Renderizador de malhas e primitivas 3D.

Bianca Faria Dutra Fragoso

PROJETO FINAL DE PROGRAMAÇÃO – INF2102

Centro Técnico Científico – CTC

Departamento de Informática

Orientador: Waldemar Celes

Matrícula: 2012384

Rio de Janeiro, junho de 2021

Sumário

Introdução e Motivação

O objetivo desse relatório é cumprir com os requisitos da disciplina Projeto Final de Programação (INF2102), apresentando a motivação para desenvolvimento do programa, a fundamentação teórica por trás dos algoritmos, especificação e projeto do programa, roteiro de testes e um manual de utilização.

Esse trabalho tem como objetivo desenvolver um renderizador de malhas e primitivas 3D com o intuito de testar e aprofundar o estudo de algumas técnicas de renderização. As técnicas exploradas são: Rasterização, *Ray Tracing* e *Path Tracing*.

Todas essas técnicas são muito utilizadas para renderização, no entanto, são utilizadas para propósitos diferentes. A Rasterização é utilizada quando se quer uma renderização em tempo real, mas sem muita qualidade. O *Ray Tracing* é utilizado quando queremos uma renderização com mais qualidade, mais realística. Ela pode ser implementada de forma que fique rápida, mas costuma ser mais lenta do que a Rasterização. Finalmente, o *Path Tracing* é um avanço na técnica de *Ray Tracing* que proporciona mais qualidade para a imagem e possibilita a implementação de efeitos adicionais como emissão e sombras suaves.

O programa é capaz de desenhar formas simples pré-definidas como esferas e planos, mas também suporta quaisquer malhas 3D.

1. Fundamentação teórica

1.1 Rasterização

A técnica de rasterização [1] é dividida em duas etapas. A primeira consiste em projetar malhas de triângulos no plano projetivo usando a projeção perspectiva. A segunda consiste em converter as primitivas em fragmentos para poder preencher todos os pixels da imagem que cobrem cada triângulo daquela malha. Para se renderizar uma esfera, por exemplo, é preciso transformá -la em um conjunto de triângulos e, dependendo de sua projeção no plano projetivo, tais triângulos serão exibidos ou não na tela (z buffer).

A rasterização é um método rápido e barato e permite uma renderização em tempo real de cenas muito complexas. No entanto, sua iluminação é local. Isso significa que apenas a luz que vem diretamente das fontes de luz é levada em consideração. A luz que não vem diretamente de uma fonte de luz, como a luz refletida por um espelho, não é levada em consideração.

Para implementação da rasterização foi utilizada a API OpenGL, juntamente com os *shaders* em GLSL. Cálculos simples de iluminação são utilizados para determinar como fontes de luz iluminam os objetos da cena. Nesse trabalho o modelo de iluminação utilizado foi o modelo de Phong (ver, p.ex., [2]), implementado no *shader* de fragmentos.

Nesse trabalho, a rasterização não é o foco, ela é apenas utilizada como uma forma de renderizar a cena em tempo real e permitir que o usuário a manipule e a coloque na inclinação que ele deseja, utilizando a ferramenta *Arcball (ver, p.ex., [3])*.

Abaixo, na Figura 1, é possível visualizar uma das cenas padrão do programa sendo renderizada com a técnica de rasterização.

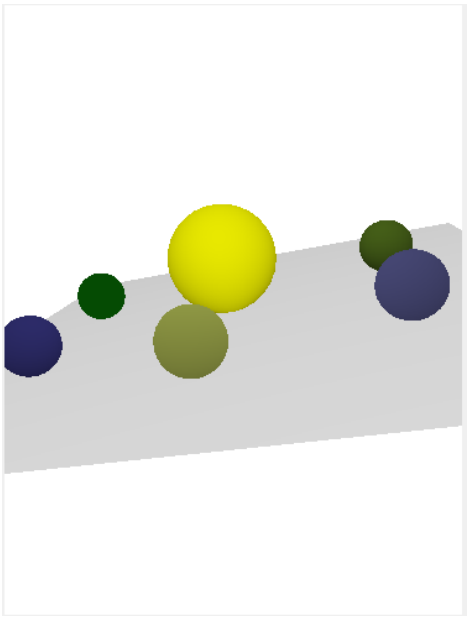

Figura 1: Cena 1 renderizada utilizando rasterização

1.2 Ray Tracing

Em contraste com a rasterização, o traçado de raios é um método de renderização de iluminação global. Isso significa que a luz que é refletida de outras superfícies também é levada em consideração. Isso é essencial para efeitos como reflexão e sombras. Por exemplo, se quisermos modelar uma superfície de água refletindo a cena corretamente, precisamos de um método de renderização de iluminação global.

O algoritmo do traçado de raios consiste em lançar pelo menos um raio para cada pixel da tela e calcular a cor desse pixel. Usando a técnica de Whitted [4], quando um raio encontra um objeto na cena, as informações de cor e iluminação no ponto de impacto daquele raio na superfície do objeto contribuem para a cor do pixel.

Como já foi dito, para cada pixel é preciso lançar pelo menos um raio. No entanto, lançando apenas um raio é possível que se gere uma imagem final com *aliasing*. O efeito *aliasing* consiste no aparecimento de bordas irregulares ou "serrilhadas" nas imagens renderizadas como mostrado na Figura 2a. Esse problema acontece, pois, com apenas um raio por pixel atingimos apenas um local do pixel (no meio) e quando calculamos a cor dos pixels em volta há uma mudança de cor repentina ocasionando esse efeito serrilhado.

Para consertar o problema de *aliasing* é necessário usar mais de um raio por pixel de forma que o raio atinja mais de um local do pixel e a cor seja calculada fazendo uma média de resultados desse traçado de raios para o pixel. Na Figura 2 podemos ver uma comparação entre o *Ray Tracing* com apenas um raio gerando o efeito de *aliasing* e ao lado o com mais de um raio por pixel, consertando esse problema e gerando uma imagem sem serrilhados e com mais qualidade.

Algumas imagens utilizando *Ray Tracing*, geradas por esse programa também são apresentadas na Figura 3.

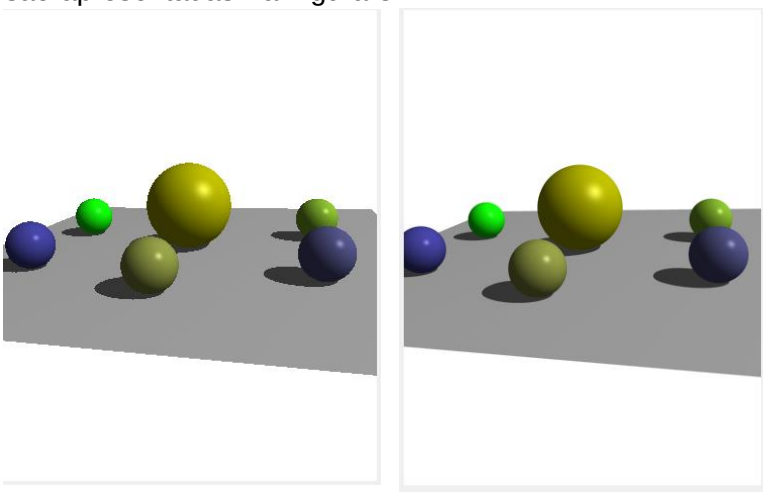

Figura 2: a) Ray Tracing com apenas um raio apresentando o efeito de Aliasing b) Ray Tracing utilizando 10 raios com anti - aliasing implementado

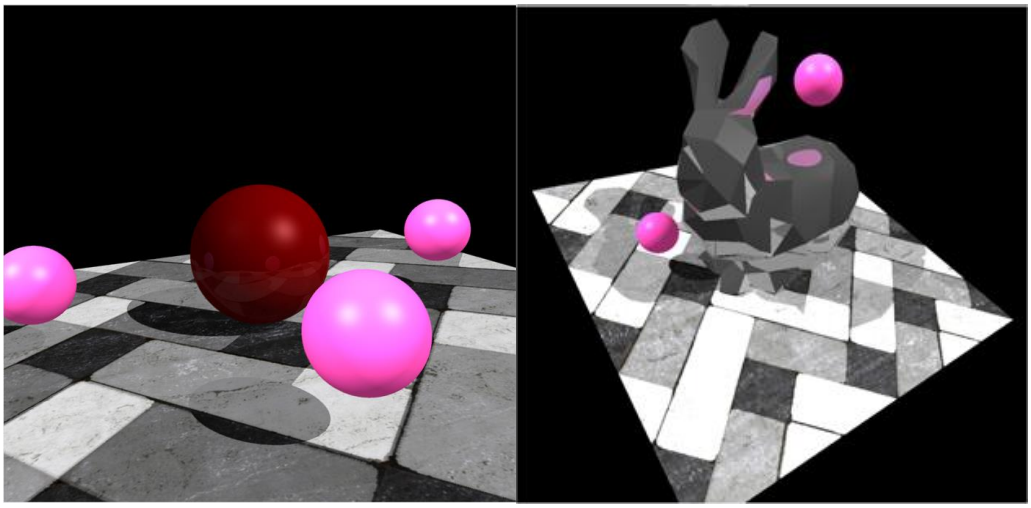

Figura 3: Cenas renderizadas utilizando Ray Tracing. As cenas apresentam os efeitos de sombra e reflexão.

1.3 Path Tracing

Em 1986, Jim Kajiya publicou o artigo “The Rendering Equation” [5], em que introduzia uma equação integral que generaliza uma variedade de algoritmos de renderização conhecidos, chamada de Equação de Renderização. Nesse artigo, além de introduzir a nova equação, Kajiya também propunha um novo algoritmo, variante do *Ray Tracing*, que recebeu o nome de *Path Tracing*.

O *Path Tracing* é uma melhoria no algoritmo de *Ray Tracing*. Ele permite a implementação de outros efeitos como: inter-reflexão difusa, emissão e sombras suaves. Esses efeitos são possíveis graças à formulação da Equação de Renderização [6].

No algoritmo de *Ray Tracing*, um raio é lançado para cada pixel e a cor é calculada. Já no algoritmo de *Path Tracing*, há uma simulação mais realística do movimento da luz no ambiente, ou seja, cada raio lançado ricocheteia pelos objetos e acumula as contribuições de cor e iluminação durante toda a trajetória desse raio. O raio só para de ricochetear quando encontra uma fonte de luz ou quando ricocheteia um número máximo de vezes.

No caso do *Path Tracing*, o raio lançado para o pixel pode não atingir nenhuma fonte de luz e por isso as imagens geradas por esse algoritmo apresentam muitos ruídos. Por isso, é necessário enviar mais de um raio por pixel. Quanto mais raios, menor a quantidade de ruído.

Uma característica importante que diferencia o *Path Tracing* do *Ray Tracing* é que, no primeiro, luzes pontuais são utilizadas, já no segundo, é preciso que as fontes de luz tenham tamanho para que os raios possam intersectá-las. Além disso, como a Equação de Renderização é uma equação integral, não é possível calcular seu valor de forma exata, mas é possível se ter uma estimativa usando o método de Monte Carlo para cálculo de integrais.

Na Figura 4, Figura 5 e Figura 6, é possível ver imagens geradas pelo programa utilizando o método de *Path Tracing*.

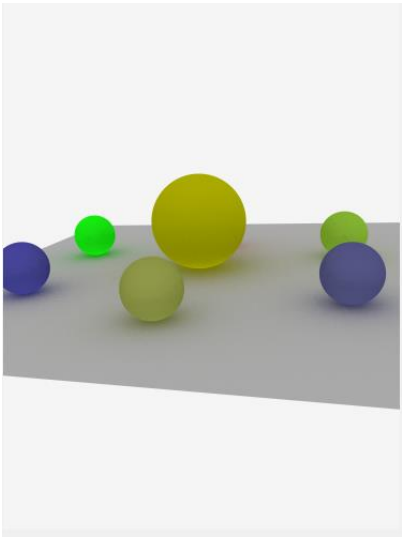

Figura 4: Imagem gerada pelo Path Tracing

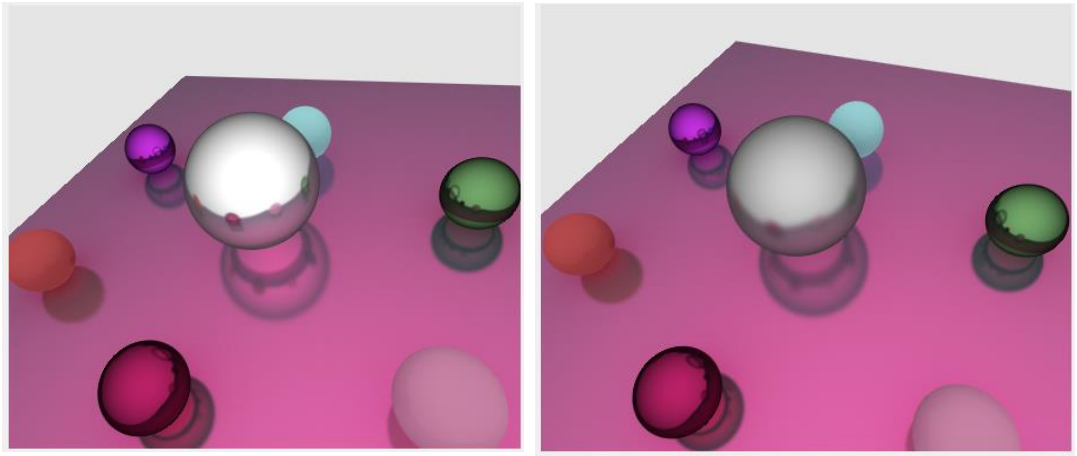

Figura 5: a) Esfera do meio com reflexão nítida b) Esfera do meio com reflexão embaçada (smooth metal)

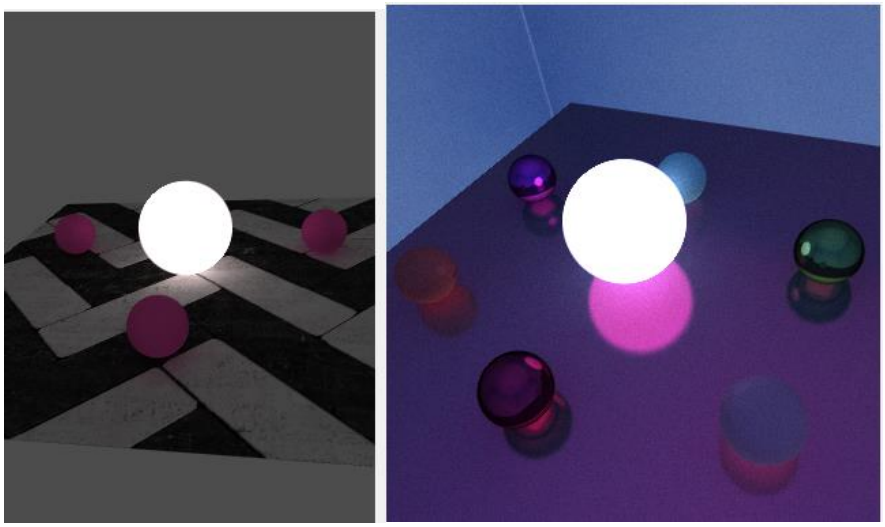

Figura 6: Imagens geradas pelo algoritmo de Path Tracing que mostram o efeito de emissão.

2. Especificação do programa

2.1 Objetivo

A partir da descrição de uma cena composta por primitivas (esferas, planos, etc) e/ ou malhas de triângulos, o programa deve ser capaz de renderizá-la usando uma das 3 técnicas disponíveis de renderização: rasterização, r*ay tracing* ou *path tracing*.

2.2 Requisitos Funcionais

- Ao renderizar com *Ray Tracing* ou *Path Tracing* o programa deve renderizar os itens da cena seguindo as propriedades dos seus respectivos materiais.

- Na janela de renderização em tempo real (rasterização), o usuário deve ser capaz de rotacionar a cena usando o mouse através da ferramenta de *arcball*, de forma que ele consiga posicionar a cena da forma desejada para usar alguma das técnicas de traçado de raios.

- O usuário deve ser capaz de manipular a cena rasterizada usando também as ferramentas de *Zoom In* e *Zoom Out* através da roda do mouse.

- Ao utilizar *Ray Tracing* ou *Path Tracing* o programa deve ser capaz de renderizar sombras corretamente. No caso do *Ray Tracing* ele deve ser capaz de renderizar sombras pesadas e no caso do *Path Tracing* sombras suaves.

- Os algoritmos de traçado de raios devem ser capazes de calcular reflexões caso um dos itens da cena tenha um material reflexivo.

- Quando o método de *Path Tracing* é utilizado, quanto mais raios por pixels o usuário utilizar, menor deverá ser a quantidade de ruídos na imagem final.

- Assim que o usuário apertar o botão de uma das técnicas de renderização e a imagem terminar de ser renderizada, o tempo levado deverá aparecer na tela.

- As técnicas de traçado de raios devem implementar também *anti-aliasing* para gerar imagens com mais qualidade.

2.3 Requisitos Não – Funcionais

- O sistema deve ser implementado com a linguagem de programação C++.
- O sistema deve ser implementado para Windows.
- O Framework utilizado para desenvolvimento do software deverá ser o Qt Creator.
- A API gráfica utilizada para renderização deve ser OpenGL.
- A interação do software com o usuário deve ser feita através da UI da *Main Window* que deve possuir elementos como botões, *spin boxes* e *labels* para facilitar as entradas do usuário.
- Todos os módulos devem ser documentados usando o mesmo padrão de documentação. O .h deve fornecer uma documentação inicial que fale em mais detalhes sobre o módulo em questão. O .cpp deve conter apenas uma documentação rápida sobre o módulo, contendo seu nome e o autor.
- As funções de cada módulo devem ser documentadas dando um breve resumo depois de @brief e comentando sobre cada parâmetro após @param. Para funções que retornam algum valor, deve ser especificado que valor será retornado depois de @return.
- Nos arquivos .cpp todas as funções devem ser separadas por 3 linhas.
- Todas as funções devem ser iniciadas com letra minúscula, apenas construtores e destrutores serão iniciados com letra maiúscula.
- As variáveis privadas de uma classe devem ser iniciadas por "_".

3. Projeto do programa

3.1 Tecnologias

Como já previsto nos requisitos não-funcionais, foi utilizada a linguagem de programação C++ para o desenvolvimento do programa, assim como a API gráfica OpenGL para a parte de rasterização. O *framework* Qt também foi utilizado. O programa foi desenvolvido para Windows.

3.2 Arquitetura

O programa é implementado através de 5 classes principais. Que são responsáveis pelas etapas pelas quais o software passa para renderizar imagens.

1 – A *Main Window* cria a cena.

2 – A *Main Window* cria o *Render*, o *Ray Tracing* e o *Path Tracing* e passa a cena para eles.

3 - O *Render* renderiza a imagem na tela da esquerda presente na *Main Window* usando rasterização.

3 – O usuário interage com a *Main Window* e pode apertar os botões de *Ray Tracing* ou *Path Tracing*.

4 - A imagem é renderizada na tela da direita da *Main Window*.

A Figura 7 mostra o modelo de arquitetura do programa.

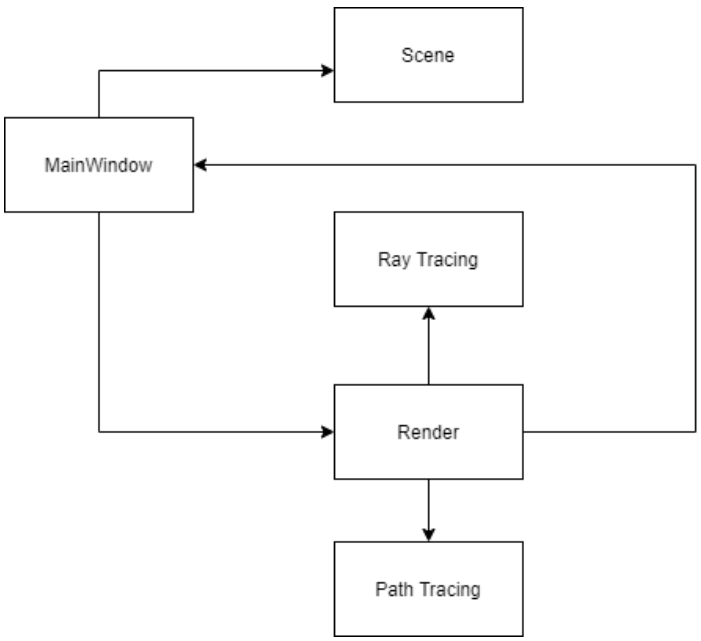

Figura 7: Modelo de Arquitetura

3.3 Modelagem de Classes

A Figura 8 mostra o diagrama de classes. Por motivos de simplificação, o diagrama não contém as funções de *set* e *get* das classes.

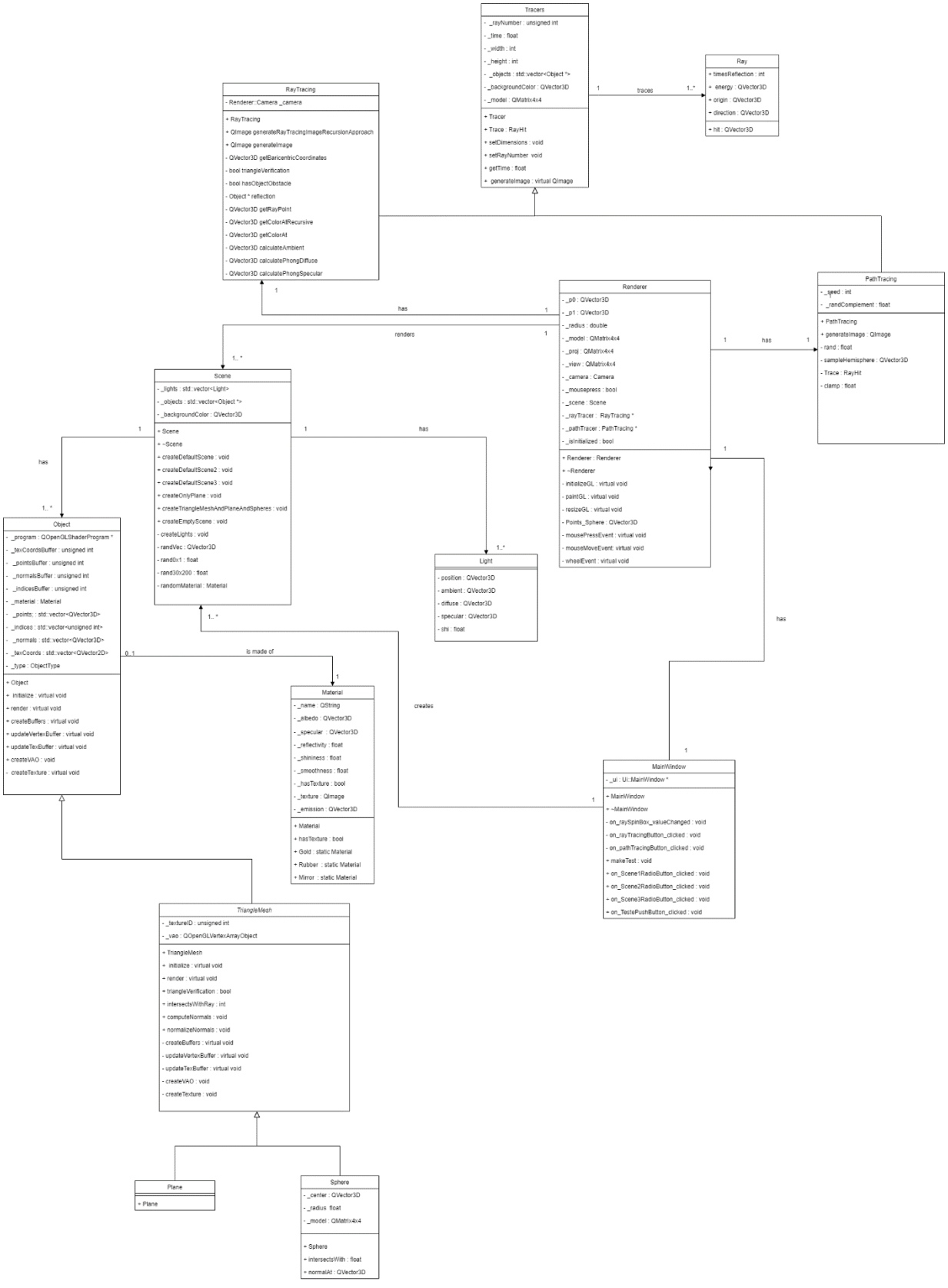

Figura 8:Modelagem de classes

4. Testes

Os testes foram realizados de forma a garantir a qualidade do programa e comprovar alguns efeitos que estão presentes nos requisitos funcionais. O resultado de alguns testes é demonstrado por imagens, visto que o programa tem o intuito de renderizar cenas e seus resultados são imagens.

4.1 Alterando quantidade de raios

Como previsto nos requisitos funcionais, quanto o maior número de raios, menor deve ser a quantidade de ruídos na imagem final gerada pelo algoritmo de *Path Tracing.* Logo, testes foram feitos com o intuito de comprovar que o programa cumpre esse requisito.

Abaixo na Figura 9, Figura 10 e Figura 11, são apresentados alguns resultados de testes feitos, aumentando a quantidade de raios para uma mesma cena. Em todos eles, quanto maior a quantidade de raios, menor a quantidade de ruídos.

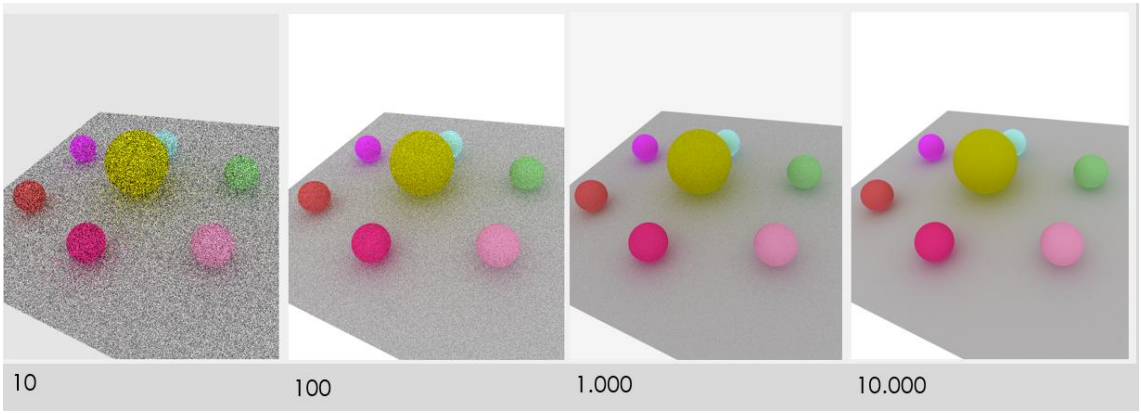

Figura 9: imagens geradas aumentando a quantidade de raios por pixel. Da esquerda para a direita o número de raios aumenta: 10 raios, 100 raios, 1000 raios e 10000 raios.

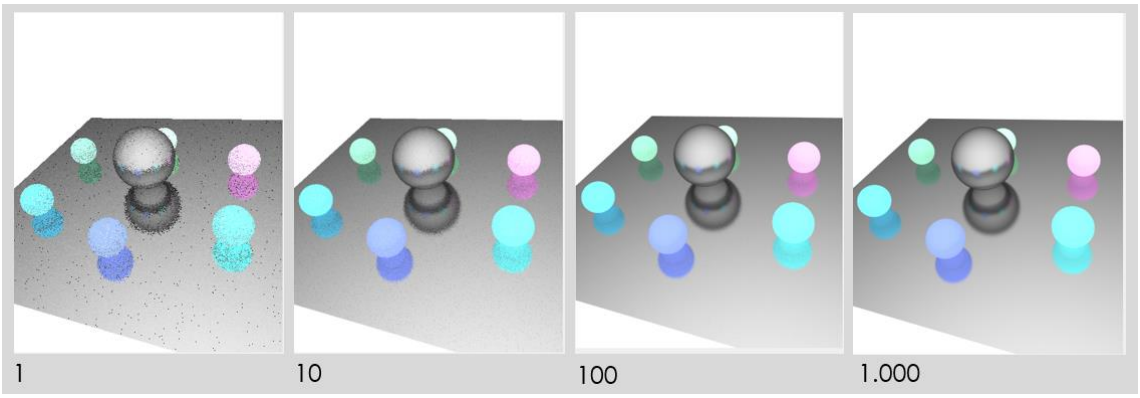

Figura 10: imagens geradas aumentando a quantidade de raios por pixel. Da esquerda para a direita o número de raios aumenta: 1 raio, 10 raios, 100 raios, 1000 raios.

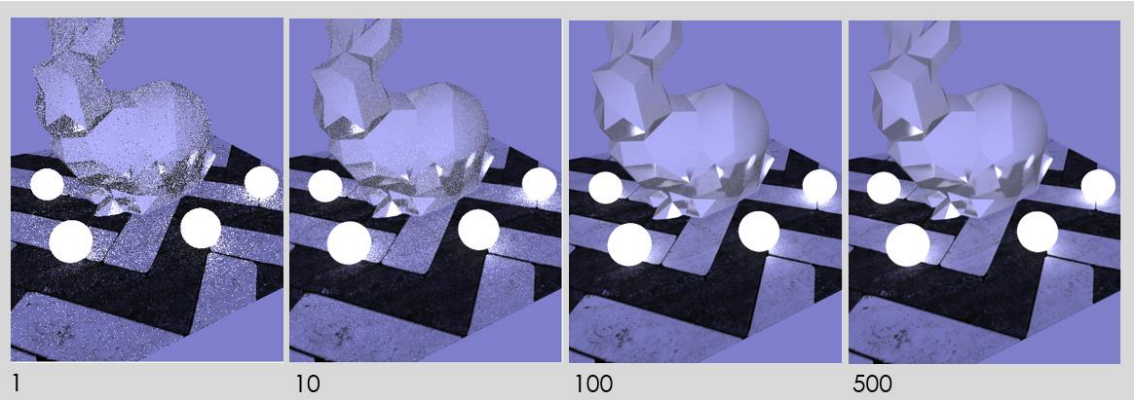

Figura 11: imagens geradas aumentando a quantidade de raios por pixel. Da esquerda para a direita o número de raios aumenta: 1 raio, 10 raios, 100 raios, 500 raios.

4.2 Testes de funcionalidade do traçado de raios

Essa segunda bateria de testes foi feita para comprovar que o algoritmo de traçado de raios funciona. Os testes foram feitos de forma automática criando uma série de cenas e rodando o algoritmo de *Ray Tracing* para cada uma delas.

Lançando um raio para cada pixel e contabilizando qual tipo de objeto esse raio atingiu primeiro, foi possível comprovar que o traçado de raios estava sendo feito de forma correta e otimizada. Abaixo é apresentado o roteiro de testes.

4.2.1 Cena Vazia só com Background

Para essa cena o arquivo de log deve ter apenas a contagem de raios atingindo o background diferente de zero. Para esferas, planos e malhas de triângulos a quantidade de raios que atingiram esses objetos deve ser 0.

4.2.2 Cena mostrando Background, Esferas e um Plano

Para essa cena o arquivo de log deve ter a contagem de raios diferente de 0 para todos os itens menos para a malha de triângulos que deve ser zero, já que a cena não possui malhas de triângulos.

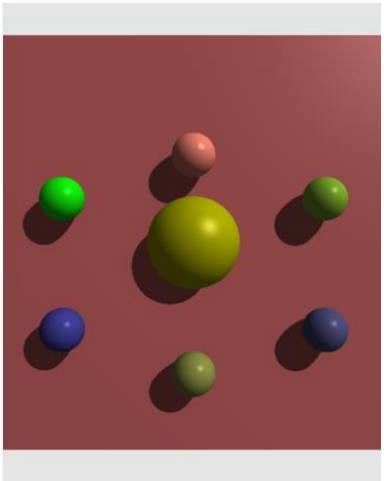

Figura 12: Cena com background cinza, esferas e plano de apoio usada como teste.

4.2.3 Cena sem Background apenas com um Plano e Esferas

Para essa cena o arquivo de log deve ter a contagem de raios diferente de 0 para planos e esferas e contagem 0 para background e malhas de triângulo.

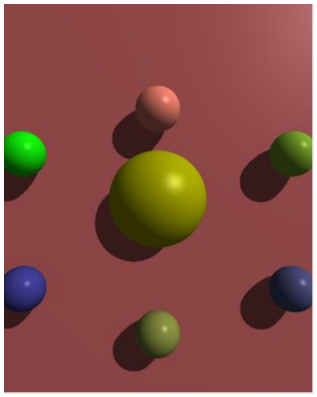

Figura 13: Cena mostrando apenas o plano de apoio e as esferas, sem background aparecer. Foi utilizada como teste.

4.2.4 Cena apenas com Plano.

Para essa cena o arquivo de log deve mostrar todos os raios atingindo apenas o plano. Todos os outros tipos de objetos devem ter a contagem zerada pois não existem na cena.

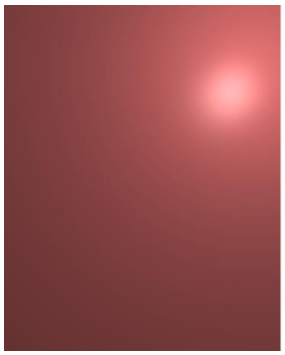

Figura 14: Cena apenas com plano de apoio usada como teste.

4.2.5 Cena com Background, Malha de triângulos, Esferas e Plano.

Para essa cena o arquivo de log deve ter a contagem de raios diferente de 0 para planos, background e malhas de triângulos e contagem 0 para esferas.

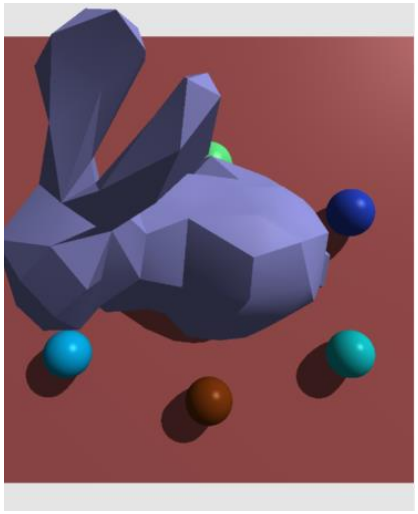

Figura 15: Cena com background cinza, malha de coelho low polly, esferas e um plano de apoio, vista de cima.

```
########################################## TESTE ##########################################

###################################### Cena Vazia so com Background ######################################
Total of Rays: 172420
Background: 172420
Sphere: 0
Plane: 0
Triangle Mesh: 0
##########################################################################################################

###################################### Cena mostrando Background, Esferas e um Plano ###############
Total of Rays: 172420
Background: 23310
Sphere: 15351
Plane: 133759
Triangle Mesh: 0
##########################################################################################################

###################################### Cena sem Background apenas com um Plano e Esferas ###########
Total of Rays: 172420
Background: 0
Sphere: 24938
Plane: 147482
Triangle Mesh: 0
##########################################################################################################

###################################### Cena apenas com Plano #############################################
Total of Rays: 172420
Background: 0
Sphere: 0
Plane: 172420
Triangle Mesh: 0
##########################################################################################################

###################################### Cena com Background, Malha de triângulos, Esferas e Plano ####
Total of Rays: 172420
Background: 21849
Sphere: 6267
Plane: 91816
Triangle Mesh: 52488
##########################################################################################################
```

Figura 16: Arquivo .log de testes.

5. Manual de Uso

O programa possui algumas alternativas de cenas que podem ser alternadas mudando a seleção dos *Radio Buttons* presentes na parte de superior da interface. O programa possui duas telas. Na tela da esquerda aparece a cena rasterizada. O usuário pode interagir com a cena utilizando o mouse e mexendo com a ferramenta *Arcball* para conseguir rotacioná-la. É possível dar zoom com a roda do mouse.

Quando o usuário chegar na posição desejada para a cena, ele pode apertar os botões de *Ray Tracing* ou *Path Tracing* para renderizá – la com uma dessas técnicas na tela da direita. O tempo levado para renderizar a cena aparecerá no canto superior direito da tela.

Além disso, o usuário também pode escolher a quantidade de raios por pixel que será utilizada para renderizar com uma dessas duas técnicas. Ele pode alterar essa quantidade no spin box que se encontra no canto superior esquerdo.

O botão Make Test executa a bateria de testes prevista na seção 4.2. Abaixo na Figura 17 é possível visualizar a interface do programa.

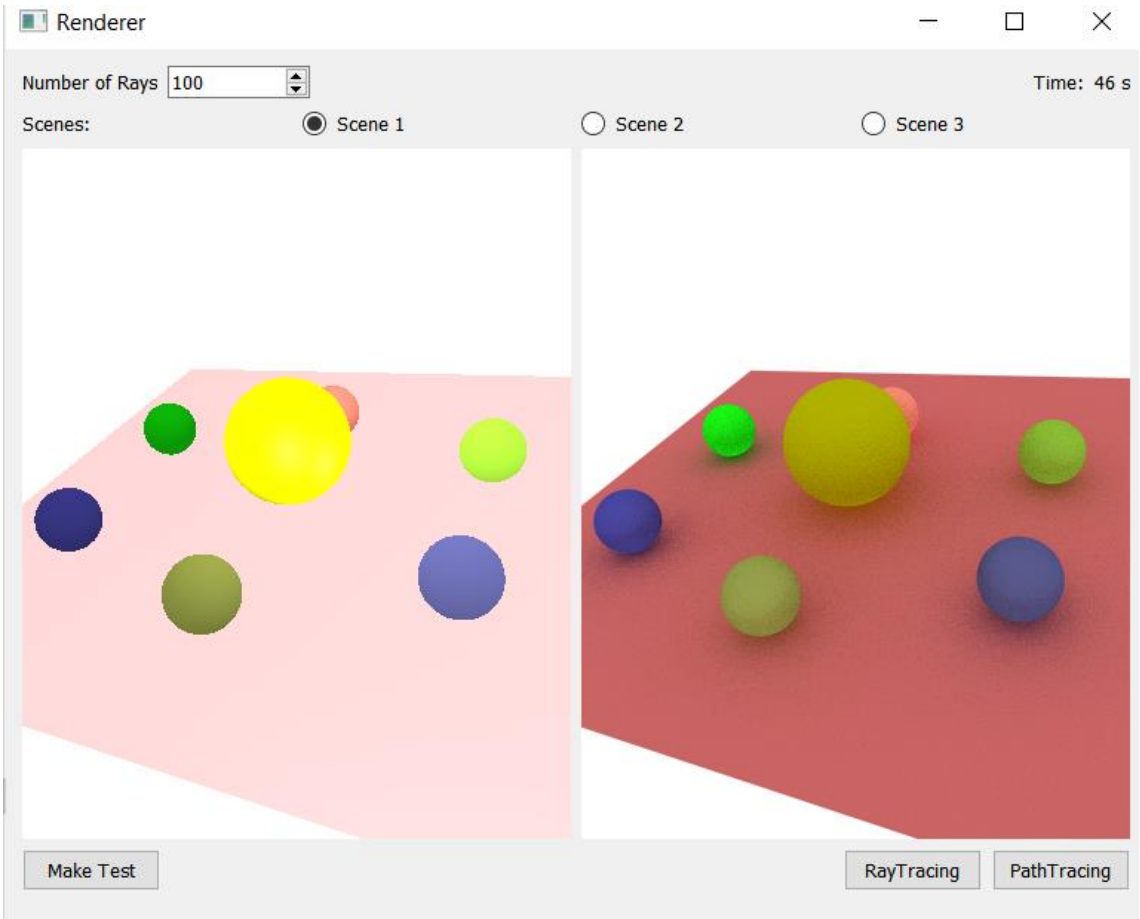

Figura 17: Interface do programa.

Referências Bibliográficas

[1] Neider, Jackie, Tom Davis, and Mason Woo. *OpenGL programming guide*. Vol. 478. Reading, MA: Addison-Wesley, 1993.

[2] Phong, Bui Tuong. "Illumination for computer generated pictures." Communications of the ACM 18.6 (1975): 311-317.

[3] Shoemake, Ken. "ARCBALL: A user interface for specifying three-dimensional orientation using a mouse." *Graphics interface*. Vol. 92. 1992.

[4] Whitted, Turner. "An improved illumination model for shaded display." *ACM Siggraph 2005 Courses*. 2005. 4-es.

[5] Kajiya, James T. "The rendering equation." *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986.

[6] Kuri, David. Disponível em: < http://three-eyed-games.com/2018/05/12/gpu-path-tracing-in-unity-part-2/> Acesso em: 21 de jun. de 2021.