UNIVERSIDADE FEDERAL DA PARAÍBA

DEPARTAMENTO DE INFORMÁTICA

INTRODUÇÃO A COMPUTAÇÃO GRÁFICA

CIÊNCIA DA COMPUTAÇÃO

Aluno: Vinícius Medeiros Wanderley **Matrícula:** 20160133667

Aluno: José Ítalo Alves de Oliveira Vitorino **Matrícula:** 2016005182

TRABALHO 2 – IMPLEMENTAÇÃO DO PIPELINE GRÁFICO

Introdução

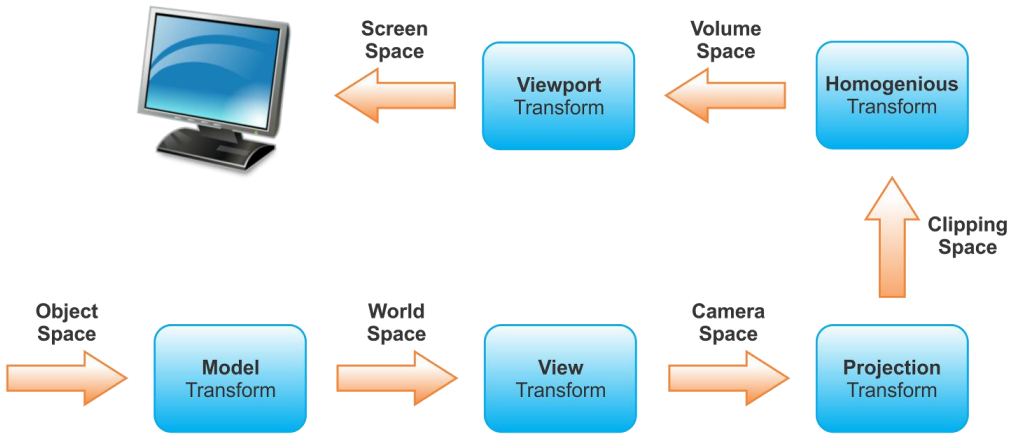
O pipeline gráfico é um dos conceitos básicos da Computação Gráfica, é fundamental entendê-lo para compreender a API gráfica OpenGL, pois várias funções dessa API agem em diferentes estágios do pipeline e a compreensão dessas etapas farão o estudante ter uma melhor experiência ao utilizar esta API.

O que é o pipeline gráfico?

É a sequência de passos necessários para transformar uma cena geométrica em uma imagem discreta na tela. Basicamente cada passo do pipeline transforma de um sistema de coordenadas (espaço) para outro, vamos falar sobre isso mais adiante.

Estágios do pipeline

A figura abaixo mostra todo o pipeline gráfico, a maioria das transformações entre os espaços é feita através de matrizes, exceto do Clipping Space (Espaço de Recorte) para o Volume Space (Espaço Canônico) onde dividimos os vértices pela coordenada homogênea.



Coordenadas Homogêneas

Por todo o pipeline usamos esse tipo de coordenada pois ela permite que transformações afins se tornem transformações lineares possibilitando assim o uso de matrizes para suas representações. É muito importante usar CH pois normalmente a gpu/cpu tem em algum compartimento uma parte especifica para calcular matrizes e por ser um cálculo executado em várias situações, os engenheiros de hardware tiveram a brilhante ideia de fazer essas contas em um lugar separado do resto, otimizando e acelerando todo o processo.

Como convertemos coordenadas cartesianas em coordenadas homogêneas?

Digamos que temos um ponto geral P (x, y, z) em coordenadas cartesianas no espaço 3D, na prática representaremos P com uma dimensão w *a mais, a*ssim teremos P' (wx, wy, wz, w) com na mesma posição em CH. Deixando mais explicita essa conversão, consideramos o ponto P (x, y, z), vamos adicionar a dimensão w com valor 1, e depois multiplicaremos o ponto P ($x, y, z, 1$) por w ficando com P' (wx, wy, wz, w). O mesmo vale para a volta da operação, dividimos P' por w e descartamos o 1 retornando para P (x, y, z).

Principais matrizes do Pipeline

Matriz de Escala: Usada para aumentar ou diminuir a coordenada em um ou mais eixos

$$\begin{aligned} x' &= x \cdot s_x \\ y' &= y \cdot s_y \\ z' &= z \cdot s_z \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Matriz de Rotação: Usada para girar em torno de um dos eixos

Rotação em X

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotação em Y

$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotação em Z

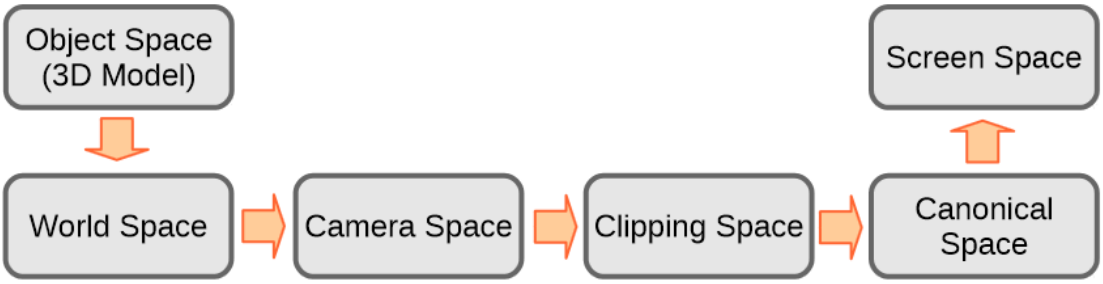
$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matriz de Translação: Usada para mover os objetos entre os eixos

$$\begin{aligned} x' &= x + d_x \\ y' &= y + d_y \\ z' &= z + d_z \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Matriz de Translação

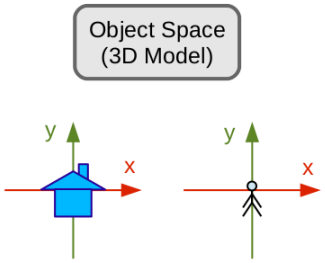
Voltando ao Pipeline



Conforme mostramos anteriormente existem vários espaços e transformações ao longo do pipeline. Com a noção de Coordenadas Homogêneas e as principais transformações usando matrizes, podemos nos aprofundar no caminho em que cada vértice da cena passa até ser exibido na tela.

Usamos algumas bibliotecas auxiliares como glm para trabalharmos como vetores e matrizes e um loader de objetos para transporta-los para o nosso programa em C++ onde implementamos o pipeline.

Espaço do Objeto – Os objetos são modelados nesse espaço, além disso, é infinito em todas as direções e não há importância em qual sistema de coordenadas foi utilizado.

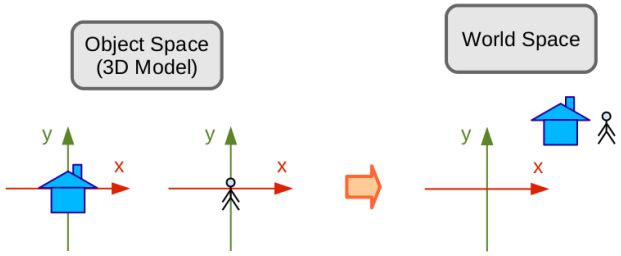


Transformação:

```
77      /******Espaço do Objeto para Espaço do Universo******/
78      glm::mat4 Scale = glm::mat4(2.0);
79      Scale[3].w = 1.0;
80
81      glm::mat4 Trans = glm::mat4(1.0);
82      glm::vec4 v(1.0,1.0, 1.0, 1.0);
83      Trans[3] = v;
84
85      glm::mat4 Rotate = glm::mat4(1.0);
86      Rotate[0].x = cos(angle +=0.0 * PI/180.0);
87      Rotate[2].x = sin(angle +=0.0 * PI/180.0);
88      Rotate[0].z = - sin(angle +=0.0 * PI/180.0);
89      Rotate[2].z = cos(angle +=0.0 * PI/180.0);
90      Rotate[3].w = 1.0;
91
92      glm::mat4 M_Model = Rotate * Scale;
```

Explicação: Neste trecho de código temos três matrizes: A primeira de Rotação onde foi escolhido o fator de iniciação 2.0, logo tivemos que alterar a variável W como 1.0 pois ela não pode variar junto com as coordenadas X, Y e Z, ou seja, o objeto foi aumentado de tamanho duas vezes. A segunda matriz é a de Translação, ela é iniciada com 1.0 e a última coluna é alterada com um vetor identidade que realiza a translação do objeto em +1.0 em cada coordenada. Por último temos a Rotação, nessa temos a rotação em Y, onde o os valores de X e Z são alterados, nessa matriz a rotação é anti-horária, por último é criada a Matriz Modelo que é gerada pela multiplicação dessas 3 últimas matrizes explicadas anteriormente, na imagem não foi multiplicada a matriz de translação pois a mesma será usada na próxima transformação.

Espaço do Universo – Onde os objetos estão reunidos, este espaço congrega os objetos em um sistema de coordenadas sendo capaz de montar a cena.

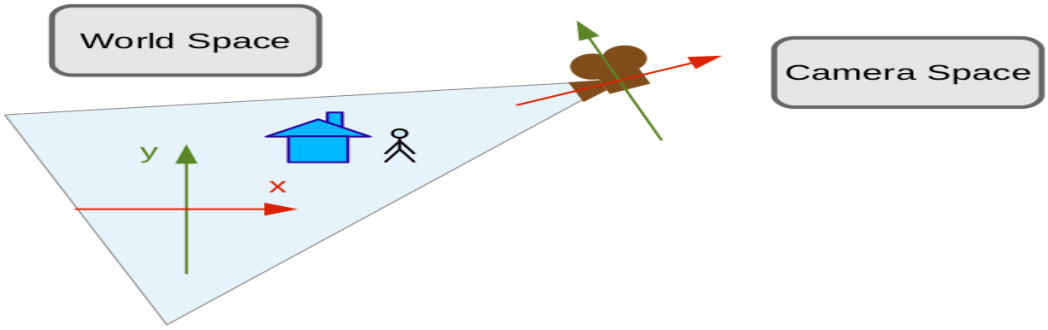


Transformação:

```
95      /******Espaço do Universo para Espaço da Camera******/
96
97      glm::vec3 camera_pos(-2.0,1.0,5.0);
98      glm::vec3 look_at(0.0,0.0,0.0);
99      glm::vec3 camera_up(0.0,1.0,0.0);
100
101     glm::vec3 camera_dir = look_at - camera_pos;
102
103
104     glm::vec3 z_camera = -normalize(camera_dir);
105     glm::vec3 x_camera = normalize(cross(camera_up, z_camera));
106     glm::vec3 y_camera = normalize(cross(z_camera, x_camera));
107
108
109     glm::vec4 homog(0.0,0.0,0.0,1.0);
110
111     glm::mat4 B = glm::mat4(1.0);
112
113     B[0]= glm::vec4 (x_camera,0.0);
114     B[1]= glm::vec4 (y_camera,0.0);
115     B[2]= glm::vec4 (z_camera,0.0);
116     B[3]=homog;
117
118     glm::mat4 trans = glm::mat4(1.0);
119     trans[3] = glm::vec4 (-camera_pos, 1.0);
120
121     glm::mat4 M_View = transpose(B) * trans;
122
123     glm::mat4 Model_View = M_View * M_Model;
```

Explicação: Neste trecho de código são criados 3 vetores: Vetor posição da câmera, vetor de foco, e o vetor “UP” da câmera. Depois é gerado o vetor direção, que é uma subtração entre o vetor de foco e o vetor posição. Em seguida foram criados 3 vetores, Z, X e Y, que serão as linhas da Matriz View, **z_camera** é o vetor direção normalizado e invertido, **x_camera** é o produto vetorial do vetor “UP” com o **z_camera** que foi obtido anteriormente, por isso o z teve que ser calculado antes. O **y_camera** é obtido pelo produto vetorial de **z_camera** e **x_camera**. Após o cálculo desses vetores, eles são unidos com um vetor homogêneo e formam uma matriz da câmera que será transposta e multiplicada por uma matriz de translação, onde será transladada a câmera para sua devida posição, assim formando a Matriz View, que ao ser multiplicada pela Matriz Model forma a Matriz Model View, finalizando assim boa parte do Pipeline.

Espaço de Camera – No espaço de câmera, todos os objetos precisam ser transformados para um novo sistema de coordenadas, a fim de serem visualizados

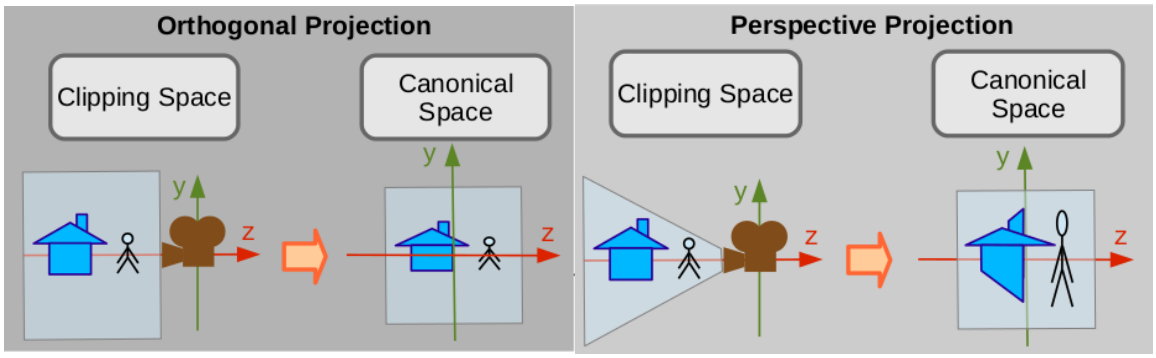


Transformação:

```
126     /******Espaço da Camera para Espaço de Projeção (Recorte)******/
127
128     double d=1.0;
129
130     glm::mat4 M_Projection = glm::mat4(1.0);
131     M_Projection[2] = glm::vec4(0.0, 0.0, 1.0, -1.0/d);
132     M_Projection[3] = glm::vec4(0.0, 0.0, d, 0.0);
133
134
135     glm::mat4 M_ModelViewProjection = M_Projection * Model_View;
```

Explicação: Neste trecho de código temos a matriz de projeção, usamos o valor da distância do plano de projeção até a câmera, e então criamos uma matriz identidade e adicionamos dois vetores ,que adicionam o valor da distância na terceira coluna da última linha e -1 dividido por esse valor na última coluna da terceira linha, onde esses valores definirão a projeção perspectiva baseado no valor de z e d, deixando o que está mais distante menor e o que está mais próximo maior, dando realismo ao objeto. No fim multiplicamos então essa matriz pela Model View, e conseguimos a Matriz Model View Projection.

Espaço de Recorte – Com os pontos do espaço da câmera, se faz necessário descartar todos os pontos que não são vistos, para obter mais eficiência na renderização. Assim, projetamos tudo em um *viewplane* e descarta-se o que não for visível. Para isso, escolhemos que tipo de projeção será utilizada, que pode ser ortogonal ou perspectiva.

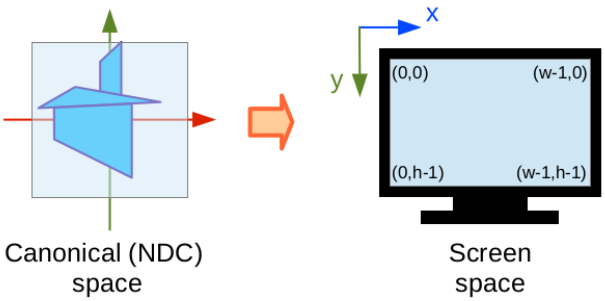


Transformação:

```
138     /******Espaço de Projeção (Recorte) para Espaço Canônico******/
139
140     for (int i = 0; i < objData->vertexCount; ++i)
141     {
142         vertex_list[i]=M_ModelViewProjection*vertex_list[i];
143         vertex_list[i].x=vertex_list[i].x/vertex_list[i].w;
144         vertex_list[i].y=vertex_list[i].y/vertex_list[i].w;
145         vertex_list[i].z=vertex_list[i].z/vertex_list[i].w;
146         vertex_list[i].w=vertex_list[i].w/vertex_list[i].w;
147     }
```

Explicação: Neste trecho de código iremos dividir todos os vetores que já foram multiplicados pela Matriz Model*View*Projection pelo valor da coordenada homogênea, por fim os vetores estão no espaço canônico e em breve serão colocados na tela.

Espaço Canônico – Nesse espaço as matrizes de todos os objetos são convertidas para coordenadas homogêneas, de maneira que, devido a uma distorção dos vértices, objetos mais próximos da câmera aparecem maiores e objetos distantes, aparecem menores.

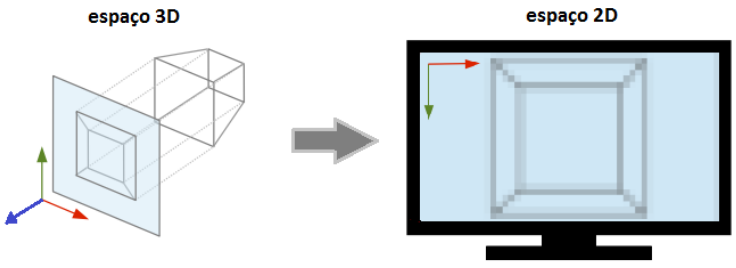


Transformação:

```
150     /******Espaço Canônico para Espaço da Tela******/
151     glm::mat4 Translation_Screen = glm::mat4(1.0);
152     Translation_Screen[3] = glm::vec4((IMAGE_WIDTH -1)/2, (IMAGE_HEIGHT -1)/2, 0.0, 1.0);
153
154     glm::mat4 Scale_Screen = glm::mat4(1.0);
155     Scale_Screen[0].x = IMAGE_WIDTH/2;
156     Scale_Screen[1].y = IMAGE_HEIGHT/2;
157
158     glm::mat4 InvertY_Screen = glm::mat4(1.0);
159     InvertY_Screen[1].y = -1.0;
160
161     glm::mat4 Final_Matrix = glm::mat4(1.0);
162     Final_Matrix = Translation_Screen * Scale_Screen * InvertY_Screen;
163
164
165     for (int i = 0; i < objData->vertexCount; ++i)
166     {
167         vertex_list[i] = Final_Matrix * vertex_list[i];
168     }
169
170
171     for (int i = 0; i < objData->vertexCount; ++i)
172     {
173         Vertex tmp;
174
175         tmp.setX(round(vertex_list[i].x));
176         tmp.setY(round(vertex_list[i].y));
177         tmp.setZ(round(vertex_list[i].z));
178         tmp.setW(round(vertex_list[i].w));
179
180         my_list.push_back(tmp);
181     }
182
183
184     for (int i = 0; i < objData->faceCount; ++i)
185     {
186         faces = objData->faceList[i];
187         my_list[faces->vertex_index[0]].DrawTriangle(my_list[faces->vertex_index[1]], my_list[faces->vertex_index[2]]);
188     }
```

Explicação: Neste trecho de código a transformação final dos vetores, onde iremos transladar os vetores do plano cartesiano para o plano da tela, usando a Matriz de Translação e a Matriz de escala, que garante que as coordenadas X e Y sejam desenhadas perfeitamente na tela, e também a matriz de inversão de Y, pois o y da tela é invertido apontando assim para baixo, após a multiplicação dessas três matrizes temos uma Matriz final que será multiplicada por cada um dos vértices que estarão transformado perfeitamente para serem desenhados na tela, então adicionamos esses vértices em variáveis da classe criada para assim começarmos a desenhar cada face do objeto na tela, usando o método DrawTriangle que é uma função da classe Vertex e usa as coordenadas de cada vértice da face para desenhar as 3 linhas do triângulo.

Espaço de tela: Com todos os vértices homogeneizados**,** podemos transferir informação para a tela, multiplicando a matriz de coordenadas homogêneas pela matriz *view port.*



Prints dos Resultados:

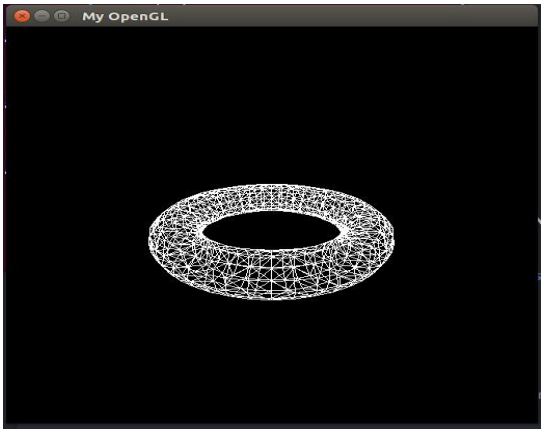


Figura 1 – Toro

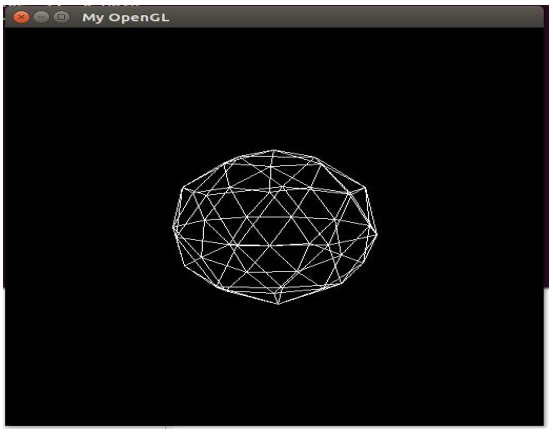


Figura 2 – Ico Sphere

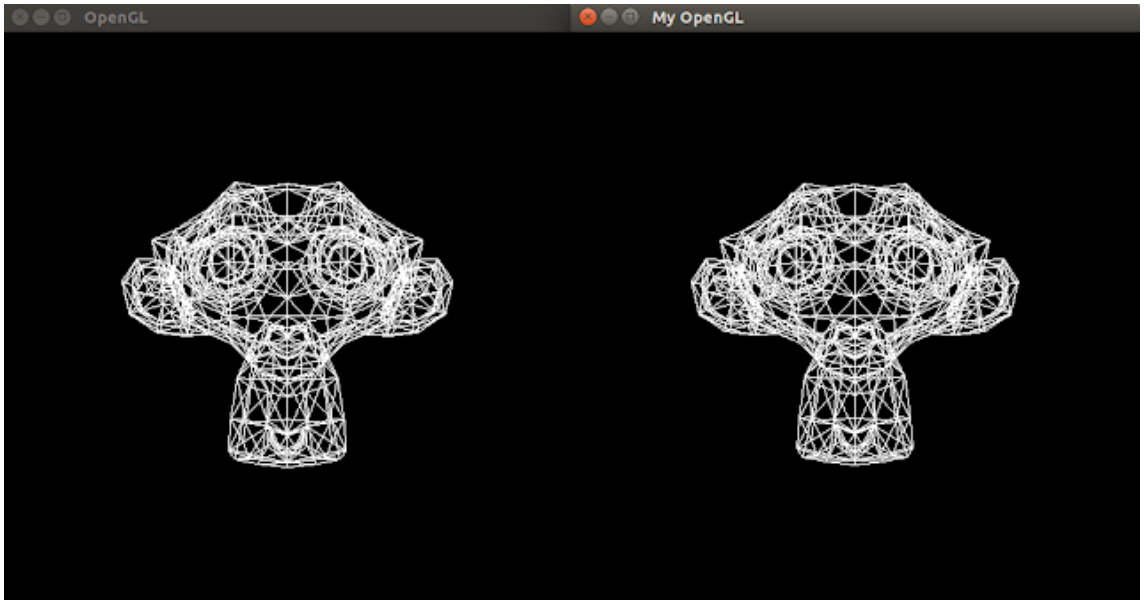


Figura 3 – OpenGL a esquerda e no nosso programa My OpengGL a direita.

Como podemos ver, há uma grande semelhança entre os objetos renderizados, concluímos então que o pipeline foi corretamente implementado.

Vídeos dos Resultados:

Toro : https://www.youtube.com/watch?v=nQ6caXfy7dc

Ico Sphere : https://www.youtube.com/watch?v=-qrpUjX7JJw

Monkey : https://www.youtube.com/watch?v=zeWm9jBeU_Q

Dificuldades:

Ocorreram algumas dificuldades intermitentes durante a realização do trabalho, mas depois de horas pesquisando na internet, conversando com outros alunos da cadeira, entre outras formas de obter informação, conseguimos realizar com sucesso o objetivo do trabalho proposto.

Uso do loader – No começo perdemos uma quantia considerável de tempo procurando um loader que funcionasse corretamente para importar os arquivos.obj, depois que o encontramos esse pequeno problema foi resolvido rapidamente. Importamos a biblioteca e instanciamos na classe principal sem maiores dificuldades.

Uso da biblioteca gml – Logo após o loader, fomos ler a documentação dessa biblioteca que usamos para fazer operações com vetores e matrizes, demoramos um menos tempo nessa tarefa, assim que fizemos alguns exemplos e testes, conseguimos com sucesso usar os métodos para realizar as operações.

Unindo todas as funções necessárias – Assim que conseguimos o loader funcionando e aprendemos o básico para usar a biblioteca gml, a parte mais difícil do projeto foi juntar tudo, fazer funcionar da forma correta e que gerasse o melhor resultado na medida do possível

Referencias:

Slides professor Christian Azambuja Pagot
https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
https://pt.wikipedia.org/wiki/OpenGL
https://en.wikipedia.org/wiki/Graphics_pipeline
Foley, J. (2010). Computer graphics. 2nd ed. Boston [u.a.]: Addison-Wesley.