Universidade Federal da Paraíba

Centro de Informática

Engenharia da Computação

Introdução à Computação Gráfica

Atividade Prática 3

Implementação do Pipeline Gráfico

Aléxandros Augustus

11501517

João Pessoa

2020

1. Objetivo

Neste trabalho, a partir de um programa *template* fornecido pelo professor, fizemos a modificação das matrizes *Model*, *Projection* e *View*, já estudadas na disciplina e implementadas no *template* pelo professor para exibir a cena definida pelo professor com os parâmetros fornecidos.

2. Instalação de ferramentas de terceiros

Para o funcionamento adequado do programa *template* foi necessário instalar bibliotecas adicionais, inclusas pelo professor na descrição da tarefa.

2.1. glm

Bilbioteca que contém algumas funções utilizadas no *template*. Foi disponibilizada pelo professor no repositório do GitHub como um submódulo, então sua instalação foi feita junto com a clonagem do programa *template*, através do comando a seguir.

```
$ git clone --recurse-submodules https://github.com/capagot/icg.git
```

Terminal 1 - Clonagem do programa template e instalação da biblioteca glm

2.2. GLEW

Biblioteca que gera referências para todas as funções OpenGL, necessária para programas escritos em C++ ou outras linguagens que não aceitem declaração implícita de funções. Foi instalada a partir da execução do comando a seguir.

```
$ git clone https://github.com/nigels-com/glew.git glew
```

Terminal 2 - Instalação da biblioteca GLEW

3. Compilação e teste do programa *template*

Após instalar as bibliotecas necessárias, utilizamos os seguintes comandos para compilar e executar o programa.

```
$ make
$ ./transform_gl
```

Terminal 3 - Compilação e execução do programa template

Infelizmente, a janela do programa, assim que foi aberta, fechou-se e a execução foi abortada, na janela de terminal encontramos erros informando que a versão 3.30 utilizada pelo *shader* não era suportada, buscamos na internet possíveis soluções sem muito sucesso, porém, lembramos que um problema semelhante ocorreu na primeira atividade prática, em que o triângulo não era exibido com a cor esperada, então tentamos executar o programa utilizando os mesmos parâmetros utilizados na atividade prática 1.

```
$ MESA_GL_VERSION_OVERRIDE=3.3 MESA_GLSL_VERSION_OVERRIDE=330 ./transform_gl
```

Terminal 4 - Nova tentativa de execução do programa template

Desta vez, o programa foi executado sem problemas e obtivemos a tela esperada.

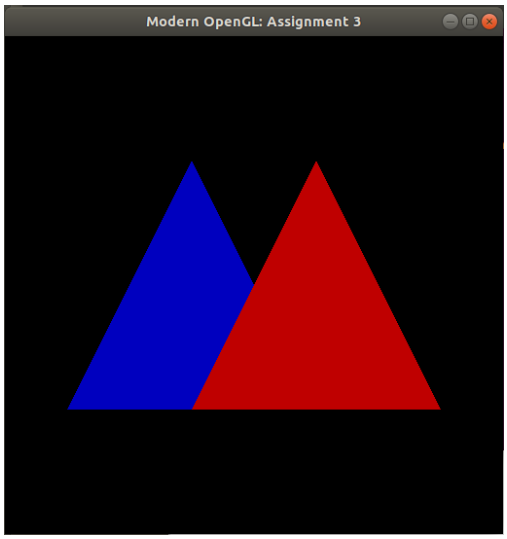

Figura 1 - Cena original do programa template

4. Alterações no código

Nesta atividade o professor solicitou que as matrizes responsáveis pela visualização final da cena fossem alteradas segundo os parâmetros dados de forma a implementar transformações gemoétricas e diferentes perspectivas da câmera, para evitar a alteração direta das matrizes implementadas, adicionamos alguns trechos de código para receber os parâmetros necessários e então calcular a nova matriz a partir destes.

4.1. Bibliotecas e funções extra

Adicionamos ao programa a biblioteca “cmath” e duas funções que serão necessárias para os cálculos: uma que calcula o produto vetorial entre dois vetores de dimensão 3, e outra que calcula a norma de um vetor de dimensão 3.

4.1.1. Biblioteca “cmath”

Para adicionat a biblioteca, bastou acrescentar a linha de código a seguir, logo no início do programa. Ela foi necessária pois ao longo das alterações feitas precisamos calcular raiz quadrada, seno e cosseno.

```
#include <cmath>
```

Excerto de código 1 - Inclusão da biblioteca cmath

4.1.2. Função CrossP

Esta função foi implementada para calcular o produto vetorial entre dois vetores de dimensão 3, chamados por A e B, e salvar o resultado desse produto no vetor C. A implementação adotada segue o teorema do produto vetorial por notação de coordenada, descrito abaixo.

$$Sejam\ os\ vetores\ A = (a1)\hat{\imath} + (a2)\hat{\jmath} + (a3)\hat{k}\ \ e\ B = (b1)\hat{\imath} + (b2)\hat{\jmath} + (b3)\hat{k}, temos\ que:$$

```
void CrossP(float *A, float *B, float *C) {
	C[0] = (A[1]*B[2] - A[2]*B[1]);
	C[1] = (A[2]*B[0] - A[0]*B[2]);
	C[2] = (A[0]*B[1] - A[1]*B[0]);
}
```

Excerto de código 2 - Função CrossP

4.1.3. Função Norm

Esta função foi implementada para calcular a norma de um vetor de dimensão 3, implementando o teorema descrito abaixo.

$$Seja\ V\ um\ vetor\ tal\ que\ V = a\hat{\imath} + b\hat{\jmath} + c\hat{k}, temos\ que:$$

$$\|V\| = \sqrt{a^2 + b^2 + c^2}$$

```
float Norm(float *V) {
	float sum = 0.0f;
	for(int w = 0; w < 3; w++){sum = sum + V[w] * V[w];}
	return sqrt(sum);
}
```

Excerto de código 3 - Função Norm

4.2. Matriz *Model*

Esta é a matriz que determina as transformações geométricas sofridas pela imagem, sendo assim implementamos as matriz de algumas transformações estudadas e no final atribuímos à matriz model a multiplicação das transformações implementadas com o trecho a seguir.

```
model_mat = scale_mat * translation_mat;
```

Excerto de código 4 - Composição de transformações geométricas

Este trecho não precisa ser alterado pois a ordem de aplicação de escala e translação é desprezível, além disso, os parâmetros de ambas as transformações têm seu valor padrão definido de forma a gerar como resultado a matriz identidade, assim fazendo com que a transdormação que não esteja sendo aplicada não afete o resultado final. Detalharemos abaixo a implementação dessas transformações.

4.2.1. Escala

Para a escala no espaço 3D são necessários 3 parâmetros, um para cada dimensão, chamados s_x, s_y e s_z. A partir destes a matriz será:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

No código, criamos três variáveis para receber cada um destes parâmetros, e então geramos esta mesma matriz utilizando a biblioteca glm. O valor padrão dos três parâmetros é 1.

4.2.2. Translação

Utilizamos três parâmetros, que chamaremos de d_x, d_y e d_z, que representam o deslocamento em cada eixo relativo à origem do sistema de coordenadas. A matriz que representa esta operação é dada por:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

No código, de forma semelhante à operação de escala, criamos uma variável para cada parâmetro e então utilizamos a glm para criar a matriz acima. O valor padrão dos três parâmetros é 0.

4.3. Matriz *View*

É a matriz responsável por determinar a posição da câmera em relação à cena. É calculada através da multiplicação de duas outras matrizes, B^T e T, que representam a base ortonormal da câmera transposta e a matriz de translação da posição da câmera, respectivamente, necessárias para fazer a mudança de base que coloca a câmera como a origem do sistema de coordenadas. Para gerar esta matriz precisamos de três parâmetros: a posição da câmera, o ponto para o qual ela aponta, e seu vetor *up*. A partir desses devemos calcular o vetor direção da câmera, obtida subtraindo a posição da câmera do ponto para o qual ela aponta. Em seguida, utilizamos o vetor direção calculado e o vetor *up* definido para gerar uma base ortonormal para a câmera, utilizando as fórmulas a seguir.

$$z_{cam} = -\frac{d_{cam}}{\|d_{cam}\|}, onde\ d_{cam}\ \text{é}\ o\ vetor\ direção\ da\ câmera$$

$$x_{cam} = \frac{u \times z_{cam}}{|u \times z_{cam}|}, onde\ u\ \text{é}\ o\ vetor\ \text{up}\ da\ câmera$$

$$y_{cam} = z_{cam} \times x_{cam}$$

A partir destas fórmulas, temos que a matriz B^T é dada por:

$$B^T = \begin{bmatrix} x_{cam(i)} & x_{cam(j)} & x_{cam(k)} & 0 \\ y_{cam(i)} & y_{cam(j)} & y_{cam(k)} & 0 \\ z_{cam(i)} & z_{cam(j)} & z_{cam(k)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

E a matriz T é dada por:

$$T = \begin{bmatrix} 1 & 0 & 0 & -p_i \\ 0 & 1 & 0 & -p_j \\ 0 & 0 & 1 & -p_k \\ 0 & 0 & 0 & 1 \end{bmatrix}, onde\ P = (p_i, p_j, p_k)\ \text{é}\ o\ ponto\ posição\ da\ câmera$$

No código, criamos três variáveis, representando cada parâmetro necessário, então calculamos o vetor direção, e os vetores x_{cam}, y_{cam} e z_{cam}, em seguida geramos as matrizes B^T e T utilizando a bilbioteca glm, por fim atribuímos à matriz *view* o produto destas. Os valores padrão dos parâmetros é definido de forma que ao calcular B^T e T, ambas resultem na matriz identidade. Sendo P o ponto posição da câmera, D o ponto para o qual ela aponta, e u seu vetor *up*, os valores padrão de P, D e u são os seguintes:

$$P = (0, 0, 0)$$

$$D = (0, 0, -1)$$

$$u = (0)\hat{\imath} + (1)\hat{\jmath} + (0)\hat{\kappa}$$

4.4. Matriz *Projection*

É a matriz que faz a projeção perspectiva da cena na tela. Para cálculá-la precisamos do parâmetro d, que representa a distância do centro de projeção ao centro do sistema de coordenadas da câmera. A matriz é então dada por:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & -1/d & 0 \end{bmatrix}$$

No código, criamos uma variável para o parâmetro d, que por padrão recebe o valor 0. Então adicionamos uma condição *if* que verifica o parâmetro d, caso este seja zero, o array definido pelo professor é passado para a glm para se transformar na matriz identidade. Caso seja diferente de zero, o array então sofrerá alteração em três de seus elementos para que a matriz gerada pela glm após o *if* terminar seja a matriz mostrada acima.

5. Exercícios

5.1. Escala

O objetivo é alterar apenas a matriz *view*, utilizando a transformação de escala com os seguintes parâmetros definidos pelo professor: $s_x = 1/3$, $s_y = 3/2$ e $s_z = 1$. Alteramos apenas estes 3 parâmetros já que o de todas as outras transformações e matrizes resultam na matriz identidade. Abaixo podemos ver que a imagem gerada pelo programa foi igual à esperada.

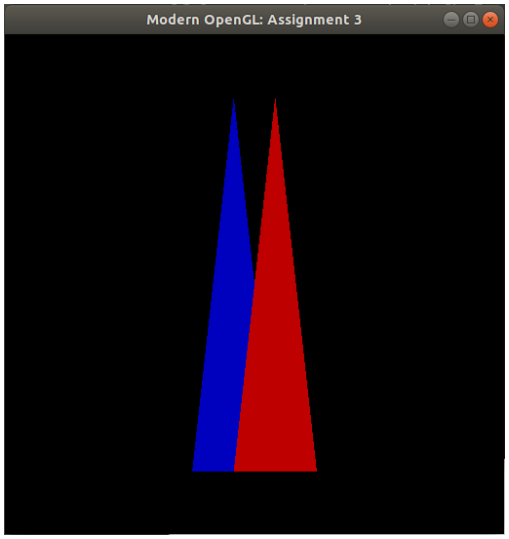

Figura 2 - Aplicação de escala

5.2. Translação

O objetivo é alterar apenas a matriz *view*, utilizando a transformação de translação com os seguintes parâmetros definidos pelo professor: $d_x = 1$, $d_y = 0$ e $d_z = 0$. Retornamos os parâmetros de escala ao seu valor padrão, e alteramos os fatores de translação como definido pelo professor, mais uma vez não foi necessária nenhuma outra alteração já que todas as outras matrizes são a identidade. Abaixo podemos ver que a imagem gerada pelo programa foi igual à esperada.

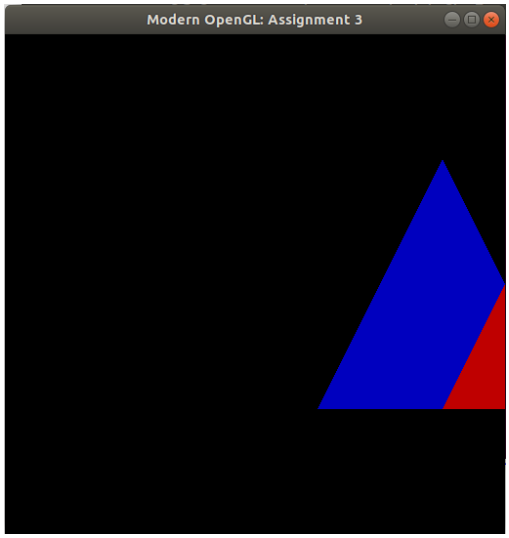

Figura 3 - Aplicação de translação

5.3. Projeção Perspectiva

O objetivo é alterar apenas a matriz *projection*, utilizando o seguinte parâmetro definido pelo professor: d = 1/8. Retornamos os parâmetros de translação ao seu valor padrão, e alteramos o parâmetro da matriz *projection* para o valor definido. Abaixo podemos ver que a imagem gerada pelo programa foi o esperado.

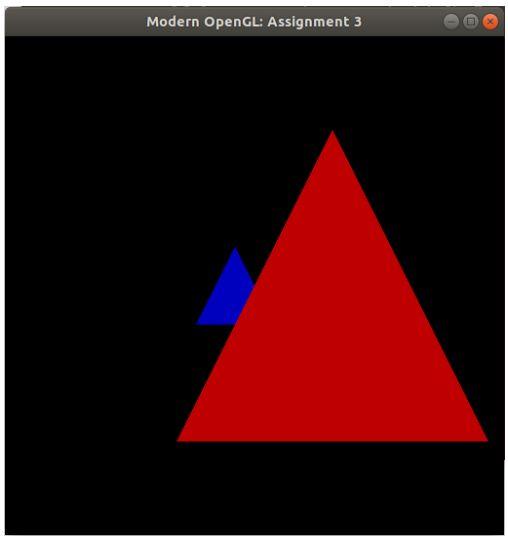

Figura 4 - Aplicação de projeção perspectiva

5.4. Posição da Câmera

O objetivo é, mantendo a alteração feita na matriz *projection* no item anterior, alterar também a matriz *view*, com os seguintes parâmetros definidos pelo professor: P = (-1/10, 1/10,

1/10), u = (0)î + (1)ĵ + (0)k̂ e D = (0, 0, -1), onde P, u e D são o ponto posição da câmera, o vetor *up* da câmera, e o ponto para o qual a câmera aponta, repsectivamente. Alteramos os parâmetros da câmera no código de acordo com os definidos e obtivemos uma imagem igual à esperada.

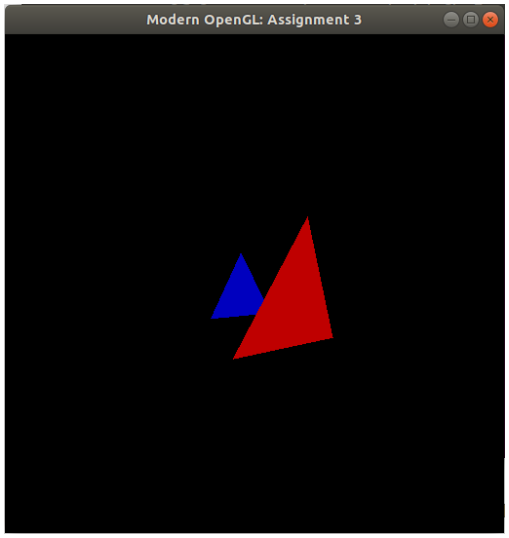

Figura 5 - Aplicação de projeção perspectiva com mudança da posição da câmera

5.5. Transformações Livres

O objetivo é modificar as matrizes *model*, *view* e *projection* de forma a gerar diferentes cenas. A seguir esão os testes feitos pelo aluno.

5.5.1. Aplicação de escala e translação

O objetivo é utilizar escala -1 no eixo X, para espelhar os triângulos neste eixo, e então subí-los no eixo Y em meia unidade, sendo assim os parâmetros que precisaremos alterar são: s_x = -1 e d_y = 1/2. Além disso, os parâmetros das matrizes *view* e *projection* foram definidos novamente como seu valor padrão. Abaixo podemos ver a imagem gerada pelo programa, como esperado, o triângulo azul, que antes ficava à esquerda do vermelho, está à sua direita, e os triângulos estão mais próximos da borda superior da tela do que os triângulos mostrados na cena original, vista na Figura 1.

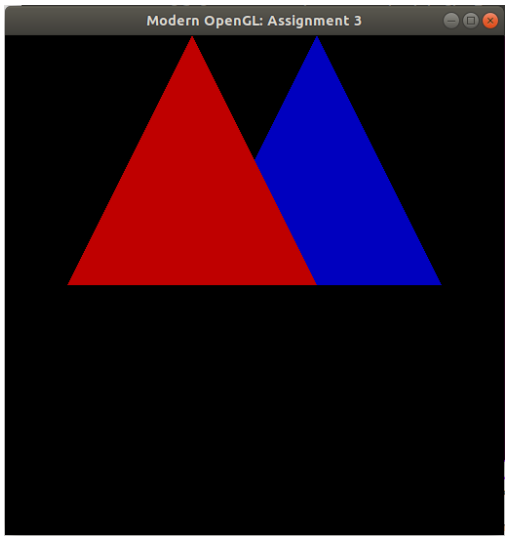

Figura 6 - Aplicação de espelhamento no eixo X e translação no eixo Y

5.5.2. Aplicação da posição da câmera

Mantendo as alterações do item anterior, vamos modificar a matriz *view*, de modo a fazer com que os parâmetros da câmera sejam P = (0, 1, 0), D = (1/2, -1, 1) e o vetor *up* se manterá o padrão. Abaixo vemos a imagem gerada, podemos perceber que temos uma visão de cima da imagem, porque a câmera está posicionada acima dos triângulos e aponta para baixo, além disso, como a câmera aponta na direção positiva do eixo Z, estamos tendo uma visão das "costas" da figura, e por isso o triângulo azul está à frente do vermelho.

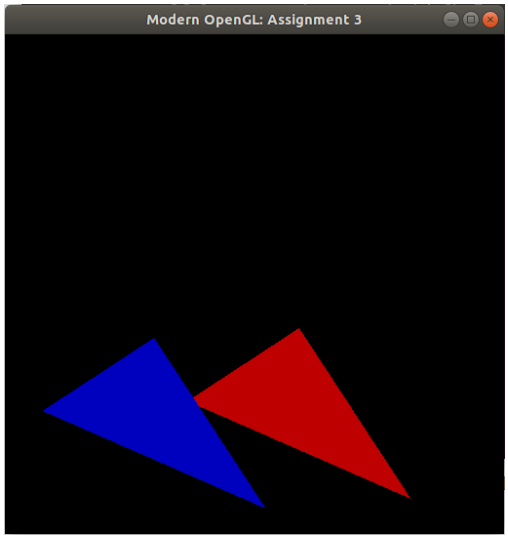

Figura 7 - Posicionamento da câmera

5.5.3. Aplicação da projeção perspectiva

Ainda mantendo as alterações de itens anteriores, modificamos o parâmetro d da matriz *projection* para que fosse 0,001. Abaixo podemos ver a imagem gerada, notamos que a imagem sofreu muita distorção, pois os triângulos estão muito próximos da origem do sistema de coordenadas, assim sendo necessário um d muito pequeno para que o ângulo da câmera consiga capturar a imagem inteira.

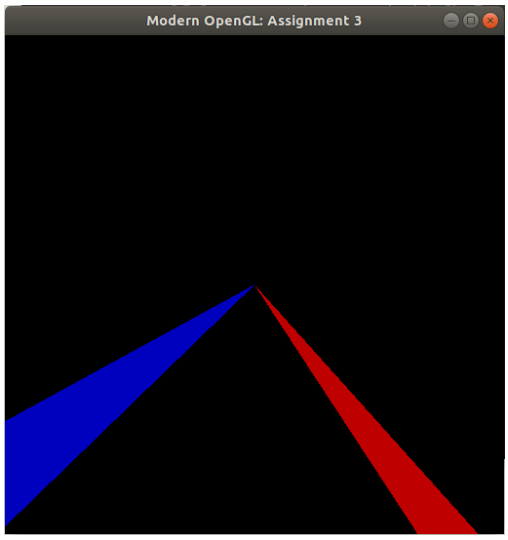

Figura 8 - Projeção perspectiva (primeira tentativa)

Para gerar uma imagem com perspectiva menos distorcida, alteramos um pouco a cena, retornando as matrizes *model* e *view* a seus valores padrão, e então utilizando d = 0,1.

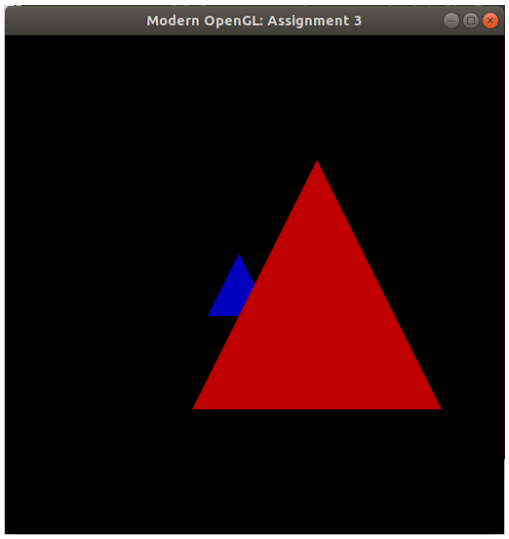

Figura 9 - Projeção perspectiva (segunda tentativa)

Podemos ver que a projeção agora se aplica mais visivelmente. Vamos então tentar fazer com que a câmera nos dê uma visão lateral dos triângulos, para isso, alteramos a posição da câmera para (-1/4, 0, 0), e fizemos com que ela apontasse para (0, -1/8, -1), gerando a figura abaixo, concluindo que obtivemos o resultado desejado.

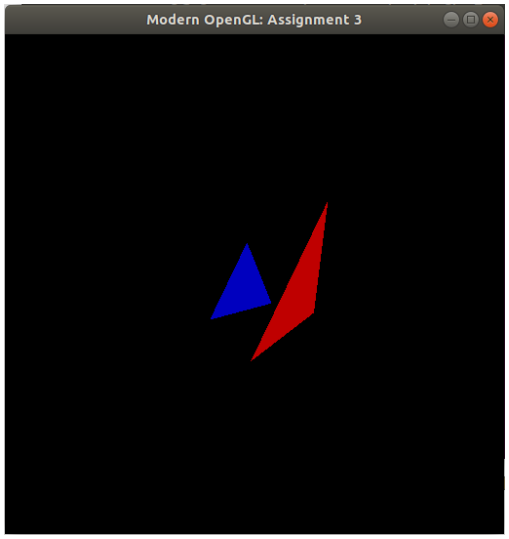

Figura 10 - Projeção perspectiva (tentativa final)

6. Referências

Vídeo-aulas e material didático disponibilizado pelo professor Christian.