# Wrapper: Shader

Este descreve os recursos da classe Shader como wrapper de recursos OpenGL para criação e vinculo de sombreadores.

Dependências:

São dependências existências para classe Shader os seguintes cabeçalhos:

* vec3.hpp (Uma classe que descreve vetores e operações básicas de vetores)
* Light.hpp (Uma classe que encapsula atributos de interação de uma pseudo luz)
* Material.hpp (Uma classe que armazena atributos de interação de um pseudo objeto)

Estruturas:

A classe Shader usa as seguintes estruturas definidas no corpo do seu cabeçalho e no próprio escopo para fins de encapsulamento de dados:

* Uniform3f: Encapsula um nome uniforme e um ponteiro para uma matriz de floats.
* Uniform1f: Encapsula um nome uniforme e um valor do tipo float.
* Uniform1i: Encapsula um nome uniforme e um valor inteiro.
* Texture: Uma estrutura para encapsular os dados de uma textura tais como um buffer, dimensões, canais e id de textura OpenGL gerado pelas funções membro. Atualmente a estrutura possui duas funções: uma gera texturas para o alvo GL\_TEXTURE\_2D e a outra para GL\_TEXTURE\_CUBE\_MAP\_\*. Essa estrutura depende da biblioteca externa stbi para suas operações de carregamento de imagens.

## Definição da classe Shader

A classe possui dois construtores: o padrão é definido com o propósito de permitir que a classe seja agregada a outras classes sem necessitar usar uma lista de inicialização na classe que a agrega. Já o sobrecarregado leva os caminho para o sombreador de vértices e de fragmento.

A função create é usada internamente recebendo o caminho dos sombreadores diretamente do sombreador sobrecarregado, identificando a natureza do programa de sombreador (se leva texturas ou não) e por fim chamando funções globais responsaveis pela criação real dos sombreadores e vinculação do programa de sombreamento.

## Atributos

samplers: Um vetor da estrutura Uniform1i para inserir dinâmicamente o nome de amostradores e seu número identificador. setUniform1i é seu setter e leva dois parâmetros: nome e valor.

vectorUnif1f: Um vetor da estrutura Uniform1f para inserir dinâmicamente um nome uniform de tipo float. setUniform1f é seu setter eleva dois parâmetros: nome e valor.

vectorUnif3f: Um vetor da estrutura Uniform3f para inserir dinâmicamente um nome e valor de um uniform do tipo vec3. Nesse o parâmetro de valor deve ser um endereço de uma matriz de floats.

textureID: Um vetor para armazenar identificadores de texturas inseridos por chamadas a função add\* .

light: Um ponteiro de vetor para armazenar uma coleção de instâncias da classe Light.

material: Um ponteiro para uma instância de Material.

dirLightIndex: um inteiro assinado de 2 bytes para guardar o índice de uma luz no vetor de ponteiros light.

is\_cubmap: um sinalizador usado para dizer como determinadas funções devem usar as texturas armazenadas pela instância – no momento há apenas dois tipos de texturas sendo tratadas.

fragmentShaderPath: um tipo string para armazenar o caminho do sombreador de fragmento. É util para algumas funcionalidades que apresentam polimorfismo dado um tipo de programa de sombreamente vinculado.

program: Um identificador para um programa de sombreamento vinculado.

## Funções membro

As funções setUniform3f, setUniform1f, setUniform1, setMaterial, setDirectionalLight e setLights seguem o comportamento natural dos setters.

addTexture: leva uma caminho de textura, gera uma instância temporária de Texture e usa uma função membro de Texture para gerar um id de textura e o insere no vetor textureID.

addTextureFromFramebuffer: No momento, gerar framebuffers é um tarefa externa que gera um id de textura que deve ser passado como parametro para essa função. Tal como em addTexture, o parâmetro de passagem sera inserido no vetor textureID.

addCubeMap: leva um vetor de caminhos (apenas 6 caminhos serão usados), instância temporariamente um vetor de objetos da classe Texture, redimensiona-o para conter a quantidade de samplers definidos por setUniform1i – dado tal, essa e todas funções de adicinamento de texturas só fazem sentido após ao menos uma chamada de setUniform1i, idelamente para cada chamada setUnifrom1i, há uma chamada a função add\* - e então gera n ids de texturas e os insere no vetor textureID e por fim define o sinalizador is\_cubemap para verdadeiro.

renderLights: Essa função faz chamadas a glUniform\* alimentadas pelo vetor de ponteiros light.

setVertexAttribPointer: Essa função presume que o sombreador de vértices básico sempre estará em uso para poder obter as localizações de atributos de entrada position, normal e uvcoord, comuns a todos objetos padrões. Conforme, define o formato dos dados provenientes de um VAO e os respectivos deslocamentos lógicos no VBO vinculado ao VAO. Note que essa é uma função destinada a ser usada após a definição de um VAO completo.

initUniforms: Essa função deve ser chamada antes de um chamada glDraw\* e após glBindVertexArray. Ela é responsavel por definir as unidades de texturas ativas e os identificadores a serem vinculados ao destino GL\_TEXTURE\* e inicializar os respectivos amostradores uniformes, uniformes vec3 e float. Por fim, chama renderLights.

operator=: A sobrecarga do operador de atribuição foi definido para que fosse possivel copiar os dados de incialização de uniformes para um id diferente de programa de sombreamento já criado.

free: uma função que desaloca os recursos armazenados pela instância.

# Wrapper: OBJ

Define funcionalidades comuns a objetos tridimensionais tais como: interpretador de formato .obj, matriz de modelo e espaço de visualização, espaço de luz e a posição do observador. Conta com uma instância de Shader agregada, criação de VAOs, VBOs, UBOs e IBOs, funções para configurar e renderizar – renderização instânciada e não instânciada - e desalocador de recursos.

## Atributos

currentShader: o shader interno usado por instâncias da classe.

name: a string que armazena o nome do objeto.

vertexStream: o vetor que armazena o fluxo de vértices – posição, normal e coordenadas de textura.

indexStream: o vetor que armazena o fluxo de índices.

vertexBytes: o membro size\_t que armazena a quantidade de bytes ocupada pelo vetor vertexStream.

indexBytes: o membro size\_t que armazena a quantidade de bytes ocupada pelo vetor indexStream.

vertexArrayObject: o identificador para o VAO em questão.

vertexBufferObject: o identificador para o VBO em questão.

indexBufferObject: o identificador para o IBO em questão.

model: um ponteiro de tipo mat4 para a matriz de modelo.

view: um ponteiro de tipo mat4 para a matriz de visualização.

lightSpace: um ponteiro de tipo mat4 para a matriz de espaço de luz.

camera: um ponteiro de tipo vec3f para a posição do observador.

to\_instance: o shader iterno usado pelas insâncias da classe para renderização instânciada. Atualmente esse shader apenas é especializado em renderização instânciada por meio de UBOs, futuramente espera-se que se possa usar a renderização instânciada com base em arrays de uniformes para aproveitar a vantagem de não armazenar um UBO para renderizações de poucas cópias.

ubo: o identificador para o UBO em questão.

instance\_count: o número de instâncias pretendidas para renderização instanciada.

## Funções membro

A classe possui um construtor sobrecarregado que recebe o caminho de um arquivo .obj, e o caminho para os sombreadores de vértice e fragmentos desejados.

parserOBJ: a função responsavel por interpretar o arquivo .obj passado para o construtor da classe. Na versão atual, os dados são salvos no formato de desenho matricial, portanto essa função não corrige os vértices repetidos e por tal não é considerada eficiente. Obs.: A uma versão em testes de parserOBJ para corrigir os vértices repetidos, contudo atualmente o congelamento causado pela busca linear o torna ineficiente para uso em objetos com milhares de vértices.

instantiate: essa função deve ser chamada após configurar todos os uniformes para uma instância. Leva dois parâmetros: a quantidade de instâncias pretendidas e uma matriz de mat4 – a matriz mat4 é o único atributo de instância a diferir, futuramente espera-se que possa poder passar atributos como texturas/material para geometrias identicas que se repetem e diferem nesses parâmetros em uma versão futura dessa função.

generate: A função que efetivamente cria o VAO, VBO, e IBO para o objeto, configura o formato e deslocamentos no VBO por meio de setVertexAttribPointer do atributo membro currentShader. Uma vez que os dados tenha sido migrados para GPU, o fluxo de vértices e índices é limpo para economia de espaço.

renderMatrices: inicializa os uniformes model, view\_projection, lightSpace e camerapos usando os atributos membros respectivos.

render: Renderiza para tela efetivamente usando glDrawElements.

render(unsigned count): essa sobrecarga usa o shader de instâncias para renderizar.

Os setters setModelMatrix, setViewMatrix, setLightSpace, setCameraPosition agem como o esperado de setters.

shader: retorna o Shader interno padrão da instância.

free: desaloca os recursos usados.

# Wrapper: Terrain

Essa classe se especializa nos passos para criação de terrenos tais como, carregamento de mapas de altura, geração de vertíces a partir do mapa de altura carregado, calculo de normais, criação e vinculação de texturas e teste de colisão do observador com o terreno.

## Atributos

## Tecnicas de sombreamento

Considerações sobre cores: para computação gráfica usamos a seguinte abstração: dada uma luz de cor qualquer, a cor que percebemos de um objeto é o componente de cor da luz que não foi absorvido pelo objeto resultando na sua rejeição e portando reflexão. Um objeto não tem cor próprio apenas refletindo a gama de cores presente em uma luz de acordo com as propriedades intrisicas do objeto.

A gama de cores presente na computação é limitada se comparada ao mundo real e são definidas pelo espectro de cor RGB variando entre o intervalo [0, 1].

### Modelo de iluminação de phong

Consistem em três componentes principais:

* ambiente: as multiplas reflexões das luzes no ambiente contribuiem indiretamente para iluminação.
* difuso: o impacto da iluminação de acordo com a direção dos raios de luz ao a superfície, quanto mas perpendicular a superfície maior impacto.
* especular: a propriedade de reflexão da superficie.

O princípio do modelo de iluminação de Phong basea-se em determinar um espaço – de mundo, ou de visualização – para realizar os calculos de iluminação, o resultado final é o mesmo porém no espaço de visualização é possivel realizar simplificações que geram economia de dados.

### Vertex Shader

No sombreador de vértices temos o calculo padrão para geração de vértices. O diferencial está nos seguites passos:

* definir uma variável de saída para normais transformadas preferencialmente para o espaço de visualização a partir de sua posição em coordenadas de mundo pré multiplicado pela matriz normal – para correção de normais em escalas não uniformes.
* definir uma variável de saída para as coordendas de fragmento em espaço de visualização a partir de sua posição no espaço de mundo.
* definir uma variável de saída para a posição da luz em espaço de visualização a partir de sua posição no espaço de mundo.

### Fragment Shader

Aqui determinamos os componentes do modelo de iluminação de phong.

* o componente ambiente deve ser apenas uma constante de intensidade da cor de uma luz desejada para cena.
* o componente difuso diz quanto de impacto a luz tem sobre o objeto de acordo com sua direção ao objeto. Para tanto, calculamos a distância ângular entre a direção da luz pela normal do fragmento, obtemos a direção da luz pela diferença entre o vetor posição da luz é o vetor posição do fragmento (ambos em espaço de visualização) e então multiplicamos a constante difusa resultando pela cor da luz.
* o componete especular define a propriedade reflexiva do objeto. A especularidade está ligada ao ângulo de visualização do observador, portanto devemos obter a distância ângular entre a direção do visualizador e a direção refletida da direção de luz em torno da normal. A constante de especulariade resultante é então multiplicada por um constante de intensiade e pela cor da luz.

Ao obter os três componentes ambiente, difuso e especular, someos e então o valor resultante deve ser multiplicado pela cor do objeto.

### Considerações

Podemos refinar a iluminação resultante em um objeto definindo propriedades ambiente, difusa e especulares tanto para o material do objeto quanto para a intensidade da luz quando pensada em iluminação ambiente, difusa e especular.

Naturalmente, isso significa maior controle sobre cada componente – RGB – nos diferentes “estágios” da iluminação. Assim podemo dizer quanto de vermelho, verde e azul determinada luz lança quando olhamos a contribuição de iluminação ambiente de determinada luz, quanto dos compontes RGB da luz refletem quando os raios de luz estão imediatamente voltados para o objeto, e quanto o brilho de cada componente da luz será refletido. Analogamente, definir as mesma propriedades para o objeto permite-nos refinar tais parâmetros de modo que os componentes, ambiente, disfuso e especular finais seja a mistura de propriedades de luz e objeto.

Essa sequência de passos é a base para o algoritmo de iluminação baseado no modelo de Phong. Ele pode ser estendido para usar texturas para amostragem dos componentes especular, difuso e ambiente - mapas difusos e especulares - nos permitindo determinar a iluminação com base na amostragem de um fragmento ao inves de definir uma única cor material de objeto que nos da pouca fexibilidade de representação. Outras formas de mapas para amostragem são: normais/bump/reflexão/emissão.

### Luzes Direcionais

Uma luz direcional é uma luz modelada para estar infinitamente longe, o que faz com que percebamos seus raios atingindo a superfície de forma paralela uns aos outros já que todos tem a mesma direção – são idependentes da localização da fonte de luz. Como os raios de luz são paralelos, não importa como os objetos se relacionam com a posição da luz, os raios continuaram paralelos sobre toda cena.

Para definir a direção de uma luz normal usamos a lógica convencional, por exemplo um valor de -1.0 em Y significa que a luz aponta para baixo. Isso está certo para nossa interpretação mas nossas equações de iluminção sempre que cáculamos a distância angular entre dois vetores fixa o resultado final para 0.0 se o valor for negativo. Isso porque não há comportamento definido para valores de cor negativos que é o que acontecerá nesse caso já que nossa constante resultante do produto escalar é influência a cor do fragmento. Para fins de interpretação do programador, seguimos a lógica habitual para descrever direções, mas precisamos negar essa direção antes de inciarmos os calculos de iluminação para não corrermos o risco de obter constantes nulas quando não for preterido.Veja exemplo abaixo:

//Global Light

vec3 direction = vec3(0.0f, -1.0f, 0.0f);

vec3 lightDir = normalize(-direction);

### Luzes Pontuais

O que vimos até antes do modelo da luz direcional era um modelo simplista de luz pontual, vamos estender o seguinte fator para essa luz: atenuação.

O princípio da atenuação é diminuir a intensidade da luz conforme a distância que o raio de luz viaja. Antes de começar a quebrar a cabeça em cima de equações lineares, saiba que isso seria desatroso e seus resultados não se assemelhariam a realidade, felizmente, há uma equação já pronta para isso:

d: a distância do fragmento à fonte de luz.

Kc: o termo constante – geralmente definido como 1.0, é definido aqui para garantir que o denominador nunca seja menor que 1, pois de outra forma aumentaria a instensidade em determinada distâncias.

Kl: o termo multiplicado pela distância que reduz a instensidade linearmente.

d²: o termo quadrático é multiplicado pelo quadrante da distância e define uma diminuição quadrática de intensidade para a fonte de luz. O termo quadrático será menos significativo em comparação com o termo linear quando a distância é pequena, mas fica muito maior à medida que a distância aumenta.

Aqui há uma tabela com valores de variáveis que fazem sentido sobre certa distãncia:

|  |  |  |  |
| --- | --- | --- | --- |
| **Distância** | **Constante** | **Linear** | **Quadrático** |
| 7 | 1.0 | 0.7 | 1.8 |
| 13 | 1.0 | 0.35 | 0.44 |
| 20 | 1.0 | 0.22 | 0.20 |
| 32 | 1.0 | 0.14 | 0.07 |
| 50 | 1.0 | 0.09 | 0.032 |
| 65 | 1.0 | 0.07 | 0.017 |
| 100 | 1.0 | 0.045 | 0.0075 |
| 160 | 1.0 | 0.027 | 0.0028 |
| 200 | 1.0 | 0.022 | 0.0019 |
| 325 | 1.0 | 0.014 | 0.0007 |
| 600 | 1.0 | 0.007 | 0.0002 |
| 3250 | 1.0 | 0.0014 | 0.000007 |

### Luz Holofote

Uma luz holofote comporta-se como uma luz pontual com um cone de alcance. Objetos fora desse cone não são iluminados. Para configurar uma luz holofote usamos os seguitnes parâmetros:

* posição do holoféte
* direção dos raios do holofote
* ângulo de corte
* distancia ângular entre a direção do fragmento para luz e a direção dos raios de luz.

Definir uma luz holofote com apenas um ângulo de corte resultará em um corte abrubto e bem siluetado da região iluminada para a não iluminada, corrigiremos isso usando dois ângulos, um para o cone interno e outro para o cone externo e então com base no valor do ângulo entre a posição do fragmento e o vetor de direção da luz determinamos que a intensidade da luz vai diminuindo linearmente ao longo que o fragmento se aproxima do fim do raio externo – representado pelo cosseno entre o o vetor de direção da luz e o vetor do cone externo. Para valores menores e iguais a zero após a interpolação o fragmento estará na sombra. Valores maiores que 0 são mais iluminados na medida que deslocam-se para o interior do cone interno.

\*O material completo pode ser verificado em [C:\Users\Qital-PC\Desktop\Learn OpenGL\Iluminação\LearnOpenGL - Iluminação Básica](file:///C:\Users\Qital-PC\Desktop\Learn%20OpenGL\Iluminação\LearnOpenGL%20-%20Iluminação%20Básica.pdf), C:\Users\Qital-PC\Desktop\Learn OpenGL\Iluminação\LearnOpenGL – Materiais, C:\Users\Qital-PC\Desktop\Learn OpenGL\Iluminação\LearnOpenGL - Mapas de iluminação, C:\Users\Qital-PC\Desktop\Learn OpenGL\Iluminação\LearnOpenGL - Lançadores de Luz