UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Renan Teston Inácio

VISUALIZAÇÃO DE CONJUNTOS DE DADOS GRANDES FORMADOS POR ESFERAS

Florianópolis

2012

Renan Teston Inácio

VISUALIZAÇÃO DE CONJUNTOS DE DADOS GRANDES FORMADOS POR ESFERAS

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Aldo von Wangenheim

Florianópolis

2012

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Renan Teston Inácio

VISUALIZAÇÃO DE CONJUNTOS DE DADOS GRANDES FORMADOS POR ESFERAS

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 03 de setembro 2012.

Ronaldo dos Santos Mello
Coordenador do Curso

Banca Examinadora:

Prof. Dr. rer. nat. Aldo von Wangenheim
Presidente

Dr. Eng. Rodrigo Surmas

Prof. Dr. Celso Peres Fernandes

Prof. Dr. rer. nat. Eros Comunello

RESUMO

Este trabalho trata da visualização gráfica de conjuntos de dados cujas principais características é ser formado por esferas e possuir um grande volume de informações. As principais dificuldades em se trabalhar com dados numerosos é, além da performance, exibir a informação de forma clara. Para solucionar estes problemas foi criado um renderizador básico baseado na técnica de *sprites* com otimizações em nível de objeto e de imagem. Posteriormente foram analisados algoritmos de renderização com suporte a transparência, onde o *depth peeling* se mostrou adequado. O algoritmo foi adaptado para permitir a geração de uma imagem incompleta, tendo em vista a performance, porém foi feita uma quantificação do erro para limites aceitáveis poderem ser estabelecidos. Finalmente a modalidade de renderização utilizando a técnica de *ambient occlusion* foi implementada para melhor compreensão espacial dos dados, usando otimização de *deferred shading*. A performance obtida foi suficiente para a visualização interativa dos conjuntos de dados.

Palavras-chave: Renderização de sprites. Depth peeling. Ambient occlusion. GPU.

ABSTRACT

This work deals with the graphical visualization of datasets which the main feature is being formed by spheres e have a great volume of information. The main dificulties when working with numerous data is, in addition to performance, show the information in a clear way. In order to solve these problems a basic renderer was created based on sprites techniques with optimizations at object and image level. Afterwards rendering algorithms with translucency support were analyzed, where depth peeling shown itself adequate. The algorithm was adapted to allow the generation of an incomplete image, in view of performance, but a quantification of the error was created so acceptable thresholds can be established. Finally the rendering mode using the ambient occlusion technique was implemented for better spatial comprehension of the data, using deferred shading optimization. The performance was enough for a interactive visualization of the datasets.

Keywords: Sprite rendering. Depth peeling. Ambient Occlusion. GPU.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O sistema visual humano é altamente complexo e nos fornece uma grande capacidade de processamento de informação (BLUMBERG; KREIMAN, 2010). Por este motivo, a visualização gráfica é uma ferramenta poderosa para a compreensão de um conjunto de dados, podendo nos fornecer maneiras alternativas de enxergar esses dados. Segundo Card, Mackinlay e Shneiderman (1999), muitos avanços na ciência foram possíveis com invenções que permitiram ver as mesmas coisas, mas de maneira diferente. Um objeto pode ser observado através de diferentes instrumentos, como microscópios, osciloscópios e tomógrafos, como também pode-se utilizar diagramas, mapas e desenhos para representar as características mais importantes. Com ferramentas computacionais é possível combinar representações visuais com dados instrumentais para melhor compreensão daquilo que se deseja analisar.

Conjuntos de dados possuem diferentes características a serem observadas e por isso existem diferentes técnicas de visualização que podem ser empregadas. Considere, por exemplo, um conjunto de dados gerados por simulação computacional de fluidos. Um dos modelos para esse tipo de simulação é o Smoothed Particle Hydrodynamics (GINGOLD; MONAGHAN, 1977), também chamado de SPH, que discretiza as equações de dinâmica de fluidos através de partículas que interagem entre si. Essas partículas formam implicitamente um campo de densidades, podendo ser visualizado através de técnicas de renderização volumétrica (INÁCIO et al., 2010), adequado para aplicações onde é desejada uma representação visualmente realista da superfície do fluido. Por outro lado, pode-se querer observar os atributos calculados pela simulação, como pressão e velocidade em cada região do fluido. Nesse caso, cada partícula pode ser representada por uma esfera cuja posição, raio e cor são definidos com base nesses atributos (NOBREGA; CARVALHO; WANGENHEIM, 2009). Pode-se verificar na Figura 1 essas diferentes modalidades de visualização de sistemas SPH.

Esferas também podem ser utilizadas para inspecionar os dados intermediários do processo de geração de redes de poros. Uma das aplicações das redes de poros é ser utilizada como um modelo simplificado para realizar a caracterização de rochas com o objetivo de obter informações relevantes para extração de petróleo. Uma amostra de rocha do reservatório é digitalizada através de microtomografia e então se obtém um volume no qual é possível identificar o que é rocha e o que é ar. Neste volume são aplicados algoritmos para a geração da rede, sendo um deles o algoritmo de esferas máximas, que consiste em distribuir esferas com o maior tamanho possível no espaço oco

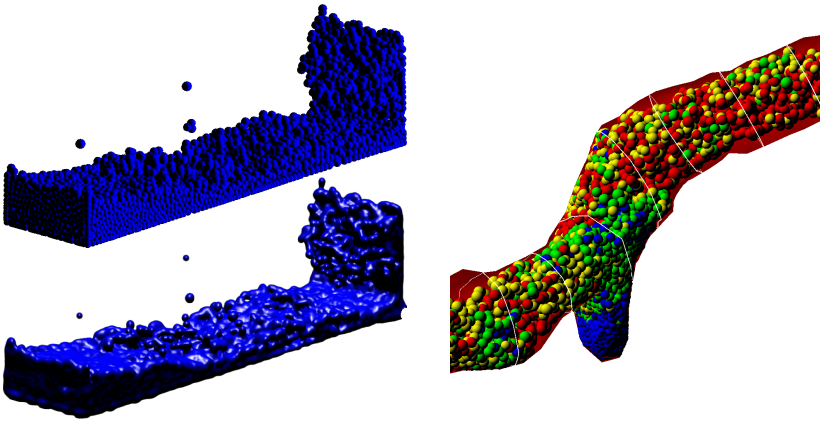

Figura 1: Representações visuais de sistemas SPH. A figura da esquerda ilustra a diferença entre renderizar as partículas como esferas e através de *volume raycasting*. A figura da direita exemplifica a representação atributos da partícula, como a velocidade, através de características visuais.

do volume. Em seguida as esferas redundantes são eliminadas e as restantes são aglomeradas de acordo com critérios de definição de poro (DONG, 2007). Por fim os poros são interconectados e a rede de poros é formada. O próprio algoritmo de esferas máximas trabalha com esferas, então a visualização pode ser feita diretamente sobre o resultado.

1.1 PIPELINE GRÁFICO

O *pipeline* de renderização gráfica, ou apenas *pipeline*, é considerado o núcleo gráfico de aplicações gráficas de tempo real (MÖLLER; HAINES; HOFFMAN, 2008). A função principal é gerar, ou renderizar, uma imagem bidimensional a partir de uma descrição tridimensional de objetos, câmera, luzes, etc. As placas de vídeos mais modernas possuem uma unidade de processamento gráfico, também chamada de GPU, específica para acelerar o *pipeline* gráfico. Em placas de vídeo de médio e alto desempenho a GPU acessa diretamente uma memória de vídeo dedicada que fica na própria placa, enquanto nas outras uma região da memória principal do sistema é reservada para a GPU.

A Figura 2 mostra a visão geral do *pipeline*. O dado de entrada é um conjunto de primitivas geométricas e atributos associados a seus vértices. Esses atributos, tais como posição e cor, podem ter sido previamente trans-

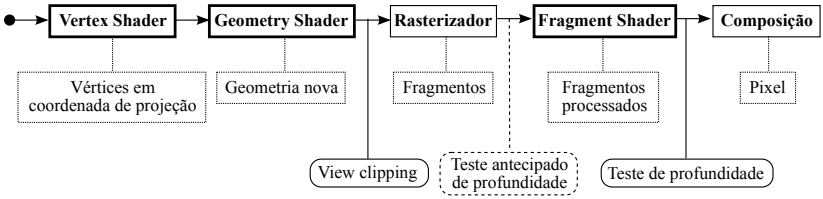

Figura 2: Visão geral do *pipeline* gráfico.

feridos para a memória da GPU, onde ficam armazenados em um *buffer*, ou podem ser alimentados a cada renderização. As etapas com borda em negrito no diagrama, os chamados *shaders*, são programáveis, ou seja, sua funcionalidade pode ser alterada pela aplicação. No caso do OpenGL, a linguagem usada para programar os *shaders* é o GLSL. Cada etapa programável tem suas restrições quanto à entrada e saída esperada, uma vez que essas etapas ainda devem seguir o fluxo do *pipeline*.

Ao longo do pipeline são feitas diversas transformações de coordenadas, dentre os quais: de modelo, de câmera, de projeção e de tela. O sistema de coordenadas de objeto é aquele em que os vértices são posicionados para definir um objeto e geralmente a origem é um ponto de referência do próprio objeto, como o centro. A transformação de modelo posiciona os objetos na cena, colocando os vértices em coordenadas de mundo. A transformação de câmera transforma a cena de acordo com o ponto de vista do observador. A transformação de projeção dita como o espaço 3D será transformado em coordenadas 2D. As coordenadas de tela são as coordenadas dos fragmentos ou *pixels* na imagem resultante da renderização. Existem ainda outros sistemas de coordenadas, sendo estes os principais para este trabalho.

O processamento do pipeline inicia com o processamento individual dos vértices no Vertex Shader. Nesta etapa a posição dos vértices deve passar do sistema de coordenadas de objeto para coordenadas de projeção. O Vertex Shader tem acesso aos atributos do vértice em processamento, porém não tem acesso aos outros vértices ou o tipo de primitiva a que esse vértice pertence (ponto, linha, triângulo, etc).

Após o Vertex Shader existe uma etapa opcional, chamada Geometry Shader. Diferente da etapa anterior, este *shader* processa uma primitiva inteira e tem a informação de todos os vértices que a compõem. Nesta etapa pode-se descartar primitivas ou até mesmo incluir novas. A funcionalidade padrão do *pipeline* é pular esta etapa.

Antes da rasterização da geometria é realizado o *view clipping*, uma etapa em que as primitivas são recortadas para eliminar o que não é visível

na tela. Em seguida o rasterizador discretiza as formas geométricas em fragmentos. Chamamos de fragmento o *pixel* que ainda não está associado a uma imagem. Se não há regras de composição (translucência) e em determinada coordenada há diversos fragmentos, apenas um deles estará visível na imagem final. A seleção de qual fragmento estará visível é feito através do teste de profundidade. Normalmente este teste pode ser feito antes do Fragment Shader, uma vez que se conhece o valor de profundidade do fragmento logo após a rasterização.

O Fragment Shader processa cada fragmento individualmente, sem acesso à informação dos outros fragmentos em processamento, atribuindo um valor de cor. Nesta etapa podem ser feitos cálculos de iluminação e aplicação de texturas. Pode-se alterar também a profundidade do fragmento, porém neste caso a otimização de realizar o teste de profundidade antecipadamente é desabilitada, ocorrendo após o Fragment Shader. Por fim ocorre a composição dos fragmentos que passaram no teste de profundidade para a imagem final.

1.2 SPRITES

A opção de desenhar uma esfera tesselada em triângulos é inicialmente descartada. Isso implicaria em uma grande quantidade de triângulos a serem rasterizados para que as esferas exibam uma silhueta arredondada. Idealmente a geometria seria a mais simples possível e sua rasterização geraria apenas fragmentos que façam parte da esfera. A solução então é utilizar uma figura impostora, ou *sprite*, que é uma forma mais simples do que a que se deseja representar, mas que possui seus fragmentos coloridos de forma a dar noção de um objeto mais complexo.

No caso de esferas, seria utilizado um círculo e a noção de profundidade se daria através de cálculos de iluminação, mas não bastam apenas cálculos de iluminação para simular a profundidade de esferas. Considere o caso de duas esferas se intersectando, ambas esferas deveriam ser desenhadas parcialmente, mas ao desenhá-las como círculos uma esfera ficaria completamente sobreposta à outra, como é ilustrado pela Figura 3. É possível obter a renderização correta através da alteração da profundidade dos fragmentos, de forma que o *sprite* assumiria a forma final de uma concha, correspondente à superfície da esfera.

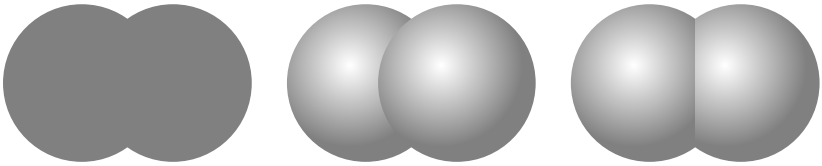

Figura 3: Renderização de esferas através de círculos. A imagem da esquerda mostra os círculos ainda sem cálculos de iluminação, apenas definindo os *pixels* que serão pintados. Na imagem do meio foi aplicado o cálculo de iluminação para dar a noção de profundidade nas esferas, porém as esferas aparecem sobrepostas ao invés de interpenetradas. O resultado esperado é o da imagem da direta.

1.3 OBJETIVOS GERAIS

Visualizar interativamente e de forma compreensível grandes conjuntos de dados cuja característica é serem representáveis por esferas. As esferas devem permitir a exposição de diferentes características do conjunto de dados, por exemplo através dos atributos de cor e tamanho.

1.4 OBJETIVOS ESPECÍFICOS

- Pesquisar as técnicas de renderização na literatura que lidam com grandes quantidades de *sprites*;
- Implementar e adaptar otimizações de caráter geral para renderização de *sprites*;
- Implementar e adaptar técnicas que ajudem na compreensão visual dos dados;
- Realizar testes sistemáticos para identificar o comportamento da implementação em conjuntos com diferentes características.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo será apresentada uma revisão sistemática baseada nos moldes de Kitchenham (2004) para procurar publicações que abordem o problema de renderização de uma grande quantidade de esferas. Para tornar essa busca mais específica, já se considera a renderização através de *sprites*, portanto as pergunta a serem respondidas por esta revisão sistemática é "Como renderizar múltiplos *sprites* em tempo real?" e "Como a visualização pode se tornar mais rica, mantendo performance interativa?".

Na última seção será apresentado um texto mais aprofundado sobre os trabalhos correlatos, não se limitando somente às publicações encontradas pela revisão sistemática, mas considerando também suas referências e publicações as quais já se possuía prévio conhecimento.

2.1 METODOLOGIA

A estratégia de levantamento dos estudos é realizar pesquisas nas bases bibliográficas para, dentre os resultados, selecionar e descartar os artigos que não contribuam para a questão proposta. São consideradas as publicações realizadas em periódicos e conferências e a busca é feita nas bases da ACM Digital Library, IEEE Xplore e ScienceDirect com as expressões de busca listadas na Tabela 1. A seleção das publicações é feita dentre os resultados da busca, verificando através do título, resumo e palavras-chave se elas atendem os seguintes critérios:

1. A publicação se trata de renderização ou visualização tridimensional;
2. É feita renderização de *sprites*;
3. Múltiplos objetos são renderizados.

Os critérios 1 e 2 objetivam descartar publicações que, apesar de serem encontradas através das expressões de busca, possuam um assunto diferente do contexto deste trabalho. O critério 3 tenta impedir que sejam selecionadas publicações cujo foco seja a simplificação de objetos complexos em *sprites*, ao oposto de renderizar grandes quantidades de *sprites*. Se os critérios forem verdadeiros ou não ser possível determinar se são falsos apenas pelos metadados indicados, o artigo é considerado para inspeção do texto completo.

As publicações selecionadas pelos critérios anteriores são inspecionadas para que sejam aplicados os seguintes critérios de exclusão:

Tabela 1: Expressões de busca usadas em cada fonte.

Fonte	Expressão de busca
ACM	((Title:sprite) OR (Abstract:sprite) OR (Title:billboard) OR (Abstract:billboard) OR (Title:impostor) OR (Abstract:impostor) OR (Title:glyph) OR (Abstract:glyph)) AND (real-time OR realtime OR interactive) AND render
IEEE[1]	(("Title":sprite) OR ("Abstract":sprite) OR ("Title":sprites) OR ("Abstract":sprites) OR ("Title":glyph) OR ("Abstract":glyph) OR ("Title":glyphs) OR ("Abstract":glyphs) OR ("Title":billboard) OR ("Abstract":billboard) OR ("Title":billboards) OR ("Abstract":billboards) OR ("Title":impostor) OR ("Abstract":impostor) OR ("Title":impostors) OR ("Abstract":impostors)) AND (real-time OR realtime OR interactive) AND render*
SD[2]	(Title(sprite) OR Abstract(sprite) OR Title(billboard) OR Abstract(billboard) OR Title(impostor) OR Abstract(impostor) OR Title(glyph) OR Abstract(glyph)) AND (real-time OR realtime OR interactive) AND render

[1] A expressão de busca foi aplicada também em texto completo, além de em metadados.

[2] Busca feita nos assuntos Computer Science e Engineering.

1. Não há contribuição para o aumento de performance na renderização de múltiplos *sprites*;
2. Foram feitos experimentos com menos de 10.000 objetos;
3. Não é adequado para uso interativo ou tempo real.

O principal objetivo desse conjunto de critérios é excluir publicações que se tratam de renderizar múltiplos objetos complexos, mas não em quantidade o suficiente para motivar otimização na renderização dos *sprites* em si. Esses artigos, em geral, realizam otimizações de redução de detalhes dos objetos ou de um subconjunto de objetos, como é o caso de renderização de florestas, nuvens e aglomeração de pessoas.

As publicações que restaram são utilizadas também para buscar outras que as referenciam. O Google Scholar é utilizado para este fim e aplicam-se os procedimentos descritos anteriormente para seleção e exclusão.

Por fim as características mais relevantes das publicações, de acordo com o propósito desta revisão sistemática, devem ser extraídas na forma de resumo. Neste resumo devem conter pelo menos as técnicas utilizadas para otimização e os tipos de objetos representados pelos *sprites*.

2.2 RESULTADOS

As buscas nas bases de dados encontraram um total de 241 artigos, dentre os quais 79 foram selecionados para acesso ao texto completo e após aplicados os critérios de exclusão restaram 5 artigos.

A Tabela 2 mostra as publicações resultantes em ordem de publicação. Embora o *frame rate* seja dependente do hardware utilizado em cada publicação, não podendo ser usado para comparação, pode-se ter uma ideia da adequação para uso interativo. O interesse principal é a performance da aplicação, mas foram consideradas também técnicas para melhorar o apelo visual desde que pudessem ser aplicadas a *sprites* arbitrários e a performance continuasse razoável. As características relevantes destas publicações se encontram na Tabela 3. Todas as publicações encontradas tratam de objetos opacos.

2.3 TRABALHOS CORRELATOS

O uso de imagens para substituir uma definição de objeto mais complexa e computacionalmente mais cara é algo muito utilizado em aplicações de tempo real (MÖLLER; HAINES; HOFFMAN, 2008). Catmull (1974)

Tabela 2: Publicações obtidas após critério de exclusão.

Cód.	Qtd.[1]	*Frame rate*	Foco	Referência
P1	60.000	15 fps	Visual	Ambient Occlusion and Edge Cueing for Enhancing Real Time Molecular Visualization (TARINI; CIGNONI; MONTANI, 2006)
P2	52.000	15 fps	Visual	GPU-Based Ray-Casting of Quadratic Surfaces (SIGG et al., 2006)
P3	391.872	9 fps	Performance	Two-Level Approach to Efficient Visualization of Protein Dynamics (LAMPE et al., 2007)
P4	1.000.000	10 fps	Performance	Optimized data transfer for time-dependent, GPU-based glyphs (GROTTEL; REINA; ERTL, 2009)
P5	100.000.000	5 a 20 fps	Performance	Coherent Culling and Shading for Large Molecular Dynamics Visualization (GROTTEL et al., 2010)

[1] O tipo de objeto depende da aplicação, mas o importante é que cada um é desenhado através de um *sprite* e em geral como esfera.

Tabela 3: Características das publicações selecionadas.

Cód.	Otimização	Visualização
P1		Esferas e cilindros; *Ambient occlusion;* Halo.
P2	Simplificação do cálculo de superfícies quádricas.	Esferas, elipsoides e cilindros; *Deferred shading;* Sombra; Silhueta.
P3	Criação do *sprite* no *geometry shader.*	Esferas.
P4	Estratégias de *upload* para GPU; Quantização.	Esferas e cilindros.
P5	*Frustum culling;* *Occlusion culling;* *Caching;* Quantização.	Esferas; *Deferred shading.*

desenvolveu um algoritmo de subdivisão para renderizar superfícies e uma das consequências desse trabalho foi um método para mapear fotografias em partes da superfície. Este trabalho foi estendido (BLINN; NEWELL, 1976) na área de texturização e reflexão para gerar resultados com maior riqueza de detalhes. O mapeamento de texturas foi um grande avanço por permitir de forma eficiente o aumento da complexidade aparente dos objetos em uma cena, sendo utilizado para especificar outros atributos de superfície além da cor, como opacidade, normal e reflexão (HECKBERT, 1986).

Segal et al. (1992) demonstraram como projetar imagens sobre uma geometria qualquer de forma correta em relação à perspectiva, inclusive com aceleração por *hardware*. Com base nesta técnica, surgiram outras que permitiram a representação de cenas tridimensionais complexas apenas através da composição de imagens (CHEN; WILLIAMS, 1993) ou de forma híbrida, utilizando geometria mais simples o que a cena que deseja-se representar e nelas projetando as imagens (DEBEVEC; TAYLOR; MALIK, 1996; DEBEVEC; YU; BOSHOKOV, 1998). Essas técnicas são conhecidas como técnicas de renderização baseada em imagem.

Gumhold (2003) descreve como renderizar elipsoides através de geometria impostora, também chamada de *sprite,* corrigindo os valores de profundidade para cada pixel. Bajaj e Djeu (2004) aceleraram a renderização esferas, cilindros e hélices de forma parecida, enquanto Sigg et al. (2006)

simplificaram as equações quádricas e tornaram o cálculo correto em perspectiva. Tarini, Cignoni e Montani (2006) aperfeiçoou a visualização de múltiplas esferas e cilindros através da aplicação de *ambient occlusion* para realçar a noção de profundidade dos objetos. Lampe et al. (2007) utilizou a etapa programável do *geometry shader* para criar a geometria impostora a partir de um único vértice. Assim como na renderização baseada em imagem, esses trabalhos utilizam a técnica de projetar imagens sobre geometria simples através de um mapeamento de coordenadas. A diferença é que ao invés de simplesmente projetar imagens, utiliza-se o mapeamento de coordenadas para gerar a imagem dos objetos durante a renderização, mais especificamente através da programação de *shaders*. Estas técnicas, porém, impedem uma otimização do teste de profundidade feita pela GPU por causa do momento em que os valores de profundidade são definidos. Grottel et al. (2010) diminui a quantidade de geometria a ser rasterizada através de *culling* em 2 etapas, amenizando o impacto de performance causado pela falta dessa otimização. Previamente, Grottel, Reina e Ertl (2009) já haviam feito testes de performance entre os modos de transferência de dados entre CPU e GPU. Todas as essas técnicas renderizam apenas esferas opacas.

Segundo Foley et al. (1996), a incorporação de translucidez em sistemas baseados em *z-buffer*, tal como o pipeline padrão das APIs modernas como OpenGL e Direct3D, é difícil porque os polígonos são desenhados na ordem em que são transmitidos. Mammen (1989) desenvolveu um algoritmo para renderizar objetos translúcidos independente da ordem dos objetos. Esse método trabalha com os fragmentos gerados pela rasterização e a ordenação deles é feita através de múltiplos passos de renderização, desenhando uma camada de transparência após a outra e por fim compondo-as para resultar na imagem final. Futuramente esse algoritmo foi chamado de *depth peeling* por Everitt (2001), que mapeou a implementação para GPUs, e é considerado o estado da arte em renderização com transparência sem dependência de ordem (HUANG et al., 2011). Esse método possui algumas variações para tentar otimizar o processo através da renderização de múltiplas camadas por passo de renderização (LIU; WEI; XU, 2006; BAVOIL; MYERS, 2008; LIU et al., 2009), e portanto diminuindo a quantidade de passos.

3 RENDERIZAÇÃO DE ESFERAS OPACAS

Neste capítulo serão apresentadas técnicas para renderização de grandes quantidades de esferas, com otimizações que não prejudiquem a qualidade visual. Isso significa que os resultados visuais dos algoritmos otimizados não devem ser inferiores à versão sem otimização, sendo portanto um critério de validação. As otimizações podem implicar em um custo de processamento adicional na etapa de preparação do algoritmo, mas o tempo total de renderização deve ser menor para um grande número de esferas.

Na introdução foi mencionada que as esferas seriam renderizadas através de figuras impostoras, ou *sprites*, mais especificamente um círculo preenchido. Para desenhar um círculo que seja sempre redondo, em qualquer nível de ampliação, pode-se desenhar uma forma simples que cubra toda a área que o círculo ocuparia e então descartar os fragmentos que não fazem parte do círculo. Dentre as formas simples pode-se usar um triângulo ou quadrado. A vantagem do triângulo é ser definido com menos vértices, mas o quadrado simplifica os cálculos. Outra opção é utilizar a primitiva de ponto, a geometria mais simples que as APIs gráficas suportam, sendo definido apenas por um vértice. Um tamanho diferente pode ser atribuído a cada ponto, de forma que eles ocupem mais do que apenas um *pixel* na tela.

O OpenGL originalmente possui algumas restrições impostas ao se trabalhar com pontos. Isso porque o suporte à renderização dessa primitiva foi concebido para desenhar pequenos círculos, mas algumas dessas restrições e outras deficiências podem ser contornadas de diferentes maneiras. Uma delas é a extensão *ARB_point_sprite* (CRAIGHEAD; KILGARD; BROWN, 2003). Quando os mecanismos dessa extensão são utilizados, os pontos passam a ser tratados como *sprites*, ou seja, eles são rasterizados como quadrados e as coordenadas de texturas são automaticamente calculadas. Essa extensão permite, então, que sejam desenhados quadrados no plano da tela, com textura, através da definição de um único vértice e um valor de tamanho. Não é necessário utilizar uma imagem como textura, mas com as coordenadas é possível mapear a posição do fragmento em relação ao *sprite* que o contém. Outra alternativa é utilizar o *geometry shader* para transformar o vértice em um quadrado, ou seja, realizando o papel da extensão de *point sprite*. Esta alternativa é mais flexível, mas suas vantagens não ficam claras para a renderização básica de esferas opacas. No capítulo seguinte essa alternativa será melhor elaborada, mas para este capítulo o importante é que serão rasterizados quadrados com coordenadas de textura, efeito que pode ser obtido com ambas as técnicas.

Após a rasterização do *sprite* são gerados fragmentos a serem proces-

sados pelo *fragment shader*. Este processamento consiste em descartar os fragmentos que não fazem parte da esfera, calcular a normal dos fragmentos restantes, aplicar o algoritmo de iluminação e definir o valor de profundidade correto.

O teste de profundidade é uma das soluções para o problema de visibilidade de superfícies, ou seja, determinar o que é visível e o que está oculto por outros objetos da cena. Esta solução utiliza um mapa de profundidade, também chamado de *depth map* ou *z-buffer*, no qual são armazenados os valores de profundidade dos pixels já desenhados no *buffer* de imagem (CATMULL, 1974; STRASSER, 1974). Ao desenhar um novo *pixel*, sua profundidade é comparada com o valor que está no *z-buffer* na mesma posição: se o novo *pixel* estiver mais distante ele é descartado, senão ele é desenhado e então o *z-buffer* é atualizado com o novo valor.

Apenas a posição do fragmento é necessária para realizar o teste de profundidade, pois a cor ou outros atributos são relevantes apenas se ele passar no teste e for de fato escrito na imagem. Por esse motivo o teste de profundidade pode ser realizado logo após a rasterização, uma vez que já se tem a posição do fragmento nesta etapa, reduzindo a quantidade de fragmentos a passar pelo *fragment shader* e portanto reduzindo a quantidade de cálculos a serem realizados. Essa otimização é conhecida como *early z-test*, ou teste antecipado de profundidade. A exceção é no caso do *fragment shader* estar programado para alterar a profundidade do *pixel*, pois nesse caso o *pipeline* não sabe de antemão a profundidade dos fragmentos e se faz necessário que todos os fragmentos passem pela etapa de *shading*.

3.1 IMPLEMENTAÇÃO BÁSICA

Esta seção descreve a implementação básica do algoritmo, sem otimizações, considerando os conceitos apresentados no começo do capítulo. Esta implementação servirá como base para os algoritmos otimizados e como referência de comparação tanto de performance quanto de resultados visuais. As APIs gráficas usadas são OpenGL 2 e GLSL 1.20.

A etapa de preparação do algoritmo preenche *buffer objects* com os atributos de todas as esferas de forma a poderem ser acessadas diretamente pela GPU. Cada esfera é armazenada como um único vértice, com os atributos de posição e raio sendo armazenados em um *buffer*, enquanto a cor e opacidade ficam armazenados em outro, como esquematizado na Figura 4. Uma vez os *buffers* preenchidos, não ocorre mais transferência de dados para a memória de vídeo.

As esferas armazenadas nos *buffers* são desenhadas na forma de pon-

Posições e raios

x_0	y_0	z_0	r_0	x_1	y_1	z_1	r_1	...	x_n	y_n	z_n	r_n

Cores e opacidades

R_0	G_0	B_0	A_0	R_1	G_1	B_1	A_1	...	R_n	G_n	B_n	A_n

Figura 4: Conteúdo dos *buffer objects*. Cada bloco representa um valor escalar em ponto flutuante. A i-ésima esfera possui posição (x_i, y_i, z_i), raio r_i, cor (R_i, G_i, B_i) e opacidade A_i.

tos e passam pelos estágios de *shading* para serem transformadas em esferas de fato. O *vertex shader* é o ponto de partida, sendo responsável pela transformação das posições e raios de acordo com os parâmetros de tela e de projeção. As posições das esferas são armazenadas em coordenadas de objeto e portanto a transformação é feita através das matrizes de transformação usuais: *Model*, *View*, *Projection*. Os raios também estão na escala usada pelas coordenadas de objeto e precisam estar na escala de coordenadas de tela, sendo aplicada então a seguinte equação, no caso de projeção ortogonal:

$$gl\_PointSize = r\frac{v_h}{2n_h}$$

onde r é o raio, v_h é a altura da tela e n_h é a altura do plano de corte próximo, definido pela projeção. Para projeção em perspectiva o valor do *gl_PointSize* obtido por essa equação deve ser dividido pela distância do ponto até o observador. Além dos atributos necessários para renderização, o raio e posição em coordenadas de mundo são repassados para as etapas seguintes através de variáveis do tipo *varying*.

Em uma etapa seguinte os pontos são preparados para rasterização, momento em que ou a extensão *ARB_point_sprite* ou o *geometry shader* entram em ação para transformar esses pontos em *sprites*, ou seja, quadrados que cubram um círculo de raio igual ao tamanho do *gl_PointSize*. No *fragment shader*, descartam-se os fragmentos que não fazem parte da silhueta da esfera e calcula-se o vetor normal em cada fragmento para os cálculos de iluminação. As coordenadas de textura do *sprite* foram geradas no intervalo $[0,1]$ e são usadas para obter a normal da seguinte forma:

$$\begin{aligned} U &= 2T - 1 \\ N &= \left(U_x, U_y, \sqrt{1 - \|U\|}\right) \end{aligned}$$

onde T é a coordenada de textura. O valor de N já se encontra normalizado, a não ser quando $\|U\| > 1$, ou seja, o fragmento não faz parte da silhueta esfera. Por fim, a profundidade de cada fragmento é calculada:

$$\begin{aligned} P &= c + rN \\ P' &= M_{proj} P \\ gl\_FragDepth &= \frac{1}{2}\left(1 + \frac{P'_z}{P'_w}\right) \end{aligned}$$

onde P é o ponto correspondente ao fragmento na superfície da esfera, c é o centro da esfera, M_{proj} é a matriz de projeção 4x4 e P' é o ponto P transformado pela projeção. Note que se não fosse especificado, o valor de $gl\_FragDepth$ seria calculado com $P = c$ para todos os fragmentos. A relação entre essas variáveis com os fragmentos do *sprites* podem ser visualizadas na Figura 5.

Após o *fragment shader* e o teste de profundidade, a imagem final é escrita em uma textura previamente reservada e ela está pronta para ser mostrada na tela.

3.2 OTIMIZAÇÕES DE VISIBILIDADE

Ao desenhar muitas esferas, é esperado que algumas delas fiquem na frente de outras e portanto nem todas serão exibidas na tela ao mesmo tempo. Isso ocorre particularmente quando há aglomerações, mas também quando as esferas estão posicionadas de forma esparsa e são em grande quantidade. Além disso, podem existir esferas fora do campo de visão do observador e portanto também não estarão visíveis. As otimizações dessa seção têm como objetivo fazer com que as esferas que estejam ocultas sejam eliminadas do processo de renderização, tornando o algoritmo mais eficiente.

Todos os objetos que aparecem na imagem resultante da renderização estão incluídos, totalmente ou parcialmente, dentro do campo de visão do observador. Esse campo de visão é descrito formalmente através de um volume de visualização, ou seja, um sólido geométrico que representa a região no espaço que contém os objetos visíveis ao observador. Quando a projeção é em perspectiva o volume de visualização possui o formato de tronco de pirâmide, sendo chamado de *view frustum*, enquanto o da projeção ortogonal possui a forma de paralelepípedo. A otimização que rejeita a renderização dos objetos fora do volume de visualização se chama *view culling*.

Objetos opacos próximos ao observador podem obstruir completa-

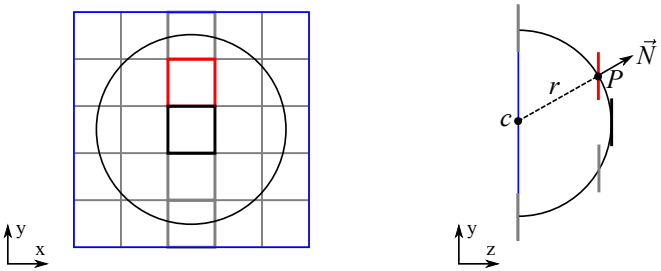

Figura 5: Variáveis usadas no cálculo de profundidade dos fragmentos. A figura da esquerda mostra os fragmentos compondo o *sprite*, juntamente com a esfera projetada sobre eles. Na figura da direita pode-se observar a diferença de profundidade entre os fragmentos do mesmo *sprite*.

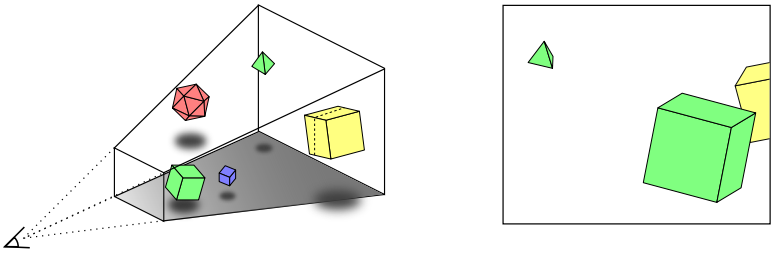

Figura 6: Volume de visualização e a projeção dos objetos visíveis em tela. O objeto vermelho está completamente fora do campo de visão, sendo eliminado pelo *view culling*, enquanto o objeto amarelo está apenas parcialmente fora. O objeto azul está sendo ocluído por outro objeto e portanto não pode ser observado.

mente a observação de objetos mais distantes, mesmo que estes estejam dentro do volume de visualização. A otimização que evita a renderização desses objetos ocluídos é conhecida como *occlusion culling*. A diferença entre essa otimização e o teste de profundidade é que neste último o objeto chega até a etapa de rasterização e o descarte é feito por fragmento, enquanto o *occlusion culling* elimina o objeto nas etapas iniciais da renderização.

A Figura 6 demonstra o conceito de volume de visualização e as situações de *view* e *occlusion culling*. Ambas as otimizações necessitam de uma estrutura de organização espacial para decidir que objetos serão descartados de forma eficiente. Este trabalho segue a proposta de Grottel et al. (2010) e utiliza um *grid* regular.

Como descrito anteriormente, o *early-z test* não pode ser realizado na

renderização dos *sprites* por ser necessário alterar o valor de profundidade dos fragmentos. Para amenizar o impacto na performance, além das otimizações de *culling*, que possuem alta granularidade, é realizado um teste de profundidade aproximado antes da rasterização utilizando uma pirâmide de profundidades.

3.2.1 Construção do *grid* de oclusão

Considerando um conjunto de dados estático, ou seja, as esferas não se movem durante a visualização, o *grid* de oclusão não se modifica após inicializado. O *grid* é regular, isto é, todas as células possuem o mesmo tamanho, e alinhado aos eixos cartesianos. A inicialização consiste em primeiramente encontrar a caixa delimitadora do conjunto de dados, ou *bounding box*, para então dividir este espaço em células. Uma grande quantidade de células implica em testes de visibilidade mais precisos, mas também mais demorados. Quando não especificado neste trabalho, a resolução do *grid* é de 9^3.

Após definido o formato do grid, cada esfera é adicionada na célula correspondente e por fim são calculadas as caixas delimitadoras das células. Diz-se que a célula contém uma esfera se o centro desta esfera está contido na região espacial que a célula ocupa, enquanto a caixa delimitadora de uma célula corresponde à caixa delimitadora das esferas que ela contém. Note que a caixa delimitadora de uma célula podem ocupar um espaço maior do que a própria célula e até mesmo englobar outras células dependendo da resolução do grid e do raio das esferas.

Cada célula pode ser marcada como visível ou invisível, inicialmente todas sendo consideradas invisíveis. As células visíveis são aquelas que passaram nos testes de visibilidade e portanto são aquelas que serão renderizadas. A visibilidade das células é a única informação do grid que se altera durante a visualização, mais precisamente quando o observador muda de posição ou direção.

3.2.2 *View culling*

O *view culling* consiste em verificar se a caixa delimitadora de cada célula está dentro do volume de visualização. As faces do volume de visualização definem planos de corte e se a caixa delimitadora estiver totalmente atrás de um desses planos, então ela não está visível. Assarsson e Möller (2000) apresentam como determinar se uma caixa está atrás de um plano verificando se a distância com sinal do p-vértice da caixa com o plano é ne-

gativa. Detalhes desse procedimento se encontram no Algoritmo 3.1. Para verificar se a caixa vai ser descartada da renderização, basta basta aplicar este algoritmo em todos os planos de corte e verificar se em algum deles retorna verdadeiro.

Algoritmo 3.1 Função que verifica se a caixa está totalmente atrás do plano.

1: **parâmetro** C**:** Caixa alinhada aos eixos cartesianos.
2: **parâmetros** a, b, c, d**:** Coeficientes da equação cartesiana do plano.
3: **início**
4: $V \leftarrow (C_{xmín}, C_{ymín}, C_{zmín})$
5: **se** $a \geq 0$ **então** $V_x \leftarrow C_{xmáx}$
6: **se** $b \geq 0$ **então** $V_y \leftarrow C_{ymáx}$
7: **se** $c \geq 0$ **então** $V_z \leftarrow C_{zmáx}$
8: $dist \leftarrow aV_x + bV_y + cV_z + d$
9: **retorna** $dist < 0$
10: **fim**

3.2.3 *Occlusion culling*

As células que se encontram dentro do volume de visualização passam em seguida pelo teste de oclusão. Os passos necessários para esse teste envolvem o ordenamento das células em relação ao observador, geração de um mapa de profundidades de estimado e teste de oclusão auxiliado por GPU.

O mapa de profundidades é aproximado porque não é gerado a partir dos valores de profundidade das esferas. Cada *sprite* é transformado em um círculo que possui a mesma profundidade do centro da esfera, resultando em um mapa cujos valores de profundidade nunca são inferiores (mais próximos) aos valores do mapa exato. Isso garante que se um fragmento for descartado por estar atrás de outro, de acordo com o mapa aproximado, então ele seria descartado também pelo mapa exato. O motivo de construir esse mapa de forma aproximada é que dessa forma as profundidades dos fragmentos não são alteradas, permitindo a realização do teste antecipado de profundidade. Com o objetivo de obter um melhor aproveitamento, as células mais próximas são renderizadas primeiro, minimizando a quantidade de fragmentos que passam no teste e depois venham a ser sobrescritos. As células são previamente ordenadas de acordo com o vértice mais próximo ao observador da caixa delimitadora da célula.

O OpenGL possui um mecanismo para facilitar testes de oclusão chamado *occlusion query* (CUNNIFF et al., 2003). Antes da renderização é in-

dicado que uma consulta de oclusão será realizada e depois é possível saber quantos fragmentos foram de fato desenhados, ou seja, os que passaram por todos os testes do *pipeline*. Utilizando essa consulta é possível testar se as as células estão visíveis de acordo com o mapa de profundidades aproximado, através da renderização das caixas delimitadoras das células na forma de paralelepípedo. A quantidade de células exerce forte influência na performance desta etapa pois é necessário criar uma consulta de oclusão para cada uma delas. Considera-se que uma célula está ocluída quando nenhum fragmento de sua caixa delimitadora foi desenhado.

A Figura 7 exemplifica as etapas do *occlusion culling*. A imagem da esquerda ilustra a criação do mapa de profundidades aproximado utilizando apenas o círculo alinhado ao observador que passa pelo centro da esfera. Na imagem da direita estão representadas as caixas delimitadores das células e os testes de oclusão que falham. Pode-se observar que as caixas delimitadoras podem ter tamanhos e posições espaciais bem diferentes das células correspondentes. Note que como a caixa maior passou no teste, nenhuma esfera daquela célula é considerada ocluída nesta etapa, nem mesmo a esfera pequena que não passaria no teste de profundidade.

3.2.4 Pirâmide de profundidades

As otimizações de *culling* impedem a renderização de esferas no nível de célula, descartando um conjunto de objetos com base na visibilidade da caixa que delimita todos esses objetos. Porém há situações em que apenas uma das esferas está realmente visível, ou então uma região da caixa delimitadora está visível sem conter uma esfera ali. Nesses casos todas as esferas entrarão no processo de renderização e serão gerados e processados fragmentos que não estarão presentes no resultado. Para descartar as esferas ocluídas em uma granularidade mais baixa é utilizada uma pirâmide de profundidades.

Greene, Kass e Miller (1993) propõem uma estrutura de pirâmide para realizar o teste de visibilidade em nível de polígonos, além de fragmentos, descartando antecipadamente alguns polígonos ocultos sem a necessidade de rasterizá-los. A estrutural piramidal é semelhante à ideia de *mipmaps*, que são coleções da mesma imagem em diferentes níveis de detalhes. A base da pirâmide é o mapa de profundidades completo, que é o que possui o nível de detalhes mais fino, enquanto em cada nível acima há um mapa de menor resolução, metade da largura e altura, gerado a partir do nível inferior. Nesses níveis intermediários um elemento do mapa armazena a profundidade mais distante dentre os correspondentes do nível inferior, de forma que o nível mais alto da pirâmide possua apenas a profundidade máxima da cena inteira.

O teste de profundidade hierárquico é recursivo: começa pelos níveis mais altos e, caso o polígono não esteja oculto, é feito o teste no nível inferior. No melhor caso, o polígono pode ser determinado como oculto apenas checando poucos elementos nos níveis mais altos da pirâmide, enquanto no pior caso o teste chegará até a base e se torna equivalente ao teste de profundidade por fragmento. Ao passar no teste, o polígono atualiza a pirâmide de acordo com seus valores de profundidade.

Embora a técnica descrita no parágrafo anterior possa substituir o teste de profundidade do *pipeline*, neste trabalho ela é utilizada para determinar rapidamente a visibilidade das caixas delimitadoras dos *sprites*. As diferenças são que a pirâmide não será modificada uma vez construída e o teste não será recursivo. A pirâmide é construída tendo como base o mapa de profundidades aproximado que foi gerado para o *occlusion culling*. O teste é realizado apenas em um nível da pirâmide, sendo este o mais detalhado tal que um elemento do mapa possa cobrir totalmente o *sprite*, assim a caixa delimitadora só precisa ser testada com no máximo 4 elementos do mapa.

A construção da pirâmide é feita em CPU, enquanto a GPU está ocupada com as consultas de oclusão. Após terminadas as consultas e a construção da pirâmide, inicia-se a renderização das esferas e no *vertex shader* é feito o teste de profundidade com a pirâmide. Neste *shader* não é possível descartar vértices, porém o *sprite* pode ser posicionado em uma região fora do volume de visualização para que a rasterização não gere fragmentos.

A Figura 8 exemplifica uma situação, ilustrando o mapa de profundidades exato, um teste de profundidade em um nível da pirâmide e o nível imediatamente inferior. Para o *sprite* azul passar no teste de profundidade, ele precisa possuir profundidade menor que a dos 2 elementos sobrepostos por ele, em caso contrário ele já pode ser descartado. O *sprite* verde certamente passa no teste, pois uma parte dele se encontra sobre um elemento com valor de profundidade máximo. Note que se o teste fosse feito no nível inferior, mais detalhado, o *sprite* verde poderia ter sido eliminado, porém checamos apenas um nível, uma vez que o objetivo desse teste é eliminar *sprites* rapidamente e não determinar a visibilidade com precisão.

3.3 RESULTADOS

Para medir o desempenho foram escolhidos alguns conjuntos de dados com diferentes características de distribuição de esferas para verificar a efetividade do algoritmo em diversas situações, como mostra a Tabela 4. O conjunto C2 é o resultado do algoritmo de esferas máximas sobre um volume que representa o espaço de poros de uma rocha, enquanto os conjuntos C1,

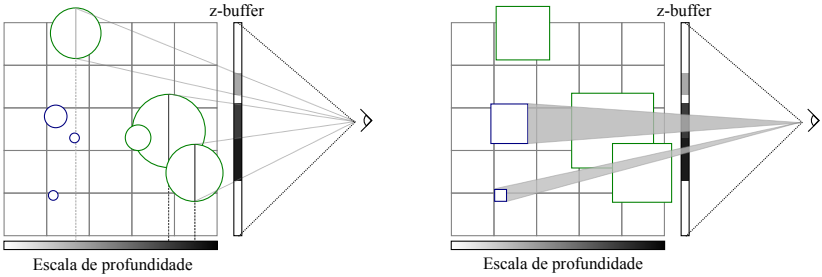

Figura 7: Criação do *z-buffer* e teste de oclusão das caixas delimitadoras para *occlusion culling*.

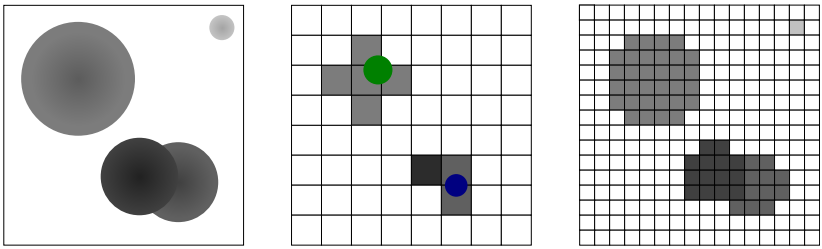

Figura 8: Teste utilizando um nível da pirâmide de profundidade. À esquerda se encontra o mapa de profundidades exato, enquanto a imagem do meio mostra o nível da pirâmide em que é feito o teste de profundidade para os *sprites* destacados. À direita pode-se observar o nível imediatamente inferior àquele utilizado pelo teste.

C2 e C4 foram gerados sinteticamente para forçar as características descritas.

O cenário de teste é uma navegação com a câmera orbitando suavemente em volta do conjunto de dados. No início o conjunto é todo visível na tela, mas conforme a navegação progride a câmera se aproxima e por fim se afasta. Dessa forma pode-se observar o comportamento dos algoritmos durante a variação de tamanho, em *pixels*, e da quantidade visível das esferas. O teste renderiza o total de 180 *frames* com resolução de 1024x1024 *pixels*.

O *hardware* em que os testes foram realizados é um notebook Asus G50VT-X1, seu processador sendo um Intel Core2 Duo 2.26 GHz, 4 GB de RAM e placa de vídeo NVIDIA GeForce 9800M GS (512MB). O sistema operacional é Ubuntu 11.04 versão 64 bits com o *driver* de vídeo da NVIDIA versão 270.41.19.

Os conjuntos de dados descritos, juntamente com o cenário de teste e plataforma, serão os mesmos usados nos capítulos seguintes para permitir uma comparação sob mesmas condições. Como para os capítulos seguintes é importante que a técnica de geração de *sprites* seja através de *geometry shader*, os testes foram realizados nesta condição.

A Figura 9 sintetiza a média dos tempos de renderização de todos os conjuntos com e sem a otimização, enquanto cada gráfico da Figura 10 corresponde a uma execução do cenário de teste para determinado conjunto de dados, permitindo a comparação entre o algoritmo básico e o algoritmo com otimizações de visibilidade ao longo da execução. Para os conjuntos C3 e C4, que são os maiores, a otimização mostrou claros benefícios no tempo de renderização, uma vez que nesses casos há maior chance de haver oclusão entre os objetos e portanto há uma grande redução na quantidade de dados processados. Para os casos C1 e C2 obteve-se um ganho maior no momento da visualização em que o observador está mais próximo, ou seja, há menos esferas no campo de visão e que ocupam uma área maior em *pixels*.

No conjunto C1 nota-se claramente uma proporcionalidade entre a aproximação do observador e o tempo de renderização. Isso acontece porque a característica desse conjunto é haver uma concentração das esferas no centro e à medida que o observador se aproxima dessa aglomeração as esferas ficam maiores, ocupando mais *pixels* da tela e sobrecarregando o rasterizador. Essa situação exemplifica o principal problema que as otimizações implementadas buscam resolver, que é evitar a geração de fragmentos que não serão aproveitados. Note que para todos os conjuntos de dados, os momentos em que o observador está mais próximo são os de melhor performance para o algoritmo otimizado, tendo em vista que há uma chance maior de haver mais esferas oclusas nesses momentos.

Tabela 4: Conjuntos de dados para medição de performance.

	Qtd.	Características
C1	1.000.000	Conjunto esparso com aglomeração no centro
C2	2.463.048	Aglomerações pequenas e interconectadas
C3	8.000.000	Muitas aglomerações pequenas e separadas
C4	27.000.000	Grande aglomeração em bloco

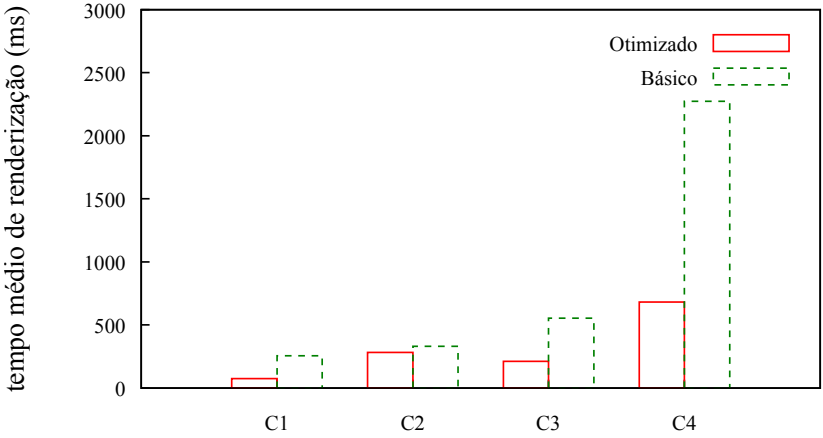

Figura 9: Tempos médios de renderização de cada conjunto.

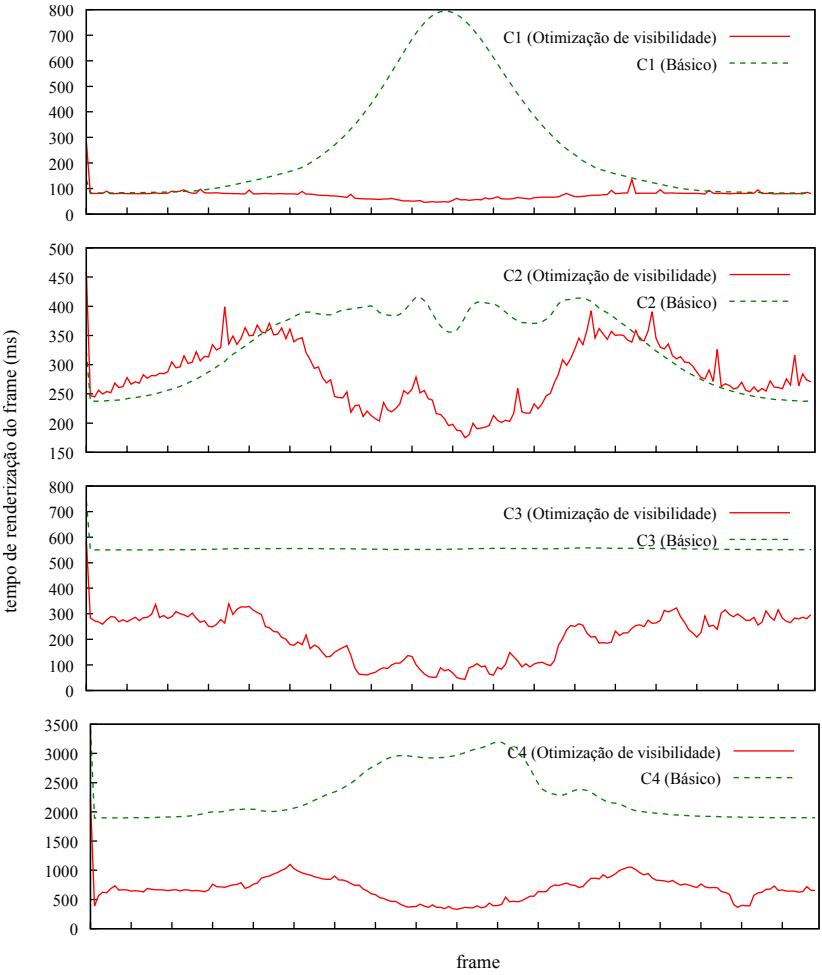

Figura 10: Gráficos de tempo de renderização no cenário de teste. O eixo horizontal indica o frame de renderização e o eixo vertical indica o tempo de renderização em milissegundos.

4 RENDERIZAÇÃO DE ESFERAS TRANSLÚCIDAS

Porter e Duff (1984) formalizaram o conceito de composição de imagens digitais com uma álgebra de composições. Além dos 3 canais de cores RGB para o vermelho, verde e azul, utiliza-se o canal *alpha*, que representa a opacidade do pixel. A técnica de composição través do canal *alpha* já havia sido demonstrada anteriormente (WALLACE, 1981), porém com a simplificação de pré-multiplicar as cores pelo *alpha*, foi possível definir operadores para compor as imagens de formas diferentes. No contexto de renderização 3D, a composição nos permite renderizar objetos translúcidos através da mesclagem dos fragmentos gerados pela rasterização dos objetos.

A grande dificuldade em lidar com renderização de objetos semitransparentes é a determinação de visibilidade e em que ordem os objetos devem ser renderizados e mesclados. Enquanto o teste de profundidade descarta fragmentos para dar lugar a outros que estão mais próximo ao observador, o mesmo não pode ser feito se os fragmentos em questão forem translúcidos. Determinados os fragmentos visíveis, eles devem estar ordenados adequadamente pois a álgebra de composição não é comutativa.

Maule et al. (2011) categorizam as técnicas de transparência em sistemas de rasterização em: ordenação de geometria, ordenação de fragmentos, algoritmos híbridos, independentes de ordem e probabilísticos. A ordenação de geometria e algoritmos híbridos não são adequados para *sprites* cuja profundidade dos fragmentos não corresponde à profundidade da geometria original, enquanto aproximações independentes de ordem e probabilísticas não geram resultados visualmente satisfatórios em cenas com alta complexidade de profundidade. Investigamos então as técnicas de ordenação de fragmentos, que pode ser via *depth peeling* ou *buffer* temporário.

O algoritmo de *depth peeling* possibilita a renderização das camadas visíveis em ordem de profundidade (EVERITT, 2001). A extração das camadas se dá em múltiplos passos de renderização, sendo que em cada passo a camada extraída possui fragmentos imediatamente atrás da camada anterior. O processo de geração de camadas termina quando não há mais fragmentos para formar uma camada, mas pode-se interromper prematuramente quando a quantidade de fragmentos é muito baixa ou já foram geradas camadas suficientes. Em geral, após a extração de uma camada, ela pode ser composta imediatamente e então descartada.

Outra forma de ordenação de fragmentos é armazenar todos os fragmentos gerados em um *buffer* temporário, para então ordená-los e realizar a composição. Myers e Bavoil (2007), por exemplo, utilizam uma textura com suporte a multiamostragem como *buffer* temporário, podendo coletar até 8

fragmentos por passo de renderização, e a ordenação ocorre no momento da composição. Quando há mais fragmentos para capturar, mais *buffers* são utilizados. A desvantagem desses métodos é a quantidade de memória necessária para armazenar os fragmentos.

Foi considerada para estudo também a técnica de Meshkin (2007), que remove os termos dependentes de ordem da equação de composição, obtendo resultados razoáveis para objetos com alta translucidez. Quando muitos objetos estão sendo compostos ou são pouco translúcidos, porém, o resultado é ter algumas regiões muito escuras ou muito claras.

Neste capítulo será explorada a técnica de *depth peeling* pelo requisito de memória reduzido e a habilidade de interromper o algoritmo prematuramente e ainda assim obter um resultado parcial com a ordem correta dos fragmentos mais próximos ao observador.

4.1 GERAÇÃO DE SPRITES

No caso de renderização de esferas opacas, é gerado somente um fragmento para cada ponto da esfera, pois apenas a parte da frente é visível e oclui exatamente todos os fragmentos de trás. A partir do momento que as esferas podem ser translúcidas, esta premissa se torna inválida e passa a ser necessário gerar fragmentos para a parte de trás também. A extensão *ARB_point_sprite* então torna-se insuficiente para a tarefa, visto que os vértices deveriam ser duplicados para poder gerar 2 *sprites* por esfera.

O *geometry shader* é uma etapa programável que ocorre após o *vertex shader* e antes da rasterização, consequentemente antes também do *fragment shader*. Enquanto o *vertex shader* tem apenas a capacidade de transformar os vértices que foram enviados pela aplicação, o *geometry shader* é capaz de criar novas primitivas geométricas a partir das que foram transformadas pelo *vertex shader*. A transformação das novas primitivas deve ser aplicada no próprio *geometry shader* que as cria.

A funcionalidade do *ARB_point_sprite* pode ser então substituída por um *geometry shader* que crie os *sprites* correspondentes para cada ponto. Para isso a geometria de saída é configurada para ser uma tira de triângulos, sendo necessário então definir 4 vértices, como ilustra a Figura 11. Além do posicionamento desses vértices, é atribuído a cada um uma coordenada de textura para uso do *fragment shader*, com componentes no intervalo $[-1, 1]$.

Existem algumas situações em que não é necessário ou adequado utilizar ambos *sprites* para uma esfera. No caso de renderização de esferas opacas, apenas o *sprite* da frente é necessário, por exemplo. Já em uma das etapas de geração do *grid* de oclusão, os *sprites* devem intersectar o centro da esfera

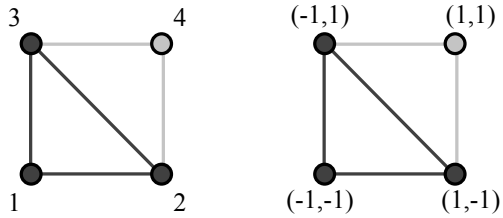

Figura 11: Geração do *sprite* com tira de triângulos. Em uma tira, os primeiros 3 vértices definem o primeiro triângulo e cada vértice subsequente define um novo triângulo utilizando os 2 vértices imediatamente anteriores. O desenho da direita indica as coordenadas de textura.

para formar a silhueta. O *geometry shader* implementado possui parâmetros para suportar essas opções.

4.2 DESCRIÇÃO DAS TÉCNICAS

4.2.1 *Depth peeling* frontal

Também chamado apenas de *depth peeling*, esta é a forma mais básica do algoritmo: extrai as camadas de frente para trás e permite a composição imediata de novas camadas ao *buffer* de resultado. A extração da camada é feita através de uma etapa de renderização com um algoritmo no *fragment shader* que compara a profundidade do fragmento com o mapa de profundidades da camada anterior. Se o fragmento possui uma profundidade menor que o da camada anterior, ele é descartado por já estar em outra camada, caso contrário ele permanece. Na inicialização do algoritmo, o mapa de profundidades anterior é definido com o valor mínimo de forma a não descartar nenhum fragmento no primeiro passo. Após o *fragment shader*, o teste de profundidade elimina os fragmentos mais profundos, restando apenas aqueles que vão compor a nova camada. O mapa de profundidades é atualizado para ser utilizado na extração da próxima camada. Quando nenhum fragmento passa no teste de profundidade, significa que não há mais camadas para gerar e o algoritmo para.

A composição das camadas pode ser feita durante o processo de extração, logo após a formação da nova camada. A direção da composição é de frente para trás, com as seguintes equações:

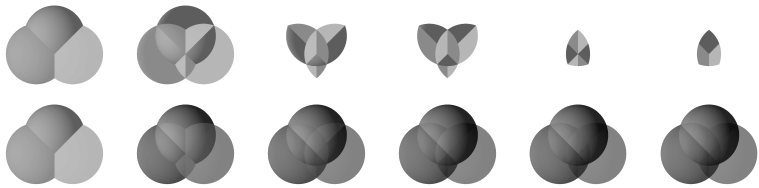

Figura 12: Passos do algoritmo de *peeling* frontal. A primeira linha mostra as camadas na sequência em que são extraídas e a segunda linha corresponde aos estágios da composição parcial.

$$\begin{aligned} I_{RGB} &= C_{RGB} \times C_A \times (1 - D_A) + D_{RGB} \\ I_A &= C_A \times (1 - D_A) + D_A \end{aligned}$$

onde I é o resultado da composição, C é a cor do fragmento a ser composto e D é a cor presente no *buffer* de resultado. Note que a equação para os canais RGB é diferente para a do canal A e o termo $C_{RGB} \times C_A \times (1 - D_A)$ não pode ser configurado no mecanismo padrão de composição do OpenGL. A solução é realizar a multiplicação $C_{RGB} \times C_A$ na geração da camada, ou seja, o valor de C_{RGB} é na verdade $C_{RGB} \times C_A$. A equação para todos os canais se torna então:

$$I = C \times (1 - D_A) + D \tag{4.1}$$

Em uma cena com alta complexidade de profundidade há muitas camadas, porém a composição pode chegar em uma situação em que determinado *pixel* do resultado se torne quase opaco e por causa disso as camadas seguintes não terão contribuição significativa para alterar a sua cor. Nestes casos é possível parar de gerar as camadas prematuramente sem degradação na qualidade visual.

A Figura 12 ilustra a sequência de camadas geradas pelo algoritmo na renderização de 3 esferas e o resultado da composição parcial. Pode-se observar que a primeira camada corresponde à uma renderização sem considerar a translucência dos objetos.

4.2.2 *Dual depth peeling*

Esta variação do algoritmo busca reduzir a quantidade de passos de renderização a ser feito, sem alterar a quantidade de camadas. Enquanto o

depth peeling frontal somente extrai as camadas de frente para trás, o *dual depth peeling* extrai ao mesmo tempo as camadas de trás para frente, parando ao chegar na camada do meio, extraindo então duas camadas a cada passo de renderização. Cada camada é escrita em um alvo de renderização diferente e é usado ainda um terceiro alvo para atualizar o mapa de profundidades. A composição é feita durante a extração das camadas, havendo duas imagens intermediárias correspondentes aos dois sentidos de extração.

O teste de profundidade da funcionalidade padrão é desativado, de forma que as comparações são feitas no *fragment shader* utilizando um mapa de profundidades com 2 valores: um para as camadas da frente e outro para as de trás. Este mapa é utilizado para determinar os fragmentos mais próximos e os mais distantes que devem fazer parte das camadas sendo extraídas, além do descarte de fragmentos já extraídos.

Em um passo de renderização, os fragmentos que ainda não foram extraídos para uma camada escrevem o seu valor de profundidade z no mapa de profundidade com o par $(-z, z)$ e a composição dos fragmentos é feita com o operador MAX, ou seja, o valor de profundidade com que o mapa será atualizado é o menor para a primeira posição, com sinal negativo, e o maior para a segunda. Ainda neste mesmo passo, são selecionados os fragmentos que possuem o valor de profundidade igual ao que estava presente no mapa obtido no passo anterior para formar as novas camadas. A camada da frente é composta no próprio *shader* de extração e o seu valor é escrito na imagem intermediária correspondente. Por não ser possível selecionar operadores diferentes para alvos de renderização diferentes, é utilizado o operador MAX para compor esta camada na imagem intermediária, porém isso não é um problema uma vez que o valor das cores sempre aumenta na composição de frente para trás. A camada de trás é extraída separadamente para ser composta em uma etapa seguinte exclusiva para este fim.

Após todas as camadas serem extraídas, as imagens intermediárias que correspondem às composições de frente para trás e trás para frente são compostas. Este algoritmo não lida bem com paradas prematuras, pois ao compor as imagens intermediárias com camadas faltando, serão visualizados somente os objetos mais próximos e os mais distantes do observador, como se não houvesse nada entre eles. Por outro lado pode-se ignorar as camadas de trás, obtendo o mesmo resultado visual do *front peeling*, mas com menos eficiência já que cada passo de renderização do *dual depth peeling* é mais custoso para poder suportar a geração simultânea de camadas.

A Figura 13 mostra os passos do algoritmo sobre as mesmas esferas da Figura 12 e resultando na mesma imagem. No passo de inicialização o mapa de profundidades é preenchido com os valores de profundidade dos fragmentos que serão extraídos no primeiro passo. Nos passos seguintes a camada

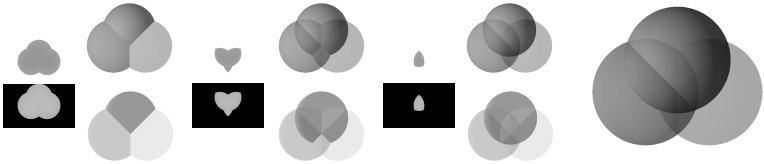

Figura 13: Passos do algoritmo de *dual depth peeling*. Alternadamente se encontram o mapa de profundidades duplo em escala reduzida e a composição parcial das imagens. As extrações frontais e traseiras correspondem respectivamente às imagens de cima e de baixo. A última imagem é o resultado da composição final dos resultados das extrações nas 2 direções.

é extraída com os fragmentos selecionados pelo mapa de profundidades do passo anterior e composta na imagem intermediária correspondente, além de também ser gerado um mapa de profundidades para uso do passo seguinte. Por fim as imagens intermediárias são compostas.

4.2.3 Soma ponderada

Meshkin (2007) expandiu a fórmula de composição para múltiplas camadas com o objetivo de detectar padrões nos termos da fórmula. Separou esses termos então em termos que são dependente da ordem de composição, e termos que são independentes. O algoritmo então se baseia no cálculo apenas dos termos independentes de ordem, ignorando os dependentes. A implementação utiliza 3 alvos de renderização, onde os conteúdos são definidos por:

$$\begin{aligned} X_{RGB} &= \sum C^i_{RGB} C^i_A \\ X_A &= \sum C^i_A \\ Y_A &= \sum \frac{1}{C^i_A} \\ Z_A &= \prod C^i_A \end{aligned}$$

onde X, Y e Z são os valores a serem colocados em determinado *pixel* nos alvos de renderização. X e Y podem ser calculado no mesmo passo de renderização, uma vez que a operação de composição é a soma, enquanto o Z realiza composição por produto. C^i é a cor do i-ésimo fragmento no *pi-*

Figura 14: Passos do algoritmo de soma ponderada. As primeiras imagens correspondem, na ordem, aos termos X_{RGB}, Y_A e Z_A. A última imagem é o resultado, considerando um plano de fundo totalmente preto.

xel correspondente, sendo que são gerados apenas os fragmentos dos objetos translúcidos e o teste de profundidade é habilitado. A composição final é implementada em um *fragment shader* que realiza o seguinte cálculo:

$$I = W_{RGB} + X_{RGB} - (W_{RGB}X_A) + (W_{RGB}Z_A) + (W_{RGB}Y_AZ_A)$$

onde W_{RGB} é a cor do fundo, que é definida em um passo de renderização anterior onde somente os objetos opacos são desenhados.

A Figura 14 ilustra os termos utilizados no algoritmo. A imagem correspondente ao Y_A possui valores a partir de 1, porém na figura ela está normalizada para se adequar ao intervalo $[0,1]$ e com fundo preto aonde não há fragmentos. Esta cena possui esferas com pouca opacidade, uma situação que não é adequada para o algoritmo como pode-se perceber pela região altamente saturada na intersecção com a esfera do lado esquerdo.

4.3 IMPLEMENTAÇÃO

Neste trabalho foram implementadas as 3 técnicas mencionadas, mas dentre elas o *depth peeling* frontal apresentou melhores resultados. A soma ponderada não é adequada para representação de objetos muito translúcidos e também não apresentou resultados satisfatórios com uma grande quantidade de esferas. O *depth peeling* teve uma performance melhor em relação ao *dual depth peeling* porque embora este último gere 2 camadas por passo de renderização, uma delas pode não ser relevante para o resultado final, principalmente quando a cena possui diversos níveis de profundidade.

4.3.1 Encerramento prematuro

Um dos principais pontos a favor do *depth peeling* é que todas as camadas geradas contribuem para o resultado final, mesmo quando encerra-se a geração de camadas antes de esgotar todos os níveis de profundidade da cena. Uma vez que as etapas de composição e de formação de camada são intercaladas, esta última pode consultar o resultado da última composição para determinar se o *pixel* na posição de determinado fragmento já está com um valor de opacidade suficientemente alto, descartando tal fragmento em caso positivo. Nas camadas seguintes todos os fragmentos nessas posições serão consistentemente descartados e possivelmente camadas inteiras deixarão de ser geradas. O critério para determinar se o valor de opacidade é suficientemente alto é descrito pela seguinte condição:

$$1 - D_A \leq \frac{t}{255 \times 2} \tag{4.2}$$

onde t é um inteiro no intervalo $[0, 255]$ que indica o erro máximo tolerável na cor de um pixel. Analisando a equação de composição 4.1, pode-se concluir que o valor no *buffer* não aumentará mais do que $1 - D_A$. Embora o *buffer* de composição possua armazenamento em ponto flutuante 32 bits para evitar acúmulo de erro numérico nas composições intermediárias, a imagem final é de 8 bits, portanto há apenas 256 valores possíveis em cada canal. Com base nesses conceitos, a condição foi formulada para permitir uma tolerância de t nas cores da imagem final. Por exemplo, no caso de um pixel com valor correto de 170 e uma tolerância $t = 4$, então o valor do pixel estaria no intervalo $[168, 172]$. Nos experimentos, um valor de $t = 8$ foi suficiente para ter um aumento de performance sem diferenças perceptuais.

Mesmo deixando de renderizar alguns fragmentos, a cena pode ser bastante complexa e ainda assim muitas camadas serem geradas. Para diminuir o impacto na interatividade da aplicação, pode-se impôr um limite no número total de camadas. Existe uma relação que envolve a opacidade dos objetos, a quantidade de camadas necessárias e o erro no resultado da composição. Para encontrar essa relação, define-se a função $f(n)$ que calcula o resultado final da composição de n fragmentos com mesma opacidade α e que corresponde à equação de composição 4.1:

$$f(n) \quad = \quad \begin{cases} 0 & n = 0, \\ \alpha \times (1 - f(n-1)) + f(n-1) & n > 0 \end{cases}$$

Como estamos considerando que todos os fragmentos tenham o mesmo

valor de opacidade, essa função pode ser simplificada para:

$$f(n) = 1 - (1 - \alpha)^n$$

Substituindo esta função na equação 4.2 no lugar da variável D_A, a equação pode ser resolvida para determinar quais valores de opacidade α permitidos em uma cena com um número máximo de $n = 32$ camadas e erro $t = 8$:

$$\begin{aligned} 1 - f(32) &\leq \frac{8}{255 \times 2} \\ (1-\alpha)^{32} &\leq \frac{4}{255} \\ \alpha &\geq 1 - \left(\frac{4}{255}\right)^{\frac{1}{32}} \end{aligned}$$

O resultado aproximado é $\alpha \geq 0.121767$, ou seja, se na cena inteira houver apenas objetos cuja opacidade respeite essa condição, não haverá erro maior que o especificado na imagem de resultado, mesmo que hajam vários objetos sobrepostos. Para comportar valores menores, deve-se permitir a renderização de mais camadas ou aumentar a tolerância à erros. Note que esta análise considera o pior caso, que é todos os objetos possuindo um mesmo valor de opacidade mínimo e sobrepostos de forma a maximizar a quantidade de camadas.

4.3.2 Seleção de objetos

No *depth peeling*, o tempo de renderização da cena influi drasticamente no tempo total, uma vez que a cena é renderizada múltiplas vezes. Não é necessário, porém, renderizar a cena completa, apenas a parte da cena que tem o potencial de gerar alguma camada. Os objetos que não passaram na primeira vez pelo teste de profundidade descrito na Subseção 3.2.4 certamente não irão passar nos passos de renderização seguinte e portanto jamais gerarão camadas. Outra situação é o caso da renderização de objetos opacos, pois seus fragmentos visíveis serão sempre os mais profundos, ou seja, eles podem ser renderizados apenas uma vez e seus fragmentos extraídos para uma camada de fundo, de forma que as múltiplas renderizações considerem somente os objetos translúcidos.

As técnicas de otimização descritas na Seção 3.2 contribuem para reduzir a quantidade de objetos que passam pelo *pipeline* de renderização, po-

rém ao contrário das outras técnicas, o teste de profundidade com pirâmide precisa ser realizado em todos os passos de renderização. É possível realizar esse teste uma única vez, com auxílio de uma extensão do OpenGL chamada *transform feedback* (WOOLLEY; CARTER, 2006). Esta funcionalidade permite que atributos de primitivas transformadas pelo *vertex shader* e pelo *geometry shader* sejam armazenados, opcionalmente interrompendo o *pipeline* antes da rasterização. O *geometry shader* é responsável por descartar os vértices que não passaram no teste de profundidade com pirâmide, então configura-se o *transform feedback* de forma a capturar o índice desses vértices e gravar em um *buffer*, que será utilizado para definir quais vértices participarão das múltiplas etapas de renderização do algoritmo.

Para que os passos de renderização considerarem apenas os objetos translúcidos, apenas estes devem ser incluídos na formação do *buffer* de índices pelo *transform feedback*. Os *sprites* opacos devem ser renderizados em uma imagem temporária que será tratada como plano de fundo. O mapa de profundidades do plano de fundo deve ser informado ao *shader* do *depth peeling* para que sejam descartados fragmentos atrás do plano de fundo. Terminado o procedimento de geração e composição de camadas, a composição com os *sprites* opacos é realizado. Note que, adicionalmente, é possível renderizar *sprites* translúcidos sobre uma cena opaca qualquer, desde que haja um mapa de profundidades.

4.4 RESULTADOS

A Figura 15 mostra o gráfico de tempo de renderização do conjunto C1 utilizando a versão do algoritmo com e sem a otimização do *transform feedback*. Como este conjunto é bastante esparso, a otimização de *occlusion culling* é pouco efetiva, sendo a pirâmide de profundidade a responsável por reduzir a quantidade de objetos rasterizados e portanto é vantajoso armazenar o resultado do teste de profundidade com pirâmide para as sucessivas renderizações.

As média de tempo de renderização, com otimização de *transform feedback*, podem ser observadas na Tabela 5. De forma geral não houve um bom desempenho para renderização interativa, pois devido à complexidade das cenas o limite de 32 camadas sempre foi atingido, exceto no caso de C1. Fatores que aumentam complexidade da cena é a quantidade de esferas translúcidas e o quão translúcidas elas são.

Na Figura 16 é possível observar as diferentes características de distribuição de translucidez. O conjunto C2 possui muitas esferas translúcidas e com alta translucidez, assim como o C4, sendo as esferas vermelhas e verdes

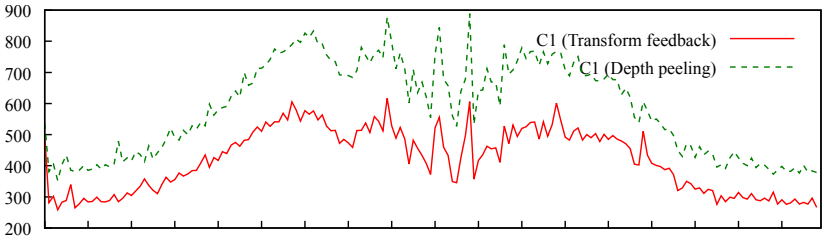

Figura 15: Gráfico de tempo de renderização do conjunto C1 com *depth peeling*.

Tabela 5: Tempo de renderização com *depth peeling*.

	Tempo médio	Média de camadas	Tempo médio por camada
C1	373	27	14
C2	26.612	32	831
C3	1.227	32	38
C4	14.808	32	462

translúcidas e fortemente aglomeradas. Já o conjunto C1 possui uma distribuição mais esparsa e o conjunto C3, embora não seja possível observar, possui esferas translúcidas localizadas nas bordas.

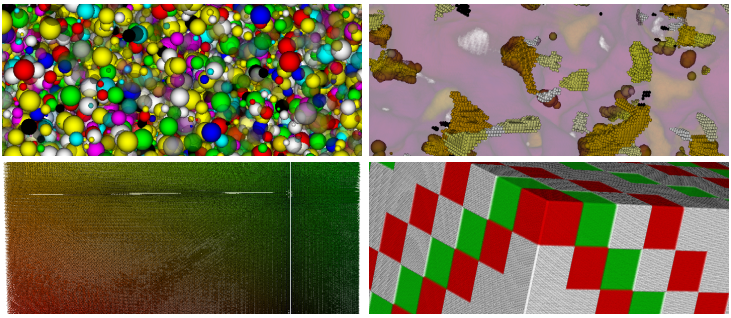

Figura 16: Resultados de *depth peeling* sobre os conjuntos de teste.

5 APRIMORAMENTO DO MODELO DE ILUMINAÇÃO

É difícil identificar o posicionamento de milhares de objetos quando estão todos presentes na cena ao mesmo tempo. No caso de esferas, a posição e o tamanho são as características espaciais que se deseja distinguir. A projeção em perspectiva é uma forma de indicar a profundidade dos elementos da cena, diminuindo o tamanho dos objetos mais distantes, mas não é o suficiente. A Figura 17 mostra um conjunto de esferas renderizado com diferentes modelos de iluminação, ilustrando a influência que estes modelos podem ter na percepção da distribuição dos objetos.

A razão pela qual o modelo de iluminação direta, utilizando apenas *Phong shading*, se torna inadequado em situações como essa é que, por ser um modelo de iluminação local, cada objeto é sombreado de forma independente, sem considerar os outros elementos da cena. Por outro lado, os algoritmos de iluminação global são muito custosos para aplicações interativas. Uma solução simples é o *depth cueing*, que consiste em tornar mais escuras as cores dos objetos mais distantes. Para isso, basta utilizar a profundidade do próprio fragmento para variar a sua cor (MÖLLER; HAINES; HOFFMAN, 2008). Embora seja possível ter uma ideia melhor de como as esferas estão organizadas de maneira geral, ainda é difícil perceber as diferenças de profundidade entre esferas bastante próximas em dada região. Segundo o trabalho de Langer e Bülthoff (1999), o princípio de que "mais escuro significa mais longe" é conhecida há muito tempo, mas o ser humano utiliza mais do que isso para perceber profundidade.

Landis (2002) desenvolveu algumas técnicas para substituir o uso da iluminação global para obter efeitos similares, mas de forma mais rápida. Uma dessas aproximações é o *ambient occlusion*, que gera sombras suaves em superfícies que não estão completamente expostas ao ambiente. Considera-se que a fonte de luz seja uma atmosfera envolvendo a cena emitindo raios de luz em todas as direções. A luminosidade que chega no ponto P de uma superfície, que possui normal N neste ponto, é descrita por uma integral sobre o hemisfério Ω. Este hemisfério é a porção da atmosfera que está na direção de N e que emite raios de luz em direção a P em todas as direções ω.

$$A(P) = \int_{\Omega} V(P,\omega) N \cdot \omega \, \mathrm{d}\omega$$

V é uma função de visibilidade que para avaliar se o ponto P é atingido por um raio na direção ω, considerando os obstáculos que possam existir na cena. Esta função é definida em 1 se não há obstáculos e 0 em caso contrário. Esta é uma aproximação de primeira ordem da equação de renderização

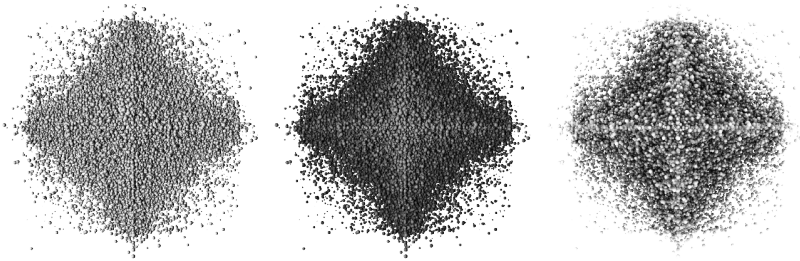

Figura 17: Comparação entre iluminação direta, *depth cueing* e *ambient occlusion*. A imagem da esquerda é difícil de compreender porque não há indicação da profundidade entre os objetos. Com *depth cueing* é possível ter uma noção geral da organização das esferas, enquanto com *ambient occlusion* o posicionamento das mesmas fica mais claro.

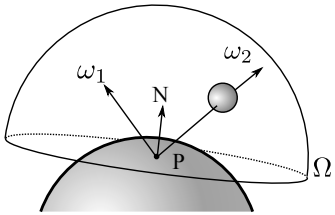

Figura 18: Representação visual dos elementos do *ambient occlusion*. Os vetores ω_1 e ω_2 representam algumas possíveis direções ω contidas no hemisfério Ω, sendo o segundo uma direção tal que $V(P,\omega_2) = 0$.

(KAJIYA, 1986) e também pode ser compreendida no contexto de iluminação por harmônicos esféricos (GREEN, 2003) como uma simplificação que usa apenas o primeiro harmônico. Na prática esta integral deve ser aproximada numericamente porque é difícil definir a função V analiticamente a não ser para casos bastante simples, portanto as direções ω são amostradas para um conjunto finito. A Figura 18 representa os elementos da equação de forma visual.

Uma implementação de *ambient occlusion* em GPU foi apresentada por Sattler et al. (2004), utilizando teste de profundidade e consulta de oclusão para criar uma matriz de visibilidade. As direções são amostradas em uma distribuição uniforme e a iluminação ocorre por vértice, ou seja, a matriz de visibilidade indica se determinado vértice está sendo iluminado por determinada direção. A cena é renderizada em cada direção amostrada e para cada

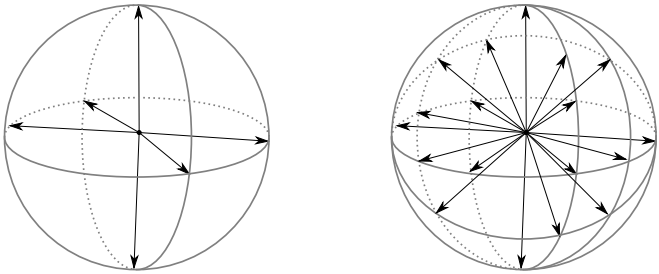

Figura 19: Amostragem de direções em uma atmosfera. À esquerda, os vetores foram definidos com base nos vértices de um octaedro, resultando em 6 direções, enquanto à direita o octaedro foi subdividido, obtendo-se 18 direções.

vértice é feita uma consulta de oclusão, resultando na matriz que é utilizada, então, como função de visibilidade V.

Tarini, Cignoni e Montani (2006) implementaram *ambient occlusion* em GPU para visualização molecular. A atmosfera também é discretizada e para cada direção é gerado um mapa de sombras, ou seja, o mapa de profundidade correspondente à renderização da cena do ponto de vista da fonte de luz. Depois, a cena é renderizada do ponto de vista do observador e o mapa de sombras é utilizado para verificar se determinado *pixel* é iluminado pela fonte de luz considerada. O resultado é acumulado para a textura que será aplicada nas moléculas na visualização da cena.

Neste capítulo será detalhada a implementação de *ambient occlusion* com o objetivo de melhorar a qualidade da visualização, mas ainda manter uma performance interativa para uma grande quantidade de objetos.

5.1 ATMOSFERA

No contexto do *ambient occlusion*, a atmosfera pode ser entendida como uma fonte de luz em área no formato de uma esfera cujo raio é grande o suficiente para envolver toda a cena e a posição do centro é a mesma do ponto considerado para o cálculo da fórmula. Como não há o conceito de distância até a atmosfera, o raio e o centro da esfera não importam e sim que de todas as direções se originem raios de luz, por isso é possível discretizar a atmosfera apenas através de uma amostragem de direções. A Figura 19 ilustra amostragens uniformes de vetores indo em direção a atmosfera utilizando a técnica descrita nesta seção.

As direções devem ser amostradas uniformemente de forma a iluminar a cena por igual, sendo este um problema equivalente a distribuir pontos regularmente na superfície de uma esfera. No trabalho de Sattler et al. (2004), para a aproximação menos detalhada foram utilizados os vértices um octaedro, pois por ser um poliedro regular a distribuição dos vértices é uniforme. Para obter uma aproximação mais detalhada, as arestas podem ser subdivididas sucessivamente e os novos vértices serem projetados para a esfera circunscrita, mantendo uma distribuição razoável de pontos. A vantagem desta técnica é que ao aumentar a quantidade de luzes direcionais entre as renderizações, as direções do nível menos detalhado também estão presentes nos níveis mais detalhados, portanto pode-se reutilizar os seus mapas de sombras. A maior limitação é que a quantidade de vértices fica restrita à $2+4^{s+1}$, onde s é o número de subdivisões.

5.2 MAPA DE SOMBRAS

O mapa de sombras foi um conceito introduzido por Williams (1978) para uma técnica de geração de sombras baseada em imagem. Este mapa é apenas o mapa de profundidades da cena quando renderizada sob o ponto de vista da fonte de luz, ou seja, contém a superfície dos objetos que os raios de luz conseguem atingir. Ao renderizar a cena do ponto de vista do observador, verifica-se se determinado ponto, ao ser transformado para coordenadas de luz, está visível de acordo com o mapa de sombras: em caso positivo, este ponto está sendo iluminado pela fonte de luz. Em caso de cenas estáticas e com fonte de luz inalterada, o mapa de sombras pode ser reutilizado.

O *ambient occlusion* utiliza mapas de sombras para resolver a função V, que é verificar a visibilidade de um ponto em determinada direção. Uma vez que o conjunto de direções permanece o mesmo entre renderizações, os mapas podem ser gerados uma única vez para visualização de cenas estáticas.

Alguns detalhes devem ser observados na criação dos mapas de sombras. Cada direção dará origem a um mapa de sombras e como os mapas são independentes da posição do observador, eles devem estar preparados para abranger toda a cena, caso contrário parte da cena será sombreada incorretamente. Além disso, as otimizações de visibilidade da Seção 3.2 servem apenas para renderizar do ponto de vista do observador e portanto não podem ser aplicadas na renderização dos mapas de sombras.

A preparação para criação do mapa para determinada amostra de direção se inicia com o cálculo da transformação de rotação para definir esta amostra como a direção de observação. Em seguida, para que toda a cena caiba no mapa, obtém-se a caixa delimitadora da cena, com a transformação

de rotação aplicada, e define-se o volume de visualização de forma a comportar a caixa completamente. A projeção é ortogonal, para simular raios de luz paralelos. Quanto melhor "encaixada" a cena ficar, melhor será aproveitada a resolução do mapa de sombras. Embora os passos descritos aqui não garantam o melhor aproveitamento, a aproximação obtida é aceitável.

Embora não seja possível utilizar as otimizações de visibilidade descritas anteriormente, Tarini, Cignoni e Montani (2006) observam que o mapa de sombras se torna mais robusto quando desenhado somente a silhueta da esfera, através de um *sprite* que passe pelo seu centro, o que torna desnecessária a correção de profundidade e portanto o teste de profundidade antecipado pode ser aproveitado. Outra otimização proposta por esses autores é armazenar os mapas de sombras correspondentes a direções opostas em um único mapa, lado a lado, pois dessas duas direções um ponto será sombreado somente por uma delas, dependendo da normal da superfície, reduzindo a quantidade de acessos a textura pela metade quando os mapas forem aplicados no *fragment shader*.

5.3 *DEFERRED SHADING*

O cálculo do *ambient occlusion* pode ser feito uma única vez, armazenando os resultados em textura e descartando os mapas de sombras, ou ser calculado a cada renderização. O espaço de armazenamento necessário para o armazenamento dos resultados é proporcional à quantidade de objetos da cena, enquanto os mapas de sombras ocupam um espaço proporcional à quantidade de amostras de direção. Como o foco deste trabalho é grandes quantidade de esferas, optou-se por não armazenar os resultados e otimizar o cálculo do *ambient occlusion* com a técnica de *deferred shading*.

Deferred shading é uma técnica que adia os cálculos de iluminação, de forma a não desperdiçar processamento em fragmentos que serão sobrescritos, sendo particularmente útil neste contexto de *ambient occlusion* em cenas com alta complexidade de profundidade.

Na primeira etapa, ao invés de calcular o *ambient occlusion*, são geradas texturas com a posição e a normal dos fragmentos, além da cor que foi definida para a esfera correspondente. A segunda etapa trabalha apenas com texturas, acessando os atributos gravados anteriormente e os mapas de sombras. O resultado desta segunda etapa é o termo do *ambient occlusion*, então para obter o resultado final este valor é multiplicado pela cor do fragmento.

Mesmo que haja muita sobrescrita de fragmentos na primeira etapa, a parte mais custosa do algoritmo, que é acessar múltiplas texturas, é feita somente nos *pixels* que estão visíveis. Se houver poucos mapas de sombras, é

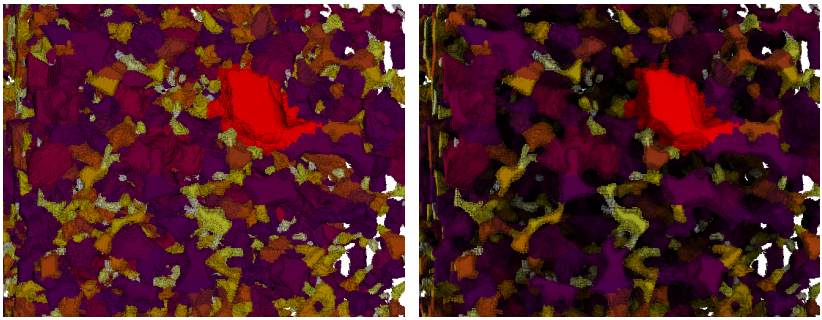

Figura 20: Comparação entre *Phong shading* e *ambient occlusion* com o conjunto C2.

possível realizar esta segunda etapa em apenas um passo, mas como a quantidade de texturas ativas possui um limite, mais passos podem ser necessários. O *deferred shading* se torna ainda mais vantajoso neste caso, pois o processamento da geometria e geração de fragmentos não precisam ser repetidos.

5.4 RESULTADOS

Visualmente os resultados foram significantes, principalmente para os conjuntos C1 (Figura 17) e C2 (Figura 20), que são mais irregulares. Neste último, é possível diferenciar claramente as estruturas mais internas das mais externas, o que não é possível com iluminação por *Phong shading*.

A Tabela 6 compara os tempos médios de renderização, em milissegundos, do *ambient occlusion* com os tempos do *Phong shading*, onde é possível constatar uma perda de performance de 25% (C3) a 235% (C2) dependendo do conjunto de dados. Ainda sim é possível obter interatividade e ser uma escolha mais interessante para visualização, considerando a qualidade visual que a técnica oferece. Nesta tabela os tempos do Ambient Occlusion foram obtidos ao renderizar com 66 mapas de sombras e não é considerado o tempo de pré-processamento para gerar estes mapas.

O tempo de pré-processamento depende diretamente da quantidade de mapas a serem gerados, mas outro fator é o tamanho do conjunto de dados. A Tabela 7 mostra alguns resultados interessantes como alguns conjuntos maiores sendo mais rápidos que algum menor para quantidades menores de mapas. Ocorrem também alguns saltos de diferença ao gerar mais mapas de sombras para os conjuntos maiores quando comparado às diferenças dos menores. A causa provável para a primeira situação é que a quantidade de objetos não está

Tabela 6: Tempos médios de renderização com *Phong shading* e *ambient occlusion*.

Conjunto	Vis.[1]	Phong	AO	Total (Phong)[2]	Total (AO)[2]
C1	39	29	99	68	138
C2	96	186	566	282	662
C3	94	115	169	209	263
C4	180	507	785	687	965

[1] Construção das estruturas para otimização de visibilidade.
[2] Tempo total do frame, com as estruturas e o algoritmo de renderização.

Tabela 7: Tempo de pré-processamento do *ambient occlusion*

Conjunto	Qtd. de mapas de sombras			
	8	18	66	258
C1	112	153	1.107	2.563
C2	80	238	2.941	5.177
C3	184	205	9.747	39.462
C4	977	3.872	19.423	57.676

sendo um gargalo e como neste pré-processamento é utilizado o teste de profundidade antecipado, a distribuição das esferas e as direções de renderização dos mapas causam uma variação maior no tempo de renderização de cada mapa. A segunda situação ocorre pois a quantidade de memória utilizada pelo conjunto de dados e mais os mapas de profundidade pode ultrapassar o limite da memória de vídeo. Quando isto ocorre, o driver de vídeo passa a realizar trocas entre memória de vídeo e de sistema que tornam o acesso aos dados mais lento.

Para se ter noção da diferença visual que a quantidade de mapas de sombras causa, a Figura 21 exibe o conjunto C1 renderizado com 8, 18, 66 e 258 mapas. A pouca quantidade de mapas degrada bastante a qualidade das esferas localizadas mais internamente porque há poucos ângulos de entrada da luz atmosférica. Mesmo as esferas mais externas são prejudicadas por causa da sombra dura que as vizinhas projetam. Embora a qualidade fique cada vez melhor conforme se utiliza mais mapas, o consumo de memória e processamento deve ser levado em conta, portanto considerando as quantidades limitadas de mapas que pode-se escolher gerar, a quantidade de 66 é a que, em geral, possui um resultado visual suficientemente bom para a performance obtida.

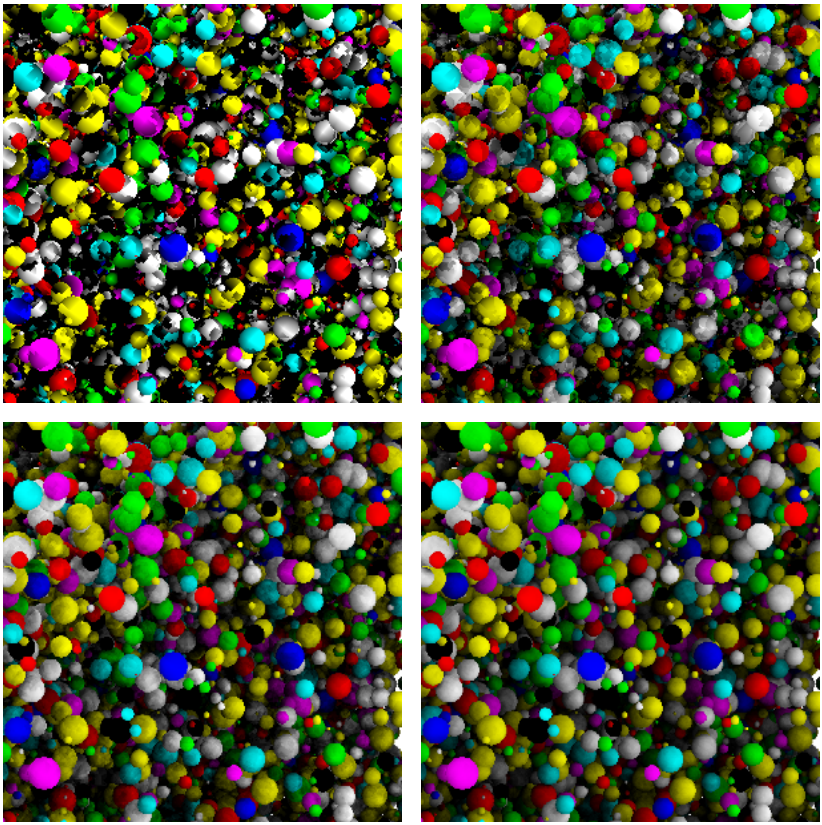

Figura 21: Comparação entre diferentes quantidades de mapas de sombras para o *ambient occlusion*. Da esquerda para a direita, as imagens do topo utilizam 8 e 18 mapas, enquanto as abaixo utilizam 66 e 258.

6 CONCLUSÃO

Com as diferentes técnicas de renderização abordadas e algumas adaptações pôde-se implementar um renderizador com suporte a grandes quantidades de esferas com modos de visualizações que proporcionam diferentes características visuais e de performance. Para todos os casos, as otimizações de renderização opaca com *Phong shading* contribuíram para tornar a exploração dos dados interativa, assim como o *ambient occlusion* permitiu uma visualização mais compreensiva sobre os dados. O modo de renderização com translucência torna a visualização mais flexível, porém deve ser usada de forma moderada ou perde-se a interatividade. Procurou-se definir um conjunto de testes que pudesse representar conjuntos de dados com diferentes características e expor os casos extremos de aglomeração e esparsidade.

Todas as técnicas foram implementadas utilizando o OpenGL 2 e extensões presentes em *hardware* com capacidade para o OpenGL 3.3, que compreende uma geração de GPUs na época de 2006. Não foram investigadas a fundo alternativas para implementação em tecnologias como CUDA ou OpenCL, como o trabalho de Huang et al. (2011), nem funcionalidades presentes somente em GPUs mais modernas que permitem listas encadeadas de *pixels* (THIBIEROZ, 2011). Esses caminhos valem a pena serem explorados em trabalho futuro, uma vez que as placas gráficas com essas novas tecnologias tendem a ficar mais comuns.

As otimizações de visibilidade foram integradas nas outras técnicas, amenizando os problemas relacionados a falta de teste de profundidade antecipado. Contribuiu ao *depth peeling* pois a cena precisa ser renderizada várias vezes sob o mesmo ponto de observação e ao *ambient occlusion* pela grande quantidade de acessos a textura que é feita para sombrear um fragmento, que por sua vez pode acabar não sendo utilizado na imagem resultante. Demonstrou-se também os casos em que essa otimização é mais efetiva, que é na renderização de muitas esferas grandes.

A renderização de esferas translúcidas no contexto de cenas complexas ainda é um desafio, oferecendo performance interativa somente em casos controlados. O principal problema é a dependência de ordem na composição dos fragmentos, que exige a captura de forma ordenada ou então ordenação posterior à geração dos fragmentos. No primeiro caso, que é o abordado pelo *depth peeling*, são necessárias múltiplas renderizações de uma cena que por si só já é custosa, enquanto no último caso é necessária uma grande quantidade de memória. As alternativas que evitam usar os fatores de composição dependentes de ordem não levam a resultados visuais satisfatórios.

O *ambient occlusion* se mostrou muito eficaz para entender como os

objetos estão distribuídos no espaço a um custo razoável na performance, que ainda assim pode ser controlado. As decisões de implementação se basearam na escalabilidade quanto ao número de objetos, ou seja, a quantidade de memória utilizada é proporcional ao nível de qualidade e não à quantidade de esferas. A utilização de *deferred shading* ofereceu um ganho significativo de performance.

Durante o desenvolvimento foi criado um protótipo para escolher a técnica de renderização e poder visualizar as imagens auxiliares usadas para compor os resultados, tais como mapas de profundidade, de sombras e camadas intermediárias. As técnicas podem ser escolhidas durante a execução, sem ser necessário recarregar o conjunto de dados, possibilitando contornar os problemas de performance que podem surgir em determinadas condições de observação. O usuário pode, por exemplo, utilizar o modo mais simples para manipular o conjunto até localizar um ponto de interesse e então ativar o modo de renderização com *depth peeling* ou *ambient occlusion.* Esta característica possibilita futuramente implementar uma troca de modos adaptativa, que permita ao usuário escolher uma técnica mais custosa, mas caso durante a navegação a performance caia para um determinado limite, o modo mais básico seria acionado.

REFERÊNCIAS BIBLIOGRÁFICAS

ASSARSSON, U.; MÖLLER, T. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, v. 5, p. 9–22, 2000.

BAJAJ, C.; DJEU, P. Texmol: Interactive visual exploration of large flexible multi-component molecular complexes. In: *In VIS '04: Proceedings of the conference on Visualization '04*. [S.l.]: IEEE Computer Society Press, 2004. p. 243–250.

BAVOIL, L.; MYERS, K. *Order independent transparency with dual depth peeling*. fev. 2008. NVIDIA OpenGL SDK 10.

BLINN, J. F.; NEWELL, M. E. Texture and reflection in computer generated images. *Commun. ACM*, ACM, New York, NY, USA, v. 19, p. 542–547, out. 1976. ISSN 0001-0782. <http://doi.acm.org/10.1145/360349.360353>.

BLUMBERG, J.; KREIMAN, G. How cortical neurons help us see: visual recognition in the human brain. *The Journal of clinical investigation*, v. 120, n. 9, p. 3054–3063, set. 2010. ISSN 1558-8238. <http://dx.doi.org/10.1172/JCI42161>.

CARD, S. K.; MACKINLAY, J.; SHNEIDERMAN, B. *Readings in Information Visualization: Using Vision to Think (Interactive Technologies)*. 1. ed. [S.l.]: Morgan Kaufmann, 1999. Paperback. ISBN 1558605339.

CATMULL, E. E. *A subdivision algorithm for computer display of curved surfaces.* Tese (Doutorado), 1974. AAI7504786.

CHEN, S. E.; WILLIAMS, L. View interpolation for image synthesis. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1993. (SIGGRAPH '93), p. 279–288. ISBN 0-89791-601-8. <http://doi.acm.org/10.1145/166117.166153>.

CRAIGHEAD, M.; KILGARD, M.; BROWN, P. *ARB_point_sprite*. [S.l.]: OpenGL Architecture Review Board, jul. 2003.

CUNNIFF, R. et al. *ARB_occlusion_query*. [S.l.]: OpenGL Architecture Review Board, jun. 2003.

DEBEVEC, P.; YU, Y.; BOSHOKOV, G. *Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping*. [S.l.], 1998. <http://www.eecs.berkeley.edu/Pubs/TechRpts/1998/5676.html>.

DEBEVEC, P. E.; TAYLOR, C. J.; MALIK, J. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1996. (SIGGRAPH '96), p. 11–20. ISBN 0-89791-746-4. <http://doi.acm.org/10.1145/237170.237191>.

DONG, H. *Micro-CT Imaging and Pore Network Extraction*. Tese (Doutorado) — Imperial College, London, dez. 2007.

EVERITT, C. *Interactive Order-Independent Transparency*. 2001. <http://developer.nvidia.com/content/interactive-order-independent-transparency>.

FOLEY, J. D. et al. *Computer graphics (2nd ed. in C): principles and practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 0-201-84840-6. <http://portal.acm.org/citation.cfm?id=208249>.

GINGOLD, R. A.; MONAGHAN, J. J. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Mon. Not. Roy. Astron. Soc.*, v. 181, p. 375–389, nov. 1977.

GREEN, R. Spherical harmonic lighting: The gritty details. *Archives of the Game Developers Conference*, mar. 2003. <http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf>.

GREENE, N.; KASS, M.; MILLER, G. Hierarchical z-buffer visibility. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1993. (SIGGRAPH '93), p. 231–238. ISBN 0-89791-601-8. <http://doi.acm.org/10.1145/166117.166147>.

GROTTEL, S. et al. Coherent culling and shading for large molecular dynamics visualization. *Computer Graphics Forum*, Blackwell Publishing Ltd, v. 29, n. 3, p. 953–962, 2010. ISSN 1467-8659. <http://dx.doi.org/10.1111/j.1467-8659.2009.01698.x>.

GROTTEL, S.; REINA, G.; ERTL, T. Optimized data transfer for time-dependent, gpu-based glyphs. In: *Visualization Symposium, 2009. PacificVis '09. IEEE Pacific*. [S.l.: s.n.], 2009. p. 65–72.

GUMHOLD, S. Splatting illuminated ellipsoids with depth correction. In: ERTL, T. (Ed.). *Proceeding of Vision, Modelling, and Visualization*. [S.l.]: Aka GmbH, 2003. p. 245–252. ISBN 3-89838-048-3.

HECKBERT, P. Survey of texture mapping. *Computer Graphics and Applications, IEEE*, v. 6, n. 11, p. 56–67, nov. 1986. ISSN 0272-1716.

HUANG, M. et al. A programmable graphics pipeline in cuda for order-independent transparency. In: *GPU Computing Gems*. Emerald edition. [S.l.]: Morgan Kaufmann, 2011. cap. 28. ISBN 978-0123849885.

INÁCIO, R. et al. Interactive simulation and visualization of fluids with surface raycasting. In: *Graphics, Patterns and Images (SIBGRAPI), 2010 23rd SIBGRAPI Conference on*. [S.l.: s.n.], 2010. p. 142–148.

KAJIYA, J. T. The rendering equation. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1986. (SIGGRAPH '86), p. 143–150. ISBN 0-89791-196-2. <http://doi.acm.org/10.1145/15922.15902>.

KITCHENHAM, B. *Procedures for performing systematic reviews*. [S.l.], 2004.

LAMPE, O. et al. Two-level approach to efficient visualization of protein dynamics. *Visualization and Computer Graphics, IEEE Transactions on*, v. 13, n. 6, p. 1616–1623, nov. 2007. ISSN 1077-2626.

LANDIS, H. Production-Ready global illumination. In: *Siggraph Course Notes*. [S.l.: s.n.], 2002. v. 16.

LANGER, M. S.; BÜLTHOFF, H. H. Perception of shape from shading on a cloudy day. *JOURNAL OF THE OPTICAL SOCIETY OF AMERICA A*, v. 11, n. 11, p. 467–478, 1999.

LIU, B.; WEI, L.-Y.; XU, Y.-Q. *Multi-Layer Depth Peeling via Fragment Sort*. [S.l.], jun. 2006. <http://research.microsoft.com/apps/pubs/default.aspx?id=70307>.

LIU, F. et al. Efficient depth peeling via bucket sort. In: *Proceedings of the Conference on High Performance Graphics 2009*. New York, NY, USA: ACM, 2009. (HPG '09), p. 51–57. ISBN 978-1-60558-603-8.

MAMMEN, A. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *Computer Graphics and Applications, IEEE*, v. 9, n. 4, p. 43–55, jul. 1989. ISSN 0272-1716.

MAULE, M. et al. A survey of raster-based transparency techniques. *Computers & Graphics*, v. 35, n. 6, p. 1023–1034, 2011. ISSN 0097-8493. <http://www.sciencedirect.com/science/article/pii/S009784931100135X>.

MESHKIN, H. *Sort-Independent Alpha Blending*. mar. 2007. GDC Session.

MÖLLER, T. A.; HAINES, E.; HOFFMAN, N. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008. 1045 p.

MYERS, K.; BAVOIL, L. Stencil routed a-buffer. In: *ACM SIGGRAPH 2007 sketches*. New York, NY, USA: ACM, 2007. (SIGGRAPH '07). <http://doi.acm.org/10.1145/1278780.1278806>.

NOBREGA, T.; CARVALHO, D. D. B.; WANGENHEIM, A. von. Simplified simulation and visualization of tubular flows with approximate centerline generation. In: *Proceedings of the 22nd IEEE Symposium on Computer-Based Medical Systems (CBMS)*. [S.l.: s.n.], 2009.

PORTER, T.; DUFF, T. Compositing digital images. *SIGGRAPH Comput. Graph.*, ACM, New York, NY, USA, v. 18, p. 253–259, jan. 1984. ISSN 0097-8930. <http://doi.acm.org/10.1145/964965.808606>.

SATTLER, M. et al. Hardware-accelerated ambient occlusion computation. In: GIROD, B.; MAGNOR, M.; SEIDEL, H.-P. (Ed.). *Vision, Modeling, and Visualization 2004*. [S.l.]: Akademische Verlagsgesellschaft Aka GmbH, Berlin, 2004. p. 331–338. ISBN 3-89838-058-0.

SEGAL, M. et al. Fast shadows and lighting effects using texture mapping. In: *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1992. (SIGGRAPH '92), p. 249–252. ISBN 0-89791-479-1. <http://doi.acm.org/10.1145/133994.134071>.

SIGG, C. et al. Gpu-based ray-casting of quadratic surfaces. In: BOTSCH, M. et al. (Ed.). *SPBG*. [S.l.]: Eurographics Association, 2006. p. 59–65. ISBN 3-905673-32-0.

STRASSER, W. *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten*. Tese (Doutorado) — Technical University of Berlin, 1974.

TARINI, M.; CIGNONI, P.; MONTANI, C. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *Visualization and Computer Graphics, IEEE Transactions on*, v. 12, n. 5, p. 1237–1244, set. 2006. ISSN 1077-2626.

THIBIEROZ, N. Order-independent transparency using per-pixel linked lists. In: ENGEL, W. (Ed.). *GPU Pro 2*. [S.l.]: A K Peters, 2011. p. 409–431.

WALLACE, B. A. Merging and transformation of raster images for cartoon animation. *SIGGRAPH Comput. Graph.*, ACM, New York, NY, USA, v. 15, p. 253–262, ago. 1981. ISSN 0097-8930. <http://doi.acm.org/10.1145/965161.806813>.

WILLIAMS, L. Casting curved shadows on curved surfaces. In: *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1978. (SIGGRAPH '78), p. 270–274. <http://doi.acm.org/10.1145/800248.807402>.

WOOLLEY, C.; CARTER, N. *NV_transform_feedback*. [S.l.]: OpenGL Architecture Review Board, 2006.