# Universidade Federal de Minas Gerais
# Computação Gráfica Avançada
# Trabalho Prático – Path Tracing

Vinicius Graciano Santos

26 de Novembro de 2015

## Conteúdo



## 1 Objetivo

O objetivo do trabalho é implementar um *path tracer* básico. O programa deve possuir no mínimo as funcionalidas descritas na tabela abaixo.

| | |
|---|---|
| **Iluminação Direta** | Luzes Puntuais e BRDF de Blinn-Phong |
| **Iluminação Indireta** | Reflexões e Refrações |
| **Mapeamento de texturas** | Planar (formato PPM) |
| **Textura procedural** | Padrão de tabuleiro |
| **Descrição das cenas** | Definidas em um arquivo externo |

Tabela 1: Funcionalidades básicas

Com relação à organização desta documentação, a Seção 2 apresenta as decisões de implementação e as funcionalidades extras inclusas no trabalho, e a Seção 3 mostra as imagens geradas pelo programa desenvolvido. Por fim, a Seção 4 descreve as instruções para compilar o código fonte e executar o *path tracer*.

# 2 Metodologia

## 2.1 Path Tracing na GPU

O path tracer foi implementado para rodar na GPU utilizando o pipeline gráfico programável da OpenGL. Basicamente, apenas dois triângulos são submetido para a placa gráfica de forma a cobrir totalmente a *viewport* e um *fragment shader* é responsável pelo processo de path tracing durante o estágio de rasterização. Essa metodologia permite que o usuário acompanhe a convergência do algoritmo, já que o valor atual do estimador Monte Carlo pode ser diretamente renderizado em uma janela do sistema operacional.

*Fragment shaders* não suportam métodos recursivos devido a certas limitações do pipeline gráfico. Dessa maneira, foi necessário transformar o *path tracer* em um método iterativo. Felizmente, o algoritmo possui recursão de cauda e logo pode ser facilmente escrito como uma único estrutura de repetição indexada pelo número de vértices no caminho de um raio.

A maior dificuldade na implementação foi a geração de números aleatórios na GPU e o cálculo do valor esperado das amostras, visto que uma execução de um *fragment shader* não possui realimentação de estados. Dessa maneira, para cada execução do shader, foi necessário renderizar dois mapas de texturas, nos quais o último valor aleatório gerado e a média móvel das amostras foram salvos para cada pixel na imagem. Essas texturas são utilizadas como entradas da próxima execução do shader, realimentando a semente aleatória e o estimador Monte Carlo, respectivamente.

## 2.2 Sphere-tracing

Para disparar os raios em cada fragmento e identificar os pontos de colisão com os objetos na cena foi implementado um método denominado *sphere-tracing* [1]. Nesse contexto, um objeto deve ser descrito como uma função $f : \mathbb{R}^3 \mapsto \mathbb{R}$ que mapeia um ponto no espaço para a distância até a superfície do objeto, especificada pelo conjunto $\mathcal{S} = \{x \in \mathbb{R}^3 \mid f(x) = 0\}$. Assim, dado o raio $r_0 + tr_d$, sendo $r_0 \in \mathbb{R}^3$ o ponto de origem, $r_d \in \mathbb{R}^3$ uma direção normalizada e $t \in \mathbb{R}$, basta encontrar o valor mínimo para $t$ que leve o raio a interceptar a superfície do objeto, isto é,

$$t_{min} = \arg\min_t f(r_0 + tr_d) = 0.$$

O método *sphere-tracing* caminha sobre o raio em incrementos de $f(r_0 + tr_d)$, já que por definição a esfera centrada em $r_0 + tr_d$ de raio $f(r_0 + tr_d)$ garantidamente não contém objeto algum em seu interior, então é possível incrementar o valor de $t$ em $f(r_0 + tr_d)$ sem riscos de colisão. A Figura 1 ilustra as iterações do algoritmo de um ponto inicial até atingir uma superfície.

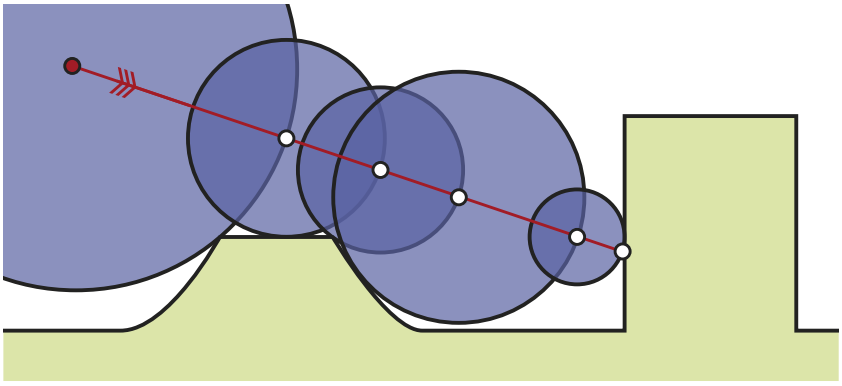


Figura 1: Ilustração dos passos do algoritmo sphere-tracing (Fonte: [2])

As vantagens do método sphere-tracing sobre as soluções analíticas de interseção entre raios e superfícies são várias. Por exemplo, para encontrar o vetor normal em um ponto $x$ basta normalizar o vetor $\nabla f(x)$, que pode ser aproximado pelo método de diferenças centrais. Além disso, técnicas de modelagem CSG podem ser facilmente implementadas através dos operadores min e max, já que as superfícies são definidas por funções implícitas. Finalmente, não é necessário encontrar a função de

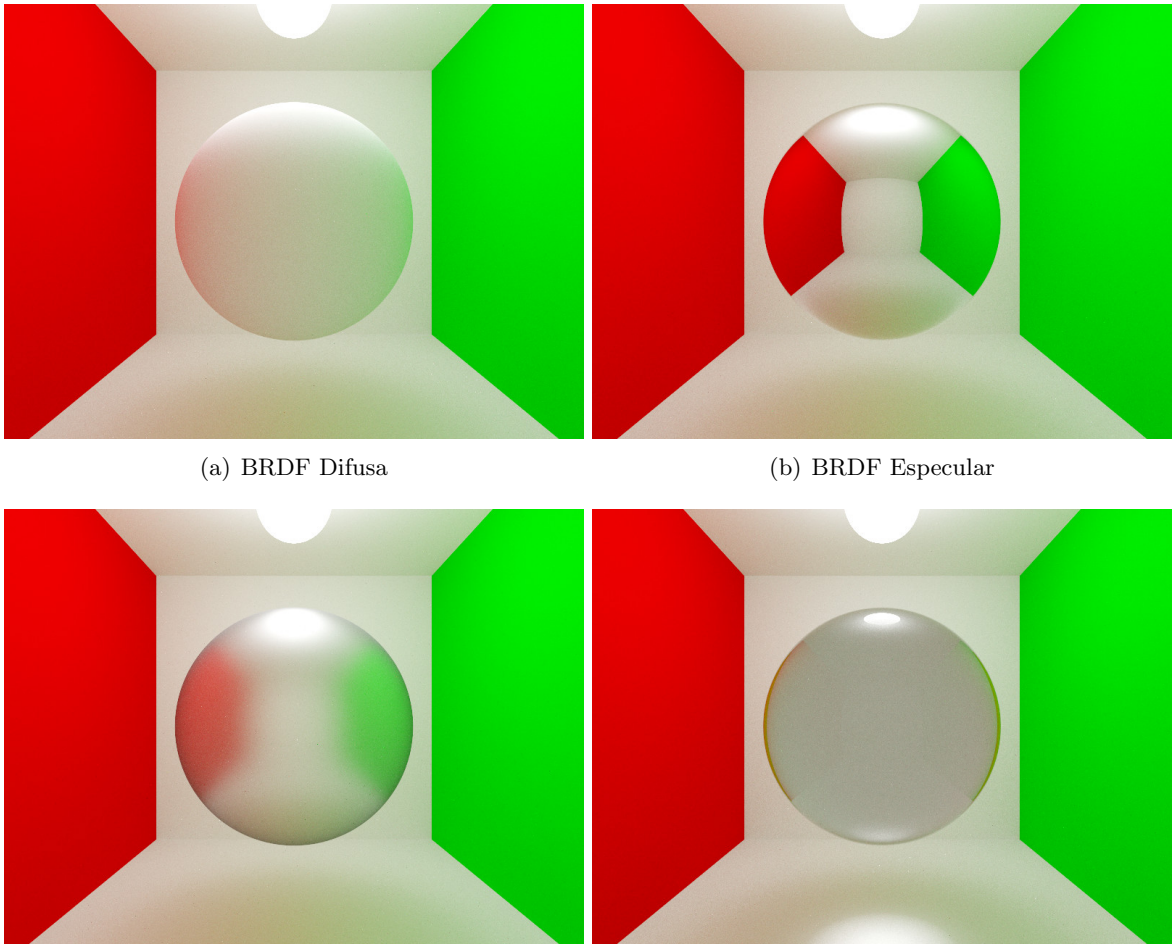
distância exata até a superfície que se deseja renderizar, mas apenas um limite superior é requerido pelo método. Com isso, é possível especificar objetos complexos como curvas de alta ordem e até mesmo fractais.

## 2.3 Bidirectional Scattering Distribution Functions

As seguintes funções de espalhamento dos raios foram implementadas:

- BRDF Difusa, Especular e Blinn-Phong.
- BSDF Transmissiva com reflexões internas e totais.

Com exceção da BRDF difusa, todas as BxDFs empregam o **Efeito Fresnel** a partir da aproximação de Schlick para materiais dielétricos. A figura abaixo mostra diversas esferas renderizadas com esses modelos de espalhamento.



(a) BRDF Difusa

(b) BRDF Especular

(c) BRDF de Blinn-Phong

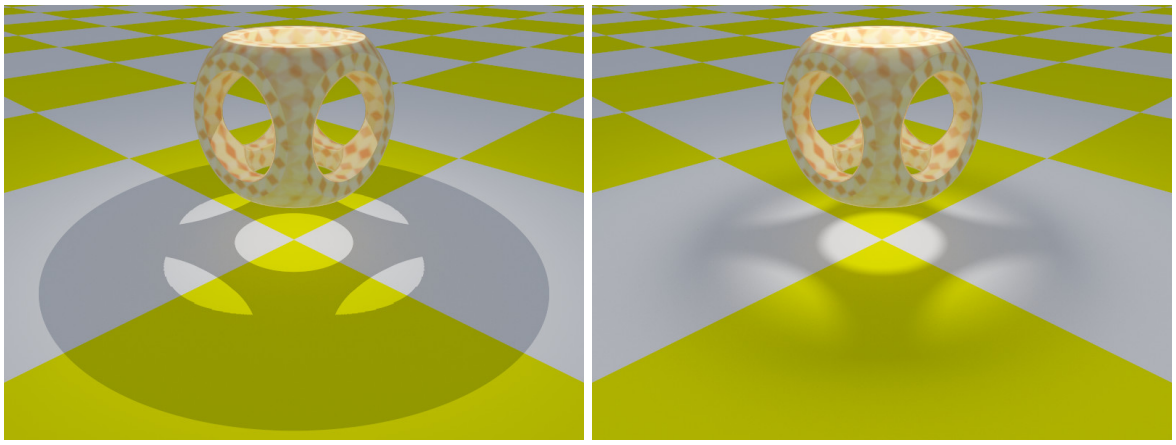
(d) BTDF Transmissiva

Figura 2: Esferas renderizadas com diferentes materiais dentro de uma Caixa de Cornell.

## 2.4 Luzes Puntuais e com Área

O renderizador permite o uso de luzes puntuais e/ou esféricas para iluminar a cena. Além disso, se a geometria da mesma não for fechada — por exemplo, como na Caixa de Cornell — uma luz que encapsula toda a cena (*environment light*) será utilizada para retornar a radiância dos raios que não atingirem objeto algum. A Figura 3 mostra duas cenas similares sendo iluminadas por luzes distintas: uma puntual e uma esférica. Perceba como o uso da luz esférica cria uma região de transição suave entre a umbra e a penumbra.

Luzes com área requerem estratégias mais eficientes de amostragem para que a a eficiência do *path tracer* não seja comprometida. Esses detalhes são discutidos na próxima seção.



(a) Luz puntual (b) Luz esférica

Figura 3: Sombras geradas por diferentes tipos de luzes.

## 2.5 Amostragem por Importância, Roleta Russa e Antialiasing

*Amostragem por Importância* consiste em retirar amostras de uma distribuição que seja a mais similar possível à distribuição sendo integrada pelo estimador de Monte Carlo. No trabalho foram implementadas algumas técnicas que variam de acordo com a BxDF das superfícies e a geometria das luzes. Resumidamente, os seguintes métodos estão inclusos:

| | |
|---|---|
| BRDF Difusa | Cosine-weighted Sampling |
| BRDF de Blinn-Phong | Half Vector Sampling |
| Luz Esférica | Solid Angle Sampling |

Tabela 2: Métodos de Amostragem por Importância implementados no renderizador.

A **BRDF difusa** é amostrada de acordo com o Método de Malley: um disco unitário é amostrado uniformemente e a amostra é elevada no eixo perpendicular ao disco até a superfície de uma esfera unitária. Na **BRDF de Blinn-Phong**, o *half vector* é amostrado e a direção de entrada é refletida sobre ele para obter uma amostra na direção de saída da luz. O processamento é feito em coordenadas esféricas, cujos ângulos $(\phi, \theta)$ são amostrados de acordo com uma distribuição uniforme e uma de potência (*power distribution*), respectivamente, sendo o último de acordo com o parâmetro $\alpha$ da BRDF.

As **luzes esféricas** são amostradas baseadas no ângulo sólido formado pela luz e pelo ponto $p$ sobre o qual a equação de iluminação está sendo calculada. Dessa maneira, apenas amostras visíveis em relação a esse ponto serão geradas na esfera. O processo consiste em amostrar uniformemente direções em um cone, com ápice em $p$, que aponta na direção do centro da esfera, e depois projetá-las na esfera com uma operação de *raycast*.

A técnica de **roleta russa** é utilizada para terminar os raios com maior probabilidade conforme a radiância total do caminho diminui. No entanto, ela é ativada depois das quatro primeiras rebatidas/transmissões da luz, o que garante uma convergência mais rápida da renderização da iluminação direta, reflexões primárias e transparências. Mesmo assim, o caminho máximo de um raio de luz é limitado a 16 vértices, por motivos de eficiência.

Para evitar a subamostragem nas bordas dos objetos, o renderizador escolhe uma posição aleatória dentro de cada pixel antes de iniciar a amostra de um caminho para um raio de luz (o peso do amostrador Monte Carlo é ajustado de acordo). Isso garante que diversas amostras sejam avaliadas dentro de um único pixel, evitando problemas de **aliasing** nas bordas dos objetos.

Todos os métodos de amostragem por importância foram retirados do livro *Physically-baseD Rendering* [3].

## 2.6 Geometria Sólida Construtiva com Operadores Suaves

Como todas as superfícies são representadas por funções implícitas, operações de união, interseção e diferença podem ser trivialmente implementadas através dos operadores max e min. Considere duas superfícies $\partial f$ e $\partial g$ representadas pelas funções de distância $f : \mathbb{R}^3 \mapsto \mathbb{R}$ e $g : \mathbb{R}^3 \mapsto \mathbb{R}$. Assim, para encontrar as superfícies resultantes das operações booleanas basta utilzar o método de sphere-tracing sobre as seguintes funções.

| | |
|---|---|
| $\partial f \cup \partial g$ | $\min(f(x), g(x))$ |
| $\partial f \cap \partial g$ | $\max(f(x), g(x))$ |
| $\partial f \setminus \partial g$ | $\max(f(x), -g(x))$ |

Tabela 3: Funções de distância para operações de geometria sólida construtiva.

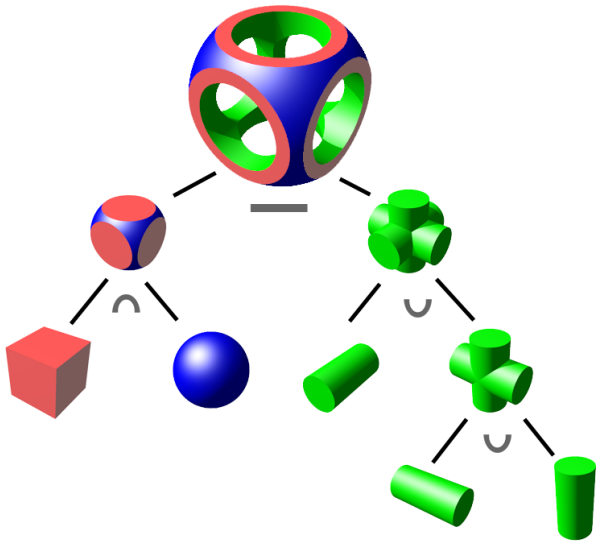


Figura 4: Exemplo de geometria sólida construtiva.

Além disso também é possível substituir as funções min e max por operados suaves como a função *smoothmin*, que é geralmente utilizada para retirar discontinuidades em problemas de otimização que requerem a existência das derivadas por todo o espaço de busca.

$$\text{smoothmin}(a, b) = -\log(e^{-ka} + e^{-kb})$$

## 2.7 Mapeamento Triplanar de Texturas com Filtragem Trilinear

O mapeamento de texturas descrito na especificação do trabalho foi alterado para comportar um método conhecido como mapeamento triplanar, que é razoavelmente apropriado para ser utilizado em superfícies genéricas, gerando poucas distorções indesejadas durante mapeamento. A ideia é mapear a textura em um cubo ao redor do objeto e projetá-la sobre a superfície do objeto, de maneira semelhante ao famoso *cube mapping* ou *environment mapping* (Figura 5). No entanto, a textura é ponderada pela direção da normal no ponto sendo processado. Esse último passo é fundamental e garante que em média as texturas dos diferentes planos serão combinadas de maneira suave de acordo com a curvatura da superfície.

Seja $p$ e $n$ um ponto na superfície e seu respectivo vetor normal e assuma uma função $\text{tex}(u, v) : \mathbb{R}^2 \mapsto \mathbb{R}^3$, sendo a imagem da função a cor (RGB) da textura. Assim, a cor $c$ retornada pelo mapea-

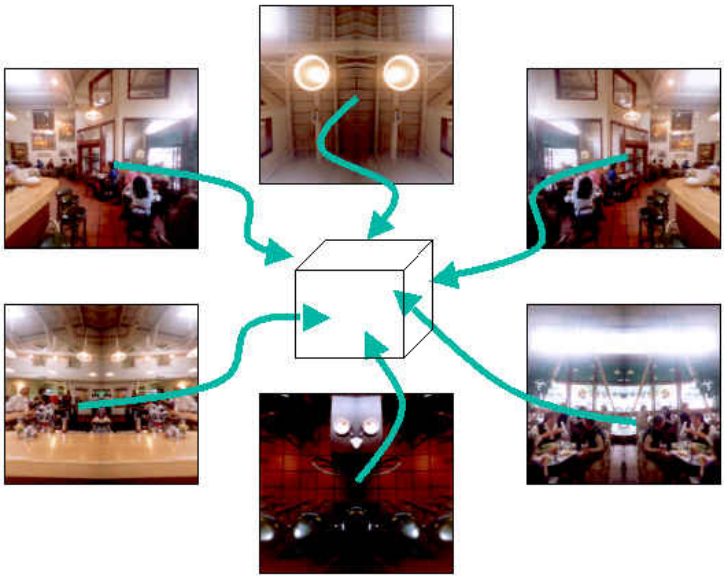


Figura 5: Environment Mapping.

mento triplanar é definida como

$$c = \frac{n_x \text{tex}(p_y, p_z) + n_y \text{tex}(p_x, p_z) + n_z \text{tex}(p_x, p_y)}{n_x + n_y + n_z}$$

O resultado do mapeamento triplanar é mostrado na Figura 6. Note como a aplicação da textura respeita de maneira automática a curvatura do objeto. O mapeamento não é satisfatório apenas para o cone ao fundo da cena, visto que existem distorções indesejadas. Para evitar *aliasing* nas texturas, devido ao uso do hardware gráfico, a filtrigem trilinar de texturas foi ativada (filtro bilateral linear com mipmaps). Além disso, o arquivo de entrada foi alterado e os parâmetros de textura da especificação foram removidos. Um novo parâmetro de escala foi adicionado no lugar para variar o tamanho da textura nos objetos.

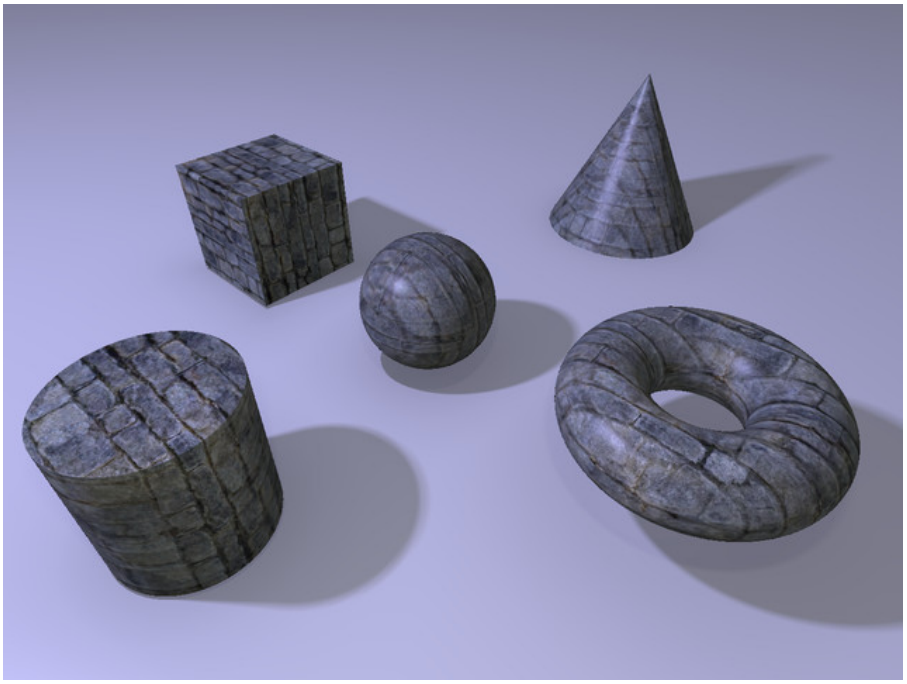


Figura 6: Mapeamento triplanar para diversos objetos distintos (renderizado no TP1).

# 3 Resultados

Esta seção apenas lista algumas das imagens geradas a partir do programa desenvolvido. As figuras foram anexadas nas próximas páginas por motivos de espaço. O número de amostras em cada imagem é a quantidade de caminhos avaliados pelo *path tracer* em cada pixel.

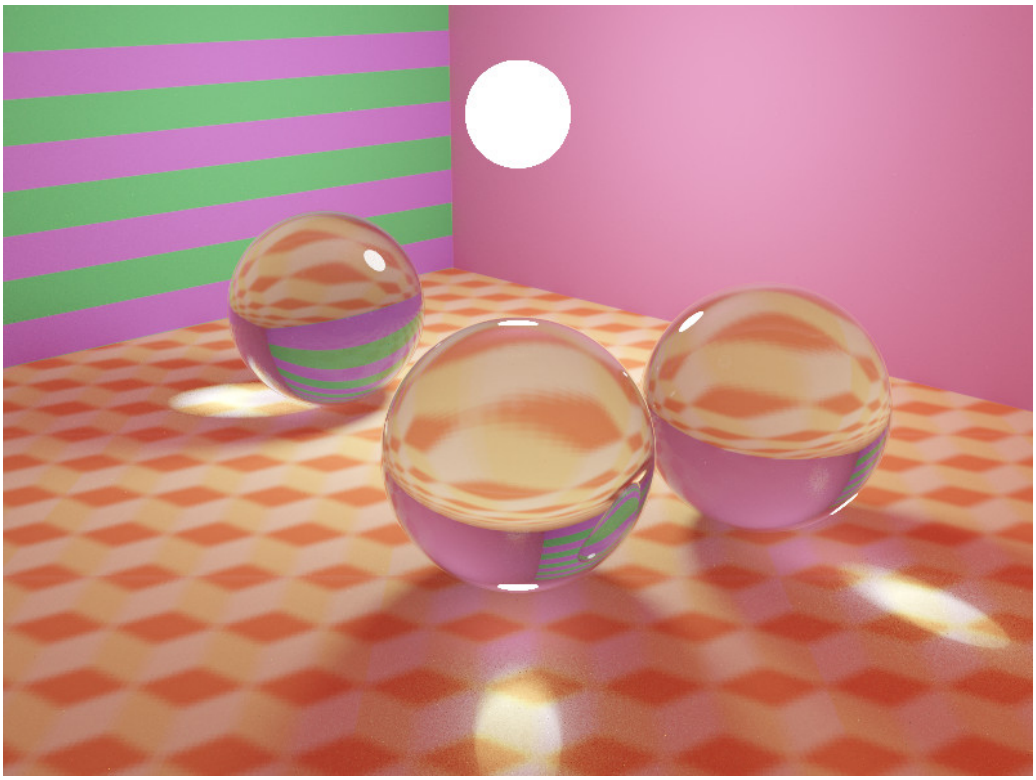


Figura 7: Transmissão da luz, mapeamento de texturas procedural e por arquivo externo, luzes esféricas, sombras suaves e cáusticas. **(800x600 – 3000 amostras – 21.8 minutos)**

(a) 400x300 – 50 amostras – 5.82 segundos

(b) 400x300 – 200 amostras – 23.3 segundos

Figura 8: Com poucos segundos de processamento o renderizador consegue sintetizar uma imagem com qualidade razoável. A maior parte do ruído é concentrado nas cáusticas.

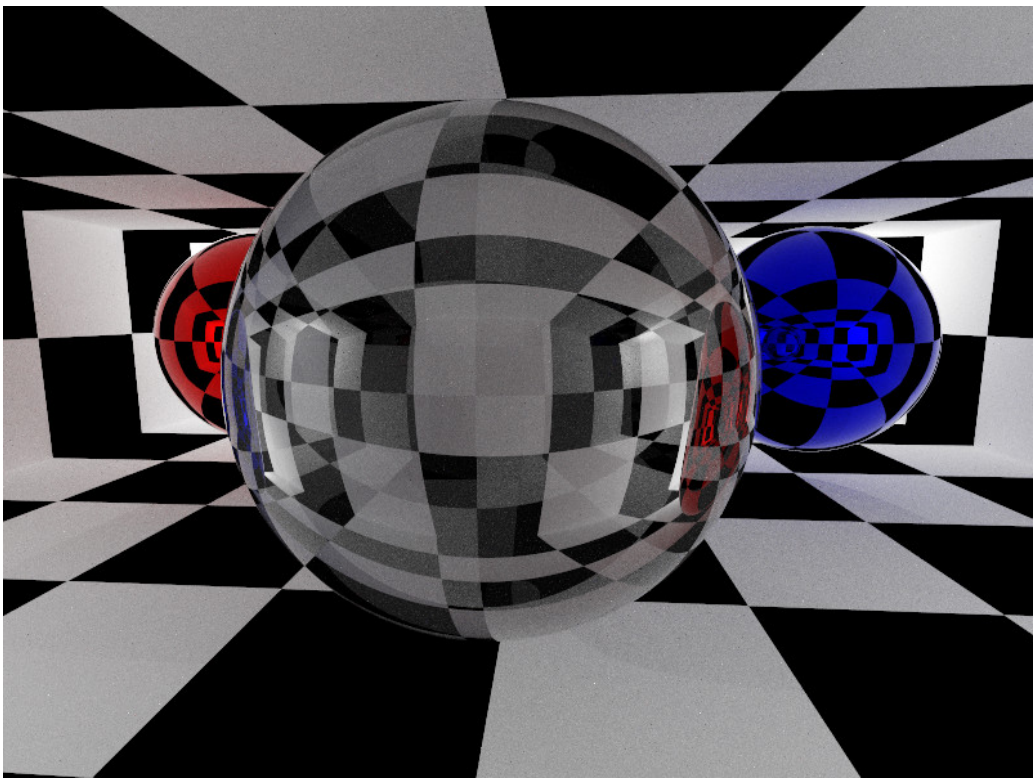


Figura 9: Três esferas, uma de vidro e outras duas refletoras, em uma cena com textura procedural. O efeito de *light bleeding* no teto e no solo devido às esferas coloridas refletoras é bem aparente. **(800x600 – 2200 amostras – 14.84 minutos)**

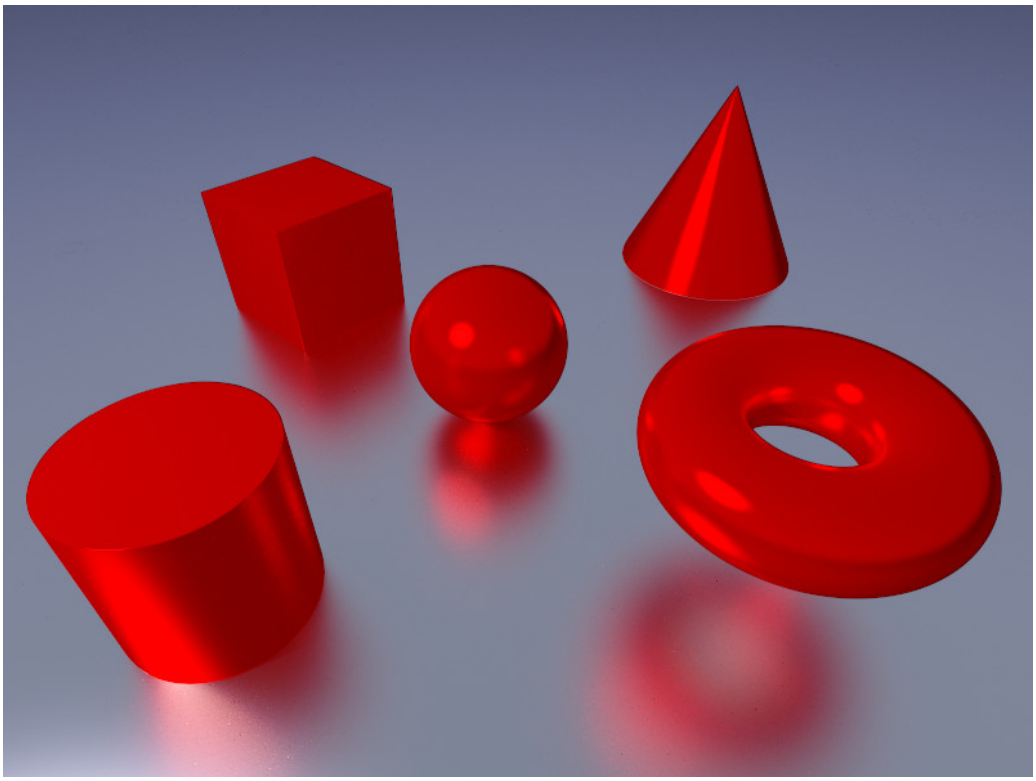


Figura 10: Diversos objetos iluminados com uma luz esférica. Perceba as reflexões do modelo Blinn-Phong e o efeito Fresnel — particularmente visível no chão. **(800x600 – 4150 amostras – 9.56 minutos)**

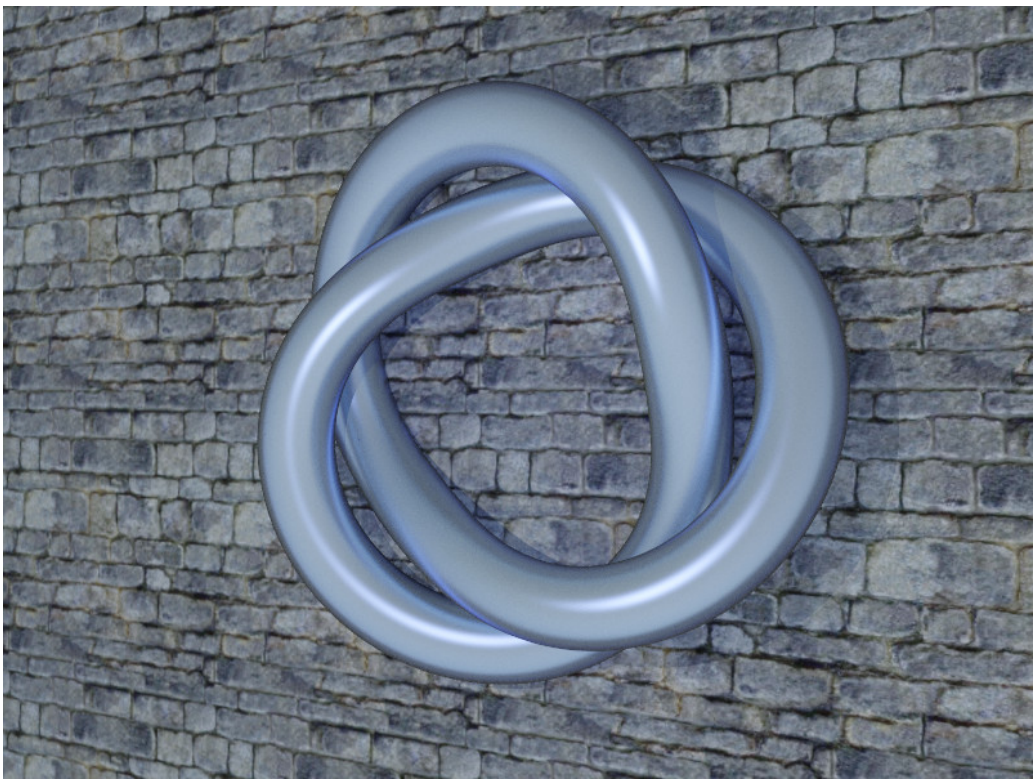


Figura 11: Uma superfície representando um *torus-knot*. O modelo foi renderizado usando uma luz puntual e a BRDF de Blinn-Phong. **(800x600 – 550 amostras – 1.82 minutos)**

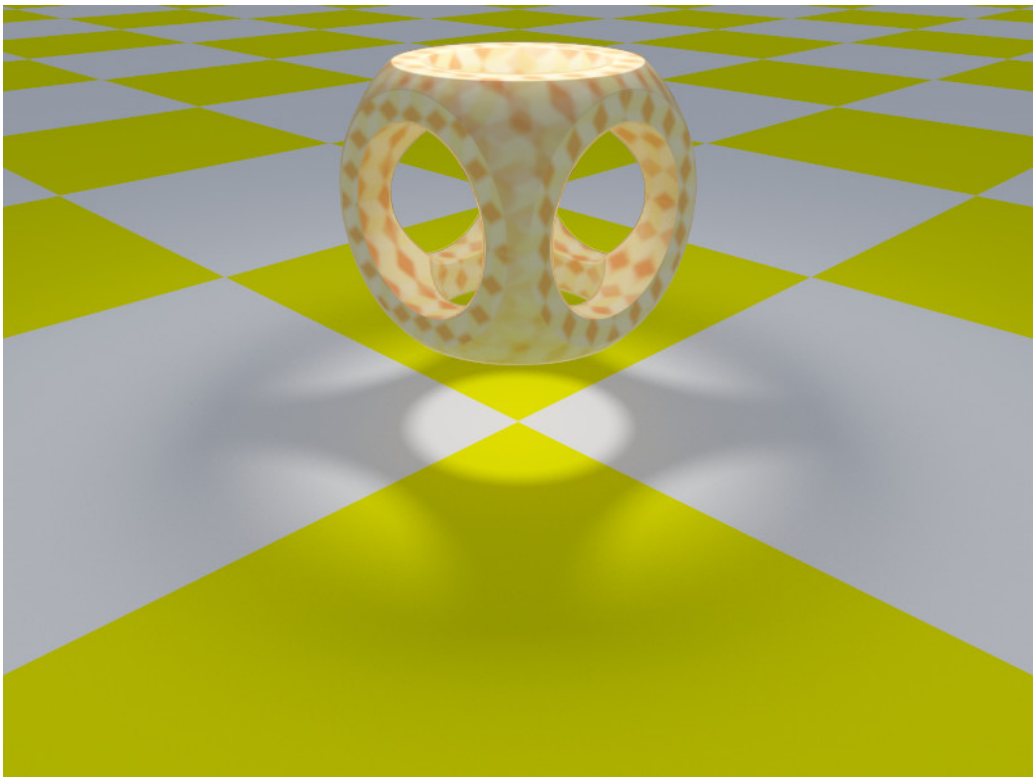


Figura 12: Superfície CSG difusa iluminada por luz esférica. **(800x600 – 1300 amostras – 2.18 minutos)**

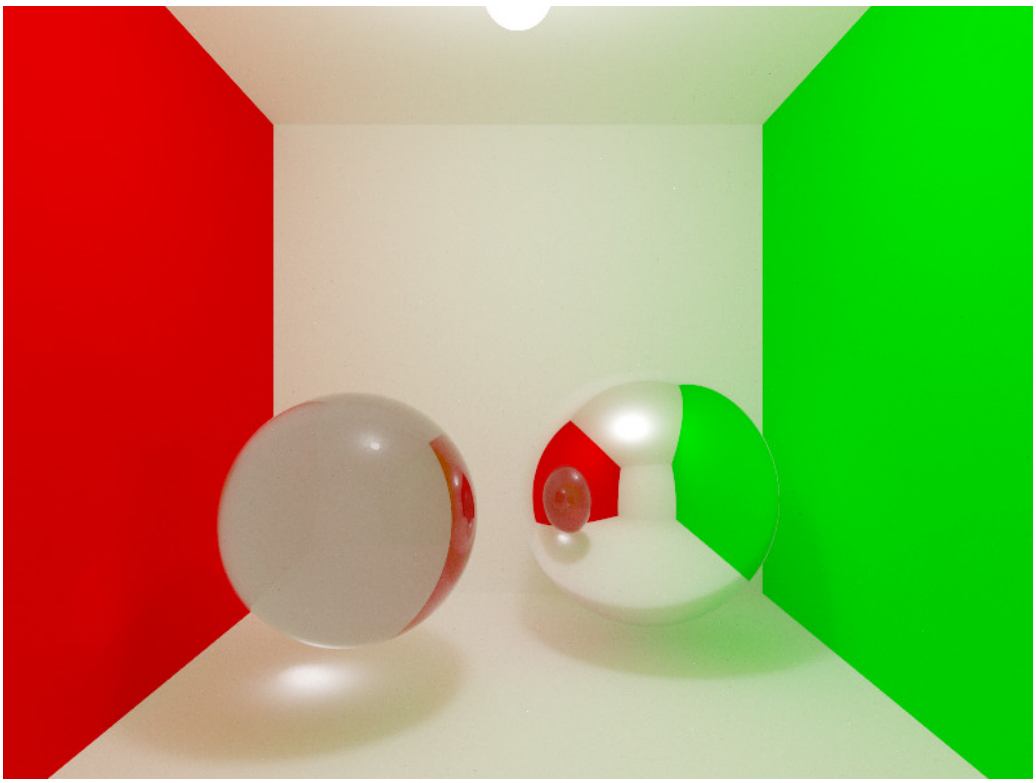


Figura 13: Caixa de Cornell com uma esfera de vidro e outra refletora. A transmissão da parede vermelha e as cáusticas do vidro são inclusive renderizadas na superfície da esfera metálica. **(800x600 – 2550 amostras – 17.82 minutos)**

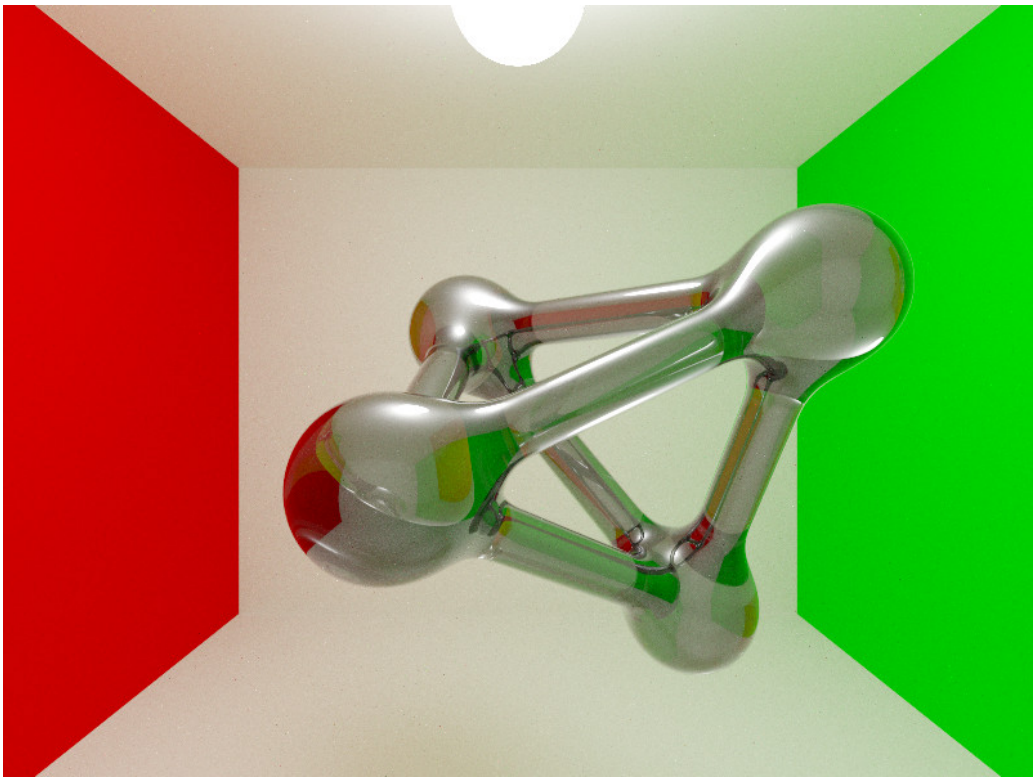


Figura 14: Caixa de Cornell contendo um modelo gerado com operações de geometria sólida construtiva suave. **(800x600 – 2640 amostras – 43.64 minutos)**

# 4 Instruções para Compilar e Executar

Para compilar o programa é necessário ter instalado o CMake (versão mínima 3.0.2) e a biblioteca GLEW (OpenGL Extensions). Acesse a raíz do projeto e utilize os comandos abaixo para compilar. Para executá-lo, basta usar o comando `raytracer [entrada] [largura] [altura] [tempo]`. O parâmetro `[entrada]` pode ser um dos arquivos do diretório `scenes`, enquanto `[largura]` e `[altura]` são as dimensões da imagem. O parâmetro `[tempo]` se não for definido irá ativar a renderização contínua, caso contrário irá renderizar apenas um frame no tempo especificado.

É extremamente recomendado que o programa seja executado em uma máquina relativamente moderna que possua uma placa de vídeo com boa capacidade gráfica.

```
mkdir build
cd build
cmake ..
make
./raytracer scenes/scene1.in
./raytracer scenes/scene2.in 800 600
./raytracer scenes/scene3.in 640 480 10
```

# 5 Conclusão

Neste trabalho foi desenvolvido um *path tracer* que roda iterativamente na GPU e diversas cenas de teste foram renderizadas para verificar o seu funcionamento. O método numérico *sphere-tracing* foi escolhido para determinar as interseções dos raios com os objetos e técnicas de *importance-sampling* foram devidamente empregadas para diminuir o tempo de convergência do estimador Monte Carlo.

De maneira geral, o renderizador desenvolvido consegue sintetizar cenas simples relativamente em pouco tempo (dado um certo nível de ruído) apenas contando com a força bruta da GPU — o que indica que a técnica de *path tracing*, em um futuro relativamente próximo, poderá até mesmo ser aplicada na área de renderização em tempo real. No entanto, efeitos ópticos mais complexos como cáusticas necessitam de um número elevado de amostras para diminuir o ruído a níveis aceitáveis, o que compromete significativamente o tempo de execução.

Quando comparado ao trabalho anterior (TP1), o aumento na qualidade das imagens é evidente. Além disso, a implementação do *path tracing* na GPU mostrou-se mais simples do que o *raytracing* tradicional, já que o primeiro possui recursão de cauda e pode ser facilmente codificado em um único laço de repetição. Em conclusão, *path tracing* é um método robusto de fácil implementação para avaliar a equação de renderização e sintetizar imagens com efeitos de iluminação global. Como opinião pessoal, acho que não temos motivos para continuar a utilizar o *raytracing* clássico além de motivos históricos e/ou didáticos.

# Referências


[1] John C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1994.

[2] Benjamin Keinert, Henry Schäfer, Johann Korndörfer, Urs Ganse, and Marc Stamminger. Enhanced Sphere Tracing. In Andrea Giachetti, editor, *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. The Eurographics Association, 2014.

[3] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.