Criação de Objetos 3D no OpenGL Parte 2

Rafael Eitaró Oshiro, Matheus Tavares Guerson e Salomão Silva

FACOM – Faculdade de Computação,

UFMS – Universidade Federal de Mato Grosso do Sul

rafael.oshiro@ufms.br, mathes.guerson@ufms.br, salomao_silva@ufms.br

Resumo. *Esta parte do trabalho se baseia em criar uma iluminação para os sólidos phong para os sólidos que fizemos na primeira parte do trabalho*

1. Introdução

Esta parte do trabalho tem como objetivo demonstrar a aplicação de conceitos fundamentais de computação gráfica e programação gráfica em um ambiente 3D utilizando a biblioteca OpenGL. O código-fonte fornecido como base implementa a renderização de três sólidos 3D - um paralelepípedo, um cubo e uma pirâmide de base quadrada - que rotacionam em torno do eixo y no sentido anti-horário. A rotação dos objetos é acompanhada por cores vibrantes, tornando mais evidente o efeito visual da transformação.

O aplicativo permite aos usuários escolher o tipo de iluminação que desejam aplicar aos sólidos, com três opções disponíveis: "phon", "ambient", "diffuse" e "specular". Essa escolha afeta diretamente a forma como os objetos são iluminados e exibidos na tela.

O código-base utilizado como referência foi adaptado a partir do repositório "OpenGL" disponível no site "Replit", que é implementado em linguagem C/C++. Essa escolha de base de código facilitou o desenvolvimento e permitiu a exploração de conceitos abordados nas aulas de computação gráfica de maneira prática e interessante.

Este trabalho serve como uma introdução prática à programação gráfica 3D, abordando conceitos essenciais, como a criação de janelas, renderização de objetos, aplicação de iluminação e uso de shaders. Ao longo deste documento, detalharemos os principais aspectos da implementação, destacando os desafios enfrentados e as soluções adotadas para criar uma aplicação gráfica.

2. Estrutura de pastas e arquivos

Em nosso projeto possuímos uma estrutura de pastas divididas, e vamos explicar como foi dividido nosso código.

Nome Da Pasta	Função
bin	Arquivo executável é salvo após a compilação do Makefile

lib	São as bibliotecas customizadas para facilitar a manipulação de dados do OpenGL em C
res	Armazena os shaders utilizados na aplicação
src	Armazena todo nosso código fonte necessário
Makefile	Arquivo que compila nosso trabalho com todas as flags necessárias

Na pasta lib possui algumas bibliotecas personalizadas, que na verdade são cópias de bibliotecas muito utilizadas em C++ para o desenvolvimento de ambientes virtuais mas escritas em C para facilitar a conversão de alguns dados e gerar alguns números de forma mais fácil.

Na pasta src possuímos 2 subpastas "solid" e "world" e nelas estão armazenados a maior parte do nosso código, e claro temos nosso "main.c" que é a porta de entrada de toda nossa aplicação.

A pasta "world" são códigos do repositório do "Replit" que nós utilizamos como base para o desenvolvimento, nela possui os arquivos "window.c" e "shader.c" que e seus respectivos cabeçalhos. O arquivo "window.c" é responsável pela criação, atualização e remoção da janela de um modo geral. Já o "shader.c" é responsável pela compilação e aplicação do shader em nossa cena, contendo também algumas funções auxiliares que nos ajudaram muito.

A pasta "solid" é onde os códigos responsáveis pela renderização e atualização dos sólidos estão armazenados, nele possui os arquivos "cube.c", "pyramid.c" e "parallelepiped.c" e seus respectivos cabeçalhos. Cada um desses arquivos é responsável por renderizar o objeto, animá-lo para que ele fique rotacionando em seu eixo y, e para aplicar os shaders nos sólidos.

3. Window.c

Como já explicado anteriormente "window.c" foi utilizado do repositório base, mas iremos explicar como cada função funciona.

A função createWindow(). Este método é encarregado pela iniciação do GLFW, definição da versão do OpenGL, composição da janela e a importação das funções OpenGL por meio da GLAD. Configurações adicionais como teste de profundidade e culling de faces também são estabelecidas aqui.

O encerramento da janela é gerenciado pela função terminateWindow(), assegurando que todos os recursos sejam devidamente liberados e que o GLFW seja encerrado.

A função updateWindow(), que entra em cena a cada ciclo de renderização, atualizando o tempo entre frames e preparando a janela para receber novos gráficos, definindo a cor de fundo e limpando os buffers.

4. Shader.c

Como já explicado anteriormente "shader.c" foi utilizado do repositório base, mas iremos explicar como cada função funciona.

A função principal createShader(), que compila e vincula os shaders de vertex e fragment, criando um programa de shader completo. Esta função é o núcleo do arquivo, pois prepara os shaders para serem usados na renderização.

A função useShader() ativa o programa de shader para que ele possa ser utilizado na renderização atual, enquanto que as funções setShaderBool(), setShaderInt(), setShaderFloat(), setShaderMat4(), setShaderVec4() e setShaderVec3() são utilitários que permitem definir variáveis uniformes nos shaders. Essas funções são fundamentais para a passagem de dados do nosso programa para os shaders, permitindo que manipulemos aspectos como transformações, cores e iluminação.

5. Cube.c, Pyramid.c e Parallelepiped.c

Primeiro, são definidos vetores para os vértices e índices do sólido. Esses vetores são essenciais para definir a forma e estrutura do sólido que será renderizado na cena. Para essa segunda parte do trabalho nós tivemos que acrescentar a normal, aos vértices de cada sólido para que nosso shader pudesse realizar as contas da iluminação.

A função create() é encarregada de iniciar o shader específico para o sólido (no caso estamos utilizando o mesmo para os 3), configurar os buffers de vértices (VBO) e elementos (EBO), e associar estes ao objeto de array de vértices (VAO). Ela também estabelece os atributos de posição, cor e normais para os vértices e configura as matrizes de projeção, visualização e modelo, que são utilizadas para posicionar, orientar e transformar o sólido na cena. A matriz model transforma as coordenadas locais do objeto para o mundo; a matriz view define a posição e orientação da câmera; e a matriz projection aplica a perspectiva, dando a sensação de profundidade. Além disso, ela também é responsável por configurar a iluminação do shader sendo passado como variável Uniform para o shader.

Já a função draw() é utilizada para aplicar uma rotação contínua ao cubo com base em um delta de tempo, o que proporciona uma animação fluida e contínua. Após ajustar a matriz do modelo com a nova rotação, a função emite o comando para desenhar o cubo usando os índices definidos anteriormente o modelo atualizado para o shader redesenhar o sólido.

6. main.vs

Este arquivo de shader é executado para cada vértice do objeto 3D. A função principal, main(), descreve as operações realizadas nos vértices antes de serem passados para o próximo estágio do pipeline gráfico.

Primeiramente, são declaradas três entradas, aPos, aColor e aNormal, que recebem a posição, a cor e a normal de cada vértice. Estas estão ligadas aos atributos definidos no

programa principal através da notação layout(location=0), layout(location=1) e layout(location=2).

O shader também declara três saídas, FragPos, Normal e Color, que passarão a posição do fragmento no espaço do mundo, a normal transformada e a cor dos vértices para o próximo estágio do pipeline.

Além disso, são definidas três matrizes uniformes: projection, view e model, que são passadas para o shader quando iniciamos o objeto 3D e quando o atualizamos.

Dentro da função main(), é realizada a multiplicação dessas matrizes para obter a matriz MVP (Model-View-Projection), que é então aplicada à posição do vértice para calcular FragPos e à normal do vértice para calcular Normal. Os resultados são armazenados nas variáveis FragPos e Normal.

A cor dos vértices é simplesmente passada para a variável Color.

Por fim, as coordenadas dos vértices transformados, a normal transformada e a cor são passadas para o próximo estágio do pipeline através das variáveis gl_Position, Normal e Color. Este processo é fundamental para que o fragment shader possa utilizar essas informações para calcular a cor final de cada pixel do objeto renderizado.

7. main.fs

Este arquivo de shader tem a responsabilidade de calcular a cor final de cada pixel do objeto 3D, levando em consideração a iluminação. A função principal, main(), descreve as operações executadas no fragment shader para determinar a cor do fragmento.

Dentro da função main(), o shader recebe as seguintes entradas: FragPos (representando a posição do fragmento), Normal (indicando a normal do fragmento) e Color (que representa a cor do vértice correspondente). Além disso, há uma estrutura denominada Light, que contém informações sobre a luz, como a sua posição, componente ambiente, componente difusa e componente especular. Também é fornecida a posição da câmera (viewPos) como uniforme.

Cálculo do Componente Ambiente:

O shader calcula o componente ambiente multiplicando a cor do vértice (Color) pela cor ambiente da luz (light.ambient).

Cálculo do Componente Difuso:

A normal do fragmento (Normal) e a direção da luz incidente são normalizadas para garantir que tenham comprimento unitário. É calculado o ângulo entre a normal e a direção da luz usando o produto escalar (dot). O resultado é limitado ao mínimo de 0 para evitar valores negativos. O componente difuso é calculado multiplicando o ângulo calculado pela cor difusa da luz (light.diffuse) e pela cor do vértice.

Cálculo do Componente Especular:

É definido um valor de força especular (specularStrength). A direção da visão (viewDir) e a direção da reflexão da luz (reflectDir) são calculadas. É calculado o ângulo entre a direção da visão e a direção da reflexão usando o produto escalar. O resultado é elevado a uma potência para controlar o brilho especular. O componente especular é calculado multiplicando o resultado pela cor especular da luz (light.specular) e pela força especular.

Combinação dos Componentes:

Os componentes ambiente, difuso e especular são somados para obter a cor final do fragmento. A cor final é armazenada na variável "result".

Definição da Cor do Fragmento:

A cor final é atribuída à variável "FragColor" como um vec4, com um componente alfa de 1.0 (totalmente opaco).

Este shader desempenha um papel fundamental no processo de renderização, calculando a cor final de cada fragmento com base na iluminação, o que contribui para a criação de uma aparência realista de superfícies 3D na cena renderizada.

8. Considerações Finais

Neste trabalho demoramos muito para entender os conceitos dos shader no OpenGL, pois não estávamos conseguindo fazer os sólidos rotacionarem por conta de uma conta errada na projeção passada para o shader.
Além disso, também precisamos aprender a lidar com índices, pois não queríamos criar vértices duplicados no mesmo vetor. Então utilizamos os índices para facilitar a criação das faces de cada sólido. Mas para isso tivemos que ter um entendimento de como isso seria aplicado, principalmente com os shaders.

Para executar nosso trabalho, é necessário utilizar um terminal Linux, no nosso caso utilizamos a WSL, um terminal Linux dentro do Windows. Tendo isso em vista, basta executar os seguintes comandos.

Compilar nosso código:

```
make
```

Para executar nosso código:

```
bin/main <TipoDeIluminaçãoDesejada>
```

Opções:

1. phong
2. diffuse
3. ambient
4. specular

Por exemplo:

```
bin/main phong
```

Aqui está um link para o google drive, onde estão vídeos mostrando os códigos rodando

Link do google drive:

https://drive.google.com/drive/folders/1WJoXxdiw4_7xKSqT9ckl_p2-QyePpaz-?usp=drive_link

Por fim, esse trabalho foi muito gratificante por poder colocar em prática tudo o que vimos durante as nossas aulas.