Aluno: Renilton Ribeiro de Oliveira Matrícula: 11328395

Atividade 2

Introdução

Esse é um relatório sobre a segunda atividade de computação gráfica, implementação de um pipeline gráfico, essa disciplina é ministrada pelo professor Christian Pagot, na UFPB, no período 2020.2. Para esse trabalho foi utilizada a linguagem de programação Javascript juntamente com o framework disponibilizado pelo professor que simula o acesso à memória de vídeo.

Bom, O pipeline gráfico basicamente é uma sequência de passos necessários para transformar uma descrição geométrica/matemática de uma cena 3D em uma descrição visual na tela 2D. A imagem final é obtida por meio da rasterização das primitivas projetadas na tela, cada passo do pipeline gráfico consiste de uma transformação geométrica de um sistema de coordenadas (espaço) para outro

Estratégia

Optei pela por dedicar tempo a entender com é o funcionamento de cada transformação para que pudesse passar para um algoritmo, fui fazendo por partes e testando através de valores gerados no console, ou visualizações de esboço do desenho em cada transformação, decidi utilizar a biblioteca three.js para me auxiliar em caso onde precisava fazer norma, transposição, multiplicação, subtração e aplicar transformações nos vetores, na parte de rasterização apenas utilizei o código do primeiro trabalho.

Resultados

Transformação: Espaço do Objeto → Espaço do Universo

Esta etapa do pipeline é responsável por transformar vértices, originalmente descritos no espaço do objeto, para o espaço do universo. Isto é feito através da multiplicação destes vértices por uma matriz denominada matriz de modelagem.

```
let m_model = new THREE.Matrix4();

m_model.set(1.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 1.0);

for (let i = 0; i < 8; ++i)
  vertices[i].applyMatrix4(m_model);
```

Transformação: Espaço do Objeto → Espaço do Universo

Esta etapa é responsável por transformar vértices do espaço do universo para o espaço da câmera. Isto é feito através da multiplicação dos vértices por uma matriz denominada matriz de visualização, A matriz de visualização é composta por uma translação e uma rotação. Esta translação e esta rotação são definidas a partir das informações da câmera.

```
let cam_pos = new THREE.Vector3(1.3, 1.7, 2.0);
let cam_look_at = new THREE.Vector3(0.0, 0.0, 0.0);
let cam_up = new THREE.Vector3(0.0, 1.0, 0.0);
```

Desses vetores, pode-se obter a direção da câmera subtraindo-se o vetor look_at do cam_pos. Após isso descobrimos as coordenadas x, y e z da câmera através da norma e do produto vetorial dos vetores encontrados.

```
let cam_dir = new THREE.Vector3().subVectors(cam_look_at, cam_pos);

let z_cam = cam_dir.normalize().negate();
let x_cam = new THREE.Vector3().crossVectors(cam_up, z_cam).normalize();
let y_cam = new THREE.Vector3().crossVectors(z_cam, x_cam).normalize();
```

Após obter os vetores de coordenadas da câmera, podemos criar uma matriz_bt com as coordenadas x, y e z em cada ponto da câmera, dessa forma ela já é a transposta.

```
let m_bt = new THREE.Matrix4();

m_bt.set(1.0, 0.0, 0.0, 0.0,
         0.0, 1.0, 0.0, 0.0,
         0.0, 0.0, 1.0, 0.0,
         0.0, 0.0, 0.0, 1.0);

m_bt.set(x_cam.getComponent(0), x_cam.getComponent(1), x_cam.getComponent(2), 0.0,
         y_cam.getComponent(0), y_cam.getComponent(1), y_cam.getComponent(2), 0.0,
         z_cam.getComponent(0), z_cam.getComponent(1), z_cam.getComponent(2), 0.0,
         0.0, 0.0, 0.0, 1.0);
```

Finalizando assim o espaço da câmera a partir da matriz view, tida como a multiplicação entre a matriz de translação m_t da posição da câmera, com a matriz transposta de m_bt. Multiplicando-se a model pela view temos a matriz m_view.

```
let m_t = new THREE.Matrix4();

m_t.set(1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0);

let t = cam_pos.negate();
//Translação da câmera

let cam_trans = new THREE.Matrix4();
m_t.set(1.0, 0.0, 0.0, -t.getComponent(0),
        0.0, 1.0, 0.0, -t.getComponent(1),
        0.0, 0.0, 1.0, -t.getComponent(2),
        0.0, 0.0, 0.0, 1.0);

// Constrói a matriz de visualização 'm_view' como o produto
//  de 'm_bt' e 'm_t'.
let m_view = m_bt.multiply(m_t);

for (let i = 0; i < 8; ++i)
  vertices[i].applyMatrix4(m_view);
```

Transformação: Espaço da Câmera → Espaço de Recorte

Esta etapa é responsável por transformar vértices do espaço da câmera para o espaço de recorte. Após criação da matriz de projeção adiciona uma variável d que representa a distância entre o objeto e a view, quanto maior valor de d, mais próximo o objeto estará, consequentemente quanto menor for, mais distante estará, definimos d como 1

```
let m_projection = new THREE.Matrix4();

m_projection.set(1.0, 0.0, 0.0, 0.0,
                 0.0, 1.0, 0.0, 0.0,
                 0.0, 0.0, 1.0, 0.0,
                 0.0, 0.0, 0.0, 1.0);

let d = 1;
m_projection.set(1.0, 0.0,  0.0, 0.0,
                 0.0, 1.0,  0.0, 0.0,
                 0.0, 0.0,  1.0,  d,
                 0.0, 0.0, -1/d, 0.0);

for (let i = 0; i < 8; ++i)
  vertices[i].applyMatrix4(m_projection);
```

Espaço de Recorte ⟶ Espaço Canônico

Esta etapa é responsável por transformar pontos do espaço de recorte para o espaço canônico. Isto é feito a, dividir as coordenadas dos vértices no espaço de recorte pela sua coordenada homogênea.

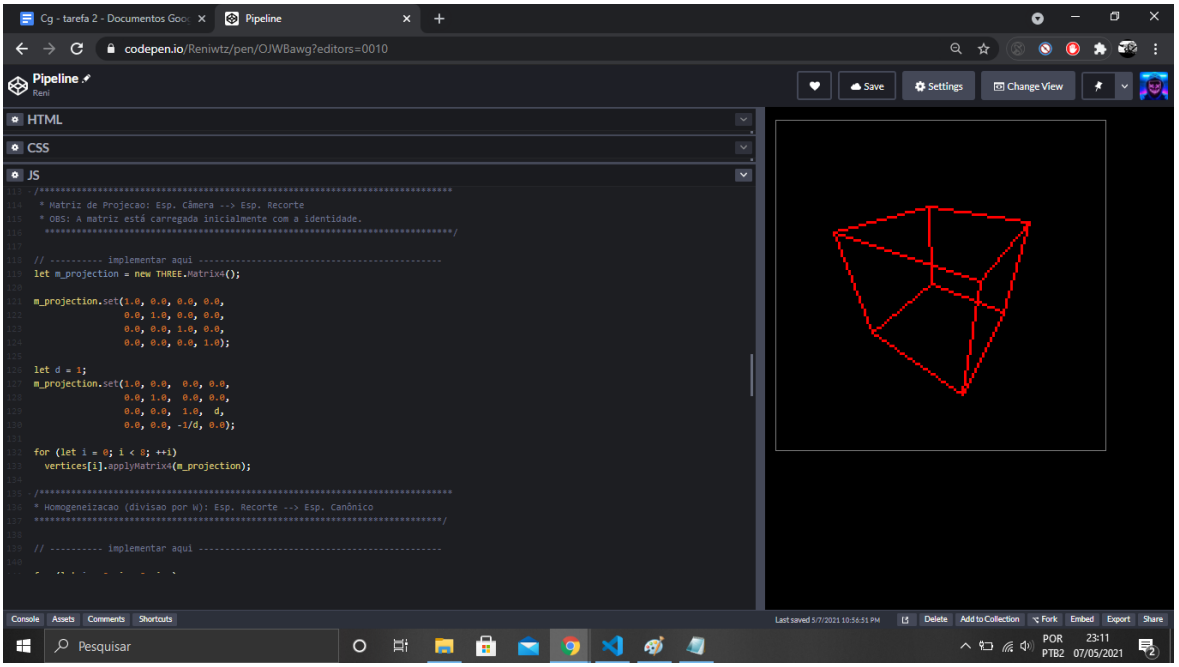
```
for (let i = 0; i < 8; i++)
  vertices[i].divideScalar(vertices[i].w);
```

Transformação: Espaço Canônico ⟶ Espaço de Tela

Esta etapa é responsável por transformar pontos do espaço canônico para o espaço de tela. Isto é feito através da multiplicação dos vértices por uma matriz específica que envolve escala e translação. após obter os vértices no espaço canônico, ajustamos os valores obtidos para o tamanho da tela desejada por meio de uma escala que expande o objeto para as dimensões da tela e uma translação, depois aplicamos aos vértices por meio da viemport.

```
let m_viewport = new THREE.Matrix4();

m_viewport.set(1.0, 0.0, 0.0, 0.0,
               0.0, 1.0, 0.0, 0.0,
               0.0, 0.0, 1.0, 0.0,
               0.0, 0.0, 0.0, 1.0);


let scale = new THREE.Matrix4();
scale.set(128/2, 0.0, 0.0, 0.0,
          0.0, 128/2, 0.0, 0.0,
          0.0,  0.0, 1.0, 0.0,
          0.0,  0.0, 0.0, 1.0);


let translate = new THREE.Matrix4();
translate.set(1.0, 0.0, 0.0, 1.0,
              0.0, 1.0, 0.0, 1.0,
              0.0, 0.0, 1.0, 0.0,
              0.0, 0.0, 0.0, 1.0);


let viewport = scale.clone().multiply(translate);

for (let i = 0; i < 8; ++i) {
  vertices[i].applyMatrix4(viewport);
}
```

Abaixo podemos ver o resultado final:



Dificuldade e Melhorias Possíveis

Tive dificuldade em entender como eram feitas as transformações em cada passo, na implementação do código pela falta de experiência no uso da biblioteca three.js e o próprio javascript.

O Trabalho poderia ser refeito em outra linguagem que o deixasse mais otimizado, poderia ter sido feito o desenvolvimento de funções que permitisse rotacionar o cubo em tempo de execução, poderia ter sido feito preenchimento da figura (cubo), além disso, no meu código com certeza deve haver coisas que poderiam ter sido feitas de forma mais eficaz por falta de experiência de minha parte.

Referências

1. Notas de aula do Prof. Christian Azambuja Pagot .
2. https://threejs.org/docs/index.html#api/en/
3. https://cgworksblog.wordpress.com/2016/05/07/trabalho-02-pipeline-grafico/
4. http://matheuspraxedescg.blogspot.com/2016/10/pipeline-grafico.html
5. http://pt.slideshare.net/thild/computao-grfica-transformaes-geomtricas-no-plano-e-no-espao
6. https://cflavs.wordpress.com/2016/05/06/cgt2-pipeline-grafico/

Link para o código

https://codepen.io/Reniwtz/pen/OJWBawg