

Uma opção é utilizar impostores poligonais criados na *CPU* e transferidos para a *GPU*. Para cada ponto, duas primitivas *GL_TRIANGLE_STRIP*, com o vetor normal alinhado com o vetor direção da câmera, devem ser geradas (Veja em Figura 5.5).

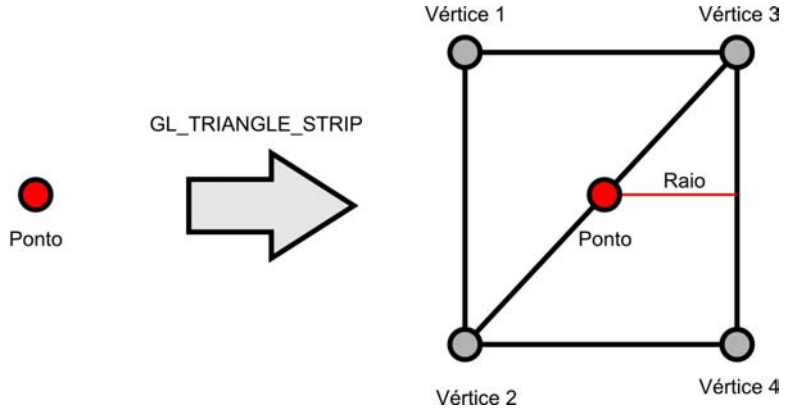


Figura 5.5: Geração de impostores utilizando *GL_TRIANGLE_STRIP* na *API OpenGL*.

Uma vantagem deste processo é requerer somente funcionalidades básicas do pipeline *OpenGL*, o que simplifica a implementação em *hardware* gráfico antigo. Entretanto, o número de vértices gerados, armazenados e processados pelo *pipeline* gráfico é quatro vezes maior do que a quantidade de pontos da nuvem. A principal desvantagem é o custo de transferência da malha de polígonos da memória da *CPU* para a memória da *GPU*. Para reduzir este gargalo de transferência de dados, é possível utilizar *Vertex Buffer Objects*, uma técnica de otimização de performance do *OpenGL*.

Uma forma mais eficiente de gerar impostores poligonais alinhados com o vetor direção da câmera é utilizar o *Geometry Shader* existente no *pipeline* das placas gráficas modernas, o qual torna possível criar geometria dentro da placa gráfica. Para cada ponto enviado para a *GPU*, criam-se dois triângulos alinhados com o vetor da direção da câmera utilizando a primitiva *triangle strip*. Este processo é muito similar ao processo, descrito no capítulo anterior, desenvolvido para gerar os *surfels* para a renderização baseada em *Surface Splatting*.

Visando uma forma de armazenamento de primitivas mais compacta e com melhor performance, os impostores podem ser criados com a utilização de quadrados alinhados com a imagem, ou seja, alinhados com o vetor direção da câmera no espaço de imagem. Pode-se criar estes impostores com o *OpenGL* utilizando a primitiva *GL_POINT_SPRITE*, ou ainda a primitiva *GL_POINTS* ajustando o seu tamanho com a utilização o parâmetro *gl_PointSize* do *shader* de vértice. Os últimos dois métodos utilizam o *pipeline* gráfico do *OpenGL* de forma eficiente, pois desenham os impostores transmitindo à *GPU* somente os pontos da nuvem. Ainda, a função *glDrawArrays(GL_POINTS,...)* da *API OpenGL* envia para a placa gráfica todos pontos de uma única vez.

Para transformar as primitivas de ponto em impostores acelerados por *hardware* é necessário renderizar os pontos com a primitiva *GL_POINT_SPRITE* habilitada e especificar os parâmetros que ajustam o tamanho dos impostores em relação à distância da câmera. O tamanho do impostor é dado por s, calculado pela seguinte expressão:

$$s = p * \sqrt{\frac{1}{a + b * d + c * d^2}}$$

Onde s é o tamanho do impostor desenhado, p é o tamanho do ponto ajustado por *glPointSize*, a, b, c são os coeficientes especificados no ajuste dos parâmetros de atenuação de distância (*GL_POINT_DISTANCE_ATTENUATION*) e d é a distância da câmera.

O *hardware* gráfico vem se tornando mais flexível durante os últimos anos, graças à introdução do *pipeline* programável, através da especificação de programas ou *sha-*

ders de fragmento e de vértice, e à utilização de texturas com suporte a valores de ponto flutuante com 32 *bits*. Isso possibilitou a utilização da primitiva *GL_POINTS* com o tamanho em *pixels* ajustado dentro de um *shader* de vértice para criar os impostores alinhados com o vetor direção da câmera. Primeiramente é necessário habilitar o parâmetro *GL_VERTEX_PROGRAM_POINT_SIZE*, antes do desenho dos pontos utilizando a função *glDrawArrays*. Isso permite, dentro do *shader* de vértice modificar o tamanho do ponto de forma independente, ou seja, para cada ponto é possível ajustar um tamanho diferente.

A seguinte fórmula é utilizada dentro do *shader* de vértice para definir o tamanho dos pontos em *pixels* em relação a distância da câmera.

$$gl\_PointSize = \frac{pixels\_per\_radian * point\_diameter}{0.5 * length(camera - gl\_Vertex)}$$

O parâmetro $point\_diameter$ é o raio do impostor, $length(camera - gl\_Vertex)$ é a distância da câmera até o centro do impostor e $pixels\_per\_radian$ é a quantidade de *pixels* existente no plano de projeção dentro de um ângulo de 1 radiano. O parâmetro $pixels\_per\_radian$ é computado segundo a fórmula a seguir:

$$pixels\_per\_radian = \frac{w}{fov\_y * \frac{Pi}{180}}$$

fov_y é o ângulo de abertura da câmera em relação a y. É dado em graus e transformado em radianos multiplicando-o por $\frac{Pi}{180}$. O parâmetro w é o tamanho em *pixels* da largura da janela de renderização.

O resultado do processo de geração dos impostores utilizando a primitiva *GL_POINTS* com o tamanho em *pixel* ajustado dentro de um *shader* de vértice pode ser visto na Figura 5.6. Na Figura 5.6 a) os pontos não preenchem completamente o *depth buffer*. Já na Figura 5.6 b) o tamanho dos pontos está ajustado corretamente de forma que o *depth buffer* é completamente preenchido.

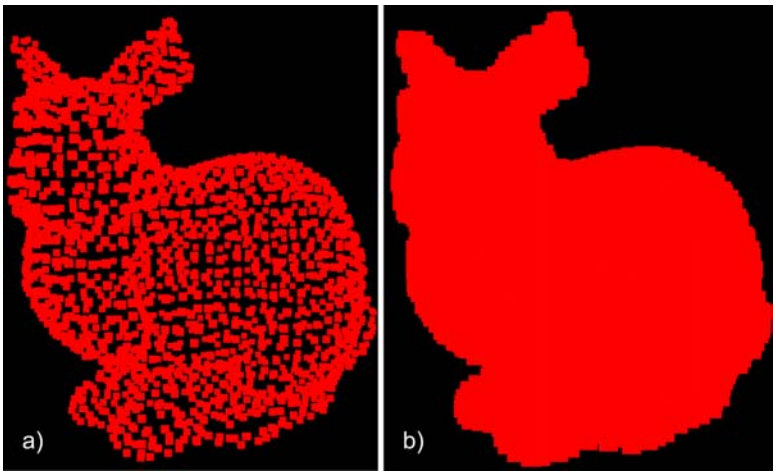


Figura 5.6: Modelo *Bunny* com os impostores gerados utilizando a primitiva *GL_POINTS* junto com o controle do parâmetro *glPointSize* do *GLSL*. Ajuste ruim do parâmetro em a) e ajuste correto em b)

5.1.2 Ajustando o Efeito de Silhueta

A utilização de um raio único para todos os impostores gerou problemas nas partes do modelo onde existe grande curvatura em relação ao vetor da direção da câmera. A técnica de remover os pontos ocultos com a utilização do algoritmo de *Z-Buffering* mostrou-se bastante eficiente, porém, devido à falta de adaptabilidade no tamanho dos impostores, um efeito de silhuetas na imagem final é gerado. Esse efeito ocorre quando são retirados no *depth buffer* pontos das bordas do modelo que deveriam estar visíveis.

Para reduzir este efeito de silhueta é necessário criar impostores com raios menores quando estão em uma região de grande curvatura. Como a geração dos impostores é feita no espaço de imagem utilizando a primitiva *GL_POINTS* com o tamanho em *pixel* ajustado dentro de um *shader* de vértice, torna-se caro calcular informações sobre a curvatura do modelo. Logo, é necessária uma abordagem no espaço de imagem para simular o efeito de adaptabilidade dos impostores.

A primeira intuição foi trabalhar no espaço de imagem aplicando uma operação morfológica de erosão no *depth buffer*. Uma função erosão é um operador básico na área de morfologia matemática que trabalha analisando valores da vizinhança dos *pixels* operados. O efeito básico deste tipo de filtragem em imagens *2D* é criar uma desgaste nas bordas de regiões de *pixels* de primeiro plano. *Pixels* de primeiro plano são aqueles que possuem valor maior, e *pixels* de segundo plano são aqueles que possuem valor menor dentro da imagem. Deste modo, áreas onde existem *pixels* de primeiro plano terão seu tamanho reduzido, enquanto áreas onde existem *pixels* de segundo plano serão aumentadas.

Para implementar esta abordagem foi criado um passo extra no operador utilizando um *shader* de fragmento em *GLSL*. Este *shader* implementa um operador morfológico de erosão que analisa a vizinhança de cada *pixel* da imagem gerada pelos impostores dentro do *depth buffer*. A erosão de uma imagem em tons de cinza f por um elemento estruturante plano b em uma localização (x, y) dentro desta imagem, é definida como o valor mínimo da imagem na região coincidente com b quando a origem de b é em (x, y). Na forma de equação, a erosão em (x, y) de uma imagem f por um elemento estruturante b é dado por:

$$[f \ominus b](x, y) = \min_{(s,t) \in b} \{f(x + s, y + t)\}$$

onde, x e y são incrementados através de s e t, o qual permite que a partir da origem de b se visite cada pixel em f. Por exemplo, se b é um elemento estruturante quadrado de tamanho 3 x 3, para obter a erosão no ponto (x, y), é necessário achar o valor mínimo dos noves valores de f contidos na região 3 x 3 definida por b quando sua origem é o ponto (x, y) (RAFAEL C. GONZALEZ, 2007). Em outras palavras, a operação de erosão troca o valor de profundidade do *pixel* analisado pelo menor valor de profundidade da sua vizinhança.

Para acessar e modificar o *depth buffer* foi necessário utilizar a arquitetura *Frame Buffer Object* do *OpenGL*, o qual torna possível fazer renderizações para texturas (*off-screen rendering*). Os parâmetros de *GL_TEXTURE_WRAP_S* e *GL_TEXTURE_WRAP_T* na leitura da textura utilizada para renderizar o *depth buffer* foram ajustados para *GL_CLAMP*. Isto faz com que, quando o operador morfológico de erosão 3 x 3 acessa valores fora dos limites da textura, sempre resulte os valores borda da textura.

A aplicação da operação morfológica de erosão de tamanho 3 x 3 mostrou-se eficiente em termos de performance, mas os resultados obtidos ainda não eram bons, pois, quando aplicado em todo o *depth buffer*, o operador morfológico de erosão 3 x 3 reduz toda a

imagem projetada no *depth buffer*. Este efeito de redução da imagem pode ser visto na Figura 5.7.

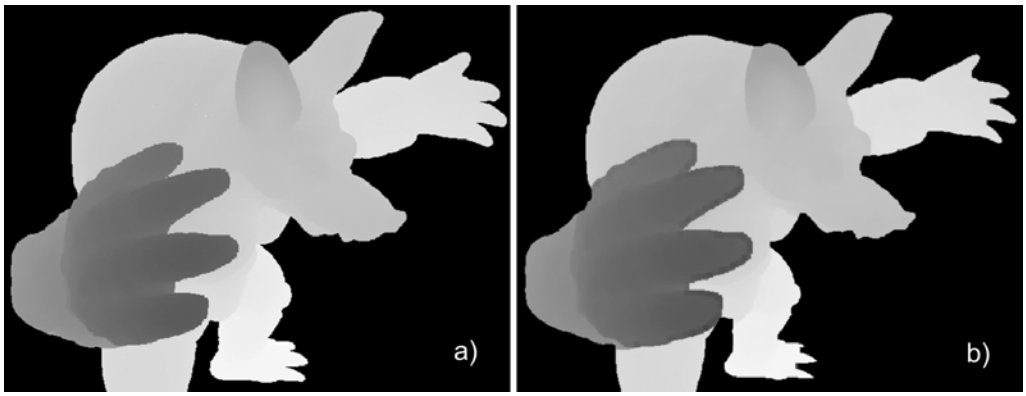
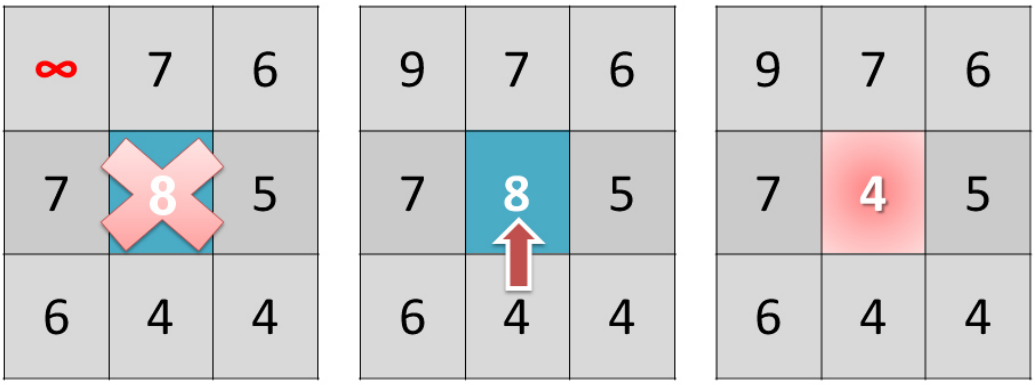


Figura 5.7: Problemas no *depth buffer* com a aplicação de um operador morfológico de erosão. Em a) é apresentado o *depth buffer* original. Em b) o *depth buffer* é modificado pelo operador morfológico de erosão.

Para atenuar o efeito de redução de toda a imagem do *depth buffer*, foi modificada a operação morfológica de erosão 3 x 3 de modo que fosse aplicada somente onde os valores de profundidade do *depth buffer* são diferentes do valor do fundo (segundo plano). No caso desta implementação, o valor do segundo plano do *depth buffer* foi ajustado para 1.0, logo, quando o operador morfológico acha um valor de vizinhança igual a 1.0, ele não altera o *pixel* analisado. Este processo poder ser verificado na Figura 5.8. Esse modificação fez com que o operador morfológico de erosão fosse aplicado somente as partes que estão próximas das bordas internas do modelo.



(a) (b) (c)

Figura 5.8: Em a) a troca de *pixels* não ocorre, pois é verificado um pixel vizinho com valor do segundo plano. Em b) e c) é mostrado um exemplo onde a troca ocorre, pois a vizinhança do pixel analisado contém um valor de profundidade menor.

O ajuste do *depth buffer* pelo operador morfológico de erosão 3 x 3 é apresentado na Figura 5.9, onde, na primeira coluna, podem ser vistos as variações do *depth buffer*, e, na segunda coluna, podem ser vistas as variações do resultado do operador utilizando o algoritmo de *Z-Buffering* para o modelo *Armadillo*.

Na Figura 5.9 a) é apresentado o *depth buffer* criado utilizando um raio fixo para os impostores. Já em b) é apresentado o *depth buffer* gerado após uma aplicação da operação

morfológica de erosão 3 x 3 modificada. Nota-se uma redução sutil nas bordas internas do *depth buffer*, apontadas pelas fechas vermelhas. Em c) é apresentada a diferença perceptual entre as imagens do *depth buffer* a) e b). Com esta imagem é possível notar que o método trabalha modificando somente as bordas internas.

Na Figura 5.9 d) é mostrado o resultado da remoção de pontos ocultos quando utilizado o *depth buffer* apresentado em a). Já na imagem e) são apresentados os pontos visíveis obtidos quando utilizado o *depth buffer* modificado pelo operador morfológico de erosão 3 x 3, apresentado na imagem b). Nota-se uma aumento sutil na quantidade de pontos visíveis nas bordas internas, salientadas pelas fechas vermelhas. Em f) é apresentada a diferença perceptual entre as imagens geradas a partir do *depth buffer* em a) e do *depth buffer* em b). Nota-se que este processo consegue simular a redução do raio dos impostores no espaço de imagem.

5.1.3 Removendo os Pontos Ocultos

Após a criação do *depth buffer* ajustado pelo operador morfológico de erosão 3 x 3, é necessário renderizar os pontos utilizando-o para remover os pontos ocultos.

Cada ponto é projetado com o tamanho máximo de um *pixel*, portanto, suas coordenadas espaciais são armazenadas em no máximo um *pixel* da imagem final. Para lidar com múltiplas projeções em um mesmo *pixel* da imagem final, o teste de oclusão *GL_DEPTH_TEST* é habilitado para renderizar os impostores, mantendo assim somente o *pixel* mais à frente.

Para reduzir o efeito de *Z-Fighting*, já mencionado na seção anterior, os valores do *depth buffer* são modificados de forma que eles sejam deslocado para um valor de profundidade maior. Para isto, nesta etapa do operador um *shader* de fragmento calcula os novos valores do *depth buffer*. A formula utilizada para este cálculo é apresentada a seguir:

$$gl\_FragDepth = (depth) + (1.0 - depth) * epsilon$$

$Depth$ é o valor lido do *depth buffer* armazenado em textura resultante do operador morfológico de erosão. O valor $epsilon$ é o tamanho do deslocamento positivo aplicado no *depth buffer*. Nesta implementação, o valor 0.05 para $epsilon$ mostrou-se adequado, de modo que o fenômeno de *Z-Fighting* não pode ser percebido. Para remover pontos que não estão visíveis, é necessário utilizar, no mesmo *shader* de fragmento, a função $discard$ da linguagem *GLSL*. A sua utilização causa o término do *shader* de fragmento para o fragmento atual (neste caso para o ponto atual da nuvem de pontos) sem escrevê-lo no *buffer* de renderização. Isto faz com que o ponto seja eliminado do resultado final do operador, transformando-o em ponto oculto.

O uso de $discard$ é determinado pelo teste lógico

$$gl\_FragCoord.z > gl\_FragDepth$$

O valor $gl\_FragDepth$ é o valor de profundidade computado a partir do *depth buffer*, e o valor $gl\_FragCoord.z$ é o valor de profundidade do fragmento atual, ou, neste caso, o valor de profundidade do ponto atual. Em outras palavras, se o valor de profundidade do ponto atual é maior que o valor de profundidade da superfície visível reconstruída no *depth buffer*, este ponto não é visível.

5.1.4 Obtendo os Pontos Visíveis

Neste último passo do operador, os pontos visíveis são extraídos da *GPU* através da desprojeção dos pontos rasterizados. Após a remoção dos pontos ocultados pela superfí-

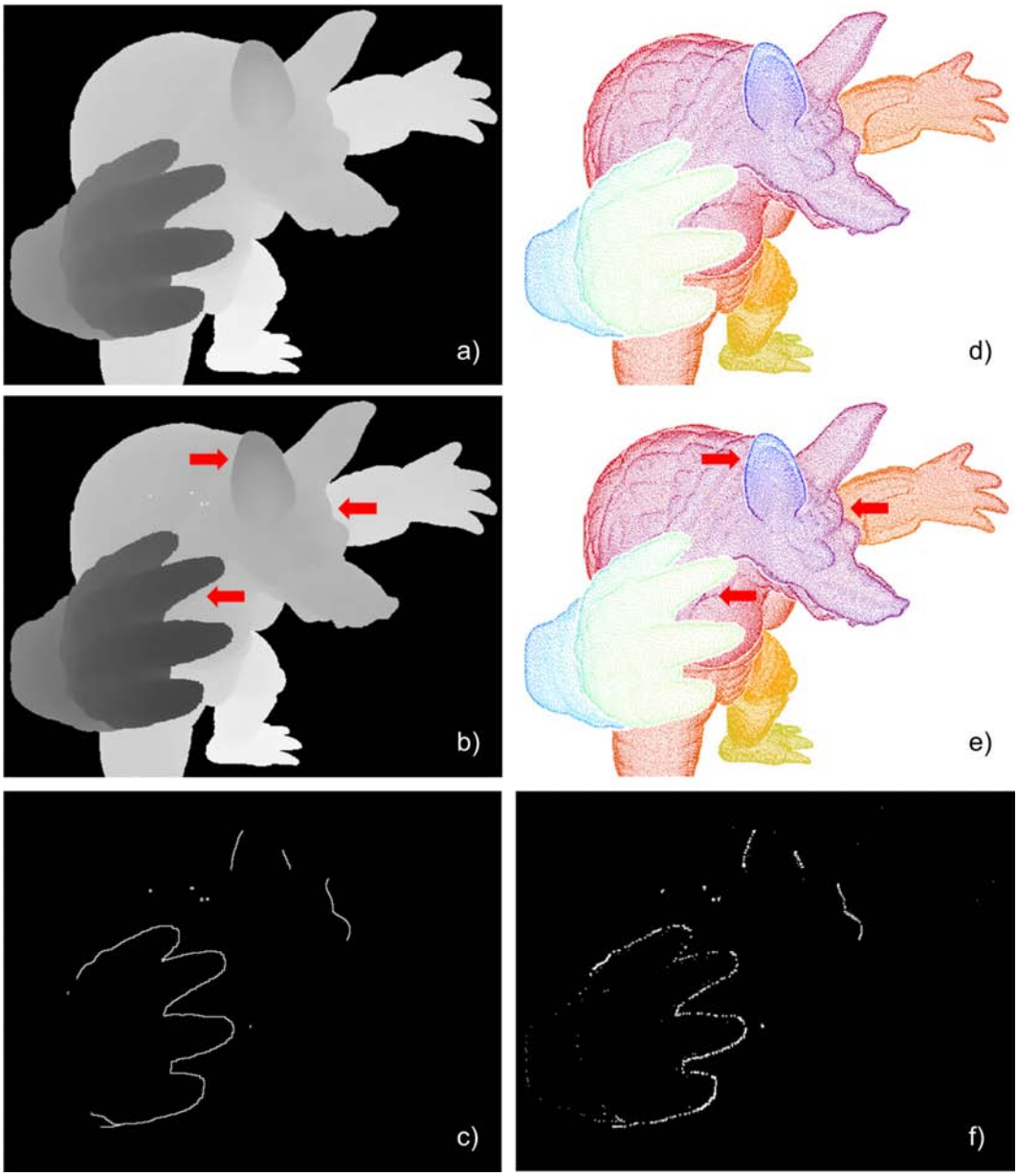


Figura 5.9: Ajuste do *depth buffer* pelo operador morfológico de erosão 3 x 3. Em a) é apresentado o *depth buffer* original, em b) o *depth buffer* modificado, em c) a diferença perceptual entre a) e b). A imagem final gerada com o *depth buffer* original é apresentada em d), em e) a imagem final gerada com o *depth buffer* modificado, em f) é a diferença perceptual as imagens d) e e).

cie reconstruída no *depth buffer*, é necessário obter os pontos visíveis para que o método de remoção de pontos ocultos proposto neste trabalho seja utilizado como um operador. Para isso foi utilizada novamente a arquitetura *Frame Buffer Object*, a qual possibilita renderizar os pontos em texturas com precisão de 32 *bits*. Para que cada fragmento rasterizado na textura seja transformado novamente em um ponto em *3D*, é necessário utilizar uma função que inverta o processo de projeção. Esta função é apresentada em detalhes no Algoritmo 2.

O método de desprojeção é dependente do tamanho do *viewport* e da matriz de projeção inversa e é executado na *GPU* e seu resultado armazenado em textura. No caso desta

Algorithm 2 *Função de desprojeção*

Require: $Integer\ ViewportWidth$ {Largura do *Viewport*}
Require: $Integer\ ViewportHeight$ {Altura do *Viewport*}
Require: $Point\ FragCoord$ {Coordenada do Fragmento}
Require: $Matrix\ ProjectionMatrixInverse$ {Matriz Inversa da Projeção}

{**Passo 1**}
$Real\ width\_half \leftarrow \lfloor \frac{ViewportWidth}{2.0} \rfloor$
$Real\ height\_half \leftarrow \lfloor \frac{ViewportHeight}{2.0} \rfloor$
{**Passo 2**}
$Point\ ndc\_coord$
$ndc\_coord.x \leftarrow (FragCoord.x * (\frac{1.0}{width\_half}) - 1.0) * (\frac{1.0}{FragCoord.w})$
$ndc\_coord.y \leftarrow (FragCoord.y * (\frac{1.0}{width\_half}) - 1.0) * (\frac{1.0}{FragCoord.w})$
$ndc\_coord.z \leftarrow (FragCoord.z) * (\frac{1.0}{FragCoord.w})$
$ndc\_coord.w \leftarrow \frac{1.0}{FragCoord.w}$
{**Passo 3**}
$Point\ out \leftarrow ProjectionMatrixInverse * ndc\_coord$
return out

implementação, o parâmetro de entrada *FragCoord* é a variável *gl_FragCoord*, disponível nos *shaders* de fragmento do *GLSL*; *ProjectionMatrixInverse* é a variável *uniform* do *GLSL gl_ModelViewProjectionMatrixInverse*, a qual permite acessar o estado da matriz de projeção inversa do *OpenGL*.

Após a aplicação da função de desprojeção nos fragmentos, é necessário transferir os pontos visíveis da *GPU* armazenados em textura, para a *CPU*. Para isso é utilizado a função *glGetTexImage*, a qual copia para a *CPU* a textura habilitada pela função *glBindTexture* na máquina de estado do *OpenGL*. Com a textura copiada para a *CPU*, é necessário extrair os pontos visíveis. Para isso é utilizado um laço de repetição onde é testado se o *pixel* lido da textura é igual à cor de fundo ajustada anteriormente. Caso não seja igual à cor de fundo, o *pixel* é um fragmento desprojetado, logo pode ser adicionado a uma lista de pontos visíveis. Este passo do operador mostrou ser o gargalo do método, pois é bastante cara a transferência de grande quantidade de informação da *GPU* para a *CPU*. Entretanto, esse procedimento não comprometeu de forma significativa a performance do operador.

O resultado final do operador proposto neste capítulo, após os quatro passos necessários para remover os pontos ocultos da nuvem de pontos, é a lista criada com os pontos visíveis. Esta lista agora pode ser utilizada como entrada para outros métodos e aplicações. Alguns resultados obtidos com este operador na visualização direta de nuvens de pontos são apresentados na próxima sessão.

5.1.5 Resultados

A aplicação mais óbvia do operador apresentado neste trabalho é a visualização direta de nuvens de pontos de forma eficiente e sem a utilização de informação de topologia. Como já foi apresentado, o resultado do operador proposto - os pontos visíveis - é dependente da escolha do parâmetro R, que define o raio dos impostores. No caso da escolha de um raio muito pequeno, o resultado esperado para o operador é afetado por uma grande quantidade de pontos que não deviam estar visíveis, ou seja, uma grande quantidade de falsos positivos. Este fenômeno pode ser observado na Figura 5.10 b). Já com a escolha

de uma valor para R muito alto, o resultado esperado do operador é prejudicado por uma grande quantidade de falsos negativos, ou seja, pontos que não são indentificados como visíveis mas deveriam. Isso cria o fenômeno de silhuetas na imagem final da visualização direta de nuvens de pontos. Esse processo pode ser observado na Figura 5.10 d).

É possível ajustar o valor de R de forma que ele seja um raio ótimo para uma determinada posição de câmera. Para isto é observado que R deve produzir a menor quantidade de falsos negativos e falsos positivos no resultado do operador, ou seja, o erro do operador deve ser minimizado. A visualização direta de uma nuvem de pontos utilizando um valor ótimo para R pode ser vista em 5.10 c). A Figura 5.11 mostra que o valor de R igual a 0.023 minimiza o erro do operador para a nuvem de pontos *Armadillo* na configuração de câmera apresentada na Figura 5.10, logo minimiza a quantidade de falsos negativos e pontos falso positivos.

Quando as nuvens de pontos são bem amostradas é possível utilizar uma heurística simples para determinar um raio R próximo do raio ótimo, simplesmente escolhendo um ponto aleatoriamente dentro da nuvem de pontos e procurando a distância do seu vizinho mais próximo. Este procedimento tem complexidade $O(n)$ e é de simples implementação.

As Figuras C.1, C.2, C.3 e C.4 no anexo C apresentam alguns resultados para a visualização direta para diferentes nuvens de pontos utilizando um valor de R ótimo.

As tabelas C.1 e C.2 do anexo C apresentam os resultados de performance e de precisão obtidos com o operador remoção de pontos ocultos acelerado por *GPU*. As tabelas primeiramente descrevem os parâmetros de configuração da câmera utilizados no teste para os diferentes *Datasets*. Após, são apresentadas estatísticas de precisão e performance para o método utilizado como operador, tanto para o que utiliza o *depth buffer* original como para o que utiliza o *depth buffer* modificado pelo operador morfológico de erosão 3 x 3.

A propriedade de gerar resultados o mais próximos possíveis do correto para o operador de remoção de pontos ocultos acelerado por *hardware* pode ser verificada pelo baixo erro introduzido no resultado. Quanto maior o tamanho do *viewport*, menor é o erro introduzido no resultado, pois são gerados menos projeções de um ponto no mesmo fragmento. O menor erro para o operador utilizando o *depth buffer* original foi verificado quando o *viewport* utilizado foi de 2048 x 1536 no *Dataset Bunny*, gerando um erro de 9.83%.

Outra propriedade verificada foi que, quando introduzida a operação morfológica de erosão 3 x 3 no operador, o efeito de silhuetas no resultado foi reduzido. O melhor resultado para o operador, utilizando o *depth buffer* modificado pelo operador morfológico de erosão, foi verificado quando o *viewport* utilizado foi de 1024 x 768 no *Dataset Bunny*, gerando um erro de 9.02%.

Todas as comparações de precisão foram calculadas utilizando o conjunto correto de pontos visíveis para cada posição de câmera específica. O conjunto de pontos corretos foi computado utilizando um algoritmo geométrico, o qual remove os pontos ocultos calculando as interseções dos segmentos de reta gerados pela câmera e os pontos da nuvem e a malha de triângulos existente no modelo original.

A performance do operador foi medida utilizando a quantidade de pontos visíveis transferidos da *GPU* para a *CPU* por segundo, mas considerando o tempo de todo o processo que define o operador, não somente da transferência de dados. No melhor caso, o operador conseguiu tratar 1568220 pontos por segundo para o *Dataset Bunny* com a utilização do *depth buffer* original em um *viewport* de tamanho 640 x 480. Para o operador com o *depth buffer* modificado com o operador morfológico de erosão, foi conseguido 1569690 pontos por segundo para o *Dataset Bunny* em um *viewport* de tamanho 640 x 480.

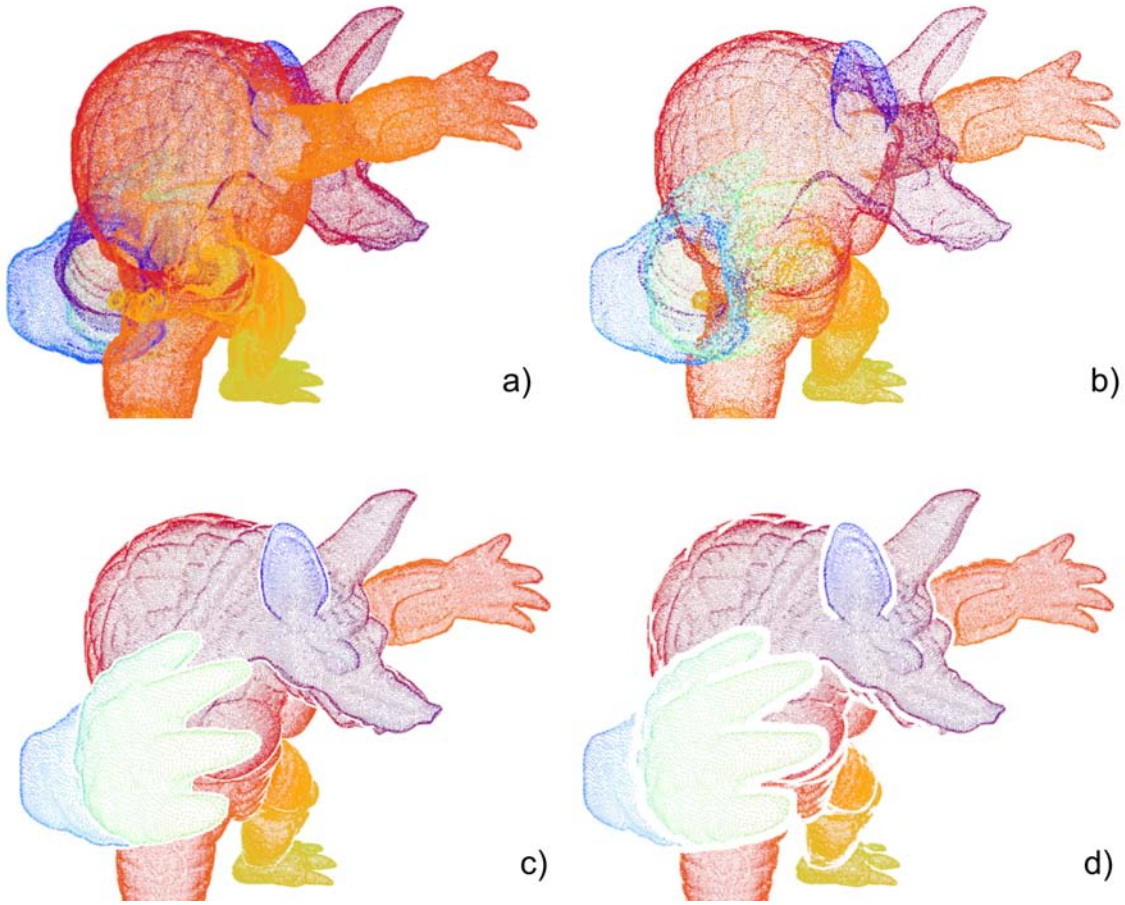
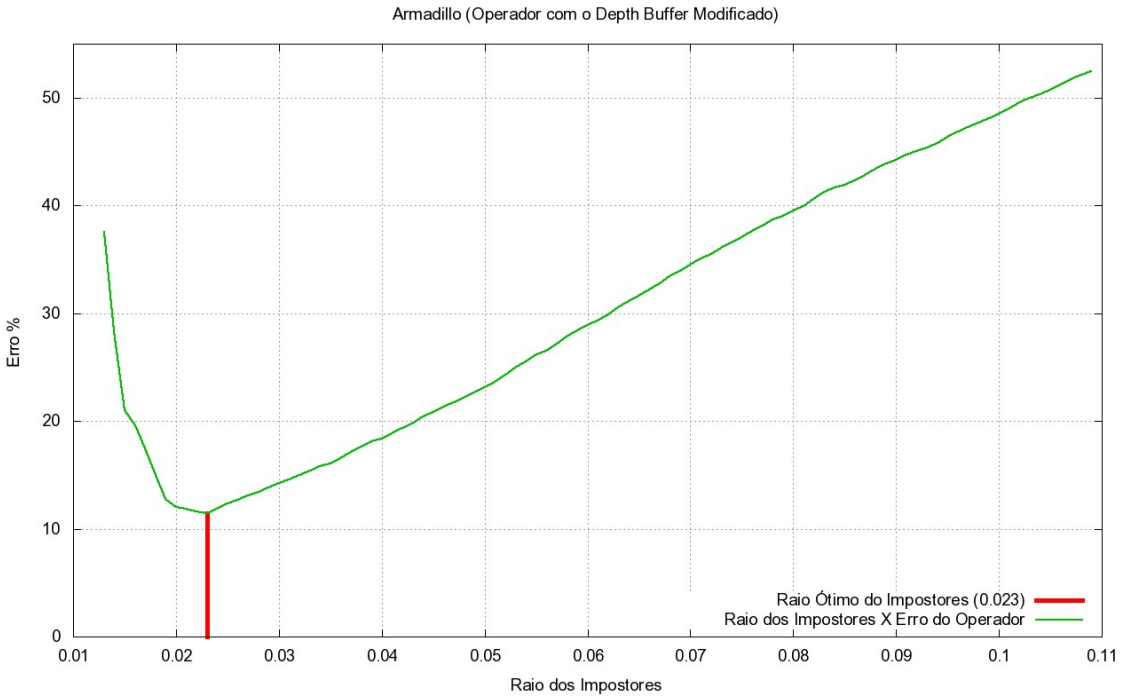


Figura 5.10: Resultado do operador para o modelo *Armadillo* com diferentes valores de R. Em a) a nuvem de pontos original, em b) o resultado do operador com um R menor que o valor ótimo, em c) o resultado do operador com um R ótimo (0.023) e em d) o resultado do operador para um R maior que o valor ótimo.

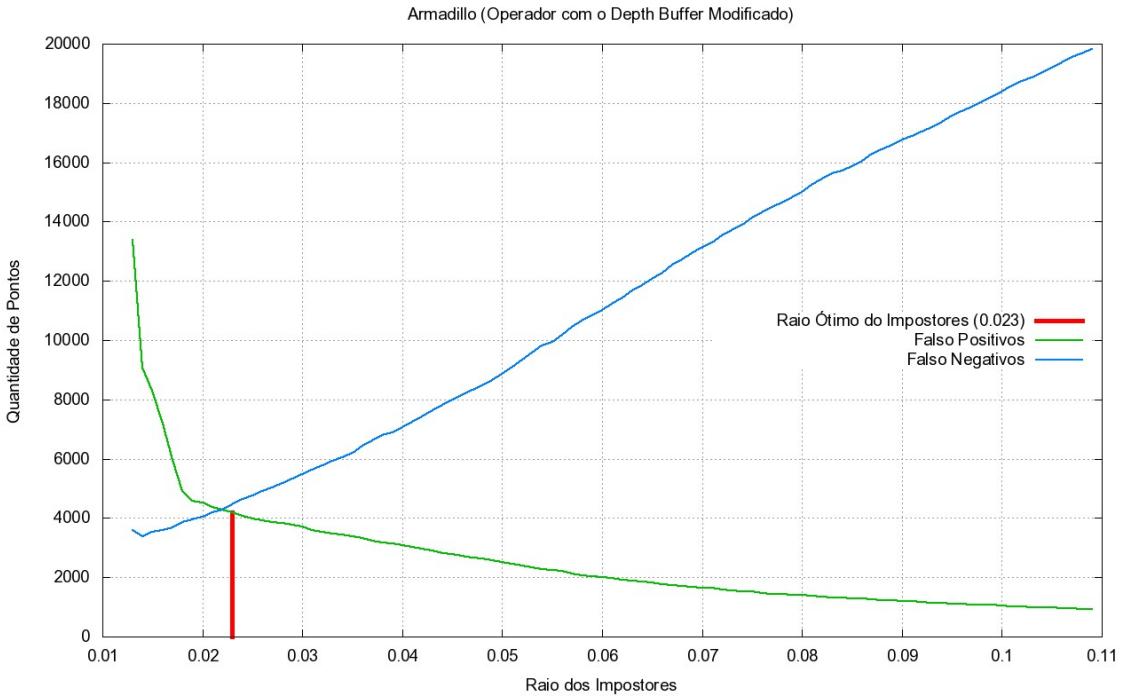
O operador utilizando o *depth buffer* modificado pelo operador morfológico de erosão conseguiu melhor performance, porque removeu mais falsos negativos, logo transferiu mais pontos da *GPU* para a *CPU*, aumentando o valor da métrica de eficiência.

As tabelas C.1 e C.2 do anexo C apresentam estatísticas de precisão e performance para os métodos utilizados como visualizador, sem a transferência dos pontos da *GPU* para a *CPU*, tanto para o que utiliza o *depth buffer* original como o que utiliza o *depth buffer* modificado pelo operador morfológico de erosão. Outra propriedade verificada do operador foi a possibilidade de utilizá-lo em aplicações em tempo real. As estatísticas de performance para o método na visualização direta de nuvem de pontos mostram que este é bastante eficiente, pois, no melhor caso, o operador remoção de pontos ocultos acelerado por *GPU* conseguiu processar 1128.87 quadros em um segundo, para o *Dataset Bunny* com a utilização do *depth buffer* original em um *viewport* de tamanho 640 x 480. Para o operador com o *depth buffer* modificado com a operação morfológica de erosão foram conseguidos 417.17 quadros por segundo para a mesma configuração apresentada anteriormente.

Todos resultados de performance e imagens foram gerados em uma máquina com processador *QuadCore Intel Core 2 Quad Q6600*, *2400 MHz*, *4.0 GB RAM* de memória e uma placa gráfica *NVIDIA GeForce 8600 GT* (*256 MB*).



(a) Gráfico do Erro do operador em relção ao tamanho do impostor.



(b) Gráfico dos falsos positivos/negativos em relação ao tamanho do impostor.

Figura 5.11: O gráfico a) mostra o erro do operador em relação ao raio dos impostores para o modelo *Armadillo*. O Gráfico b) mostra os falsos positivos/negativos para o mesmo modelo. O raio ótimo computado é apresentado em vermelho.

Para mais detalhes sobre as configurações utilizadas e sobre as estatísticas de performance e precisão, deve-se consultar as tabelas inseridas no anexo C. Com a análise destas estatísticas, pode-se verificar que as propriedades sugeridas para o operador foram

alcançadas.

5.2 Sumário

Este capítulo apresenta um método aproximativo eficiente baseado em *GPU* para remoção de pontos ocultos para nuvens de pontos sem informação de conectividade. O operador determina a visibilidade das nuvens de pontos reconstruindo uma aproximação da superfície visível da nuvem de pontos, dentro do *depth buffer*, utilizando impostores. O algoritmo de *Z-Buffer* modificado por uma operação morfológica de erosão para reduzir o efeito de silhuetas na imagem final, é utilizado para remover os pontos ocultos. O método mostra-se flexível como operador, tanto para a obtenção dos pontos visíveis, quanto na visualização direta de nuvens de pontos.

6 CONCLUSÃO E TRABALHOS FUTUROS

As abordagens baseadas em *cluster* do operador *HPR* apresentadas neste trabalho no capítulo 3 mostraram-se otimizações eficientes para visualização direta de nuvens de pontos. Os operadores *HPR* baseado em *Grade Regular* e em *Octree* proporcionaram uma melhora na performance, comparado com o método original, a um custo de produzir informação de visibilidade aproximada.

Foram medidas as diferenças perceptuais entre a visibilidade aproximada que os operadores *HPR* baseados em *cluster* produziram comparada com o resultado gerado pelo operador original. Os resultados obtidos tiveram um erro perceptual aceitável, o que justifica o método como uma otimização do operador original.

Em visualizações interativas de conjuntos de pontos, onde abordagens que utilizam *LOD* são frequentemente utilizadas, o operador baseado em *cluster* se justifica, pois ele consegue manter o equilíbrio entre a velocidade e a qualidade do método.

A abordagem baseada em *Octree* demonstrou ser mais rápida e visualmente melhor, pois apresentou menor erro perceptual do que a abordagem baseada em *Grade Regular* para animações baseadas em pontos com grande quantidade de dados. Para animações com menores quantidades de dados, o processamento introduzido para criar a *Octree* torna o operador mais lento do que o baseado em *Grade Regular*.

Como qualquer método baseado em particionamento espacial, busca-se um equilíbrio entre velocidade e precisão. A qualidade da imagem final renderizada é diretamente proporcional à performance. Se a subdivisão espacial aumenta, a qualidade da imagem final renderizada aumenta também, mas isto traz uma diminuição no desempenho da técnica.

Alguns possíveis trabalhos futuros dentro desta técnica são acelerar o cálculo da construção da envoltória convexa utilizando uma solução *multi-thread* e utilizar *CPUs multi-core*, ou mesmo uma implementação em *GPU*. Também revisitar os métodos de subdivisão espacial considerado neste trabalho e propor o uso de outros tipos de estruturas de dados para hierarquia espacial. Além disso, é possível introduzir estruturas de dados cinéticas (ABAM; BERG, 2007) ou auto-ajustáveis (ACAR, 2005), as quais não requerem uma completa restruturação em cenas altamente dinâmicas, possibilitando uma adaptação da envoltória convexa, não uma construção completa do mesmo em cada quadro da animação. Outro fonte de trabalho para o futuro é a utilização de algoritmos mais robustos para o cálculo da envoltória convexa *3D* no operador baseado em *cluster*, pois os algoritmos tais como o *Quickhull* (BARBER; DOBKIN; HUHDANPAA, 1996) utilizado aqui possuem muitos problemas numéricos.

O método de renderização baseada em pontos apresentado neste trabalho no capítulo 4 é uma alternativa aos atuais algoritmos de renderização por *Surface Splatting*. Com alguns pequenos ajustes o método *Surface Splatting* baseado em *Geometry Shader* pode-se tornar uma sistema de renderização mais genérico. Uma lista de possíveis trabalhos

futuros para o método de *Surface Splatting* baseado em *Geometry Shader* é mostrada a seguir:

- Explorar o uso de *Deferred Shading* para *Surfel Lighting*;
- Explorar clusterização espacial para superfícies densamente amostradas;
- Explorar modelos com multi-resolução de *surfels*;
- Explorar conjunto de *surfels* dinâmicos com sistema de animação;

No capítulo 5 é proposto um operador que resolve a visibilidade de nuvens de pontos em tempo real utilizando aceleração por *hardware*. O operador apresentado mostrou-se um método aproximativo bastante eficiente para remoção de pontos ocultos pois utiliza técnicas de *Surface Splatting* na *GPU* para produzir eficientes aproximações comparado com a informação de visibilidade correta.

O método produziu bons resultados, mesmo sem estimar os vetores normais da superfície implícita ou computar informação de conectividade da nuvem de pontos. O uso do algoritmo de *Z-Buffering* implementado em *hardware* proporcionou a base para resolver o problema de visibilidade de forma eficiente. A utilização do operador morfológico de erosão modificado produziu resultados melhores, pois a sua aplicação no *depth buffer* reduz a quantidade de falsos negativos, ou seja, reduz o efeito de silhuetas no resultado final. Porém a utilização da filtragem no *depth buffer* traz uma perda na performance no algoritmo devido a necessidade da implementação de um passo a mais, embora ainda justifique-se o seu uso.

A utilização da técnica na visualização direta de nuvem de pontos sem informação de conectividade produziu resultados bastante satisfatórios, pois permitiu a investigação em tempo real de nuvens de pontos largamente amostradas. Uma lista de possíveis extensões no operador *HPR* baseado em *Splatting* e apresentado a seguir:

- Explorar a geração do parâmetro R ótimo de forma automática e genérica;
- Explorar a utilização de operadores morfológicos de vizinhança adaptativos (DEBAYLE; PINOLI, 2005) para tentar simular a adaptabilidade no tamanho;
- Explorar a utilização do operador em *datasets* dinâmicos;
- Explorar a utilização do operador em outras aplicações diferentes de visualização direta de nuvem de pontos;
- Explorar a técnica de *Depth Peeling* utilizando múltiplas aplicações do operador para renderizar objetos com transparência.

REFERÊNCIAS

ABAM, M. A.; BERG, M. de. Kinetic sorting and kinetic convex hulls. **Comput. Geom. Theory Appl.**, Amsterdam, The Netherlands, The Netherlands, v.37, n.1, p.16–26, 2007.

ACAR, U. A. A library for self-adjusting computation. In: IN ACM SIGPLAN WORKSHOP ON ML, 2005. **...** [S.l.: s.n.], 2005.

ADAMS, B. **Point-Based Modeling, Animation and Rendering of Dynamic Objects**. 2006. Tese (Doutorado em Ciência da Computação) — Katholieke Universiteit Leuven.

ADAMS, B. **Surfel Viewer**. 2007.

ALEXA, M. et al. Computing and Rendering Point Set Surfaces. **IEEE Transactions on Visualization and Computer Graphics**, Piscataway, NJ, USA, v.9, n.1, p.3–15, 2003.

ALEXA, M. et al. Point-based computer graphics. In: SIGGRAPH 04: ACM SIGGRAPH 2004 COURSE NOTES, 2004, New York, NY, USA. **...** ACM, 2004. p.7.

APPEL, A. Some Techniques for Shading Machine Renderings of Solids. In: AFIPS 1968 SPRING JOINT COMPUTER CONFERENCE PROCEEDINGS, 1968. **...** [S.l.: s.n.], 1968. v.32, p.37–45.

BARBARA SOLENTHALER, Y. Z.; PAJAROLA., R. **Ef cient Re nement of Dynamic Point Data**. 2007.

BARBER, C. B.; DOBKIN, D. P.; HUHDANPAA, H. The quickhull algorithm for convex hulls. **ACM Trans. Math. Softw.**, New York, NY, USA, v.22, n.4, p.469–483, 1996.

BITTNER, J.; WONKA, P. Visibility in Computer Graphics. **Environment and Planning B**: planning and design, [S.l.], v.30, n.5, p.729–756, Sept. 2003.

BLYTHE, D. The Direct3D 10 system. In: SIGGRAPH 06: ACM SIGGRAPH 2006 PAPERS, 2006, New York, NY, USA. **...** ACM, 2006. p.724–734.

BOARD, C. E. **CGAL User and Reference Manual**. 3.3 ed. [S.l.: s.n.], 2007.

BOTSCH, M. et al. High-quality surface splatting on today s GPUs. **Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics**, Los Alamitos, CA, USA, v.0, p.17–141, 2005.

CARSTEN DACHSBACHER, C. V.; STAMMINGER, M. Sequential point trees. **ACM Trans. Graph.**, New York, NY, USA, v.22, n.3, p.657–662, 2003.

CATMULL, E. E. **A Subdivision Algorithm for Computer Display of Curved Surfaces**. 1974. Tese (Doutorado em Ciência da Computação) — Department of Compuer Science, University of Utah.

CMU, L. R. et al. **Object Space EWA Surface Splatting**: a hardware accelerated approach to high quality point rendering. 2002.

COHEN-OR, D.; CHRYSANTHOU, Y. **Abstract A Survey of Visibility for Walkthrough Applications**. 2003.

DEBAYLE, J.; PINOLI, J. C. Spatially Adaptive Morphological Image Filtering using Intrinsic Structuring Elements. **Image Analysis and Stereology**, [S.l.], v.24, n.3, p.145–158, November 2005.

DRETTAKIS, M. S. G. Interactive sampling and rendering for complex and procedural geometry. In: 2001. **...** Springer-Verlag, 2001. p.151–162.

ETH ZURICH, C. G. L. at. **Pointshop3D**. 2007.

FUCHS, H.; KEDEM, Z. M.; NAYLOR, B. F. On visible surface generation by a priori tree structures. In: SIGGRAPH 80: PROCEEDINGS OF THE 7TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1980, New York, NY, USA. **...** ACM, 1980. p.124–133.

GHOSH, S. **Visibility Algorithms in the Plane**. [S.l.]: Cambridge University Press, 2007.

GOTTSCHALK, S.; LIN, M. C.; MANOCHA, D. OBBTree: a hierarchical structure for rapid interference detection. In: SIGGRAPH 96: PROCEEDINGS OF THE 23RD ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1996, New York, NY, USA. **...** ACM, 1996. p.171–180.

GREENE, N.; KASS, M.; MILLER, G. Hierarchical Z-buffer visibility. In: SIGGRAPH 93: PROCEEDINGS OF THE 20TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1993, New York, NY, USA. **...** ACM, 1993. p.231–238.

GROSS, M.; PFISTER, H. **Point-Based Graphics**. [S.l.]: Morgan Kaufmann, 2007.

GROSSMAN, J. **Point Sample Rendering**. 1998. Dissertação (Mestrado em Ciência da Computação) — Department of Electrical Engineering and Computer Science, MIT.

HERT, S.; SCHIRRA, S. 3D Convex Hulls. In: BOARD, C. E. (Ed.). **CGAL User and Reference Manual**. 3.3 ed. [S.l.: s.n.], 2007.

HERT, S.; SEEL, M. dD Convex Hulls and Delaunay Triangulations. In: BOARD, C. E. (Ed.). **CGAL User and Reference Manual**. 3.3 ed. [S.l.: s.n.], 2007.

HOPPE, H. et al. Surface reconstruction from unorganized points. **Computer Graphics**, [S.l.], v.26, n.2, p.71–78, 1992.

INC., A. T. **AGEIA, The PhysX processor.** http://www.ageia.com. [Online; accessed April-2008].

JACOBSSON, D. **Hardware Accelerated Point-based Rendering of Granular Matter for Interactive Applications**. 2007.

KATZ, S.; LEIFMAN, G.; TAL, A. Mesh segmentation using feature point and core extraction. **The Visual Computer**, [S.l.], v.21, n.8-10, p.649–658, 2005.

KATZ, S.; TAL, A.; BASRI, R. Direct visibility of point sets. In: SIGGRAPH 07: ACM SIGGRAPH 2007 PAPERS, 2007, New York, NY, USA. **...** ACM, 2007. p.24.

KOBBELT, L.; BOTSCH, M. A survey of point-based techniques in computer graphics. **Computers & Graphics**, [S.l.], v.28, n.6, p.801–814, December 2004.

LEVOY, M. et al. The Digital Michelangelo Project: 3D scanning of large statues. In: ACM SIGGRAPH 2000, 2000. **Proceedings...** [S.l.: s.n.], 2000. p.131–144.

LEYVAND, T.; SORKINE, O.; COHEN-OR, D. Ray space factorization for from-region visibility. **ACM Transactions on Graphics (TOG)**, [S.l.], v.22, n.3, p.595–604, 2003.

LUQUE, R.; COMBA, J. L. D.; FREITAS, C. M. D. **Broad-Phase Colision Detection Using Semi-Adjusting BSP-Trees**. 2005. 179-186p. (1595930132).

MARC LEVOY, T. W. **The use of points as display primitives**. 1985.

MIGUEL SAINZ, R. P.; LARIO, R. Points Reloaded: point-based rendering revisited. **Proceedings Symposium on Point-Based Graphics**, [S.l.], 2004.

PAJAROLA, R.; SAINZ, M.; GUIDOTTI, P. Object-space point blending and splatting. In: SIGGRAPH 03: ACM SIGGRAPH 2003 SKETCHES & APPLICATIONS, 2003, New York, NY, USA. **...** ACM, 2003. p.1–1.

PFISTER, H. et al. Surfels: surface elements as rendering primitives. In: SIGGRAPH 00: PROCEEDINGS OF THE 27TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2000, New York, NY, USA. **...** ACM Press/Addison-Wesley Publishing Co., 2000. p.335–342.

PHOTOGRAMMETRIE, I. für. **Laser Splatting - Universität Stuttgart**. 2007.

RAFAEL C. GONZALEZ, R. E. W. **Digital Image Processing**. [S.l.]: Prentice Hall, 2007.

SAINZ, M.; PAJAROLA, R. Point-based rendering techniques. **Computers & Graphics**, [S.l.], v.28, n.6, p.869–879, December 2004.

SOLENTHALER, B.; ZHANG, Y.; PAJAROLA, R. **Ef cient Re nement of Dynamic Point Data**. 2007.

STANFORD. **The Stanford 3D Scanning Repository.** http://graphics.stanford.edu/data/3Dscanrep/. [Online; accessed April-2008].

SUTHERLAND, I. E.; SPROULL, R. F.; SCHUMACKER, R. A. A Characterization of Ten Hidden-Surface Algorithms. **ACM Comput. Surv.**, New York, NY, USA, v.6, n.1, p.1–55, 1974.

SZYMON RUSINKIEWICZ, M. L. **QSplat**: a multiresolution point rendering system for large meshes. 2000.

UTILITY., P. I. D. **The Stanford 3D Scanning Repository.** http://pdiff.sourceforge.net/. [Online; accessed April-2008].

WALD, I.; SEIDEL, H.-P. Interactive ray tracing of point-based models. In: SIGGRAPH 05: ACM SIGGRAPH 2005 SKETCHES, 2005, New York, NY, USA. **...** ACM, 2005. p.54.

WANG, W.; YANG, J.; MUNTZ, R. **PK-tree**: a dynamic spatial index structure for large data sets. 1997.

WARNOCK, J. E. **A hidden surface algorithm for computer generated halftone pictures**. 1969. Tese (Doutorado em Ciência da Computação) — .

WHITTED, T. An improved illumination model for shaded display. In: SIGGRAPH 05: ACM SIGGRAPH 2005 COURSES, 2005, New York, NY, USA. **...** ACM, 2005.

ZWICKER, M. et al. Surface splatting. In: SIGGRAPH 01: PROCEEDINGS OF THE 28TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2001, New York, NY, USA. **...** ACM, 2001. p.371–378.

APÊNDICE A RESULTADOS DOS MÉTODOS OPERADOR *HPR* BASEADO EM *CLUSTERS*

As tabelas a seguir mostram os resultados de performance e de diferença perceptual obtidos pelos operador *HPR* baseados em *clusters*.

(a) Parâmetros de configuração para todos os métodos *HPR*.

Método	Ponto de Visão	$log(R)$	Tamanho dos clusters
HPR Original	$(-7, 7, 7)$	2.9	-
HPR Grade Regular	$(-7, 7, 7)$	2.9	37x37x37
HPR Octree	$(-7, 7, 7)$	2.9	17 Pontos/Nodo Máx.

(b) Estatísticas das animações baseadas em nuvem de pontos para todos os métodos *HPR*.

Método	Removidos	Visíveis	Visíveis %
HPR Original	3330002	1895098	36.27%
HPR Grade Regular	2988863	2236237	42.79%
HPR Octree	47817070	17514680	44.15%

(c) Diferença perceptual de imagem comparada com o operador *HPR* original.

Método	Diferença Total	Diferença Relativa
HPR Grade Regular	253318 *pixels*	14.14%
HPR Octree	233495 *pixels*	13.03%

(d) Estatísticas de performance para todos os métodos *HPR* em quadros por segundos.

Método	Tempo Total	SPQ	Ganho
HPR Original	159.391 s	1.06	-
HPR Grade Regular	28.468 s	0.19	5.60
HPR Octree	33.328 s	0.22	4.78

Tabela A.1: Animação *Bunny*.

(a) Parâmetros de configuração para todos os métodos *HPR*.

Método	Ponto de Visão	$log(R)$	Tamanho dos clusters
HPR Original	$(-7, 7, 7)$	2.9	-
HPR Grade Regular	$(-7, 7, 7)$	2.9	85x85x85
HPR Octree	$(-7, 7, 7)$	2.9	18 Pontos/Nodo Máx.

(b) Estatísticas das animações baseadas em nuvem de pontos para todos os métodos *HPR*.

Método	Removidos	Visíveis	Visíveis %
HPR Original	18181879	7764221	29.92%
HPR Grade Regular	17106703	8839397	34.06%
HPR Octree	16934273	9011827	34.73%

(c) Diferença perceptual de imagem comparada com o operador *HPR* original.

Método	Diferença Total	Diferença Relativa
HPR Grade Regular	619789 *pixels*	14.73%
HPR Octree	589745 *pixels*	14.02%

(d) Estatísticas de performance para todos os métodos *HPR* em quadros por segundos.

Método	Tempo Total	SPQ	Ganho
HPR Original	788.703 s	5.26	-
HPR Grade Regular	308.516 s	2.08	2.56
HPR Octree	191.078 s	1.28	4.13

Tabela A.2: Animação *Armadillo*.

(a) Parâmetros de configuração para todos os métodos *HPR*.

Método	Ponto de Visão	$log(R)$	Tamanho dos clusters
HPR Original	$(-7, 7, 7)$	2.9	-
HPR Grade Regular	$(-7, 7, 7)$	2.9	120x120x120
HPR Octree	$(-7, 7, 7)$	2.9	16 Pontos/Nodo Máx.

(b) Estatísticas das animações baseadas em nuvem de pontos para todos os métodos *HPR*.

Método	Removidos	Visíveis	Visíveis %
HPR Original	47817070	17514680	26.80%
HPR Grade Regular	44685475	20646275	31.60%
HPR Octree	44904016	20427734	31.26%

(c) Diferença perceptual de imagem comparada com o operador *HPR* original.

Método	Diferença Total	Diferença Relativa
HPR Grade Regular	1047687 *pixels*	16.25%
HPR Octree	956867 *pixels*	14.85%

(d) Estatísticas de performance para todos os métodos *HPR* em quadros por segundos.

Método	Tempo Total	SPQ	Ganho
HPR Original	2092.66 s	13.95	-
HPR Grade Regular	815.031 s	5.43	2.57
HPR Octree	503.39 s	3.35	4.16

Tabela A.3: Animação *Dragon*.

APÊNDICE B RESULTADOS DA VISUALIZAÇÃO DIRETA DE NUVENS DE PONTOS UTILIZANDO O MÉTODO DE *SURFACE SPLATTING*

As imagens a seguir mostram alguns resultados obtidos pelo método de renderização baseada em pontos *Surface Splatting* utilizando *Geometry Shader*.

Figura B.1: *Head* (40880 *splats*, 248 *FPS*).

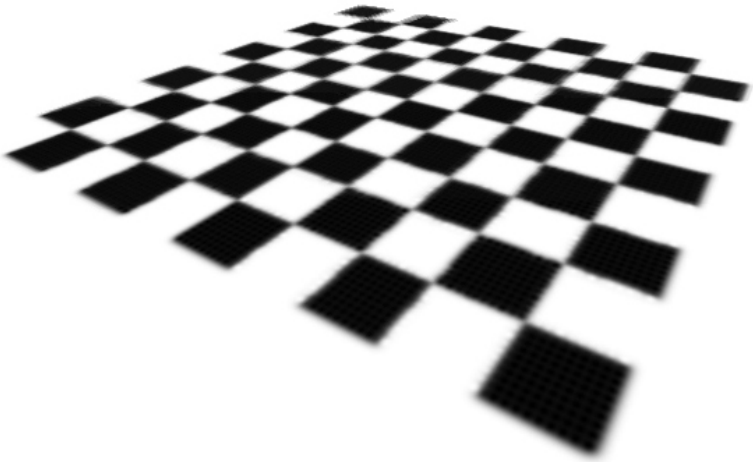


Figura B.2: *Checker* (10000 *splats*, 779 *FPS*).

Figura B.3: *Manhead* (87183 *splats*, 132 *FPS*) e *Igea* (134345 *splats*, 89 *FPS*).

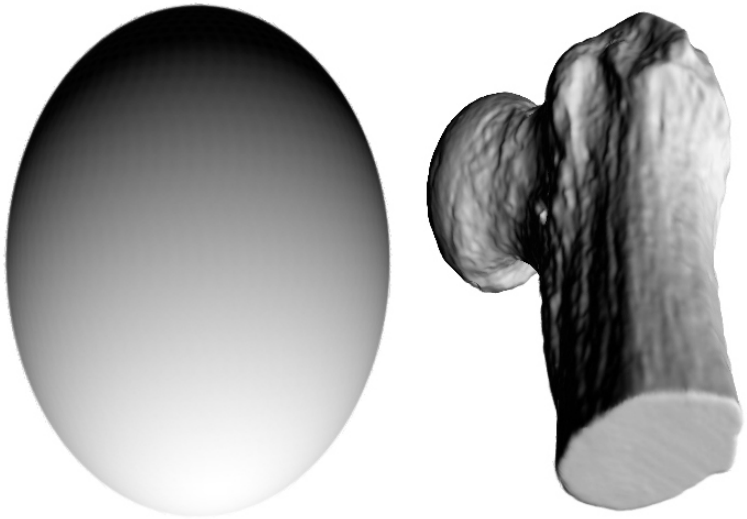


Figura B.4: *Sphere* (3203 *splats*, 648 *FPS*) e *Balljoint* (137062 *splats*, 87 *FPS*).

Figura B.5: *Dinosaur* (56194 *splats*, 196 *FPS*) e *Santa* (75781 *splats*, 147 *FPS*).

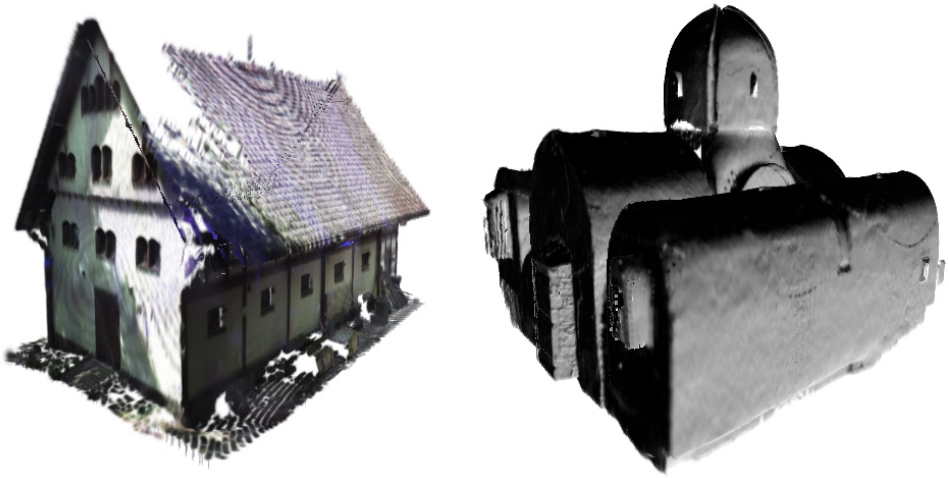


Figura B.6: *Farm House* (241169 *splats*, 50 *FPS*) e *Byzantine Basilica on the island of Crete* (2187366 *splats*, 5.7 *FPS*).

Figura B.7: *Male* (148138 *splats*, 58 *FPS*).

APÊNDICE C RESULTADOS DO MÉTODO DE *HPR* BASEADO EM *SPLATTING*

As imagens a seguir mostram alguns resultados obtidos na visualização direta de nuvens de pontos utilizando o método de remoção de pontos ocultos acelerado por *GPU*.

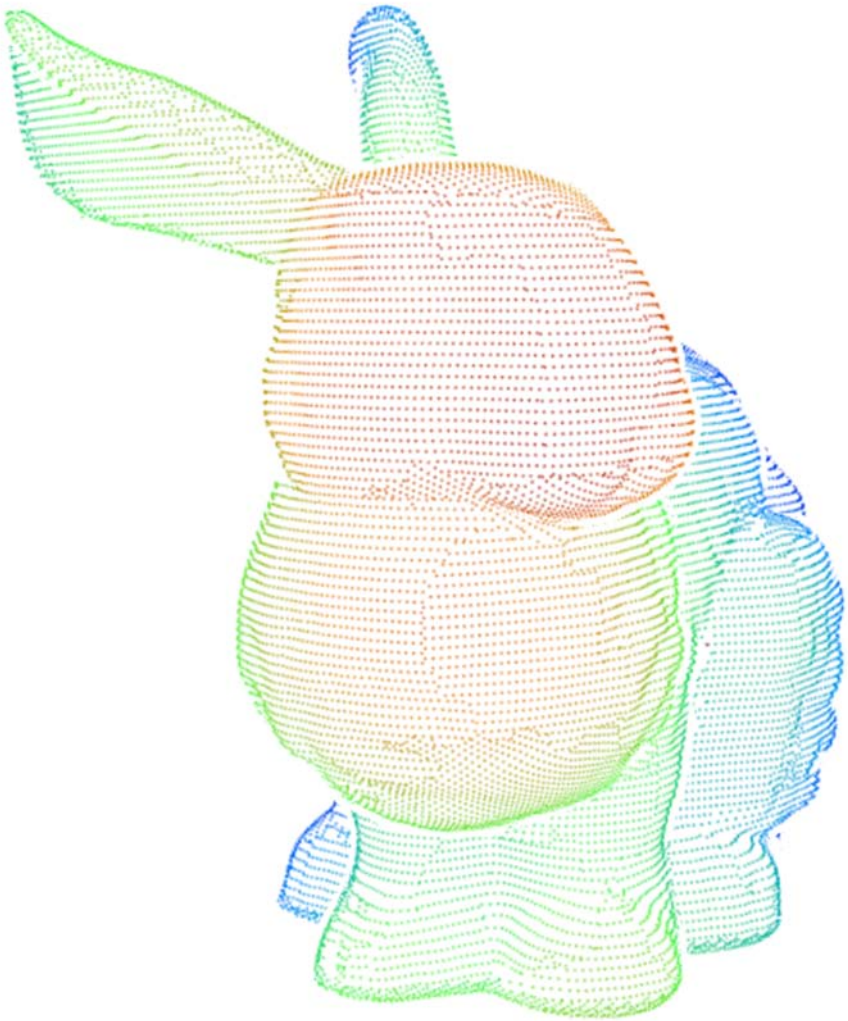


Figura C.1: *Bunny* (35947 pontos, 417.17 *FPS* a *640 x 480*).

Figura C.2: *Armadillo* (172974 pontos, 203.80 *FPS* a *640 x 480*).

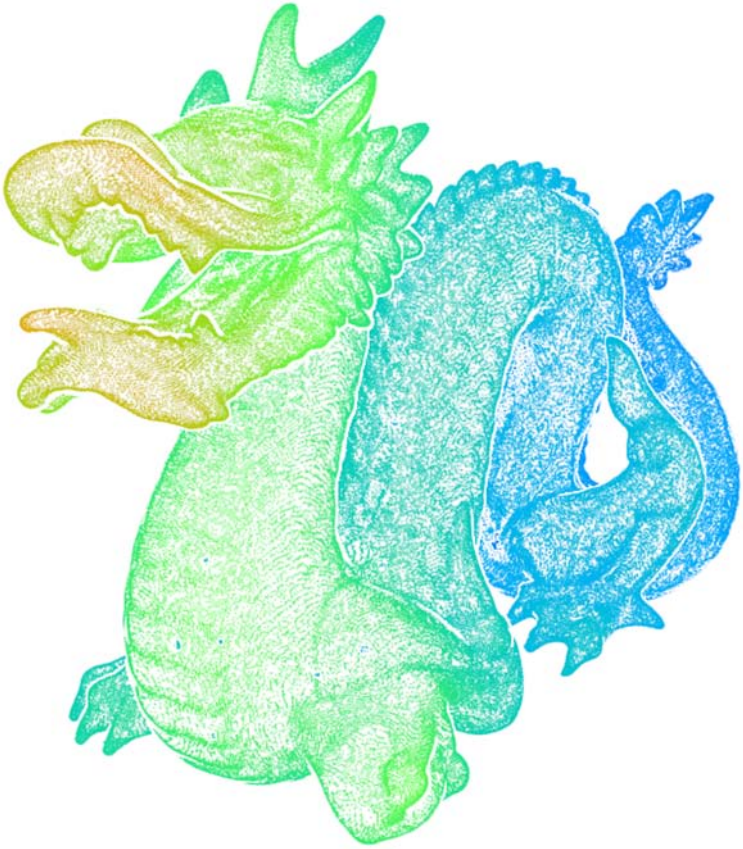


Figura C.3: *Dragon* (437645 pontos, 64.37 *FPS* a *640 x 480*).

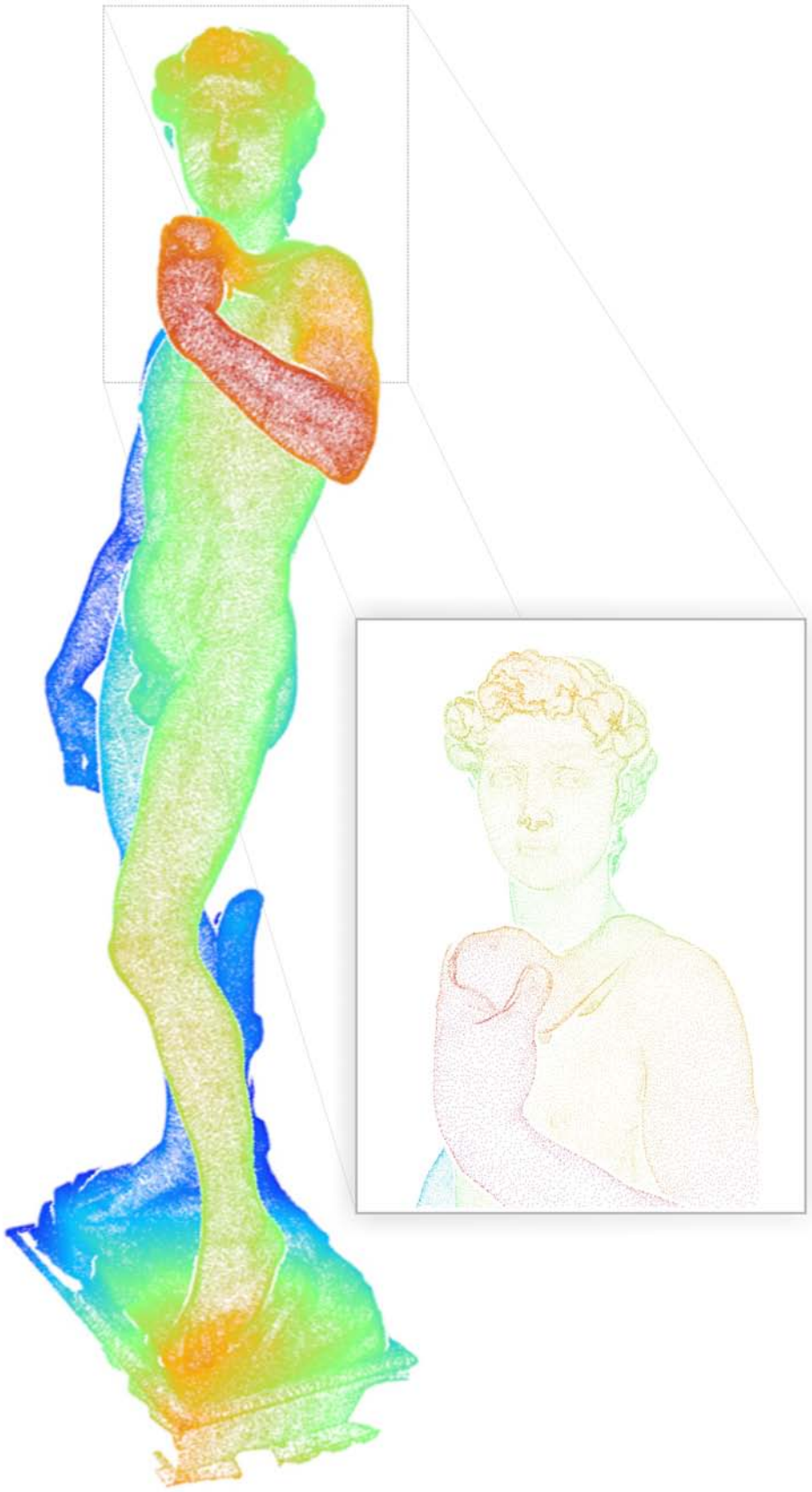


Figura C.4: *David* (257778 pontos, 110.38 *FPS* a *640 x 480*).

As tabelas a seguir mostram os resultados de performance e de precisão obtidos com operador remoção de pontos ocultos acelerado por *GPU*.

(a) Parâmetros de configuração da câmera.

Modelo	Posição	Direção	Cima	*znear*	*zfar*	*fov*
Bunny	$(8.5, 10.0, 10.5)$	$(-1.0, 5.5, 0.0)$	$(0.0, 1.0, 0.0)$	10.0	30.0	45.0

(b) Estatísticas de precisão para o operador utilizando *depth buffer* original.

Viewport	Raio dos Impostores	Pontos Visíveis Corretos	Pontos Visíveis	Falsos Positivo	Falsos Negativo	Erro (%)
640 x 480	0.048	10640	10584	1247	1303	11.91%
1024 x 768	0.048	11317	11273	1119	1201	10.22%
2048 x 1536	0.05	11510	11464	1107	1153	9.83%

(c) Estatísticas de precisão para o operador utilizando *depth buffer* modificado.

Viewport	Raio dos Impostores	Pontos Visíveis Corretos	Pontos Visíveis	Falsos Positivo	Falsos Negativo	Erro (%)
640 x 480	0.048	10640	10632	987	995	9.32%
1024 x 768	0.048	11317	11265	997	1049	9.02%
2048 x 1536	0.05	11510	11485	1041	1066	9.2%

(d) Estatísticas de performance para o operador utilizando *depth buffer* original.

Viewport	Raio dos Impostores	Pontos/Segundo
640 x 480	0.048	1568220
1024 x 768	0.048	619405
2048 x 1536	0.05	363101

(e) Estatísticas de performance para o operador utilizando *depth buffer* modificado.

Viewport	Raio dos Impostores	Pontos/Segundo
640 x 480	0.048	1569690
1024 x 768	0.048	608013
2048 x 1536	0.05	360353

(f) Estatísticas de performance para visualização direta de nuvens de pontos utilizando *depth buffer* original.

Viewport	Raio dos Impostores	*Frames*/Segundo
640 x 480	0.048	1128.87
1024 x 768	0.048	605.79
2048 x 1536	0.05	387.23

(g) Estatísticas de performance para visualização direta de nuvens de pontos utilizando *depth buffer* modificado.

Viewport	Raio dos Impostores	*Frames*/Segundo
640 x 480	0.048	417.17
1024 x 768	0.048	194.45
2048 x 1536	0.05	78.84

Tabela C.1: Resultados obtidos para o Modelo *Bunny* utilizando o método *HPR* acelerado por *hardware*.

(a) Parâmetros de configuração da câmera.

Modelo	Posição	Direção	Cima	*znear*	*zfar*	*fov*
Armadillo	$(6.35, 8.01, -5.01)$	$(-2.66, 2.08, 1.20)$	$(0.0, 1.0, 0.0)$	3.0	18.0	45.0

(b) Estatísticas de precisão para o operador utilizando *depth buffer* original.

Viewport	Raio dos Impostores	Pontos Visíveis Corretos	Pontos Visíveis	Falsos Positivo	Falsos Negativo	Erro (%)
640 x 480	0.021	37508	37278	5088	5318	13.84%
1024 x 768	0.024	40994	40654	4345	4685	11.03%
2048 x 1536	0.026	41805	41449	4222	4578	10.48%

(c) Estatísticas de precisão para o operador utilizando *depth buffer* modificado.

Viewport	Raio dos Impostores	Pontos Visíveis Corretos	Pontos Visíveis	Falsos Positivo	Falsos Negativo	Erro (%)
640 x 480	0.023	37508	37227	4181	4462	11.41%
1024 x 768	0.025	40994	40665	3715	4044	9.53%
2048 x 1536	0.026	41805	41614	3837	4028	9.45%

(d) Estatísticas de performance para o operador utilizando *depth buffer* original.

Viewport	Raio dos Impostores	Pontos/Segundo
640 x 480	0.021	6663470
1024 x 768	0.024	2744530
2048 x 1536	0.026	1760700

(e) Estatísticas de performance para o operador utilizando *depth buffer* modificado.

Viewport	Raio dos Impostores	Pontos/Segundo
640 x 480	0.023	6584100
1024 x 768	0.025	2784890
2048 x 1536	0.026	1791520

(f) Estatísticas de performance para visualização direta de nuvens de pontos utilizando *depth buffer* original.

Viewport	Raio dos Impostores	*Frames*/Segundo
640 x 480	0.021	339.98
1024 x 768	0.024	197.80
2048 x 1536	0.026	161.52

(g) Estatísticas de performance para visualização direta de nuvens de pontos utilizando *depth buffer* modificado.

Viewport	Raio dos Impostores	*Frames*/Segundo
640 x 480	0.023	203.80
1024 x 768	0.025	104.06
2048 x 1536	0.026	49.46

Tabela C.2: Resultados obtidos para o Modelo *Armadillo* utilizando o método *HPR* acelerado por *hardware*.