UNIVERSIDADE FEDERAL FLUMINENSE

THALES LUIS RODRIGUES SABINO

UMA ARQUITETURA DE PIPELINE HÍBRIDA PARA RASTERIZAÇÃO E TRAÇADO DE RAIOS EM TEMPO REAL

NITERÓI

2012

UNIVERSIDADE FEDERAL FLUMINENSE

THALES LUIS RODRIGUES SABINO

UMA ARQUITETURA DE PIPELINE HÍBRIDA PARA RASTERIZAÇÃO E TRAÇADO DE RAIOS EM TEMPO REAL

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Computação Visual

Orientador:
ESTEBAN WALTER GONZALEZ CLUA

NITERÓI

2012

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

S116 Sabino, Thales Luis Rodrigues
Uma arquitetura de pipeline híbrida para rasterização e traçado de raios em tempo real / Thales Luis Rodrigues Sabino. – Niterói, RJ : [s.n.], 2012.
66 f.

Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2012.
Orientador: Esteban Walter Gonzalez Clua.

1. Traçado de raios. 2. Jogo tridimensional. 3. Computação gráfica. 4. Tempo real. 5. Unidade de processamento gráfico. I. Título.

CDD 006.693

THALES LUIS RODRIGUES SABINO

UMA ARQUITETURA DE PIPELINE HÍBRIDA PARA RASTERIZAÇÃO E TRAÇADO DE RAIOS EM TEMPO REAL

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Computação Visual

Aprovada em Junho de 2012.

BANCA EXAMINADORA

Prof. Esteban Walter Gonzalez Clua - Orientador, UFF

Prof. Leandro Augusto Frata Fernandes, UFF

Prof. Manuel Menezes de Oliveira Neto, UFRGS

Niterói

2012

Agradecimentos

À Deus, por me dar forças para perseguir meus objetivos.

Aos meus pais, Eva e Luis, e minha família, pelo apoio incondicional em todos esses meus anos de vida. Agradecimento especial para minha irmã Thársila, que me recebeu em sua casa em vários fins de semana.

À Julia, que de um momento para outro se tornou uma pessoa muito especial em minha vida.

Ao professor Esteban, pela orientação e ajuda durante todo o tempo de mestrado.

Ao Instituto de Computação da UFF e todos os seus professores e funcionários, pela oportunidade de desenvolver meus estudos.

Aos amigos do MediaLab, por fazer do ambiente de trabalho um lugar divertido de se estar.

Também a todos os grandes amigos que conheci durante o tempo de faculdade: Tássio Knop, Marco Aurélio, Daniel Madeira, João Paulo Peçanha, Eder Perez e João Paulo Scoralick.

Agradeço a todas as pessoas que, direta ou indiretamente contribuíram com esse trabalho e cujos nomes não estão aqui.

Ao CNPq, pela ajuda financeira, sem a qual não poderia ter realizado este trabalho.

Resumo

A renderização em jogos 3D normalmente utiliza técnicas de rasterização com o objetivo de garantir uma taxa de quadros interativa, pois, o traçado de raios, mesmo sendo uma técnica superior no que se refere a geração de imagens foto-realistas, possui um custo computacional elevado. Com o advento dos processadores massivamente paralelos na forma de unidades de processamento gráfico (GPUs), traçado de raios paralelo tem sido investigado como uma alternativa viável a técnicas de rasterização. Enquanto muitos trabalhos apresentam métodos de paralelização para o algoritmo clássico de traçado de raios, neste trabalho é apresentada uma técnica híbrida de rasterização e traçado de raios completamente executada em GPU. Enquanto um modelo de *deferred rendering* determina a iluminação de raios primários, um estágio posterior de traçado de raios computa outros efeitos como reflexão especular e transparência. O último estágio consiste na composição das duas imagens geradas de forma a obter uma imagem rasterizada composta com reflexos, refrações e sombras raio-traçadas, efeitos que não são trivialmente antingidos com rasterização.

Palavras-chave: Traçado de raios, Rasterização, OptiX, CUDA, GPU, Renderização Híbrida, OpenGL, GLSL, Tempo-Real, efeitos de iluminação global, *deferred shading*.

Abstract

Rendering in 3D games typically uses rasterization approaches in order to guarantee interactive frame rates, since ray tracing, a superior technique for photorealistic rendering, has a greater computational cost. With the advent of massively parallel processors in the form of GPUs, parallelized ray tracing has been investigated as a viable alternative to rasterization techniques. While many works present parallelization methods for the classical ray tracing algorithm, in this work we present a rasterized and ray traced hybrid technique, completely done in GPU. While a deferred render model determines the shading of primary rays, a ray tracing phase compute other effects such as specular reflection and transparency, in order to achieve effects that are not trivially obtained with rasterization. The last stage consists in composition of the two generated images in order to obtain a rasterized image composed with ray traced reflections, refractions and shadows, effects that are not trivilly obtained with rasterization.

Keywords: Ray Tracing, Rasterization, OptiX, CUDA, GPU, Hybrid Rendering, OpenGL, GLSL, Real-Time, Global Illumination Effects, Deferred Shading.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

RTRT	:	*Real-Time Ray Tracing*;
GPU	:	*Graphics Processing Units*;
RTRT	:	*Real-Time Ray Tracing*;
CUDA	:	*Compute Unified Device Architecture*;
OpenGL	:	*Open Graphics Library*;
GLSL	:	*OpenGL Shading Language*;
HLSL	:	*High Level Shading Language*;
REYES	:	*Renders Everything You Ever Saw*;
BVH	:	*Bounding Volume Hierarchies*;
SBVH	:	*Split Bounding Volume Hierarchies*;
HLBVH	:	*Hierachical Linear Bounding Volume Hierarchies*;
VLSI	:	*Very-Large-Scale Integration*;
G-Buffer	:	*Geometric Buffer*;
MRT	:	*Multiple Render Targets*;
API	:	*Application Program Interface*;
Pixel	:	*Picture Element*;
ASSIMP	:	*Open Asset Import Library*;
DevIL	:	*Developer's Image Library*;
Texel	:	*Texture Element*;
PTX	:	*Parallel Thread Execution*;
SDK	:	*Software Development Kit*;
NVCC	:	*NVIDIA C Compiler*;
QPS	:	Quadros por Segundo;
DR	:	*Deferred Rendering Time*;
RT	:	*Ray Tracing Time*;
BSP	:	*Binary Space Partitioning*;
SIMD	:	*Single Instruction Multiple Data*;
BRDF	:	*Bidirectional reflectance distribution function*

Sumário

Capítulo 1

Introdução

No campo da computação gráfica, as técnicas de rasterização foram desenvolvidas de maneira a suprir a demanda de gráficos em tempo real, enquanto, o traçado de raios sempre foi mais utilizado para geração de imagens foto-realistas. Dentre as várias áreas relativas à computação gráfica, estamos interessados, neste trabalho, na renderização em tempo real de cenas fazendo uso de técnicas modernas de rasterização e uso eficiente de traçado de raios para obter efeitos que são difíceis, se não impossíveis, de serem resolvidos com o uso exclusivo da primeira, mantendo, entretanto, uma taxa interativa de geração de quadros.

O traçado de raios tem sido objeto de estudos intensos pela academia nas últimas duas décadas sendo uma técnica superior para síntese de imagens a outras abordagens, como a rasterização. Traçar o caminho da luz em cenas virtuais é, ao mesmo tempo, uma solução elegante e flexível para os problemas de renderização e pode ser implementado facilmente, o que contribuiu para a popularização da técnica. No entanto, o uso prático do traçado de raios permaneceu restrito a geração de imagens em taxas não interativas, mas que possuem um auto grau de realismo.

A rápida evolução das unidades de processamento gráfico (GPU), que também funcionam como processadores massivamente paralelos de proposito geral [28, 33], promoveram a possibilidade do traçado de raios em tempo real (RTRT) utilizando as GPUs para geração de imagens de uma maneira alternativa àquela para que foram originalmente projetadas.

O traçado de raios tem sido amplamente utilizado para renderização *offline* de gráficos de jogos e para pré-processamento de luz, tal como ocorre no jogo *Quake 2*. Os jogos também são frequentemente utilizados como casos de teste para traçado de raios em tempo real. Em 2005, uma versão completa do jogo *Quake 2* foi mostrada por Wächter

e Keller [27] utilizando como algoritmo principal de renderização o traçado de raios. Uma implementação de traçado de raios no console de videogame *PlayStation 3™* é demonstrada em Benthim et al [7].

A renderização de tempo real é uma área dominada pela rasterização, um método que pode ser grosseiramente descrito como o mapeamento de vértices de um espaço 3D em um plano 2D e a determinação dos *pixels* que fazem parte das primitivas formadas por esses vértices. Apesar da alta qualidade dos gráficos gerados em tempo real nos videogames modernos com o uso quase exclusivo da rasterização [26], muitos fenômenos ainda são difíceis ou impossíveis de modelar utilizando essa técnica. Dentre esses fenômenos estão a reflexão, refração e iluminação indireta. Estes e muitos outros fenômenos físicos são facilmente modelados com o uso de traçado de raios. O uso do traçado de raios para pré-processamento de iluminação indireta e mapas de reflexão produz bons resultados, mas possui a limitação de somente funcionar bem para fontes de luzes que são estáticas no tempo. No caso de objetos dinâmicos, a geração, em tempo real, de mapas de reflexão sofre com a degradação de desempenho da aplicação como um todo. Este tipo de abordagem se restringe a casos específicos e é mais utilizada para gerar mapas de cenas estáticas.

Nesta dissertação temos o objetivo geral de implementar técnicas modernas de rasterização de forma a obter uma imagem de boa qualidade em termos de realismo e incorporar o traçado de raios neste *pipeline* de forma a obter os efeitos de sombra, refração e reflexão em cenas rasterizadas. O objetivo principal deste trabalho é acrescentar uma etapa de traçado de raios neste pipeline sem alteração do que já é feito, de forma a manter um nível de interatividade aceitável para os critérios de tempo real. Como objetivos específicos temos:

- Implementação de um *pipeline* de rasterização utilizando a técnica de *deferred shading*;
- Utilizando os dados gerados pelo *deferred shading* determinar raios a serem traçados utilizando a API OptiX, uma API para construção de aplicações baseadas em traçado de raios em tempo real utilizando GPUs;
- Combinar a imagem gerada por *deferred shading* com o resultado gerado pelo traçador de raios de forma a obter uma imagem rasterizada com sombras, reflexos e refrações gerados por traçado de raios;
- Dar suporte a cenas animadas;

- Comparar o resultado obtido, em termos de desempenho, com um traçador de raios cujos raios primários são disparados a partir da câmera.

Renderizadores que utilizam o algoritmo de rasterização para geração de imagens não apresentam problemas para cenas animadas já que todos os vértices e fragmentos visíveis devem ser avaliados a cada quadro gerado. Traçadores de raio, normalmente, fazem uso de alguma estrutura de dados de aceleração que elimina testes de intersecção de raios desnecessários. Para cenas animadas tais estruturas devem ser modificadas ou inteiramente reconstruídas dado que ocorreu uma mudança na geometria de algum objeto. O suporte a cenas animadas é importante para justificar a incorporação de um estágio extra no *pipeline* de renderização de cenas interativas. Tais cenas devem reagir a estímulos de usuários e, normalmente, as reações são feitas na forma de elementos animados.

As principais contribuições alcançadas neste trabalho são a implementação de um *pipeline* de renderização em tempo real capaz de compor efeitos raio-traçados em imagens rasterizadas com o uso de *deferred shading*, totalmente executado em GPU além da possibilidade de adição de elementos dinâmicos. Como dinâmicos, são considerados os elementos que podem sofrer transformações ao longo do tempo ou aqueles que podem ser deformados com o uso de um esqueleto guia. A arquitetura proposta foi desenvolvida de forma que o traçado de raios fosse incluído no *pipeline* de renderização como um estágio extra, sem alteração do que é feito na rasterização, mas fazendo uso dos dados produzidos pela técnica de *deferred shading* para evitar traçar raios primários de câmera. Esta decisão de projeto torna possível a possibilidade de ativar ou não os efeitos raio-traçados além de acelerar a computação de efeitos raio-traçados.

Pretende-se que o traçador de raios híbrido desenvolvido neste trabalho possa ser empregado como ferramenta na busca de melhores gráficos de aplicações interativas, principalmente jogos, no quesito de gráficos mais realistas.

A principal dificuldade ao utilizar a técnica de traçado de raios juntamente com a rasterização, em uma aplicação de tempo real, é o requisito de geração de quadros de forma rápida. Para isso, é necessário extrair o máximo de desempenho de cada técnica utilizada. A técnica de *deferred shading*, apesar de seu alto desempenho para uma grande quantidade de fontes de luz, deve ser implementada com cuidado, pois, seus requisitos de banda de memória também são altos. Para realizar o traçado de raios em conjunto com *deferred shading* de forma eficiente e sem desperdício de memória, deve-se manter somente uma cópia dos dados geométricos da cena alocados em GPU. Estes dados devem estar disponíveis tanto para o contexto do rasterizador quanto para o contexto do traçador de

raios. Um dos desafios na implementação desse compartilhamento é que não se tem, a priori, informação sobre a cena a ser renderizada. Essa limitação levou ao desenvolvimento de um módulo de carga capaz de gerenciar quaisquer tipos cenas. Este módulo deve ser capaz de carregar malhas, materiais e texturas de quaisquer tamanhos, formas e valores. O traçador de raios *OptiX* possui sua própria representação interna de cenas. A conversão automática de um grafo de cena qualquer para o grafo de objetos utilizado pelo *OptiX* também foi desenvolvido de forma a tratar quaisquer tipos de cenas. Os desafios nas implementações dos módulos de carga e construção do grafo de objetos serão abordados com detalhes neste trabalho.

Esta dissertação está organizada da seguinte maneira: Após esta breve introdução, o Capítulo 2 mostra diferentes abordagens de renderização, como o *deferred shading* e traçado de raios. O Capítulo 3 mostra a teoria necessária para o desenvolvimento de um traçador de raios híbrido que faz uso dos dados armazenados no *G-Buffer*, um *buffer* para armazenamento de dados intermediários do processo de iluminação. A arquitetura proposta e implementada de um traçador de raios híbridos é mostrada no Capítulo 4 e detalhes de implementação serão descritos no Capítulo 5. O Capítulo 6 apresenta os resultados obtidos em termos de qualidade visual e desempenho. Finalmente, o Capítulo 7 apresenta as conclusões obtidas com o desenvolvimento deste trabalho além de propostas de trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Todas as técnicas de renderização, entre elas a rasterização e traçado de raios, tentam modelar o mesmo fenômeno físico: o espalhamento de luz sobre vários tipos de superfície. Apesar dos fenômenos simulados por esse modelo estarem bem documentados na literatura de transferência radiativa de calor, foi James T. Kajiya [25] quem apresentou o modelo em uma forma adequada para uso em computação gráfica, além de vários métodos de aproximação da solução.

O modelo mencionado é regido pela equação:

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right] \tag{2.1}$$

onde:

- $I(x, x')$ é relacionado com a intensidade de luz passante do ponto x' para o ponto x;
- $g(x, x')$ é um termo "geométrico";
- $\epsilon(x, x')$ é relacionado com a intensidade de luz emitida de x' para x;
- $\rho(x, x', x'')$ é relacionado com intensidade de luz espalhada de x'' para x por um pedaço de superfície x'.

Resumidamente, a equação estabelece que a intensidade de luz transportada de um ponto para outro é simplesmente a soma da luz emitida e da intensidade total de luz refletida na direcção x de todos os outros pontos.

2.1 *Deferred Shading*

Embora apenas recentemente a técnica de *deferred shading* tenha se tornado prática para aplicações de tempo real, essa técnica foi publica em 1988 por Michael Deering *et al* [14]. Neste trabalho os autores propuseram um sistema *VLSI* onde um *pipeline* de processadores de triângulos rasterizasse a geometria e então, um *pipeline* de processadores de iluminação aplicasse o modelo de *Phong* com múltiplas fontes de luz em tal região. Outro trabalho relevante sobre *deferred shading* foi desenvolvido por Saito e Takahashi em 1990 [40]. Este foi o primeiro trabalho a utilizar o termo *G-Buffer* para expressar valores intermediários no *pipeline* gráfico. Através do uso de *G-Buffers*, a aplicação de algoritmos de iluminação (colorização e mapeamento de textura) ficaria separado do processamento da geometria (projeção e remoção de superfícies oclusas).

Desde 2006, a técnica de *deferred shading* tem sido utilizada em uma grande quantidade de jogos eletrônicos devido à sua habilidade de resolução do problema de superfícies oclusas de maneira eficiente e complexidade de iluminação reduzida, o que permite um número elevado de fontes de luz nas cenas.

A iluminação de cenas em tempo real apresenta três opções comuns de escolha para os desenvolvedores:

- Passada única, múltiplas luzes;
- Múltiplas passadas, múltiplas luzes;
- *Deferred Shading*.

Cada uma das opções possui restrições associadas, a saber:

Passada única, múltiplas luzes: Dada uma cena a ser renderizada, a iluminação é calculada para cada objeto de maneira separada. Todas as fontes de luz que afetam um objeto são aplicadas em um único programa de *shader*:

Algoritmo 2.1 Passada única, múltiplas luzes

Para Cada Objeto **Faça**
  Renderizar objeto
  Aplicar todas as luzes em um único *shader*
Fim Para

Como todos os objetos da cena são renderizados, objetos oclusos (total ou parcialmente) potencialmente podem causar desperdício de tempo na aplicação do programa de

shader. A gerência de múltiplos tipos de fontes de luz é difícil de ser feita já que se torna necessária a escrita de milhares de combinações de programas de *shader* para um único modelo de iluminação. Tome-se como exemplo um jogo que suporta três tipos fontes de iluminação (pontual, direcional ou holofote): cada malha pode ser afetada por não mais do que seis fontes de luz simultâneas e é composta por um dentre cinco tipos de materiais. Neste caso relativamente simples, existem 420 combinações de luz/material que precisam ser codificadas em programas de *shader*. Este número cresce com cada adição de possibilidade - adicionando um quarto tipo de fonte de iluminação (por exemplo, uma luz com projeção de textura) o número de combinações aumenta para 1050. Em aplicações reais, este número é ainda maior. No jogo *Half-Life 2* da *Valve*, existem 1920 combinações de *shader* [2].

Múltiplas passadas, múltiplas luzes: Esta é outra solução para o problema de combinar luzes e materiais. A ideia é processar cada fonte de luz em uma passada de renderização separada fazendo uso das capacidades de mistura aditiva de cor do *hardware* para acumular os resultados em um *buffer* de cor.

Algoritmo 2.2 Múltiplas passadas, múltiplas luzes

```
Para Cada Fonte de Luz Faça
    Para Cada objeto afetado por essa fonte de luz Faça
        buffer de cor += brdf( objeto, luz )
    Fim Para
Fim Para
```

Assim como o Algoritmo 2.1, este tem o potencial de desperdiçar tempo de processamento em caso de oclusão de objetos além da dificuldade de implementação eficiente para cenas de larga escala. Uma das vantagens deste algoritmo é que o número total de combinações de programas de *shader* é menor que no Algoritmo 2.1. Tomando o mesmo exemplo com quatro fontes de iluminação direta, até seis fontes de luz afetando um objeto e cinco possíveis tipos de materiais, esta abordagem requer somente 25 programas de *shader* (um para cada combinação de material e tipo de luz, incluindo luz ambiente) ao invés de 1050.

Deferred Shading: A disponibilidade de múltiplos alvos de renderização (MRT) tornou o *deferred shading* uma alternativa viável comparado a técnicas tradicionais de renderização em GPU. A ideia básica é executar todos os testes de visibilidade antes de qualquer processo de iluminação. Na renderização tradicional de GPU, o *Z-buffer* normalmente é preenchido a medida que objetos são renderizados. Este processo pode ser ineficiente, pois, um único *pixel* frequentemente corresponde a mais de um fragmento.

Se um fragmento é completamente coberto por outro que apareça posteriormente, e o objeto é opaco, todo o tempo gasto no cálculo do fragmento anterior fica desperdiçado. A ordenação espacial dos objetos mais próximos aos mais distantes ajuda a minimizar esse problema, mas o *deferred shading* o resolve completamente e ainda resolve o problema de combinação material/fontes de luz.

Ao contrário de técnicas tradicionais de rasterização que enviam a geometria para ser rasterizada e imediatamente aplicam efeitos de iluminação nas primitivas, a técnica de *deferred shading* envia a geometria somente uma vez, armazenando atributos por pixel em uma memória de vídeo local, chamada *G-Buffer*, que é usada em passadas de renderização subsequentes. A Figura 2.1 mostra uma representação em forma de imagem do conteúdo armazenado no *G-Buffer* para uma cena simples. Posições e normais são armazenadas em coordenadas de câmera por questões de conveniência no momento do cálculo da cor dos pixels da imagem final. Na Figura 2.1f, as propriedades óticas mostradas representam atributos que serão utilizados no estágio de traçado de raios e seu uso está descrito no Capítulo 3.

Depois desse ponto, um programa de *shader* separado é aplicado em cada pixel, de forma a executar algoritmos de iluminação ou efeitos de pós-processamento, como o borrão de movimento e profundidade de campo. Estes *shaders* de iluminação leem os atributos de objeto armazenados para cada pixel do *G-Buffer*, a fim de determinar a cor da superfície a qual o específico pixel pertence. Um pseudo-código que ilustra esse processo é apresentado no Algoritmo 2.3:

Algoritmo 2.3 *Deferred Shading*

```
Para Cada Objeto Faça
    G-Buffer = propriedades de iluminação do objeto
Fim Para
Para Cada fonte de luz Faça
    buffer de cor += brdf(G-Buffer, fonte de luz)
Fim Para
```

Apesar de tudo, *deferred shading* possui alguns inconvenientes. A memória de vídeo necessária e o custo de preenchimento do *G-Buffer* são significativos. A principal desvantagem da técnica comparada com outros algoritmos é a dificuldade de aplicação de *antialiasing* e transparência. Para superar estas limitações, técnicas de *antialiasing* que funcionam em espaço de imagem podem ser utilizadas. Um dos métodos mais utilizados é o de detecção de bordas de Shishkovtsov [41]. Um método recente que produz bons resultados é o *Morphological Antialiasing* [38] que suaviza padrões de *aliasing* conhecidos

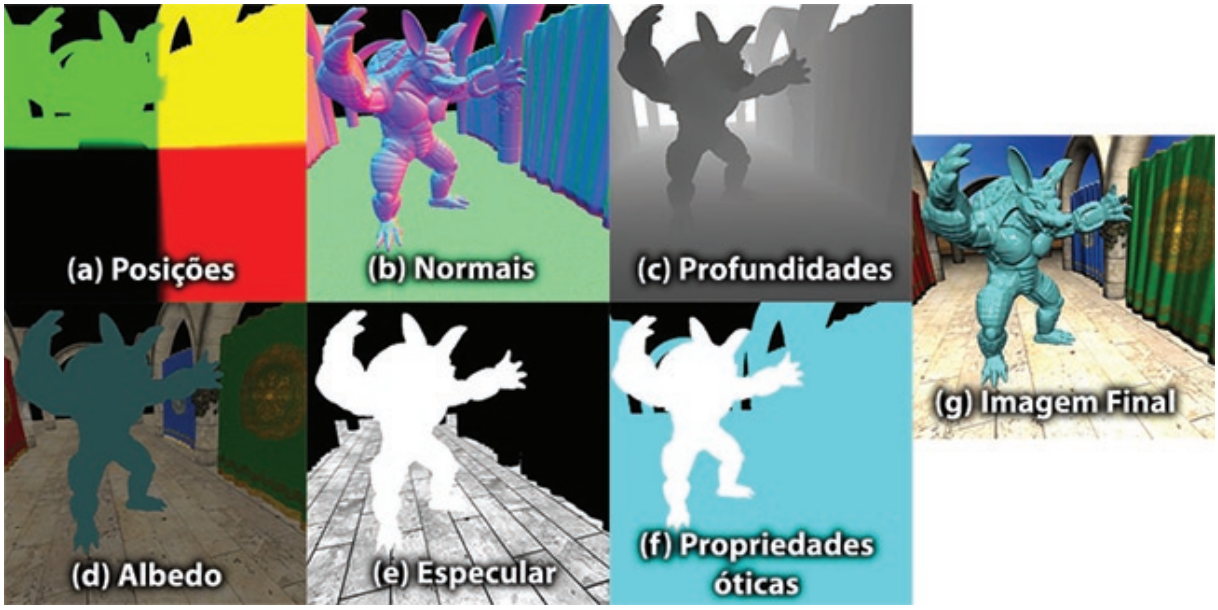


Figura 2.1: Representação na forma de imagens do conteúdo armazenado no *G-Buffer*

em imagens geradas por computador.

A Figura 2.2 apresenta o resultado do *deferred shading*, que equivale a renderização com traçado de raios utilizando somente raios primários. Note que algumas partes do modelo apresentado não possuem um modelo de iluminação aplicado. Essas partes correspondem a materiais translúcidos e sua iluminação fica a cargo do traçador de raios.

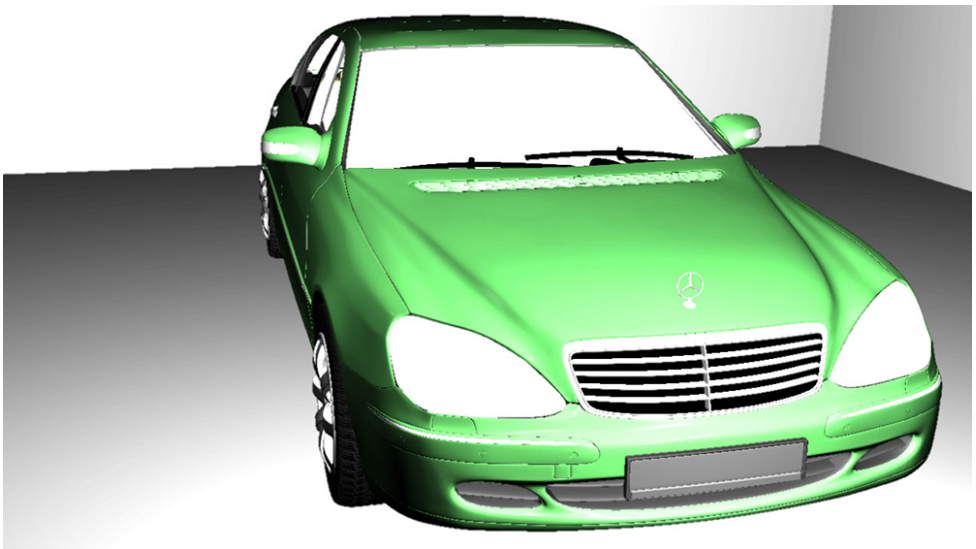


Figura 2.2: Resultado da etapa de *deferred shading*.

2.2 Traçado de Raios

O traçado de raios tem intrigado programadores a muito tempo. Ele é um algoritmo interessante: um traçador de raios pode ser implementado em cem linhas de código C e ainda ser capaz de produzir resultados que superam, em qualidade visual, muitos rasterizadores. A ideia central do algoritmo é compacta e não muito difícil de ser implementada, mas uma implementação ingênua é incrivelmente lenta [9].

O traçado de raios é um método de renderização em que raios são usados para determinar a visibilidade de vários elementos. O mecanismo básico é muito simples e, de fato, traçadores de raios funcionais que cabem em cartão de visitas já foram desenvolvidos [21]. No traçado de raios clássico, raios são disparados a partir da câmera através da grade de *pixels* na direção da cena. Para cada raio, o objeto mais próximo é encontrado. O ponto de intersecção com esse objeto é determinado para ser iluminado ou sombreado, traçando raios a partir deste ponto em direção às fontes de luz da cena. Se algum objeto estiver no caminho até uma das fontes de luz, o ponto está em uma área de sombra. Objetos opacos bloqueiam a luz e objetos transparentes a atenuam.

Outros raios podem ser disparados do ponto de intersecção mais próximo encontrado. Se a superfície é especular, um raio é gerado na direção de reflexão. Este raio é traçado e o objeto mais próximo determinado. A cor deste objeto é determinada traçando raios de sombra em direção as fontes de luz. Este processo de reflexão é recursivamente repetido até que uma superfície difusa, um nível máximo de recursão ou um peso mínimo de contribuição sejam encontrados. Raios também podem ser gerados na direção de refração para objetos transparentes sólidos e novamente, avaliados recursivamente. Quando o número máximo de refrações ou reflexões é atingido, uma árvore de raios é formada. Essa árvore é então avaliada, partindo das folhas em direção a raiz, compondo a cor que foi amostrada. A cor final de um pixel é determinada, então, pela acumulação das contribuições de cada nó desta árvore.

O traçado de raios clássico produz reflexões, refrações e sombras rígidas. Contudo, cada amostra do plano de imagem pode ser feita independemente das demais, portanto, qualquer esquema de amostragem pontual e filtragem pode ser utilizado para obtenção de *antialiasing*. Outra vantagem do traçado de raios é a fácil incorporação de objetos descritos por curvas analíticas e outros objetos não poligonais.

Como descrito, com o traçado de raios clássico, raios são disparados na direção mais significativa: na direção de fontes de luz e nas direções de reflexão e refração perfeitas. Isso

é capaz de resolver o problema da iluminação direta de superfícies e inter-reflexões para superfícies puramente especulares, mas não é capaz de resolver interreflexões difusas. Para resolver estes problemas, métodos estocásticos de traçado de raios devem ser empregados. Um desses métodos é o traçado de raios que usa o método de integração *Monte Carlo*. Este método é capaz de resolver completamente a equação de renderização de Kajiya [25], dada uma quantidade suficiente de raios e o tempo necessário para traçar esses raios.

O traçado de raios possui suas limitações. Por exemplo, a eficiência da estrutura de aceleração escolhida é de vital importância para o desempenho. Quando um objeto se move ou é deformado, essa estrutura, normalmente, deve ser atualizada ou reconstruída. Existe também o problema da incoerência na direção de raios secundários, que pode levar a uma incoerência de acesso a memória cache, degradando o desempenho. Em [16] é encontrada uma discussão sobre o uso de traçado de raios como mecanismo de renderização para jogos eletrônicos.

2.2.1 Estruturas de Aceleração

Muitos trabalhos sobre traçado de raios abordam a construção e travessia eficiente de estruturas de dados de aceleração. De acordo com Whitted [45], uma implementação do algoritmo de traçado de raios gasta entre 75% e 95% de seu tempo de execução no cálculo de intersecção entre raio e objetos. Portanto, o desenvolvimento de estruturas de dados eficientes é de vital importância para o traçado de raios. Lauterbach [29] apresenta dois algoritmos de construção rápida de hierarquia de volumes envolventes (BVH) em GPUs. Foley, Sugerman e Horn et al. [15, 22] descrevem o uso de *kD-Trees* para traçado de raios de forma interativa. Reinhard [37] descreve estimativas de custo para o algoritmo de traçado de raios de acordo com a estrutura de dados de aceleração escolhida. Uma estrutura de aceleração chamada *hierarquia de intervalos envolventes* é apresenta em [43] e é comparada a outras estruturas de dados. Barbosa [5] apresenta a implementação do algoritmo de traçado de raios em GPU utilizando uma *octree* baseada em *hash* como estrutura de aceleração proposta por Madeira em [30].

2.2.2 Renderização *Offline*

Whitted e Cook [45, 12] propuseram outras aproximações para encontrar a solução da equação de renderização. Cook utiliza um método de amostragem estocástica para geração de imagens e também introduz o conceito de traçado de raios distribuídos. A ideia chave

de seu trabalho é que, pela distribuição da direção dos raios de acordo com a função analítica sendo amostrada, o traçado de raios pode incorporar fenômenos difusos. Isso fornece soluções simples e corretas para alguns problemas que, até então, eram somente parcialmente resolvidos, incluindo o desfoque de movimento, profundidade de campo, penumbras, translucência e reflexões difusas. Cook apresenta um estudo formal sobre amostragem estocástica no campo da computação gráfica.

Christesen [11] descreve como o renderizador *Pixar RenderMan™* foi estendido para suportar o traçado de raios. O *RenderMan* utiliza o algoritmo de *scanline REYES*. O algoritmo *REYES* é muito eficiente para renderização de cenas complexas, pois trabalha com cache de geometria em múltiplas resoluções, cache de textura de múltiplas resoluções e utiliza o conceito de micro polígonos [13]. Apesar de ser utilizado principalmente como renderizador *offline* de imagens realistas, é discutido como, para o filme *Carros* da *Pixar*, o traçado de raios foi uma solução viável para renderização de pinturas automotivas, sombras marcantes e oclusão de ambiente.

Pharr e Humphreys [36] descrevem os conceitos e a teoria da renderização foto-realista junto com o código fonte de um renderizador sofisticado de código aberto chamado *pbrt (Physically Based Ray Tracer)*. O *pbrt* foi projetado com três metas principais. Ele deve ser completo, ilustrativo e fisicamente embasado. As fundações básicas de um renderizador fisicamente embasado são as leis da física e suas expressões matemáticas. O *pbrt* foi projetado para uso correto de conceitos e unidades físicas.

2.2.3 Renderização em Tempo Real

Mesmo sendo considerado um algoritmo naturalmente paralelo, implementações paralelas de um traçador de raios ainda são muito mais lentas quando comparadas com técnicas de rasterização [8]. Fazendo uso de modernas GPUs comerciais no papel de processadores massivamente paralelos, a possibilidade da substituição de técnicas de rasterização por traçado de raios começa a ser considerada como uma opção viável, considerando a vantagem da possibilidade de geração de imagens foto-realistas. Contudo, resultados de diferentes trabalhos indicam que RTRT ainda é uma tarefa desafiadora [1, 19, 23, 32, 24].

Beck [6] propõe um traçador de raios híbrido de tempo real seguindo a linha de balanço de carga entre CPU e GPU. Seu trabalho divide os estágios clássicos do traçado de raios em tarefas que são executadas de forma independente em CPU e GPU. Estas tarefas podem ser resumidas em três etapas de GPU e duas de CPU. As primeiras três etapas de GPU consistem na geração de um mapa de sombras, identificação de geometria e uma

etapa de borramento como uma forma de *antialiasing*. A CPU recebe os dados gerados pelas etapas de GPU e fica responsável pela geração e traçado de raios de refração e reflexão, utilizando o algoritmo de traçado de raios. A etapa final consiste na iluminação utilizando o modelo de *Phong* que também faz a composição do resultado final.

O trabalho de Pawel Bak [4] consiste na implementação de RTRT utilizando *DirectX 11* e *HLSL*. Similar ao trabalho de Beck, também faz uso de etapas de rasterização com o objetivo de atingir o melhor desempenho para raios primários.

Wald [44] descreve o sistema traçador de raios de tempo real *OpenRT*, que é uma solução puramente executada em software. Ele é capaz de atingir taxas interativas de geração de quadros através do uso de várias técnicas que incluem um método rápido de intersecção raio-triângulo, uma estrutura otimizada de *BSP* que permite criação e travessia rápido de raios. O *OpenRT* também faz uso de extensivo de instruções *SIMD*, fornecendo um algoritmo de traçado de raios otimizado para CPUs. O *OpenRT* foi construído de forma que a escalabilidade fosse possível através da adição de mais CPUs ou por clusterização.

O traçado de raios de cenas dinâmicas em GPU é explorado em [31]. Nesse trabalho é feito um estudo e implementação de uma estrutura de dados espacial de aceleração de traçado de raios cuja manutenção é feita em GPU utilizando *CUDA*. É implementado um traçador de raios inteiramente em GPU, de forma a explorar o paralelismo inerente do algoritmo. Uma solução para contornar o problema da recursão em GPU é apresentada, pois foram utilizadas GPUs sem suporte a esse recurso.

Chen [10] apresenta um traçador de raios híbrido que faz uso de CPU e GPU para gerar o resultado final. Em seu trabalho, a técnica de rasterização *Z-buffer* é utilizada para determinação de visibilidade de triângulos, ao mesmo tempo em que a intersecção de raios primários, que seriam originados no ponto de origem da câmera, são determinadas. Os dados gerados pela etapa de *z-buffer* são lidos pela CPU com o objetivo de traçar raios secundários.

Em [39] é descrito uma extensão para o trabalho de Chen. A determinação da visibilidade de triângulos também é feita utilizando o algoritmo de *Z-Buffer* de forma que o traçado e intersecção de raios primários de câmera seja eliminado. A extensão acontece no momento de calcular o traçado de raios como efeito secundário. Ao invés de ler as informações do *framebuffer* para a memória da CPU, é executada uma etapa de traçado de raios utilizando *CUDA*. Essa abordagem tem a vantagem de eliminar o gargalo de transferência de dados entre CPU e GPU melhorando o desempenho da renderização.

Finalmente, Hachisuka [20] pesquisa vários algoritmos de traçado de raios para GPU. Uma visão geral destas diferentes técnicas, considerando a arquitetura de *hardware* gráfico, é descrita. Mesmo o traçado de raios sendo um problema bem estudado e dado a sua natureza paralela, muitos problemas específicos precisam ser tratados para que sua execução seja eficiente. Alguns destes problemas são discutidos, como a retenção de dados, a necessidade do *hardware* gráfico de executar a mesma operação sobre elementos diferentes e sua natureza recursiva.

2.2.4 Traçado de Raios com *OptiX*

OptiX é um motor de traçado de raios de propósito geral da *NVIDIA* que faz uso das GPUs que suportam *CUDA* a fim de facilitar a construção de traçadores de raios. Utilizando *OptiX*, desenvolvedores podem focar nas etapas de geração de raios e iluminação, abstraindo a construção e travessia de estruturas de dados de aceleração. *OptiX* oferece um motor de traçado de raios de alto nível, um *pipeline* programável de traçado de raios com uma linguagem de *shader* baseado em *CUDA C/C++*, um compilador específico para geração de código executável em GPU e uma representação na forma de grafo de objetos da cena e seus respectivos materiais.

As GPUs são boas ao explorar problemas com um alto grau de paralelismo e o traçado de raios se encaixa perfeitamente nesse quesito. No entanto, algoritmos típicos de traçado de raios podem ser muito irregulares e podem criar muitos desafios para qualquer um que tenta explorar o poder computacional das GPUs. O motor traçador de raios *OptiX* aborda esses desafios e fornece uma maneira de suavizar a quantidade de trabalho necessária para incorporar traçado de raios interativo em aplicações fazendo uso intensivo de GPUs.

OptiX não é, sozinho, um traçador de raios. Trata-se de um motor para construir aplicações baseadas em traçados de raios. O motor *OptiX* é composto por duas partes: uma *API* que define as estruturas de dados e uma linguagem baseada em *CUDA C/C++* utilizada para produzir raios, intersectar raios com superfícies e responder a essas intersecções. *OptiX* pode ser utilizado para aplicações que fazem uso de traçado de raios não necessariamente gráficas. Detecção de colisão, propagação de som e determinação de visibilidade são alguns exemplos de aplicações não gráficas deste algoritmo [35].

2.2.4.1 Modelo de Programação

O *OptiX* fornece uma *API* em C baseada em objetos que implementa um modelo hierárquico de retenção simples. Também é fornecida uma *API* orientada a objetos em C++ que funciona como um envelope para a *API* em C. Os principais objetos fornecidos pelo *OptiX* são:

- **Contexto**: Uma instância do motor do *OptiX*;
- **Programa**: Uma função escrita em *CUDA C/C++*, compilada para a linguagem de *assembly* virtual *NVIDIA PTX*;
- **Variável**: Um nome utilizado para passar dados para programas *OptiX*;
- ***Buffer***: Um arranjo multidimensional que pode ser associado a uma variável;
- ***TextureSampler***: Um ou mais *buffers* associados com um mecanismo de interpolação;
- **Geometria**: Uma ou mais primitivas que um raio pode intersectar, tais como triângulos, esferas e outras primitivas definidas pelo usuário;
- **Material**: Um conjunto de programas executados quando um raio é intersectado com uma primitiva mais próxima em potencial;
- ***GeometryInstance***: Uma associação de uma geometria com objetos do tipo material;
- **Grupo**: Um conjunto de objetos organizados em hierarquia;
- **Transformação**: Um conjunto de nós que transformam raios geometricamente. Podem ser vistos como transformações aplicadas na geometria em si;
- **Seletor**: Uma hierarquia programável que pode selecionar qual filho um raio deve atravessar;
- **Aceleração**: Uma estrutura de aceleração que pode ser associada a um nó na hierarquia.

Os objetos são criados, destruídos e associados uns com os outros através da *API* em C. O comportamento que o *OptiX* terá depende de como esses objetos são associados uns aos outros.

2.2.4.2 Programas, Variáveis e Modelo de Execução

O *pipeline* fornecido pelo *OptiX* contém diversos componentes programáveis. Estes programas são executados em GPU em pontos específicos durante a execução do algoritmo genérico de traçado de raios. No total, são oito tipos de programas:

- **Geração de raios**: É o ponto de entrada do *pipeline* de traçado de raios, é invocado em paralelo para cada *pixel*, amostra ou outro tipo de tarefa designado pelo usuário;
- **Exceção**: Tratamento de exceções. É invocado por condições como estouro de pilha, falta de memória ou outros erros;
- **Intersecção mais próxima**: Chamado quando um raio encontra um ponto de intersecção mais próximo, como por exemplo, um ponto para iluminar uma superfície;
- **Qualquer intersecção**: Chamado quando um raio encontra qualquer colisão em potencial. Comumente utilizado para cálculo de sombras;
- **Intersecção**: Um programa que determina a intersecção de um raio com uma primitiva, chamado durante o travessia de um raio pela cena;
- **Caixa envolvente**: Computa uma caixa envolvente, em coordenadas de mundo. É chamado quando o sistema constrói uma nova estrutura de aceleração sobre a geometria;
- **Falha**: Um programa chamado quando um raio não é intersectado com nenhuma geometria da cena. Comumente utilizado para preencher com uma cor ou plano de fundo;
- **Visita**: Chamado durante o travessia de um nó *Seletor* para determinar quais dos nós filhos um raio deve atravessar.

A linguagem de entrada para estes programas é *PTX*. O *SDK* do *OptiX* fornece uma série de classes e cabeçalhos que podem ser utilizados em conjunto com o compilador *NVCC* para uso da linguagem *CUDA* C/C++ como uma maneira de gerar programas em *PTX*.

O *OptiX* possui um sistema flexível de variáveis utilizadas para comunicar dados com os programas. Quando um programa do *OptiX* referencia uma dessas variáveis existe um conjunto de escopos bem definidos onde a definição dessas variáveis serão buscadas. Isto permite substituições dinâmicas das variáveis baseado no escopo onde esta foi definida.

Uma vez que todos esses objetos, programas e variáveis são criados e associados em um contexto válido, um programa de geração de raios pode ser invocado. As invocações recebem um parâmetro de dimensionalidade e outro de tamanho para, então, invocar o programa de geração de raios um número de vezes igual a esse tamanho especificado.

2.2.4.3 Grafo de Objetos

A *API OptiX* emprega uma estrutura flexível para representação de informação de cena e as operações programáveis associadas a cada objeto. Essa estrutura é representada por um objeto do tipo *contexto*. Essa representação é também um mecanismo de associação de *shaders* programáveis com os dados específicos de um objeto. A estrutura de grafo representa a cena de forma compacta, pois, somente nós que devem ter sua intersecção com raios determinada necessitam estar presentes na estrutura.

A cena é representada como um grafo. Essa representação é muito eficiente e é utilizada para controlar a travessia de raios através da cena. A estrutura também pode ser utilizada para implementar hierarquias para animações de objetos rígidos ou outros tipos comuns de estruturas de cena. Quatro tipos de nós podem ser utilizados para representação da cena por um grafo acíclico direcionado. Qualquer nó pode ser utilizado como raiz na travessia da cena. Isto permite, por exemplo, diferentes representações para diferentes tipos de raios.

A estrutura de grafo fornecida pelo *OptiX* não tem o objetivo de ser um grafo de cena no sentido clássico. Essa estrutura tem o objetivo de associar diferentes programas e ações a porções da cena. A Figura 2.3 ilustra um exemplo de um grafo que pode ser utilizado para atravessar raios através de programas *OptiX*.

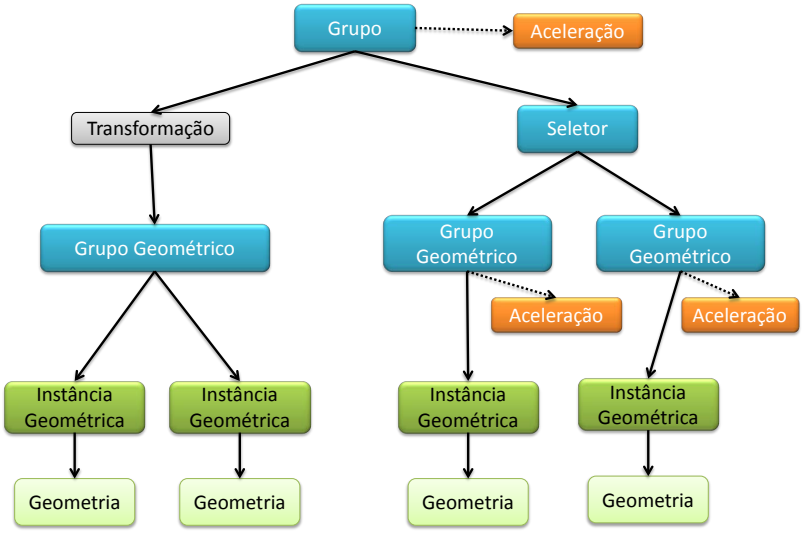


Figura 2.3: Exemplo de grafo implementado utilizando a estrutura do *OptiX*

Capítulo 3

Traçado de Raios Híbrido

O objetivo principal deste trabalho é desenvolver um renderizador que faz tanto uso de rasterização quanto traçado de raios de forma a obter o que as duas técnicas oferecem de melhor, mas mantendo uma taxa interativa de geração de quadros. Este objetivo pode ser alcançado através da interpretação dos dados produzidos pela técnica de *deferred shading* e armazenados no *G-Buffer* como o estágio de traçado e colisão de raios primários de um traçador de raios completo. A partir desse ponto, raios de sombra, refração e reflexão podem ser traçados somente para os *pixels* cujo material associado seja reflexivo ou refrativo.

Neste capítulo apresentamos a teoria necessária para implementação da proposta deste trabalho além de um resultado preliminar da composição das imagens rasterizadas e raio-traçadas.

Os dados produzidos e armazenados no *G-Buffer*, através da técnica de *deferred shading* descrita na Seção 2.1, podem ser interpretados como o primeiro estágio de um *pipeline* completo de traçado de raios, correspondente ao traçado e colisão de raios primários. O atributo de posição armazenado no *G-Buffer* representa a posição em coordenadas de câmera de cada *pixel* que deve ser iluminado. Pode-se tomar essa posição como a distância Euclidiana de cada ponto em relação à câmera. Sabendo a posição da câmera no mundo, pode-se tomar o *buffer* de posições como a resolução do traçado de raios primários. A fim de recuperar a posição em coordenadas de mundo, aplica-se a transformação inversa de câmera no ponto em questão. O mesmo é feito a fim de recuperar a normal em coordenadas de mundo para a posição calculada. As Equações 3.1a e 3.1b demonstram essa operação:

$$\mathbf{p} = \mathbf{M}^{-1}\mathbf{G}(a, (i, j)) \tag{3.1a}$$

$$\mathbf{n} = \left\|\mathbf{M}^{-1}\left[\mathbf{G}(b, (i, j)) - 0.5\right]\right\| \tag{3.1b}$$

onde:

- $\mathbf{p}$ é o valor da posição transformado para coordenadas de mundo;
- $\mathbf{n}$ é o vetor normal transformado para coordenadas de mundo;
- $\mathbf{M}$ é a matriz de transformação da câmera;
- (i, j) é uma posição na grade de *pixels*;
- a é um elemento que identifica de qual *buffer* interno do *G-Buffer* será mapeado. Deve ser um elemento do conjunto a, b, c, d, e, f onde a representação de cada *buffer* pode ser vista na Figura 2.1;
- $\mathbf{G}$ é uma função que mapeia um dado solicitado do *G-Buffer*, dada uma posição (i, j) na grade de *pixels* amostrada.

A função $\mathbf{G}$ é utilizada para mapear um valor do *G-Buffer*. Nas Equações 3.1a e 3.1b, a função $\mathbf{G}$ é amostrada com os valores a e b que representam os *buffers* de posições e normais, respectivamente. O valor 0.5 utilizado na Equação 3.1b é necessário pois as normais são armazenadas no intervalo $[0, 1]$ de forma a economizar espaço e largura de banda da memória de vídeo. Mais detalhes sobre como cada *buffer* do *G-Buffer* é criado podem ser vistos na Seção 5.1. Finalmente, o vetor calculado na Equação 3.1b é normalizado para ser utilizado nas equações de iluminação.

A representação do conteúdo do *G-Buffer* mostrado na Figura 2.1f apresenta o buffer de propriedades óticas. Embora as informações tipicamente armazenadas em um *G-Buffer* sejam suficientes para determinar a origem e direção de raios de sombra, não é possível estabelecer a necessidade de geração de raios de reflexão e/ou refração somente com informações geométricas. O *G-Buffer* é, então, estendido com um *buffer* extra utilizado para armazenar propriedades óticas da superfície rasterizada de cada objeto. Refletividade, índice de refração, opacidade e expoente especular são incluídos nesse *buffer*.

Sombras: Quando comparado com a técnica de *Shadow Map* [46], o traçado de raios apresenta uma abordagem mais simples e de fácil implementação. Dados $\mathbf{p}$ e $\mathbf{n}$ como

sendo a posição e normal, em coordenadas de mundo, de um ponto onde ocorreu uma colisão de um raio primário com um objeto, oriundos das Equações 3.1a e 3.1b, traça-se um raio de sombra em direção a cada uma das fontes de luz para determinar se um ponto está em uma área de sombra ou não. O Algoritmo 3.1 ilustra essa etapa.

Algoritmo 3.1 *Sombras Raio-Traçadas*

Entrada: Ponto de colisão h_p e normal $\mathbf{n}$
Saída: Cor resultante C_R do ponto h_p
  $C_R = 1$
  Para Cada Fonte de Luz L_s **Faça**
    $\mathbf{L} = \|L_s^p - h_p\|$ ▷ L_s^p é a posição da fonte de luz
    $\theta = \mathbf{n} \cdot \mathbf{L}$
    Se $\theta > 0$ **Então**
      $w = trace(h_p, \mathbf{L})$
      Se $w > 0$ **Então**
        $L_c = wL_s^cL_s^e$ ▷ L_s^c e L_s^e são a cor e a energia
        $C_R \mathrel{+}= \theta L_c$ ▷ da fonte de luz, respectivametente
      Fim Se
    Fim Se
  Fim Para

Algoritmo 3.2 *Atenuação de luz de um raio*

Entrada: Normal $\mathbf{n}$
Entrada: Raio R
Entrada: Opacidade α
Entrada: Cor da superfície K_d
Saída: Atenuação de luz A
  Se $\alpha < 1$ **Então**
    $\phi = \|\mathbf{n} \cdot R_d\|$
    $A = 1.3A - schlick(\phi, \alpha)$ ▷ *schlick* é retorna uma
    $A = AK_d$ ▷ aproximação do efeito fresnel.
  Se não
    $A = 0$
  Fim Se

O Algoritmo 3.1 faz uso do procedimento *trace*. Este procedimento é responsável por traçar um raio com origem e direção determinadas e retornar a atenuação da luz na direção especificada. Essa atenuação é calculada fazendo uma consulta ao material do objeto cujo raio traçado colidiu. O Algoritmo 3.2 descreve o pseudo-código do algoritmo que calcula a atenuação de luz, dado que um raio colidiu com um objeto.

O próximo passo é utilizar a função **G** para amostrar o *buffer* de propriedades óticas e, assim, determinar a necessidade de traçado de raios de reflexão e refração.

Reflexão: O termo de refletividade armazenado no primeiro canal do *buffer* de propriedades óticas define a necessidade de traçado de raios secundários de reflexão e também define a quantidade de luz refletida que é absorvida pelo material antes de chegar ao olho do observador. Sabendo a direção do raio que atingiu uma superfície de um objeto, a direção de reflexão $\mathbf{v}_{reflect}$ pode ser calculada através da Equação 3.2:

$$\mathbf{v}_{reflect} = \mathbf{v} - 2 \times \mathbf{n}(\mathbf{n} \cdot \mathbf{v}) \tag{3.2}$$

onde $\mathbf{v}$ é a direção do vetor incidente e $\mathbf{n}$ é o vetor normal à superfície no ponto em questão. Ambos $\mathbf{n}$ e $\mathbf{v}$ são vetores cuja norma é igual a 1. Pode ser utilizado, por exemplo, para amostrar o lóbulo de reflexão da *BRDF*.

Refração: O termo de refração, armazenado no segundo canal do *buffer* de propriedades óticas, representa o índice de refração da superfície. Raios de refração somente são traçados caso a superfície atingida seja translucida. O termo de opacidade é utilizado para determinar a necessidade de traçado de raios de refração. Para calcular a direção de refração $\mathbf{v}_{refract}$, utilizamos a Equação 3.3 derivada da lei de Snell [18]:

$$\mathbf{v}_{refract} = \frac{\eta_1}{\eta_2}\mathbf{v} + \left(\frac{\eta_1}{\eta_2} \cos\theta - \sqrt{1 + \left(\frac{\eta_1}{\eta_2}\right)^2 (\cos^2\theta - 1)} \right) \mathbf{n} \tag{3.3}$$

onde η_1 e η_2 são os índices de refração dos meios em questão. $\mathbf{v}$ é vetor incidente e θ é o ângulo entre o vetor incidente e o vetor normal à superfície $\mathbf{n}$.

Figura 3.1: Resultado da etapa de traçado de raios.

A Figura 3.1 apresenta o resultado gerado pela etapa de traçado de raios descrita nesta

Figura 3.2: Resultado da etapa de composição.

seção. O modelo do automóvel mostrado apresenta o interior colorido. Isso acontece pois o para-brisas possui índice de refração igual a $1,075$, portanto as cores do interior do veículo são determinadas pelo traçador de raios. O símbolo no capô do automóvel é composto por um material reflexivo, assim como alguns detalhes na parte dianteira do automóvel e nas rodas. As várias sombras são resultado de várias fontes de iluminação contidas na cena.

A Figura 3.2 apresenta o resultado da composição das imagens apresentadas nas Figuras 2.2 e 3.1, de forma a adicionar na imagem rasterizada efeitos produzidos por traçado de raios. Para fins de ilustração, é mostrado na Figura 3.3 o conteúdo do *G-Buffer* para a cena do veículo.

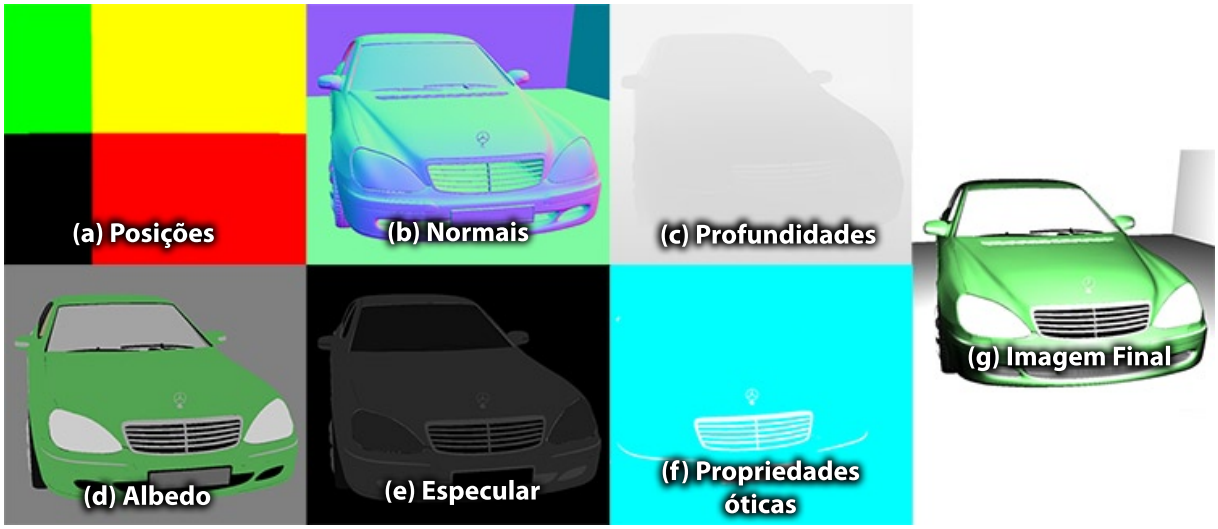


Figura 3.3: Conteúdo do *G-Buffer* para a cena do veículo.

Capítulo 4

Arquitetura Proposta

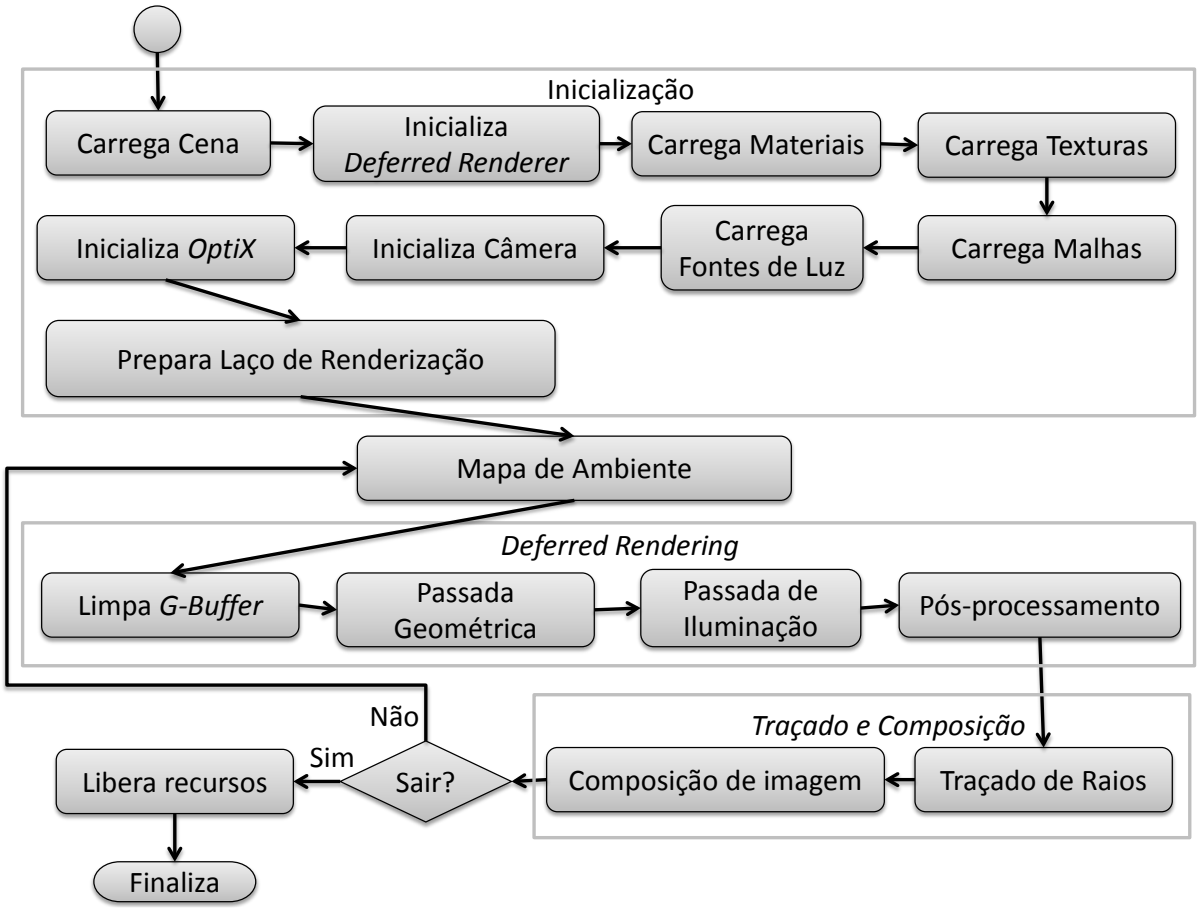


Figura 4.1: Representação esquemática do traçador de raios híbrido.

Neste capítulo é descrita a arquitetura do sistema híbrido implementado descrito na Capítulo 3. Os desafios e particularidades de cada etapa serão explorados de forma a colocar em prática a teoria apresentada.

A Figura 4.1 apresenta a arquitetura básica da aplicação implementada na forma de

fluxograma.

Uma maneira simples de representar cenas é pela definição explícita e isolada de cada objeto que a representa. No entanto, a renderização de cenas tridimensionais não é restrita somente a geometria. Controle de animação, métodos de determinação de visibilidade, propriedades de iluminação e outros elementos são usualmente implementados com o uso de um *grafo de cena*. Essa é uma estrutura de dados que pode ser estendida com texturas, transformações, níveis de detalhe, estados de renderização (propriedade de materiais, por exemplo), fontes de luz e quaisquer outros elementos que sejam necessários [2]. O grafo de cena se mostrou a estrutura de dados ideal, pois, o controle de animação é feito pela interpolação dos nós de transformação entre os quadros-chave, além de permitir uma fácil integração com o *OptiX*. Um grafo de cena possui um mapeamento direto com a representação interna em forma de grafo de objetos do *OptiX*. Para animações, cada transformação alterada é imediatamente disponibilizada para uso interno do *OptiX* com o objetivo de colidir raios com um objeto deslocado ou deformado. A escolha das estruturas de dados de aceleração também é trivial, como será visto adiante. O grafo de cena divide a cena em elementos estáticos e dinâmicos. Para objetos dinâmicos é escolhida uma estrutura de aceleração que seja de rápida reconstrução ou que seja de rápida adaptação.

O fluxograma mostrado na Figura 4.1 mostra a ordem dos passos executados para a implementação do traçador de raios híbrido proposto. O sistema foi dividido em três partes: Inicialização, *Deferred Rendering* e, finalmente, Traçado e Composição. Nas seções a seguir, cada etapa será explorada com seus respectivos detalhes e desafios.

4.1 Inicialização

A primeira etapa do processo, chamada de *Inicialização*, consiste em alocar todos os recursos necessários e preparar todos os algoritmos implementados para execução.

- **Carrega Cena**: Cenas e modelos tridimensionais podem ser representados de diversas maneiras. Existe uma grande variedade de representações de cenas com suas respectivas vantagens e desvantagens. A etapa de carregar a cena deve abstrair o formato de arquivo utilizado de forma a prover uma interface unificada de representação de cena. Essa etapa também é responsável por aperfeiçoar os modelos carregados em termos de eliminar duplicata de vértices e malhas, garantir que todas as faces sejam triângulos, calcular a base do espaço tangente para o uso de mapas de normais, aperfeiçoar a localidade espacial no acesso a memória cache pela re-

ordenação das faces em memória, remoção de materiais redundantes, verificar por primitivas degeneradas e remoção de nós desnecessários no grafo de cena.

- **Inicializa *Deferred Renderer***: Consiste na alocação de espaço e configuração do *G-Buffer* que será utilizado na etapa de *deferred shading*. Essa etapa também carrega o mapa de ambiente utilizado como plano de fundo onde raios de reflexão, que escapam da cena, podem buscar informação de radiância para iluminação de uma superfície;

- **Carrega Texturas**: Cada material utilizado na cena pode conter até cinco texturas diferentes: difusa, ambiente, especular, mapa de normais e mapa de transparência. Cada textura utilizada na cena é carrega na memória de vídeo de forma que esteja pronta para uso a qualquer momento e somente uma cópia é mantida em memória de forma a ser utilizada na etapa de *deferred shading* e traçado de raios;

- **Carrega Malhas**: Cada objeto pode ser composto por uma ou mais malhas de triângulos. Essa etapa é responsável por disponibilizar os objetos tridimensionais para uso nos renderizadores. Os objetos são alocados diretamente em memória de vídeo de forma a garantir um acesso rápido ao seu conteúdo;

- **Carrega Fontes de Luz**: Consiste em alocar recursos para que as fontes de iluminação da cena estejam disponíveis para os renderizadores;

- **Inicializa Câmera**: A inicialização da câmera consiste em, dado as características da câmera, montar as transformações de câmera e projetivas. As características da câmera utilizadas são: posição em coordenadas de mundo, direção de apontamento, vetor de orientação, abertura focal e os dois planos que delimitam o cone de visão;

- **Inicializa *OptiX***: Escolhida como *API* de traçado de raios para este trabalho, a inicialização do *OptiX* consiste em preparar o contexto de traçado de raios. A representação em forma de grafo de cena é disponibilizada para o *OptiX* com a criação de nós de transformação, geometria, e escolha das estruturas de aceleração;

- **Prepara o Laço de Renderização**: Antes de ser iniciado o laço da aplicação, o contexto *OptiX* criado na etapa anterior deve ser validado e compilado. Caso alguma inconsistência seja encontrada, o processo é interrompido.

4.2 Mapa de Ambiente

Antes de executar a etapa de *deferred shading*, um mapa de ambiente é renderizado de forma a preencher o fundo da cena. Este mapa de ambiente possui dois objetivos. O primeiro é o de simplificar o cálculo da reflectância de um determinado ponto e o segundo é o de simular um ambiente externo onde os objetos estão sendo renderizados. Na etapa de *deferred shading*, este mapa é utilizado somente para preencher o fundo de acordo com a direção da câmera. Na etapa de traçado de raios, o mapa é amostrado por raios que escapam da cena de modo a simular a reflectância provida pelo ambiente a um determinado ponto.

A Figura 4.3a apresenta um mapa de ambiente que é lido de um arquivo e utilizado como uma textura. A Figura 4.3b mostra um objeto renderizado com o uso do mapa de ambiente para determinar a reflectância. Neste exemplo, raios secundário são traçados a partir do objeto que é reflexivo e, dado a direção de reflexão calculada através da Equação 3.2, amostra-se o mapa de ambiente como a quantidade de luz incidente em um determinado ponto.

Dada a direção de visada da câmera, ou a direção de um raio que não acertou nenhum objeto, calcula-se a latitude e a longitude (u, v) da direção em questão utilizando as Equações 4.1:

$$\theta = \arctan(R_x/R_z) \tag{4.1a}$$

$$\phi = \frac{1}{2} - \arccos R_y \tag{4.1b}$$

$$u = \frac{\theta + \pi}{2\pi} \tag{4.1c}$$

$$v = \frac{1}{2}(1 + \sin\phi) \tag{4.1d}$$

onde R_x, R_y e R_z são as direções em x, y e z que se deseja amostrar o mapa de ambinte. A Figura 4.2 mostra um diagrama onde são mostradas das componentes da direção de um raio além dos ângulos calculados. A conversão para coordenadas esféricas das Equações 4.1a e 4.1b resultam nos valores de θ e ϕ pertencentes aos intervalos $[0, \pi]$ e $[0, 2\pi)$, respectivamente. A fim de determinar as coordenadas de texturas (u, v) a serem utilizadas para amostrar o mapa de ambiente, faz-se uso das Equações 4.1c e 4.1d, cujo objetivo é mapear os valores calculados de θ e ϕ para o intervalo $[0, 1]$.

As Equações 4.1 são também utilizadas no programa que trata um raio que não colidiu

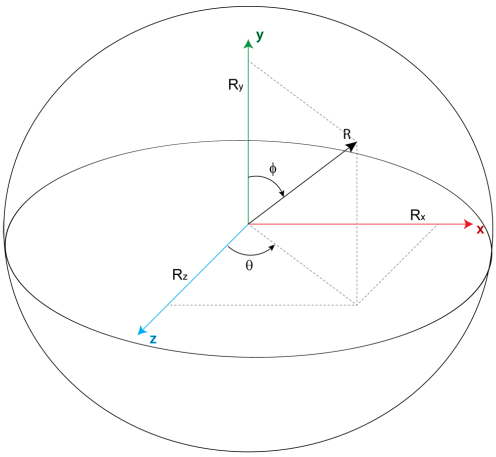
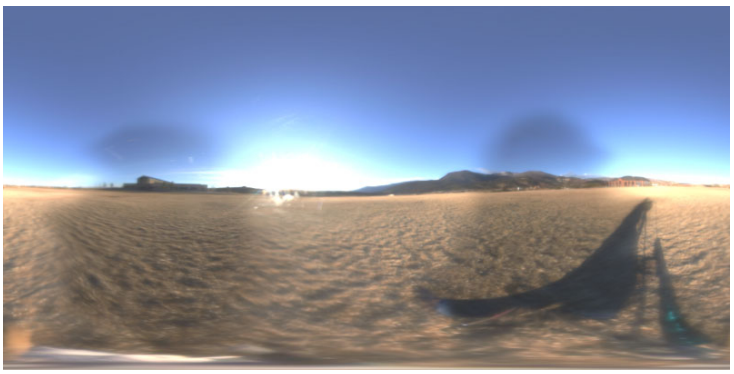
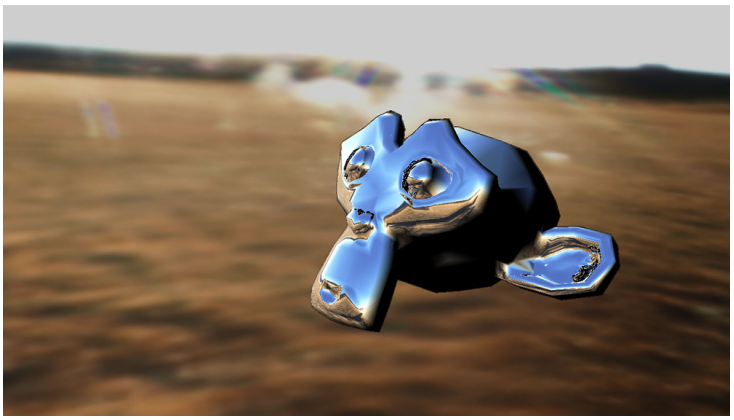


Figura 4.2: Conversão da direção de um raio para coordenadas esféricas.

com nehum objeto.



(a) Exemplo de mapa de ambiente.



(b) Mapa de ambiente renderizado.

Figura 4.3: Exemplo de mapa de ambiente.

4.3 *Deferred Rendering*

A primeira etapa do laço de renderização consiste no uso de *deferred shading* para preencher o *G-Buffer*, como mostrado na Figura 2.1. O uso de *deferred shading* para renderização de cenas é chamado de *Deferred Rendering*.

- **Esvazia *G-Buffer***: O *G-Buffer* é preenchido a cada quadro renderizado. O *buffer* de profundidade é utilizado para resolução do problema da visibilidade de objetos e deve ser esvaziado de modo a garantir que os objetos do quadro atual passem no teste de profundidade. Os *buffers* de cor também são inicializados para garantir o funcionamento da etapa de composição;

- **Passada Geométrica**: Dentro do *pipeline* de *deferred rendering*, a única etapa que realmente faz uso de dados geométricos tridimensionais é a passada geométrica. O grafo de cena é percorrido e cada malha encontrada é renderizada utilizando um programa de *shader* que preenche o *G-Buffer* com as informações necessárias para iluminar cada *pixel* que contribui para a cena final.

- **Passada de Iluminação**: Depois que o *G-Buffer* é preenchido, este estágio utiliza um programa de *shader* para ler as informações necessárias para iluminação de *pixel* de forma que, através da renderização de um retângulo com as dimensões da janela de visualização, resulte na representação da cena final iluminada. O resultado dessa etapa pode ser visto nas Figuras 2.1g e 2.2;

- **Pós-processamento**: Um dos inconvenientes do uso de *deferred shading* é que a primitiva que forma a imagem é um retângulo. Isto impede que técnicas *antialiasing* implementado em *hardware* seja aplicado. Para implementar um filtro de *antialiasing*, neste caso, um programa de *shader* é utilizado para aplicar um borrão na imagem final somente em *pixels* com grandes descontinuidades em normais ou profundidades. Para isso utilizamos o método de detecção de bordas de Shishkovtsov [41]. O filtro de detecção faz a soma dos quadrados das distâncias do *pixel* alvo para os seus vizinhos, gerando um fator de borramento. Todos os oitos vizinhos de um *pixel* são levados em conta para detecção de descontinuidades em todas as direções possíveis. A fórmula para calcular o fator de borrão F_{blur} é descrita na Equação 4.2:

$$F_{blur}(u,v) = \sum_{i=0}^{7} \left(t(u + \mathbf{O}_x^i, v + \mathbf{O}_y^i) - t(u,v) \right)^2 \tag{4.2}$$

onde $t(u, v)$ é o *pixel* (u, v) alvo e $\mathbf{O}$ é uma tabela de deslocamento que contém os passos de acesso aos vizinhos de um *pixel*. Após a definição do fator de borramento, a cor da imagem renderizada é amostrada utilizando a tabela de deslocamentos e uma média ponderada pelo fator de borramento calculado é utilizada para determinar a cor final. A Figura 4.4 mostra um exemplo de aplicação do filtro de *antialiasing*. A Figura 4.4a mostra uma imagem sem a aplicação do filtro de *antialiasing* enquanto a Figura 4.4b apresenta o resultado depois da aplicação do filtro. A Figura 4.4c mostra o resultado da aplicação da Equação 4.2 para obter o fator de borramento. No exemplo desta figura foi utilizado o *buffer* de normais para detecção de bordas.

Definição da Tabela $\mathbf{O}$		
i	$\mathbf{O}_x^i$	$\mathbf{O}_y^i$
0	-1	1
1	1	-1
2	-1	-1
3	1	1
4	-1	0
5	1	0
6	0	-1
7	0	1

Tabela 4.1: Tabela com os passos de acesso aos vizinhos de um *texel*

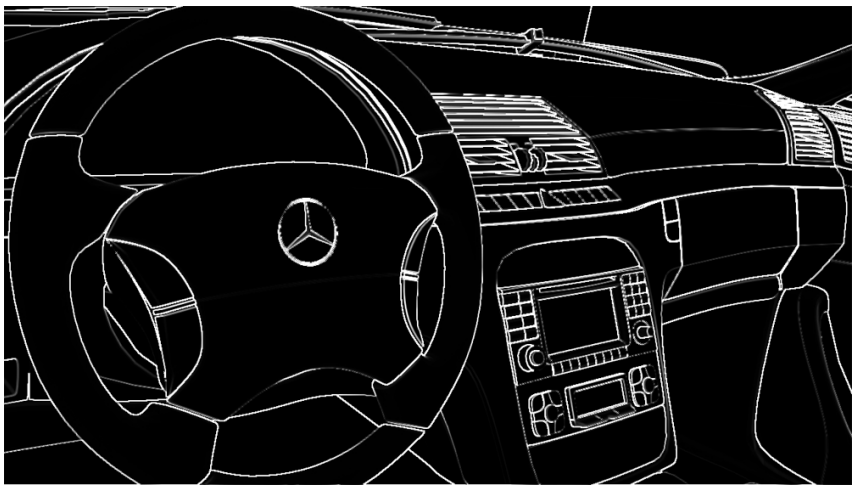
4.4 Traçado e Composição

A segunda etapa do laço de renderização consiste no traçado de raios e composição da imagem final. Essa etapa acontece conforme descrito no Capítulo 3. Utiliza-se os dados do *G-Buffer* (Figura 2.1) para determinar a origem, direção e sentido de raios que devem ser traçados para obtenção de sombras, reflexões e refrações.

O principal objetivo deste trabalho é construir um *pipeline* híbrido que utiliza informações de rasterização para execução de uma etapa posterior de traçado de raios. A fim de manter um nível de desacoplamento grande entre o rasterizador e o traçador de raios, uma consideração deve ser feita ao traçar raios de sombra primários. Como pode ser visto na Figura 2.2, o resultado da etapa de *deferred shading* é criado através do uso de um modelo de iluminação local, não levando em conta oclusão de objetos e sombras projetadas. Para compor uma imagem de traçado de raios com esta, deve-se assumir que cada *pixel* da imagem renderizada pelo traçador de raios possui valor de luminância máximo. O traçador de raios deve, portanto, calcular a atenuação de luz a ser aplicada

(a) Imagem sem filtro de *antialiasing*

(b) Imagem com filtro *antialiasing*



(c) Fator de borramento

Figura 4.4: Destaque da aplicação do filtro de *antialiasing*.

a cada *pixel*. A Figura 3.1 mostra um exemplo de imagem calculada pelo traçador de raios. Os *pixels* que não sofrem nenhum tipo de atenuação de luz são renderizados como

brancos. Ao fazer a composição da imagem da Figura 2.2 com a imagem da Figura 3.1, temos o resultados mostrado na Figura 3.2.

Temos, portanto, um estágio extra que funciona de maneira independente, capaz de calcular efeitos de traçado de raios a partir de dados gerados pela utilização de um algoritmo de *deferred shading*.

4.5 Resumo do *Pipeline* híbrido

Nas seções anteriores foram descritos os estágios que compõem o *pipeline* híbrido de traçado de raios e rasterização proposto. Nessa seção, apresentamos um resumo do *pipeline* completo.

Para ilustrar o processo utilizamos uma cena simples composta por três objetos texturizados e uma fonte de luz pontual. Essa cena é composta por 4746 vértices e 9470 triângulos. A Figura 4.5 mostra os estágios do *pipeline* desenvolvido:

- ***Deferred rendering* e resolução de raios primários**: A Figura 4.5a consiste na renderização da cena utilizando a técnica de *deferred shading*, resultando em um *G-Buffer* preenchido. O *G-Buffer* obtido contém informações sobre fragmentos visíveis e pode ser tomado como a resolução do traçado de raios primários de um traçador de raios tradicional.

- **Sombras**: Com os dados extraídos do *G-Buffer*, raios de sombra são gerados para pontos válidos. Na Figura 4.5a, pontos inválidos são aqueles que devem ser iluminados com o mapa de ambiente. Na figura, estes são representados por *pixels* azuis. O resultado da aplicação da sombra é mostrado na Figura 4.5b.

- **Reflexões e refrações**: A Figura 4.5c utiliza o algoritmo de traçado de raios para disparar e colorizar raios de reflexão e refração. Neste exemplo, utilizamos um material reflexivo no chão para ilustrar a aplicação de tal efeito.

- **Composição final**: O estágio final de composição é ilustrado pela Figura 4.5d. Este estágio é responsável por compor as imagens geradas nos estágios 4.5b e 4.5c. As duas imagens são compostas utilizando um programa de *shader* de forma a tirar vantagem do *hardware* especializado para tal tarefa.

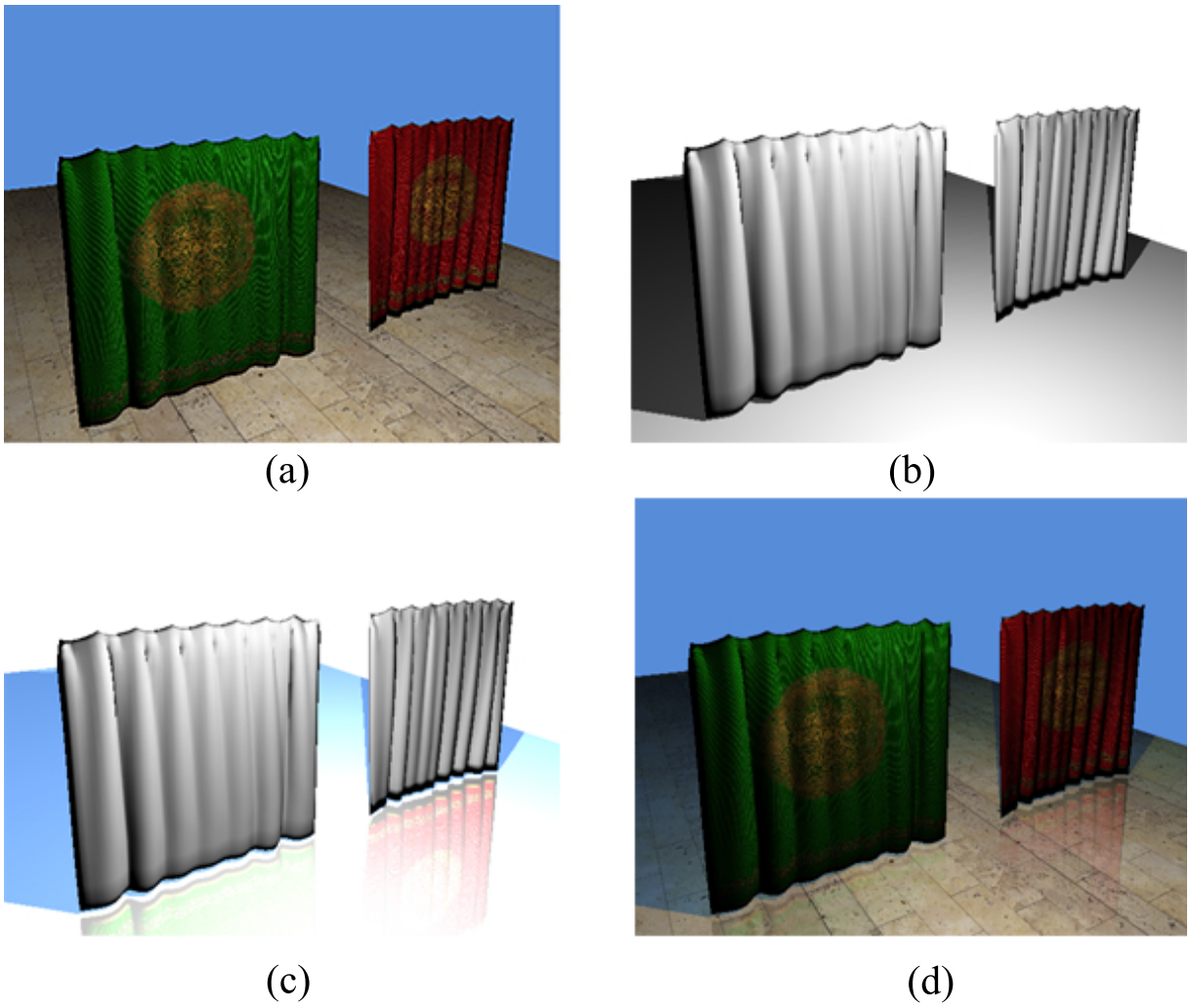


Figura 4.5: Resumo do *pipeline* híbrido de traçado de raios e rasterização. O primeiro estágio *(a)* consiste na renderização da cena utilizando a técnica de *deferred shading*, resultando em um *G-Buffer* preenchido e uma imagem base rasterizada. No estágio *(b)*, raios de sombras são traçados, a partir dos pontos de colisão obtidos do *G-Buffer*, em direção a cena. O estágio *(c)* é responsável por traçar raios secundários de reflexão e refração, onde for aplicável. Neste exemplo, o plano do chão é composto por um material reflexivo. Finalmente, o estágio *(e)* é responsável por compor as imagens geradas nos estágios *(c)* e *(a)*.

Capítulo 5

Implementação

Este capítulo aborda detalhes de implementação levando em consideração aspectos de eficiência e robustez.

O sistema descrito foi implementado utilizando C++ como linguagem principal. Para importar cenas tridimensionais, foi utilizado a biblioteca *Assimp* [3]. Para carregar imagens de texturas foi utilizado a biblioteca *DevIL* [47]. *GLSL* foi escolhida como a linguagem de escrita dos programas de *shader*. A *API OptiX* da *NVIDIA* foi escolhida como motor traçador de raios.

5.1 *G-Buffer*

Uma das desvantagens do uso de *deferred shading* são os requisitos de memória que a técnica exige para preenchimento do *G-Buffer*. Esses requisitos de memória são altos e podem levar a uma degradação de desempenho. Uma das principais otimizações que podem ser feitas ao fazer uso de *deferred shading* é no processo de criação do *G-Buffer*. A escolha de um formato apropriado para os diversos *buffers* que o compõe é de vital importância para o desempenho dessa etapa. A desvantagem de utilizar um formato grande para os *texels*, isto é, *buffers* que utilizam mais de 32 bits por *texel*, é a velocidade baixa de leitura. Por exemplo, *texels* de 64 bits podem demorar até o dobro do tempo necessário para ler um *buffer* com *texels* de 32 bits. Mesmo fazendo uso coerente do cache de textura, formatos largos para *texels* podem degradar muito o desempenho. *Texels* de 128 bits são ainda mais lentos e seu uso deve ser minimizado [41].

Neste trabalho, o *G-Buffer* foi implementado como mostrado na Tabela 5.1. A coluna rotulada de *Formato* mostra a macro do *OpenGL* que descreve o formato dos *texels* e a

coluna rotulada de *Tipo* descreve o tipo de dados armazenados em cada canal do *buffer* correspondente.

Escolhas de formatos para o *G-Buffer*		
Buffer	*Formato*	*Tipo*
Posição	GL_RGBA32F[1]	GL_FLOAT
Normal	GL_RGBA8	GL_UNSIGNED_BYTE
Profundidade	GL_DC24[2]	GL_DC[2]
Albedo	GL_RGBA8	GL_UNSIGNED_BYTE
Especular	GL_RGBA8	GL_UNSIGNED_BYTE
Propriedades Óticas	GL_RGBA32F[1]	GL_FLOAT
Antialiasing	GL_RGBA8	GL_UNSIGNED_BYTE
Destino	GL_RGBA8	GL_UNSIGNED_BYTE

Tabela 5.1: Configurações de formato do *G-Buffer*

Como pode ser visto, o uso de *buffers* com *texels* de 128 bits foi reduzido ao *buffer* de posições e propriedades óticas. A necessidade de uso de 128 bits por *texel* nesses casos ocorre pela necessidade de armazenar valores que estão fora do intervalo $[0, 1]$.

5.1.1 Estrutura de Objetos Implementada com *OptiX*

Na Seção 2.2.4.2 foram descritos os diversos objetos que podem fazer parte de um contexto *OptiX* e também foi descrito qual a função que cada um desses objetos deve desempenhar dentro de um contexto do *OptiX*. Nesta seção, descreve-se como cada um dos programas que podem existir dentro de um contexto *OptiX* foram implementados de forma dar suporte ao funcionamento do traçador de raios híbrido proposto:

- **Programa de geração de raios**: O programa de geração de raios implementado é responsável por calcular as posições e normais dos das superfícies intersectadas através das Equações 3.1a e 3.1b, traçar raios de sombra utilizando o Algoritmo 3.1 e decidir se raios de refração e/ou reflexão devem ser traçados;

- **Programa de qualquer intersecção**: Utilizado para atenuação da luminância carregadas por raios de luz. Quando uma intersecção é encontrada, este programa faz uma busca na textura de transparência do objeto, caso esteja associado a uma, e determina se a colisão deve ser ignorada. O Algoritmo 5.1 descreve este programa;

[1] GL_RGBA32F foi escolhido por causa da necessidade de armazenar as coordenadas no espaço de câmera no *buffer* de posição e para os valores que estão fora do intervalo $[0, 1]$ para o *buffer* de propriedades óticas.

[2] GL_DC é uma abreviação para GL_DEPTH_COMPONENT.

- **Programa de intersecção mais próxima**: Quando a intersecção mais próxima de um raio é encontrada, o ponto de colisão e a sua normal são determinados e é verificado se o nível de recursão máximo foi alcançado para traçar novos raios, caso o material do objeto intersectado necessite de raios adicionais. O Algoritmo 5.2 descreve este programa;

- **Programa de falha**: Quando um raio não intersecta nenhum objeto da cena, o programa de falha é chamado e faz uso das Equações 4.1 para determinar a radiância a ser devolvida por tal raio. Este programa é a aplicação direta das Equações 4.1;

- **Programa de exceção**: Um programa de exceção é chamado caso ocorra algum erro e preenche o *buffer* de saída do *OptiX* com uma especificada;

- **Programa de intersecção**: Neste trabalho foram utilizadas malhas de triângulos para representar diferentes objetos. O programa de intersecção mais próxima calcula a intersecção de um raio com um triângulo e reporta um tripla (β, γ, t) onde β e γ são as coordenadas baricêntricas do ponto intersectado e t é a distância de intersecção. O Algoritmo 5.3 apresenta o pseudocódigo que determina essa intersecção. Uma versão deste programa que trata de malhas deformadas por um esqueleto de controle também foi implementada para determinar colisão de raios com objetos animados;

- **Programa de caixa envolvente**: Computa uma caixa envolvente de um triângulo. Consiste em obter o mínimo e máximo dos componentes de cada vértice. O Algoritmo 5.4 mostra o pseudocódigo deste programa;

5.1.2 Construção do Grafo de Objetos

Nesta seção será descrito o procedimento implementado que transforma um grafo de cena em um grafo de objetos pronto para ser utilizado em um contexto *OptiX*. Este grafo de objetos, juntamente com os programas *OptiX* implementados e descritos na Seção 2.2.4.2, representam o estágio de traçado de raios do *pipeline* híbrido proposto.

Após a cena escolhida ser carrega através do uso da biblioteca *ASSIMP*, um grafo de cena é entregue de forma que a cena representa o nó raiz e os diversos objetos, com suas respectivas transformações, são os nós desse grafo. Câmeras, luzes, malhas, ossos e transformações constituem os possíveis nós desse grafo.

Ao iniciar o procedimento que carrega a geometria da cena para uso do *OptiX*, são criados os objetos que contém os programas de *intersecção* e *caixa envolvente*. Como

Algoritmo 5.1 Programa de qualquer intersecção

Entrada: Raio R
Entrada: Coordenadas de textura (u, v) do ponto de colisão
Entrada: Textura de transparência T se exister
Entrada: Opacidade O do objeto intersectado
Saída: Radiância carregada por R ou ignora intersecção

```
Se Possui textura de trânsparência Então
    Alpha = textura(T, u, v)
    Se Alpha ≤ 0 Então
        Ignore intersecção
    Fim Se
Se não
    Se O < 1 Então
        Determina atenuação de luz dado o material
        Ignore intersecção
    Fim Se
Fim Se
Se Intersecção não foi ignorada Então
    Atenue radiância de R
    Termine raio R
Fim Se
```

Algoritmo 5.2 Programa de intersecção mais próxima

Entrada: Raio R
Entrada: Distância de intersecção t
Entrada: Normal **n** do ponto intersectado
Entrada: Material Mat do objeto intersectado
Saída: Radiância R_R carregada por R

```
h_p = R_o + R_d + t                          ▷ Calcula o ponto intersectado
C_refract = 0                                ▷ Radiância refratada
C_reflect = 0                                ▷ Radiância refletida
S = 0                                        ▷ Atenuação se for area de sombra
Se R_depth < MaxDepth Então
    Se Mat é translucido Então
        C_refract += Radiância refratada
    Fim Se
    Se Mat é reflexivo Então
        C_reflect += Radiância refletida
    Fim Se
    Se h_p estiver sombreado Então
        S = Atenuação de área sombreada      ▷ Algoritmo 3.1
    Fim Se
Fim Se
R_R = S + C_refract + C_reflect
```

Algoritmo 5.3 Intersecção de raio com triângulo

Entrada: Raio R
Entrada: Triângulo $(\boldsymbol{p_0}, \boldsymbol{p_1}, \boldsymbol{p_2})$
Saída: Rejeita ou Aceita
Saída: (β, γ, t)
$e_1 = p_1 - p_0$
$e_2 = p_2 - p_0$
$q = R_d \times e_2$ ▷ R_d é a direção do raio R
$a = e_1 \cdot q$
Se $(a > -\epsilon)$ & $(a < \epsilon)$ **Então**
Retorne Rejeita, $(0, 0, 0)$
Fim Se
$f = 1/a$
$s = R_o - p_0$
$u = f(s \cdot q)$
Se $u < 0$ **Então**
Retorne Rejeita, $(0, 0, 0)$
Fim Se
$r = s \times e_1$
$v = f(d \cdot r)$
Se $(v < 0) \| (u + v) > 1$ **Então**
Retorne Rejeita $(0, 0, 0)$
Fim Se
$t = f(e_2 \cdot r)$
Retorne Aceita, (u, v, t)

Algoritmo 5.4 Programa de caixa envolvente

Entrada: Triângulo $(\boldsymbol{p_0}, \boldsymbol{p_1}, \boldsymbol{p_2})$
Saída: Caixa envolvente B_b ou Invalide
$A = |(p_1 - p_0) \times (p_2 - p_0)|$ ▷ Área do triângulo
Se $A > 0$ **Então**
$B_{b_{min}} = min(p_0, p_1, p_2)$
$B_{b_{max}} = max(p_0, p_1, p_2)$
Se não
Invalide B_b
Fim Se

estão sendo considerados somente objetos formados por malhas de triângulos, os programas de intersecção e caixa envolvente podem ser reaproveitados. Um objeto de *grupo* é criado como sendo a raiz do grafo de modo que todos os objetos estáticos estejam em um ramo enquanto os objetos dinâmicos, se houver algum, façam parte de outro ramo. Isto permite que uma estrutura de aceleração mais adequada seja utilizada para o ramo dinâmico do grafo. A cada nó do grafo de cena está associada uma transformação. Um objeto *transformação* é definido e a matriz associada a este. Sob esta transformação é formado um objeto *grupo* onde cada malha associada ao presente nó é adicionada a um *grupo geométrico* com sua respectiva transformação e *instância geométrica*. O processo prossegue de forma recursiva fazendo uma busca em largura no grafo de cena e criando os objetos necessários para atravessamento de raios utilizando o *OptiX*. O Algoritmo 5.5 descreve o pseudocódigo deste procedimento.

5.1.3 Exemplo de Criação de um Grafo de Objetos

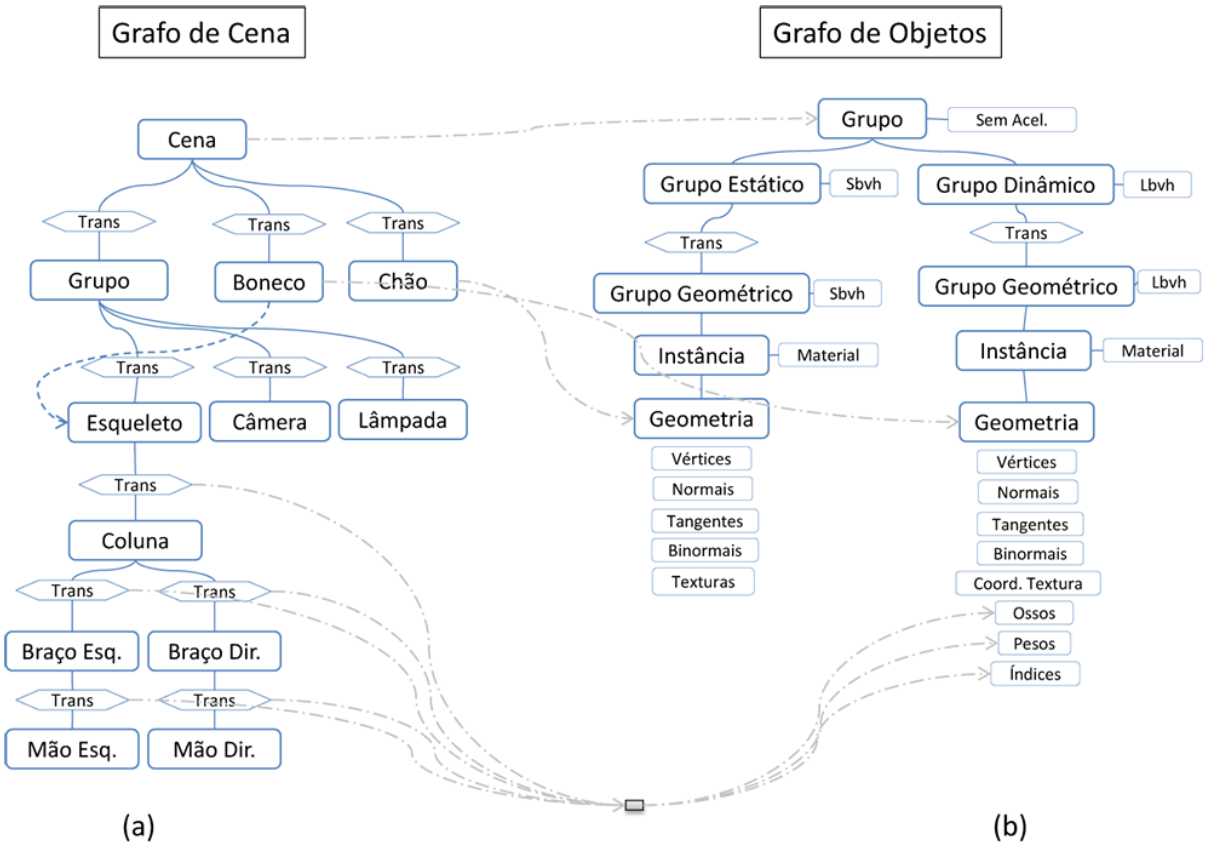


Figura 5.1: Exemplo de um grafo de objetos construído.

Tomando como exemplo o grafo de cena simples mostrado na Figura 5.1a, nesta seção é mostrado um exemplo de criação de um grafo de objetos a ser utilizado em um contexto

Algoritmo 5.5 Construção do grafo de objetos

Entrada: Grafo de cena **G**
Entrada: Raiz do Grafo de Cena R
Saída: Grafo de objetos **G′**

```
Procedimento criaInstância(M)                          ▷ Malha M
    Cria objetos buffer para conter a geometria
    Cria um objeto geometria com os buffers criados
    Se M contém animações Então
        Cria objetos buffer para conter informações de esqueleto
        Associa os objetos buffers ao objeto de geometria
    Fim Se
    Cria um objeto material
    Cria um objeto de instância geométrica
    Associa o material e a geometria à instância geométrica
    Retorne A instância geométrica
Fim Procedimento
Procedimento carregaGeometria(N)          ▷ N é um nó do grafo de cena
    Cria objeto transformação T a partir de N
    Cria objeto grupo G_r
    Para Cada malha M de N Faça
        Cria objeto aceleração A
        Cria um objeto grupo geométrico G_g
        Associa A com G_g
        I ← criaInstância(M)
        Associa I a G_g
        Associa G_g como filho de G_r
    Fim Para
    Para Cada nó N_i filho de N Faça
        T_i ← carregaGeometria(N_i)
        Associa T_i como filho de G_r
    Fim Para
    Associa G_r como filho de T
    Retorne T
Fim Procedimento
Cria objeto aceleração A
Cria objeto grupo G_r como sendo raiz do grafo
Associa A a G_r
T ← carregaGeometria(R)                   ▷ R é o nó raiz do grafo de cena
Associa T como raiz do grafo de objetos G′
```

OptiX destacando a correspondência entre os nós do grafo de cena com os nós do grafo de objetos.

A Figura 5.1a mostra um grafo de cena contendo dois objetos visíveis. O objeto dinâmico *boneco*, que está associado a um esqueleto e um objeto estático *chão*, que consiste de um plano. A cada nó é associado uma transformação que posiciona esse objeto no espaço tridimensional da cena. As transformações do esqueleto variam com o tempo e afetam a forma do boneco. A Figura 5.2 mostra uma representação visual dos objetos que compõe a cena de exemplo.

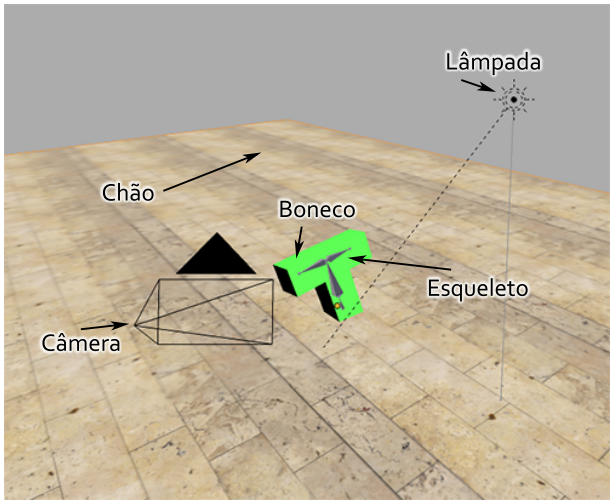


Figura 5.2: Representação visual do grafo de cena de exemplo.

Neste trabalho, objetos dinâmicos são aqueles que podem ser deformados por um esqueleto, todos os outros objetos são considerados estáticos. Através da transformação associada aos objetos estáticos, ao fazer o travessia de um raio pelo grafo de objetos, este é transformado para o sistema de coordenadas do objeto. Isto permite que geometria estática possua animações globais como translação, rotação e escala.

O processo é iniciado pela construção de um objeto *grupo* que será interpretado como o ponto de entrada para raios no grafo. Caso a cena possua algum objeto dinâmico, o grupo inicial terá dois nós filhos, um objeto *grupo* estático, que conterá a partição geométrica que não sofre deformações da cena, e um objeto *grupo* dinâmico, para os objetos que possuem deformações associadas. Em seguida, o nó do grafo de cena *grupo* é visitado mas, após a verificação de que o mesmo não possui nenhuma geometria associada, visita-se o nó *boneco* e um objeto *grupo geométrico* é criado para conter a geometria do objeto. O objeto *boneco* possui associado um esqueleto, portanto, os *buffers* para conter as matrizes de cada osso, os pesos de cada vértice e os índices do vetor de ossos para cada vértice também são criados e associados à geometria do objeto *boneco*. Essa geometria

é adicionada como um nó filho do grupo de objetos dinâmicos no grafo de objetos. Por fim, o nó contendo o objeto *chão* é visitado e um objeto *grupo geométrico* é criado para conter a geometria deste objeto. Em seguida, é adicionado como filho do grupo de objetos estáticos.

Cada objeto do tipo *grupo* deve possuir uma estrutura de aceleração associada. Para o *grupo* de objetos estáticos foi escolhida a estrutura *Sbvh*[42] por se tratar de um hierarquia de volumes envolventes de alta qualidade. A implementação do *OptiX* dessa estrutura fornece uma melhoria significativa quando utilizada em uma geometria não uniforme, como triângulos de diferentes tamanhos. Para o ramo dinâmico do grafo foi escolhida a estrutura *Lbvh*[17]. Essa estrutura utiliza um método rápido de construção de hierarquias de volumes envolventes em GPU e é recomendada para aplicações que possuem uma geometria muito grande ou conteúdo animado. Para o *grupo* raiz do grafo de objetos não foi utilizado uma estrutura de aceleração pois é sabido que este grupo terá no máximo dois nós filhos. A não utilização de estruturas de aceleração é recomendada para casos muitos simples, como este, onde um nó possui poucos nós filhos.

Capítulo 6

Resultados

Neste capítulo serão mostrados os resultados obtidos com a implementação do renderizador híbrido proposto em termos de desempenho e qualidade. Os resultados são comparados com uma implementação de um traçador de raios em GPU, que também faz uso do *OptiX*. Neste traçador de raios de comparação, ao invés de serem utilizadas informações armazenadas no G-Buffer, os raios primários são disparados a partir da câmera. Nesta configuração o *OpenGL* somente é utilizado para mostrar a imagem gerada.

Os testes foram efetuados em uma máquina *desktop* equipada com um processador AMD Phenom II X4 965 3.4GHz com 16GB de memória RAM e uma GPU *NVIDIA GeForce* GTX570. Como sistema operacional foi utilizado o *Microsoft Windows* 7 *Professional* versão de 64bits.

A Tabela 6.1 sumariza os resultados em termos de desempenho com medidas baseadas na taxa de quadros por segundo (QPS) para cada cena. Todas as cenas foram renderizadas com resolução de 1280x720 *pixels*.

Comparação de desempenho em QPS					
Cena	*Vértices*	*Triângulos*	**DR**	**RT**	**Híbrido**
Sponza (Fig. 6.2)	145.173	262.187	449	15	29
Sponza Animada (Fig. 6.3)	184.178	266.923	388	8	18
Vitrine (Fig 6.4)	112.603	224.440	428	25	43
Armadillo (Fig. 6.5)	193.737	380.655	319	40	54
Sala de Jantar (Fig. 6.6)	386.541	224.954	481	13	32

Tabela 6.1: Comparação de desempenho em QPS do renderizador híbrido proposto com uma implementação pura de traçado de raios.

Na Tabela 6.1, *DR* representa o tempo de renderização da cena utilizando *deferred shading*. *RT* é o tempo necessário para renderizar a cena utilizando um traçador de raios

puro. A coluna *Híbrido* apresenta os resultados em quadros por segundo da implementação híbrida proposta.

A Tabela 6.2 foi construída de forma a mostrar o tempo gasto pela aplicação implementada em cada um dos estágios. *Amb* representa o tempo necessário para renderização do mapa de ambiente de fundo. *DR* e *RT* são os tempos gastos nos estágios de *deferred rendering* e traçado de raios híbrido, respectivamente. O tempo gasto com a etapa de composição é mostrado na coluna rotulada como *Comp*. A última coluna foi reservada para o tempo gasto na renderização da cena utilizando um algoritmo de traçado de raios que dispara raios de câmera como forma de amostragem de cor *pixels*.

Tempo gasto em *ms* em cada estágio					
Cena	*Amb*	*DR*	*HRT*	*Comp*	*PRT*
Sponza	0,026	0,830	29,214	0,050	64,648
Sponza Animada	0,027	0,98775	53,454	0,054	123,105
Vitrine	0,027	0,733	35,811	0,044	38,131
Armadillo	0,027	0,585	15,251	0,039	23,378
Sala de Jantar	0,076	0,804	27,346	0,057	74,923

Tabela 6.2: Tempo gasto em cada etapa do *pipeline* híbrido.

A Figura 6.3 apresenta um exemplo de cena animada. As esferas que aparecem com materiais reflexivos e refrativos fazem uma volta de 360 graus em relação a origem da cena. O gráfico apresentado na Figura 6.1 mostra os resultados da Tabela 6.1 de forma visual.

Como pode ser visto na Tabela 6.1 e no gráfico da Figura 6.1, a implementação do traçador de raios híbrido foi capaz de atingir uma taxa de quadros por segundo superior a uma implementação pura de traçado de raios.

Na Tabela 6.2 é possível perceber que os tempos necessários para renderização do mapa de ambiente e para o estágio de composição são quase nulos, logo, não possuem impacto na taxa de quadros gerada. Como pode ser visto, o estágio de traçado de raios híbrido é responsável por, em média, 97% do tempo gasto para renderizar as cenas. Isto implica que este estágio deve ser o alvo de otimizações que possam ser feitas na aplicação. Essas otimizações incluem a redução do número de instruções divergentes nos programas implementados com *OptiX* além da otimização das operações utilizadas nos cálculos dos modelos de iluminação.

É fácil perceber que as cenas das Figuras 6.4 e 6.5 possuem uma taxa de quadros por segundo superior, mesmo com um número de vértices e faces também maior. Isso acontece pois essas duas se tratam de cenas abertas, portanto, muitos raios, que são refletidos ou

refratados, escapam da cena sem colidir com nenhum objeto, deixando para o programa de *falha* tratar este caso. Como este programa somente faz uma amostragem da textura do mapa de ambiente temos que o número de quadros por segundo nessas cenas fica maior.

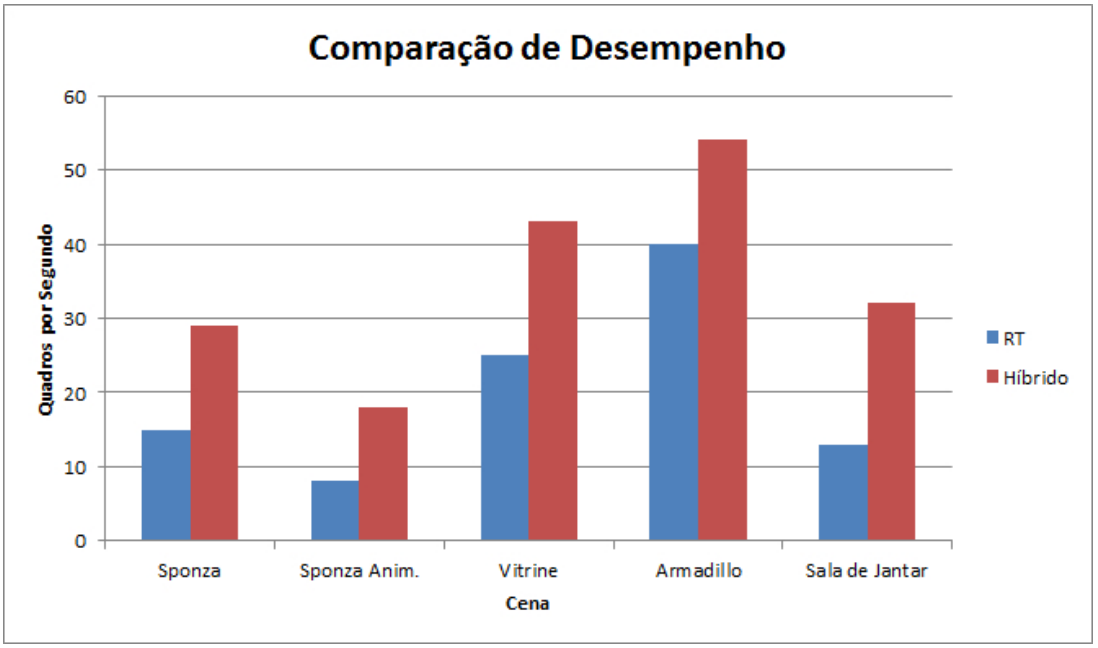


Figura 6.1: Gráfico de comparação de desempenho.

Apesar da variação em termos de QPS de uma cena para outra ser significativa, a diferença de QPS entre as implementações híbridas e traçado de raios pura é, em média, de 15 quadros por segundo. Um aumento de 15 QPS na renderização de uma cena pode ser o diferencial entre uma aplicação interativa e totalmente em tempo real. Temos, portanto, que a eliminação do estágio de traçado e colisão de raios primários partindo da câmera, pode acelerar o *pipeline* de traçado de raios de maneira significativa.

Como discutido de forma geral no Capítulo 2 e com mais detalhes na Seção 2.2, muitos problemas que são difíceis de serem resolvidos utilizando técnicas de rasterização, podem ser resolvidos de maneira simples com o uso de traçado de raios. A partir da incorporação de um estágio de traçado de raios fazendo uso de informações produzidas por *deferred shading*, é possível adicionar efeitos raio-traçados em cenas rasterizadas de forma a atender os requisitos de uma aplicação interativa de tempo real.

Figura 6.2: Cena 1: Sponza. Exemplo de composição com material reflexivo e refrativo.

Figura 6.3: Cena 2: Sponza. Exemplo de composição com material reflexivo e refrativo.

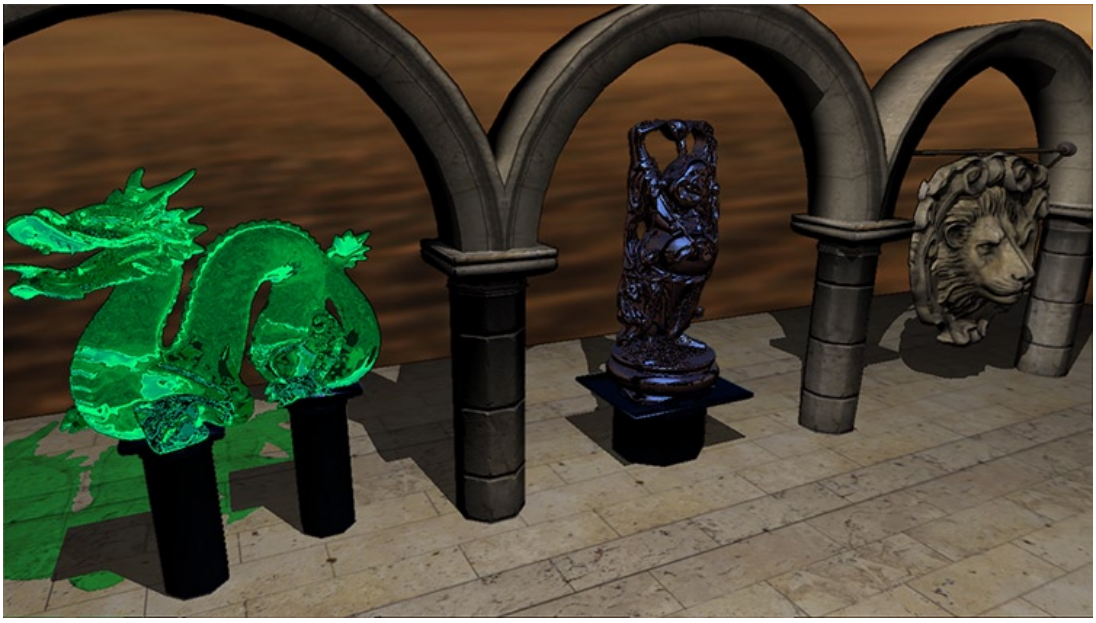


Figura 6.4: Cena 3: Vitrine. Exemplo de cena com material refrativo. Destaque para a coloração da sombra.

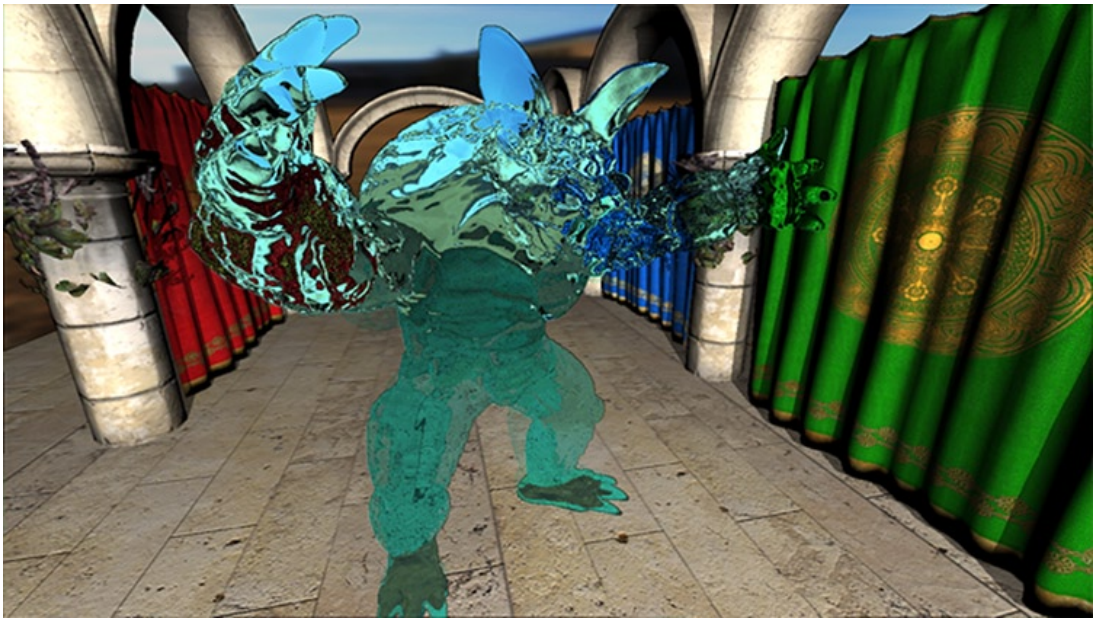


Figura 6.5: Cena 4: Armadilo: Exemplo de cena com material refrativo.

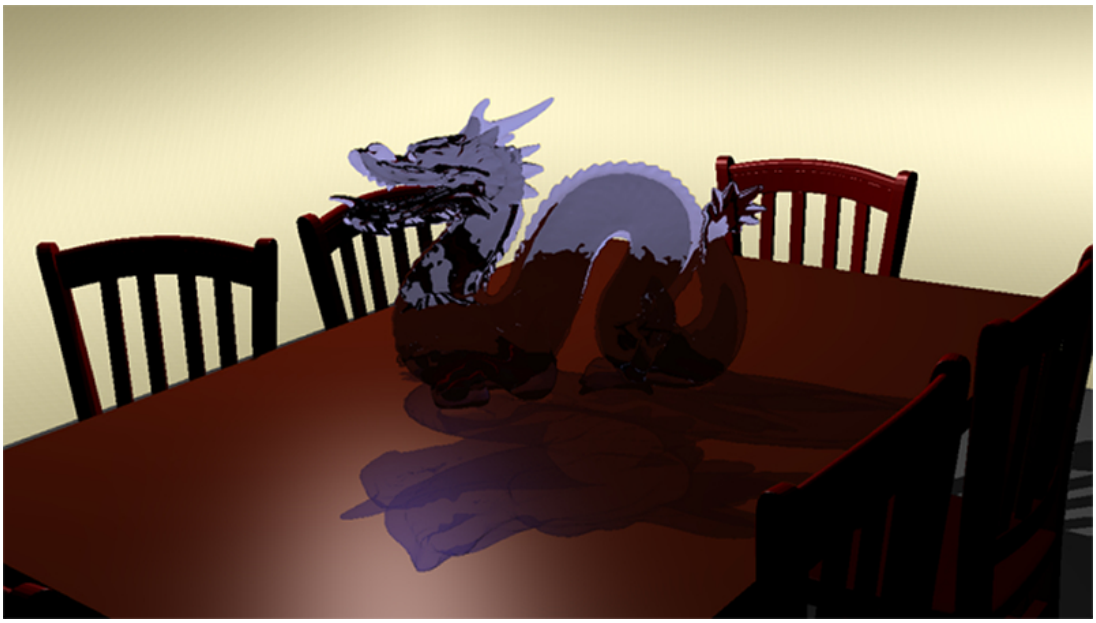


Figura 6.6: Cena 5: Sala de jantar. Exemplo de cena com material refrativo.

Capítulo 7

Conclusão e Trabalhos Futuros

Neste trabalho, foi descrita uma abordagem de traçado de raios onde os raios primários de câmera são resolvidos utilizando dados gerados pela técnica de rasterização *deferred shading*. O uso desta técnica permite utilizar o algoritmo de *Z-Buffer* e os dados armazenados no *G-Buffer* de forma a evitar traçar raios primários de câmera.

Foram discutidos teoria e prática das técnicas utilizadas na implementação deste trabalho bem como foram levantados os pontos positivos e negativos do uso de *deferred shading* e traçado de raios em aplicações interativas de tempo real. Apresentou-se os principais detalhes do motor traçador de raios da *NVIDIA*, *OptiX* e também foi apresentada uma maneira de transformar grafos de cena em grafos de objetos para uso do *OptiX* de forma automática.

Resultados foram apresentados em termos de desempenho para diferentes cenas e foi mostrado que a implementação híbrida proposta pode acelerar o traçado de raios, tornando o traçado e colisão de raios primários uma tarefa de rasterização. Isto permite adicionar efeitos raio-traçados em cenas rasterizadas. Destaca-se que o sistema foi construído de forma que programas de *shader* já existentes possam ser utilizados com poucas ou nenhuma modificação, fazendo uso do efeitos criados até o momento.

Como trabalhos futuros, propõe-se a geração de sombras com uso da técnica de mapa de sombras, deixando para o traçador de raios, somente efeitos difíceis de serem resolvidos com rasterização. Apesar dos problema de *aliasing* apresentados pela técnica clássica de mapa de sombras, isto pode reduzir drasticamente o número de raios a serem traçados. Propõe-se, também, a implementação de efeitos visuais, através de programas de *shaders*, mais sofisticados a fim de aumentar a qualidade visual da cena gerada. A melhoria dos modelos de iluminação no estágio de traçado de raios também é proposta de forma a

melhorar a qualidade da imagem final gerada.

Uma das vantagens do traçado de raios sobre técnicas de rasterização é a sua facilidade para renderizar superfícies curvas. A representação implícita de superfícies requer somente o armazenamento dos parâmetros associados e os algoritmos de intersecção são mais rápidos com esse tipo de representação. Propõe-se a incorporação de superfícies paramétricas nesta implementação híbrida de forma a obter superfícies curvas em cenas rasterizadas.

Com o objetivo de melhorar o desempenho da etapa de traçado de raios do *pipeline* implementado, propõe-se o desenvolvimento de heurísticas para selecionar objetos mais relevantes dinamicamente para o traçado de raios. Os objetos selecionados seriam raio-traçados em um tempo fixo predefinido. Essa heurística deve ser capaz de selecionar os objetos mais relevantes em uma cena e ainda assim manter uma taxa de frames elevada. Como relevantes, define-se aqueles objetos que mais contribuem para a experiência visual do usuário em um tempo específico.

Oferecendo um poder de processamento muito maior do que a geração anterior de GPUs e fornecendo novos métodos de otimização de execução paralela de código na própria GPU, a arquitetura *Kepler* GK110 [34] simplifica a criação de programas paralelos. A nova funcionalidade de paralelismo dinâmico permite a geração de trabalho, sincronização e controle de tarefas pela própria GPU. Na área de traçado de raios, essa nova funcionalidade pode ser utilizada para um refinamento progressivo das regiões onde um número maior de amostras deve ser tomado. A partir do traçado e colisão de raios primários, novos *kernels* podem ser disparados de forma a concentrar o trabalho onde o número de amostras a ser tomado é mais denso.

Como foi discutido, o traçado de raios é uma técnica capaz de sintetizar imagens de alta qualidade. Através de implementações mais eficientes, junto com o avanço do *hardware* gráfico, no futuro próximo, será possível simular efeitos antes somente possíveis em renderizadores *offline*. No futuro, poderemos ver efeitos de iluminação global aplicados em tempo real e a altas taxas de quadros dentro de jogos e ambientes de visualização.

Referências

[1] Aila, T.; Laine, S. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009* (2009), pp. 145–149.

[2] Akenine-Möller, T.; Haines, E.; Hoffman, N. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[3] ASSIMP. Open asset import library. Webpage, 10 2011.

[4] Bak, P. Real time ray tracing. Master's thesis, IMM, DTU, 2010.

[5] Barbosa, D. C. Implementação do ray tracing acelerado por uma octree em gpu. Master's thesis, Universidade Federal Fluminense, 2011.

[6] Beck, S.; c. Bernstein, A.; Danch, D.; Frohlich, B. Cpu-gpu hybrid real time ray tracing framework, 2005.

[7] Benthin, C.; Wald, I.; Scherbaum, M.; Friedrich, H. Ray tracing on the cell processor. pp. 15–23.

[8] Bigler, J.; Stephens, A.; Parker, S. Design for parallel interactive ray tracing systems. In *Interactive Ray Tracing 2006, IEEE Symposium on* (sept. 2006), pp. 187 –196.

[9] Bikker, J. Real-time ray tracing through the eyes of a game developer. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 1–10.

[10] Chen, C.-C.; Liu, D. S.-M. Use of hardware z-buffered rasterization to accelerate ray tracing. In *Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 1046–1050.

[11] Christensen, P. H.; Fong, J.; Laur, D. M.; Batali, D. Ray Tracing for the Movie 'Cars'. *Symposium on Interactive Ray Tracing 0* (2006), 1–6.

[12] Cook, R. L. Stochastic sampling in computer graphics. *ACM Trans. Graph. 5*, 1 (Jan. 1986), 51–72.

[13] Cook, R. L.; Carpenter, L.; Catmull, E. The reyes image rendering architecture. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), SIGGRAPH '87, ACM, pp. 95–102.

[14] Deering, M.; Winner, S.; Schediwy, B.; Duffy, C.; Hunt, N. The triangle processor and normal vector shader: a vlsi system for high performance graphics. vol. 22, ACM, pp. 21–30.

[15] Foley, T.; Sugerman, J. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), HWWS '05, ACM, pp. 15–22.

[16] Friedrich, H.; Günther, J.; Dietrich, A.; Scherbaum, M.; Seidel, H.-P.; Slusallek, P. Exploring the use of ray tracing for future games. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (New York, NY, USA, 2006), Sandbox '06, ACM, pp. 41–50.

[17] Garanzha, K.; Pantaleoni, J.; McAllister, D. Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 59–64.

[18] Glassner, A. S., Ed. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989.

[19] Gunther, J.; Popov, S.; Seidel, H.-P.; Slusallek, P. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 113–118.

[20] Hachisuka, T. Ray tracing on graphics hardware. Tech. rep., University of California at San Diego, 2009.

[21] Heckbert, P. S. Graphics gems iv. Academic Press Professional, Inc., San Diego, CA, USA, 1994, ch. A minimal ray tracer, pp. 375–381.

[22] Horn, D. R.; Sugerman, J.; Houston, M.; Hanrahan, P. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 167–174.

[23] Hou, Q.; Qin, H.; Li, W.; Guo, B.; Zhou, K. Micropolygon ray tracing with defocus and motion blur. In *ACM SIGGRAPH 2010 papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 64:1–64:10.

[24] Jin, B.; Ihm, I.; Chang, B.; Park, C.; Lee, W.; Jung, S. Selective and adaptive supersampling for real-time ray tracing. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 117–125.

[25] Kajiya, J. T. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), SIGGRAPH '86, ACM, pp. 143–150.

[26] Kaplanyan, A. Real-time diffuse global illumination in cryengine 3. Presentation, 2010.

[27] Keller, A.; Wachter, C. To trace or not to trace, that is the question. Presentation, 2005.

[28] Kirk, D. B.; Hwu, W.-m. W. *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[29] Lauterbach, C.; Garland, M.; Sengupta, S.; Luebke, D.; Manocha, D. Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2, 375–384.

[30] Madeira, D. L. A. Uma estrutura baseada em hash table para buscas otimizadas em octree em gpu. Master's thesis, Universidade Federal Fluminense.

[31] Martins, M. A. Traçado de raios de cenas dinâmicas em cuda. Master's thesis, Universidade Federal do Mato Grosso do Sul, 2010.

[32] McGuire, M.; Luebke, D. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics* (New York, NY, USA, August 2009), ACM.

[33] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[34] NVIDIA. Nvidia's next generation cuda compute architecture. Tech. rep., NVIDIA Corporation, 2012.

[35] Parker, S. G.; Bigler, J.; Dietrich, A.; Friedrich, H.; Hoberock, J.; Luebke, D.; McAllister, D.; McGuire, M.; Morley, K.; Robison, A.; Stich, M. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics* (August 2010).

[36] Pharr, M.; Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[37] Reinhard, E.; Kok, A. J. F.; Jansen, F. W. Cost prediction in ray tracing. Tech. rep., Bristol, UK, UK, 1996.

[38] Reshetov, A. Morphological antialiasing. In *Proceedings of the 2009 ACM Symposium on High Performance Graphics* (2009).

[39] Sabino, T.; Andrade, P.; Lattari, L.; Clua, E.; Montenegro, A.; Pagliosa, P. Efficient use of in-game ray-tracing techniques. In *Proceedings of the X Brazilian Symposium on Computer Games and Digital Entertainment - Computing Track - Short papers* (Porto Alegre, Rio Grande do Sul, Brasil, 2011), Sociedade Brasileira da Computação.

[40] Saito, T.; Takahashi, T. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4 (Sept. 1990), 197–206.

[41] Shishkovtov, O. Deffered shading in s.t.a.l.k.e.r. *GPU Gems 2 2* (2005), 143–166.

[42] Stich, M.; Friedrich, H.; Dietrich, A. Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics 2009* (2009).

[43] Wachter, C.; Keller, A. Instant ray tracing: The bounding interval hierarchy. In *In Rendering Techniques 2006: Proceedings of the 17th Eurographics Synposium on Rendering* (2006), pp. 139–149.

[44] WALD, I. *Realtime Ray Tracing and Interactive Global Illumination.* Tese de Doutorado, Computer Graphics Group, Saarland University, 2004.

[45] WHITTED, T. An improved illumination model for shaded display. *Commun. ACM 23* (June 1980), 343–349.

[46] WILLIAMS, L. Casting curved shadows on curved surfaces. In *In Computer Graphics (SIGGRAPH 1978 Proceedings* (1978), pp. 270–274.

[47] WOODS, D.; WEBER, N.; DARIO, M. Developer's image library. Webpage, 2001.