T2 - Pipeline Gráfico

Em Computação Gráfica, o pipeline gráfico, é a sequência de passos que deve ser seguida para se criar uma representação 2D de uma cena 3D, ou seja, é o processo de transformar o modelo 3D em algo que o computador possa exibir.

Objetivo

A tarefa 2 proposta tem por objetivo a implementação do pipeline gráfico a fim de compreender a estrutura e o funcionamento do mesmo, transformando vértices do espaço do objeto em primitivas rasterizadas no espaço de tela.

Estágios do Pipeline Gráfico

Existem seis estágios que devem ser percorridos para que o objeto modelado chegue até o espaço de tela. Eles estão dispostos de acordo com a imagem abaixo.

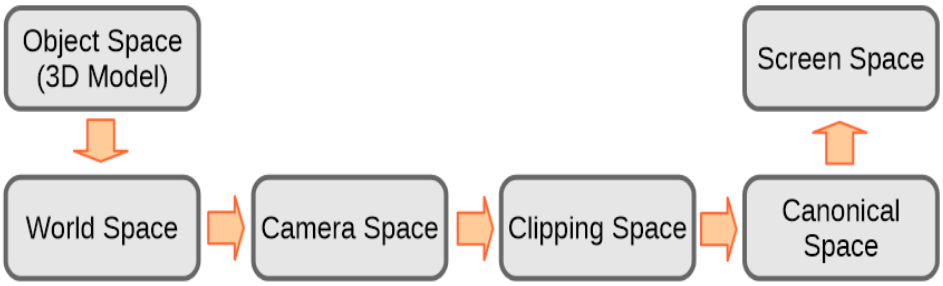


Imagem 1 – Estágios do Pipeline Gráfico.

Diante disso, será explicado cada um deles, compreendendo suas características para posteriormente fazermos as transições entre eles.

1) Espaço do Objeto (Object Space)

No espaço do objeto a cena é criada a partir das primitivas geométricas. Neste espaço não há uma importância de qual sistema de coordenadas foi adotado e o objeto ainda está modelado em espaços infinitos.

Um ponto a destacar é a utilização do triângulo como primitiva geométrica, sobretudo por ser a menor primitiva geométrica capaz de gerar qualquer cena desejada e por definir um único plano, o que facilita em processos futuros que poderão ser adicionados no pipeline para melhorar o desempenho do mesmo.

2) Espaço do Universo (World Space)

No espaço do objeto é feito a transição do sistema de coordenadas do objeto para o sistema de coordenadas 3D. Além disso, ele contém todos os objetos modelados que irão formar a cena.

Desta maneira, é necessário que ocorram as transformações geométricas que irão nos habilitar a: mover, estender e deformar os objetos até que estes atinjam as suas devidas posições.

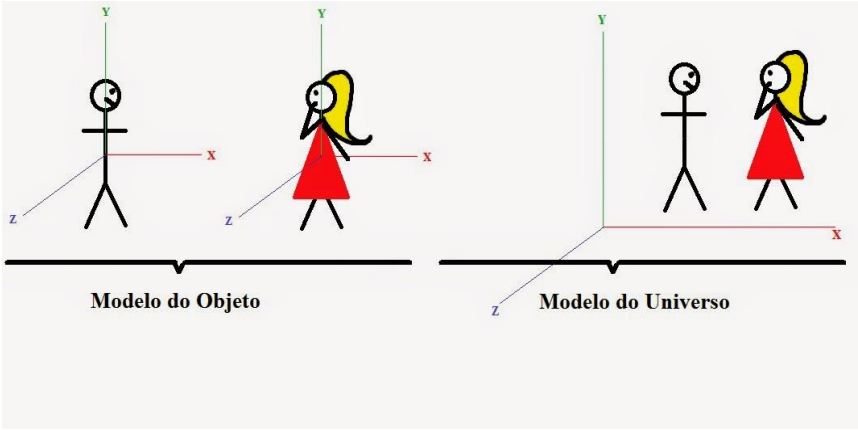


Imagem 2 – Comparação entre os espaços do objeto e do universo.

3) Espaço da Câmera (Camera Space)

No espaço da câmera é determinado o posicionamento da câmera, seu alcance de visualização e qual o tipo de projeção que deverá ser utilizado.

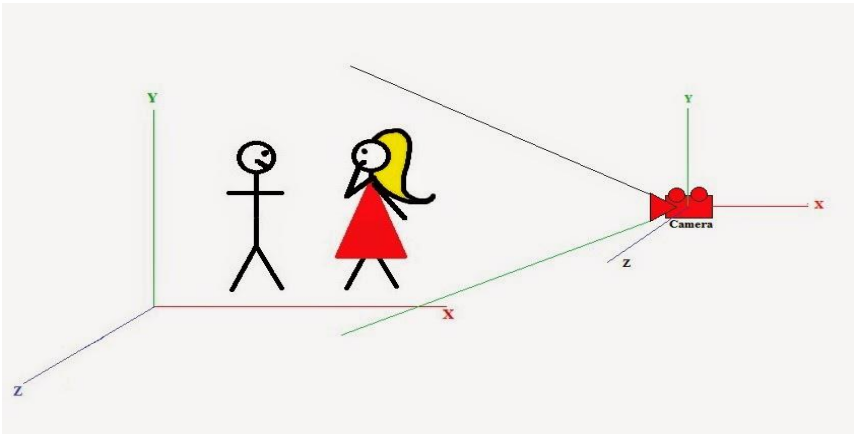


Imagem 3 – Espaço da câmera.

4) Espaço de Recorte (Clipping Space)

Este espaço é muito importante por influenciar diretamente na performance, pois é nele que ocorre a "limpeza" das primitivas. Relembrando o que ocorreu até agora, temos:

- No espaço do Objeto, tínhamos várias primitivas formando uma cena e transformamos essas primitivas para que cheguem até o espaço do Universo.
- Logo depois foi feito o posicionamento da câmera, indicando assim, quais são as primitivas que estarão dentro do espaço de visualização e que futuramente devem exibidas na tela.

No entanto, se não tratarmos as primitivas fora da visualização da câmera, elas passarão por todo o pipeline até serem excluídas. Isso acarretaria num desastre na performance da aplicação. Portanto, é necessário que tais primitivas não passem adiante no pipeline sendo essa a finalidade do espaço de recorte.

Por exemplo, na imagem abaixo deve-se garantir que os três bonecos atrás da câmera não passem adiante do pipeline, já que não serão rasterizados na tela.

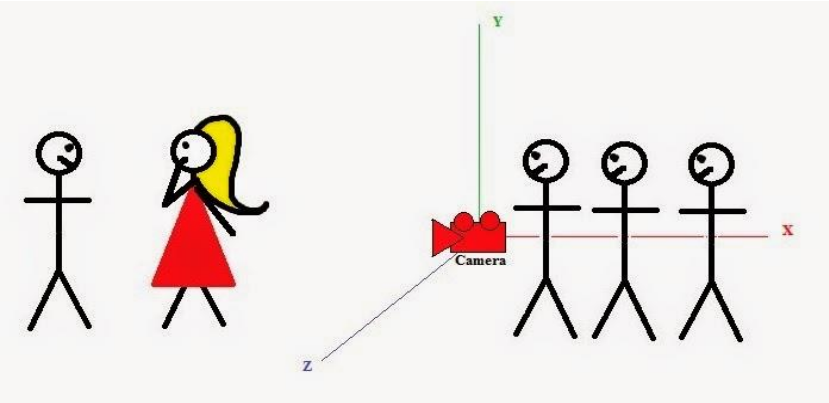


Imagem 4 – Espaço de recorte.

5) Espaço Canônico (Canonical Space)

É importante lembrar que as coordenadas da tela se resumem aos valores de Largura e Altura sempre positivos e que a origem do seu "eixo" é no canto superior esquerdo da tela. Desta maneira, um ponto que se deve levar em consideração é como tratar as coordenadas dos objetos da cena, já que estas podem assumir qualquer valor, ou seja, temos que garantir que toda a cena aparecerá na tela.

Para isso, o espaço canônico é utilizado para facilitar a transformação das coordenadas dos objetos na cena para o espaço de tela. Ele sugere que o ambiente visualizado seja limitado ao volume de um cubo, que será formado pelos vértices entre o intervalo [-1, 1]. Além disso, o cubo também deverá estar posicionado na origem do plano.

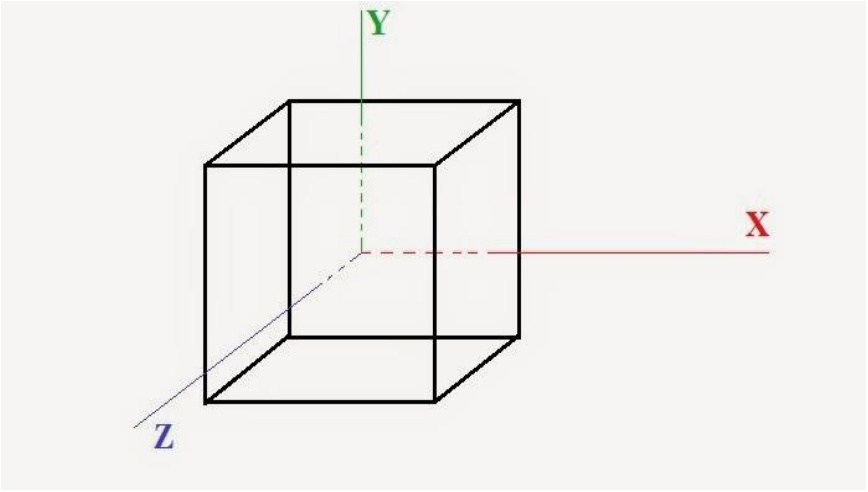


Imagem 5 – Cubo [-1,1] do espaço canônico.

6) Espaço de Tela (Screen Space)

Nessa etapa ocorre o processo de rasterização. Portanto, para implementação do pipeline iremos utilizar as funções de rasterização da tarefa 1.

Matrizes de Transição

Depois de explicado cada um dos estágios do pipeline é preciso saber como transformar as coordenadas do espaço do objeto até o espaço de tela. Para isso, é necessário a utilização de matrizes que serão responsáveis por tais transições. Iremos implementar 4 e são elas: Model, View, Projection e ViewPort.

1) Model (Espaço do Objeto para o Espaço do Universo)

A matriz model é responsável por trazer os vértices do espaço do objeto para o do universo. Inicialmente ela é uma matriz identidade, mas que pode ser aplicada a ela as transformações geométricas.

$$MODEL = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2) View (Espaço do Universo para o Espaço da Câmera)

A matriz *view* é responsável por carregar e modificar os vértices de acordo com as configurações da câmera. Para construirmos a *view* precisa-se de algumas informações da câmera como:

- Posição da Câmera (*Camera Position*) no espaço do Universo, pois ela sempre está na origem no espaço da Câmera. Tal posição é composta de coordenadas x, y e z.

$$camera\_position = (p_x, p_y, p_z)$$

- Direção da Visão (*View Direction*), ou seja, para onde a câmera está olhando no espaço do Universo. Por exemplo, se o objeto a ser visualizado está na origem, a direção de visualização da câmera será (0, 0, 0).

$$view\_direction = (d_x, d_y, d_z)$$

- *Up vector* que geralmente sempre tem valor (0, 1, 0). Ele informa qual é a "parte de cima" da cena, ou seja o Up. Como na maioria das vezes o Up é o eixo Y, esse vetor recebe este valor.

$$Up = (u_x, u_y, u_z)$$

A fim de reposicionar corretamente os vértices dos objetos no espaço do universo vamos identificar os eixos da câmera nesse espaço com as informações dadas pelo usuário criando o plano da câmera.

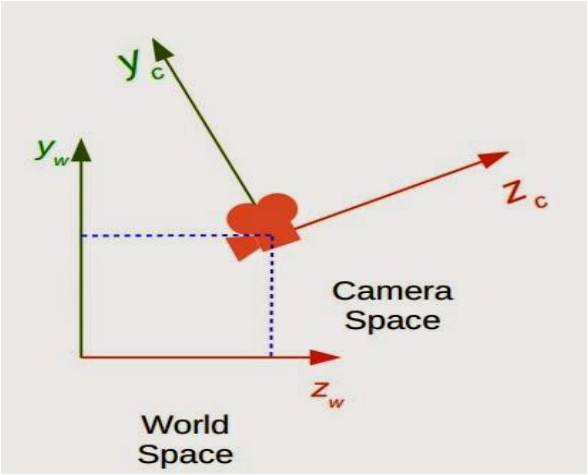


Imagem 6 – Plano da câmera no espaço universo.

Teremos coordenadas x,y e z, calculadas da seguinte maneira.

$$z_c = -\frac{d}{|d|} \qquad x_c = \frac{(Up\ X\ y_c)}{|Up\ X\ y_c|} \qquad y_c = \frac{(z_c\ X\ x_c)}{|z_c\ X\ x_c|}$$

É preciso levar toda a cena incluindo a câmera para origem. Isto é feito através da matris abaixo.

$$T = \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Todo esse processo é necessário para construção da matriz *view*, a qual é calculada da seguinte maneira.

$$VIEW = \begin{pmatrix} x_{cx} & x_{cy} & x_{cz} & 0 \\ y_{cx} & y_{cy} & y_{cz} & 0 \\ z_{cx} & z_{cy} & z_{cz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3) Matriz Projection (Espaço da Câmera para o Espaço Recorte)

Para construir a matriz de projeção é preciso entender o funcionamento do olho humano e como isso está relacionado a distorção perspectiva.

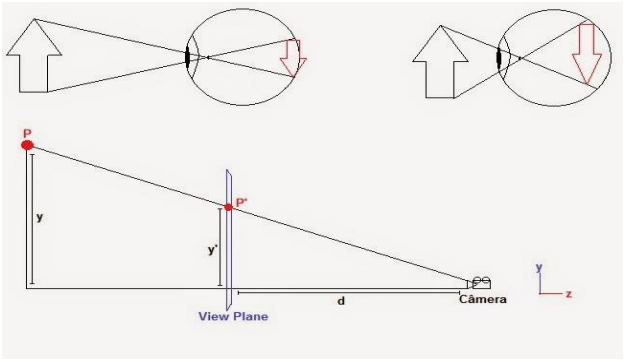


Imagem 7 – Projeção Perspectiva.

- A View Plane seria o plano de visão da câmera.
- O ponto P de um objeto que após sofrer o efeito de distanciamento irá ficar diminuir de tamanho e terá um novo valor P'.
- Y e Y' serão respectivamente, o valor de Y do objeto antes e depois de passar pelo processo.
- d é a distância focal da câmera.

Para construir a matriz de projeção é preciso saber o valor de d. Abaixo encontra-se a matriz.

$$Projection = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 1 \end{pmatrix}$$

4) Divisão por W (Espaço de Recorte para o Espaço Canônico)

Na construção da matriz de projeção é utilizado as coordenadas homogêneas de forma que para voltar ao espaço euclidiano é necessário dividir todas as coordenadas por w, a coordenada.

Após feita tal divisão, teremos os vértices no nosso espaço Canônico.

5) ViewPort (Espaço Canônico para o Espaço de Tela)

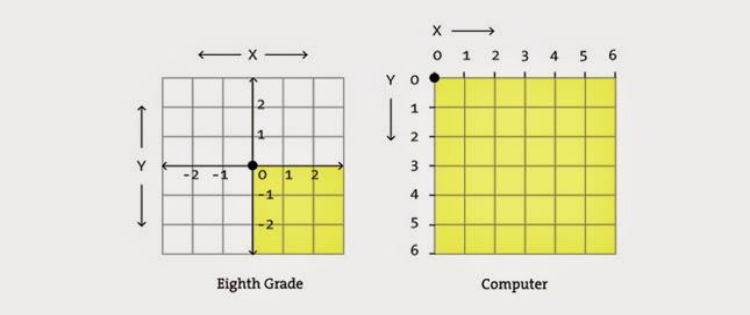


Imagem 8 – comparação da organização das coordenadas.

Para transformar as coordenadas dos objetos para o espaço de tela é preciso fazer um espelhamento da cena para que a mesma não fique invertida. Dessa maneira, utiliza-se a matriz M definida abaixo.

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Após isso, teremos que escalar a cena de forma que ela ocupe a tela. Para isso, utiliza-se os coeficientes de escala com valores W(largura da tela)/2 e H(altura da tela)/2. Dessa forma, monta-se a matriz S definida abaixo.

$$S = \begin{pmatrix} \frac{W}{2} & 0 & 0 & 0 \\ 0 & \frac{H}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Após isso, já tem-se o objeto na posição correta e ocupando a tela em um espaço considerável, porém os vértices que contêm valores negativos não aparecerão na tela (ver imagem 8). Dessa forma, teremos que transladar tais vértices de maneira que eles sejam rasterizados. Com isso, define-se a matriz T abaixo.

$$T = \begin{pmatrix} 1 & 0 & 0 & \frac{W-1}{2} \\ 0 & 1 & 0 & \frac{H-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Para construir a matriz *ViewPort* multiplicaremos as matrizes T, S e M.

ViewPort = T*S*M

Implementação

Para implementar o pipeline foi utilizado a IDE (*Integrated Development Environment)* Code::blocks, o código *obj_loader* fornecido pelo professor via SIGAA, a glm para manipulação de vetores e matrizes e o OpenGl.

Inicialmente foi feito uma alteração no main do *obj_loader,* mais especificamente no *for* da função *display* com o objetivo de, ao ler cada face do arquivo importado, cada vértice dessa face passe pela função *pipeline* que retorna o vetor já em coordenadas de tela. A cada dois vetores transformados em coordenadas de tela é chamada a função drawLine feita na tarefa1 que fará o processo de rasterização. Abaixo tem-se o for mencionado.

```
51      for(int i=0; i<objData->faceCount; i++)
52          {
53              obj_face *o = objData->faceList[i];
54
55              vec4 vertice1 = vec4 (objData->vertexList[o->vertex_index[0]]->e[0], // primeira linha
56                          objData->vertexList[o->vertex_index[0]]->e[1],
57                          objData->vertexList[o->vertex_index[0]]->e[2],
58                          1);
59              vertice1 = pipeline(vertice1);
60
61              vec4 vertice2 = vec4 (objData->vertexList[o->vertex_index[1]]->e[0],
62                          objData->vertexList[o->vertex_index[1]]->e[1],
63                          objData->vertexList[o->vertex_index[1]]->e[2],
64                          1);
65
66              vertice2 = pipeline(vertice2);
67
68              drawLine(vertice1, vertice2, &azul_claro);
69
70
71
72              vec4 vertice3 = vec4 (objData->vertexList[o->vertex_index[1]]->e[0],    // segunda linha
73                          objData->vertexList[o->vertex_index[1]]->e[1],
74                          objData->vertexList[o->vertex_index[1]]->e[2],
75                          1);
```

Imagem 9 – For modificado.

```
76              vertice3 = pipeline(vertice3);
77
78              vec4 vertice4 = vec4 (objData->vertexList[o->vertex_index[2]]->e[0],
79                          objData->vertexList[o->vertex_index[2]]->e[1],
80                          objData->vertexList[o->vertex_index[2]]->e[2],
81                          1);
82              vertice4 = pipeline(vertice4);
83
84              drawLine(vertice3, vertice4, &azul_claro);
85
86
87              vec4 vertice5 = vec4 (objData->vertexList[o->vertex_index[2]]->e[0],    // terceira linha
88                          objData->vertexList[o->vertex_index[2]]->e[1],
89                          objData->vertexList[o->vertex_index[2]]->e[2],
90                          1);
91              vertice5 = pipeline(vertice5);
92
93              vec4 vertice6 = vec4 (objData->vertexList[o->vertex_index[0]]->e[0],
94                          objData->vertexList[o->vertex_index[0]]->e[1],
95                          objData->vertexList[o->vertex_index[0]]->e[2],
96                          1);
97              vertice6 = pipeline(vertice6);
98
99              drawLine(vertice5, vertice6, &azul_claro);
```

Imagem 10 – Continuação do for.

A implementação do pipeline de fato é feita no arquivo pipeline.cpp de acordo com as imagens a seguir.

```
8   vec4 pipeline(vec4 vert){
9
10
11      vec4 vertice = vert;
12      /*****************************************************/
13      /***** Matriz model: Esp. Obj. --> Esp. Univ.  *****/
14      /*****************************************************/
15      mat4 Model = mat4(1,0,0,0,
16                        0,1,0,0,
17                        0,0,1,0,
18                        0,0,0,1); //Matriz model (iniciada com 1.0 na diagonal principal)
19
20
```

Imagem 11 – Construção da matriz *Model*.

```
21      /*************************************/
22      /***** parametros da camera **********/
23      /*************************************/
24      vec3 camera_pos = vec3(2,3,5); // posicao da camera no universo
25      vec3 camera_lookat = vec3(0,0,0); // ponto pra onde a camera esta olhando
26      vec3 camera_up = vec3(0,1,0);
27
28      /*****************************************/
29      /***** sistema ortonormal da camera *****/
30      /*****************************************/
31      vec3 camera_dir = camera_lookat - camera_pos;
32
33      vec3 z_camera = -(normalize(camera_dir));
34      vec3 x_camera = normalize(cross(camera_up, z_camera));
35      vec3 y_camera = cross(z_camera, x_camera);
36
```

Imagem 12 – Construção do sistema da câmera.

```
37  /*************************************************************/
38  /***** Construcao da matriz view: Esp. Univ. --> Esp. Cam. *****/
39  /*************************************************************/
40  mat4 Bt = mat4(x_camera[0], y_camera[0], z_camera[0], 0, // coluna 1
41                 x_camera[1], y_camera[1], z_camera[1], 0, // coluna 2
42                 x_camera[2], y_camera[2], z_camera[2], 0, // coluna 3
43                 0,0,0,1);                                 // coluna 4
44
45  mat4 T1 = mat4(1,0,0,0,
46                 0,1,0,0,
47                 0,0,1,0,
48                 -camera_pos[0], -camera_pos[1], -camera_pos[2], 1);
49
50  mat4 View = Bt * T1;
51
52  /*************************************************************************/
53  /***** Construcao da matriz de ModelView: Esp. Obj. --> Esp. Cam. *****/
54  /*************************************************************************/
55  mat4 ModelView = View * Model;
56
```

Imagem 13 – Construção da Matriz *View*.

```
57  /*************************************************************************/
58  /***** Construcao da matriz de Projecao: Esp. Cam. --> Esp. Recorte *****/
59  /*************************************************************************/
60  int d = 2;// distancia do centro de projecao para o viewplane
61
62  mat4 Projection = mat4(1, 0, 0, 0,          //coluna 1
63                         0, 1, 0, 0,          //coluna 2
64                         0, 0, 1, -(1.0/d),   //coluna 3
65                         0, 0, d, 0);         //coluna 4
66
67  /*****************************************************************************/
68  /***** Construcao da matriz ModelViewProjection: Esp. Obj. --> Esp. Recorte *****/
69  /*****************************************************************************/
70
71  mat4 ModelViewProj = Projection * ModelView;
72
73
74  /*****************************************************************************
75   Aplicao da matriz ModelViewProjection: Esp. Obj. --> Esp. Recorte
76  *****************************************************************************/
77
78  vertice = ModelViewProj * vertice;
79
```

Imagem 14 – Construção da *Projection*.

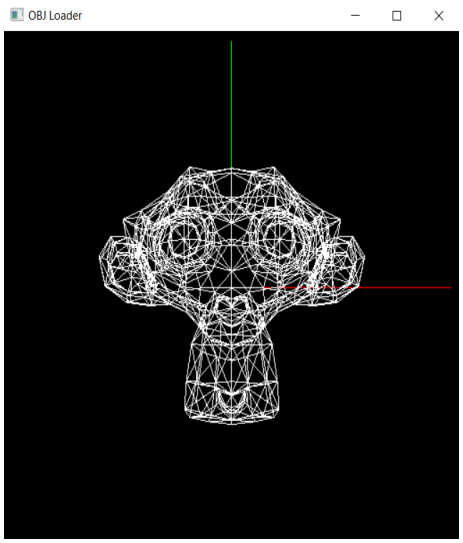
```
    /*****************************************************************************
    # Homogeneizacao (divisao por W).
    # Este passo leva os vertices normalmente para o espaco canonico.
    # Neste caso, como a matriz de projecao eh muito simples, o resultado
    # da homogeneizacao sao os vertices em um espaço que nao eh exatamente o
    # canonico. Porem, apesar de nao ser o espaco canonico, a distorcao perspectiva
    # estara presente.
    *****************************************************************************/

    vertice[0] = vertice[0]/vertice[3];
    vertice[1] = vertice[1]/vertice[3];
    vertice[2] = vertice[2]/vertice[3];
    vertice[3] = vertice[3]/vertice[3];

    /*****************************************************************************
    # Conversao de coordenadas do espaco canonico para o espaco de tela.
    *****************************************************************************/

    mat4 S1 = mat4(1, 0, 0, 0,
                   0, -1, 0, 0,
                   0, 0, 1, 0,
                   0, 0, 0, 1);

    mat4 T = mat4 (1, 0, 0, 0,
                   0, 1, 0, 0,
                   0, 0, 1, 0,
                   1, 1, 0, 1);

    float w = 512;
    float h = 512;

    mat4 S2 = mat4(w/2, 0, 0, 0,
                   0, h/2, 0, 0,
                   0, 0, 1, 0,
                   0, 0, 0, 1);


    mat4 Viewport = S2 * T * S1;

    vertice = Viewport * vertice;


return vertice;
}
```

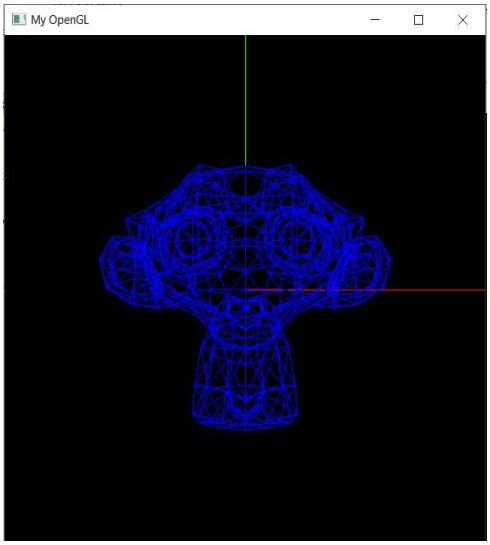
Imagem 15 – Construção da matriz *ViewPort* e retornando o vetor em coordenadas de tela.

Algumas das dificuldades encontradas consistiram na alteração do código da tarefa1 para que as funções desenvolvidas anteriormente recebessem e manipulassem corretamente os vetores da biblioteca glm, bem como, fazer o *debug* onde algumas coordenadas estavam sendo divididas por zero.

Resultados

Abaixo temos os resultados obtidos na implementação do pipeline gráfico (em azul) com o pipeline do opengl (em branco). Para isso a posição da câmera foi alterada para (0,0,4) olhando na direção (0,0,0) com o vetor *up* (0,1,0).





Referências

- http://letslearnbits.blogspot.com/2014/11/icgt2-pipeline-grafico.html
- http://matheuspraxedescg.blogspot.com/2016/10/pipeline-grafico.html
- Notas de aula do Prof. Christian Azambuja Pagot