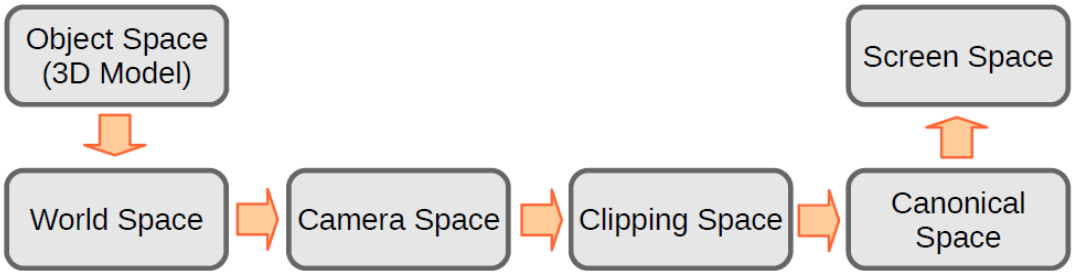
INTRODUÇÃO

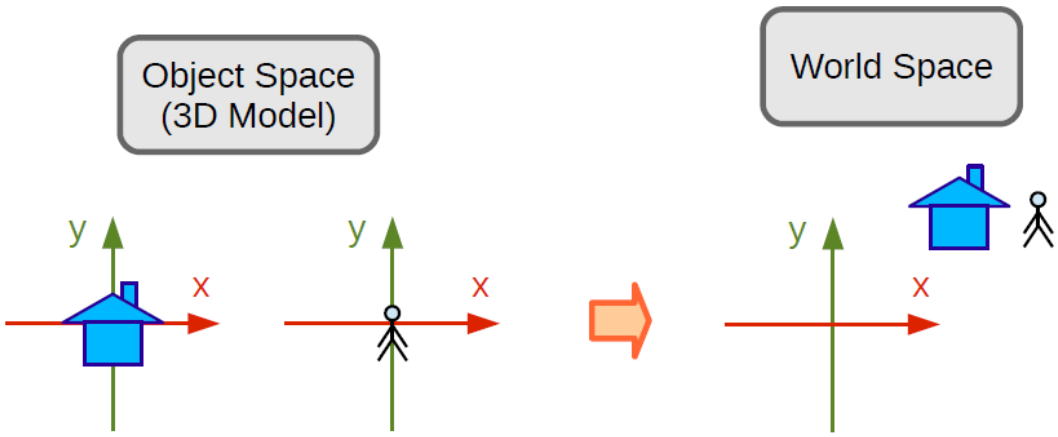
O objetivo do trabalho é implementar um pipeline gráfico completo, obedecendo todas as suas etapas. O trabalho foi codificado em C++.

O pipeline gráfico consiste na série de etapas necessárias para que diversos objetos possam ser levados do espaço objeto para uma cena completa e que tal cena possa ser exibida na tela com a devida fidelidade.



ETAPAS DO PIPELINE

Espaço do objeto -> Espaço do universo

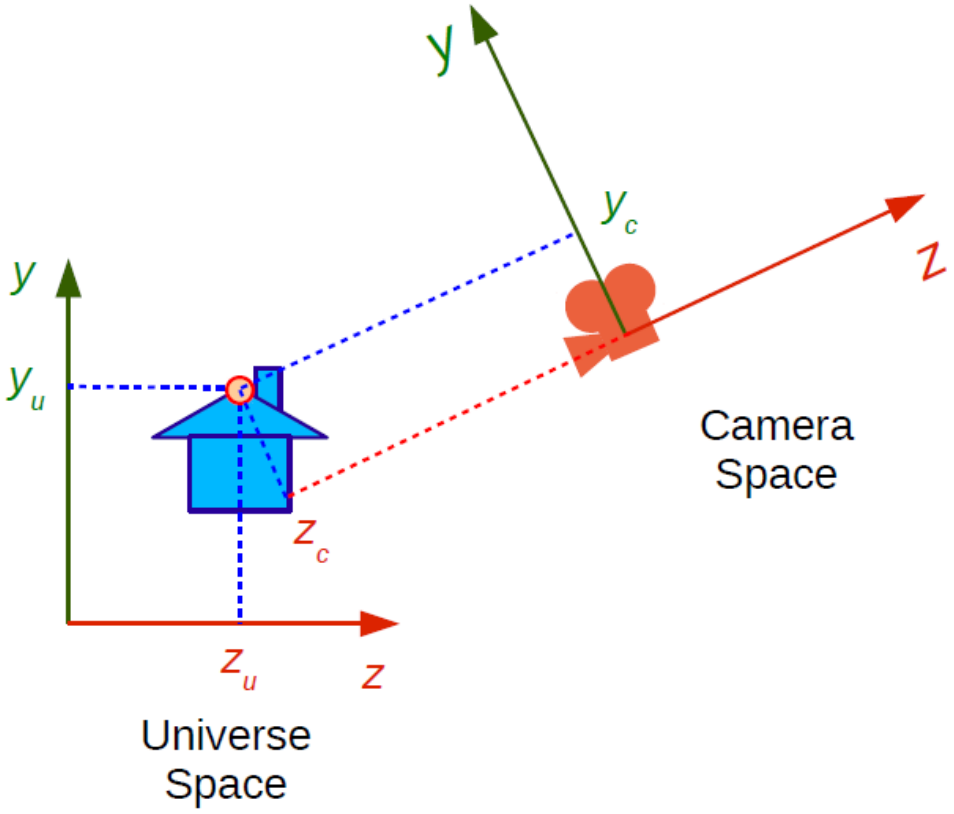


A primeira etapa do pipeline visa a transferência de um ou mais objetos para o eixo de coordenadas da cena, dessa forma, se faz necessária a utilização das operações básicas aprendidas em sala de aula, tais como: translação, rotação e escala.

Com essas três operações é possível “mover” e rotacionar cada objeto em nosso espaço, com essa necessidade de manipulação do objeto fora criadas as três funções abaixo, fundamentais para o pipeline gráfico:

```
void Pipeline::setRotation(float angle, float x, float y, float z) {
    matrixModel = glm::rotate(matrixModel, angle, glm::vec3(x, y, z));
}

void Pipeline::setTranslation(float x, float y, float z) {
    matrixModel =  glm::translate(matrixModel, glm::vec3(x, y, z));
}

void Pipeline::setScale(float x, float y, float z) {
    matrixModel = glm::scale(matrixModel,glm::vec3(x, y, z));
}
```

Espaço do universo -> Espaço da câmera

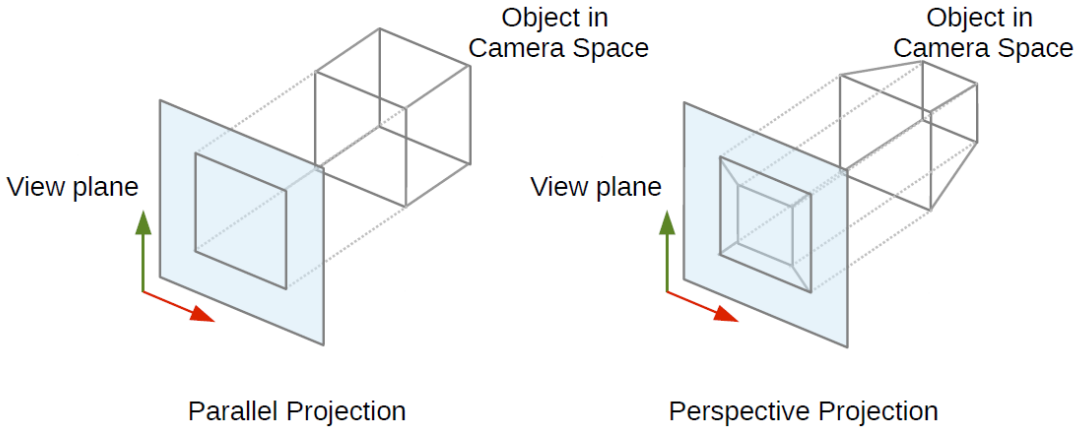


Nesse passo a cena é “mexida” novamente para que se adapte a um eixo de coordenadas o qual a visão do usuário estará no centro. Para isso devemos adaptar nosso eixo de coordenadas atual (do espaço do universo) para o eixo de coordenadas da câmera.

Dito isso, será necessária a criação de 3 vetores, sendo eles: câmera position (contendo a posição da câmera no espaço universo), view direction (contendo a direção a qual a câmera aponta), up vector (posição da câmera fixa em um determinado eixo). Agora podemos montar nossa matriz view (que nos trará o espaço da câmera).

```
glm::mat4 Pipeline::createMatrixView(const float zDistance) {
    glm::vec3 cameraPosition(0.0f, 0.0f, zDistance);
    glm::vec3 viewDirection(0.0f, 0.0f, 0.0f);
    glm::vec3 upVector(0.0f, 1.0f, 0.0f);

    glm::vec3 Zc = glm::normalize(cameraPosition - viewDirection);
    glm::vec3 Xc = glm::normalize(glm::cross(upVector, Zc));
    glm::vec3 Yc = glm::normalize(glm::cross(Zc, Xc));

    glm::mat4 B = glm::mat4(1.0f);
    B[0] = glm::vec4(Xc, 0.0f);
    B[1] = glm::vec4(Yc, 0.0f);
    B[2] = glm::vec4(Zc, 0.0f);

    glm::mat4 T = glm::mat4(1.0f);
    T[3] = glm::vec4(-cameraPosition, 1.0f);

    return glm::transpose(B) * T;
}
```

Espaço da câmera -> Espaço de recorte



Nessa etapa do pipeline é importante atentarmos que embora tenhamos o objetivo de mostrar a cena em uma tela, tal cena sendo 3d, precisamos nos lembrar a importância para que essa cena passe uma noção de profundidade para o usuário, sendo assim, é necessária a presença do espaço de recorte para que cada ponto da cena, visualizada pela câmera, sofra uma distorção perspectiva.

Tal distorção faz com que objetos mais próximos fiquem com tamanhos maiores e objetos mais distantes fiquem com tamanhos menores, para isso utilizaremos a matriz de projeção.

```
glm::mat4 Pipeline::createMatrixProjection(const float viewPlaneDistance) {
    glm::mat4 matrixProjection = glm::mat4(1.0f);
    matrixProjection[2].w = -1 / viewPlaneDistance;
    matrixProjection[3].z = viewPlaneDistance;
    return matrixProjection;
}
```

Espaço de recorte -> Espaço canônico -> Espaço de tela

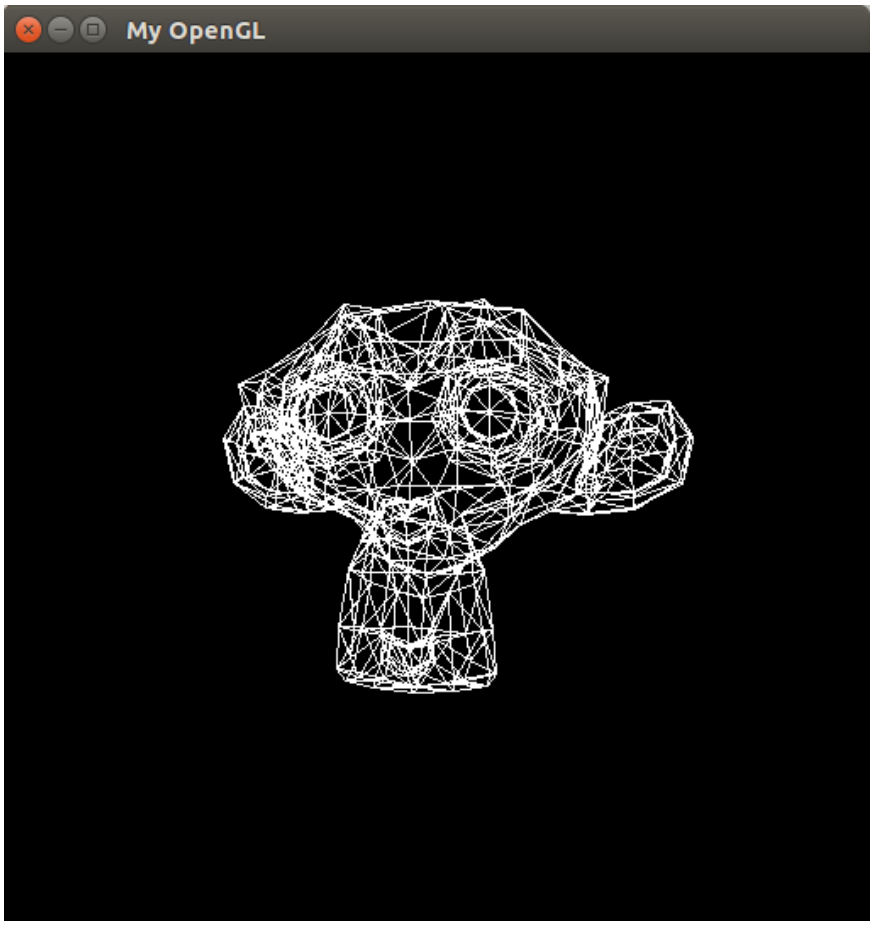
No espaço canônico é necessário usar a técnica de homogeneização para que os vértices sejam movidos para o mesmo. Para aplicar a homogeneização é preciso dividir os componentes do vértice pela sua coordenada homogênea. Ao término da passagem todos os vértices da cena que serão visíveis estarão com valores entre -1 e 1 no sistema de coordenadas do espaço canônico.

Já para passar do espaço canônico para o espaço de tela há a necessidade de preparação de cada vértice seja preparado para o espaço de tela, a matriz viewport acaba cuidado disso multiplicando os vértices do espaço canônico. Como podemos ver logo abaixo, a matriz viewport é formada por duas matrizes de escala e uma de translação:

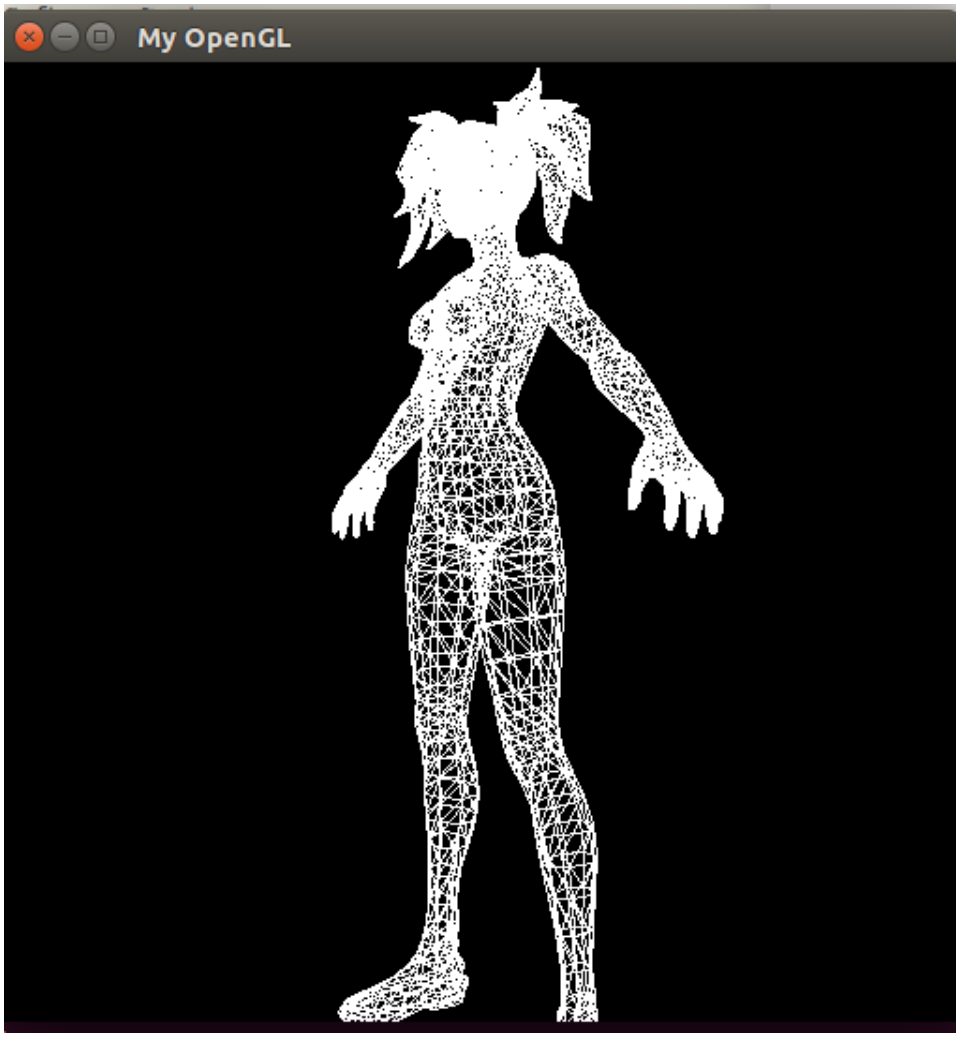
```
void Pipeline::toScreenSpace(glm::mat4& modelViewProjection, glm::vec4& firstVertex, glm::vec4& secondVertex, glm::vec4& thirdVertex) {
    glm::mat4 invert;
    invert[1].y = -1;

    glm::mat4 translate(1.0f);
    translate[3] = glm::vec4(1.0f, 1.0f, 0.0f, 1.0f);

    glm::mat4 scale(1.0f);
    scale[0].x = (IMAGE_WIDTH-1) * 0.5f;
    scale[1].y = (IMAGE_HEIGHT-1) * 0.5f;

    glm::mat4 screenMatrix = scale * translate * invert;

    firstVertex = modelViewProjection * firstVertex;
    secondVertex = modelViewProjection * secondVertex;
    thirdVertex = modelViewProjection * thirdVertex;

    firstVertex = screenMatrix * firstVertex / firstVertex.w;
    secondVertex = screenMatrix * secondVertex / secondVertex.w;
    thirdVertex = screenMatrix * thirdVertex / thirdVertex.w;
}
```

RESULTADOS

Para a amostra do código em funcionamento foram utilizadas duas imagens, a primeira é a imagem do macaco fornecida para exemplificação e a segunda se trata da personagem Mercy do jogo OverWatch.



Vídeo da execução do código com o macaco -
https://www.youtube.com/watch?v=JBwhi0MPsJc



REFERÊNCIAS:

- Slides de aula;
- https://www.cgtrader.com/ (modelo .obj da Mercy);
- Computer Graphics: Principles and Practice – John Foley;