WALLACE SANTOS LAGES

UMA ARQUITETURA PARALELA PARA A RENDERIZAÇÃO DE MÚLTIPLOS PONTOS DE VISTA

Belo Horizonte
04 de agosto de 2008

WALLACE SANTOS LAGES
Orientador: Dorgival Olavo Guedes Neto

UMA ARQUITETURA PARALELA PARA A RENDERIZAÇÃO DE MÚLTIPLOS PONTOS DE VISTA

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

Belo Horizonte
04 de agosto de 2008

Aos meus pais, João e Maria.

Agradecimentos

Agradeço a todas as pessoas que de uma maneira ou outra tornaram este trabalho possível. Em especial gostaria de mencionar meu orientador Prof. Dorgival Olavo Guedes e o Prof. João Luiz Campos do grupo de pesquisa Tecgraf, da PUC-Rio.

Gostaria de agradecer também aqueles que confiaram em mim quando este trabalho ainda não existia: o Prof. Luiz Chaimowicz, o Prof. Wagner Corradi Barbosa, o Prof. Wagner Corrêa e o Prof. Mario Campos.

Não me esqueço também de meus amigos e colegas do CGGT, com os quais discuti vários assuntos interessantes nos últimos 2 anos: Alessandro Silva, Bruno Evangelista, Daniel Maciel e Carlúcio Cordeiro. Gostaria também de agradecer sinceramente o esforço na revisão das diversas versões desse trabalho feitas pelos meus amigos André Tanus e Helenice Queiroz.

Finalmente, agradeço a minha família, sem a qual nada disso teria sido possível.

Resumo

Apresentamos uma arquitetura paralela para a renderização eficiente de múltiplos pontos de vista em um *cluster* de GPUs. Nossa abordagem utiliza um renderizador de *light field* para reconstruir os pontos de vista desejados a partir de um conjunto de imagens da cena. Para permitir a renderização de cenas dinâmicas, geramos as amostras novamente a cada quadro ao invés de gerar um *buffer* de imagens. Ao mesmo tempo, nossa paralelização permite que cada ponto de vista utilize qualquer uma das amostras geradas, evitando trabalho redundante. A eficiência da solução foi avaliada por meio de modelos matemáticos e validada por meio de medições de *speedup* e eficiência. Concluímos que a solução proposta é escalável e pode suportar renderização a taxas interativas.

Abstract

We present a parallel architecture to efficiently render multiple viewpoints on a cluster of GPUs. Our approach employs a light field renderer to reconstruct the desired views from a set of images of the scene. To allow the rendering of dynamic datasets, we sample the models for every frame, instead of sampling once and storing the resulting images in a buffer. At the same time, our parallelization allows each view to use any of the generated samples, avoiding redundant work. The efficieny of the solution was inspected through mathematical models and validated by measurements of speedup and efficiency. We concluded that the proposed solution is scalable and can support rendering at interactive rates.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A visualização de dados tridimensionais possui aplicações em diversas áreas do conhecimento humano. Por meio dela é possível avaliar projetos de engenharia antes que eles sejam construídos, realizar diagnósticos mais precisos de doenças ou mesmo embarcar nos mundos imaginários dos jogos e filmes.

A maior parte dos sistemas de visualização pressupõe que a cena está sendo observada de apenas um ponto de vista. Apesar de ser suficiente para várias aplicações, algumas situações requerem a geração de diversos pontos de vista da mesma cena. Em um ambiente de realidade virtual, por exemplo, vários usuários interagem em um mesmo cenário mas com pontos de vista distintos.

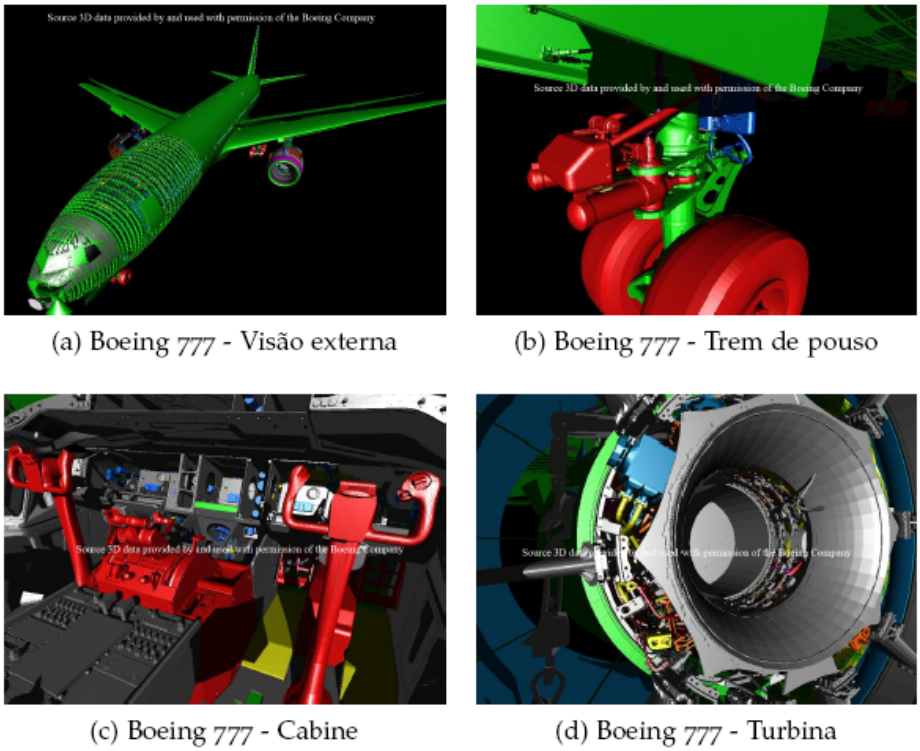
Este texto apresenta uma solução para a geração eficiente de um grande número de pontos de vista. Neste capítulo explicamos porque consideramos o problema importante e os desafios envolvidos na sua solução. Também delineamos a metodologia utilizada e as contribuições deste trabalho. Por fim, explicamos como essa dissertação foi organizada.

1.1 Motivação

As bases de dados tridimensionais aparecem naturalmente em diversas aplicações, como simulações físicas, entretenimento ou projetos de engenharia. A manipulação e visualização desses dados, entretanto, representa um desafio constante. À medida que o poder computacional e de armazenamento aumentam, buscam-se representações gráficas cada vez mais sofisticadas e criam-se bases de dados cada vez maiores.

Quando o tamanho e complexidade dessas bases de dados excede a capacidade de visualização e memória de um computador individual, é necessário utilizar arquiteturas paralelas ou agrupamentos de computadores (*clusters*). Os *clusters* são normalmente utilizados para tarefas de visualização pesadas, uma vez que podem oferecer o mesmo desempenho de uma estação gráfica especializada a um custo inferior. Cada computador possui uma placa gráfica de alto desempenho e é interligado aos demais por meio de uma rede de alta velocidade.

A maior parte dos estudos envolvendo *clusters* dedica-se ao desenvolvimento de sistemas que apresentem um bom desempenho na geração de um único ponto de vista. Em algumas



(a) Boeing 777 - Visão externa (b) Boeing 777 - Trem de pouso

(c) Boeing 777 - Cabine (d) Boeing 777 - Turbina

Figura 1.1: Exemplo de um modelo renderizado utilizando agrupamentos de PCs. O modelo do Boeing Triple-7 contém aproximadamente 350 milhões de triângulos. As imagens foram geradas pela Intrace GmbH (www.openrtrt.com)

situações, entretanto, é desejável ou necessária a geração de múltiplos pontos de vista. Em projetos multidisciplinares, por exemplo, cada usuário pode estar interessado em uma visão particular de uma mesma base de dados. Como exemplo, no projeto de um carro, é possível conceber engenheiros preocupados com a ergonomia e aerodinâmica trabalhando remotamente sobre os dados de um repositório central. Outra aplicação possível são jogos *massive multiplayer*, nos quais muitos jogadores compartilham simultaneamente o mesmo cenário. Em ambos os casos, o sistema deve sintetizar ou renderizar um ponto de vista único para cada um dos usuários conectado ao sistema.

A renderização de múltiplas vistas também pode ser utilizada para criar a percepção tridimensional em *displays* holográficos ou autoestereoscópicos (Annen et al., 2006; Halle, 1998; Hasselgren e Akenine-Möller, 2006; Hübner et al., 2006; Stewart et al., 2004; Jones et al., 2007). Esses dispositivos são capazes de criar imagens 3D para múltiplos usuários, sem a necessidade de óculos ou outros dispositivos especiais. Para isso, é necessário gerar e exibir múltiplos pontos de vista simultaneamente. Cada ponto de vista corresponde a uma posição específica do espaço e é direcionado a um dos olhos do observador (Dodgson, 2005; Matusik e Pfister, 2004).

A não ser que sejam tomadas precauções adicionais, a renderização de múltiplos pontos de vista pode ser uma tarefa pesada até mesmo para um *cluster* gráfico com dezenas de

computadores. Isso acontece porque a base de dados deve ser desenhada novamente para cada ponto de vista, o que torna o sistema inviável a partir de um pequeno número de usuários.

1.2 Objetivo

O objetivo desta dissertação é encontrar uma maneira eficiente para a renderização de múltiplos pontos de vista de um modelo tridimensional em um agrupamento de PCs. Estamos especialmente preocupados com o ganho obtido pela paralelização (*speedup*) e com o comportamento da taxa de quadros à medida que o número de pontos de vista cresce (escalabilidade).

1.3 Desafios

Realizar a síntese de múltiplas vistas é um problema difícil. A escolha do ponto de vista é um dos primeiros passos executados nos algoritmos de renderização tradicionais e influencia todos os passos subsequentes. Desse modo, quando se deseja renderizar mais de um ponto de vista (exemplo: visão estéreo) todo o processo é executado para cada um dos pontos de vista. Os algoritmos mais inteligentes tentam fazer isso explorando a similaridade entre dois pontos de vista para reduzir o esforço computacional. Entretanto, a mudança do ponto de vista pode introduzir diversas mudanças na imagem final que não são facilmente previsíveis. Podem ocorrer *desoclusões*, mudanças de iluminação e mudanças na quantidade de detalhes que podem ser vistos em um objeto.

Quanto maior o número de vistas que se deseja sintetizar, maior deve ser a escalabilidade do sistema; caso contrário, ele se mostraria inadequado para exibir mais que um certo número de pontos de vista.

O primeiro trabalho que explorou a coerência entre pontos de vista foi desenvolvido por Halle (1998). Esse trabalho utilizou a relação entre a posição de um ponto na cena e sua projeção na imagem para obter uma renderização mais eficiente. Recentemente, Hasselgren e Akenine-Möller (2006) propuseram uma solução na qual um novo hardware é utilizado para alterar a ordem de rasterização dos triângulos, obtendo maior coerência no acesso à memória.

Tanto a abordagem de Halle quanto a de Hasselgren e Akenine-Möller visam a aplicação em *displays* estereoscópicos e portanto podem utilizar restrições fortes no posicionamento das câmeras para extrair coerência. Normalmente, a diferença entre as múltiplas vistas decorre apenas da paralaxe horizontal e/ou vertical. Em um sistema genérico é necessário considerar não apenas movimentos horizontais e verticais, mas também mudanças de orientação e na distância até a cena. Finalmente, os *displays* estereoscópicos estão tecnicamente limitados a um número pequeno de pontos de vista (16 ou 32). Desse modo a escalabilidade não é tão crítica quanto em um sistema multi-usuário remoto.

A dificuldade na renderização eficiente de múltiplos pontos de vista reside no fato das arquiteturas tradicionais utilizarem a informação do ponto de vista no início do processo de renderização. Com isso em mente, Stewart et al. (2004) construíram uma arquitetura denominada *PixelView*, onde a informação do ponto de vista só é necessária no último passo

da renderização. Para isso, as diversas vistas são previamente renderizadas e armazenadas em um *buffer* 4D. Assim que o ponto de vista é definido, uma varredura é realizada no *buffer* por meio de um hardware dedicado, gerando a imagem final.

Nossa abordagem para o problema é similar à de Halle e Stewart et al. no sentido em que queremos reutilizar as amostras já computadas sempre que possível. A reconstrução de imagens a partir de outras já computadas, ao invés das primitivas originais, recebe o nome de renderização baseada em imagens. As dificuldades neste tipo de reconstrução envolvem:

1. `Como amostrar a cena original`
   A posição das câmeras de amostragem define a informação que está disponível para a reconstrução: posições que não foram amostradas não podem ser reconstruídas. Idealmente deseja-se que toda a cena seja coberta o mais uniformemente possível.

2. `Como obter as amostras`
   Uma vez definida a posição das câmeras de amostragem, é necessário definir como será feita a amostragem. Informação de profundidade e modelos de câmera mais sofisticados podem melhorar o processo de reconstrução.

3. `Como reconstruir rapidamente novas imagens`
   A reconstrução é uma tarefa computacionalmente intensa. Algoritmos eficientes e a utilização do hardware gráfico podem aumentar o desempenho do sistema.

4. `Como manter a utilização de memória limitada`
   Cada amostra é uma imagem bidimensional. Uma cena complexa pode necessitar de muitas amostras para que ela seja totalmente representada. Além disso, a resolução finita limita a quantidade de detalhes que pode ser capturada por cada uma. Em geral, existe uma relação de compromisso entre a quantidade de memória utilizada e a qualidade da reconstrução.

Uma revisão sobre renderização baseada em imagens será feita no Capítulo 2. São desejáveis, também, os seguintes atributos:

1. utilizar hardware convencional;
2. ser altamente escalável;
3. operar em tempo real;
4. suportar cenas dinâmicas;
5. não ter necessidade de pré-processamento;
6. funcionar a partir de primitivas poligonais;
7. reutilizar o máximo possível computação.

1.4 Formalização do Problema

O problema de renderização paralela de múltiplas vistas pode ser dividido em 3 sub-problemas:

I- Realizar uma amostragem eficiente de cenas sintéticas.

A amostragem da cena influencia diretamente a qualidade da reconstrução. Regiões que não foram amostradas não podem ser reconstruídas adequadamente e a sobre-amostragem pode fazer com que o processo de reconstrução seja menos eficiente. Desse modo, a escolha do número de amostras, posições e da resolução de cada uma delas representa uma relação de compromisso entre o tempo de reconstrução, memória necessária e qualidade final.

II- Realizar uma reconstrução eficiente da cena com qualidade razoável.

A reconstrução é o processo mais crítico, uma vez que ele será realizado para cada um dos pontos de vista. Por essa razão, algoritmos tradicionais de reconstrução têm sido portados para a execução nos processadores das placas de vídeo (*Graphics Processing Units* ou GPUs), cujo poder de processamento têm aumentado mais rapidamente que a Lei de Moore (veja Geer, 2005). Por outro lado, a programação de GPUs confere diversas restrições à implementação, pois elas não são primariamente voltadas para processamento genérico.

III- Encontrar uma paralelização adequada para a execução em *clusters*.

A utilização de *clusters* oferece uma melhor relação custo-benefício se comparada a estações de trabalho especializadas, mas trazem consigo o problema de uma sobrecarga adicional na comunicação entre os nós (Samanta et al., 2000). A excessiva transmissão de dados entre as máquinas do *cluster* pode impactar negativamente a escalabilidade do sistema.

1.5 Linhas Gerais da Metodologia

A metodologia seguida nesta dissertação pode ser resumida nos passos a seguir:

- `Passo 1` - Definição do algoritmo de renderização a ser utilizado para a reconstrução dos pontos de vista.
- `Passo 2` - Implementação e testes do renderizador serial.
- `Passo 3` - Definição da arquitetura paralela e paralelização do algoritmo.
- `Passo 4` - Implementação do sistema paralelo.
- `Passo 5` - Realização de testes de desempenho, avaliando a eficiência e escalabilidade do sistema.
- `Passo 6` - Análise dos resultados.

1.6 Contribuições

Nesta dissertação apresentamos uma arquitetura escalável para a renderização em tempo real de múltiplas vistas. Nossa arquitetura é voltada para *clusters* de PCs com placas gráficas tradicionais e utiliza um renderizador de *light field* para sintetizar as imagens de saída. Um *light field* ou *lumigraph* é uma técnica de permite gerar imagens rapidamente utilizando uma descrição do fluxo de luz na cena (Levoy e Hanrahan, 1996; Gortler et al., 1996).

A arquitetura apresentada aqui é baseada principalmente nas ideias apresentadas por Todt et al. (2007) e Yang et al. (2002). Yang et al. apresentam um sistema que distribui alguns dos passos necessários para a reconstrução de um *light field* entre vários PCs. O objetivo é reduzir os custos de comunicação. Já Todt et al. apresentam um renderizador *light field* de parametrização esférica que utiliza informação de profundidade por *pixel* para melhorar a qualidade da reconstrução. Uma revisão das principais arquiteturas de renderização paralela é feita no Capítulo 3. Uma revisão dos princípios da renderização de *light field* e outras técnicas de renderização baseadas em imagens é feita Capítulo 2.

1.7 Estrutura da Dissertação

Nos dois capítulos seguintes faremos uma revisão dos trabalhos relevantes existentes na literatura. Em seguida apresentaremos a solução proposta e detalhes da implementação. Por fim, apresentamos os experimentos realizados para a validação da arquitetura e uma análise dos resultados obtidos.

Capítulo 2

Renderização

Renderização é o processo de sintetizar uma imagem a partir das primitivas que descrevem a cena. Neste capítulo revemos alguns conceitos básicos relacionados ao processo de renderização. Na primeira parte detalhamos as etapas principais do processo de renderização tradicional utilizando GPUs; em seguida, fazemos uma revisão dos métodos de renderização baseados em imagens e do método *light field* utilizado nesse trabalho.

2.1 O Pipeline de Renderização

As GPUs são processadores gráficos de alto desempenho que foram inicialmente projetados para a renderização de primitivas triangulares. Sua capacidade computacional provém de uma arquitetura altamente paralela, capaz de gerenciar milhares de linhas de execução ao mesmo tempo. Essa característica fez com que ela se tornasse atrativa para outros tipos de computação, como simulações físicas e processamento de dados. Apesar da arquitetura ainda ser baseada no processamento de triângulos, hoje vários estágios podem ser programados, permitindo a construção de algoritmos que operem sobre outros tipos de primitivas (Toledo e Levy, 2004). Um exemplo de um modelo descrito por triângulos pode ser visto na Figura 2.1

Para que os dados possam ser visualizados, as primitivas devem ser transformadas de acordo com a posição do observador, iluminadas e em seguida projetadas no plano que forma a imagem. Esse processo recebe o nome de renderização. O custo computacional de cada etapa varia de acordo com o realismo desejado, com o tipo de primitiva e com o tamanho total do modelo. No caso da renderização de malhas poligonais, por exemplo, é necessário realizar múltiplas operações envolvendo matrizes e variáveis em ponto flutuante.

A transformação que mapeia os pontos do espaço tridimensional para pontos na imagem bidimensional é uma transformação projetiva. Essa transformação também pode ser considerada como uma descrição matemática da câmera utilizado pelo observador da cena. O modelo de câmera mais utilizado em computação gráfica é denominado de *c*âmera *pinhole*. Ela consiste basicamente de uma abertura infinitesimal através do qual a luz proveniente da cena pode atingir o detector (imagem). Na câmera *pinhole*, os pontos ao longo de uma mesma direção são projetados no mesmo ponto 2D, perdendo-se a informação de profundidade. A direção

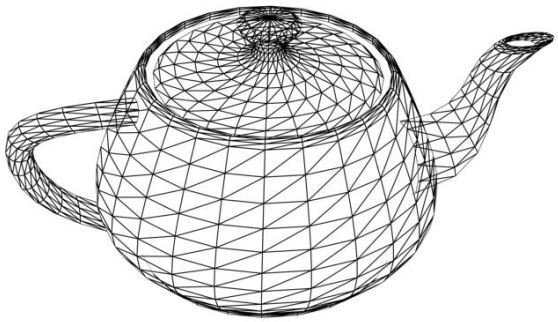


Figura 2.1: A chaleira de Utah - Exemplo de objeto descrito por uma malha de triângulos.

de um ponto na imagem, entretanto, ainda pode ser recuperada da imagem se soubermos os parâmetros da câmera que a gerou (veja Figura 2.2).

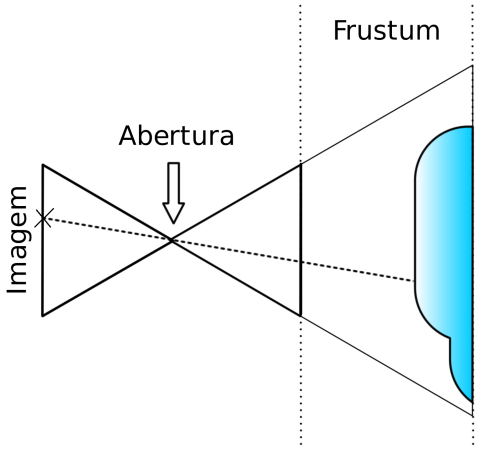


Figura 2.2: Modelo de câmera *pinhole*.

O modelo de câmera *pinhole* simplifica propositalmente o processo de projeção da imagem em troca de uma implementação mais rápida. Na prática, a maior parte dos dispositivos de captura possui aberturas não infinitesimais e utilizam lentes. Efeitos de profundidade de campo, por exemplo, não aparecem nas imagens de câmera *pinhole*. Em algumas aplicações pode ser necessário construir modelos mais reais de câmeras, ou mesmo desenvolver modelos de câmeras com propriedades especiais. Essas câmeras são conhecidas como não-*pinhole*.

Estágios da Renderização

O processo de renderização de malhas triangulares pode ser dividido conceitualmente em três estágios (Akenine-Moller e Haines, 2002): aplicação, geometria e rasterização (Figura 2.3). O objetivo e as principais operações realizadas em cada um deles são discutidas a seguir.

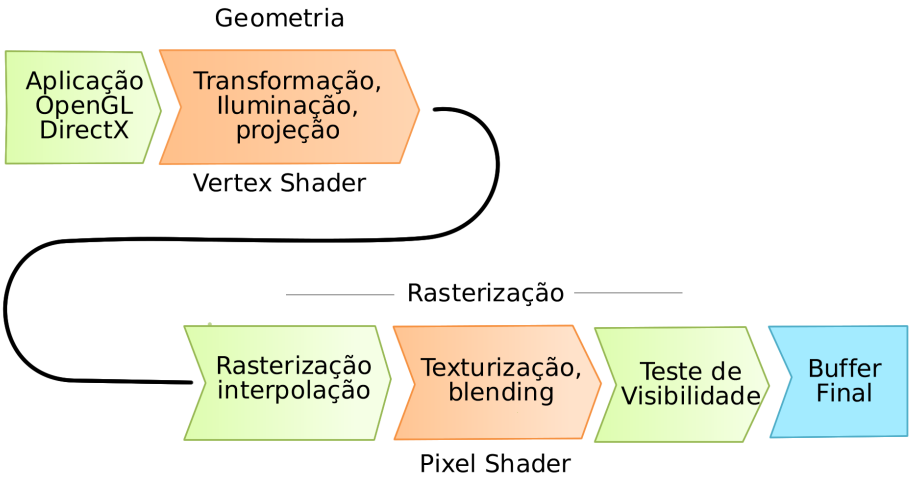


Figura 2.3: Estágios do pipeline de renderização padrão.

1. `Aplicação`
   O estágio da aplicação é responsável por preparar as primitivas gráficas que serão desenhadas nos estágios seguintes. Como as primitivas são geradas na CPU ou recuperadas do disco, esse estágio é normalmente implementado em software. A aplicação tem acesso também a entrada de usuário, rede etc e pode realizar as operações que não podem ser executadas bem em hardware, como alguns tipos de animação, física ou pré-processamento.

2. `Geometria`
   O estágio de processamento de geometria é o responsável pelas operações que são aplicadas a cada polígono ou vértice do modelo. Tanto o estágio de geometria quanto o de rasterização já podem ser implementados totalmente em hardware. O estágio de geometria pode ser subdividido em *Transformação*, *Iluminação*, *Projeção*, *Clipping* e *Mapeamento*.

   A transformação é responsável pela mudança entre o sistema de coordenadas do modelo para o sistema de coordenadas da câmera. Normalmente as transformações são codificadas em matrizes 4x4 e aplicadas em cada vértice da geometria.

   Uma vez transformadas, as primitivas são iluminadas para obter uma melhor sensação de profundidade. Para isso são calculados os valores de intensidade luminosa presentes em cada vértice por meio de modelos que aproximam o comportamento de luzes reais. A iluminação também pode ser calculada por *pixel* na etapa de rasterização.

   Depois de transformadas e iluminadas, as primitivas podem ser projetadas na tela. Isso é feito por meio de uma matriz 4x4 denominada matriz de projeção. As transformações de perspectiva definem um volume da cena que será projetado na tela. O volume é denominado *frustum*.

   Como último passo, as primitivas que não estão inteiramente no volume de visualização, devem ter novos limites construídos (*clipping*) e todos os vértices mapeados para as

coordenadas de exibição na tela (mapeamento).

As operações realizadas no estágio de transformação das GPUs modernas podem ser modificadas por meio de um programa especial denominado *vertex shader*. Os *shaders* executam diretamente na GPU e podem utilizar informações extras passadas por meio de texturas. Em algumas GPUs, a saída do *vertex shader* (triângulos projetados) pode ser processada novamente antes de ser enviada ao estágio seguinte. Esse processamento é feito pelo *geometry shader* e permite ao programador criar novas primitivas.

3. `Rasterização`
   A rasterização consiste no processo de determinar quais pontos na tela cada primitiva projetada ocupa (fragmentos). Quando ocorre uma sobreposição na posição projetada de duas ou mais primitivas, deve ser feito um teste para determinar qual delas está visível. O algoritmo normalmente implementado é o Z-Buffer [1], que verifica antes de desenhar cada *pixel* se algum outro *pixel* mais próximo da câmera já foi desenhado. Em caso afirmativo, a cor do *pixel* é simplesmente descartada.

   As GPUs modernas também permitem executar um código para cada *pixel* produzido pelo rasterizador. Esse programa é conhecido como *pixel shader*. Por meio dele é possível calcular a iluminação que incide sobre cada *pixel*, levando em conta as luzes presentes na cena. É possível também aplicar uma imagem bidimensional sobre a superfície do modelo por meio de uma função de mapeamento. Essa operação, conhecida como mapeamento de textura, é suportada em hardware e muito utilizada para simular a existência de detalhes nas superfícies.

2.2 Métodos de Renderização Baseados em Imagens

Além das malhas poligonais, podemos representar cenas utilizando apenas imagens. As imagens apresentam a vantagem de serem independentes da complexidade da cena ou objeto que representam. Entretanto, enquanto modelos geométricos podem ser renderizados de qualquer ângulo ou distância, as imagens representam apenas o que pode ser visto de um conjunto finito de posições. O conjunto ideal de tudo que pode ser visto da cena recebeu o nome de função plenótica e foi introduzido em 1991 por Edward Adelson. A função plenótica descreve o transporte da luz no ambiente e no tempo a partir do ponto de vista de um observador (Adelson e Bergen, 1991). Em sua formulação completa, a função plenótica toma a forma:

$$P(x, y, z, \theta, \phi, \lambda, t) \tag{2.1}$$

onde

(x, y, z) - a posição no espaço do observador.

(θ, ϕ) - a direção a partir da qual a luz chega ao observador.

[1] A invenção do algoritmo Z-Buffer é normalmente atribuída a Edwin Catmull, Catmull (1974)

λ - o comprimento de onda da luz.

t - o instante da observação.

Teoricamente a função plenótica poderia ser medida posicionando-se um olho imaginário em cada posição possível (x, y, z), registrando-se a intensidade dos raios de luz que passam pelo centro da pupila em todos os ângulos possíveis (θ, ϕ), para todo comprimento de onda λ, em todos os instantes de tempo t. Como a medição da função completa não é prática, é preciso trabalhar com um subconjunto dela. O objetivo das técnicas de renderização baseadas em imagens é obter uma reconstrução contínua da função plenótica a partir das amostras disponíveis (McMillan e Bishop, 1995).

Além da informação fornecida pelos raios de luz capturados, é possível utilizar informação geométrica para delimitar um subconjunto da função plenótica. De fato, as diversas técnicas de renderização baseada em imagens podem ser aproximadamente categorizadas de acordo com a quantidade de informação geométrica utilizada (Lengyel, 1998). Uma imagem, por exemplo, pode ser considerada como uma amostragem 2D da função plenótica em um determinado instante de tempo e em uma posição específica. Uma tentativa de classificação desse tipo está ilustrada na Tabela 2.1. Para ambientes sintéticos, a obtenção de informação geométrica é relativamente simples, entretanto, para imagens reais essa tarefa pode ser extremamente complicada.

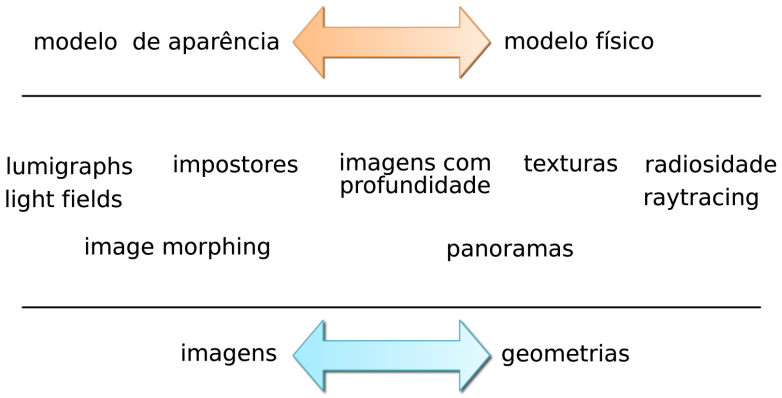


Figura 2.4: Espectro de renderização - Adaptado de Lengyel (1998).

Algumas técnicas representativas de renderização baseada em imagens são descritas a seguir. Cada uma deles captura de uma maneira própria uma parte da função plenótica. Levantamentos mais detalhados podem ser obtidos em Oliveira (2002) e Shum e Kang (2000).

2.2.1 Light fields e Lumigraphs

Devido a restrições de amostragem e armazenamento, quando trabalhamos com uma representação da função plenótica, normalmente desconsideramos o t e aproximamos λ pelo sistema de cores RGB. Desse modo a função 7D passa a ter apenas cinco parâmetros espaciais. Se

dimensão	espaço	nome	ano
7	livre	função plenótica	1991
5	livre	modelagem plenótica	1995
4	caixa 3D	lightfield/lumigraph	1996
3	círculo 2D	mosaicos concêntricos	1999
2	ponto fixo	panorama	1994
1	posição e orientação	foto	1826

Tabela 2.1: Taxonomia das funções plenóticas - Shum e He (1999)

considerarmos que o espaço de observação está livre de oclusores, a função plenótica pode ser representada por uma função 4D conhecida por *lumigraph* ou *light field* (Levoy e Hanrahan, 1996; Shum e Kang, 2000). Isso equivale a considerar apenas o sub-conjunto dos raios que saem de um objeto com uma superfície delimitada (Gortler et al., 1996).

Um *light field* pode ser descrito de diferentes formas, sendo que cada uma gera um tipo diferente de parametrização. A parametrização utilizada por Levoy e Hanrahan descreve os raios que saem da cena por meio da interseção deles com dois dois planos denominados (u,v) e (s,t). O quadrilátero definido pelos dois planos recebe o nome de *l*ightslab, ilustrado na Figura 2.5.

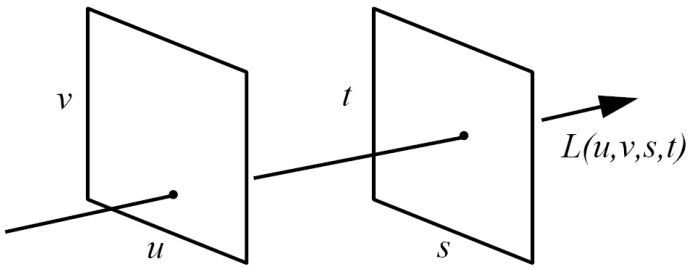


Figura 2.5: Representação da parametrização por 2 planos - Levoy e Hanrahan (1996).

Outras parametrizações utilizadas na literatura são: cilíndrica (McMillan e Bishop, 1995), esféricas (Todt et al., 2007; Ihm et al., 1997), duas esferas, esfera-plano (Camahort et al., 1998) e não estruturadas (Schirmacher et al., 2001; Buehler et al., 2001; Takahashi e Naemura, 2005). As parametrizações esféricas tem a vantagem de permitirem uma amostragem mais uniforme do *light field*, evitando as descontinuidades que aparecem na representação por dois planos. Já os não estruturados visam a aquisição de amostras por meio de câmeras portáteis.

A reconstrução do *light field* pode ser melhorada se for possível estimar a posição da superfície que emitiu cada raio, ou seja, se a informação de profundidade estiver disponível. Gortler et al. (1996) utilizou um malha poligonal para aproximar o objeto sendo representado e realizar uma correção durante a renderização. Algoritmos mais recentes utilizam o valor de profundidade por *pixel* para realizar a correção (Todt et al., 2007; Schirmacher et al., 2001;

Heidrich et al., 1999).

2.2.2 Mosaicos concêntricos e Panoramas

Se o ponto de vista do observador não se move, a função plenótica pode ser simplificada ainda mais. Os panoramas cilíndricos permitem que o observador gire 360 graus na direção de observação horizontal e tenha um movimento limitado na vertical. A parte do panorama sendo vista é deformada para obter uma imagem com a perspectiva correta. Basicamente a deformação é o mapeamento de cada ponto (x,y) no plano de imagem para uma coordenada (u,v) na imagem cilíndrica (veja a Figura 2.7). Os panoramas cilíndricos são fáceis de obter e renderizar, mas não permitem que o observador se mova do ponto central.

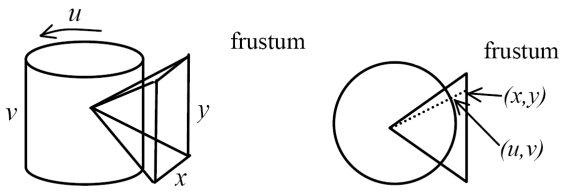


Figura 2.6: Panorama cilíndrico - Esquerda: geometria do frustum Direita: detalhe do warping - Oliveira (2002).

A técnica dos *mosaicos concêntricos* (Shum e He, 1999) é uma extensão dos panoramas, e permite a liberdade de movimento dentro de um círculo bidimensional. Essa técnica parametriza o *light field* utilizando três parâmetros: raio, ângulo de rotação e elevação vertical. A aquisição das imagens é feita movendo-se câmeras de fenda vertical em círculos concêntricos em torno de um ponto central. Uma câmera de fenda vertical é semelhante a uma câmera convencional, mas captura apenas uma linha vertical da imagem.

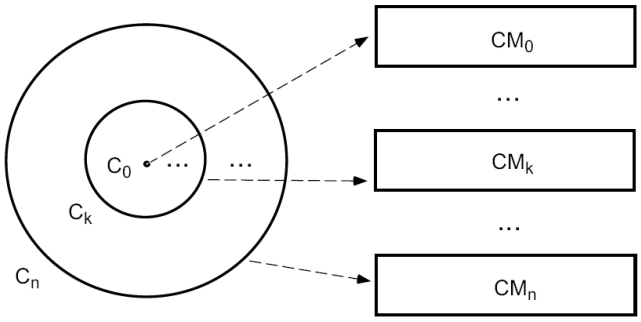


Figura 2.7: Coleção de mosaicos concêntricos - Shum e He (1999).

A reconstrução de um ponto de vista específico é feita recuperando-se cada coluna da nova imagem do mosaico concêntrico desejado. Como a mesma coluna pode ser utilizada em posições diferentes, as imagens podem apresentar distorções verticais. Ainda assim, essa

abordagem consegue capturar e reconstruir de maneira rápida efeitos de desoclusão horizontal e variações de iluminação que dependem do ponto de vista.

2.2.3 Morphing e Interpolação

Morphing (Beier e Neely, 1992) é a técnica que permite reconstruir amostras da função plenótica ao longo de um caminho. Ela utiliza como entrada um par de imagens, que podem ser considerados como início e fim de um caminho percorrido pela câmera ao longo do tempo ou ao longo da cena. Utilizando informação de correspondência entre os *pixels* adicionada manualmente e uma função que combina duas imagens, é possível recriar um ponto intermediário do caminho. A nova imagem é uma combinação linear das imagens originais, aproximada por meio do deslocamento dos *pixels* nas imagens ao longo do tempo.

A interpolação de vistas (Chen e Williams, 1993) permite a recriação de pontos de vista arbitrários baseada na observação de que sequências de imagens espacialmente próximas são bastante coerentes. O método proposto por Chen e Williams utiliza a posição e orientação da câmera e informação de distância das imagens para determinar automaticamente uma correspondência *pixel*-a-*pixel* entre duas ou mais imagens. A informação do fluxo óptico obtida é armazenada de acordo com sua profundidade, sendo possível aproximar efeitos de perspectiva devidos a profundidade.

2.2.4 Imagens com profundidade

Diversas técnicas de renderização baseada em imagens foram construídas utilizando-se imagens aumentadas com profundidade. Normalmente utiliza-se três canais de cor e um quarto canal para representar a distância da câmera (ou plano de imagem) até a superfície responsável pela cor do *pixel* em questão. Desse modo, a informação geométrica da cena pode ser inferida utilizando-se o valor de distância armazenado e o modelo da câmera utilizada. A seguir é feito uma descrição geral de algumas dessas técnicas.

2.2.4.1 3D Image Warping

Image *warping* pode ser utilizado para projetar *pixels* de um ponto de vista para outro, se a profundidade por *pixel* da cena é conhecida (McMillan, 1997). O *warping* pode ser feito a partir das imagens originais para as desejadas, (*direct warping*) ou da imagem desejada para a original (*inverse warping*). A transformação que mapeia uma imagem com profundidade i_s em uma imagem alvo i_t é conhecida como equação de *warp*. Se $\dot{x}$ é um ponto no espaço cuja a projeção no plano da imagem i_s tem coordenadas (u_s, v_s), temos que a projeção em um plano alvo é dada por:

$$x_t \doteq P_t^{-1} P_s x_s + P_t^{-1}(\dot{C}_s - \dot{C}_t)\delta_s(u_s, v_s) \tag{2.2}$$

A equação do *warp* mostra que é possível determinar a posição na imagem final aplicando uma transformação de perspectiva planar $P_t^{-1}P_s$ e depois um deslocamento proporcional a

"disparidade generalizada"δ_s na direção do epipolo $(\dot{C}_s - \dot{C}_t)$ da imagem alvo (Sawhney, 1994) [2]

O problema normalmente associado ao 3D *warping* é a geração de buracos na imagem gerada, que podem aparecer devido a diferenças de amostragem (resolução) e desoclusão de partes da cena (áreas da imagem não vistas anteriormente se tornam visíveis devido a mudança de posição da câmera).

As alternativas para a solução podem envolver interpolações ou duplicação do valor dos *pixels* próximos, a utilização de várias imagens com pontos de vista diferentes (Mark et al., 1997) e imagens especiais contendo várias camadas ou produzidas por câmeras não *pinhole* (Mei et al., 2005; Popescu e Aliaga, 2006; Rosen e Popescu, 2008)

2.2.4.2 Relief Textures

Relief texture mapping é uma maneira de se adicionar detalhes 3D e paralaxe em superfícies por meio do mapeamento de texturas com profundidade (Oliveira e Bishop, 1999; Oliveira et al., 2000). *Relief textures* normalmente são projetadas paralelamente, de modo a simplificar o *warp*. A ideia do *relief mapping* é fatorar a equação de 3D *warping* (2.2) de modo que ele seja feito por meio de um pré-*warp* seguido de uma aplicação de textura normal. Isso permite que o algoritmo seja utilizado nas aplicações que utilizam OpenGL e o hardware gráfico de maneira eficiente.

Para o passo de pré-*warp* é necessário encontrar um mapeamento p que quando composto com uma operação de mapeamento de textura m resulta em um resultado consistente com a equação de *warp* original.

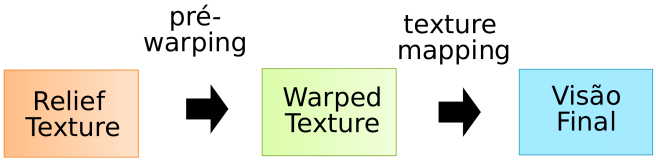


Figura 2.8: Relief Texture Mapping - Oliveira et al. (2000).

A abordagem original do *Relief texturing* pode ser utilizada para renderizar objetos 3D. Uma representação consiste em um *bounding box* ao qual é aplicado uma textura *relief*. Novos pontos de vista podem ser obtidos alterando-se o pré-*warp* de acordo com a posição da nova câmera.

Posteriormente Policarpo et al. (2005) apresentou uma nova técnica para apresentar *relief textures* em superfícies poligonais por meio da GPU. Ao invés da fatoração apresentada anteriormente, os mapas de altura são renderizados por meio de um algoritmo de emissão de raios (*raycasting*) no espaço da tangente. O ponto de interseção com o mapa de alturas é encontrado por meio de uma busca linear (Figura 2.10) seguido de um refinamento binário

[2] Projeção do centro de projeção de uma câmera no plano de imagem da outra câmera

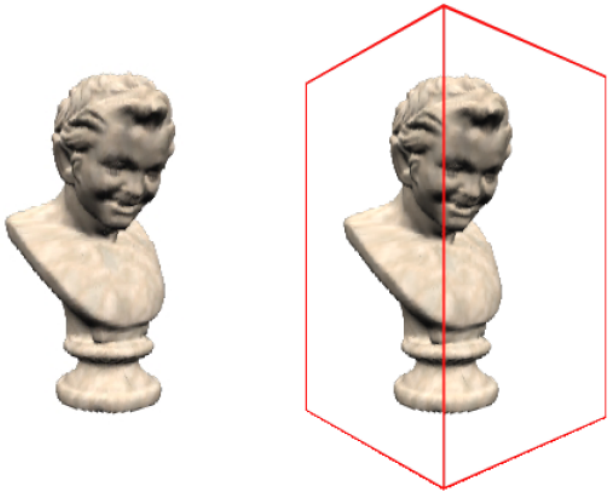


Figura 2.9: Imagem de uma estátua (esquerda) renderizada aplicando-se uma textura relief em 2 quads - Oliveira et al. (2000).

(Figura 2.11). Posteriormente o algoritmo foi refinado para dar suporte a modelos não *height-field* (Policarpo e Oliveira, 2006) e suportar ângulos de visualização extremos (Risser, 2007; Andujar et al., 2007).

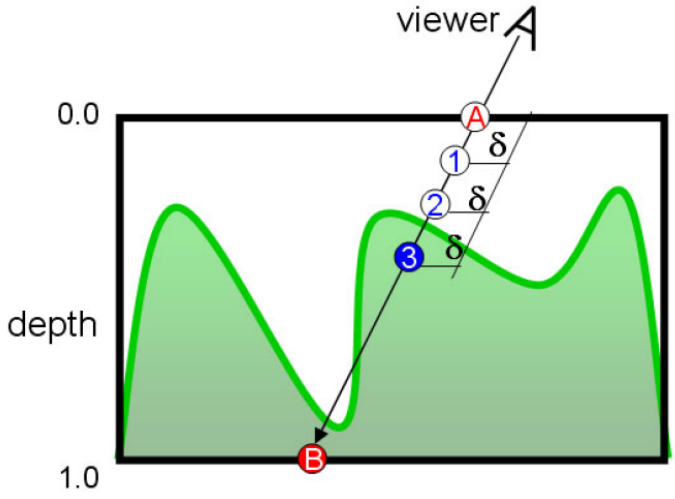


Figura 2.10: Busca linear de A até B pelo primeiro ponto dentro da superfície - Policarpo et al. (2005).

Mesmo sendo realizadas com *pixel shaders* nas GPUs, a renderização de *relief textures* ainda é cara (Nehab et al., 2006). Nehab et al. sugeriram, assim, a utilização de caches para acelerar o processo de reprojeção. A ideia do cache de reprojeção de tempo real é armazenar informação de superfície no espaço de imagem e assim utilizar a coerência entre frames consecutivos. Possivelmente um cache pode ser útil para as operações de *raycasting* do *relief mapping*.

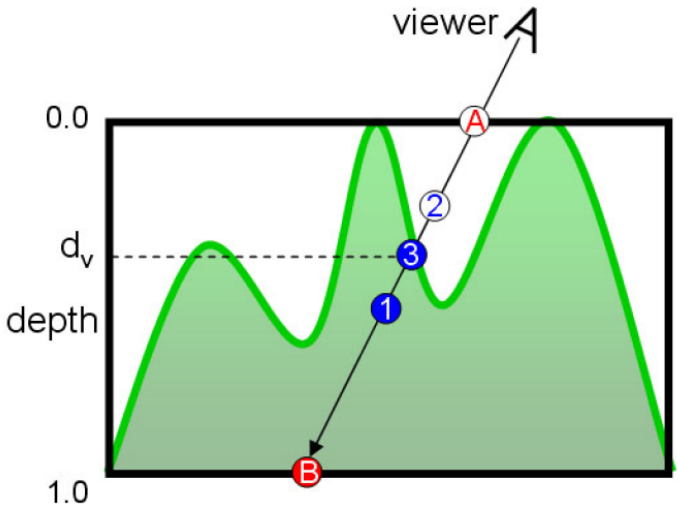


Figura 2.11: Interseção com o mapa de alturas utilizando busca binária - Policarpo et al. (2005).

2.2.4.3 Layered Depth Images

Shade et al. (1998) descrevem uma primitiva denominada *Layered Depth Image* ou LDI. Um LDI é uma vista da cena de um ponto de vista apenas, mas com múltiplos *pixels* ao longo de cada raio. Assim, os problemas de desoclusão podem ser ser razoavelmente evitados. Além disso, o sistema de coordenadas único facilita algumas operações e o tamanho da representação, depende diretamente da complexidade de profundidade da cena.

Para a representação da superfície de um objeto, seis LDIs podem ser combinadas, formando um *Image Object* (Neto e Bishop, 1999) Para isso várias vistas devem ser capturadas e reamostradas a partir de um centro de projeção comum. As seis faces são mapeadas em um paralelepípedo, de forma que seja possível utilizar o *warp* levando em consideração a oclusão (*occlusion compatible order*) e de forma eficiente.

2.3 Amostragem Plenótica

O número mínimo de amostras necessárias para reconstruir um sinal é um problema clássico em processamento de sinais e computação gráfica. Quando consideramos a diversidade das técnicas apresentadas na ultima seção, podemos ponderar sobre a relação existente na qualidade da reconstrução e da quantidade disponível de imagens e de informação geométrica.

A análise da amostragem em renderização baseada em imagens é um problema difícil porque ela envolve uma relação complexa entre três elementos: a profundidade e informação de textura presente na cena, o número de amostras disponíveis e a resolução da renderização (Chai et al., 2000; Lin e Shum, 2000, 2004). Chai et al. (2000) mostraram que é possível inferir a relação entre a complexidade da cena (informação geométrica e de textura) o número de amostras de imagem e a resolução da imagem de saída.

Sabe-se que a informação de profundidade, além de melhorar a qualidade de renderização,

também afeta a taxa de amostragem. Uma vez que a informação por *pixel* pode ser obtida sem custos adicionais em imagens sintéticas, ela tem sido bastante utilizada em trabalhos recentes (Schirmacher et al., 2001; Todt et al., 2007).

Outro ponto importante, é utilizar uma boa estrutura para a amostragem. Para objetos é comum utilizar-se uma parametrização esférica (Ihm et al., 1997; Todt et al., 2007). Para cenas genéricas, entretanto, a resposta não é tão clara. A abordagem mais simples é amostrar a cena utilizando-se um *grid* regular (Shade et al., 1996). Outra possibilidade é gerar amostras adaptativamente como em Jeschke et al. (2005).

Ao invés de considerar o problema da amostragem de maneira geral, Schirmacher et al. (1999) considera o problema de decidir quais novas amostras devem ser incorporadas ao *light field* para melhorar a qualidade da renderização. O ganho na qualidade da reconstrução é estimado avaliando-se o erro obtido ao renderizar o ponto de vista a ser incluído com as amostras já adquiridas. Aqueles que puderem ser bem reconstruidos não contribuirão para o *light field* e portanto podem ser descartados.

2.4 Renderização de Múltiplos Pontos de Vista

Renderizar múltiplas vistas utilizando o *pipeline* gráfico padrão é um problema difícil. A arquitetura atual de rasterização requer que a decisão do ponto de vista seja feita logo no início, o que torna difícil explorar a coerência entre pontos de vistas diferentes. Por essa razão, Halle (1998) sugeriu que os pontos de vista fossem renderizados a partir de um volume gerado por várias imagens epipolares.

Mais tarde, Hübner et al. (2006) desenvolveram uma técnica para fazer o *splatting* em vários pontos de vista simultaneamente. *Splatting* (Levoy e Whitted, 1985; Pfister et al., 2000) é uma técnica de renderização baseada em pontos na qual pequenos discos são desenhados sobre os diversos pontos que definem a superfície dos objetos. O sistema desenvolvido utiliza a GPU para projetar cada *splat* apenas uma vez para todas as vistas.

Outras tentativas de realizar a renderização eficiente de múltiplos pontos de vista envolveram a criação de novo hardware. Hasselgren e Akenine-Möller (2006) propuseram uma arquitetura capaz de renderizar cada triângulo simultaneamente em múltiplos pontos de vista, melhorando a coerência do acesso às texturas. Stewart et al. desenvolveram o sistema *Pixel-View*, um protótipo de hardware que empregava um *frame-buffer* 4D para permitir a escolha do ponto de vista no momento em que se lê a imagem para o dispositivo de saída. Apesar das ideias citadas ajudarem a explorar novos algoritmos, elas têm aplicabilidade limitada por não utilizarem o hardware atualmente disponível.

De certa forma, os *buffers* utilizados por Halle (1998) e por Hasselgren e Akenine-Möller (2006) podem ser considerados *light fields*. Yang et al. (2002) também utilizou o mesmo conceito para gerar novos pontos de vista para uma matriz de câmeras. Entretanto, em nosso trabalho, ao invés de construir e amostrar todo o *light field* para cada nova vista, estamos interessados em utilizá-lo como uma maneira de compartilhar os raios entre os pontos de vista e assim evitar esforço computacional desnecessário.

Mesmo utilizando o compartilhamento dos raios, o custo de renderizar múltiplos pontos de vistas interativamente é ainda maior que o poder computacional disponível em uma placa aceleradora gráfica típica. Por esse motivo, propomos uma arquitetura paralela escalável. Annen et al. (2006) descrevem um sistema distribuído para renderizar múltiplos pontos de vista para *displays* de paralaxe. Entretanto, eles não exploram nenhum tipo de coerência entre os pontos de vista e se concentram no problema de escalabilidade e balanceamento de carga. Yang et al. (2002) desenvolveram um sistema que distribui alguns dos passos necessários para reconstruir um ponto de vista único, atingindo custos de transmissão escaláveis. De maneira similar ao nosso trabalho, eles não utilizam um buffer de raios, então a técnica pode ser aplicada a cenas dinâmicas.

2.5 Visualização Remota

Para que diversos usuários possam utilizar o mesmo *cluster* simultaneamente, é que preciso que esses usuários se conectem remotamente aos nós de renderização. Nessa seção iremos rever algumas soluções adotadas na literatura para a visualização remota.

A infraestrutura de visualização *ParaView* (Cedilnik et al., 2006) mantém uma separação entre os clientes, os servidores de dados e os servidores de visualização. A geometria pode ser gerada em um conjunto de máquinas dedicadas e então transferidas ao conjunto responsável por realizar a renderização. O resultado da renderização, por sua vez, pode ser direcionado para um *display* ou para um cliente remoto.

Para permitir um menor tempo de compressão e descompressão da sequência de quadros, o *Paraview* optou por utilizar o algoritmo denominado SQUIRT (*Sequential Unified Image Run Transfer*) que é um algoritmo de codificação *run-length*. As operações são realizadas em triplas RGB para acelerar o processamento. Como os algoritmos run-length não possuem uma boa taxa de compressão em imagens sintéticas, o SQUIRT ignora alguns bits selecionados em cada cor durante a comparação.

Jeschke et al. (2002) apresentaram um sistema para a visualização remota de cenas estáticas baseado em camadas cúbicas de *impostores*. Impostores são imagens geradas em tempo execução que têm como objetivo representar de forma mais simples um modelo ou parte do cenário. Relief textures e LDIs podem ser utilizados como impostores de qualidade superior. Jeschke et al. geram os impostores durante uma etapa de pré processamento para cada célula do cenário considerando os erros de paralaxe e falhas entre os texels de texturas consecutivas. Os objetos mais próximos são mantidos como geometria, enquanto os outros são representados nas camadas de impostores. Para diminuir o espaço ocupado pelos impostores, os *pixels* transparentes em cada camada são removidos partindo-se a textura original em um conjunto justaposto de texturas menores. Durante a execução, os impostores das células vizinhas no *grid* são carregados em segundo plano.

Outro sistema que utiliza imagens para aceleração é o *Massive Model Rendering System* (MMR) (Aliaga et al., 1999a). O sistema funciona renderizando objetos longe do ponto de vista utilizando técnicas de imagem e objetos próximos utilizando geometria com nível de

detalhe e teste de visibilidade. O cenário distante é renderizado com *depth-meshes* (Sillion et al., 1997), que consiste em malhas construídas para aproximar a profundidade da cena vista pela câmera.

Uma abordagem recente para o problema de visualização remota é descrita por Callahan et al. (2006). Callahan descreve um método para a renderização progressiva de volumes não estruturados. A ideia é enviar ao cliente de maneira progressiva os triângulos que estão visíveis. O cliente então renderiza os triângulos diretamente, armazenando os dados já desenhados como imagens.

2.6 Discussão

Apresentamos nesse capítulo uma visão geral dos métodos de renderização mais relevantes para este trabalho. Primeiro, discutimos a sequência de renderização tradicional, onde as imagens são geradas a partir de descrições geométricas da cena. Em seguida, introduzimos o conceito de função plenótica e mostramos como podemos utilizar imagens para reconstruir a cena a partir de outros pontos de vista. Em especial, descrevemos a parametrização de dois planos que será utilizada como base para o nosso renderizador. Nas últimas duas seções fizemos uma revisão dos sistemas de renderização de múltiplas vistas existentes na literatura e dos sistemas de renderização remota.

Capítulo 3

Renderização Paralela

O paralelismo já foi aplicado de diversas maneiras para acelerar a renderização. Uma maneira é dividir as várias etapas do processo de renderização discutido no capítulo anterior em estágios, tornando-os paralelos na forma de um *pipeline*. Outra opção é dividir as primitivas a serem renderizadas entre os processadores disponíveis; assim o paralelismo não fica limitado ao número de estágios do processo. Por fim, é possível explorar a dimensão temporal, onde cada unidade fica responsável por determinados quadros da sequência final. Neste capítulo revisamos os trabalhos relevantes relacionados a renderização paralela; bem como as técnicas, os problemas envolvidos e como eles se relacionam com essa dissertação.

3.1 Modelos de Paralelização da Renderização

Molnar et al. (1994) classificaram as arquiteturas de renderização de acordo com o momento no qual a ordenação de visibilidade ocorre. A arquitetura *sort-first* (Popescu et al., 2000) recebe esse nome porque a determinação de visibilidade é feita no primeiro estágio. Essa arquitetura divide a imagem final a ser renderizada em retângulos justapostos. Cada máquina se encarrega de renderizar o conteúdo de um ou mais retângulos e enviar para a máquina que compõe a imagem final. Essa última máquina apenas reúne lado a lado os retângulos renderizados para formar a imagem final. Um ponto forte dessa arquitetura é a baixa comunicação entre as máquinas. Além disso, caso o dispositivo de saída seja um mosaico formado por várias placas gráficas, a etapa de reunir os diversos retângulos pode ser ser inteiramente evitada.

Na arquitetura *sort-first*, como cada processador é responsável por todo o processamento gráfico de um trecho da imagem, o balanceamento depende da distribuição das primitivas na imagem final. Além disso, uma primitiva deve ser inteiramente desenhada por todos os retângulos nos quais ela aparece. Quando o número de processadores aumenta, os retângulos ficam menores o que faz com que o trabalho de renderização dessas primitivas seja repetido, limitando a escalabilidade. Além disso, cada máquina deve possuir todos os dados necessários para construir a imagem do seu retângulo.

Outra abordagem possível é dividir os objetos entre os processadores participantes. Cada objeto é renderizado completamente, produzindo a imagem já rasterizada e a informação de

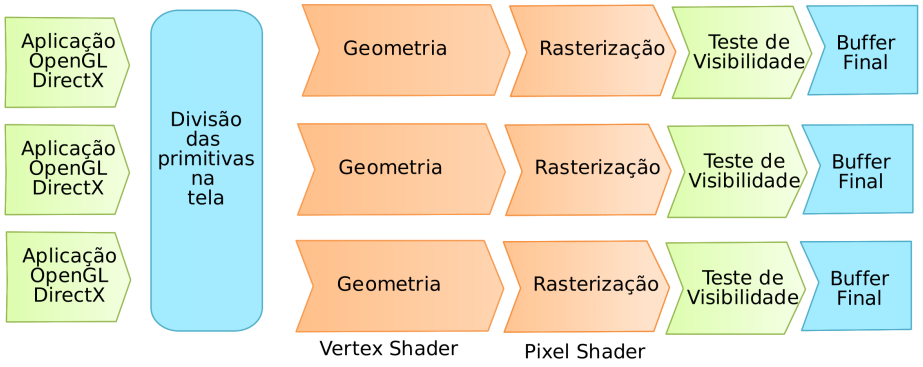


Figura 3.1: A arquitetura *sort-first* redistribui as primitivas no espaço de tela durante o processamento de geometria.

profundidade para cada pixel. O processador de composição determina então a visibilidade comparando a profundidade de todos os *pixels* com a mesma coordenada para obter a imagem final. Essa arquitetura é a denominada *sort-last* (Eilemann e Pajarola, 2007; Moreland et al., 2001).

Como cada objeto é desenhado apenas uma vez, essa arquitetura apresenta uma boa escalabilidade no processamento das primitivas. Entretanto, a necessidade de enviar ao processador de composição a informação de cor e profundidade para cada pixel da imagem pode ser um gargalo a se considerar. Além disso, o estágio de composição é obrigatório para qualquer dispositivo de saída.

Para diminuir o custo de um passo de composição serial, Roth e Reiners (2006) propuseram um algoritmo para paralelizar a composição entre as máquinas disponíveis. Após renderizar sua imagem, cada nó troca uma sub-imagem com outro nó. Em seguida, cada nó compõe sua sub-imagem e envia o resultado para o outro nó. Dessa maneira, todas as máquinas trabalham ativamente no processo de composição.

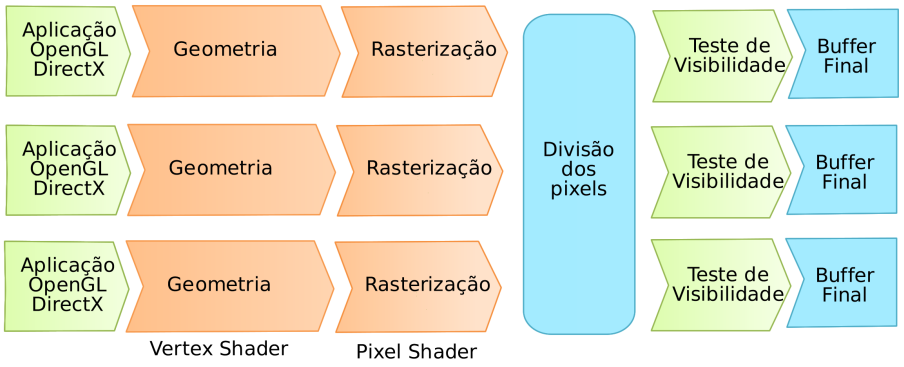


Figura 3.2: A arquitetura *sort-last* redistribui os pixels durante o estágio de rasterização.

Por fim, a arquitetura *sort-middle* é uma combinação entre a *sort-last* e *sort-first*. Nela, as primitivas primeiro são distribuídas entre os processadores para executar a transformação

para o espaço de imagem (como no *sort-last*). Em seguida, a rasterização é distribuída no espaço de tela como no *sort-first*.

A arquitetura *sort-middle* requer uma redistribuição arbitrária de primitivas entre os processadores de transformação e rasterização, o que requer uma grande largura de banda. Além disso, nas GPUs tradicionais esses estágios já estão integrados no mesmo processador. Como no *sort-first*, essa arquitetura também é sensível à distribuição das primitivas na tela.

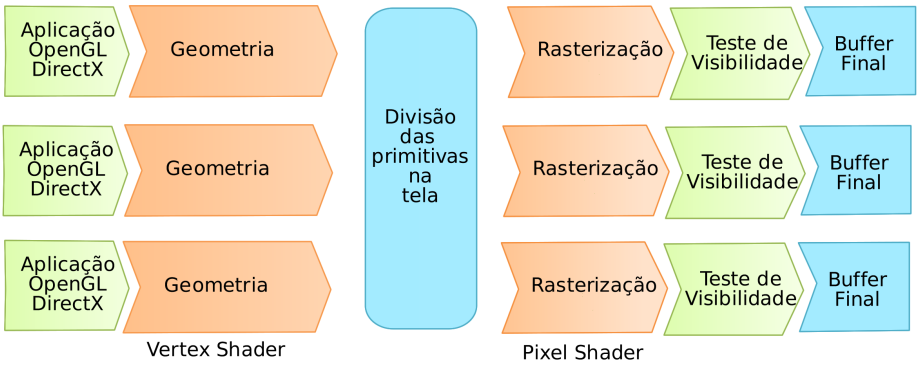


Figura 3.3: A arquitetura *sort-middle* redistribui as primitivas no espaço de tela entre os estágios de geometria e rasterização.

Samanta et al. (2000) apresentam uma arquitetura híbrida com o objetivo de combinar as vantagens das arquiteturas *sort-first* e *sort-last* realizando um particionamento dinâmico da cena e da imagem final. Isso é feito agrupando-se as primitivas para renderização por cada processador baseado na sobreposição dos seus volumes no espaço da tela.

Crockett (1995) apresenta uma exposição mais detalhada sobre os conceitos de paralelismo aplicados a renderização, bem como uma discussão sobre as questões principais de projeto e implementação desses sistemas.

3.2 Sistemas Paralelos Baseados em Imagens

Alguns dos sistemas de renderização paralela propuseram a utilização de técnicas de renderização baseada em imagens para acelerar e/ou simplificar a exibição dos modelos. Sloan e Hansen (1999) apresentam três técnicas para a reconstrução paralela de *lumigraphs*. Entretanto, eles utilizam um computador de memória compartilhada ccNUMA[1]. Um trabalho mais recente desenvolvido por Strasser et al. (2006) utiliza um cache de imagens para desacoplar a renderização da visualização. O sistema desenvolvido, denominado MLIC, distribui o cálculo de renderização entre os múltiplos computadores. Eles decompõem o espaço em volta do ponto de vista em seis pirâmides. As imagens são distribuídas em posições fixas de acordo com o ponto de vista e podem ser refinadas utilizando uma kd-tree. A visualização do banco de dados de imagens implica em visitar todos os poliedros e percorrer as kd-trees associadas de cima para baixo e de trás para frente.

[1]Cache Coherent Non-Uniform Memory Access

Muitos sistemas híbridos utilizando geometria e imagens também foram propostos para a renderização paralela de modelos gigantes. A ideia básica é renderizar os objetos distantes do ponto de vista utilizando uma técnica rápida baseada em imagens. Entre eles podemos citar o trabalho de Wilson e Manocha (2003), Aliaga et al. (1999b) e Debevec et al. (1996).

3.3 Métricas de Desempenho

Existem várias maneiras de se medir o desempenho de um sistema paralelo. As duas medidas mais utilizadas são *Speedup* e *Eficiência* (Jain, 1991). O *speedup* é definido como a razão entre o tempo gasto na execução de uma tarefa em um processador e o tempo gasto quando P processadores são utilizados:

$$S(P) = \frac{T_{serial}}{T_{paralelo}} \tag{3.1}$$

O *speedup* pode ser avaliado de diversas maneiras. Se compararmos o melhor algoritmo sequencial com o algoritmo paralelo sendo estudado, temos o *speedup* absoluto. O *speedup* absoluto pode considerar os recursos da máquina ou não. No primeiro caso, o *speedup* é definido como a razão entre o tempo gasto pelo melhor algoritmo serial em um processador sobre o tempo gasto pelo algoritmo paralelo em P processadores. No caso de independência de máquina, o *speedup* absoluto é definido como a razão do melhor algoritmo sequencial na máquina sequencial mais rápida sobre o tempo do algoritmo paralelo na máquina paralela (Sun e Ni, 1990).

Outra definição é o *speedup* relativo, que considera o paralelismo intrínseco do algoritmo. Ele é definido como a razão do tempo gasto pelo algoritmo paralelo em 1 processador sobre o tempo gasto pelo algoritmo paralelo em N processadores. O *speedup* relativo permite avaliar como o desempenho do algoritmo varia com o número de processadores, uma vez que é comparado consigo mesmo (Sun e Ni, 1990).

Duas formulações conhecidas foram baseadas no *speedup* relativo: A lei de Amdahl (Amdahl, 1967) e o *speedup* de Gustafson (Gustafson, 1988). A lei de Amdahl descreve o *speedup* obtido ao se resolver um problema de tamanho fixo sobre quando se aumenta o número de processadores. Nesse cenário, se a fração serial de um programa é s e a fração que pode ser executada em paralelo p, tal que $s + p = 1$, temos:

$$S(P) = \frac{s+p}{s+p/P} = \frac{1}{s+p/P} \tag{3.2}$$

O *speedup* máximo obtido por uma implementação paralela é portanto $1/s$.

Gustafson observou, entretanto, que a fração paralelizável p não é independente de P porque normalmente aumentamos o tamanho do problema quando dispomos de mais processadores. Como a maior parte dos tempos seriais não aumentam com o tamanho do problema a fração paralelizável tende a crescer.

O *speedup* calculado com base nesse raciocínio permite retirar o limite da Amhdal, mas pode ser pouco realístico na prática. Ao aumentarmos o número de variáveis, o tamanho

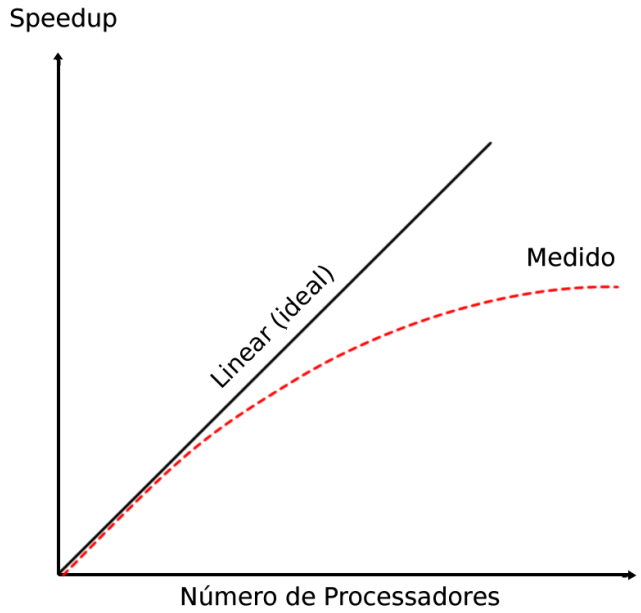


Figura 3.4: Curva típica de speedup - Adaptado de Gustafson (1992)

do problema pode crescer mais rápido que P, levando a tempos de execução maiores. Um cenário mais realista seria permitir o crescimento do problema desde que o tempo total não ultrapasse um valor especificado. A noção de *speedup* por tempo fixo foi apresentada por Gustafson (1988) e detalhada posteriormente por Sun (1990).

A análise com base em um tempo de execução fixo é realizada calculando-se o tempo que seria necessário para que o problema resolvido na máquina paralela seja resolvido na máquina serial:

$$S_{scaled}(P) = \frac{s' + p'P}{s' + p'} = P + (1 - P)s' \tag{3.3}$$

A formulação de Gustafson sugere, portanto, que é possível obter *speedup* que cresce linearmente com o número de processadores.

Na prática, um *speedup* perfeito é difícil de alcançar por diversos fatores. Sun (1990) resumiu as origens da degradação de desempenho em:

- Atrasos de Comunicação - Os atrasos de comunicação incluem o tempo de transmissão e propagação das mensagens, bem como o tempo necessário para preparar a informação para transmissão e o tempo gasto nas filas de transmissão.

- Alocação Desigual - Idealmente, o trabalho a ser realizado é distribuído igualmente entre os processadores disponíveis. Entretanto, o balanceamento perfeito é difícil de ser obtido. Tanto a granularidade das tarefas quanto características de cada uma podem fazer com que alguns processadores terminem antes dos outros. Eles então devem esperar pelos outros, o que desperdiça poder computacional.

- Trabalho Redundante - Algumas vezes pode ser interessante recalcular um determinado passo localmente ao invés de transmiti-lo pela rede. Outras vezes, a repetição do cálculo localmente pode evitar que os processadores esperem ociosamente por um dado sendo calculado em outro nó. Nesses casos, o trabalho realizado novamente impede que todo o potencial de computação seja utilizado de forma útil.
- Conhecimento Reduzido - Alguns algoritmos utilizam informações sobre o progresso do algoritmo para efetuar a próxima iteração. Quando o algoritmo é distribuído, deve-se escolher entre utilizar informações defasadas ou recorrer a sincronizações frequentes. Ambas as opções limitam o desempenho máximo que pode ser atingido.

Uma métrica bastante utilizada para determinar a utilização média dos processadores alocados é a *eficiência*. Se o tempo gasto com a transferência de dados for ignorado, a eficiência em um sistema com 1 processador é 1. A relação entre *speedup* e eficiência pode ser descrita como:

$$E(P) = \frac{S(P)}{P} \tag{3.4}$$

Se a eficiência puder ser mantida constante à medida que mais processadores são adicionados, obtemos um *speedup* linear. Entretanto, um *speedup* linear não pode ser alcançado devido a contenção por recursos compartilhados, pelo tempo necessário para realizar a comunicação entre os processos e pela dificuldade de se particionar as tarefas de modo a manter os processadores igualmente ocupados (Eager et al., 1989).

A escalabilidade de um sistema paralelo refere-se à sua capacidade de aumentar o poder de processamento adicionando-se novos elementos. A escalabilidade pode ser influenciada tanto pela escolha da arquitetura de hardware quanto pelos algoritmos utilizados. Um estudo mais detalhado sobre análise de desempenho de sistemas paralelos pode ser encontrada em Sun (1990) e Jain (1991).

3.4 Comunicação

A natureza distribuída dos processadores e memórias nos agrupamentos de computadores exige um modelo de comunicação de comunicação entre as máquinas. Os dois modelos mais comuns são: MPI - *Message Passing Interface* e PVM - *Parallel Virtual Machine* (Gropp e Lusk, 2002).

O PVM (Sunderam, 1990) surgiu em 1989 nos laboratórios da Emory University e Oak Ridge National Laboratory, com o objetivo de permitir pesquisas em computação heterogênea distribuída. A ideia geral do PVM é apresentar um conjunto de máquinas como uma única máquina virtual paralela, utilizando a transmissão de mensagens internamente (Geist et al., 1996). Devido ao foco do PVM em comunicação baseada em *sockets* entre sistemas fracamente acoplados, ele deu grande ênfase em aspectos como falhas de comunicação.

A primeira versão do padrão MPI (Message Passing Forum, 1994) foi definida pelo Fórum MPI, um comitê de 40 especialistas em computação de alto desempenho entre os anos de

1993 e 1994. O objetivo do MPI era ser uma especificação padrão de passagem de mensagens que pudesse ser implementada por cada fabricante (Geist et al., 1996). O MPI alcançou um grande sucesso como uma maneira portável de se desenvolver programas paralelos. Hoje, diversas implementações estão disponíveis para várias sistemas e arquiteturas paralelas.

Gropp (2001) aponta as razões do sucesso do MPI como sendo: portabilidade, desempenho, simplicidade, modularidade, interoperabilidade e completeza. Embora o MPI tenha um conjunto grandes de funções, a maior parte das aplicações utiliza apenas seis delas:

MPI_Init()	Inicia o MPI
MPI_Finalize()	Finaliza o MPI
MPI_Comm_size()	Determina o número de processos
MPI_Comm_rank()	Determina o identificador do processo
MPI_Send()	Envia uma mensagem
MPI_Recv()	Recebe uma mensagem

Tabela 3.1: Funções básicas do MPI

As funções MPI_Init() e MPI_Finalize() são utilizadas para iniciar ou finalizar o ambiente de troca de mensagens MPI. MPI_Comm_size() e MPI_Comm_rank() são utilizados para determinar o número de processos em execução e o identificador do processo atual respectivamente.

O envio e recepção das mensagens é feito por meio das funções MPI_Send() e MPI_Recv(). Na modo de comunicação padrão, essas chamadas não retornam até que a mensagem seja enviada ou recebida como for o caso. O ambiente de execução do MPI decide, baseado no tamanho da mensagem e *buffers* disponíveis, se o *Send()* vai retornar imediatamente (armazenando a mensagem) ou se vai esperar até toda a mensagem ser recebida.

Além do modo de comunicação padrão, existem os modos *bufferizado*, Síncrono e Pronto. No modo com *buffers*, as mensagens são sempre armazenadas em uma área de memória fornecida pelo usuário. No modo Síncrono, as chamadas retornam somente após o receptor ter recebido toda a mensagem. Por fim, no modo Pronto, a comunicação só ocorre após o receptor atingir a chamada *Recv()*. Cada um dos modos possui rotinas bloqueantes e não bloqueantes:

Nome	Bloqueante	Não Bloqueante
Padrão	send()	isend()
Bufferizado	bsend()	ibsend()
Síncrono	ssend()	issend()
Pronto	rsend()	irsend()

Tabela 3.2: Modos de comunicação e chamadas MPI

No modo bloqueante, a rotina de *Send()* só retorna quando toda a mensagem for enviada ou copiada para o buffer do sistema. Se o *Send()* bloqueante for síncrono uma negociação é feita entre o receptor e o transmissor de forma que a mensagem é entregue durante as chamadas de função. Já uma chamada *Send()* assíncrona pode retornar assim que a mensagem for copiada para um buffer do sistema. Um *Recv()* bloqueante só retorna após os dados serem recebidos e liberados para uso.

Nas rotinas não-bloqueante as chamadas a *Send()* e *Recv()* retornam imediatamente, sem esperar que qualquer tipo de comunicação termine. Dessa forma, não é seguro modificar um *buffer* até que a operação seja concluída. O estado das operações não-bloqueantes pode ser verificado com rotinas especiais de "wait"e "test".

A comunicação necessária quando se paraleliza um algoritmo é um dos fatores que pode impedir uma boa escalabilidade do sistema. Assim, uma abordagem para minimizar o impacto da comunicação é realizá-la simultaneamente com os passos de computação por meio de funções não-bloqueantes.

3.5 Discussão

Nesse capítulo descrevemos os modelos clássicos de paralelização da renderização e as métricas tradicionais de desempenho de sistemas paralelos. Enumeramos os principais problemas que afetam negativamente o *speedup* e a escalabilidade desses sistemas. Elas serão importantes posteriormente para a avaliação teórica e prática do sistema proposto.

Capítulo 4

Arquitetura do Sistema

Neste trabalho, estamos interessados em sintetizar o maior número de pontos de vista possível em um determinado tempo. O número de quadros por intervalo de tempo é conhecido como taxa de quadros ou *frame rate*. Propomos aumentar a taxa de quadros evitando a repetição de trabalho entre cada ponto de vista e ao mesmo tempo mantendo o sistema balanceado.

Gostaríamos também, de obter uma solução adequada para a renderização cenas dinâmicas. Por esta razão, evitamos armazenar resultados intermediários em *buffers* e amostramos a cena novamente a cada quadro.

Nesta seção construímos um modelo teórico para a paralelização proposta utilizando um *light field* e comparamos seu desempenho com outras alternativas por meio de análises de tempo e *speedup*. Nas expressões deste capítulo usaremos as definições a seguir:

literal	descrição
g	Tempo de geração de uma amostra a partir da geometria
c	Tempo de composição de um novo ponto de vista
V	Número de pontos de vista
P	Número total de processadores
S	Número de amostras (samples)
K	Número de processadores de amostragem

Tabela 4.1: Parâmetros relevantes para a análise dos modelos de paralelização.

O tempo de geração depende da complexidade do modelo e da complexidade da renderização. O tempo de composição, por outro lado, não depende do modelo original. Para os fins de análise consideraremos $g >> c$, ou seja, modelos de entrada com grande complexidade geométrica associados a tempos de composição pequenos. Também consideraremos que $V >> S$, pois caso contrário é certamente mais eficiente renderizar diretamente os ponto de vistas necessários.

Estamos especialmente interessados no comportamento do sistema à medida que o número de pontos de vistas V cresce. O efeito do aumento do número total de processadores disponíveis P na taxa de quadros também será avaliado.

literal	descrição
g	0,3
c	0,03
V	50-300
P	5-11
S	4
K	1-2

Tabela 4.2: Faixa de valores típicos para os parâmetros relevantes

4.1 Tempo de Renderização

O tempo necessário para renderizar um número V de pontos de vistas utilizando o *pipeline* padrão de renderização é gV, onde g é o tempo para renderizar um único ponto de vista:

$$T_{trivial} = (gV) \tag{4.1}$$

Para renderizar múltiplos pontos de vista de uma maneira escalável, precisamos de uma arquitetura paralela. A paralelização mais simples é dividir os V pontos de vista entre os P processadores disponíveis, de modo que o tempo transcorrido entre a renderização de dois quadros de um mesmo ponto de vista seja:

$$T_{||trivial} = \frac{gV}{P} \tag{4.2}$$

Podemos assumir que a carga está igualmente balanceada, uma vez que uma imagem pode ser fatiada entre dois ou mais processadores. Essa solução está ilustrada na Figura 4.1.

Essa arquitetura apresenta a vantagem de praticamente não haver comunicação entre as máquinas. Entretanto, para melhorar o desempenho ao se renderizar múltiplos pontos de vista, nós gostaríamos de reutilizar a computação entre os pontos de vista sendo gerados. Para isso, propomos a utilização de um renderizador que utilize um *light field*, ao invés da geometria original.

Se S é o número de imagens necessárias para amostrar a cena e c o tempo necessário para se gerar uma nova imagem a partir das amostras, podemos dividir a computação gV em um passo de composição e um passo de geração:

$$T_{proposto} = (Sg + Vc) \tag{4.3}$$

Como Sg é constante com o aumento do número de pontos de vista, essa formulação é mais rápida que $T_{trivial}$ a partir de $(Sg/g - c)$ pontos de vista desde que $c < g$.

A nova formulação pode agora ser aplicada à arquitetura da Figura 4.1 para explorar a coerência entre os pontos de vista sendo renderizados na *mesma* GPU. Essa nova situação está ilustrada na Figura 4.2. Nela, cada nó gera suas próprias amostras, que são utilizadas

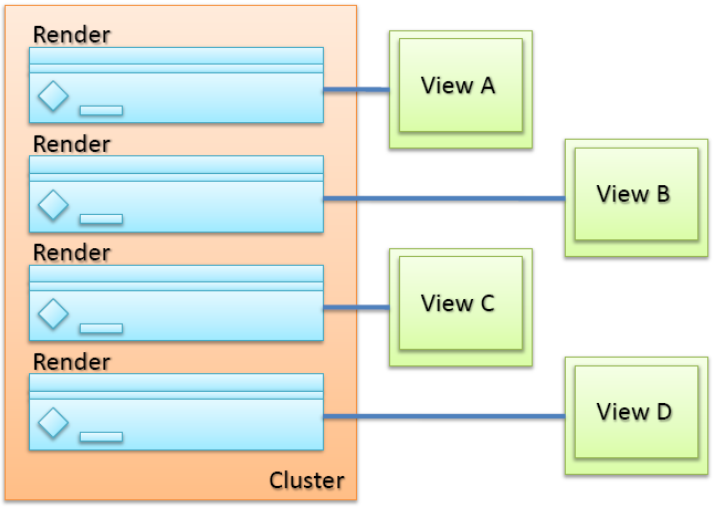


Figura 4.1: Arquitetura intuitiva - Os pontos de vista são distribuídos entre as GPUs disponíveis. Cada GPU implementa todas as etapas do pipeline de renderização.

para sintetizar todos pontos de vista atribuídos ao mesmo nó. Não há comunicação entre os nós. Apesar de melhor que o sistema anterior, essa solução ainda não permite a reutilização dos raios comuns a pontos de vistas sendo renderizados em máquinas diferentes. Apesar do problema não ser expressivo quando o número de GPUs é pequeno, ele torna-se relevante quando o número de máquinas aumenta, limitando a escalabilidade do sistema. O tempo de renderização desse sistema pode ser expresso como:

$$T_{||melhorado} = \frac{V}{P}c + Sg \tag{4.4}$$

A solução ideal, portanto, deve permitir que as amostras geradas possam ser compartilhadas por *todas* as vistas sendo compostas. A paralelização eficiente desse novo algoritmo requer uma mudança na arquitetura original. Assim, dividimos os processadores em um *pipeline* superescalar de dois estágios. O primeiro estágio é responsável pela geração das amostras e o segundo pela composição dos novos pontos de vista (Figura 4.3).

Dos P processadores disponíveis, K são utilizados para amostrar a cena, enquanto $(P-K)$ são utilizados para compor os pontos de vista finais. No novo sistema, o tempo entre dois quadros de um mesmo ponto de vista é igual ao tempo gasto pelo estágio mais lento:

$$T_{||proposto} = Max\left(\frac{Sg}{K}, \frac{Vc}{P-K}\right) \tag{4.5}$$

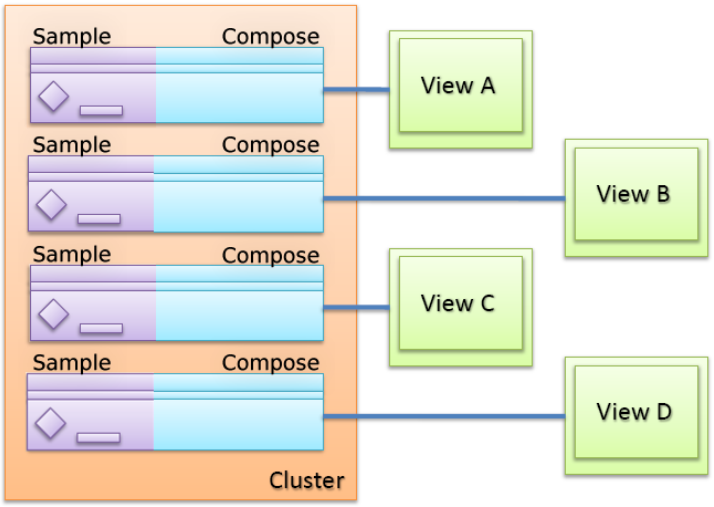


Figura 4.2: Arquitetura Melhorada - Os pontos de vista são distribuídos entre as GPUs disponíveis. Cada GPU gera as amostras e as imagens finais.

4.2 Distribuição das GPUs Entre os Estágios

Uma vez proposto o modelo de *pipeline*, é necessário descobrir uma maneira de distribuir as GPUs disponíveis de maneira a minimizar o tempo total (Equação 4.5). $T_{||proposto}$ é mínimo quando ambos os parâmetros da função max forem o menor possível. Como os termos são inversamente proporcionais, o mínimo é obtido quando ambos os estágios gastam o mesmo tempo. Igualando as expressões de ambos os estágios e isolando K obtemos:

$$K_f = \frac{SgP}{Vc + Sg} \tag{4.6}$$

A equação anterior (4.6) fornece o valor de K que minimiza o tempo em um sistema ideal, onde cada processador pode ser particionado de forma arbitrária entre os dois estágios. Em uma situação real, entretanto, apenas um tipo de tarefa roda em cada GPU. Essa divisão também facilita a integração do sistema com sistemas de renderização já existentes. Por esta razão, consideraremos que apenas um número inteiro de processadores pode ser alocado para cada estágio.

Ao contrário do que poderia se esperar, um simples arredondamento da função K_f não fornece mais a melhor solução para essa situação. Um exemplo pode ser visto na Figura 4.4. Note que no intervalo entre 100 e 130 vistas, o arredondamento sugere $K = 1$. Entretanto, podemos ver que $K = 2$ ofereceria uma melhor solução. O mesmo problema se repete em todos os pontos de transição.

A restrição que $K \in \mathbb{Z}_+^*$ faz com que um balanceamento perfeito não possa mais ser

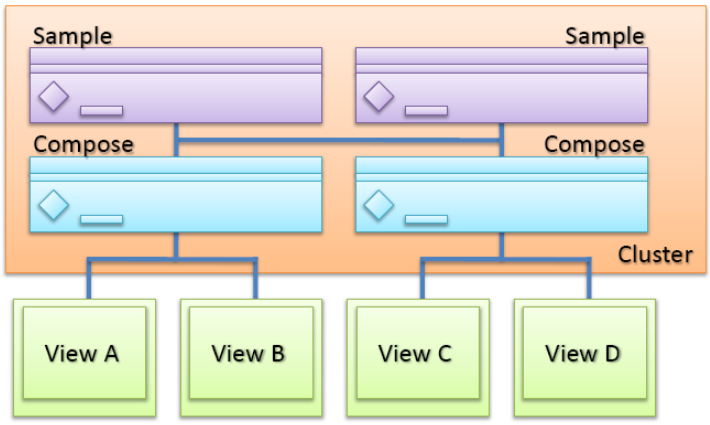


Figura 4.3: Arquitetura Proposta - O pipeline é dividido em um estágio de amostragem e um de composição. As GPUs são atribuidas para cada estágio de acordo com o número de pontos de vista sendo gerados.

alcançado para todos os valores de V. Isso transforma o problema de encontrar o mínimo de uma função contínua no problema de minimizar uma função que admite apenas valores inteiros. Especificamente, queremos encontrar uma função que forneça o valor de K no qual $T_{||proposto}$ seja o menor possível.

Observando a expressão do tempo da paralelização Proposta (Equação 4.5), notamos que quando variamos apenas o número de pontos de vista, o tempo é dado pelo maior valor entre uma função constante igual a Sg/k e uma reta de inclinação $c/(P-K)$. Ou seja, para um certo $k \in [1, P)$, temos:

$$\begin{cases} T_{||proposto} = \frac{Sg}{K}, \ p/V \leq \frac{Sg}{c}. \\ T_{||proposto} = \frac{Vc}{(P-K)}, \ p/V \geq \frac{Sg}{c}. \end{cases} \tag{4.7}$$

O ponto onde K deixa de ser ótimo em relação a $K-1$ é aquele onde:

$$\frac{Vc}{P-K} = \frac{Sg}{K-1} \tag{4.8}$$

Isolando K obtemos:

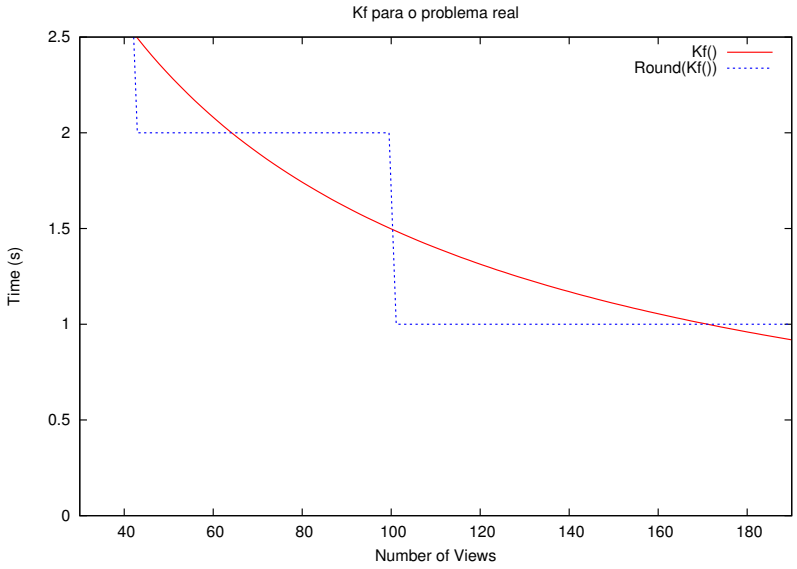
$$K_{fd}(S, g, V, c, P) = \left\lfloor \frac{SgP + Vc}{Vc + Sg} \right\rfloor \tag{4.9}$$

onde:

$$0 < K_{fd} < P \tag{4.10}$$

Um exemplo desse comportamento pode ser visto na Figura 4.5 para $K = 2$. Quando o valor de K diminui, obtemos retas de inclinação cada vez maior e constantes de valor menor.

[A função K_f e sua versão discretizada.]



[Tempo obtido utilizando-se como K o valor arredondado da função K_f (linha contínua). O valor de K no intervalo 50-130 não corresponde ao menor tempo possível.]

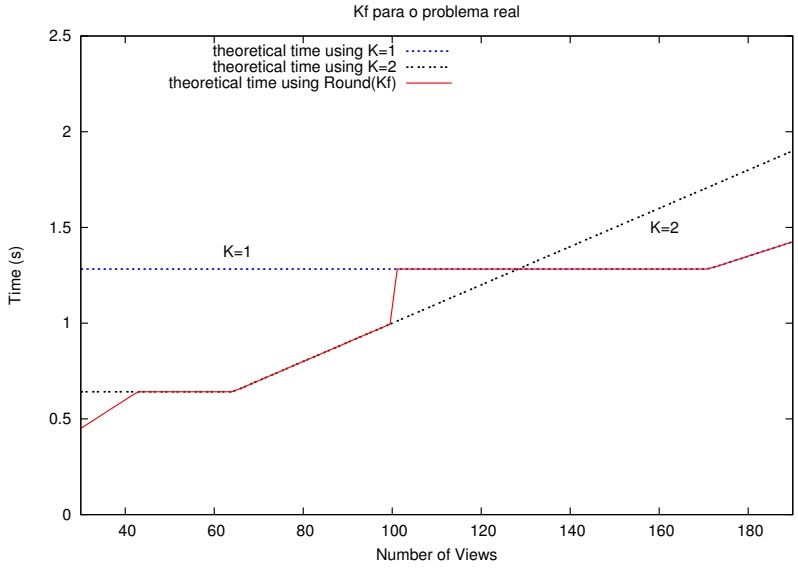


Figura 4.4: Influência do arredondamento da função K_f no tempo de renderização.

O gráfico para os valores de K em um *cluster* de 5 máquinas pode ser visto na Figura 4.6. A curva de menor tempo começa com $K = 4$ (1 compositor e 4 amostradores), e decresce até $k = 1$ (4 compositores e 1 amostrador). De maneira geral, sempre que $(Vc)/(P - K)$ se torna maior que o próximo K $((Sg)/(k - 1))$ o valor seguinte de K deve ser adotado.

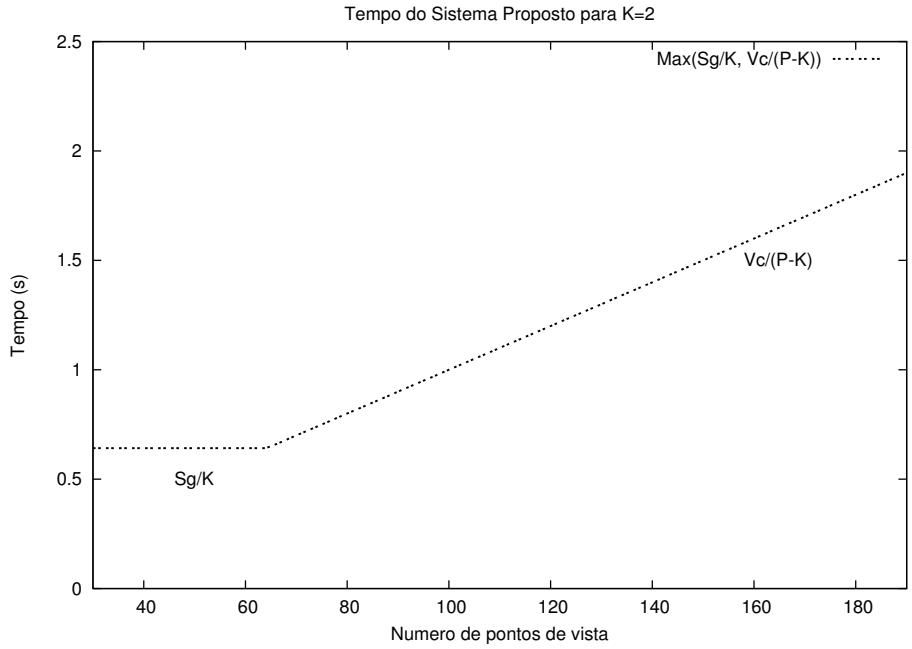


Figura 4.5: Comportamento do tempo de execução do sistema proposto para cinco processadores, quatro amostras e K=2. O tempo de geração utilizado foi 0,3s e o de composição 0,03s.

4.3 Análise da Escalabilidade

Para avaliar os três sistemas descritos anteriormente, vamos analisar o tempo e *speedup* obtido por cada uma delas. As expressões do tempo derivadas na seção anterior foram:

$$T_{proposto} = (Sg + Vc) \tag{4.11}$$

$$T_{||trivial} = \frac{gV}{P} \tag{4.12}$$

$$T_{||melhorado} = \frac{V}{P}c + Sg \tag{4.13}$$

$$T_{||proposto} = Max\left(\frac{Sg}{K}, \frac{Vc}{P-K}\right) \tag{4.14}$$

Para facilitar a comparação com o Sistema Proposto vamos mostrar que um limite assintótico firme para seu tempo de execução é:

$$T_{||proposto}(V) = \Theta\left(\frac{Vc + Sg}{P}\right) \tag{4.15}$$

$T_{||proposto}$ é mínimo sempre que o balanceamento for perfeito. Nesse caso :

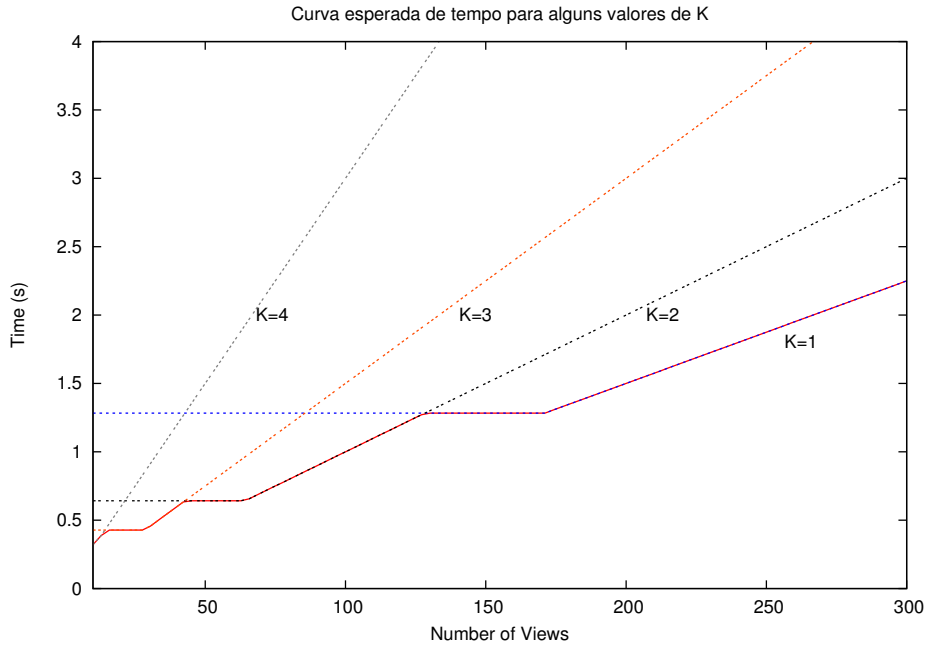


Figura 4.6: Comportamento do tempo total para diversos valores de K para um *cluster* de 5 máquinas. A curva contínua corresponde ao tempo ótimo obtido pela Equação 4.9. O tempo de geração utilizado foi 0,3s, de composição 0,03s e o número de amostras 4.

$$T_{||proposto}(V) \geq \frac{Vc + Sg}{P} \tag{4.16}$$

Por outro lado, o desbalanceamento máximo que pode ocorrer é o devido à diferença de um processador entre os estágios. Assim o limite superior pode ser dado por:

$$T_{||proposto}(V) \leq \frac{Vc + Sg}{P - 1} \tag{4.17}$$

A razão entre as duas expressões é uma constante em relação a V: $P/(P-1)$. O limites podem ser vistos na Figura 4.7.

Quanto ao número de pontos de vistas

O Sistema Trivial apresenta a maior dependência com o número de pontos de vista, uma vez que o tempo cresce com a inclinação g/P que é claramente maior do que a do Sistema Melhorado c/P. O Sistema Melhorado inicia com um tempo menor devido a constante menor $(Sg)/(P-1)$ no pior caso. Entretanto seu crescimento é ligeiramente maior. Para valores de V maiores que $(SgP(P-2))/c$ o Sistema Melhorado obtém um tempo menor que o Sistema Proposto se mantivermos P fixo. Mostraremos, entretanto, que esta solução é menos escalável com o número de processadores.

Quanto ao número de amostras

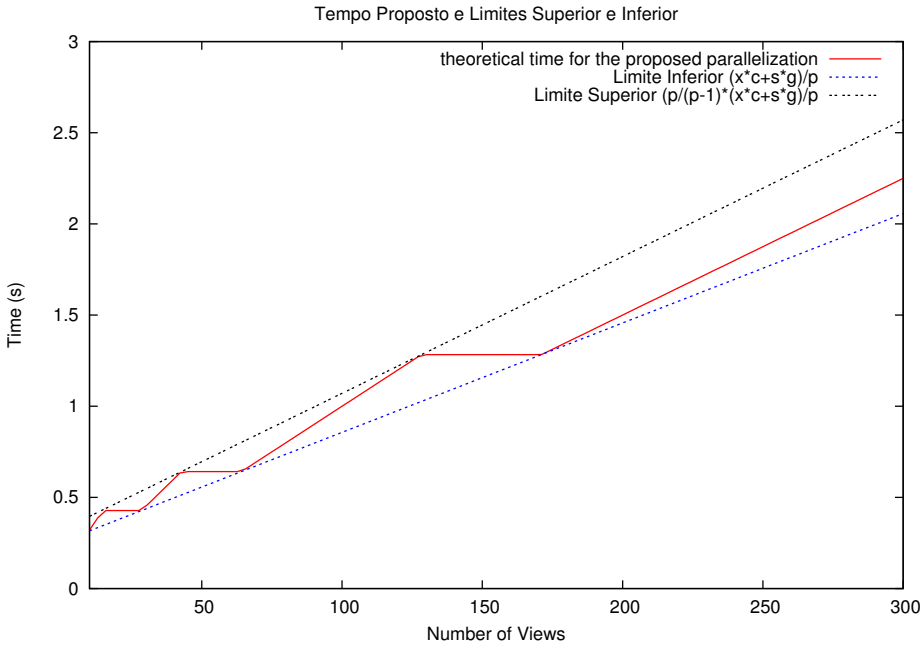


Figura 4.7: Limites superior e inferior para o tempo da paralelização Proposta para valores de K inteiros.

O número de amostras S necessárias para a síntese de uma imagem depende de diversos fatores, como a presença ou não de informações de profundidade, do tipo de câmera utilizada (*pinhole* ou não) e da topologia da cena. Entretanto, de maneira geral, um número maior de amostras possibilita uma reconstrução de melhor qualidade. Desse modo, estamos interessados em avaliar o comportamento do sistema com o aumento do número de amostras.

Apesar do custo de composição de novos pontos de vista ser relativamente independente do número de amostras, a geração de amostras representa uma sobrecarga adicional em relação ao processo de renderização a partir da geometria. Em geral, a formulação $Sg + Vc$ se torna interessante quando o número de pontos de vista é tal que:

$$Sg + Vc < Vg \tag{4.18}$$

$$V > \frac{Sg}{g - c} \tag{4.19}$$

Se o número de número de amostras S ou o tempo de amostragem g cresce, é necessário gerar mais pontos de vista para compensar a sobrecarga adicional da geração de amostras. A Equação 4.19 também nos mostra que quanto maior for a diferença entre o tempo de geração e o de composição, mais cedo o benefício da nova formulação aparece.

Quanto ao número de processadores

A capacidade de se reduzir o tempo quando mais processadores são adicionados ao sistema é um reflexo de sua escalabilidade. Particularmente:

$$\lim_{P\to\infty} T_{||trivial} = \lim_{P\to\infty} T_{||proposto} = 0 \tag{4.20}$$

Entretanto, como a constante $gV > (Sg + Vc)$, para um dado P o Sistema Proposto apresenta um tempo menor. Por outro lado temos que:

$$\lim_{P\to\infty} T_{||melhorado} = Sg \tag{4.21}$$

Isso mostra que no Sistema Melhorado não é possível reduzir o tempo abaixo de Sg, não importando o número de processadores adicionados ao sistema.

Speedup

Para capturar melhor as diferenças no desempenho dos três sistemas considerados, vamos analisar o aumento de desempenho de cada um deles frente a um carga fixa e a um tempo fixo.

Uma vez que o algoritmo de renderização utilizando *light fields* também pode ser utilizado para acelerar a renderização em um processador serial, utilizaremos esse tempo para obter o *speedup* S para os três sistemas.

Fixed Size Speedup

Dado uma carga de tamanho pré-definido, o speedup é a razão entre o tempo gasto pelo sistema serial sobre o tempo gasto pelo sistema paralelo. Como comparação utilizamos o sistema serial mais rápido ($T_{proposto}$).

$$Ss_{trivial} = \frac{T_{proposto}}{T_{||trivial}} = \frac{Sg + Vc}{gV} \cdot P \tag{4.22}$$

$$Ss_{melhorado} = \frac{T_{proposto}}{T_{||melhorado}} = \frac{Sg + Vc}{Vc + PSg} \cdot P \tag{4.23}$$

$$Ss_{proposto} = \frac{T_{proposto}}{T_{||proposto}} = P - 1 \tag{4.24}$$

Verificando os limites quando $V \to \infty$, vemos que a diferença de speedup entre os Sistemas Melhorado e Proposto é 1 e que o Sistema Trivial apresenta o pior speedup de todos, tendendo a $(cP)/g$.

$$\lim_{V\to\infty} Ss_{trivial} = P\frac{c}{g} \tag{4.25}$$

$$\lim_{V\to\infty} Ss_{melhorado} = P \tag{4.26}$$

$$\lim_{V\to\infty} Ss_{proposto} = P - 1 \tag{4.27}$$

Fixed Time Speedup

O *speedup* de tempo fixo é a razão entre o tempo necessário para que o sistema serial realize o mesmo trabalho executado pelo sistema paralelo em um intervalo fixo de tempo. Ou seja, o speedup de tempo fixo reflete diretamente o aumento na taxa de quadros, que é a métrica importante nos sistemas interativos. Novamente utilizamos como comparação o melhor sistema serial (máquina serial rodando o algoritmo proposto):

$$St_{trivial} = \frac{T_{proposto}}{T_{||trivial}} = P \cdot \frac{cT}{g} + Sg \tag{4.28}$$

$$St_{melhorado} = \frac{T_{proposto}}{T_{||melhorado}} = P(T - Sg) + Sg \tag{4.29}$$

$$St_{proposto} = \frac{T_{proposto}}{T_{||proposto}} = PT - T - Sg \tag{4.30}$$

T é o intervalo de tempo fixo considerado.

Observamos que o speedup de tempo fixo dos três sistemas cresce linearmente com o aumento do número de processadores, ao contrário do speedup de carga fixa. Apesar disso, observamos que $S_{proposto}$ cresce muito mais rapidamente que as demais alternativas. Para $T = 2Sg$ por exemplo temos:

$$St_{trivial} = c(PSg) \tag{4.31}$$

$$St_{melhorado} = (Psg) \tag{4.32}$$

$$St_{proposto} = 2(PSg) \tag{4.33}$$

4.4 Discussão

Apresentamos nesse capítulo as conclusões mais importantes dessa dissertação. Demonstramos analiticamente que a arquitetura proposta apresenta maior escalabilidade e *speedup* do que alternativas mais simples. Desenvolvemos também a expressão que permite minimizar o desbalanceamento se considerarmos apenas um número inteiro de GPUs. As análises desse capítulo partiram do pressuposto de que o tempo de comunicação é pequeno perto do tempo de renderização de cada estágio. A validação dessa premissa e a comparação do modelo teórico com experimentos reais serão apresentados no Capítulo 6.

Capítulo 5

Implementação

5.1 Renderizando um Ponto de Vista

Nosso renderizador de *light field* baseia-se nos trabalhos sobre a utilização de informação de profundidade por *pixel* para a correção dos raios durante a reconstrução (Schirmacher et al., 2001; Todt et al., 2007; Heidrich et al., 1999; Vogelgsang e Greiner, 2000).

Como discutido na seção 2.2.1, existem diversas maneiras de se parametrizar um *light field* 4D. A melhor depende do domínio da aplicação, mas geralmente uma boa parametrização possui boas características de amostragem e pode facilitar a tarefa de reconstrução. Entre as parametrizações propostas na literatura podemos citar a parametrização de dois planos (2PP)(Levoy e Hanrahan, 1996), cilíndrica (McMillan e Bishop, 1995), esférica (Todt et al., 2007; Ihm et al., 1997), duas esferas, plano-esfera (Camahort et al., 1998) e não estruturada (Schirmacher et al., 2001; Buehler et al., 2001; Takahashi e Naemura, 2005).

Nesse trabalho, decidimos utilizar como geometria de teste um terreno descrito por um mapa de altitude. Por essa razão, concluímos que a parametrização de 2 planos (2PP) seria uma boa escolha. A parametrização de dois planos indexa cada raio que entra na cena por dois pares de pontos (u, v) e (s, t), um em cada plano. O plano (s, t) é conhecido como plano no ponto de vista e o plano (u, v) como plano de imagem.

5.1.1 Obtendo Imagens com Profundidade

Para obter as imagens, particionamos a cena em blocos. Cada bloco foi delimitado pelo volume formado pelos planos $z = 1$, $z = 0$, $x = \pm 1$ e $y = \pm 1$. Como o custo de amostragem não cresce com o aumento do número de pontos de vista, escolhemos utilizar 4 câmeras de amostragem por bloco, o que permitiu um mapeamento eficiente para a implementação em *shaders*. Cenas mais complexas podem ser partidas em blocos menores, aumentando assim a densidade de câmeras por volume. As quatro câmeras de amostragem foram dispostas nas posições $(-1, -1, z)$ $(-1, 1, z)$, $(1, -1, z)$ e $(1, 1, z)$. Cada câmera tem um campo de visão de 45 graus e está apontada para a origem do mundo.

A cada quadro, as quatro câmeras novamente renderizam a cena (Figura 5.1) e armazenam a profundidade do *pixel* no canal de alpha da textura como:

$$d = \frac{|P_g - P_1|}{|P_1 - P_0|} \tag{5.1}$$

onde P_1 e P_0 denotam os pontos de interseção do raio que sai da câmera com o plano $z = 1$ e $z = 0$ respectivamente. P_g é o ponto onde o raio de visão intercepta a geometria (isso pode ser visto na Figura 5.3, para o raio que sai da câmera Ca). Um exemplo do canal de alpha pode ser visto na Figura 5.2.

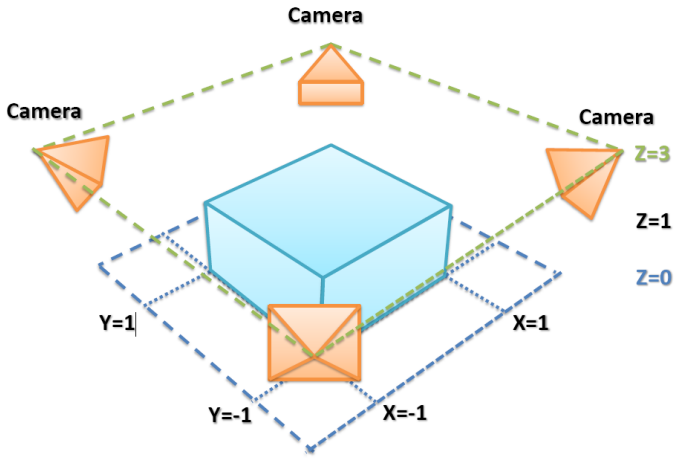


Figura 5.1: Quatro câmeras são utilizadas para amostrar cada bloco da cena. O volume de amostragem é delimitado pelos planos $z = 0$, $z = 1$, $x = \pm 1$, $y = \pm 1$

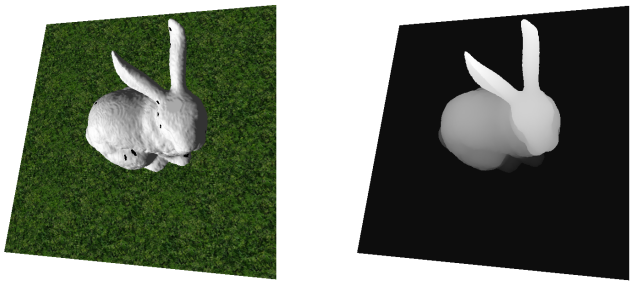


Figura 5.2: Canal RBG e canal alpha de uma amostra da cena

5.1.2 Renderizando Novas Vistas

O algoritmo de renderização foi implementado utilizando-se um programa de fragmento em GLSL. Para que seja possível renderizar um ponto de vista, precisamos da posição da câmera a ser sintetizada P_v, a posição de cada uma das k câmeras de amostragem P_{ck} e a imagem+profundidade de cada uma I_{ck}. Precisamos também da matriz de projeção $Proj$ que especifica o modelo da câmera. Neste trabalho consideramos que todas as câmeras estão

sempre olhando para o centro de cada bloco, portanto, a informação de orientação não é necessária.

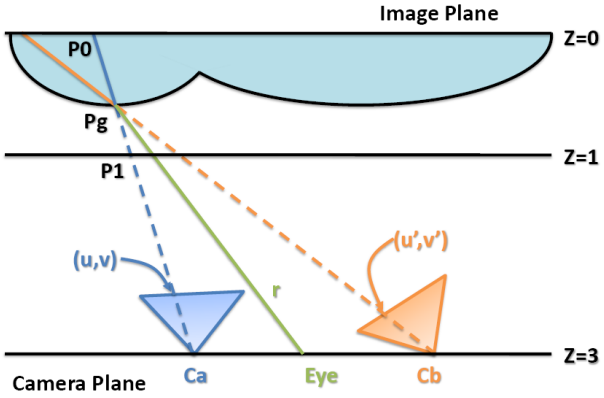


Figura 5.3: Dado um raio r, devemos encontrar as coordenadas (u,v) e (u',v') que correspondem ao ponto de interseção P_g

Para que os *shaders* de fragmento tenham uma superfície para desenhar, para cada bloco descrito anteriormente, desenhamos um retângulo. O quad é paralelo ao plano $z = 0$ e posicionado na origem. Isso garante uma superfície de renderização desde que a câmera tenha sua coordenada Z maior que a distância do *near clipping plane*.

Para renderizar novos pontos de vista, devemos decidir para cada *pixel* qual das amostras utilizar. Se a profundidade é conhecida, podemos fazer um warp para projetar uma vista para a outra (McMillan, 1997). Isso pode ser feito realizando-se uma busca ao longo do raio que sai da câmera, reprojetando o *pixel* em cada câmera de amostragem e comparando a distância com o valor armazenado na imagem (Todt et al., 2007).

Primeiro, precisamos calcular o vetor que descreve o raio de visualização para cada *pixel*. Isso pode ser feito passando-se a posição do vértice como uma variável interpolada do vertex shader:

$$v = normalize(VertexPos - Cam\_Pos) \tag{5.2}$$

Como todas as câmeras de amostragem C_{kp} são idênticas, todas elas podem descritas pela mesma matriz de projeção $Proj$. Apesar disso, cada uma possui uma posição e orientação própria. A matriz de projeção completa pode ser obtida compondo-se a matriz de projeção $Proj$ com a posição $offset_k$ e orientação rot_k de cada câmera.

$$Proj_k = Proj * rot_k * offset_k \tag{5.3}$$

Escolhemos utilizar o mapeamento inverso para que fosse possível utilizar a interpolação da GPU e evitar buracos na imagem. Para encontrar a correspondência entre as câmeras, é necessário saber a profundidade de cada *pixel* na nova imagem. Existem várias maneiras

de se fazer isso mas todas elas envolvem algum tipo de busca ao longo do raio de visão v para encontrar a interseção com a superfície implicitamente definida pelas outras câmeras. Visando a simplicidade, escolhemos uma busca linear.

A busca começa no ponto onde v intersepta o plano $z = 1$. A cada iteração, o ponto atual pode ser expresso por:

$$P = plane.s1 + v * \delta \tag{5.4}$$

onde

$$\delta = \frac{|plane.s - plane.s1|}{steps} \tag{5.5}$$

e $plane.s1$ e $plane.s$ correspondem às interseções entre v e $z = 0$ e $z = 1$ respectivamente.

Para encontrar a interseção com a superfície empregamos uma abordagem similar à utilizada por Todt et al. (2007): a cada passo, projetamos o ponto corrente na câmera k. Com isso obtemos o par (x, y) que pode ser utilizado para recuperar a posição da superfície P_{sk} armazenada para cada câmera ao longo do raio P_{ck}

$$C_{ks} = plane.s * C_{ki}.depth + (1 - C_{ki}.depth) * plane.s1 \tag{5.6}$$

Entretanto, ao invés de comparar a posição P com a estimada P_{sk}, comparamos a distância de P_{ck} até o ponto atual P e a distância de P_{ck} até a superfície P_{sk}.

Se $(|P_{sk} - P_{ck}| - |P - P_{ck}|) < 0$ o ponto P entrou na superfície e pode ser refinado por meio de uma busca binária.

A busca continua até que P seja considerado dentro da superfície para a estimativa de todas as câmeras P_{sk}. Quando isso ocorre, escolhemos o ponto com a estimativa mais conservadora (aquela mais perto da superfície). Note que caso o raio não intercepte nenhuma superfície, ele acaba percorrendo todo o intervalo entre $z = 1$ e $z = 0$. Do mesmo modo, superfícies próximas de $z = 1$ podem fazer com que o raio termine mais cedo e a imagem gaste um tempo ligeiramente menor para ser sintetizada. A contribuição de cada câmera para a imagem final pode ser vista na Figura 5.4.

5.2 Renderizando Múltiplos Pontos de Vista

Nesta seção detalhamos a implementação da paralelização

5.2.1 Detalhamento dos Servidores

O sistema foi implementado em C++ utilizando dois processos: Um realiza a amostragem da cena e o outro a composição das imagens. O laço principal do produtor renderiza todas as imagens e posteriormente as envia para cada processo de composição.

O método *Render_Image* é um passo de renderização normal de geometria, apenas incluindo o cálculo da distância discutido no seção 5.1.1. O cálculo da distância pode ser

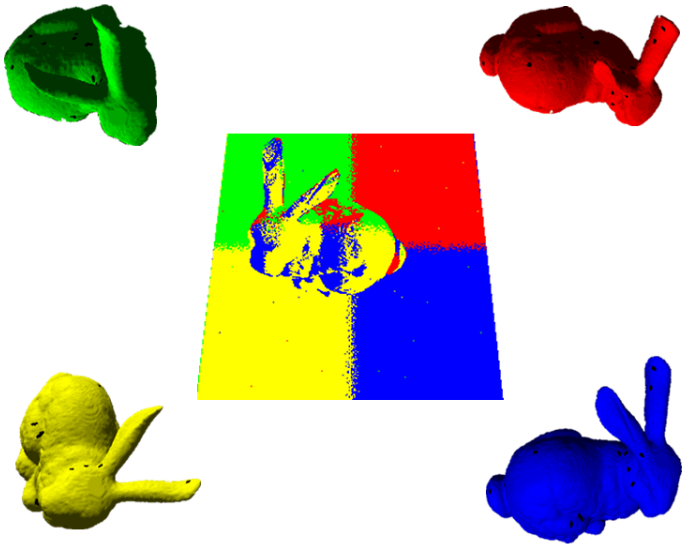


Figura 5.4: Amostras de entrada e a contribuição na imagem final

Algorithm 1 Algoritmo do Primeiro Estágio (Amostragem)

Require: Posição das k câmeras de amostragem.
Require: Geometria da cena G.
Require: Câmeras de amostragem k.
Require: ID dos processos de composição pc.

```
1:  while verdadeiro do
2:      for cada câmera de amostragem k do
3:          Image[k] ← RenderImage(k,G)
4:      end for
5:      for cada câmera de amostragem k do
6:          for cada processo de composição pc do
7:              MPI_Send(pc,Image[k])
8:          end for
9:      end for
10: end while
```

realizado no vertex shader e interpolado por hardware para cada *pixel*, antes de ser gravado no *frame* buffer. O código em GLSL está listado no Apêndice A.

O algoritmo de composição recebe as quatro imagens e as utiliza para compor todas as vistas do quadro atual. Para que o compositor não fique ocioso esperando as imagens, antes de compor os pontos de vista do *frame* i ele requisita os quadros do *frame* $i+1$ por meio do método não bloqueante *Irecv*. Desse modo, o tempo de comunicação pode ser sobreposto ao tempo de computação da GPU. As imagens são gravadas diretamente em áreas contíguas de memória, formando uma textura que pode ser indexada posteriormente no shader (Figura 5.5).

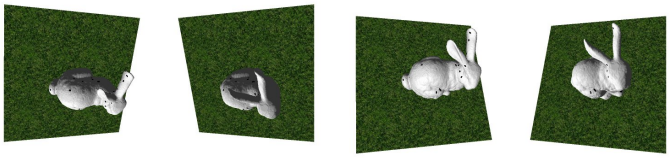


Figura 5.5: Disposição das imagens de entrada na textura

Algorithm 2 Algoritmo do Segundo Estágio (Composição)

Require: Posição das k câmeras de amostragem.
Require: Posição dos novos pontos de vista v
Require: ID dos processos de amostragem pa.

```
1: Samples[0] ← Requisita_Imagens(pa[0])
2: Espera_Imagens(pa[0])
3: while verdadeiro do
4:     Samples[1] ← Requisita_Imagens(pa[1])
5:     Clear Buffer
6:     for cada ponto de vista v do
7:         ComposeImage(v,Samples[0],k)
8:     end for
9:     Espera_Imagens(pa[1])
10: end while
```

5.3 Discussão

Nesse capítulo descrevemos a implementação do sistema paralelo proposto no Capítulo 4. O renderizador de *light field* desenvolvido utiliza o valor da profundidade por pixel para indexar os raios mais apropriados para a reconstrução e roda inteiramente na GPU. A comunicação entre os dois estágios do pipeline é realizada simultaneamente com a renderização, utilizando mensagens MPI.

Capítulo 6

Análise dos Resultados

Neste capítulo analisamos os dados coletados para investigar o *speedup*, eficiência, custos de comunicação e o balanceamento de carga em uma implementação real.

6.1 Metodologia

A arquitetura descrita nessa dissertação foi implementada em um *cluster* Linux composto por onze computadores com processadores dual-Pentiums 3.40GHz 64bits, conectados por uma rede *Gigabit ethernet*. Cada PC possui 3GB de memória principal e uma placa aceleradora Nvidia GeForce 7900 GTX/PCI/SSE2. Nós utilizamos OpenGl e GLSL para os gráficos e LAM/MPI: 7.0.6 para a comunicação entre os processos.

O sistema foi avaliado em dois aspectos: Desempenho e Qualidade. O foco deste trabalho foi obter uma boa paralelização, por isso durante as simulações, foram coletados:

1. `O tempo necessário para se renderizar uma imagem a partir da geometria` - corresponde ao tempo g de se gerar uma amostra da cena. Corresponde também ao tempo que o algoritmo serial $T_{trivial}$ gasta para renderizar um ponto de vista.

2. `O tempo necessário para se renderizar uma imagem a partir das amostras` - corresponde ao tempo de composição c e foi medido desde o momento no qual enviamos os triângulos e parâmetros do shader até o momento no qual o *frame buffer* está pronto para ser exibido. O sincronismo foi assegurado com uma chamada a $glFinish()$.

3. `O tempo gasto em comunicação entre a GPU e a CPU` - O tempo de envio das texturas para a GPU corresponde ao tempo gasto para fazer chamada a glTexImage2D() e o tempo gasto na leitura do *frame buffer* corresponde ao tempo da chamada a glReadPixels().

4. `O tempo total gasto na transferência de dados na rede` - corresponde ao tempo gasto pelo segundo estágio esperando que amostras da cena serem transmitidas. Como existe sobreposição da comunicação, esse tempo é sempre menor que o tempo realmente gasto nas transmissões de rede.

5. `O tempo total de execução` - Corresponde ao tempo total transcorrido entre o início e fim do processo de renderização.

Nos testes de desempenho utilizamos como modelo de entrada uma cena composta por uma árvore com aproximadamente 300 mil triângulos em um terreno descrito por uma mapa de alturas (ver Figura 6.1). A cena foi iluminada por um shader de *phong* simples.

Figura 6.1: Imagem do modelo utilizado nos testes de desempenho

Para cada câmera, renderizamos dez quadros, movendo o ponto de vista entre eles. Os primeiros cinco quadros foram desconsiderados. As amostras foram obtidas utilizando-se uma câmera com abertura de 45 graus e uma resolução de 400x300 *pixels*. A saída foi renderizada em 200x150 pixels.

Para melhorar a taxa de quadros, a transmissão foi intercalada com a renderização. As amostras foram transmitidas sem compressão utilizando primitivas MPI de comunicação ponto a ponto. Nos testes envolvendo um número variável de GPUS, utilizamos a função K_fd (Equação 4.9) para assegurar um melhor balanceamento de carga entre os estágios.

Para que fosse possível avaliar a qualidade do renderizador de *light field* desenvolvido, comparamos algumas imagens geradas a partir da geometria original e a partir das amostras adquiridas. A comparação entre as imagens foi feita utilizando-se a ferrramenta Perceptual Image Diff Utility[1]. Esta ferramente é baseada no *Visible Differences Predictor* (VDP) de Daly (1993). O VDP dá a probabilidade de detecção da diferença entre cada pixel de duas imagens. O modelo leva em conta três aspectos do Sistema Visual Humano (SHV): o primeiro é a não linearidade da sensibilidade a mudanças de contraste, já que o SVH é mais sensível a baixas condições de iluminação. Segundo, a sensibilidade do SVH diminui com o aumento das

[1] http://pdiff.sourceforge.net/

frequências espaciais. Por último, o efeito de *mascaramento* que ocorre quando a sensibilidade é ofuscada por sinais presentes no fundo da imagem (*masking*).

De maneira similar ao trabalho de Ramasubramanian et al. (1999), o PDIF não inclui a orientação no cálculo das frequências espaciais mas inclui a cor no processo de discriminação (Yee e Newman, 2004). Entretanto, apesar das estratégias descritas, o VDP ainda é bastante conservativo e muitas vezes marca como visíveis diferenças que normalmente não seriam perceptíveis (Ramanarayanan et al., 2007).

6.2 Desempenho

Medimos o tempo total de renderização (Figura 6.3) e o tempo total gasto em cada subtarefa (Figura 6.4) para um número crescente de pontos de vista. Nosso sistema foi capaz de renderizar 290 pontos de vista em 2.61 segundos, gastando aproximadamente 9ms por quadro. Foram utilizadas cinco GPUs, que foram distribuídas entre os estágios de acordo com a Equação 4.9 discutida na seção 4.2:

$$K_{fd}(S, g, V, c, P) = \left\lfloor \frac{SgP + Vc}{Vc + Sg} \right\rfloor \tag{6.1}$$

A equação acima foi avaliada para cada valor de V utilizando como parâmetros os tempos de composição $c = 0,03s$, tempo de geração $g = 0,3s$ medidos em uma execução prévia. Os resultados foram:

$$\begin{cases} K = 2, \ se \ V \leq 128 \\ K = 1, \ se \ V \geq 129 \end{cases} \tag{6.2}$$

Para validar o valor de K (número de GPUs de amostragem) obtido realizamos duas execuções completas com as duas opções. Conforme pode ser visto na Figura 6.2, para $V = 130$ a configuração ótima utiliza apenas um GPU para amostragem.

Os resultados obtidos mostraram que o tempo de comunicação representa uma fração pequena do tempo total, mesmo para 50 pontos de vista. O tempo gasto na chamada a *MPI_ Recv()* aumentou ligeiramente a partir de $V = 130$, quando mais nós foram alocados para amostragem. Isso era esperado, uma vez que utilizamos primitivas de comunicação ponto-a-ponto. Uma primitiva de comunicação coletiva como o *MPI_ Broadcast()*, proporcionaria um crescimento ainda mais lento dos custos de comunicação. Observamos, também, um aumento não esperado do tempo de recepção quando $V = 130$. Especulamos que ele esteja relacionado a uma redução temporária do tempo de composição.

Os custos de amostragem e de comunicação confirmaram serem relativamente constantes com o aumento da carga, apenas mudando quando a configuração é alterada. Desse modo, o tempo total de execução é dominado pelo tempo de composição dos novos pontos de vista.

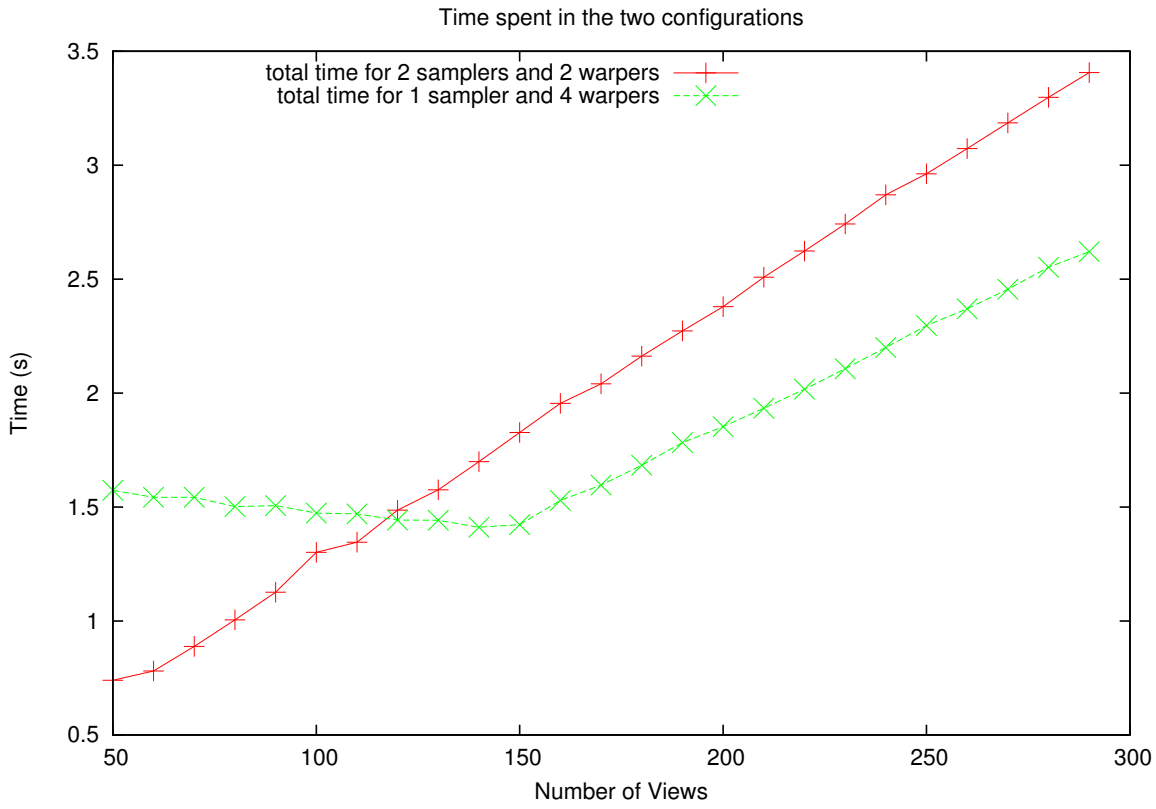


Figura 6.2: Tempo gasto utilizando duas configurações diferentes de K para o mesmo número de GPUs.

6.3 Speedup, Eficiência e Escalabilidade

Para calcular a escalabilidade do sistema, comparamos o tempo total de execução da nossa implementação com o que seria gasto por uma paralelização trivial perfeita. Como discutido anteriormente, este custo é igual a (gV). Para cada configuração foram gerados dez quadros, sendo que os primeiro cinco foram descartados.

Primeiramente avaliamos o comportamento do *speedup* e da eficiência com o aumento do número de processadores. Para isso, medimos o tempo de execução para uma carga fixa de 1000 pontos de vista, variando o número de GPUs disponíveis entre 5 e 11. Em todas as configurações, apenas uma GPU foi dedicada para a amostragem da cena (estágio 1). O gráfico do *speedup* obtido pode ser visto na Figura 6.5, a eficiência associada pode ser vista na Tabela 6.1.

Em seguida, avaliamos a variação da eficiência da paralelização Proposta e da paralelização Trivial com relação ao número de pontos de vista. Os resultados podem ser vistos na Figura 6.6. Foram utilizadas cinco GPUs e o número de pontos de vista variou de 50 a 290. Em $V = 130$, uma das GPUs alocada para a geração de amostras é transferida para o estágio de composição de acordo com a expressão do K ótimo. A configuração depois desse ponto é

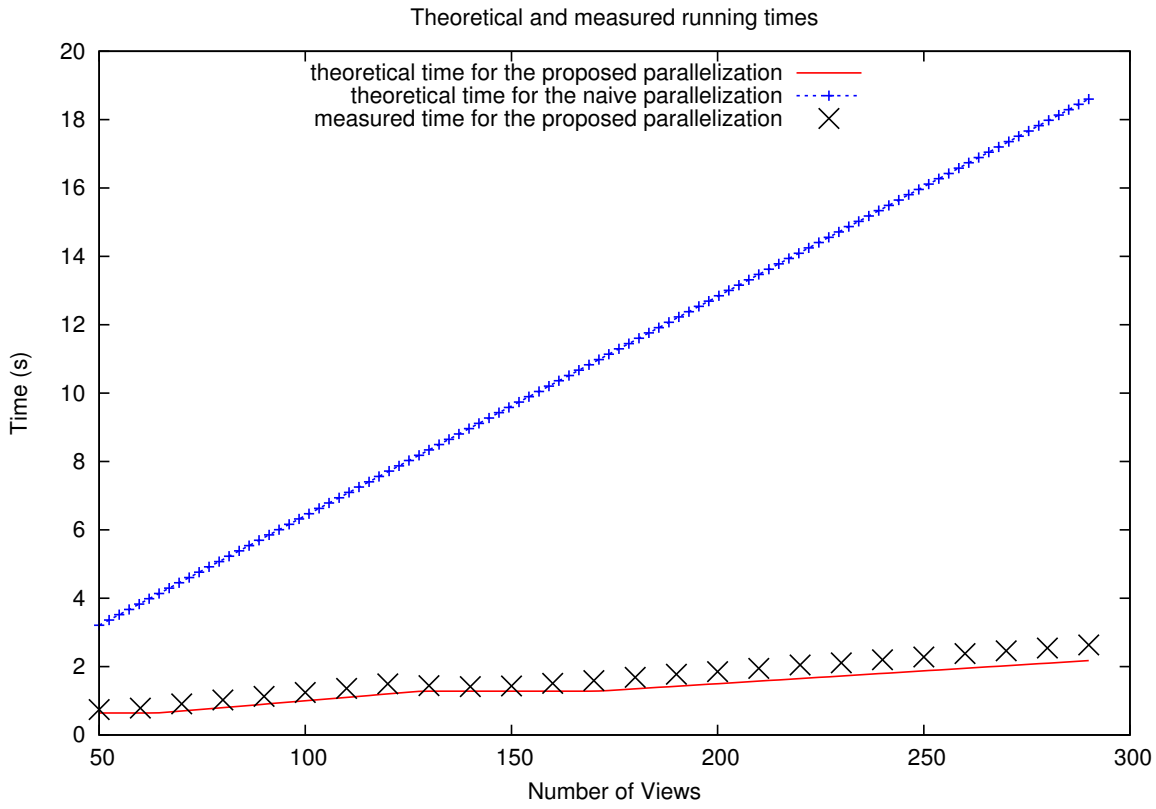


Figura 6.3: Tempo gasto pela paralelização trivial e pela nossa paralelização utilizando cinco GPUs

1 GPU para amostragem da cena e 4 para a geração de novos pontos de vista. Observamos que, como esperado, a alteração da configuração permite que a eficiência se mantenha entre 70% e 80%.

Número de GPUs	Proposta	Trivial
5	0.715	0.103
6	0.757	0.105
7	0.792	0.106
8	0.812	0.108
9	0.818	0.108
10	0.826	0.110
11	0.829	0.110

Tabela 6.1: Eficiência da paralelização proposta e trivial para 1000 pontos de vista. A eficiência pode ser expressa pelo *speedup* dividido pelo número de processadores. O Número de GPUs alocadas para amostragem: 1

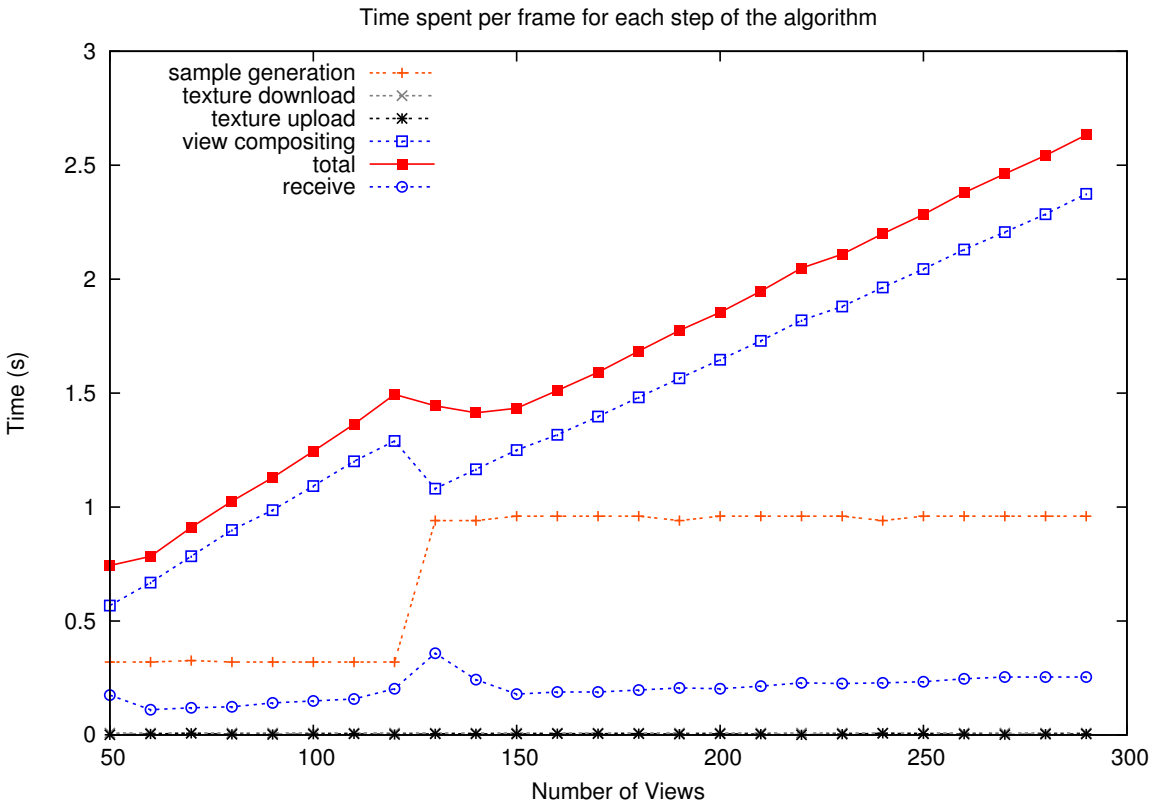


Figura 6.4: Tempo gasto nas etapas principais do algoritmo, utilizando 5 GPUs

6.4 Qualidade da Renderização

Comparamos a qualidade da renderização fornecida pelo renderizador de *light field* com a obtida renderizando-se diretamente cada ponto de vista a partir da geometria. Idealmente, ambas as imagens seriam iguais; entretanto, a fidelidade depende da amostragem utilizada. A qualidade da imagem depende também do número de passos utilizados para refinar a interseção com a geometria. Para os testes de qualidade utilizamos 100 passos lineares.

Uma comparação entre a saída do nosso método e uma renderização direta da geometria pode ser vista na Figura 6.8. As diferenças visíveis foram marcadas em vermelho. Podemos ver que a diferença é mais significativa nas posições onde ocorre maior descontinuidade na profundidade. Isso faz com que o erro na discretização na determinação da profundidade leve a erros mais visíveis. O mesmo problema aparece ao se amostrar superfícies finas (ver Figura 6.9). A melhor solução para esses casos envolve aumentar o número de passos utilizados para determinar a intersecção com a superfície. Note que o modelo original O modelo original Figura 6.9 a) contêm alguns pontos pretos. Eles não representam um problema da técnica.

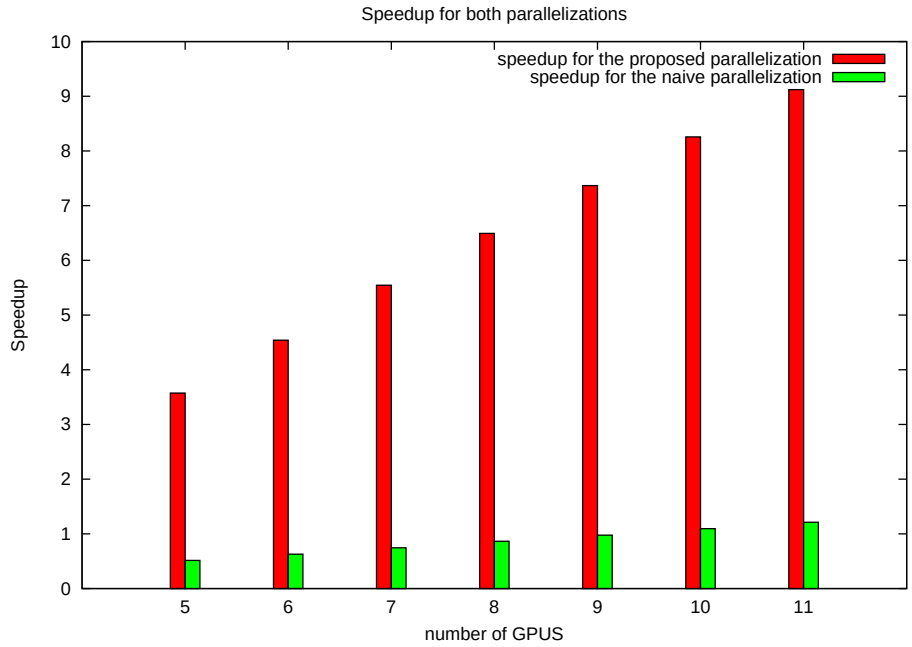


Figura 6.5: Speedup da paralelização proposta e da paralelização trivial para uma carga de 1000 pontos de vista.

6.5 Discussão

Nesse capítulo foram apresentados os resultado dos experimentos executados para avaliar o desempenho e a qualidade de saída da arquitetura proposta. Observamos que o resultado dos experimentos foi coerente com as análises teóricas executadas. Em especial, notamos que, como esperado, o tempo de comunicação não foi relevante para as configurações estudadas.

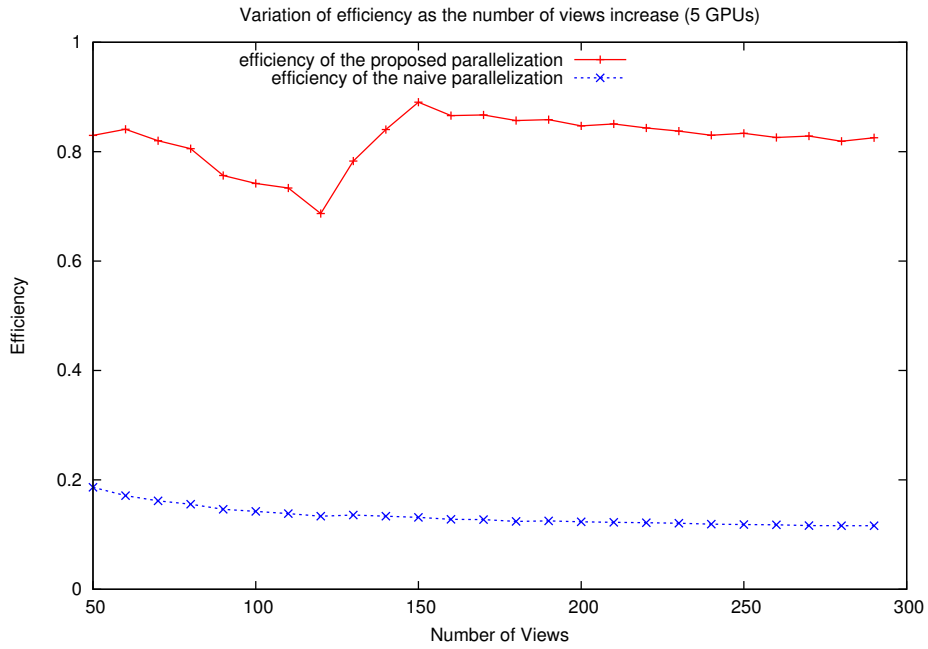


Figura 6.6: Escalabilidade da paralelização trivial e da paralelização proposta

Figura 6.7: Imagem gerada pelo renderizador proposto

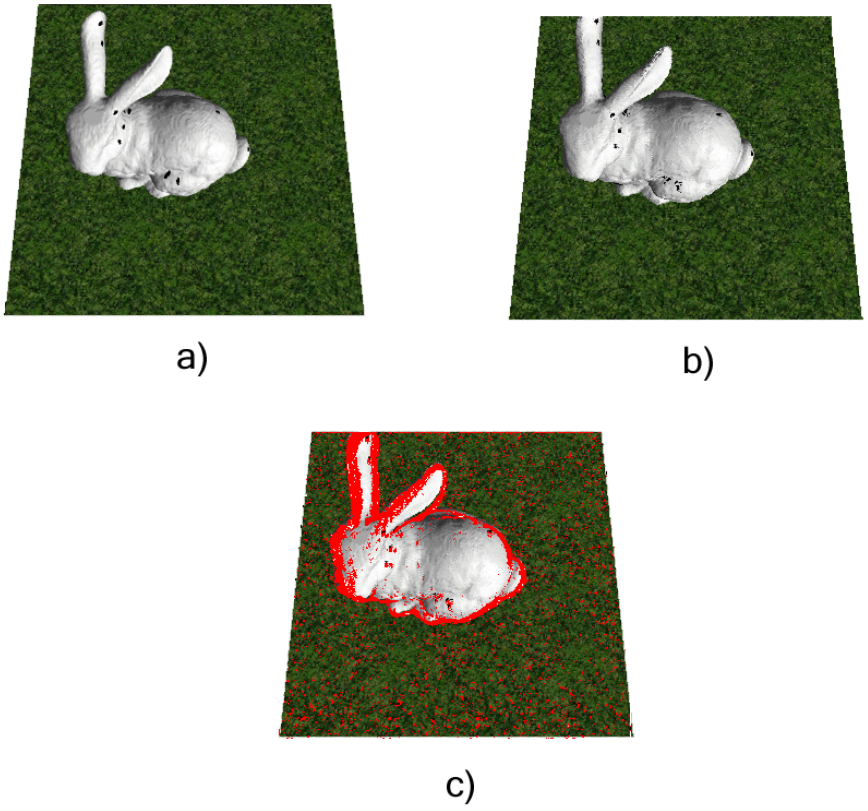


Figura 6.8: Qualidade da renderização. a) Renderização direto da geometria; b) Saída do renderizador de *light field.*, c) Diferença na percepção

Figura 6.9: Perda de qualidade devido aos erros na determinação da superfície. Detalhes muito finos como as folhas sofrem de *aliasing*.

Capítulo 7

Conclusão e Trabalhos Futuros

Neste trabalho apresentamos uma arquitetura para a renderização paralela de múltiplos pontos de vista em um *cluster* de GPUs. Mostramos a análise teórica de *speedup* e escalabilidade do sistema proposto e como ele pode ser superior aos de uma paralelização trivial. Os resultados obtidos em um cluster de renderização real confirmam essa análise.

A ideia básica do trabalho foi utilizar um renderizador de *light field* para evitar o trabalho duplicado entre os pontos de vista. Ao mesmo tempo, propomos uma paralelização eficiente que permite que o sistema escale bem quanto ao número de vistas.

O principal compromisso do sistema é relacionado ao processo de amostragem, que pode ser complexo para diversas cenas. Como trabalho futuro, gostaríamos de investigar o posicionamento adaptativo das câmeras, para que as cenas possam ser amostradas de maneira mais eficiente.

Outra área de trabalho é a otimização geral do sistema, utilizando primitivas para transferência mais rápida de textura, balanceamento de carga dinâmico e uma composição mais rápida, por meio de estratégias otimizadas de busca do ponto de interseção.

Por fim, gostaríamos de testar o desempenho da paralelização em um framework voltado para o processamento de *streams*, como o Anthill (Ferreira et al., 2005).

Apêndice A

Código GLSL para o renderizador de light field

A.1 Vertex Shader Para o Cálculo do Alpha

Algorithm 3 Vertex Shader Amostragem

```
1: vec4 pv = gl_ModelViewMatrix * gl_Vertex;
2: vec4 p0 = gl_ModelViewMatrix * CamPos;
3: //intersection with the plane z=0 and z=1
4: vec4 u = normalize(pv-p0);
5: vec4 n = gl_ModelViewMatrix * vec4(0,0,1,0);
6: vec4 w = p0 - (gl_ModelViewMatrix * vec4(0,0,0,1));
7: vec4 w1 = p0 - (gl_ModelViewMatrix * vec4(0,0,1,1));
8: float s = dot(-n,w)/dot(n,u);
9: float s1 = dot(-n,w1)/dot(n,u);
10: vec4 Ps = p0 + s*u;
11: vec4 Ps1 = p0 + s1*u;
12: //ratio: vertex_distance_to_camera/distance_to_plane_z0
13: d = length(pv-Ps1)/length(Ps-Ps1);
14: gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
15: normal = gl_Normal;
16: gl_TexCoord[0] =gl_MultiTexCoord0;
```

A.2 Vertex Shader Para o Renderizador

Algorithm 4 Vertex Shader Renderizador

```
1: varying vec4 VertexPos;
2: //camera position
3: uniform vec4 CamPos;
4: //return a matrix that project a vertex at world
5: //space into the camera positioned at the point
6: //specified cpos, looking to world origin (0,0,0)
7: mat4 ProjectAtCam(in vec3 cpos,in vec3 origin)
8:
9: vec3 d = -normalize(cpos);
10: vec3 u = vec3(-1,0,0);
11: vec3 c = cross(d,u);
12: vec3 nu = cross(d,c);
13: mat4 rot = mat4(1);
14: rot[0] = vec4(c,0);
15: rot[1]= vec4(nu,0);
16: rot[2]= vec4(-d,0);
17: rot = transpose(rot);
18: mat4 offset = mat4(1);
19: offset[3]= vec4(-cpos-origin,1);
20: return(rot * offset);
21:
22: void main(void)
23:
24: //User ProjectionView camera
25: mat4 ProjectViewMatrix =gl_ProjectionMatrix *
26: ProjectAtCam(vec3(CamPos),vec3(0,0,1));
27: gl_Position = ProjectViewMatrix * ( gl_Vertex );
28: gl_FrontColor = gl_Color;
29: VertexPos = ( gl_Vertex ) +vec4(0,0,-1,0) ;
30:
```

A.3 Pixel Shader Para o Renderizador

Algorithm 5 Pixel Shader Renderizador (variáveis globais)

```
1: uniform sampler2D Tex0;
2: varying vec4 VertexPos;
3: //offset of the data for each camera
4: uniform vec2 OffsetR;
5: uniform vec2 OffsetG;
6: uniform vec2 OffsetB;
7: uniform vec2 OffsetA;
8: //Camera array
9: uniform vec4 CamPosR;
10: uniform vec4 CamPosG;
11: uniform vec4 CamPosB;
12: uniform vec4 CamPosA;
13: uniform vec4 CamArrayOrigin;
14: //camera position
15: uniform vec4 CamPos;
16: //used to return ray intersection with the
17: //planes s1 (z=1) e s (z=0)
18: struct Plane
19:
20: vec4 s;
21: vec4 s1;
22: ;
```

Algorithm 6 Pixel Shader Renderizador

```
1: // Find intersection of the ray P1-cpos with the
2: //planes z=1 (s1) e z=0 (s)
3: float findPlaneIntersection(out Plane plane, in vec4 cpos,in vec4 P1)
4:
5: vec4 u = normalize(P1 - cpos);
6: vec4 n = vec4(0,0,1,0);
7: vec4 w = cpos - (vec4(0,0,0,1));
8: vec4 w1 = cpos - ( vec4(0,0,1,1));
9: float s = dot(-n,w)/dot(n,u);
10: float s1 = dot(-n,w1)/dot(n,u);
11: plane.s = cpos + s*u;
12: plane.s1 = cpos + s1*u;
13: return 1.0;
14:
15: //return a matrix that project a vertex at world
16: //space into the camera positioned at the point
17: //specified cpos, looking to origin
18: mat4 ProjectAtCam(in vec3 cpos,in vec3 origin)
19:
20: vec3 d = -normalize(cpos);
21: vec3 u = vec3(0,1,0);
22: vec3 c = cross(d,u);
23: vec3 nu = cross(c,d);
24: mat4 rot = mat4(1);
25: rot[0] = vec4(c,0);
26: rot[1]= vec4(nu,0);
27: rot[2]= vec4(-d,0);
28: rot = transpose(rot);
29: mat4 offset = mat4(1);
30: offset[3]= vec4(-cpos-origin,1);
31: return(gl_ProjectionMatrix * rot * offset);
32:
```

Algorithm 7 Vertex Shader Renderizador (computepixel)

```
1: float computePixel(
2: in mat4 camProj,
3: in vec2 Offset,
4: in vec4 CamPos,
5: in vec4 P,
6: in float depth_step,
7: out vec4 pixel)
8:
9: vec4 C; //point projected into camera k
10: vec4 Ckp; //position as seen by camera k
11: float delta; //distance between Ck and Ckp
12: Plane plane; //ray intersection with scene planes
13: //1 - project the current point into the camera camProj
14: C = camProj * P;
15: //2 - find the 2d coordinates of the projected point
16: // Ckd, considering texture scale and w
17: vec2 coord = vec2((1.0+C.x/C.w)/2.0, (1.0+C.y/C.w)/8.0);
18:
19: //3-read color and z information
20: coord += Offset;
21: coord = vec2(1,0)-coord;
22: pixel = texture2D(Tex0,coord);
23: //4 - calculate the space position as seen by
24: // the camera (Ckp)
25: findPlaneIntersection(plane,CamPos,P);
26: Ckp = plane.s * pixel.a +(1.0-pixel.a)*plane.s1;
27: //5 - see if point is inside the surface
28: //distance from the cam to the current point
29: float dpCam = length(P-CamPos);
30: //distance from the cam to the surface
31: float dckpCam = length(Ckp-CamPos);
32: // write the distance to the surface on alpha value
33: pixel.a = dckpCam -dpCam;
34: //if point inside surface
35: if(pixel.a < 0.0)
36: return 1.0;
37:
38: return 0.0;
39:
```

Algorithm 8 Vertex Shader Renderizador (intersect)

```
vec3 intersect(in vec4 viewVec)

float depth_step, dis,best_depth;
vec4 enable;
int i;
vec4 P; //current point
Plane plane; //intersection of the interpolated ray with scene planes
//to store colors and distance to surface;
vec4 pixelR,pixelG,pixelB,pixelA;
//compose the projection matrix for each camera
mat4 camprojR = ProjectAtCam(vec3(CamPosR),vec3(0,0,0));
mat4 camprojG = ProjectAtCam(vec3(CamPosG),vec3(0,0,0));
mat4 camprojB = ProjectAtCam(vec3(CamPosB),vec3(0,0,0));
mat4 camprojA = ProjectAtCam(vec3(CamPosA),vec3(0,0,0));
// 1- walks along the current viewRay P = current position
//find size of the ray segment between s and s1
findPlaneIntersection(plane,CamPos,VertexPos);
float rayd = length(plane.s - plane.s1);
depth_step = rayd/100.0;
dis = depth_step;
float sum;
enable = vec4(0.0,0.0,0.0,0.0);
//proceed in linear steps
for(i=0; i<100; i++ )

//walks from the ray intersection with the z=1 plane
P = plane.s1 + viewVec*dis;
enable.r = computePixel(camprojR,OffsetR,CamPosR,
P,depth_step,pixelR);
enable.g = computePixel(camprojG,OffsetG,CamPosG,
P,depth_step,pixelG);
enable.b = computePixel(camprojB,OffsetB,CamPosB,
P,depth_step,pixelB);
enable.a = computePixel(camprojA,OffsetA,CamPosA,
P,depth_step,pixelA);
if((enable.r + enable.g + enable.b + enable.a)>3.0)
break;
dis+=depth_step;

float max = dis;
float min = dis - depth_step;
//find out the pixel color to use
vec4 outpixel = vec4(0.0,0.0,0.0,0.0);
if((pixelR.a >= pixelG.a) &&( pixelR.a >= pixelB.a)&&( pixelR.a >= pixelA.a))
outpixel = pixelR + vec4(0.0,0.0,0.0,0.0);
if((pixelG.a >= pixelR.a) &&( pixelG.a >= pixelB.a)&&( pixelG.a >= pixelA.a))
outpixel = pixelG + vec4(0.0,0.0,0.0,0.0);
if((pixelB.a >= pixelR.a) &&( pixelB.a >= pixelG.a)&&( pixelB.a >= pixelA.a))
outpixel = pixelB+vec4(0.0,0.0,0.0,0.0);
if((pixelA.a >= pixelR.a) &&( pixelA.a >= pixelG.a)&&( pixelA.a >= pixelB.a))
outpixel = pixelA + vec4(0.0,0.0,0.0,0.0);
return vec3(outpixel);

```

Algorithm 9 Vertex Shader Renderizador (main)

```
1: void main(void)
2:
3: //current view matrix
4: mat4 ViewMatrix = ProjectAtCam(vec3(CamPos),vec3(CamArrayOrigin));
5: vec3 color;
6: vec4 view;
7: //view vector (world space)
8: view = VertexPos - CamPos;
9: view = normalize(view);
10: //pixel color
11: color = intersect(view);
12: gl_FragColor = vec4(color.x,color.y,color.z,1);
13:
```

Referências Bibliográficas

Adelson, E. H. e Bergen, J. R. (1991). The plenoptic function and the elements of early vision. *Computational Models of Visual Processing*, pp. 3–20. (Citado na página 10.)

Akenine-Moller, T. e Haines, E. (2002). *Real-time Rendering*. A K Peters, Ltda, 2 edição edição. (Citado na página 8.)

Aliaga, D.; Cohen, J.; Wilson, A.; Baker, E.; Zhang, H.; Erikson, C.; Hoff, K.; Hudson, T.; Stuerzlinger, W.; Bastos, R.; Whitton, M.; Brooks, F. e Manocha, D. (1999a). Mmr: an interactive massive model rendering system using geometric and image-based acceleration. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pp. 199–206, New York, NY, USA. ACM Press. (Citado na página 19.)

Aliaga, D.; Cohen, J.; Wilson, A.; Baker, E.; Zhang, H.; Erikson, C.; Hoff, K.; Hudson, T.; Stuerzlinger, W.; Bastos, R.; Whitton, M.; Brooks, F. e Manocha, D. (1999b). Mmr: an interactive massive model rendering system using geometric and image-based acceleration. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pp. 199–206, New York, NY, USA. ACM. (Citado na página 24.)

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Proc. AFIPS Conf*, pp. 483–485. (Citado na página 24.)

Andujar; C.; Boo; J.; Brunet; P.; Fairen; M.; Navazo; I.; Vazquez; P.; Vinacua e A. (2007). Omni-directional relief impostors. *Computer Graphics Forum*, 26(3):553–560. (Citado na página 16.)

Annen, T.; Matusik, W.; Pfister, H. e Seidel, H.-P. Z. M. (2006). Distributed rendering for multiview parallax displays. Technical report, Mitsubishi Electric Research Laboratories. (Citado nas páginas 2 e 19.)

Beier, T. e Neely, S. (1992). Feature-based image metamorphosis. *SIGGRAPH Comput. Graph.*, 26(2):35–42. (Citado na página 14.)

Buehler, C.; Bosse, M.; McMillan, L.; Gortler, S. e Cohen, M. (2001). Unstructured lumigraph rendering. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 425–432, New York, NY, USA. ACM. (Citado nas páginas 12 e 40.)

Callahan, S. P.; Bavoil, L.; V., P. e Silva, C. T. (2006). Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5). (Citado na página 20.)

Camahort, E.; Lerios, A. e Fussell, D. (1998). Uniformly sampled light fields. In Drettakis, G. e Max, N., editores, *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop '98)*, pp. 117–130, New York, NY. Springer Wien. (Citado nas páginas 12 e 40.)

Catmull, E. (1974). *A Subdvision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah. (Citado na página 10.)

Cedilnik, A.; Geveci, B.; Moreland, K.; Ahrens, J. e Favre, J. (2006). Remote large data visualization in the paraview framework. *Eurographics Symposium on Parallel Graphics and Visualization*, 6:163–171. (Citado na página 19.)

Chai, J.-X.; Chan, S.-C.; Shum, H.-Y. e Tong, X. (2000). Plenoptic sampling. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 307–318, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co. (Citado na página 17.)

Chen, S. E. e Williams, L. (1993). View interpolation for image synthesis. *Computer Graphics*, 27(Annual Conference Series):279–288. (Citado na página 14.)

Crockett, T. W. (1995). Parallel rendering. Icase report 95-31; nasa cr-195080, NASA Langley Research Center Hampton. (Citado na página 23.)

Daly, S. (1993). The visible differences predictor: an algorithm for the assessment of image fidelity. pp. 179–206. (Citado na página 47.)

Debevec, P. E.; Taylor, C. J. e Malik, J. (1996). Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 11–20, New York, NY, USA. ACM. (Citado na página 24.)

Dodgson, N. (2005). Autostereoscopic 3d displays. *Computer*, 38(8):31–36. (Citado na página 2.)

Eager, D. L.; Zahorjan, J. e Lozowska, E. D. (1989). Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423. (Citado na página 26.)

Eilemann, S. e Pajarola, R. (2007). Direct send compositing for parallel sort-last rendering. In Favre, J. M.; Santos, L. P. e Reiners, D., editores, *Eurographics Symposium on Parallel Graphics and Visualization*, pp. 29–36, Lugano, Switzerland. The Eurographics Association 2007. (Citado na página 22.)

Ferreira, R.; Meira, W., J.; Guedes, D.; Drummond, L.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araujo, R. e Ferreira, G. (2005). Anthill: a scalable run-time environment for data mining applications. *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, pp. 159–166. (Citado na página 56.)

Geer, D. (2005). Taking the graphics processor beyond graphics. *Computer*, 38(9):14–16. (Citado na página 5.)

Geist, G. A.; Kohla, J. A. e Papadopoulos, P. M. (1996). PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150. (Citado nas páginas 26 e 27.)

Gortler, S. J.; Grzeszczuk, R.; Szeliski, R. e Cohen, M. F. (1996). The lumigraph. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 43–54, New York, NY, USA. ACM. (Citado nas páginas 6 e 12.)

Gropp, W. (2001). Learning from the success of mpi. In *HiPC '01: Proceedings of the 8th International Conference on High Performance Computing*, pp. 81–94, London, UK. Springer-Verlag. (Citado na página 27.)

Gropp, W. e Lusk, E. (2002). Goals guiding design: Pvm and mpi. *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 257–265. (Citado na página 26.)

Gustafson, J. (1992). The consequences of fixed time performance measurement. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 2, pp. 113–124. Ames Lab., Iowa State Univ.,. (Citado nas páginas vii e 25.)

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533. (Citado nas páginas 24 e 25.)

Halle, M. (1998). Multiple viewpoint rendering. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 243–254, New York, NY, USA. ACM. (Citado nas páginas 2, 3, e 18.)

Hasselgren, J. e Akenine-Möller, T. (2006). An efficient multi-view rasterization architecture. In Akenine-Möller, T. e Heidrich, W., editores, *Eurographics Workshop/ Symposium on Rendering*, pp. 61–72, Nicosia, Cyprus. Eurographics Association. (Citado nas páginas 2, 3, e 18.)

Heidrich, W.; Schirmacher, H.; Kück, H. e Seidel, H.-P. (1999). A warping-based refinement of lumigraphs. In Thalmann, N. e Skala, V., editores, *Proc. WSCG '99*. (Citado nas páginas 13 e 40.)

Hübner, T.; Zhang, Y. e Pajarola, R. (2006). Multi-view point splatting. In Lee, Y. T.; Shamsuddin, S. M. H.; Gutierrez, D. e Suaib, N. M., editores, *GRAPHITE*, pp. 285–294. ACM. (Citado nas páginas 2 e 18.)

Ihm, I.; Park, S. e Lee, R. K. (1997). Rendering of spherical light fields. In *PG '97: Proceedings of the 5th Pacific Conference on Computer Graphics and Applications*, p. 59, Washington, DC, USA. IEEE Computer Society. (Citado nas páginas 12, 18, e 40.)

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, INC, 1st edição. (Citado nas páginas 24 e 26.)

Jeschke, S.; Wimmer, M. e Schuman, H. (2002). Layered Environment-Map Impostors for Arbitrary Scenes. In *Proc. Graphics Interface*, pp. 1–8. (Citado na página 19.)

Jeschke, S.; Wimmer, M.; Schumann, H. e Purgathofer, W. (2005). Automatic impostor placement for guaranteed frame rates and low memory requirements. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 103–110, New York, NY, USA. ACM. (Citado na página 18.)

Jones, A.; McDowall, I.; Yamada, H.; Bolas, M. e Debevec, P. (2007). Rendering for an interactive 360 light field display. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, p. 40, New York, NY, USA. ACM. (Citado na página 2.)

Lengyel, J. (1998). The convergence of graphics and vision. *Computer*, 31(7):46–53. (Citado nas páginas vii e 11.)

Levoy, M. e Hanrahan, P. (1996). Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 31–42, New York, NY, USA. ACM. (Citado nas páginas vii, 6, 12, e 40.)

Levoy, M. e Whitted, T. (1985). The use of points as a display primitive. Technical Report Technical Report 85-022, University of North Carolina at Chapel Hill. (Citado na página 18.)

Lin, Z. e Shum, H. (2000). On the number of samples needed in light field rendering with constant-depth assumption. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR-00)*, pp. 588–597, Los Alamitos. IEEE. (Citado na página 17.)

Lin, Z. e Shum, H.-Y. (2004). A geometric analysis of light field rendering. *Int. J. Comput. Vision*, 58(2):121–138. (Citado na página 17.)

Mark, W. R.; McMillan, L. e Bishop, G. (1997). Post-rendering 3d warping. In *SI3D*, pp. 7–16, 180. (Citado na página 15.)

Matusik, W. e Pfister, H. (2004). 3d tv: A scalable system for real-time acquisition, transmission, and autostereoscopic display of dynamic scenes. *ACM Transactions on Graphics*, 23:814–824. (Citado na página 2.)

McMillan, L. (1997). *An Image-Based Approach to Three-Dimensional Computer Graphics*. PhD thesis, University of North Carolina at Chapel Hill. (Citado nas páginas 14 e 42.)

McMillan, L. e Bishop, G. (1995). Plenoptic modeling: An image-based rendering system. *Computer Graphics*, 29(Annual Conference Series):39–46. (Citado nas páginas 11, 12, e 40.)

Mei, C.; Popescu, V. e Sacks, E. (2005). The occlusion camera. *Computer Graphics Forum*, 24(3):335–342. (Citado na página 15.)

Message Passing Forum (1994). Mpi: A message-passing interface standard. Technical report, Message Passing Forum, Knoxville, TN, USA. (Citado na página 26.)

Molnar, S.; Cox, M.; Ellsworth, D. e Fuchs, H. (1994). A sorting classification of parallel rendering. Technical Report TR94-023, University of North Carolina at Chapel Hill and Princeton University. (Citado na página 21.)

Moreland, K.; Wylie, B. e Pavlakos, C. (2001). Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pp. 85–92, Piscataway, NJ, USA. IEEE Press. (Citado na página 22.)

Nehab, D.; Sander, P. V. e Isidoro, J. (2006). The real-time reprojection cache. Technical Report TR-749-06. (Citado na página 16.)

Neto, M. O. e Bishop, G. (1999). Image-based objects. In *Proceedings of 1999 ACM Symposium on Interactive 3D Graphics*. (Citado na página 17.)

Oliveira, M. e Bishop, G. (1999). Relief textures. Technical Report TR99-015, University of North Carolina at Chapel Hill. (Citado na página 15.)

Oliveira, M. M. (2002). Image-based modeling and rendering techniques: A survey. *RITA - Revista de Informática Teórica e Aplicada*, IX(2):37–66. (Citado nas páginas vii, 11, e 13.)

Oliveira, M. M.; Bishop, G. e McAllister, D. (2000). Relief texture mapping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 359–368, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co. (Citado nas páginas vii, 15, e 16.)

Pfister, H.; Zwicker, M.; van Baar, J. e Gross, M. (2000). Surfels: surface elements as rendering primitives. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 335–342, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co. (Citado na página 18.)

Policarpo, F. e Oliveira, M. M. (2006). Relief mapping of non-height-field surface details. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 55–62, New York, NY, USA. ACM. (Citado na página 16.)

Policarpo, F.; Oliveira, M. M. e Comba, J. L. D. (2005). Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph.*, 24(3):935–935. (Citado nas páginas vii, 15, 16, e 17.)

Popescu, V. e Aliaga, D. G. (2006). The depth discontinuity occlusion camera. In Olano, M. e Séquin, C. H., editores, *SI3D*, pp. 139–143. ACM. (Citado na página 15.)

Popescu, V.; Lastra, A. e Eyles, J. (2000). Sort-first parallelism for image-based rendering. (Citado na página 21.)

Ramanarayanan, G.; Ferwerda, J.; Walter, B. e Bala, K. (2007). Visual equivalence: towards a new standard for image fidelity. *ACM Trans. Graph.*, 26(3):76. (Citado na página 48.)

Ramasubramanian, M.; Pattanaik, S. N. e Greenberg, D. P. (1999). A perceptually based physical error metric for realistic image synthesis. In Rockwood, A., editor, *Siggraph 1999, Computer Graphics Proceedings*, pp. 73–82, Los Angeles. Addison Wesley Longman. (Citado na página 48.)

Risser, E. (2007). Rendering 3d volumes using per-pixel displacement mapping. In *Sandbox '07: Proceedings of the 2007 ACM SIGGRAPH symposium on Video games*, pp. 81–87, New York, NY, USA. ACM. (Citado na página 16.)

Rosen, P. e Popescu, V. (2008). The epipolar occlusion camera. In Haines, E. e McGuire, M., editores, *SI3D*, pp. 115–122. ACM. (Citado na página 15.)

Roth, M. e Reiners, D. (2006). Sorted pipeline image composition. *EGPGV06 - Eurographics Symposium on Parallel Graphics and Visualization*, pp. 119–126. (Citado na página 22.)

Samanta, R.; Funkhouser, T.; Li, K. e Singh, J. (2000). Hybrid sortfirst and sort-last parallel rendering with a cluster of pcs. (Citado nas páginas 5 e 23.)

Sawhney, H. S. (1994). 3d geometry from planar parallax. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. (Citado na página 15.)

Schirmacher, H.; Heidrich, W. e Seidel, H.-P. (1999). Adaptive acquisition of lumigraphs from synthetic scenes. In Brunet, P. e Scopigno, R., editores, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pp. 151–160. The Eurographics Association and Blackwell Publishers. (Citado na página 18.)

Schirmacher, H.; Vogelgsang, C.; Seidel, H.-P. e Greiner, G. (2001). Efficient free form light field rendering. (Citado nas páginas 12, 18, e 40.)

Shade, J.; Gortler, S.; wei He, L. e Szeliski, R. (1998). Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 231–242, New York, NY, USA. ACM. (Citado na página 17.)

Shade, J.; Lischinski, D.; Salesin, D. H.; DeRose, T. e Snyder, J. (1996). Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 75–82, New York, NY, USA. ACM. (Citado na página 18.)

Shum, H. e Kang, S. (2000). A review of image-based rendering techniques. In *VCIP00*, pp. 2–13. (Citado nas páginas 11 e 12.)

Shum, H.-Y. e He, L.-W. (1999). Rendering with concentric mosaics. *Computer Graphics*, 33(Annual Conference Series):299–306. (Citado nas páginas vii, ix, 12, e 13.)

Sillion, F.; Drettakis, G. e Bodelet, B. (1997). Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum*, 16(3):C207–C218. (Citado na página 20.)

Sloan, P.-P. e Hansen, C. (1999). Parallel lumigraph reconstruction. In *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, pp. 7–14, Washington, DC, USA. IEEE Computer Society. (Citado na página 23.)

Stewart, J.; Bennett, E. e McMillan, L. (2004). Pixelview: A view-independent graphics rendering architecture. In Akenine-Möller, T. e McCool, M., editores, *in: proc of Graphics Hardware*, pp. 75–84. (Citado nas páginas 2, 3, e 18.)

Strasser, J.; Pascucci, V. e Ma, K.-L. (2006). Multi-layered image caching for distributed rendering of large multiresolution datasets. In *Eurographics Symposium on Parallel Graphics and Visualization*, pp. 171–177, Braga, Portugal. Eurographics Association. (Citado na página 23.)

Sun, X.-H. (1990). *Parallel Computation Models: Representation, Analysis and Application*. PhD thesis, Computer Science Department, Michigan State University. (Citado nas páginas 25 e 26.)

Sun, X.-H. e Ni, L. M. (1990). Another view on parallel speedup. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pp. 324–333, Los Alamitos, CA, USA. IEEE Computer Society Press. (Citado na página 24.)

Sunderam, V. S. (1990). Pvm: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339. (Citado na página 26.)

Takahashi, K. e Naemura, T. (2005). Unstructured light field rendering using on-the-fly focus measurement. In *ICME*, pp. 205–208. IEEE. (Citado nas páginas 12 e 40.)

Todt, S.; Rezk-Salama, C. e Kolb, A. (2007). Fast (spherical) light field rendering with per-pixel depth. Technical report, University of Siegen. (Citado nas páginas 6, 12, 18, 40, 42, e 43.)

Toledo, R. e Levy, B. (2004). Extending the graphic pipeline with new gpu-accelerated primitives. In *International gOcad Meeting, Nancy, France*. Also presented in Visgraf Seminar 2004, IMPA, Rio de Janeiro, Brazil. (Citado na página 7.)

Vogelgsang, C. e Greiner, G. (2000). Adaptive lumigraph rendering with depth maps. Technical Report 3, IMMD 9, Universitaet Erlangen-Nuernberg. (Citado na página 40.)

Wilson, A. e Manocha, D. (2003). Simplifying complex environments using incremental textured depth meshes. *ACM Trans. Graph.*, 22(3):678–688. (Citado na página 24.)

Yang, J. C.; Everett, M.; Buehler, C. e McMillan (2002). A real-time distributed light field camera. In *In: proc. of the 13th Eurographics workshop on Rendering, Italy*, pp. 77–86. (Citado nas páginas 6, 18, e 19.)

Yee, Y. H. e Newman, A. (2004). A perceptual metric for production testing. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, p. 121, New York, NY, USA. ACM. (Citado na página 48.)