PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Renderização de modelos CAD com traçado de raios

Eduardo Brazão Maksoud

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC

DEPARTAMENTO DE INFORMÁTICA

Graduação em Ciência da Computação

Rio de Janeiro, Novembro, 2022

Eduardo Brazão Maksoud

Renderização de modelos CAD com traçado de raios

Relatório de Projeto Final, apresentado ao curso de Ciência da Computação como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Waldemar Celes Filho

Rio de Janeiro

Novembro de 2022

Agradecimentos

Primeiramente, gostaria de agradecer à minha família por todo apoio durante a minha formação. Especialmente aos meus pais, que sempre me deram condições de seguir em frente e me ajudaram nos momentos mais difíceis.

Faço um agradecimento especial a Waldemar Celes e Paulo Ivson por terem me dado esta oportunidade e por toda a assistência provida durante o desenvolvimento deste projeto.

Agradeço também à equipe do Tecgraf, por todo auxílio na elaboração deste trabalho. Em particular, agradeço ao Felipe Moura de Carvalho pelas inúmeras dúvidas esclarecidas e suporte no decorrer do projeto.

Por fim, agradeço aos meus amigos por todos os ótimos momentos que passamos no período da graduação e pela enorme troca de experiências que tivemos.

Resumo

Maksoud, Eduardo, Celes Filho, Waldemar. Renderização de modelos CAD com traçado de raios. Rio de Janeiro, 2022. 34 páginas. Relatório Final de Projeto Final – Centro Técnico Científico, Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

O traçado de raios é uma técnica de renderização que simula o comportamento físico da luz e, por este motivo, é capaz de gerar imagens bem realistas. Este trabalho busca investigar os benefícios de visualização que o traçado de raios pode trazer para a renderização de modelos CAD complexos e comparar os resultados com a técnica de rasterização, comumente utilizada para a renderização desses modelos, devido à sua eficiência.

Palavras-chave
Traçado de raios, renderização, computação gráfica, rasterização, modelos CAD

Abstract

Maksoud, Eduardo, Celes Filho, Waldemar. Rendering of CAD models with Ray-Tracing. Rio de Janeiro, 2022. 34 pages. Centro Técnico Científico, Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Ray-Tracing is a rendering technique that simulates the physical behavior of light and, therefore, is able to generate very realistic images. This work aims to investigate the visualization benefits that Ray-Tracing can bring to the rendering of complex CAD models and to compare the results with the rasterization technique, commonly used for the rendering of these models due to its efficiency.

Keywords
Ray Tracing, rendering, computer graphics, rasterization, CAD models

Sumário

Lista de Figuras

1. Introdução

1.1. Motivação

O método de traçado de raios não é novidade para a comunidade de computação gráfica. Na verdade, o primeiro estágio do algoritmo ou *ray casting* foi descrito por Arthur Appel em 1968 **[1]**. Diferentemente da técnica de rasterização, o traçado de raios simula de forma fisicamente correta a iluminação de uma cena. Para isso, é considerado o trajeto percorrido pelos raios de luz no mundo real, limitando-se, porém, aos raios de luz que atingem os olhos do observador.

Além do custo computacional elevado do algoritmo, é importante lembrar que seu desempenho também depende da complexidade da cena. Ao contrário de modelos 3D usados para jogos ou animações, os modelos CAD (*Computer-Aided Design*) são utilizados em projetos de engenharia, como modelos digitais de plataformas e refinarias de petróleo, sendo mais complexos e difíceis de renderizar.

A motivação deste trabalho é investigar os benefícios de visualização na renderização de modelos CAD com a técnica de traçado de raios em tempo real. Em seguida, deseja-se comparar os resultados com os obtidos através da técnica de rasterização no sistema Environ **[10]**. Além disso, pretende-se aplicar uma textura para obter um efeito de “corrosão” em alguns dos componentes do modelo escolhido.

1.2. Definição do problema

Por possuir um elevado custo computacional, a execução do algoritmo de traçado de raios em tempo real é um desafio. Em um filme ou animação, as imagens podem ser pré-processadas, mas em um software interativo isso não é possível.

Por questões de eficiência, a rasterização de triângulos se tornou uma maneira comum de renderizar modelos CAD, mas a adição de efeitos como sombras, reflexões ou refrações possui uma implementação complexa e tem como resultado imagens menos realistas se comparadas com outras geradas com o método de traçado de raios.

O problema investigado por este trabalho é a renderização de modelos de alta complexidade, em tempo real, utilizando a técnica de traçado de raios para obter visualizações mais realistas.

1.3. Relevância do problema

Até pouco tempo, não se pensava em utilizar o traçado de raios para aplicações em tempo real devido ao tempo excessivo para renderizar uma cena.

Mesmo que este ainda seja um processo custoso, os avanços recentes na tecnologia de GPUs mudaram este cenário e, como a tendência do preço das GPUs

mais modernas é diminuir ao longo do tempo, esses dispositivos podem se tornar mais acessíveis.

1.4. Ambiente de desenvolvimento

O trabalho foi realizado em um computador *Windows*, com a placa de vídeo NVIDIA RTX 3080ti e CPU AMD Ryzen 9 5950X. O código apresentado no tutorial de Ingo Wald **[8]** serviu de base para renderizar os modelos com a API de traçado de raios Nvidia OptiX 7. O código do *renderer* é escrito em C++, e os programas que rodam na GPU são escritos em CUDA. As coordenadas de textura de algumas malhas dos modelos foram geradas com o auxílio da ferramenta Blender.

2. Tipo dos dados

2.1. Estrutura do arquivo obj

O arquivo **obj** contém as informações sobre a malha de polígonos e mapeamento das texturas. Os materiais usados são declarados em um arquivo **mtl**, que normalmente possui o mesmo nome do **obj**.

O **obj** é composto por um conjunto de objetos, que por sua vez possuem vértices, normais, coordenadas de textura e faces. Abaixo está o exemplo de um triângulo utilizando um material metálico.

```
usemtl metal

# Declaração do objeto
o triângulo 0

# Lista de vértices
v 257.327271 2.543350 36.819508
v 257.327271 2.679351 37.636509
v 257.327271 2.679351 36.819508

# Normal
vn 1.000000 -0.000000 0.000000

# Coordenadas de textura
vt 0.708558 0.999786
vt 0.590695 0.999786
vt 0.590695 0.000205

# Face
f 1/1/1 2/2/1 3/3/1
```

A primeira linha indica que o nome do material utilizado para este objeto é metal. Em seguida está a declaração do objeto triângulo, que por ser o único do arquivo recebe o índice 0.

Os vértices **(v)**, normais **(vn)** e coordenadas de textura **(vt)** devem ser declarados antes das faces **(f)**, pois para definir uma face é preciso indicar os índices de cada elemento seguindo o padrão **v/vt/vn**. A única face deste exemplo é composta

pelos vértices de índices 1, 2 e 3, cada um com sua coordenada de textura e todos utilizando a única normal declarada no arquivo, de índice 1.

2.2. Materiais

O arquivo **mtl** serve como uma biblioteca de materiais para um arquivo **obj**. A composição de um material pode ser observada abaixo.

Por seguir o modelo de iluminação de Phong, na área de cor e iluminação os termos **Ka**, **Kd** e **Ks** se referem a componente ambiente, difusa e especular do material, todas em RGB. O termo **map_Kd** apenas informa qual imagem será utilizada para a textura do material. Abaixo está o arquivo com a declaração do material metálico utilizado no cubo.

```
# Nome do material
newmtl metal

# Cor e iluminação
Ka 1.000000 1.000000 1.000000
Kd 1.000000 1.000000 1.000000
Ks 0.010000 0.010000 0.010000

# Texturas
map_Kd textures\metal.png
```

3. Modelo

Por lidar com dados paramétricos, o sistema Environ é capaz de renderizar o modelo da plataforma de extração de petróleo P-74 da Petrobras completamente. Como o código do *renderer* para o traçado de raios só lida com malhas de triângulos, não foi possível carregar todos os componentes dado o tamanho dos arquivos.

Assim, para a renderização com traçado de raios, foi escolhido apenas o componente **Flare System**.

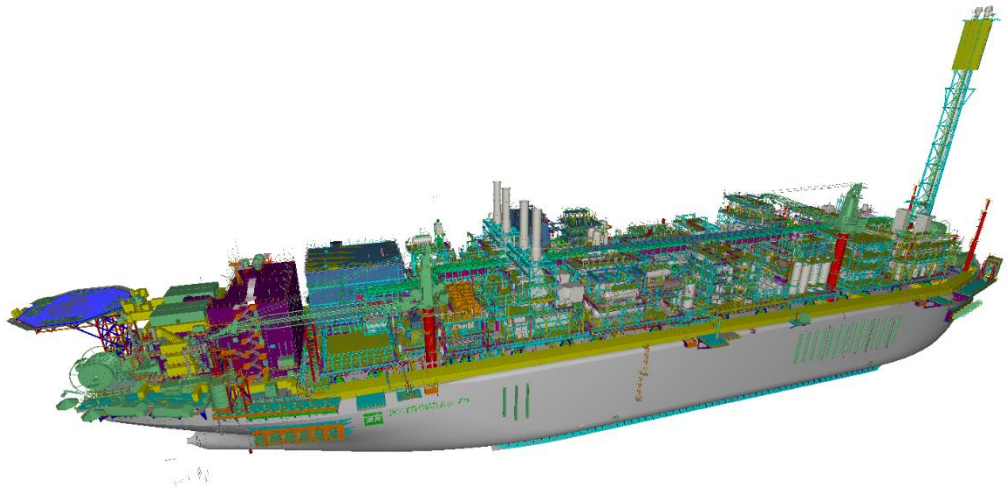

Figura 1 - Modelo completo da P-74 renderizado no Environ.

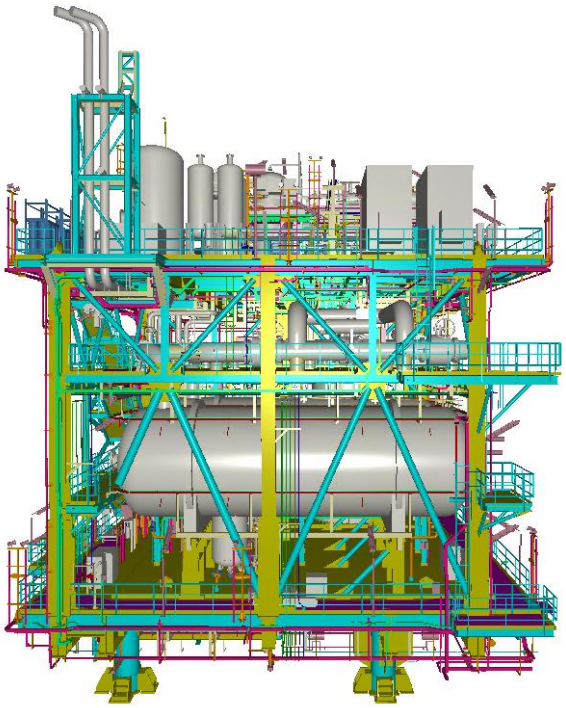

Figura 2 - Modelo do Flare System da P-74 renderizado no Environ.

Abaixo está a quantidade de cada tipo de malha presente no modelo **Flare System**. Ainda que seja apenas um pedaço do modelo original, percebe-se que ele já possui uma boa complexidade.

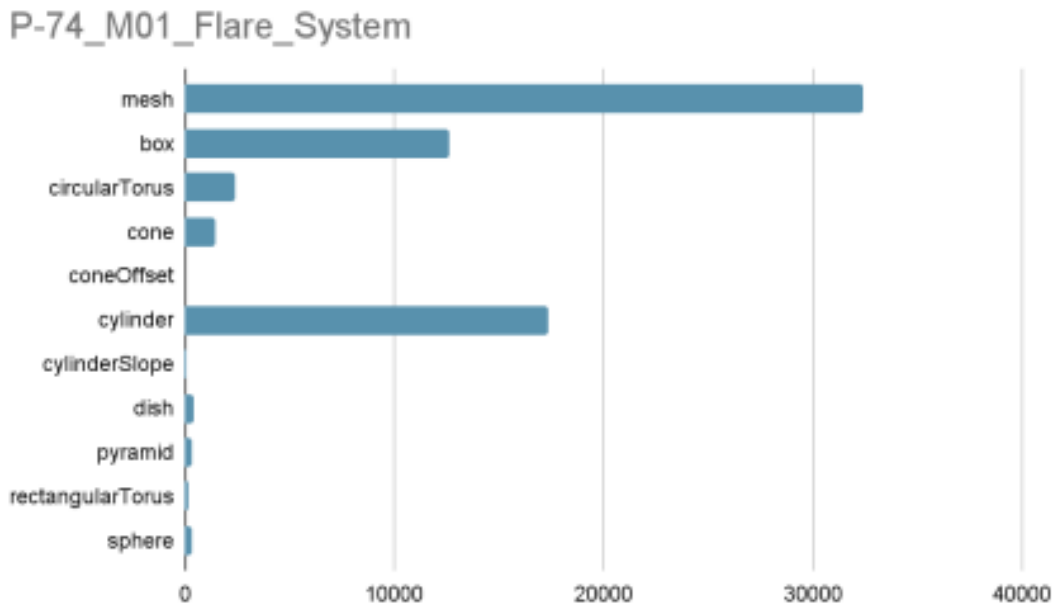

Figura 3 - Gráfico em barra informando a quantidade de cada tipo de malha do componente Flare System da P-74.

4. Iluminação de Phong

4.1. Resumo do modelo de iluminação

A iluminação de uma superfície depende de inúmeros fatores no mundo real e o modelo de iluminação de Phong é uma maneira simples de representá-los, sendo composto basicamente por três componentes.

A componente ambiente é uma iluminação uniforme em todos os pontos da superfície para simular o comportamento da luz de ricochetear em várias direções, se espalhando pela cena.

Normalmente, no mundo real, objetos que não recebem iluminação direta ainda são iluminados pela luz refletida de outras superfícies. Alguns algoritmos mais complexos podem calcular esta iluminação indireta, porém neste trabalho, para simplificar a implementação, apenas utiliza-se uma constante.

A componente difusa considera a incidência de luz sobre a superfície, já que o ângulo em que o raio de luz toca a superfície é o que define o brilho de um determinado ponto.

A componente especular serve para representar os pontos brilhantes de uma superfície, quando os raios de luz refletidos da superfície se encontram na mesma direção que o observador.

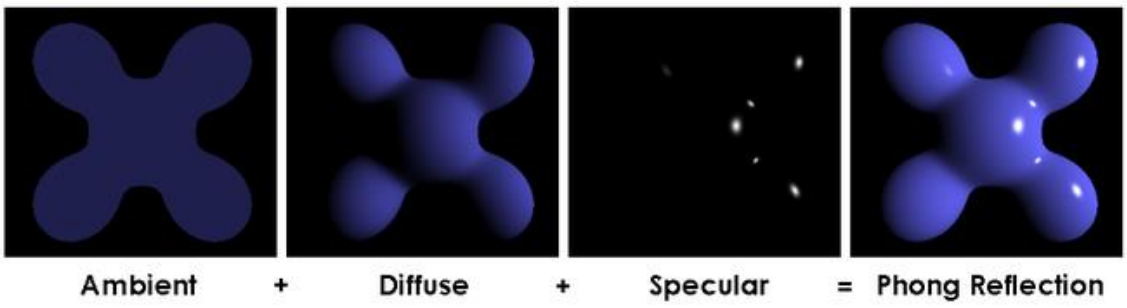

Figura 4 - Ilustração das componentes do modelo de Phong. **[12]**

4.2. Equação para a iluminação de um ponto

É possível calcular a iluminação de cada ponto da superfície com a fórmula abaixo. O símbolo $\widehat{\ }$ indica que os vetores foram normalizados.

$$k_a i_a + \sum_{m\ \epsilon\ luzes} \quad (k_d(\widehat{L_m} \cdot \widehat{N}) i_{m,d} \quad + k_s(\widehat{R_m} \cdot \widehat{V})^{\alpha} i_{m,s} \quad)$$

- k_a, k_d e k_s são as componentes ambiente, difusa e especular respectivamente.
- i_a, i_d e i_s são as intensidades ambiente, difusa e especular respectivamente.
- α é o coeficiente de brilho do material do objeto atingido.
- $\widehat{L_m}$ é o vetor do ponto atingido na superfície até a fonte luminosa.
- $\widehat{N}$ é a normal do ponto atingido.
- $\widehat{R_m}$ é o vetor de luz refletido em torno da normal do ponto atingido.
- $\widehat{V}$ é o vetor do ponto atingido até o observador.

Para calcular a iluminação ambiente, multiplica-se a componente ambiente por uma intensidade i_a . Depois, é preciso calcular o somatório da contribuição de cada fonte luminosa da cena.

O produto interno dos vetores $\widehat{L_m}$ e $\widehat{N}$ é o cosseno do ângulo entre o raio de luz que atinge o ponto e a sua normal. Ao multiplicar este cosseno pela componente difusa e por uma intensidade $i_{m,d}$ se obtém a iluminação difusa de uma luz da cena.

Para a iluminação especular, é preciso calcular o produto interno dos vetores $\widehat{R_v}$ e $\widehat{V}$, obtendo o cosseno do ângulo entre eles. Pela fórmula, nota-se que quanto menor for o ângulo entre eles, maior será o impacto da iluminação especular. Quanto maior for o coeficiente de brilho, a luz será refletida mais precisamente, sem se espalhar pela superfície do objeto, gerando um realce menor.

5. O algoritmo de traçado de raios

5.1. Resumo do funcionamento do algoritmo

A ideia simplificada do algoritmo é que para cada pixel da tela será lançado um raio que pode intersectar objetos dentro de uma cena. Se nenhum objeto for atingido, então o pixel receberá a cor do fundo. Assumindo que o raio tenha atingido um objeto e que este é o mais próximo do observador, determina-se uma interseção, de onde deve-se traçar um novo raio — comumente chamado de *shadow ray* — para cada fonte de luz da cena, determinando se o ponto de interseção avaliado está na sombra ou se está recebendo luz.

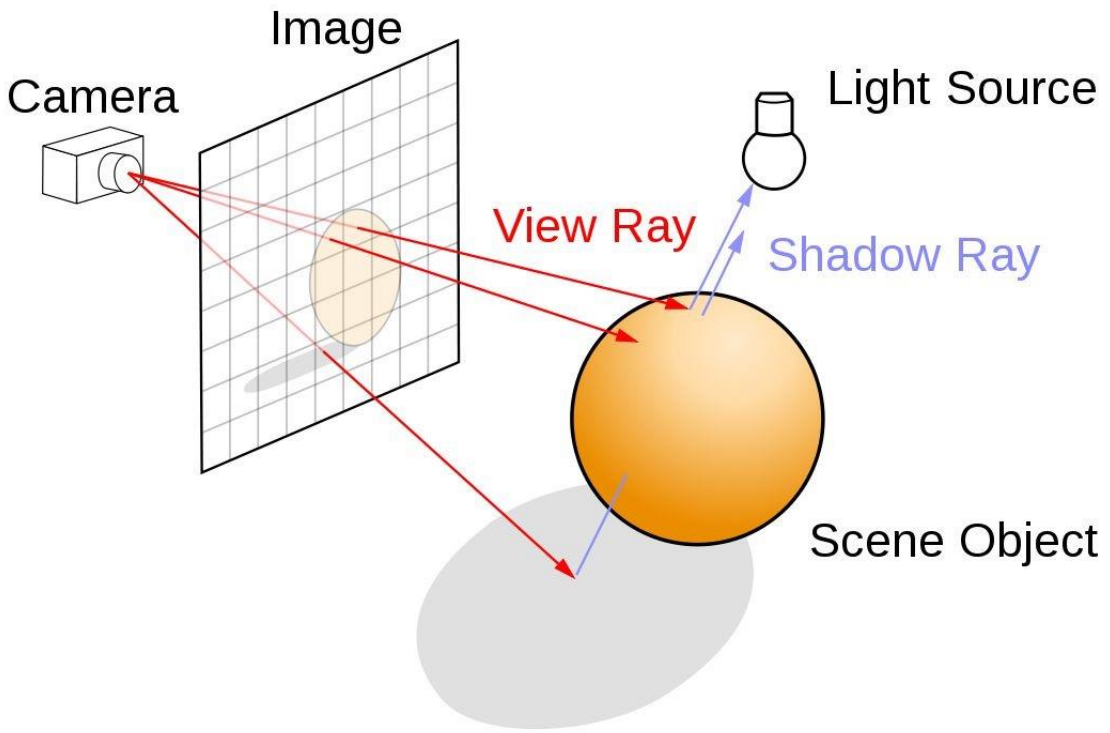

Figura 5 - Esquema para o algoritmo de traçado de raios. **[11]**

5.2. Direção do raio inicial

O primeiro passo é calcular a direção do raio inicial, a ser traçado da câmera contra a cena. Para isso, é preciso usar as coordenadas **x** e **y** da tela de onde o raio será lançado.

```
// Razão vertical e horizontal
rH = x / LARGURA_TELA
rV = y / ALTURA_TELA

// Cálculo da direção do raio
rDir = (0,0,1) + (rH - 0.5) * (1,0,0) + (rV - 0.5) * (0,1,0)
```

Seguindo o cálculo acima, caso o ponto se encontre no centro da tela, **rH** e **rV** serão iguais a 0.5 e o vetor **rDir** será **(0,0,1)**.

Em uma implementação sequencial, o código descrito acima iria percorrer a tela com uma estrutura de repetição para obter a posição dos pixels, entretanto no código usado neste trabalho utilizou-se a função **optixGetLaunchIndex** para obter as coordenadas **x** e **y** de cada ponto. Para traçar o raio, chama-se a função **optixTrace**.

5.3. Interseção

As interseções com triângulos são calculadas em hardware pela própria API, e portanto, como não foi acrescentado nenhum tipo de geometria diferente, não foi preciso implementar nada nesta etapa.

5.4. Normal

Para calcular a normal do ponto atingido, é feita uma interpolação linear das normais dos vértices com as coordenadas baricêntricas deste ponto.

A API oferece funções para obter as coordenadas baricêntricas **u** e **v** do ponto atingido no triângulo, e com isso pode-se calcular a coordenada **w** que equivale a **(1 - u - v).** Sendo N_a, N_b e N_c as normais dos vértices **A**, **B** e **C** respectivamente, do triângulo atingido, pode-se calcular a normal de um ponto da seguinte forma:

```
// Cálculo da normal para um ponto do triângulo
Ns = (1 - u - v) * Na + u * Nb + v * Nc
```

A normal do ponto é importante para definir o comportamento da luz em cada ponto da superfície. Como observado na imagem abaixo, caso a normal da face do

triângulo fosse utilizada para todos os pontos de sua superfície, então seria possível ver as faces dos triângulos na malha. A técnica para amenizar este efeito e dar uma aparência mais suave ao modelo é conhecida como *smooth shading* **[16]**.

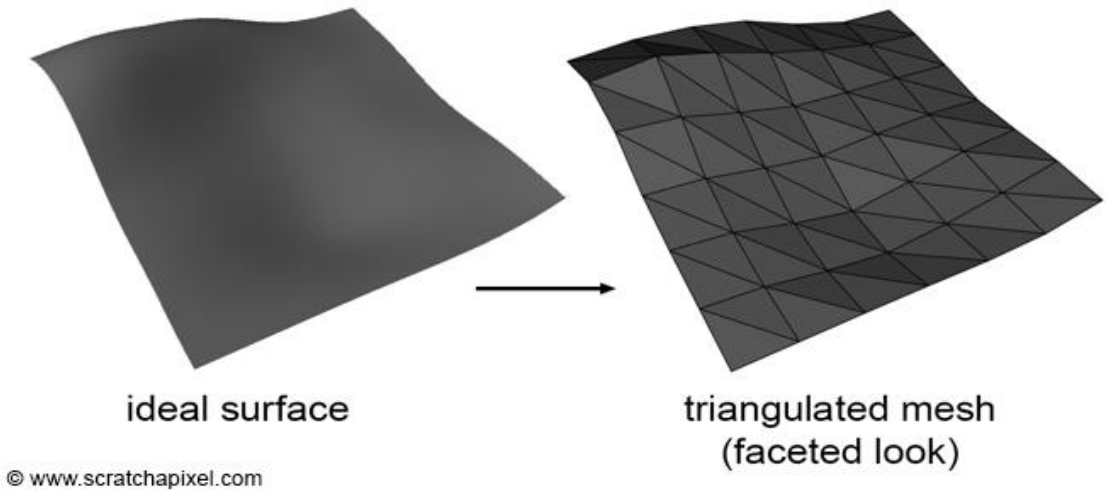

Figura 6 - Exemplo da técnica de *smooth shading* à esquerda e *flat shading* à direita.

5.5. Computando as sombras

Para determinar se um ponto está ou não na sombra deve-se traçar um raio do ponto atingido até todas as fontes luminosas, verificando se houve colisão em seu caminho.

A origem deste raio de sombra deve ser a posição do ponto atingido na superfície somado a um *bias*, para que se evite uma colisão do raio com o próprio objeto atingido — um fenômeno conhecido como *shadow acne* — que implicaria na aparição de diversos pontos pretos no modelo. Para calcular o *bias* basta multiplicar a normal do ponto em questão por uma constante pequena.

Assim, inicia-se um vetor de tamanho igual ao número de luzes da cena com todos os valores iguais a **(0.0, 0.0, 0.0)**. Caso o raio traçado até uma luz não encontre nada em seu caminho, então guarda-se o valor **(1.0,1.0,1.0)** no índice do vetor que representa esta luz, indicando que ela incide sobre o ponto. Na subseção abaixo, este vetor será utilizado para definir se é preciso calcular a cor final de um ponto ou não, dado que não seria eficiente realizar esse cálculo para um ponto que não receba nenhuma iluminação.

5.6. Cor final do ponto

Seguindo o modelo de iluminação de Phong é possível calcular a cor final do ponto atingido através do pseudocódigo abaixo. O vetor **luz_visivel** é o vetor mencionado na subseção anterior.

```
corDifusa = material.corDifusa
corEspecular = material.corEspecular

// calcular posição do ponto de interseção
surfPos = (1 - u - v) * vA + u * vB + v * vC

para cada indice, luz faça
      se luz_visivel[indice].x < 0.0 faça
            continuar
      direcao_luz = normalizar(luz.pos - surfPos)
      direcao_obs = normalizar(camera.pos - surfPos)

      cosLN = max(dot(Ns,direcao_luz),0)
      iluDifusa = (cosLN * corDifusa * 0.5) / NUMERO_DE_LUZES

      I = -direcao_luz
      brilho = 32.0
      refletido = refletir(I, Ns)
      cosRV = max(dot(direcao_obs,refletido),0.0)
      iluSpec =   pow(cosRV,brilho) * corEspecular * 0.3

      corFinal += (iluDifusa + iluSpec) / NUMERO_DE_LUZES

fim

// adicionar luz ambiente
corFinal += 0.2
```

Para que a iluminação difusa somada com a especular não ultrapasse o valor limite de uma cor, são colocados pesos de modo que o valor máximo possível para a cor final, mesmo após o acréscimo da luz ambiente, seja **(1,1,1)**. Outra medida tomada para que o valor da cor final não ultrapasse o limite é dividir a contribuição de cada luz pelo número de luzes da cena.

5.7. Texturização

A texturização é basicamente o processo de aplicar uma imagem em uma superfície, e para isso é necessário que cada um dos triângulos desta superfície possua em seus vértices **A**, **B** e **C** uma coordenada **(x,y)** mapeando-o na imagem da textura desejada.

Com estes dados, para definir qual pixel de uma textura será designado para qualquer ponto de um triângulo, basta realizar uma interpolação linear das coordenadas baricêntricas **u**, **v** e **w** com as coordenadas de textura de cada vértice **A**, **B** e **C** do triângulo atingido.

```
// Para um dado ponto do triângulo
texCoords = (1 - u - v) * TexA + u * TexB + v * TexC
```

O processo é ilustrado pela imagem abaixo, mostrando que os únicos dados necessários para aplicar texturas são as coordenadas **(0,1)**, **(0.5,0)** e **(1,1)**, uma vez que através do cálculo acima podemos obter as coordenadas dos outros pontos da superfície do triângulo.

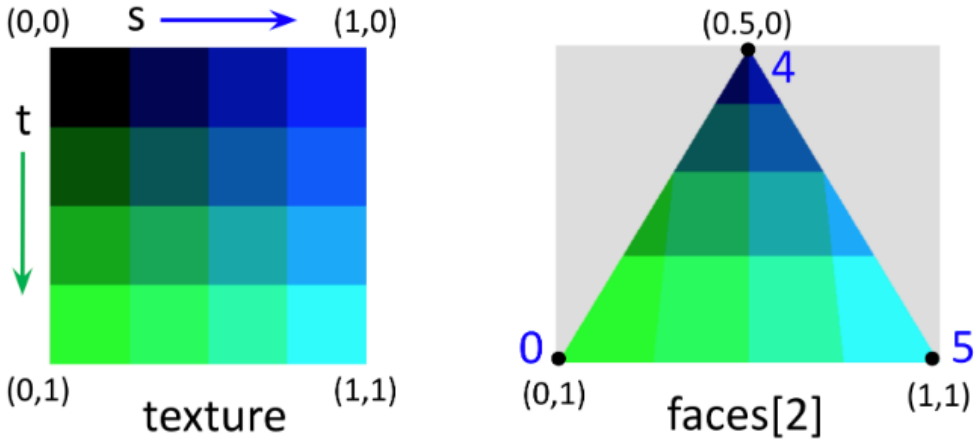

Figura 7 - Mapeamento de textura para um triângulo. **[14]**

Finalmente, obtém-se a cor da textura acessando-a no ponto (**texCoords**.x, **texCoords**.y) e multiplica-se este valor pela cor difusa antes de realizar o cálculo da subseção anterior.

5.8. Gerando e aplicando coordenadas de texturas

Devido à complexidade do modelo, a ideia para facilitar o processo de texturização seria gerar as coordenadas de textura para uma malha de um tipo específico como uma *box*, e aplicar estas coordenadas em todas as outras malhas de mesmo tipo de forma automática.

Primeiramente, escolhi uma *box* qualquer do modelo, e a abri com a ferramenta Blender **[14]**.

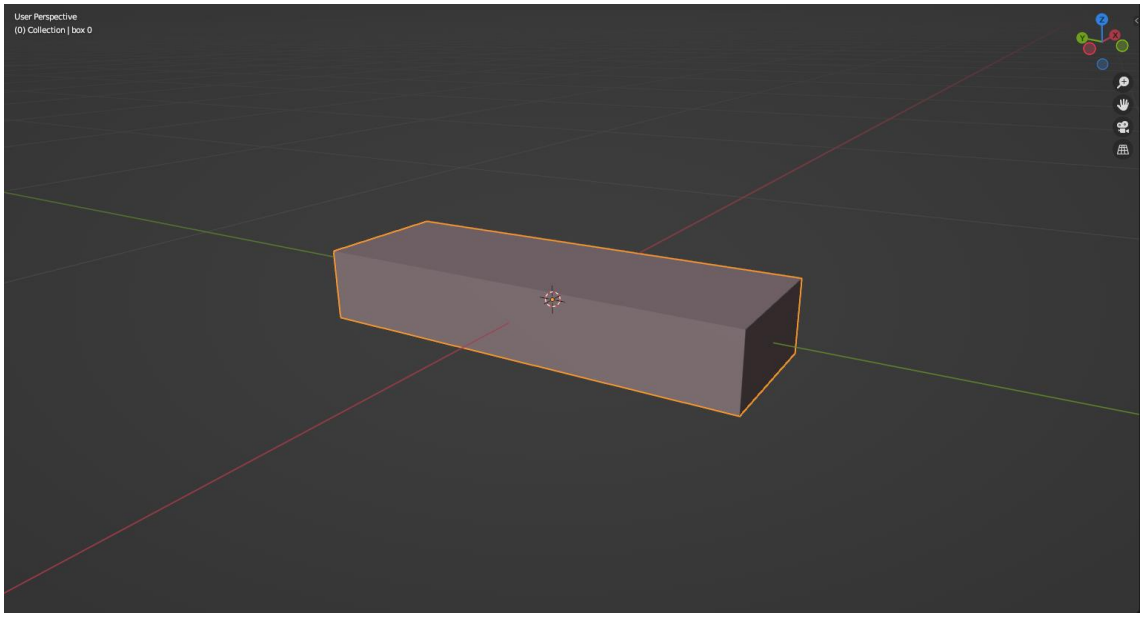

Figura 8 - Malha do tipo box aberta no *Blender*.

Depois apliquei a textura de corrosão escolhida e utilizei o método *unwrap*, que gera as coordenadas de textura.

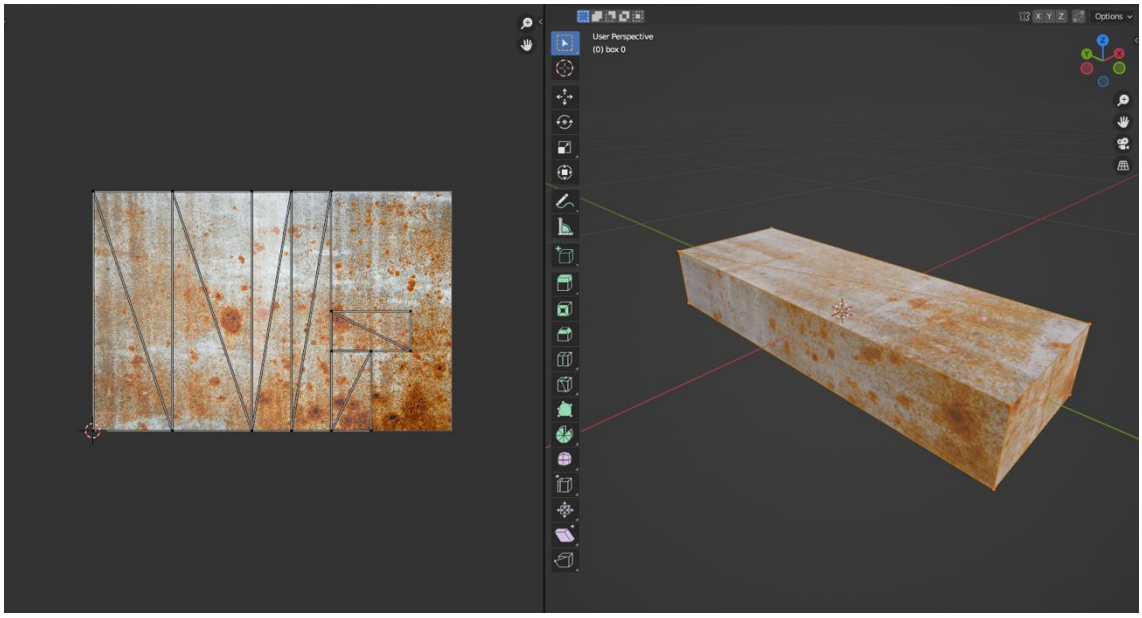

Figura 9 - Textura de corrosão aplicada na caixa.

Após exportar o modelo obtive um arquivo **obj** da caixa com as coordenadas de textura **Vt** geradas.

Com isso, apliquei estas coordenadas em todas as malhas do tipo *box* do modelo **Flare System**, iterando por todas as definições de faces das *boxes* e atribuindo os índices das coordenadas de textura geradas aos vértices de cada face.

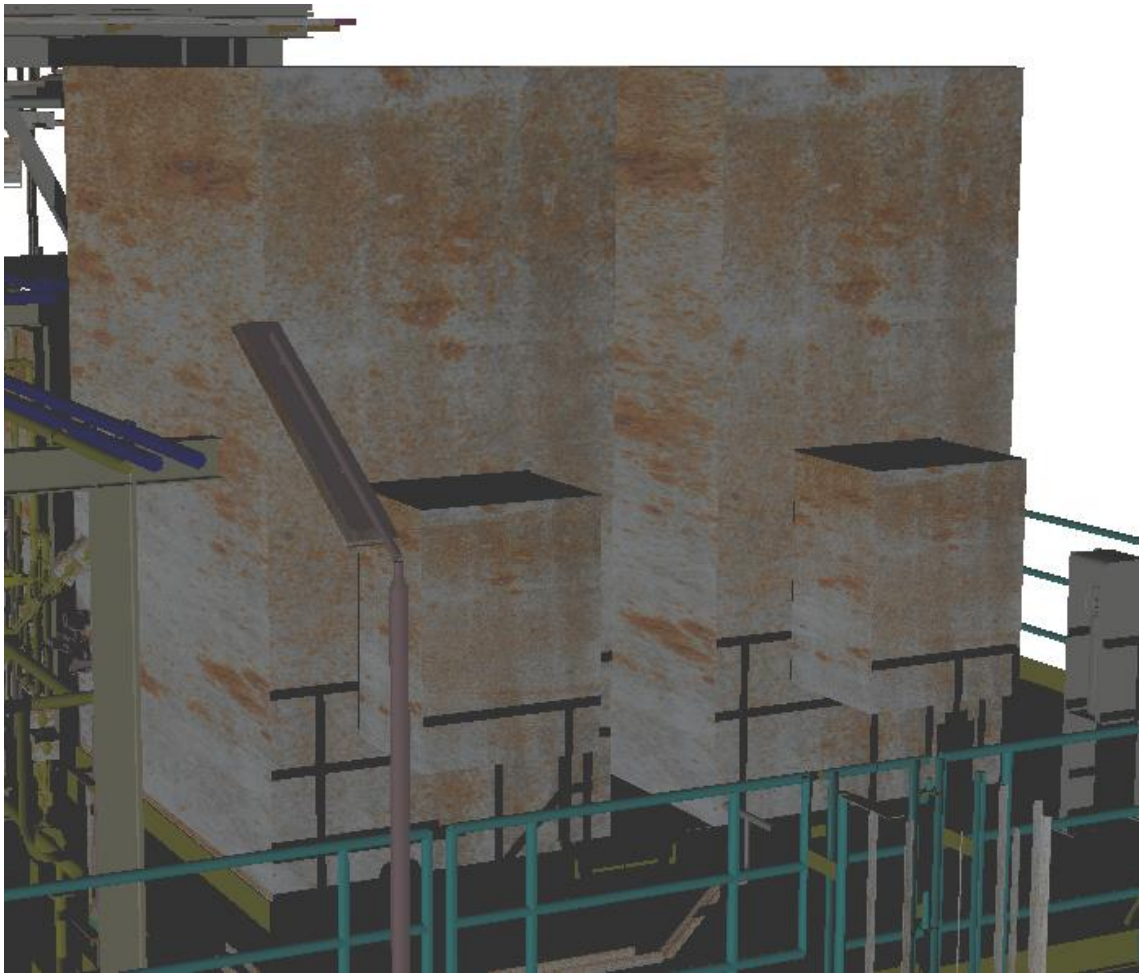

Figura 10 - Textura aplicada no modelo da plataforma P-74.

6. Funcionamento da OptiX 7.0

6.1. Pipeline

Como existem diversas maneiras de implementar o algoritmo de traçado de raios, as APIs como a OptiX precisam definir o comportamento de um programa de uma maneira mais organizada para fazer uso das ferramentas de aceleração de hardware. Com isso, surge a ideia de criar um *pipeline* para o programa, no qual algumas partes como o cálculo de interseções são executadas no *hardware* e outras podem ser configuradas pelo usuário através de programas específicos.

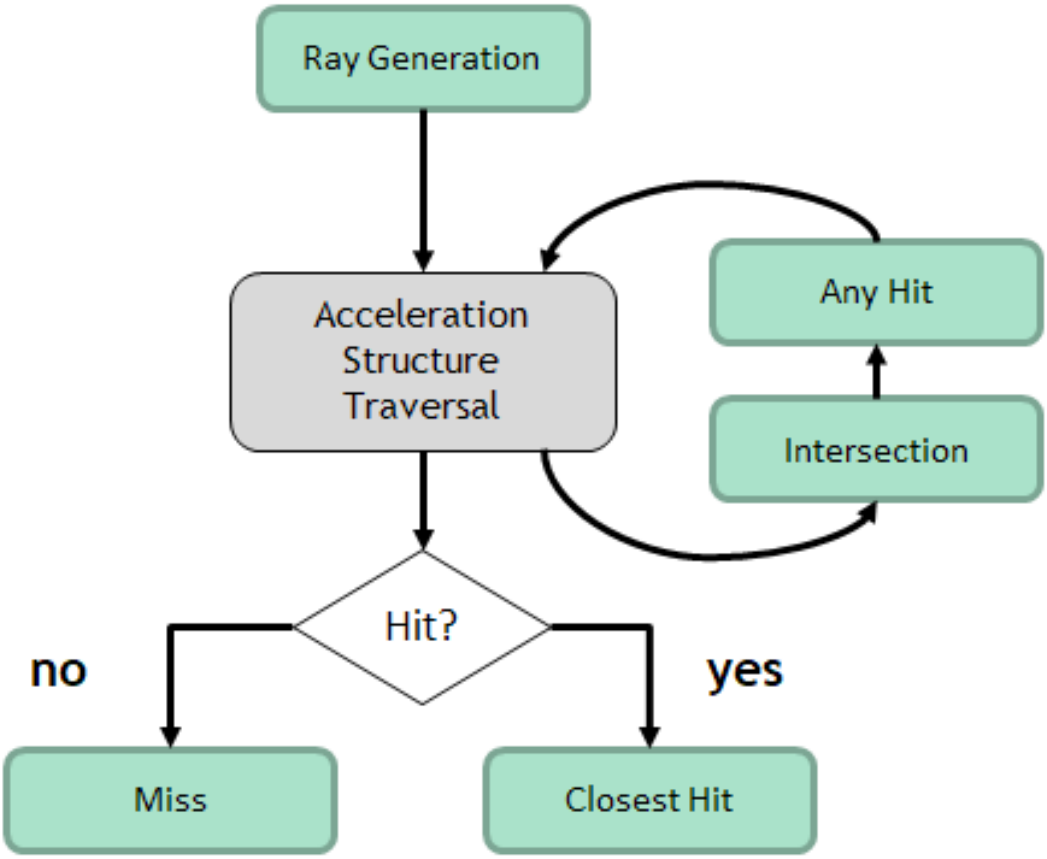

Figura 11 - Esquema do *pipeline* usado na API *OptiX*.

Inicialmente, o programa *ray generation* traça os raios através da função **optixTrace**. Normalmente, é lançado um raio por pixel partindo da câmera, mas já é possível fazer modificações nesta etapa e lançar múltiplos raios por pixel ou raios partindo de fontes luminosas ao invés da câmera.

Assumindo que o raio se aproximou de uma primitiva, caso ela seja um triângulo então os cálculos de interseção serão feitos no *hardware*, mas para geometrias definidas pelo usuário utiliza-se o programa *intersection*. Se o programa *intersection*

determinar que o raio realmente atingiu a primitiva, então os dados desta interseção são passados para o programa *any hit*, no qual é possível rejeitar uma interseção no caso do uso de *alpha textures* ou guardar quantos triângulos já foram atingidos por um certo raio, por exemplo.

Cada interseção não rejeitada é analisada pelo *pipeline*, que mantém qual foi a mais próxima até o momento. Após percorrer a estrutura de aceleração por completo e ter certeza sobre qual foi a interseção mais próxima, o programa *closest hit* é chamado recebendo os dados dela, para que o cálculo da cor do pixel seja realizado. Caso nenhuma interseção seja encontrada, então o programa *miss* é chamado para não fazer nada ou só determinar a cor do fundo da cena.

É importante notar que esta ainda é uma visão simplificada do *pipeline*, visto que os programas citados podem possuir comportamentos diferentes caso existam diferentes tipos de raio.

6.2. Exemplo de pipeline simples

Um exemplo de *pipeline* simples seria:

RayGeneration: Chama **optixTrace** para cada pixel da tela. Cor do pixel armazenada em um PRD *color*.
Miss (*radiance rays*): Seta *color* para **(1,1,1)** para definir a cor de fundo como branca.
ClosestHit: Seta *color* para a cor difusa do objeto atingido adicionando uma luz ambiente.

6.3. Shader Binding Table

Todos os programas do *pipeline* utilizam dados fornecidos pelo usuário, obtidos através de uma tabela chamada de SBT ou *Shader Binding Table*. No código do *host*, são criados registros na SBT para cada tipo de raio e programa, nos quais podem ser guardados dados como vértices, normais, coordenadas de textura e outros. Um programa pode acessar os dados da SBT usando a função **optixGetSbtDataPointer**.

6.4. Struct Launch Params

Ao contrário das versões anteriores, na OptiX 7.0 a comunicação entre *host* e dispositivo é feita através de uma única estrutura chamada de *Launch Params*. Alguns dados passados do *host* para o dispositivo são o tamanho da tela, posição da câmera e posição das luzes da cena. Do dispositivo para o *host* só é preciso enviar um *buffer* preenchido com as cores dos pixels calculadas.

6.5. PRD

O PRD ou *Per Ray Data* é basicamente uma estrutura para passar dados entre diferentes shaders ou programas. Ao chamar a função **optixTrace** passamos um ponteiro para esta estrutura, de modo que o próximo programa invocado poderá modificar seu valor.

Neste trabalho, o programa *miss* específico para raios de sombra precisou usar este mecanismo para informar o programa *closest hit* se o seu ponto se encontrava na sombra ou não. Outro exemplo é o próprio programa de *closest hit* que deve enviar a cor final calculada para o programa *ray generation*, que irá escrevê-la em um *buffer* da struct *Launch Params*, estabelecendo assim uma comunicação com o *host*.

7. Pipeline implementado

RayGeneration: Chama **optixTrace** para cada pixel da tela. Cor do pixel armazenada em um PRD *color*.
Miss (*radiance rays*): Seta *color* para **(1,1,1)** para definir a cor de fundo como branca.
Miss (*shadow rays*): Seta *visibility* para **(1,1,1)**, para indicar que não existe nada no caminho deste raio de sombra e, portanto, o ponto de origem está visível.
ClosestHit: Armazena visibilidade do ponto avaliado em um PRD *visibility*. Chama **optixTrace** para todas as fontes luminosas realizando os cálculos para definir a cor do pixel de acordo com o modelo de iluminação de Phong. Também aplica a textura do material do objeto atingido.

Na versão do tutorial, era possível adicionar apenas uma luz de forma *hard coded* no *shader*. Uma melhoria realizada foi passar os dados sobre a posição das fontes luminosas do *renderer* para o *shader*.

A maior vantagem é a de que no *renderer* temos a informação da caixa do modelo, sendo mais fácil posicionar luzes usando o centro desta caixa como referência.

Além disso, o cálculo da cor final foi refeito de acordo com o modelo de iluminação de Phong. Antes de renderizar o modelo completo com este *pipeline*, foram realizados alguns testes com modelos bem simples.

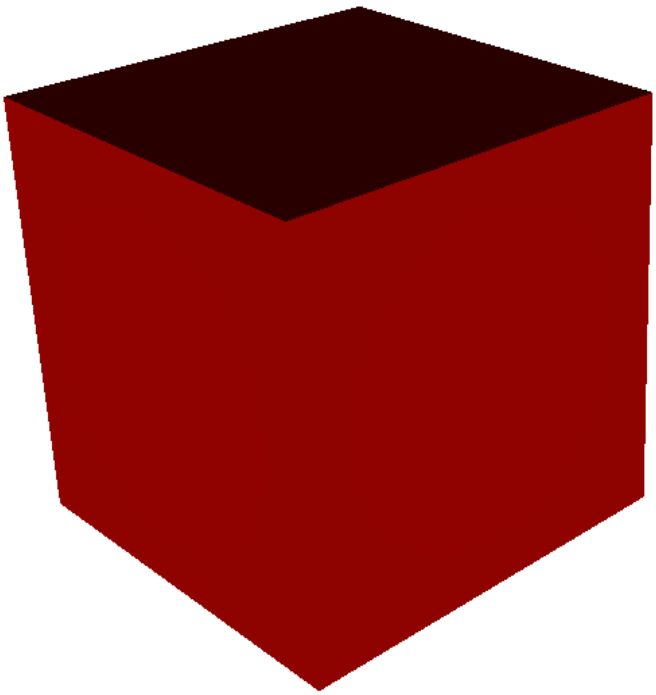

Figura 12 - Foto de um cubo renderizado com traçado de raios e duas de suas faces iluminadas.

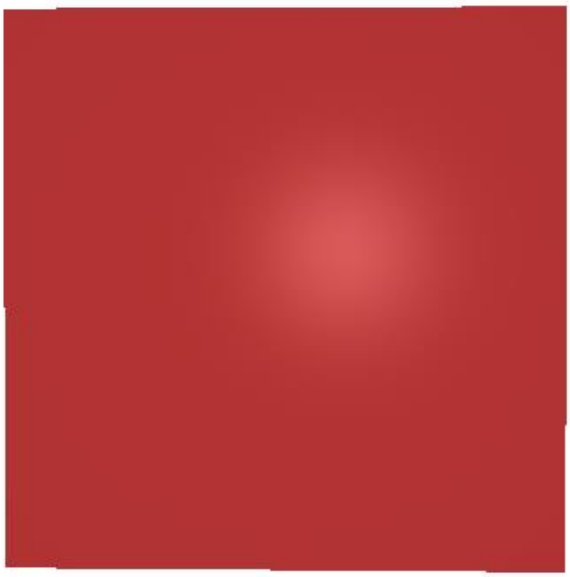

Figura 13 - Componente especular calculada no cubo.

8. Resultados

8.1. Comparações com o Environ

À esquerda estão as imagens do modelo renderizado no sistema Environ e na direita estão as renderizadas com o método de traçado de raios. As sombras calculadas no método de traçado de raios contribuíram com uma noção de profundidade na cena, isto é, a distância entre os objetos é mais evidente. A textura de corrosão adicionada nas malhas do tipo *box* contribuiu para um maior realismo da cena. Em relação ao desempenho, foi atingida uma taxa de 60 quadros por segundo.

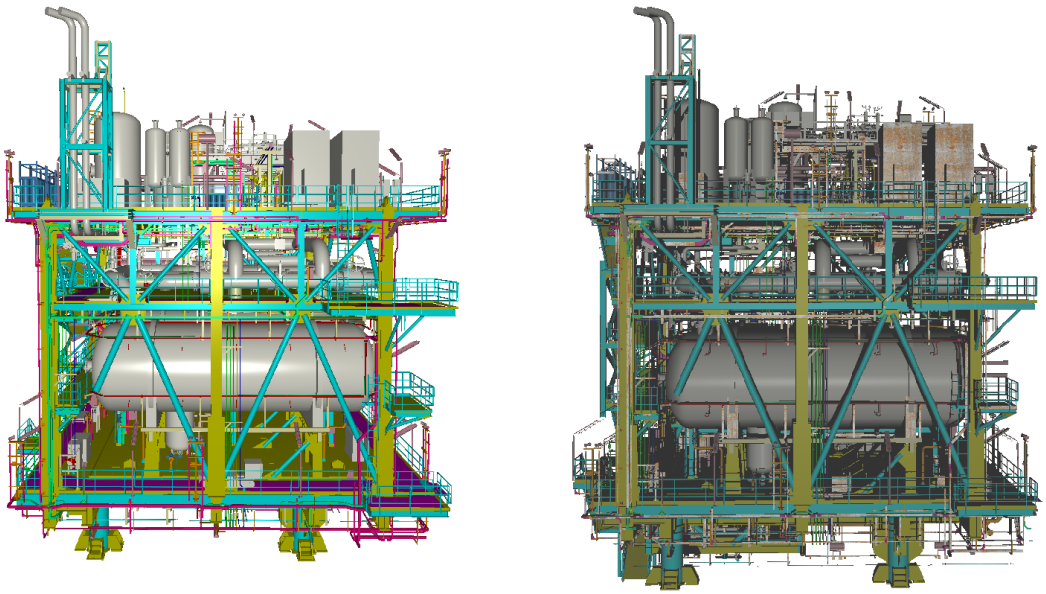

Figura 14 - Comparação lateral dos modelos. (1)

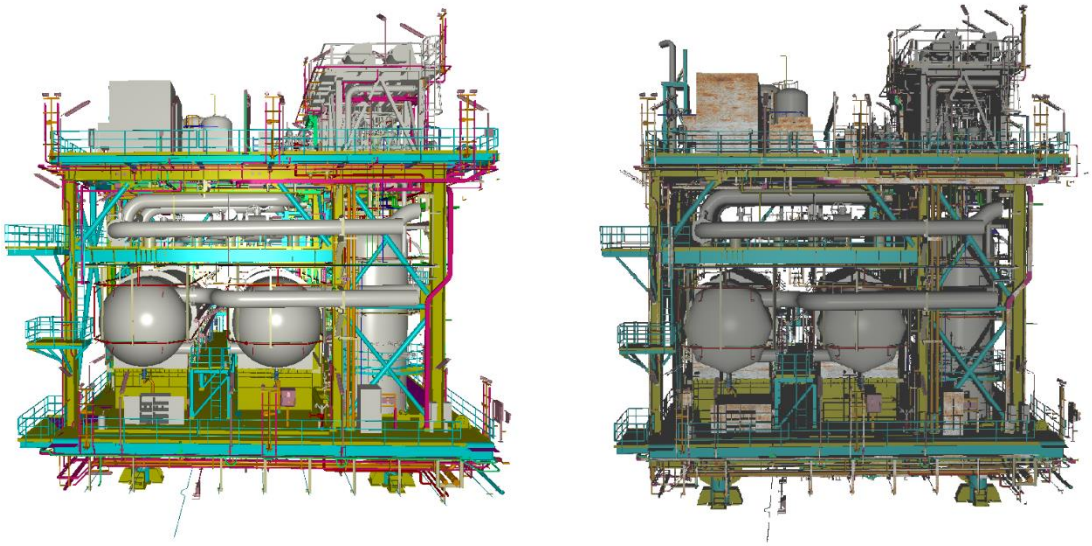

Figura 15 - Comparação lateral dos modelos. (2)

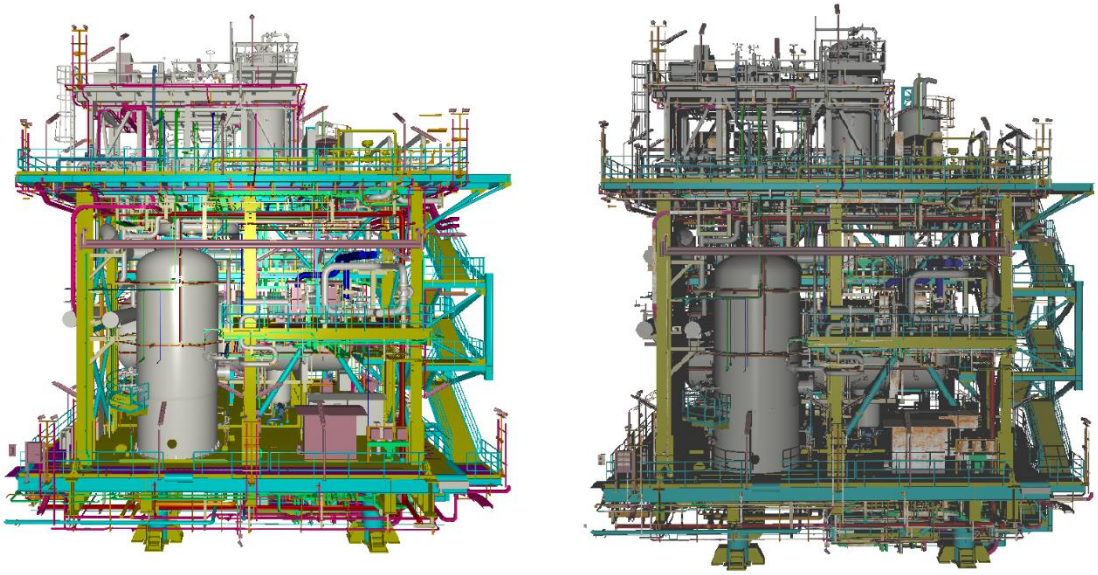

Figura 16 - Comparação lateral dos modelos. (3)

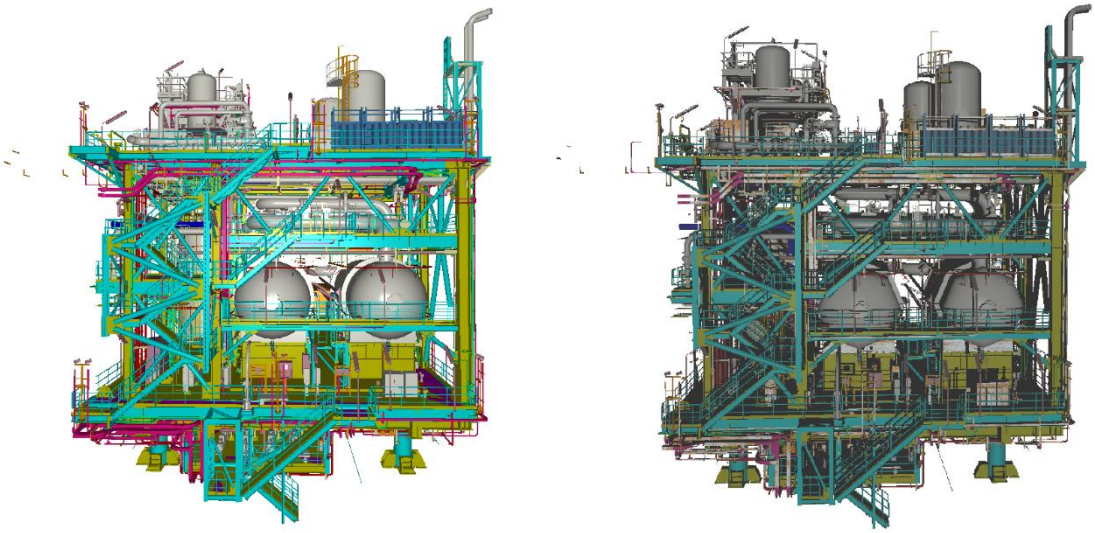

Figura 17 - Comparação lateral dos modelos. (4)

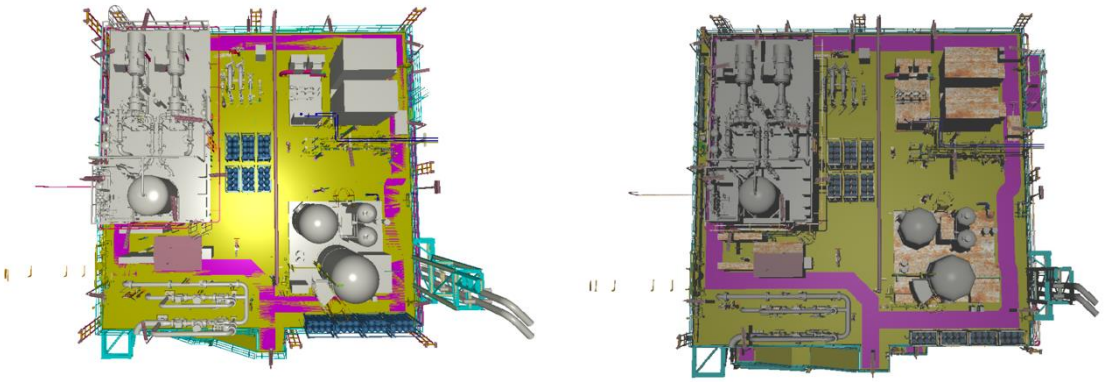

Figura 18 - Comparação do topo dos modelos.

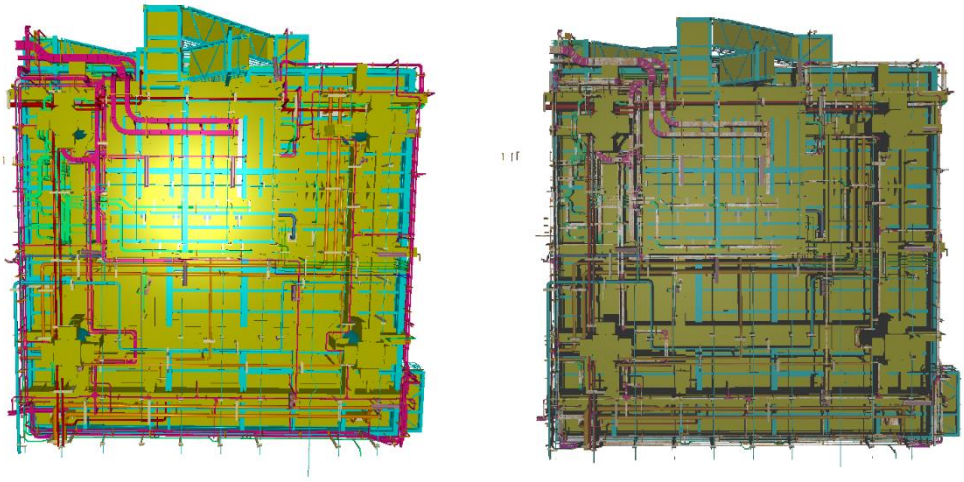

Figura 19 - Comparação do fundo dos modelos.

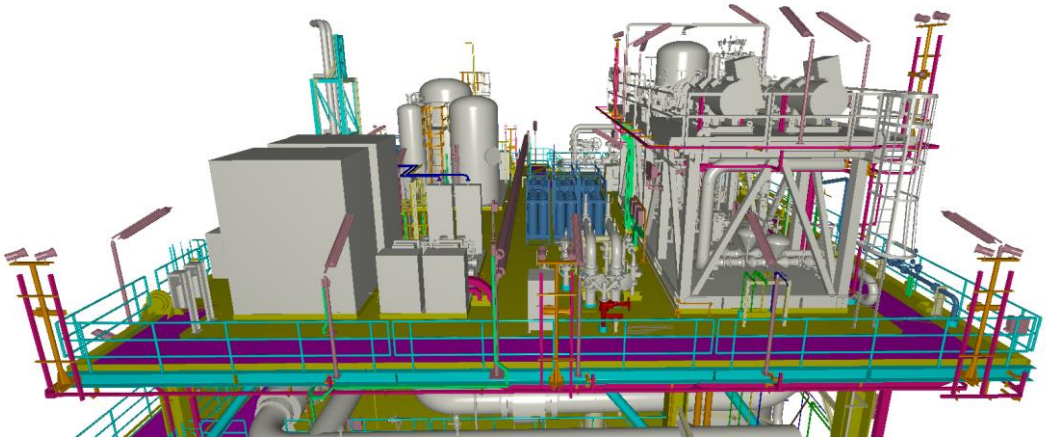

Figura 20 - Visão mais próxima do topo, renderizada no Environ.

Figura 21 - Visão mais próxima do topo, renderizada com traçado de raios.

8.2. Resultado com duas luzes na cena

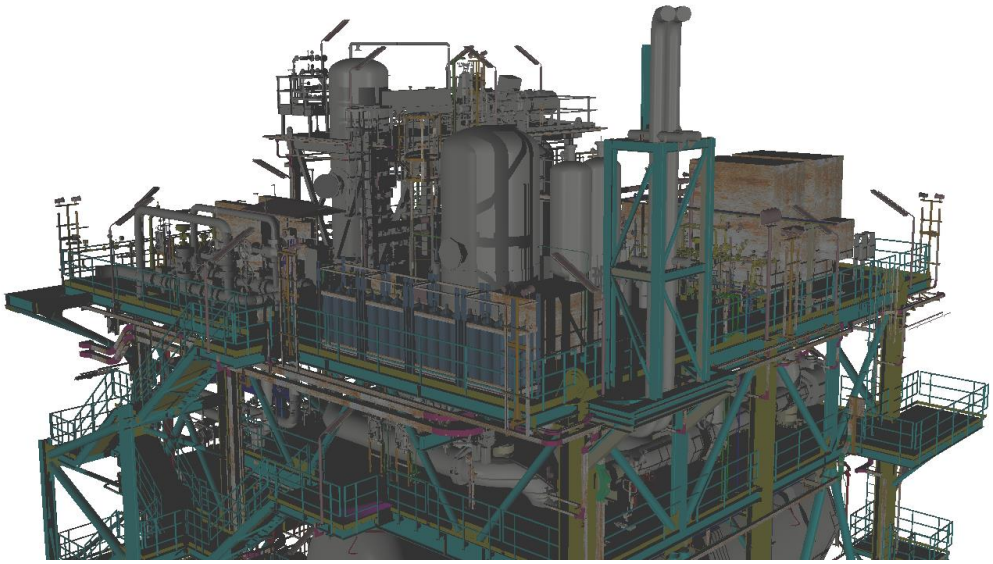

Figura 22 - Visão do topo, iluminado com duas luzes diferentes. Renderizado com traçado de raios.

9. Conclusão e Trabalhos Futuros

O sistema Environ não faz nenhum tipo de cálculo para sombras e por isso é fácil perceber que as cenas renderizadas com o traçado de raios possuem uma noção de profundidade melhor. Seria interessante comparar os resultados obtidos com os de um algoritmo de rasterização que também implementasse sombreamento fazendo uso de *shadow maps*.

Existem vários caminhos interessantes para aumentar o realismo da cena, como o cálculo da iluminação global, tratamento de objetos transparentes com o programa *any hit* ou a utilização de materiais PBR com uma função BRDF.

10. Referências

[1] Appel, Arthur. Some techniques for shading machine renderings of solids. Acessado em: Maio/2022

[2] Caulfield, Brian. What's the Difference Between Ray Tracing and Rasterization. Acessado em: Maio/2022

[3] Cummings, Chris. Why Transparency is Hard. Acessado em: Maio/2022

[4] Whitted, Turner. A Ray-Tracing Pioneer Explains How He Stumbled into Global Illumination. Acessado em: Maio/2022

[5] Ziebert, Alexandre. Precisamos falar sobre Ray Tracing. Acessado em: Maio/2022

[6] Introduction to Shading. Acessado em: Maio/2022

[7] The Difference Between CAD and 3D Modeling. Acessado em: Maio/2022

[8] RTX accelerated ray tracing with OptiX. Acessado em: Junho/2022.

[9] NVIDIA OptiX 7.0 Programming Guide. Acessado em: Junho/2022.

[10] Environ. Acessado em: Novembro/2022.

[11] Ray Tracing. Acessado em: Novembro/2022.

[12] Modelos de iluminação. Acessado em: Novembro/2022.

[13] Blender. Acessado em: Novembro/2022.

[14] CS307: Texture Coordinates, Repeating Textures & Curved Surfaces.
Acessado em: Novembro/2022.

[15] GTC 2020: RTX Accelerated Raytracing With Optix 7.
Acessado em: Dezembro/2022.

[16] Basic Lighting. Acessado em: Dezembro/2022