Universidade Federal da Paraíba
Centro de Informática

# Introdução à Computação Gráfica
# Atividade Prática 2 - Implementação do Pipeline Gráfico

Francisco Siqueira Carneiro da Cunha Neto[1]
20190029015
[1]francisconeto@eng.ci.ufpb.br

14 de outubro de 2021

## 1 Atividade Desenvolvida

Nesta atividade foi implementado o *Pipeline* Gráfico na linguagem de programação JavaScript. Foi usada a biblioteca ThreeJS [1] para realizar operações de álgebra linear, o algoritmo de Bresenham [2] implementado na Atividade Prática 1 para rasterizar os pontos obtidos ao final do *pipeline* , e partes do *framework* disponibilizado em https://codepen.io/ICG-UFPB/pen/bGWmQpN. A implementação foi feita principalmente em uma função `GraphicsPipeline()` que, dado um objeto representado por uma lista de vértices no espaço do objeto e uma matriz de transformação, retorna outra lista de vértices que representa esse mesmo objeto no espaço de tela. Também foram implementadas funções que geram algumas matrizes de transformação, bem como uma função para rasterizar os vértices obtidos por `GraphicsPipeline()`, tendo como parâmetros adicionais uma cor e as arestas do objeto.

O código fonte também pode ser encontrado em seus repositórios no GitHub e CodePen.

## 2 Estrutura

O *framework* fornecido foi modificado para dar lugar a uma nova estrutura. O código pertinente às matrizes do *pipeline* foi movido para dentro da função `GraphicsPipeline()` (cujo desenvolvimento é descrito na secção 3). As variáveis com parâmetros relacionados à câmera foram mantidas fora desta função, em conjunto com outras variáveis criadas para armazenar os parâmetros de altura e largura da tela e da distância entre o centro de projeção e o plano de projeção. Funções *set* também foram criadas para permitir a definição desses parâmetros.

Por último, a função `Render()` foi definida. Ela recebe como parâmetros a lista de vértices e arestas da geometria, uma matriz de transformação para esta e a cor em que ela deve ser renderizada. Internamente, `Render()` usa `GraphicsPipeline()` (informando a lista de vértices e a matriz de transformação) para obter as coordenadas dos vértices no espaço de tela, e então rasteriza (na cor informada) linhas entre os dois vértices de cada uma das arestas, usando as coordenadas obtidas e o algoritmo de Bresenham [2] implementado na Atividade Prática 1.

## 3 O Pipeline

O *pipeline* foi então implementado dentro da função `GraphicsPipeline()`, de forma que, para cada um de seus estágios, a operação apropriada fosse realizada sobre todos os vértices da geometria informada. O desenvolvimento de cada um dos estágios é descrito a seguir.

A **Matriz *Model*** deve refletir as transformações que se deseja aplicar no objeto no espaço do universo. Como essas transformações devem ser informadas pelo usuário como parâmetro de `GraphicsPipeline()` na forma de uma matriz de transformação, a Matriz *Model* simplesmente assume os valores da matriz do parâmetro.

Para construir a **Matriz *View***, é necessário conhecer a base do espaço da câmera. Esta base é derivada usando funções da biblioteca ThreeJS que realizam operações de álgebra linear. O eixo $Z$ é obtido pela subtração dos vetores *look at* e de posição da câmera, e sua subsequente multiplicação por $-1$ e normalização; o eixo $X$ é então obtido através do produto vetorial entre o vetor *up* da câmera e o eixo $Z$ de sua base, normalizado; e o eixo $Y$ pelo produto vetorial entre os eixos $Z$ e $X$, normalizado. Conhecendo essa base, é gerada uma matriz de mudança de base $B^T$, e uma matriz de translação $T$ que desloca a origem do espaço de acordo com a posição da câmera. A Matriz *View* é definida então como $B^T.T$.

A **Matriz de Projeção** é definida diretamente, de acordo com a matriz derivada na Aula 14 [3]. O valor de $d$ é 1 por padrão, mas pode ser definido pelo usuário através da função `set_dist_projection_plane()`.

Para a **Homogeneização**, todos os vértices são multiplicados por $1/w$, ou seja, sofreram uma divisão por sua coordenada homogênea.

A **Matriz *Viewport*** é encontrada realizando uma translação seguida por uma escala, que tem como parâmetros a altura e largura da tela. Como a matriz de translação é sempre a mesma, não é necessário multiplicar essas duas em tempo de execução. Dessa forma, a matriz resultante pôde ser calculada

previamente com auxilio de uma calculadora de matrizes e definida diretamente no código.

O método `round()` é finalmente aplicado em todos os vértices antes de serem retornados, para possibilitar o seu mapeamento em pixels (que inerentemente possuem apenas coordenadas inteiras).

### 3.1 O Cubo

Após a implementação do pipeline, os parâmetros foram configurados para aqueles informados na atividade, e o cubo de exemplo foi renderizado. A imagem gerada é exibida na Figura 1.

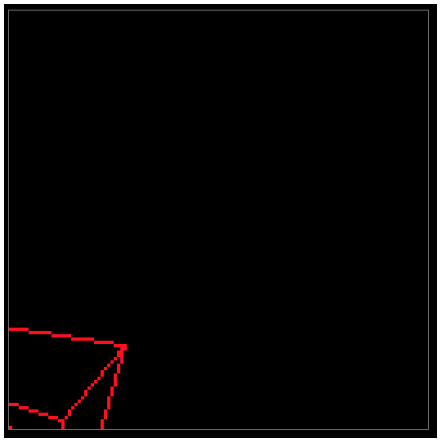


**Figura 1:** *Primeiro render do cubo de exemplo.*

Como se pode observar, a renderização não foi realizada corretamente. O problema estava na Matriz *Viewport*, cuja multiplicação entre as matrizes de translação e escala foi inicialmente realizada na ordem incorreta, resultando em uma matriz imprecisa. Corrigindo isso, obteve-se a renderização da Figura 2.

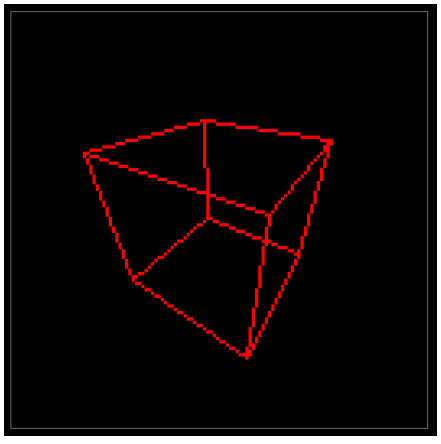


**Figura 2:** *Cubo de exemplo renderizado corretamente.*

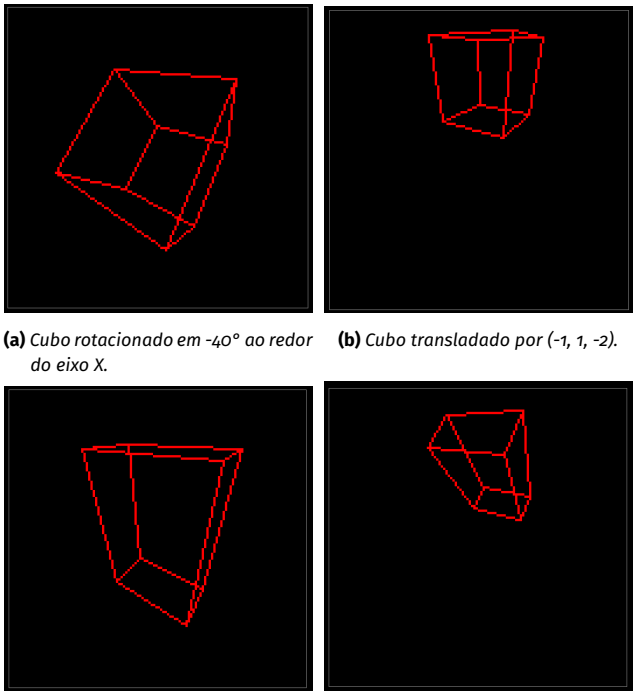
## 4 Transformações

Adicionalmente, foram criadas três funções que retornam matrizes de transformação. Essas matrizes são construídas de acordo com o capítulo 5 do livro *3D Math Primer for Graphics and Game Development* [4]. Cada uma dessas funções foi testada aplicando as transformações ao cubo de exemplo, mostrado sem nenhuma transformação na Figura 2.

A primeira delas, `RotationMatrix()`, retorna uma matriz de rotação ao redor dos eixos $X$, $Y$ ou $Z$, tomando como parâmetros o ângulo da rotação e sobre qual dos eixos ela deve ser realizada. A matriz retornada para uma rotação de -40° ao redor do eixo $X$, quando aplicada no cubo de exemplo, leva a renderização exibida na Figura 3a.

`TranslationMatrix()` retorna uma matriz de translação a partir de valores $x$, $y$ e $z$ informados, e `ScaleMatrix()` retorna uma matriz de escala a partir de parâmetros análogos. O cubo de exemplo transladado por $(-1, 1, -2)$ é exibido na Figura 3b, enquanto a Figura 3c mostra o cubo de exemplo com uma escala de $(1, 1.5, 0.5)$.

Finalmente, a Figura 3d mostra o cubo exemplo após a aplicação de todas as três transformações sucessivamente.



**(a)** *Cubo rotacionado em -40° ao redor do eixo X.*

**(b)** *Cubo transladado por (-1, 1, -2).*

**(c)** *Cubo com escala de (1, 1.5, 0.5).*

**(d)** *Cubo com rotação, translação e escala.*

**Figura 3:** *Cubo com transformações aplicadas.*

## 5 Demonstração

Para a geometria adicional, escolheu-se renderizar um octaedro. Os vértices que compõem um octaedro centralizado na origem foram obtidos utilizando a ferramenta Geogebra 3D Calcultor, e são eles: $(-1, 0, 0)$, $(0, 1, 0)$, $(1, 0, 0)$, $(0, -1, 0)$, $(0, 0, 1)$, $(0, 0, -1)$.

A distância entre o centro e o plano de projeção foi definida como 1, e os parâmetros da câmera foram configurados para:

- *Look at* – $(0, 0, 0)$;
- Posição – $(0, 0, 2)$;

- *Up* – $(0, 1, 0)$.

Contudo, a renderização dessa geometria inicialmente falhou. Acontece que o código do *framework* utilizado definia *for loops* sobre os vértices com apenas 8 iterações, a quantidade de vértices do cubo de exemplo. Foi apenas necessário mudar esse valor para `vertices.length` e a renderização foi bem sucedida. Ela pode ser observada na Figura 4.

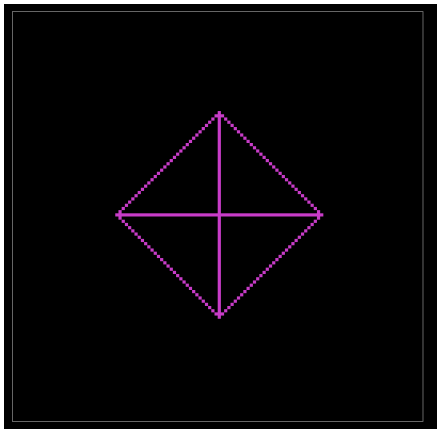
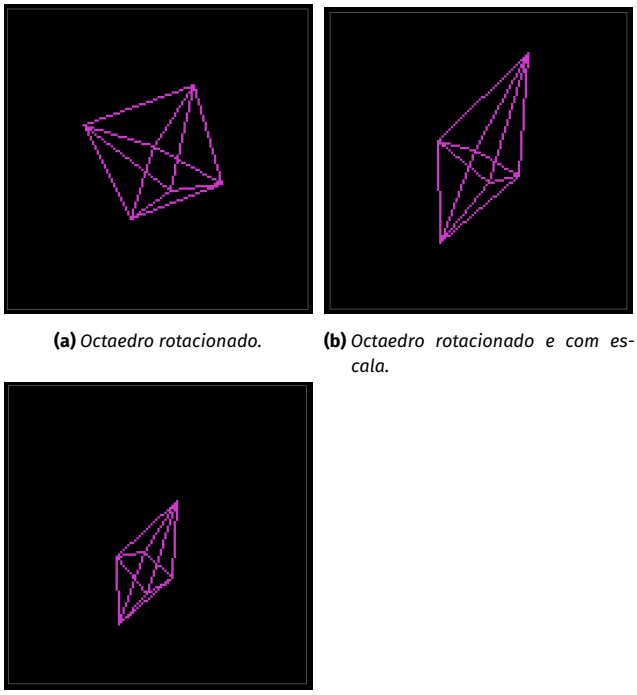


**Figura 4:** *Primeiro render do octaedro.*

O próximo passo foi aplicar sucessivas transformações na geometria, até obter um resultado final. Essas transformações estão expostas na Figura 5. Primeiro, foram realizadas três rotações na geometria, uma de 25° ao redor do eixo $Z$, seguida de uma de -10° ao redor do eixo $X$ e uma de -10° ao redor do eixo $Y$, a geometria após essas rotações é mostrada na Figura 5a. Logo após, aplicou-se uma escala de $(0.6, 1.4, 0.8)$ no octaedro, sua aparência nesse ponto pode ser vista na Figura 5b. O octaedro então sofreu uma translação de $(0.3, -0.6, -0.9)$, visível na Figura 5c.

Finalmente, foi aplicada outra escala de $(1.4, 1.4, 1.4)$ no octaedro, levando à renderização final que gerou a imagem mostrada na Figura 6.



**(a)** *Octaedro rotacionado.*

**(b)** *Octaedro rotacionado e com escala.*

**(c)** *Octaedro rotacionado, com escala e translação.*

**Figura 5:** *Transformações sucessivas aplicadas ao octaedro.*

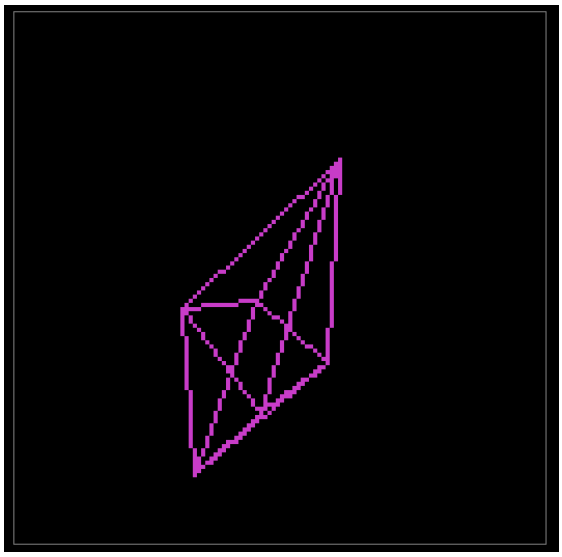


**Figura 6:** *Octaedro após transformações.*

## Referências

[1] Three.js. (2015). “three.js docs,” endereço: https://threejs.org/docs/ (acesso em 18/08/2021).

[2] W. contributors. (2021). “Bresenham’s line algorithm — Wikipedia, The Free Encyclopedia,” endereço: https://en.wikipedia.org/w/index.php?title=Bresenham%5C%27s_line_algorithm&oldid=1026473544 (acesso em 10/09/2021).

[3] C. A. Pagot. (2020). “Aula 14 - Pipeline Gráfico: Matriz de Projeção,” endereço: https://www.youtube.com/watch?v=f3S77wAvSdY (acesso em 14/10/2021).

[4] F. Dunn e I. Parberry, *3D Math Primer for Graphics and Game Development*. Jones & Bartlett Learning, 2010. endereço: https://gamemath.com/book/matrixtransforms.html.