Universidade Federal de Mato Grosso do Sul
Faculdade de Computação

Trabalho de Conclusão de Curso

Estendendo uma API Gráfica com Materiais Programáveis

Aluno: Alexandre Libório da Fonseca
Orientador: Paulo Aristarco Pagliosa

Resumo

Em computação gráfica, uma das principais formas de renderização é a rasterização, que envolve projetar os polígonos da cena sobre o plano da tela e calcular quais pixeis eles cobrem. O controle sobre a aparência visual deste processo é concedido ao desenvolvedor por meio de shaders, programas executados diretamente na GPU. Um desses shaders é o shader de fragmento, ou shader de pixel, responsável por calcular sombras e efeitos que ditam a cor final do pixel. Este trabalho se insere no contexto da biblioteca de classes Ds, uma ferramenta desenvolvida durante a disciplina de Computação Gráfica que utiliza a rasterização via OpenGL, API gráfica comum na academia. Nesta biblioteca, o material dos atores utiliza um único shader de fragmento, que aplica o modelo de iluminação de Phong, com suas propriedades modificáveis expostas na interface gráfica. O objetivo deste trabalho é, portanto, estender esta funcionalidade, permitindo que os materiais tenham shaders diferentes, inclusive aqueles definidos pelo próprio usuário. Isso exige que a interface seja construída dinamicamente, com base nas propriedades específicas do shader selecionado em um dado material. Além disso, uma reescrita do loop de renderização, para que consiga renderizar cada ator com seu devido shader, em uma mesma cena.

1 Introdução

Em computação gráfica, existem duas formas principais de síntese de imagem, ou renderização: o traçado de raios (*Ray Tracing*) e a rasterização. Atualmente, ambos os processos podem ser acelerados por hardware, utilizando a Unidade de Processamento Gráfico (GPU), com a rasterização tendo esse suporte a mais tempo. O traçado de raios consiste em lançar raios a partir da câmera, que ao colidirem com um objeto da cena, têm sua cor calculada com base na contribuição de cada fonte de luz que o atinge, levando em consideração as propriedades da superfície. Este método simula o comportamento físico da luz, ideal para o fotorrealismo, em troca de tempo de processamento. Na rasterização, em vez de um cálculo por pixel, os polígonos da cena (normalmente triângulos) são projetados sobre o plano da tela para determinar quais pixeis eles cobrem.

No âmbito da rasterização, a flexibilidade e o controle visual são alcançados por meio do pipeline de renderização programável, uma arquitetura moderna que substituiu os antigos sistemas de função fixa. O controle sobre esse pipeline é exercido através de *shaders*: pequenos programas executados diretamente na GPU que ditam a aparência final dos objetos. Dentre os principais tipos, o shader de vértice (*Vertex Shader*) atua sobre os vértices dos modelos 3D para manipular sua forma e posição, enquanto o shader de fragmento (*Fragment Shader*) é responsável por calcular a cor final de cada pixel, aplicando texturas, iluminação, sombras e outros efeitos.

Foi desenvolvida como ferramenta para a disciplina de computação gráfica da FACOM uma biblioteca gráfica, chamada de Ds, que implementa ambos os processos, além de fornecer classes e modelos para objetos matemáticos, como vetores, pontos, quaternions e transformações; estrutura de dados espaciais, como grids e árvores; e gerenciamento de cena em grafos com hierarquia de objetos. Sua rasterização utiliza o OpenGL para renderização, API gráfica que até então era o padrão na academia (mais detalhada na Seção 2). Nesta biblioteca, seus atores só têm disponível um único material, que utiliza o modelo de iluminação de Phong (explicado na Seção 5) em um shader de fragmento, e suas propriedades podem ser modificadas na interface gráfica ao selecioná-lo. Esta interface é implementada utilizando *Dear ImGui*[1], uma biblioteca de código aberto para criação de interfaces gráficas em C++, focada em simplicidade onde todos seus componentes são declarados e construídos dinamicamente em código.

Diante desta limitação, o objetivo geral do trabalho é estender a biblioteca Ds, para que o material possa utilizar outros modelos de iluminação, com propriedades variadas e incluir alguns materiais de exemplo. Os objetivos específicos consistem em:

- Incorporar modelos de iluminação personalizados, desenvolvidos pelo usuário.
- Mostrar as propriedades do material na interface gráfica de forma dinâmica.
- Renderizar em uma mesma cena, objetos com materiais que tenham modelos de iluminação distintos.

Para concretizar estes objetivos, foi desenvolvida uma extensão da Ds que integra três componentes principais: um sistema de arquivos de shader personalizados, uma interface gráfica dinâmica e uma lógica de renderização estendida. O primeiro passo foi a criação de uma linguagem de domínio específico que permite ao usuário definir a lógica de um shader e suas propriedades em arquivos de texto com a extensão `.shlb`. O design desta linguagem foi centrado na separação de responsabilidades: um bloco `Method` para abrigar a lógica de iluminação, e um bloco `Property` para declarar as variáveis que este shader expõe.

Para a criação de uma interface gráfica dinâmica, foi implementado um analisador sintático que interpreta os arquivos `.shlb` e constrói uma representação em memória de cada shader, incluindo uma lista estruturada de suas propriedades. Esta lista informa a interface gráfica sobre quais controles de usuário devem ser gerados. Ao selecionar um ator, o sistema inspeciona as propriedades de seu material e, em tempo real, instancia os componentes apropriados, como seletores de cor para propriedades do tipo `color` e controles deslizantes (*sliders*) para as do tipo `float`. As alterações feitas pelo usuário nesses controles são então armazenadas em um buffer de dados na CPU, associado à instância do material.

Finalmente, para viabilizar a renderização de múltiplos materiais distintos em uma mesma cena, a lógica de renderização foi reescrita. Durante o desenho de cada

ator, o sistema agora executa uma sequência de passos específica: primeiro, ele escolhe o shader que será usado para renderizar o ator a partir de seu material; em seguida, sincroniza os dados do buffer da CPU (modificados pela interface) para um buffer na GPU, gerenciado pelo OpenGL, chamado de *Uniform Buffer Object* (UBO) (explicado na Seção 2.3); e, por fim, vincula este UBO ao pipeline antes de emitir o comando de desenho. Este processo, repetido para cada ator na cena, garante que todo ator é renderizado seguindo a lógica do material escolhido para o mesmo.

O texto é organizado da seguinte forma: a Seção 2 explica como funciona a pipeline gráfica do OpenGL e onde os shaders se encaixam. A Seção 2.3 apresenta como são enviados os dados à GPU e como funciona o vínculo de um UBO com um shader. A Seção 4.1 explica a estrutura do arquivo de shader que o desenvolvedor pode escrever e como o programa o lê e armazena seus dados. A Seção 4.2 apresenta como os atributos do material são mostrados na interface gráfica. A Seção 4.3 apresenta como é feita a renderização dos objetos da cena, levando em consideração seus diferentes materiais. Exemplos da renderização são mostrados na Seção 5. A Seção 6 conclui o trabalho e apresenta possíveis melhorias e extensões.

2 Fundamentos

2.1 Pipeline do OpenGL

O pipeline gráfico é o conjunto de etapas para transformar uma descrição de cena tridimensional em uma imagem bidimensional. Em OpenGL, seu pipeline de renderização é composto por estágios fixos (pré-definidos) e estágios programáveis, onde o desenvolvedor insere sua lógica através de shaders. Uma visão simplificada do pipeline é apresentada na Figura 1, com os seguintes estágios:

1. **Especificação dos vértices**: A aplicação envia à GPU os dados que compõem os objetos da cena, organizados em uma lista de vértices, onde cada vértice pode conter atributos como posição, cor, coordenadas de textura e vetor normal.

2. **Shader de vértice**: O primeiro estágio programável, onde o shader é executado para cada vértice enviado, manipulando a posicão, rotação e escala do objeto.

3. **Montagem de primitivas**: Neste estágio de função fixa, os vértices processados pelo shader de vértice são agrupados em primitivas, podendo ser pontos, linhas ou triângulos, com base em como foi configurado pelo usuário.

4. **Rasterização**: Após a montagem, este estágio de função fixa determina quais pixeis na tela são cobertos por cada primitiva. O resultado não é um pixel final, mas sim um fragmento, que contém atributos interpolados dos vértices que o formam, como por exemplo coordenada de textura e vértice normal.

5. **Shader de fragmento**: Para cada fragmento gerado na rasterização, é executado o shader de fragmento, responsável por calcular sua cor final, aplicando texturas, realizando cálculos de iluminação e quaisquer outros efeitos. O resultado é uma cor e um valor de profundidade do fragmento.

6. **Operações por Amostra**: Este é o estágio final de função fixa. Antes de serem escritos na tela, os fragmentos processados passam por uma série de testes,

como o teste de profundidade, que descarta fragmentos que estão atrás de outros objetos já desenhados, garantindo a oclusão correta. Por fim, a imagem é escrita na memória, pronta para ser exibida.

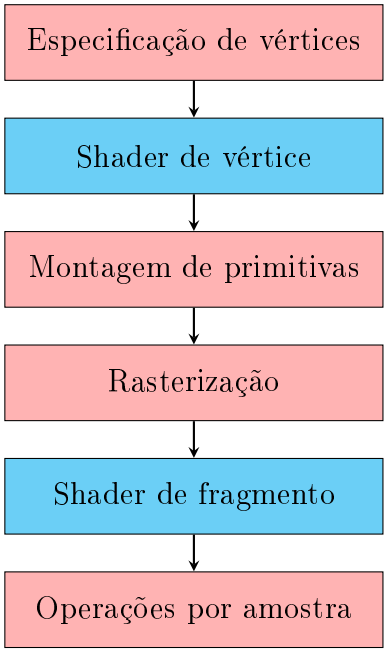


Figura 1: Pipeline simplificada, os estágios em azul são programáveis

O processo de renderização de um objeto inicia-se com a definição de seus shaders de vértice e fragmento. Estes shaders são então compilados e vinculados em uma única unidade executável, em um objeto conhecido como programa de shader (*shader program*). Durante o loop de renderização, a aplicação primeiro ativa o programa de shader desejado, configurando o estado do pipeline gráfico. Em seguida, com os dados dos vértices do objeto já disponíveis na GPU, a aplicação emite uma chamada de desenho (*draw call*). Este comando atua como um gatilho, instruindo a GPU a executar todo o pipeline de renderização sobre os vértices, utilizando o programa de shader atualmente ativo. Uma vez que a chamada de desenho é enviada, o controle do processo é transferido para a GPU, e a aplicação não tem mais interação com aquele ciclo de renderização específico. Para renderizar um objeto com uma aparência diferente, é necessário criar outro programa de shader, ativá-lo, e emitir uma nova chamada de desenho para aquele objeto.

2.2 Shaders

Conforme mencionado, shaders são os programas executados nos estágios programáveis do pipeline de renderização do OpenGL. Estes programas são escritos em GLSL (*OpenGL Shading Language*), uma linguagem de alto nível com sintaxe baseada em C, projetada para ser executada em GPU, com uma função `void main()` como seu ponto de entrada. Para isso, ela inclui um conjunto de tipos de dados e funções que simplificam as operações matemáticas comuns em computação gráfica. Entre seus tipos de dados estão vetores (`vec2`, `vec3`, `vec4`), matrizes (`mat2`, `mat3`, `mat4`) e samplers (`sampler2D`), que são usados para acessar e manipular texturas. Também são disponibilizadas funções pré-definidas como produto escalar (*dot*), produto vetorial

(*cross*), normalização (*normalize*) e interpolação linear (*mix*) para cálculos de iluminação, transformações e outros efeitos visuais.

A operação de um shader depende de suas entradas e saídas. As entradas podem vir de duas fontes: da aplicação (dados uniformes) ou do estágio anterior do pipeline (atributos ou variáveis de entrada). As saídas, por sua vez, são passadas para o próximo estágio do pipeline. A GLSL utiliza um sistema de qualificadores de variáveis para definir cada uma dessas vias de comunicação. Os três principais são:

- **in**: Define os dados de entrada do shader. No caso do shader de vértice, é sua coordenada no espaço, podendo também conter atributos como cor e coordenada de textura. No shader de fragmento, são os valores de saída do shader de vértice interpolados da rasterização.
- **out**: Define as saídas do shader. No shader de vértice, esta é a posição final do vértice, além de saídas customizadas para o shader de fragmento. No shader de fragmento, a saída é a cor final do fragmento.
- **uniform**: Variável cujo valor é definido pela aplicação (na CPU) e permanece constante para todos os vértices e fragmentos processados por um programa de shader durante uma única chamada de desenho (*draw call*). Alguns exemplos incluem matrizes de transformação e atributos da câmera (posição e direção) e das luzes (posição, cor, intensidade, direção) e do material.

As variáveis que contêm o mesmo qualificador podem ser agrupadas em blocos de interface(*interface blocks*)[2], permitindo que elas sejam tratadas como uma única unidade lógica. A declaração é similar a uma `struct`, porém é declarada usando o qualificador no lugar da palavra reservada. Também é possível nomear o bloco, assim uma variável chamada `position` em um interface block chamado `camera` seria acessada como `camera.position`. Um bloco de interfaces de uniforms também é chamado de bloco de uniform (*uniform block*).

A transferência de dados para um bloco de uniform é realizada de forma diferente de um uniform tradicional, onde é apenas necessário obter sua localização no shader e enviada seu valor, através de chamadas da API. No bloco de uniform, essa transferência deve ser feita por meio de um *Uniform Buffer Object* (UBO) (detalhado na Seção 2.3), um bloco de memória na CPU que deve espelhar com exatidão o layout esperado pelo shader na GPU. Para construir este bloco de memória corretamente, a aplicação necessita de duas informações para cada variável: seu tamanho e seu deslocamento (*offset*) a partir do início do bloco.

Enquanto o tamanho de cada variável é bem definido e corresponde diretamente aos tipos de dados em C++ (um `vec3` ocupa o espaço de três `floats`, uma `mat4` ocupa o de dezesseis `floats`, etc.), o OpenGL não especifica rigidamente o espaçamento entre essas variáveis. Essa flexibilidade é concedida para que os fabricantes de hardware possam otimizar o layout da memória conforme a arquitetura de suas GPUs. Dessa forma, o programa na CPU precisaria consultar, em tempo de execução, o deslocamento de cada membro para preencher o UBO de acordo com essa estrutura dinâmica.

2.3 Buffer Object

Dados como os vértices de um ator, seus atributos ou até os valores de blocos de uniform são armazenados e disponibilizados para o pipeline através de *Buffer Objects*,

blocos de memória gerenciados pelo OpenGL. Em vez de enviar os dados a cada quadro, a aplicação pode transferi-los para um *Buffer Object* uma única vez e instruir a GPU a utilizá-los repetidamente a partir de sua própria memória. O *Uniform Buffer Object* (UBO) é um tipo de buffer projetado para armazenar dados de blocos uniform, permitindo que um único conjunto de dados, como as matrizes de câmera, seja compartilhado de forma eficiente entre múltiplos programas de shader que declarem a mesma estrutura de bloco.[3]

A conexão entre um bloco de uniform de um shader e um UBO é mediada por um ponto de vinculação (*binding point*), que funciona como um intermediário no contexto do OpenGL. O vínculo é estabelecido em duas etapas: primeiro, o bloco de uniform dentro do programa de shader é associado a um ponto de vinculação; em seguida, o UBO que contém os dados é conectado a esse mesmo ponto. Uma vez que ambos estão apontando para o mesmo ponto de vinculação, a conexão entre eles está ativa.

Este mecanismo permite que múltiplos blocos de uniforms de diferentes programas de shader acessem o mesmo UBO, se associando ao mesmo ponto de vinculação.

A Figura 2 apresenta um exemplo, onde o bloco de uniform `Lights` tanto do shader A quanto do shader B e o UBO `uboLights` estão vinculados ao ponto 0. Os dois shaders acessam os mesmos dados no mesmo lugar, evitando duplicação e facilitando sua modificação, já que qualquer modificação no UBO se aplicará para ambos os shaders. Já seus blocos de uniform `Material` estão vinculados a pontos diferentes, logo usam UBOs diferentes.

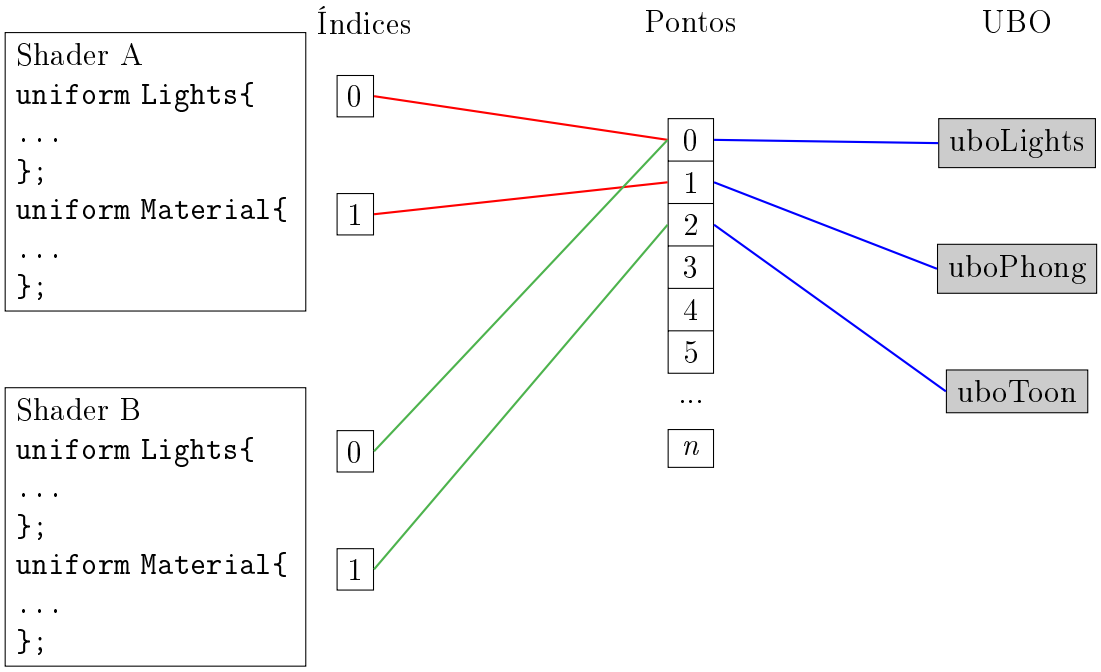


Figura 2: Exemplo de vínculo entre bloco de uniform e UBO

3 Visão Geral da Solução Proposta

A solução desenvolvida para que o usuários adicione seus próprios shaders segue os seguintes passos. Primeiro, o usuário escreve o shader com o efeito desejado que será aplicado a um material. Esta escrita consiste em criar um arquivo com a extensão

`.shlb` com as informações do shader, conforme exemplificado na Seção 4. Este arquivo contém as propriedades que ele deseja estarem expostas na interface gráfica, que serão os uniforms do shader, e o código em GLSL que aplica a tonalização desejada. Este arquivo deve estar dentro pasta `assets/shaders/` para ser lida pela aplicação. Sua estrutura é explicada na Seção 4.1 e a gramática que o arquivo deve seguir está no Anexo A.

Segundo, o arquivo é lido e qualquer erro durante sua leitura é mostrado para o usuário. As informações do shader são armazenadas em objetos `Shader` e `Property` (seus membros explicados na Seção 4.1). Um código de shader de fragmento base, com a declaração de versão do OpenGL, uniforms de câmera e luzes, e a função `main`, é combinado com as propriedades lidas, que se tornam uniforms, e o código GLSL do arquivo para formar o shader de fragmento. Este shader é compilado, com erros de compilação mostrados ao usuário, e então vinculado a um shader de vértice já existente para formar o programa de shader.

Assim, o shader está pronto para ser aplicado a um material. Na janela de inspeção do material, um novo campo com o nome `Shader` disponibiliza uma lista de shaders carregados dos arquivos para serem selecionados. Quando o shader é selecionado, suas propriedades definidas no arquivo ficam expostas na interface para o usuário modificá-las.

O próximo passo é utilizar o material com o shader assim definido para tonalizar algum ator da cena. Quando um ator é selecionado, é possível escolher seu material na interface gráfica arrastando um material existente na janela de assets para a propriedade de material. O shader do material e suas propriedades são expostas para serem modificadas. É possível notar que já temos disponíveis todas as informações necessárias para renderização, pois o ator contém um material, que contém um shader e os valores atuais de suas propriedades. No loop de renderização de cada ator, seu programa de shader é selecionado e seus uniforms são atualizados, renderizando-o assim com a lógica descrita no arquivo escrito pelo usuário.

4 Implementação

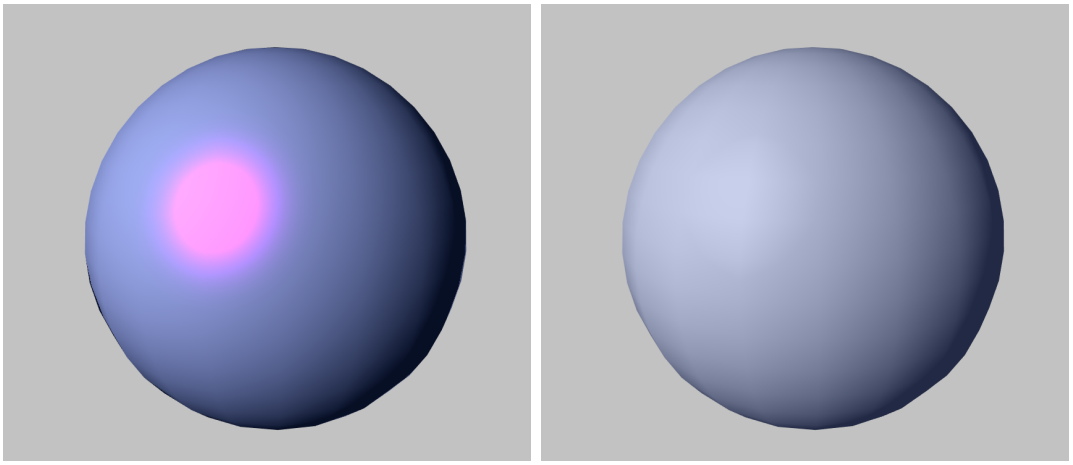
A implementação, assim como a Ds, é escrita em C++, usando OpenGL versão 4.0, com interface gráfica utilizando Dear ImGui e orientação a objetos na composição de suas classes.

Como um exemplo prático, vamos substituir o modelo de iluminação de Phong (explicado na Seção 5), que produz o resultado na Figura 3a, por um modelo feito pelo desenvolvedor, cujo resultado está na Figura 3b. O novo modelo utilizará uma simplificação do Oren-Nayar, um modelo de reflectância difusa que simula micro rugosidades na superfície, com o objetivo de simular corpos celestes.[4]

Primeiro, temos que definir quais as propriedades deste shader que queremos expor para o usuário, neste caso é a rugosidade (*roughness*) e o albedo, mais detalhados na Seção 5. Em seguida, devemos escrever o código em GLSL que aplica o efeito desejado, que neste exemplo usará apenas a primeira fonte de luz, a fim de simplificar o código. Os uniforms do código escrito devem se encaixar no modelo esperado pela aplicação, este é: os uniforms da câmera em um bloco de uniform chamado `camera`, das luzes em um bloco chamado `lights` e as propriedades do material em um bloco chamado `material`, mais detalhados na Seção 4.1.

Após definidos, estas informações são organizadas em um arquivo de texto que a aplicação sabe ler, com extensão `.shlb`.

Durante a inicialização da aplicação, este arquivo será lido. Caso não contenha erros, seu shader será compilado e disponibilizado para os materiais. Quando um material de um ator passar a utilizar o shader, suas propriedades serão disponibilizadas no inspetor, criado em tempo de execução.



(a) Phong (b) Oren-Nayar

Figura 3

4.1 Arquivo shlb

A adição de shaders personalizados é feita através de arquivos escritos pelo desenvolvedor. Estes arquivos, que devem ter a extensão `.shlb`, definem as propriedades que podem ser editadas na interface e o código do shader. Para serem reconhecidos, devem ser localizados no diretório `assets/shaders/` da aplicação.

A estrutura do arquivo é definida por uma linguagem própria, seguindo a gramática encontrada no Anexo A. É utilizada as chaves `{` e `}` como delimitadores para definir blocos de escopo, de forma similar à linguagem C. Cada arquivo deve começar com a palavra-chave *Shader*, seguida por seu nome, que deve ser único, e o corpo principal do shader delimitado por chaves.

Dentro do corpo do shader, existem dois blocos principais:

- **Property**: Este bloco é usado para declarar os atributos do material, que se tornarão `uniforms` no código GLSL e serão expostos na interface para o usuário as modificar. A sintaxe para cada propriedade é `tipo nome`, podendo ser declarado valores iniciais através de chaves e o intervalo limite dos valores com parênteses, nesta ordem. Os tipos permitidos são `int`, `float`, `vec2`, `vec3`, `vec4`, `color`, `mat3`, `mat4`.

  Alguns tipos requerem uma declaração especial de seus valores, o tipo `color` requer a palavra `rgb` seguido dos valores em parênteses, de 0 a 1, por exemplo `rgb(0.3, 0.7, 0.8)`. O mesmo é válido para `vec2`, `vec3` e `vec4`, que requerem a palavra `vec` em vez de rgb.

  Alguns exemplos de declarações de propriedades são: `float brilho {0.5} (0, 0.8)`, `color corBase (rgb(0.5, 0.5, 0.5))`. A variável `brilho` começa com

o valor 0.5, com limite inferior de 0 e superior de 0.8. A variável corBase tem valor inicial 0.5 em r, g e b, e limites padrões, no caso, 0 e 1.

- **Method**: Este bloco contém o código de renderização em si, escrito diretamente em GLSL. Apenas as funções devem estar neste bloco, pois as variáveis, tanto de entrada quanto uniforms, são padronizadas e serão adicionadas pela aplicação. É obrigatória a definição de um bloco Method com uma função chamada MainShader, sem parâmetros, com retorno do tipo `color`. Esta função é o ponto de entrada para o cálculo da cor final de um fragmento, sendo conceitualmente equivalente à função `main` de um Fragment Shader.

  Além da função MainShader, o desenvolvedor pode declarar outras funções auxiliares em outros blocos Method, seguindo a sintaxe padrão da GLSL, podendo usar quaisquer tipos e funções nativas do GLSL.

No exemplo prático, criaremos um arquivo chamado `OrenNayar.shlb` no diretório `assets/shaders/`, com o seguinte conteúdo:

```
Shader OrenNayar
{
 Property
 {
  float roughness {0.0}
  float albedo {0.2}
 }

 Method
 {
     color MainShader(){
            // GLSL...
     }
 }
}
```

Para o bloco `Method`, precisamos antes entender como estão declaradas os uniforms do material e das luzes. As propriedades declaradas serão embutidas em um código base já existente, que segue a seguinte estrutura:

- **Versão do OpenGL**: A diretiva `#version 400 core`.
- **Entradas padrão** (`in`): Variáveis que vêm do estágio anterior do pipeline, como `gPosition` (posição do fragmento no espaço do mundo), `gNormal` (vetor normal) e `gColor` (cor base do vértice).
- **Bloco de uniform** `camera`: Contém dados da câmera, como `projectionType` (perspectiva ou ortogonal) e `viewDir` (direção de visão).
- **Bloco de uniform** `lights`: Contém dados de iluminação da cena, incluindo `ambient` (cor da luz ambiente), `count` (número de luzes ativas) e um array de structs de luz (`props`). Cada struct descreve uma luz com suas propriedades: tipo (`type`), cor (`color`), posição (`position`), direção (`direction`), queda (`falloff`), alcance (`range`) e ângulo (`angle`).

- **Bloco de uniform** `material`: Caso existam propriedades definidas no bloco `Property`, elas farão parte de um bloco de uniform chamado `material`. Durante este processo, o tipo `color` é trocado por `vec4`.

Com isso, podemos escrever o código GLSL do exemplo do Oren-Nayar[1], acessando os uniforms necessários da forma correta. O arquivo completo, com suas propriedades e métodos se encontra no Anexo B.

Assim, o arquivo está pronto para ser lido pela aplicação e o shader pronto para ser aplicado a um material.

Para a leitura do arquivo, foi aproveitado o analisador léxico já existente em Ds, implementando apenas o analisador sintático, na classe `ShaderReader`, como um analisador descendente recursivo. Este analisador herda de `Reader`, uma classe de analisador genérica. Ela inclui macros para definir as palavras reservadas e mensagens de erro, sendo necessário apenas sobrescrever a função de iniciar (`void start()`) e implementar as funções que representam cada produção gramatical.

No início da leitura, é criado um objeto da classe `Shader`, onde é guardado seu nome e serão armazenados suas propriedades, em uma lista de objetos `Property`, e seus métodos, em uma lista de `string`. Dentro do bloco do shader, podem aparecer quaisquer quantidades de bloco `Property` e `Method`, sendo necessário um bloco `Method` com a função `color MainShader()`, que será o ponto de entrada do shader. Para cada propriedade lida, é verificada se seu tipo é válido, se seu nome é único para este shader e quaisquer valores padrões (entre chaves) e limite (entre parênteses). Essas informações são guardadas dentro de um objeto `Property`, que também contem seu offset no UBO do material (explicado na criação do material).

Para cada método, uma verificação similar ocorre. O nome deve ser único, com tipos válidos tanto de retorno quanto das propriedades. A declaração e corpo da função são guardadas em uma `string`.

Uma vez que o arquivo `.shlb` é completamente analisado, o passo seguinte é a geração dinâmica de um código-fonte GLSL completo para um shader de fragmento, seguindo o modelo base anteriormente citado. Os métodos lidos do bloco `Method`, incluindo as funções auxiliares e a função `MainShader`, são inseridos no código após as declarações dos bloco de uniforms.

Ao final deste processo de montagem, o texto completo do código GLSL gerado é passado para a API do OpenGL, onde é compilado como um shader de fragmento. Subsequentemente, ele é vinculado para criar um Programa de Shader completo e é armazenado em um `map` chamado `ShaderAssets`. Após a criação do programa de shader, são adquiridos do OpenGL os offsets de cada propriedade do bloco de uniform `material`, armazenados no objeto `Property` de cada propriedade.

Enquanto o objeto `Shader` representa a lógica de renderização abstrata e suas propriedades configuráveis, o `MyMaterial` representa uma instância concreta e única de um shader. Em termos práticos, um material é um contêiner de dados que armazena os valores específicos para as propriedades de um shader. Isso permite, por exemplo, que múltiplos objetos na cena utilizem o mesmo shader de Phong, mas cada um com um coeficiente de reflexão especular e brilho diferentes, pois cada um teria seu próprio material.

[1] Código modificado de um shader público, com licença MIT, acessível em `https://github.com/glslify/glsl-diffuse-oren-nayar`

Um objeto `MyMaterial` mantém uma referência ao `Shader` que ele instancia e gerencia os dados de suas propriedades em duas frentes:

- **Buffer de dados na CPU**: Internamente, o material aloca um bloco de memória contíguo na RAM na forma de um ponteiro para `void` para armazenar os valores atuais de todas as propriedades definidas no shader associado. É este buffer que a interface gráfica, descrita na Seção 4.2, lê e modifica diretamente, permitindo que as alterações do usuário sejam registradas.
- **UBO na GPU**: Para cada material, é criado um objeto `GLUniformBuffer`, classe que abstrai as chamadas do OpenGL de gerenciamento do UBO. Este UBO é projetado para corresponder exatamente à estrutura do bloco de uniform `material` gerado dinamicamente.

O ciclo de vida e a operação de um material são governados por processos chave. Quando um material é criado ou quando seu shader é alterado dinamicamente através da interface, o material se reconfigura: ele aloca novos buffers de CPU e GPU com o tamanho correto para as propriedades do novo shader e os inicializa com os valores padrão definidos no arquivo `.shlb`. Com isso, os valores e tipos das propriedades do material estão prontas para serem lidas e mostradas na interface.

A Figura 4 apresenta um diagrama de classes UML de como é organizado a relação entre ator, material e shader.

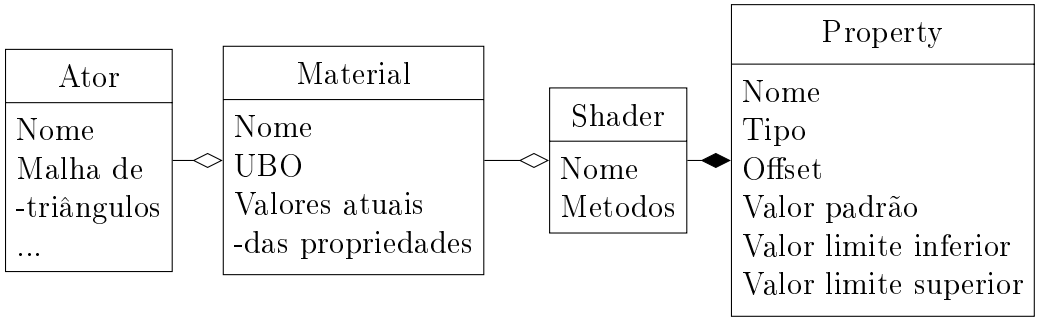


Figura 4: Diagrama UML

4.2 Interface Gráfica

Ao selecionar um ator da cena, as propriedades de seu material são expostas ao usuário através de um inspetor de materiais na interface gráfica da aplicação, utilizando a biblioteca ImGui. Esta interface é gerada dinamicamente e permite a manipulação dos atributos de um material em tempo real. Além disso, a interface oferece uma funcionalidade de arrastar e soltar (*drag and drop*), na qual o usuário pode atribuir um novo material ao ator arrastando-o de uma lista de recursos do projeto para um campo designado no inspetor.

O inspetor de materiais apresenta duas funcionalidades principais. A primeira é uma lista suspensa (*combo box*) que exibe todos os shaders carregados pela aplicação. Através desta lista, o usuário pode alterar dinamicamente o shader associado a um material. Ao selecionar um novo shader, a interface do inspetor é imediatamente atualizada para refletir as propriedades específicas do novo shader escolhido.

A segunda e principal funcionalidade é a geração automática de controles de interface para cada uma das propriedades definidas no bloco `Property` do arquivo `.shlb` correspondente. O sistema percorre a lista de propriedades do shader e, com base no tipo de cada uma, instancia um controle de interface apropriado. A correspondência entre os tipos de propriedade e os componentes da interface é a seguinte:

- Tipos numéricos como `int` e `float` são representados por controles deslizantes (*sliders*) para um único valor.
- Tipos vetoriais como `vec2` e `vec3` são mapeados para *sliders* multi-componentes.
- O tipo `color` é representado por um seletor de cores interativo (*color picker*).
- Tipos mais complexos, como matrizes (`mat3`, `mat4`), são decompostos em múltiplos *sliders* vetoriais, permitindo a edição de cada uma de suas linhas ou colunas.

Adicionalmente, os intervalos de valores, definidos opcionalmente entre parênteses na declaração da propriedade no arquivo `.shlb`, são utilizados para configurar os limites mínimo e máximo dos controles deslizantes na interface.

No exemplo usado anteriormente de Oren-Nayar, as propriedades `roughness` e `albedo` se tornam *sliders* na interface, como mostra a Figura 5, na janela da direita, destacada na Figura 6.

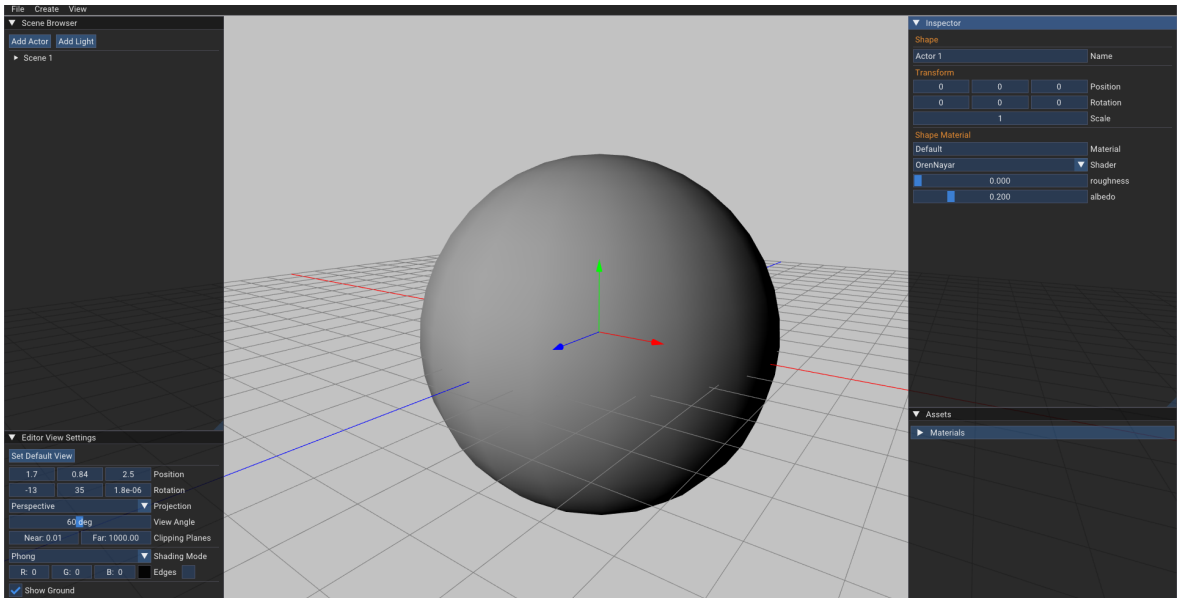


Figura 5: Janela da aplicação com um ator selecionado

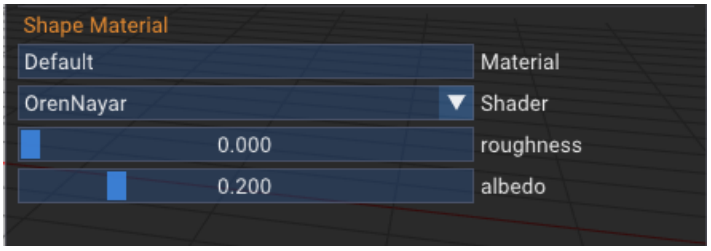


Figura 6: Inspetor de material do shader Oren-Nayar

Após manipulação dos dados do material na CPU na interface gráfica, estes dados precisam ser enviados para o pipeline, para que possa renderizar o ator com o shader escolhido e seus uniforms atualizados.

4.3 Renderização

A API Ds contém uma série de classes para o processo de renderização, como mostrado na Figura 7, que termina na classe `SceneEditor`. Dessas, a classe `GLRenderer` contém a lógica de renderização a ser alterada, nas funções `update()`, `beginRender()`, `renderLights()`, `renderActors()` e `endRender()`, que são chamadas nesta ordem dentro da função `render()`. Esta classe é responsável por configurar o estado global do OpenGL, preparar os dados da cena para a GPU e iterar sobre os objetos (atores) para emitir os comandos de desenho necessários.

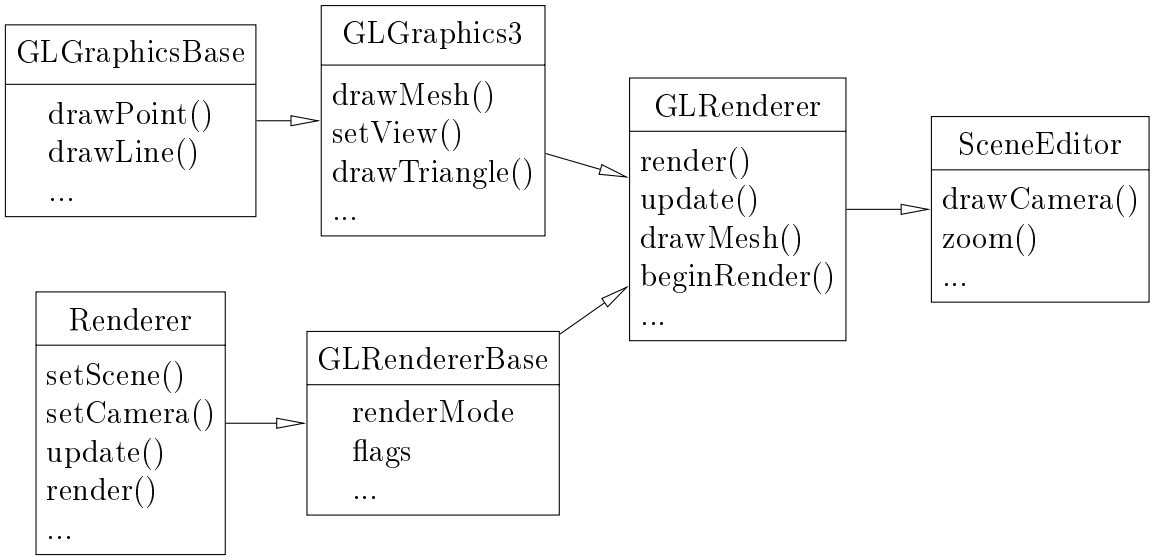


Figura 7: UML das classes para renderização

As funções de renderização foram herdadas (com exceção de `update()`, que foi substituida por `MyUpdate()`) pela nova classe `MySceneEditor`, que herda `SceneEditor`. A mudança consiste em mudar o acesso aos uniforms das luzes e câmera para utilizar UBOs e selecionar o programa de shader apenas no loop dos atores, consultando o shader que seu material utiliza.

Para gerenciar o estado e os recursos da API gráfica de forma organizada, a classe `SceneEditor` contém uma estrutura interna privada `GLData`. Esta estrutura contém as localizações dos uniforms das luzes, da câmera e do material. Na classe `MySceneEditor`, esta estrutura foi substituída por `MyGLData`, que encapsula os UBOs da luz e da câmera, os offsets de memória para cada membro dos bloco de uniforms e as localizações dos uniformes tradicionais.

Na construção do `GLData`, o programa de shader com o modelo se phong é criado e selecionado, e as localizações dos uniforms são adquiridas. Para o `MyGLData`, o construtor executa os seguintes passos:

1. Instancia a estrutura `MyGLData`.
2. Cria e aloca memória para os dois UBOs principais que serão compartilhados por todos os shaders: um para os dados da câmera (`cameraBufferObj`) e outro para

os dados de iluminação (`lightsBufferObj`). Estes buffers são imediatamente vinculados aos seus respectivos pontos de vinculação.

3. Utilizando o programa de shader padrão, o construtor consulta a API uma única vez para obter e armazenar em `MyGLData` todos os offsets de memória e as localizações dos uniformes que serão necessários durante o loop de renderização, que são comuns a todos os programas de shader (como `projectionTypeOffset`, `lightCountOffset`, `mvMatrixLoc`, etc.).

O processo de renderização na classe `MySceneEditor` segue as mesmas fases do `SceneEditor`, executadas em sequência a cada quadro.

1. **Inicialização do Frame e Configuração Global**: No início de cada quadro, a fase `beginRender` é executada. Esta etapa prepara o ambiente para o desenho: o buffer de cor e o de profundidade são limpos e os dados da câmera, como seu tipo de projeção e direção de visão, são atualizados no bloco de uniform `camera` em vez das uniforms diretamente. Este buffer é configurado apenas uma vez por quadro, pois a perspectiva da câmera é a mesma para todos os objetos renderizados naquela cena.

2. **Processamento das Luzes**: Em seguida, a fase `renderLights` prepara todas as informações de iluminação. Assim como a câmera, agora é utilizado um UBO para as luzes. O sistema itera sobre as fontes de luz presentes na cena e popula o bloco de uniform `lights` com seus dados. A cor da luz ambiente da cena e o número total de luzes ativas também são atualizados neste UBO. Assim como a câmera, os dados de iluminação são enviados uma única vez por quadro.

3. **Renderização dos Atores**: A fase central do processo é a `renderActors`, onde cada objeto visível da cena é desenhado individualmente. No `SceneEditor`, o programa itera sobre a lista de atores, envia seus dados específicos, como os vértices de sua malha e matrizes de transformação, e desenha. Na nova implementação, no `MySceneEditor`, a sequência de passos é a seguinte:

   (a) O programa de shader do material do ator é ativado.

   (b) O bloco de uniform do shader é vinculado ao ponto de vinculação do UBO do material, que contem os valores de suas propriedades (como cor, brilho, etc.). Os dados deste UBO são sincronizados com os dados do buffer em CPU do objeto `MyMaterial` do ator.

   (c) Os dados específicos do ator são enviadas ao shader.

   (d) Finalmente, com o shader ativo e todos os dados configurados, o comando de desenhar é emitido.

Assim, o ator com o material que utiliza o shader de Oren-Nayar é renderizado como mostra a Figura 3b, alcançando o objetivo do exemplo.

Após a iteração por todos os atores, o quadro está completo, renderizando uma imagem que pode conter múltiplos atores com materiais e shaders distintos.

5 Resultados

Nesta seção, são apresentados os resultados visuais obtidos com a implementação do sistema de shaders personalizados. Os resultados são imagens demonstrando a renderização de shaders distintos, controlados por uma interface gráfica gerada dinamicamente. Cada shader tem duas figuras apresentando a interface gráfica, uma com os valores das propriedades padrões de quando o shader é selecionado, e outro com seus valores alterados. Os modelos de iluminação escolhidos de exemplo foram: *Phong*, *Oren-Nayer* e *Toon*.

O primeiro, Phong, é um modelo de reflexão empírico que estende o modelo Lambertiano (que segue o princípio de que a luz que atinge um ponto na superfície é espalhada com igual intensidade em todas as direções) para simular não apenas superfícies foscas, mas também superfícies com brilho (especularidade).[5] Sua principal contribuição é a aproximação dos reflexos especulares, os destaques brilhantes que aparecem em materiais polidos como plástico ou metal. Para alcançar um resultado mais realista, o modelo de Phong decompõe a iluminação de um ponto em três componentes distintos, que são calculados separadamente e depois somados.

Na implementação, o shader contém quatro propriedades:

- **Ambient**: Este componente simula a cor refletida pela iluminação indireta e global, ou seja, a luz que foi refletida por outras superfícies na cena e atinge um objeto de todas as direções. Ele não depende da posição da fonte de luz nem do observador.
- **Diffuse**: Este componente modela a cor da reflexão de luz em superfícies foscas. Ele assume que a luz que atinge a superfície é espalhada uniformemente em todas as direções. A intensidade da luz refletida depende do ângulo entre a normal da superfície e a direção da fonte de luz. Na Figura 8, o vetor normal é $\mathbf{N}$ e o vetor de direção da luz é $\mathbf{L}_l$.
- **Spot**: Este é o componente que define a cor do brilho do modelo de Phong. Ele simula os reflexos direcionais que ocorrem em superfícies lisas e polidas. A premissa é que a reflexão especular é mais intensa quando o ângulo entre o vetor de visão do observador e o vetor de reflexão da luz é pequeno. Na Figura 9, o vetor de reflexão é $\mathbf{R}_l$ e o vetor do observador é $\mathbf{V}$.
- **Shine**: É o expoente de brilho que controla sua intensidade, determinando o quão focado ou espalhado é o reflexo. Valores altos produzem um destaque pequeno e nítido (superfícies muito polidas), enquanto valores baixos produzem um destaque maior e mais suave.

Sua implementação é uma adaptação do shader fixo de Phong que existia na biblioteca de classes Ds, uma extensão alterando seus uniforms, que devem ser chamados com o prefixo do bloco de uniform, e estruturado para o tipo de arquivo `.shlb`, sem alterar sua lógica de iluminação. Seu resultado é mostrado nas Figuras 10 e 11.

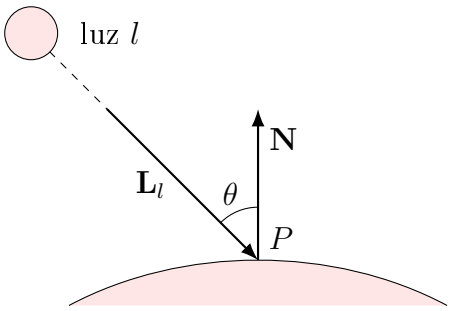


Figura 8: Reflexão difusa

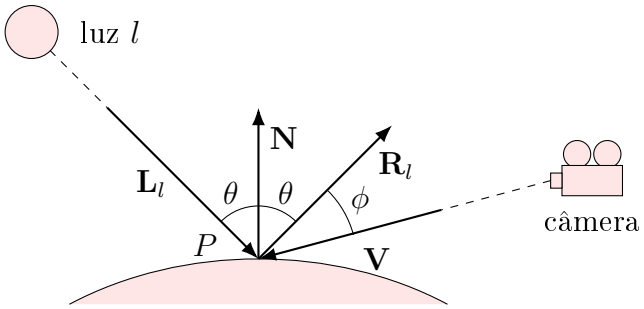


Figura 9: Reflexão especular

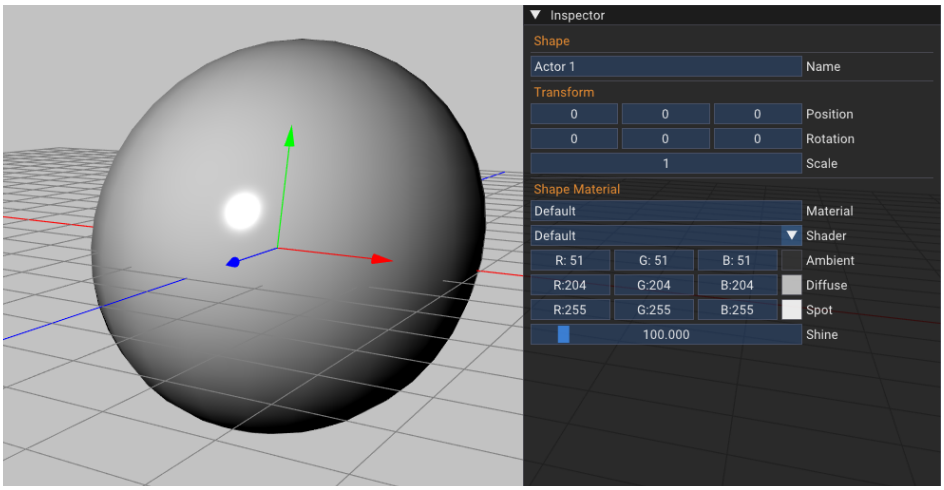


Figura 10: Phong padrão

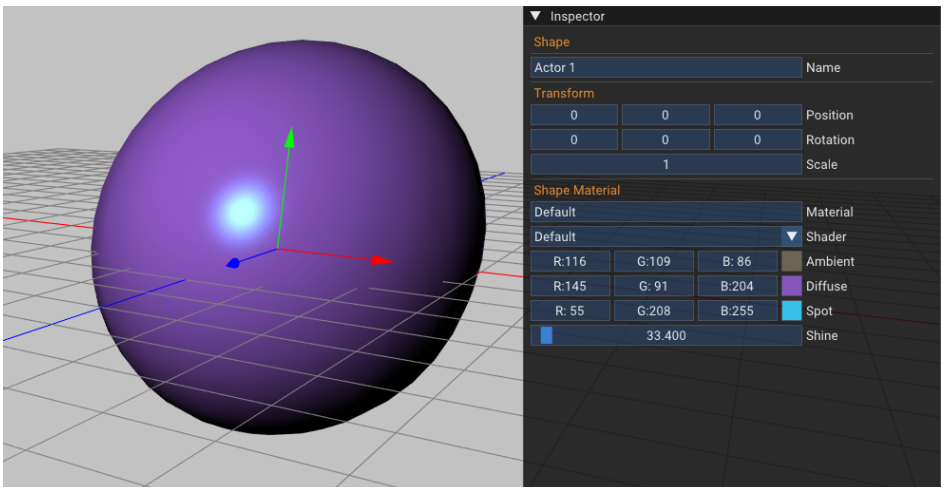


Figura 11: Phong modificado

O segundo exemplo, visto nas Figuras 12 e 13 apresentam o resultado do shader Oren-Nayar, um modelo de iluminação que descreve a reflectância de superfícies difusas (foscas) de maneira mais realista que o modelo Lambertiano tradicional. O modelo Oren-Nayar leva em consideração o fenômeno de micro rugosidade da superfície, tratando-a como uma coleção de microfacetas orientadas aleatoriamente.[4]

Ao contrário da reflexão Lambertiana, que é perfeitamente uniforme e independente da direção do observador, o modelo Oren-Nayar é *view-dependent*. Ele prevê que superfícies mais rugosas tendem a refletir a luz de forma mais uniforme em todas as direções, fazendo com que pareçam mais brilhantes quando vistas de ângulos oblíquos. Este efeito simula a aparência de materiais como argila, gesso, areia ou tecidos foscos. O shader implementado expõe duas propriedades para controlar este comportamento:

- **Albedo**: Esta propriedade representa a refletividade intrínseca da superfície, ou seja, a fração da luz incidente que é refletida. Em termos práticos, o albedo funciona como a cor base do material. Um valor de 0.0 resulta em uma superfície preta que não reflete luz, enquanto um valor próximo de 1.0 resulta em uma superfície branca altamente refletiva.

- **Roughness**: Este é o parâmetro central do modelo e controla o grau de micro rugosidade da superfície. Um valor de 0.0 faz com que o modelo se comporte de forma idêntica a um refletor Lambertiano ideal, resultando em uma aparência perfeitamente fosca e lisa. À medida que o valor de rugosidade aumenta, a superfície simulada se torna mais irregular em um nível microscópico. Isso faz com que a luz seja mais espalhada, reduzindo o contraste direcional e tornando a iluminação mais achatada e uniforme, o que é característico de superfícies porosas ou pulverulentas.

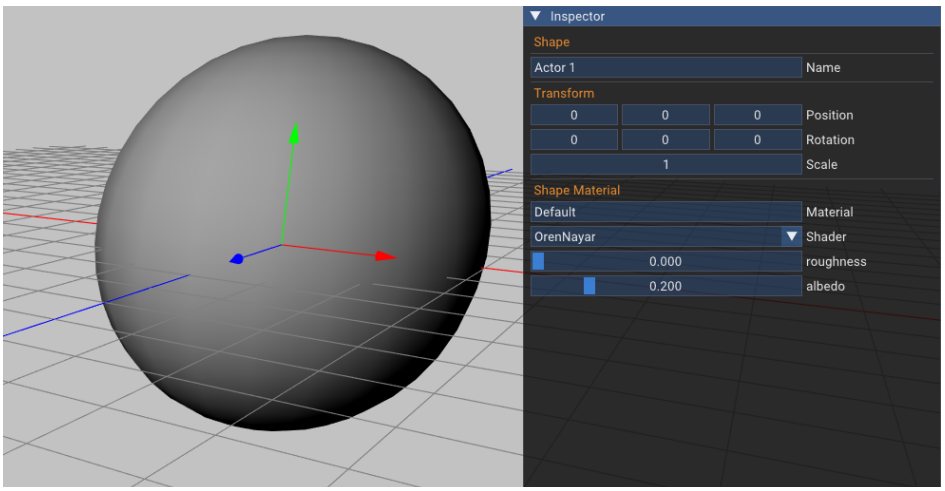


Figura 12: Oren padrão

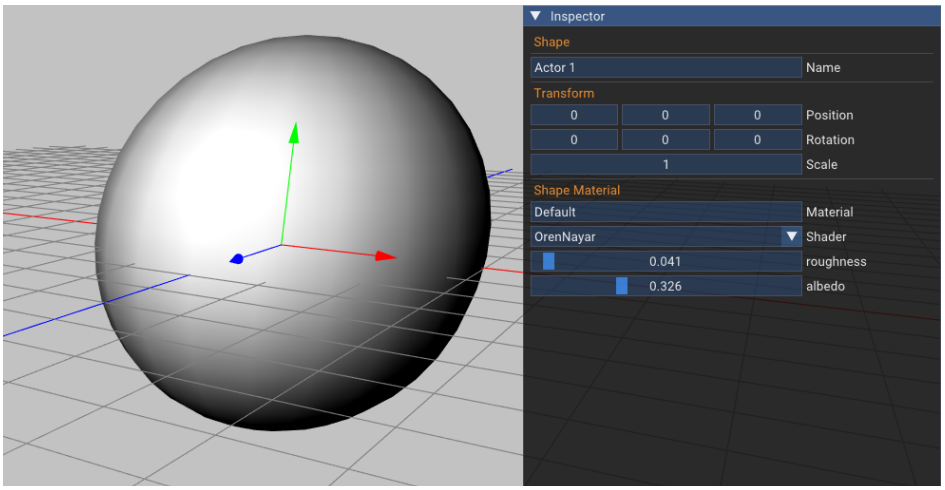


Figura 13: Oren modificado

As Figuras 14 e 15 apresentam o shader Toon, um modelo de iluminação não fotorrealista que busca emular o estilo visual de desenhos animados. Diferente de modelos como o de Phong, que produzem um gradiente de iluminação suave, o Toon Shading quantiza a luz em um número discreto de bandas de cor. O resultado é uma transição abrupta entre a área de luz e a de sombra, criando o característico efeito *cel shading*. Este efeito é alcançado ao comparar o produto escalar entre a normal da superfície e a direção da luz contra um conjunto de limiares (*thresholds*). As propriedades deste shader permitem que o usuário controle tanto a intensidade de cada banda de iluminação quanto os limiares que definem as fronteiras entre elas:

- **Full light intensity**: Define a intensidade da luz na região mais clara do objeto, correspondente à área de luz plena. Seu valor mínimo é limitado pela `Partial Light Intensity` para manter a consistência visual.
- **Partial light intensity**: Controla a intensidade da luz na região de meio-tom ou luz parcial. Este valor representa a segunda banda de iluminação e não pode ser maior que a `Full Light Intensity`.

- **Full light threshold**: Define o limiar de iluminação para a transição para a luz plena. Especificamente, ele representa o valor mínimo do produto escalar entre a normal da superfície e a direção da luz para que um fragmento seja considerado totalmente iluminado. Valores mais altos resultam em uma área de luz plena menor.
- **Partial light threshold**: Define o limiar para a transição para a luz parcial. Um fragmento cuja iluminação (produto escalar) esteja acima deste valor, mas abaixo do `Full Light Threshold`, receberá a intensidade definida por `Partial Light Intensity`. Este valor deve ser menor que o `Full Light Threshold` para definir corretamente a banda de meio-tom.

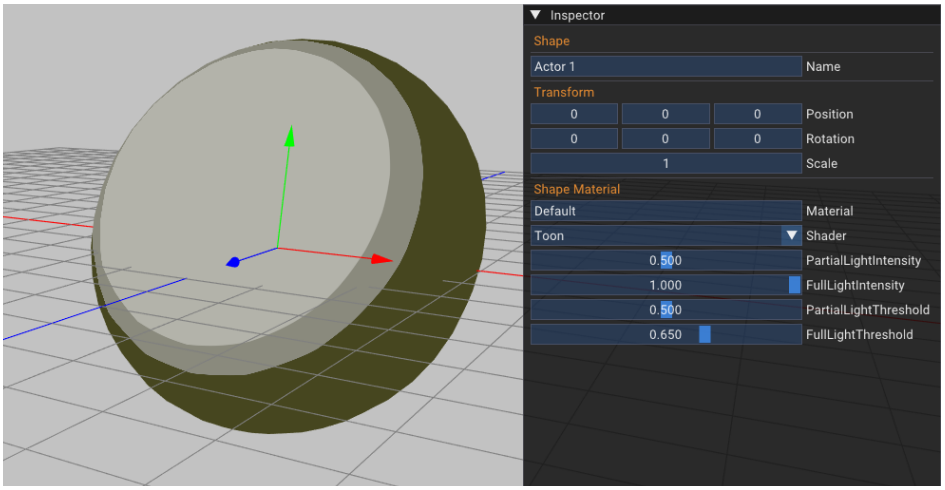


Figura 14: Toon padrão

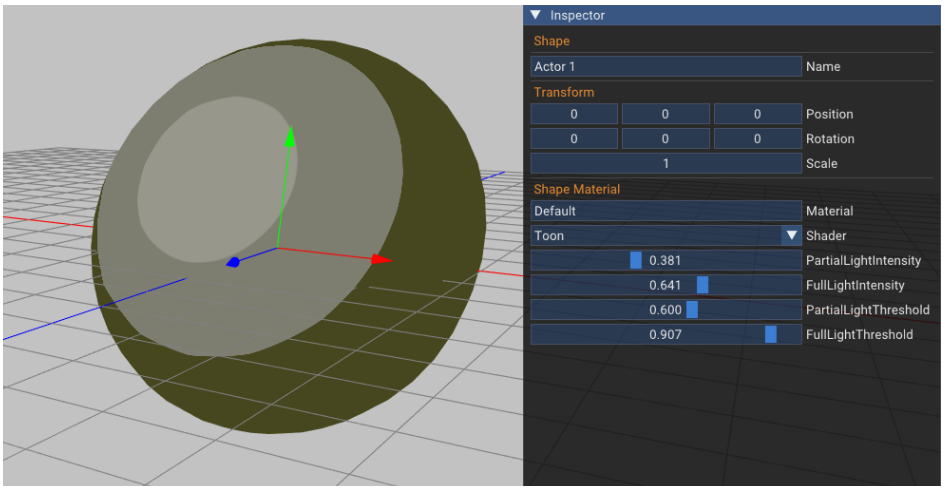


Figura 15: Toon modificado

A Figura 16 mostram em uma mesma cena, três atores com shaders diferentes, portanto, com materiais diferentes, todos renderizados em uma mesma imagem. As Figura 17 mostram vários atores, alguns com o mesmo material, logo utilizando o mesmo shader com os mesmos valores das propriedades.

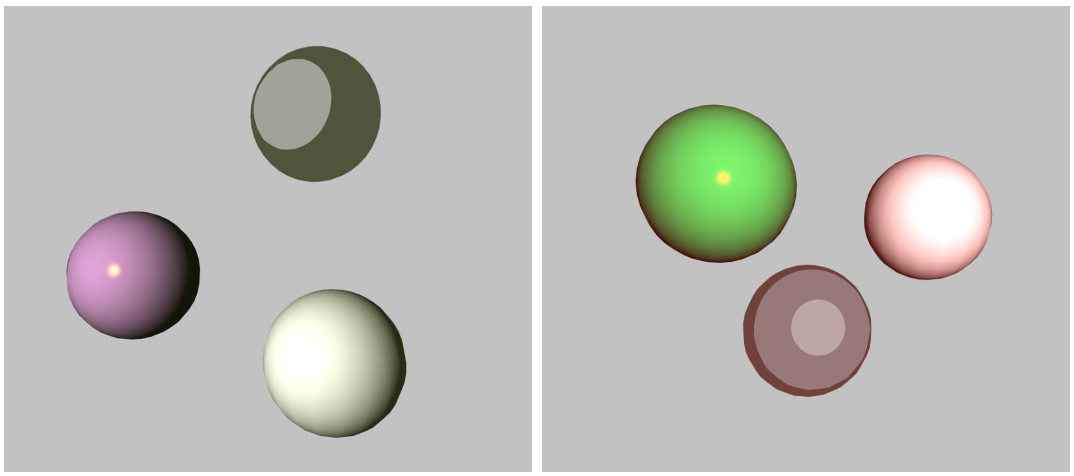


Figura 16: Cena com os três shaders diferentes

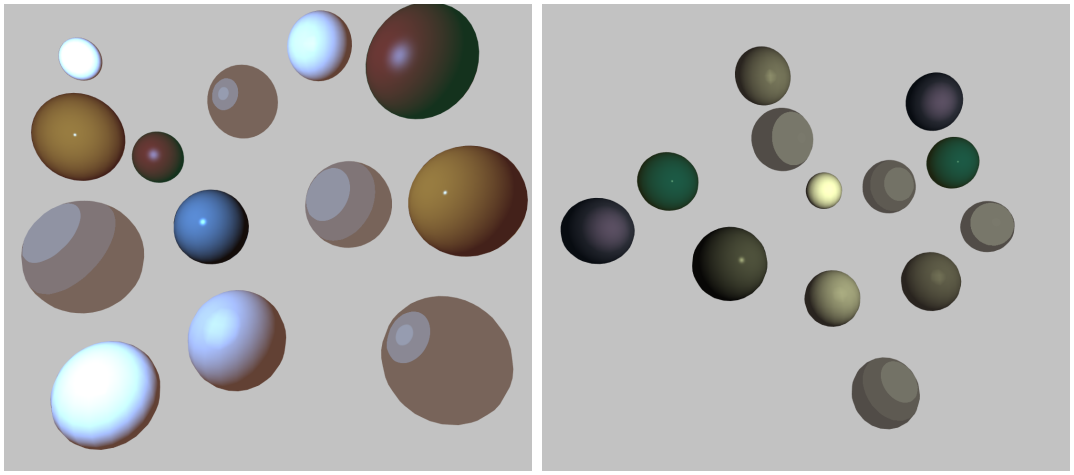


Figura 17: Cena com múltiplos materiais compartilhados

6 Conclusão

Este trabalho partiu da identificação de uma limitação na biblioteca gráfica Ds: sua arquitetura de material, que estava restrita a um único modelo de iluminação de Phong com propriedades fixas. Esta rigidez impedia a exploração de diferentes técnicas de sombreamento, limitando o potencial da biblioteca como ferramenta de aprendizado e prototipagem em computação gráfica. O objetivo central, portanto, foi desenvolver um sistema flexível e extensível que permitisse a criação, utilização e manipulação de múltiplos shaders e materiais personalizados.

Diante do proposto para o trabalho, todos os objetivos específicos foram alcançados, como visto nos resultados apresentados. Dentre as dificuldades enfrentadas no desenvolvimento deste trabalho, destacam-se as relacionadas à integração da arquitetura da Ds para garantir que as extensões propostas mantivessem a consistência de seu design. Em segundo lugar, as dificuldades inerentes à depuração com a API do OpenGL, tanto na identificação de erros em shaders GLSL quanto na resolução de problemas de renderização que não fornecem muitas ferramentas ou feedback para o desenvolvedor.

Existem diversas oportunidades para expandir ainda mais as capacidades da biblioteca. Atualmente, nem todos os tipos de variáveis do OpenGL são aceitos, limitando a escrita de shaders. Além disso, são possíveis otimizações tanto na renderização, como agrupar os atores com o mesmo material para reduzir as chamadas ao OpenGL, quanto no carregamento, armazenando os shaders compilados a fim de evitar a análise de arquivos `.shlb` que não foram modificados. Por fim, outra melhoria é a integração à renderização por traçado de raios, permitindo que um mesmo arquivo `.shlb` seja utilizado para as duas pipelines.

A Gramática do Arquivo .shlb

```
CompilationUnit:
 ShaderList?

ShaderList:
 Shader*

Shader:
 SHADER NAME ShaderBody

ShaderBody:
 "{" ShaderMember+ "}"

ShaderMember:
 PropertyBlock
| MethodBlock

PropertyBlock:
 PROPERTY "{" PropertyList "}"

PropertyList:
 Property*

Property:
 Declaration Range?

Declaration:
 TYPE NAME Initializer?

Initialiazer:
 "{" ExpressionList? "}"

Range:
 "(" Expression ("," Expression)? ")"

ExpressionList:
 Expression ("," Expression )*
```

```
MethodBlock:
 METHOD "{" Method "}"

Method:
 (TYPE|VOID) NAME "(" ParamList? ")" MethodBody

ParamList:
 Param ("," Param)*

Param:
 TYPE NAME

MethodBody:
 "{" char* "}"

Expression:
 Term (("+"|"-") Term)*

Term:
 Factor(("*"|"/"|"%") Factor)*
Fator:
 ("+"|"-")? PrimaryExpression

PrimaryExpression:
"(" Expression ")"
| NAME ("(" ExpressionList ")")?
| Literal

Literal:
 INT_LITERAL
| FLOAT_LITERAL
```

B Arquivo pbr.shlb

```
1  Shader OrenNayar
2  {
3    Property
4    {
5      float roughness {0.0}
6      float albedo {0.2}
7    }
8
9    Method
10   {
11     color MainShader(){
12       const float pi = 3.14159265f;
13
```

```
14          float LdotV = dot(lights.props[0].position -
               gPosition, gPosition);
15          float NdotL = dot(lights.props[0].position -
               gPosition, gNormal);
16          float NdotV = dot(gNormal, gPosition);
17
18          float s = max(0.0, LdotV - NdotL * NdotV);
19          float t = mix(1.0, max(NdotL, NdotV), step(0.0, s));
20
21          float sigma2 = material.roughness * material.
               roughness;
22          float A1 = material.albedo / (sigma2 + 0.13);
23          float A2 = 0.5 / (sigma2 + 0.33);
24          float A = 1.0 + sigma2 * (A1 + A2);
25          float B = 0.45 * sigma2 / (sigma2 + 0.09);
26          float C = (A + B * s / t) / pi;
27
28          float brightness = material.albedo * max(0.0, NdotL)
               * C;
29
30          vec3 shine = brightness * lights.props[0].color.xyz;
31
32          vec4 color = lights.ambient + vec4(shine, 1.0);
33
34          return color;
35      }
36    }
37  }
```

Referências

[1] O. Cornut, *Dear imgui: Bloat-free graphical user interface for c++ with minimal dependencies*. https://github.com/ocornut/imgui, 2025.

[2] T. K. G. Inc, *Arb_shader_storage_buffer_object*. https://registry.khronos.org/OpenGL/extensions/ARB/ARB_shader_storage_buffer_object.txt, 2012.

[3] ——, *Arb_uniform_buffer_object*. https://registry.khronos.org/OpenGL/extensions/ARB/ARB_uniform_buffer_object.txt, 2015.

[4] M. Oren and S. K. Nayar, *Generalization of lambert's reflectance model*, in Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH '94, New York, New York, USA, 1994, ACM Press, pp. 239–246.

[5] B. T. Phong, *Illumination for computer generated pictures*, 1975.