DESIGN E IMPLEMENTAÇÃO DE UM MOTOR DE RENDERIZAÇÃO MULTIPLATAFORMA

DESIGN AND IMPLEMENTATION OF A CROSS-PLATFORM RENDERING ENGINE

Felipe da Silva Andrioli[*]
Alessandro Viola Pizzoleto[**]

RESUMO

Uma *engine* de renderização é parte essencial de diversos tipos de *software* que precisam passar pela pipeline de renderização para produzir um resultado. Por se tratar de uma peça de *software* de complexa compreensão e desenvolvimento, muitos desenvolvedores optam por utilizar soluções prontas que priorizam somente um sistema operacional para a implementação de suas aplicações. Este trabalho propõe um *design* e implementação de uma *engine* que, utilizando tecnologias de baixo nível, seguindo o *design* proposto e utilizando bibliotecas com o mesmo princípio, suporte o desenvolvimento e execução de *software* que a utilizem em múltiplos sistemas operacionais. O objetivo é de facilitar o desenvolvimento de *software* específicos que utilizem uma *engine* de renderização permitindo que os desenvolvedores foquem em funcionalidades de seu *software* com uma base já pronta e operando em múltiplos sistemas operacionais. Todas as funcionalidades da *engine* e a *pipeline* de compilação foram testadas em dois sistemas operacionais e se comportaram conforme esperado.

Palavras-chave: Renderização. Computação Gráfica. Engenharia Software. Programação de Gráficos.

ABSTRACT

A rendering engine is an essential part of many types of software that need to go through a rendering pipeline to achieve their result. For being a complex to understand and develop piece of software, many software developers choose ready solutions that prioritize a single operating system for the implementation of its applications. This work proposes a design and implementation of an engine that, using low-level technologies, following the proposed design and using libraries with the same principle, supports the development and execution of software that uses it in multiple operating systems. The goal is to aid in the development of specific software that uses a rendering engine, allowing developers to focus on the functionalities of their software with a ready-made base working on multiple operating systems. All the engine features and the compiling pipeline were tested on two operating systems and have behaved as expected.

Keywords: Rendering. Computer Graphics. Software Engineering. Graphics Programming.

[*] Discente do curso de Ciência da Computação da FATECE. felipeandrioli56.fa@gmail.com
[**] Docente e Pesquisador da FATECE e FAMEESP; Coordenador FAMEESP. alessandro.pizzoleto@fatece.edu.br

Introdução

A evolução da computação gráfica desde o seu surgimento na década de 60, transformou completamente campos profissionais tradicionais como a medicina, a engenharia e o entretenimento. Além disso, segundo a *Future Market Insights,* o mercado da computação gráfica em 2021 foi avaliado em 178.7 milhões de dólares [1], se afirmando como um mercado aquecido e bastante rentável. *Software* como AutoCad, Maya e *engines* de jogos tais como Unity e Unreal Engine, são exemplos de *software* que revolucionaram completamente seus respectivos campos de atuação, sendo todos frutos da computação gráfica e que hoje fazem parte do cotidiano de muitos profissionais.

No processo de desenvolvimento desse *software*, todos passam por uma etapa em comum, a renderização de informações em tela, que é feito através de uma peça de *software* chamada *engine* de renderização, responsável por pegar os dados em sua forma crua e iniciar a pipeline de renderização, que após passar por uma série de etapas, tem como resultado as informações renderizadas em tela, se mostrando assim um bom ponto de partida para o desenvolvimento de *software*, *engines* e jogos.

Todo esse processo se tornou mais acessível pois no começo dos anos 2000, as empresas de fabricação de GPUs começaram a disponibilizar pipelines gráficas programáveis, mudando o conceito de funções fixas na pipeline e desenvolvendo um novo conceito que dá ao desenvolvedor, acesso à diversas partes da pipeline de renderização através de dos *shaders* de vértices, fragmentos, tesselação (*tesselation*) e de primitivas [2, 3, 4]. Devido à essa flexibilidade às aplicações de cunho gráfico começaram a ficar cada vez mais complexas exigindo um nível alto de otimização para que todos os processos, além da pipeline de renderização, possam continuar sendo executados dentro de milissegundos mantendo assim um bom nível de performance.

Para adquirir tal performance frequentemente os fabricantes de *software* optam por focar somente em alguns, ou em muitos casos, um sistema operacional, utilizando bibliotecas ou técnicas para leitura e escrita de *bytes* em memória que são específicas do sistema operacional em questão. Segundo o Statista, somente em 2018 foram lançados 8.109 novos jogos na plataforma Steam [6], e no mesmo ano, dentre todos os jogos da plataforma, somente 4.060 títulos eram suportados pelos sistemas operacionais baseados em Linux [5], sendo um excelente exemplo de *software* que por executarem diversos processos, além da pipeline de renderização, precisam de bastante trabalho em otimização

para que não percam performance, e como resultado são deixados de lado em alguns sistemas operacionais.

Como um protótipo de *engine* de funcionalidade geral, será uma tentativa do autor de expor o máximo da pipeline de renderização ao usuário, seja ao nível de configuração ou ao nível de controle total sobre a pipeline com seus respectivos *shaders*, finalizando um protótipo com o máximo de configurações e controle máximo ao usuário permitidos pelas fabricantes de Unidades de Processamento Gráfico (GPU) e Interfaces de Programação de Aplicação (API) gráficas atuais.

1 Objetivo Geral

Este trabalho tem por objetivo propor o design e a implementação de um protótipo de *engine* de renderização multiplataforma, possibilitando a execução da mesma tanto em sistemas operacionais Windows quanto Linux, para servir de ponto de partida de diversos *software* que necessitam de uma pipeline de renderização.

1.1 Objetivos específicos

- Desenvolver funcionalidades básicas para qualquer *software* de cunho gráfico.
- Abstrair as classes para serem facilmente alteradas ou expandidas.
- Desenvolver uma pipeline de compilação e execução em sistemas operacionais Windows e Linux.

2 Referencial Teórico

2.1 Desenvolvimento

O desenvolvimento do motor de renderização será realizado utilizando a linguagem de programação **C++** em sua versão 17. A linguagem foi escolhida devido à necessidade de manipulação de memória no desenvolvimento da *engine* e ao suporte que todas as bibliotecas utilizadas possuem com a linguagem.

A API gráfica escolhida foi a **OpenGL** em sua versão 4.4 que devido ao seu tempo de existência e vasta documentação se mostrou ideal para ser a primeira API a ser suportada pela *engine*.

2.2 Abstração

Para atingir um bom nível de abstração foram utilizadas as bibliotecas **GLFW** versão 3.3.8 e **GLAD** versão 2.0 que recebem comandos do desenvolvedor e executam funções específicas da OpenGL de acordo com o sistema operacional na qual a *engine* está sendo executada. Essa abstração é feita em funcionalidades fundamentais tais como a criação de janelas e o processamento de entrada e saída de informações (I/O).

2.3 Biblioteca matemática

O uso de uma biblioteca matemática se faz necessária pela *engine* ter que lidar com muitos cálculos e manipulações de vértices, sendo assim a biblioteca escolhida foi a **GLM** versão 0.9.9.8 que possui praticamente todas as operações matemáticas de álgebra linear e cálculos 3D.

2.4 Carregamento de imagens

Sendo uma das funcionalidades do motor de renderização o carregamento de modelos 3D, é necessário também carregar as texturas desses modelos que possuem o formato de imagens, sendo assim a biblioteca **STB Image** foi escolhida para lidar com esse carregamento.

2.5 Carregamento de modelos 3D

Para ler, interpretar e carregar os vértices de um modelo 3D na *engine*, foi escolhida a biblioteca **Assimp** em sua versão 5.2.5, que suporta diversos modelos e possui uma boa documentação.

2.6 Interface gráfica

O desenvolvimento da interface gráfica da *engine* será realizado utilizando a biblioteca **ImGui** versão 1.88. A biblioteca é fácil de integrar e suporta múltiplas APIs gráficas, além de ser fácil de alterar e construir painéis para realizar ações complexas de forma rápida e simples.

2.7 Compilação

Para a construção da pipeline de compilação e execução será utilizada a biblioteca **CMake** versão 3.22. A biblioteca funciona com base em um arquivo de instruções para compilar o código e instruções para compilar cada uma das bibliotecas necessárias para o projeto. Cada uma das bibliotecas a serem compiladas também precisam conter um arquivo de instruções de compilação da própria biblioteca para que cada dependência possa ser compilada recursivamente.

3 Pipeline de Renderização

A pipeline de renderização não é totalmente acessível para os desenvolvedores, uma vez que algumas etapas são executadas exclusivamente na GPU, e o fluxo de informações entre o CPU e a GPU deixaria o processo extremamente lento. Entretanto, como mencionado, as fabricantes de GPUs permitem acesso total a algumas etapas e mesmo nas etapas que são executadas exclusivamente pela GPU algum nível de configuração é disponibilizado, permitindo assim um nível bastante alto de customização da pipeline para que diferentes objetivos fossem alcançados [7]. Contudo esse estudo se limitou ao processo descrito pelo livro *Real Time Rendering* [4] (Figura 1).

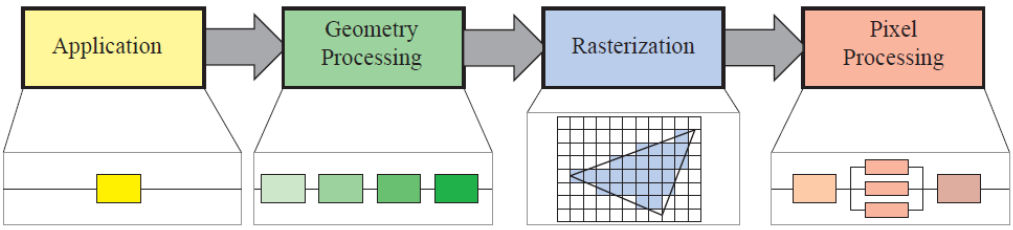

Figura 1 - Pipeline de Renderização
Fonte: Real Time Rendering, 2018.

Este texto não irá descrever todo o processo realizado pela pipeline de renderização, uma vez que é bastante extensa e complexa, saindo do escopo deste documento, entretanto descreve os processos em que a *engine* expõem para controle total ou parcial, podendo ser alterado pelo desenvolvedor.

De forma geral, como pode ser observado na Figura 1, a pipeline possui quatro etapas, sendo eles Aplicação, Processamento de Geometria, Rasterização e Processamento de Pixel, cada etapa podendo conter vários sub etapas onde diversos

processos e algoritmos como *anti aliasing* são executados. Para ter acesso à pipeline de renderização o uso de uma API gráfica se faz necessário sendo essa a responsável por enviar os códigos e configurações customizadas pelo desenvolvedor para a pipeline e assim para a GPU e no caso deste projeto foi escolhido a API OpenGL por ter uma extensa documentação e diversos exemplos em projetos, livros e artigos.

3.1 Pipeline na OpenGL

É possível ativar e desativar diversas configurações na pipeline de renderização utilizando 'glEnable' ou 'glDisable' tais como o algoritmo de *depth test*, entretanto o mais importante é o processo para enviar dados para a pipeline e a GPU como os vértices a serem renderizados, e esse processo é realizado através dos *buffers*. Vários buffers podem ser utilizados para atingir diferentes objetivos, porém os fundamentais para realizar a renderização são dois *Vertex Buffer Object* (VBO), responsável por armazenar o endereço de memória onde contém informações de um vértice, e *Element Buffer Object* (EBO), responsável por armazenar os índices de vértices que se sobrepõem, o mesmo resultado pode ser obtido utilizando somente VBOs, porém o número de vértices necessários aumenta consideravelmente como observado na Figura 2. Os dados de um vértice armazenados no trecho de memória apontado pelos VBOs podem incluir a posição do vértice, o normal desse vértice e posições de texturas.

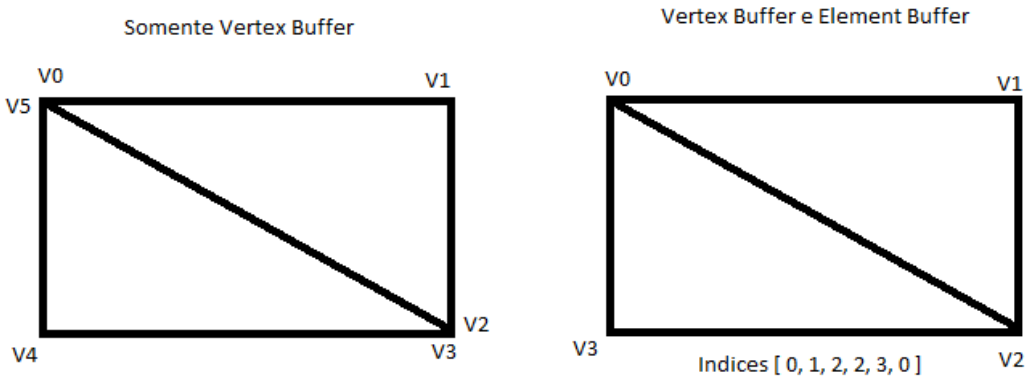

Figura 2 - VBO. VBO e EBO

Ao gerar um VBO, a OpenGL aloca um trecho de memória em sequência onde as informações que serão enviadas para a pipeline e para a GPU são armazenadas. Para armazenar essas informações a função 'glBufferData' é utilizada, recebendo como

parâmetros o buffer alvo para armazenamento, o tamanho em *bytes* do objeto a ser armazenado no *buffer*, um ponteiro apontando para a memória do valor a ser armazenado e o tipo de uso que será feito desses dados (somente leitura, somente escrita, ambos, etc). Como mencionado, toda essa informação é armazenada de forma sequencial em memória, sendo possível a formação de um padrão, uma vez que sendo armazenados vértices e coordenadas de textura por exemplo a próxima informação seria novamente os vértices, então para ler somente vértices basta indicar a posição do primeiro dado de vértices e indicar em quantos *bytes* eles serão repetidos, processo esse que é realizado através da função 'glVertexAttribPointer', identificando em qual posição da memória uma informação está localizada e em quantos *bytes* ela será repetida [8].

Toda essa informação armazenada precisa ser acessada e interpretada pela GPU, e para isso uma nova ferramenta se faz necessária, Vertex *Object Array* (VAO). Um VAO irá armazenar todas as informações de como interpretar o VBO apontado e dessa forma a GPU tem acesso a todos os vértices, coordenadas de texturas e o que mais desejarmos enviar. A comunicação completa entre VAOs, VBOs e EBOs pode ser observada na Figura 3.

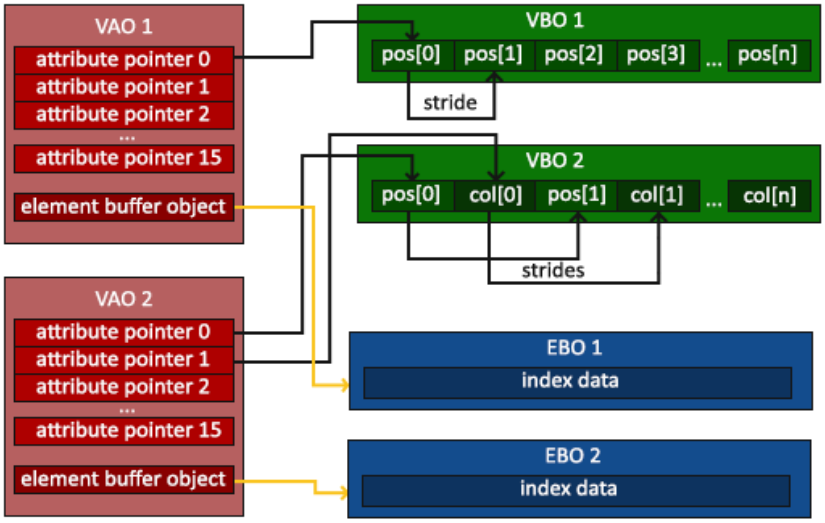

Figura 3 - Funcionamento de VAO, VBO e EBO
Fonte: Página de documentação e exemplos OpenGL[1]

3.2 Aplicação

A aplicação é o que vai definir o *software* e suas funcionalidades, o desenvolvedor tem controle total sobre este estágio e é através dele que os códigos e configurações sobre a pipeline são enviadas.

3.3 Processamento de Geometria

O estágio de processamento de geometria acontece na GPU sendo responsável pela maioria das operações por triângulo ou por vértice e é dividido nas etapas *Vertex Shading*, *Projection*, *Clipping* e *Screen Mapping*.

Dentre muitas responsabilidades, duas das maiores são: calcular a posição de um vértice e também mudá-la de acordo com a vontade do desenvolvedor para obter um resultado específico. Nem todas as sub etapas são totalmente acessíveis, algumas são configuráveis, sendo assim essa primeira versão do projeto focou em expor as etapas completamente editáveis que são o *Vertex Shading* e o *Geometry Shading*.

3.3.1 *Shaders*

Através dos *shaders* é possível editar completamente alguns pontos da pipeline de renderização. No estágio de processamento de geometria são eles os responsáveis de calcular a posição de um vértice, de alterar a posição do mesmo e também da criação de novos vértices.

O *shader* de vértices é essencial, sendo ele o responsável por calcular a posição de todos os vértices, entretanto, não é possível criar novos vértices utilizando esse *shader*, essa responsabilidade é atribuída ao *shader* de geometria. Por não ser sempre utilizado, o *shader* de geometria é considerado opcional na pipeline de renderização, sendo utilizado na maioria das vezes por sistemas de geração de partículas.

A *engine* desenvolvida expõem essa etapa ao desenvolvedor, podendo este criar novos programas de shaders, sendo esta a maneira que a API lida com os *shaders*, e após serem compilados são disponibilizados pela *engine* para serem utilizados. Para compilação os *shaders* precisam estar em um formato de “const char*” e precisam ser salvos com suas respectivas terminações (.vs para *vertex shader*, .gs para *geometry shader*, .fs para *fragment shader*).

```
Shader::Shader(const char* shader_readable_id, const char* vertexShaderPath,
    const char* fragmentShaderPath, const char* geometryShaderPath = nullptr)
```

Figura 4 - Cabeçalho da função construtora de um shader program

A compilação dos *shaders* é totalmente exposta ao desenvolvedor e será descrita com detalhes na sessão de funcionalidades, entretanto, o cabeçalho da função construtora dos *shaders* pode ser encontrado na Figura 4. Nota-se que como constatado, o *shader* de

geometria não é obrigatório para a pipeline, sendo este considerado opcional e sendo assim seu valor padrão definido como nulo. Valor este que será alterado automaticamente caso um endereço de *shader* de geometria seja providenciado.

3.4 Rasterização

Uma vez que os vértices estão transformados e projetados, o objetivo da etapa de rasterização é encontrar todos os pixels que farão parte de suas primitivas. É uma etapa dividida em duas sub etapas chamadas de *Triangle Setup* e *Triangle Transversal*, onde acontecem diversas equações para determinar se um pixel está dentro ou fora de uma primitiva. Mesmo recebendo o nome de *Triangle* a rasterização suporta todos os tipos de primitivas, mas carrega esse nome por ser a primitiva mais comum utilizada no processo de renderização. Esta é uma etapa executada exclusivamente pela GPU, porém podendo ser configurada através de diversos parâmetros.

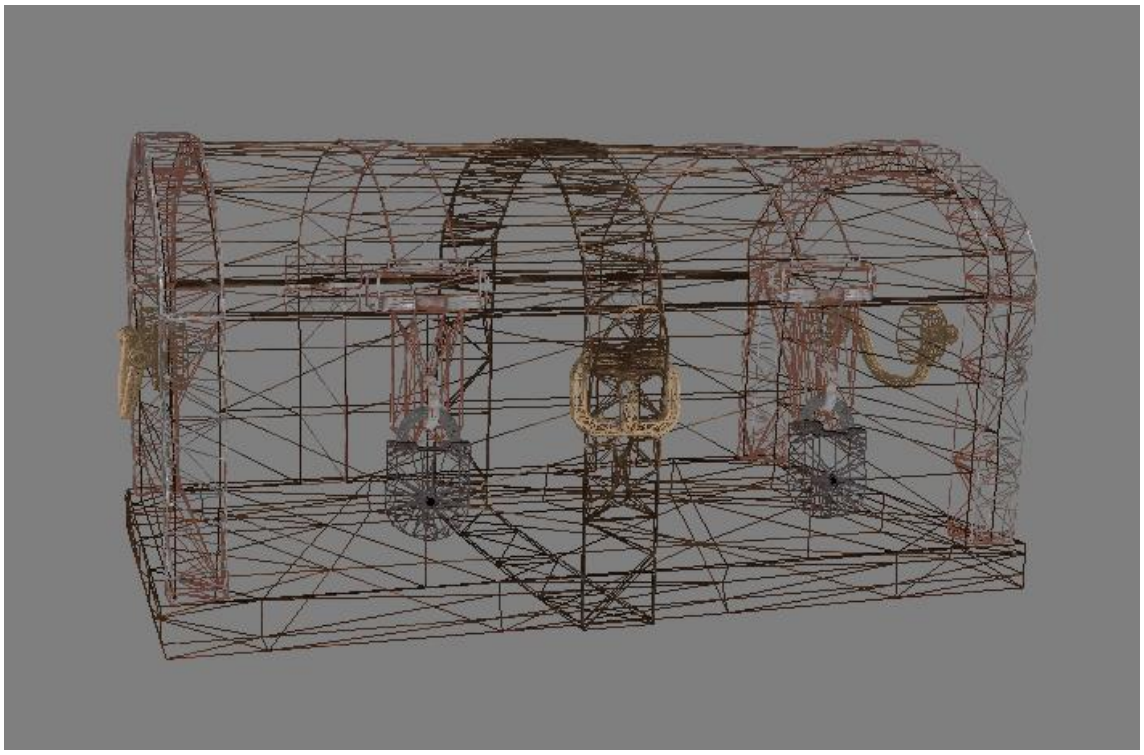

Figura 5 - Wireframe View

A *engine* desenvolvida expõem duas configurações da etapa de rasterização para o desenvolvedor sendo elas a *Wireframe View* (Figura 5) e o *Depth Test* (Figura 6 e Figura 7) . No futuro muitas outras estarão disponíveis.

Figura 6 - Depth Test desativado

Figura 7 - Depth Test ativado

3.5 Processamento de Pixel

No estágio de processamento de pixel, todos os pixels já estão dentro de suas respectivas primitivas, resultado da combinação dos estágios passados e todas as operações por pixels iniciarão. Essa etapa é dividida em duas sub etapas sendo Pixel Shading e Merging.

3.5.1 *Pixel Shading*

Todas as operações por pixel acontecem neste sub estágio, sendo este responsável por definir a cor final de cada pixel. Esta etapa é totalmente acessível ao desenvolvedor através do *fragment shader*, como é chamada na OpenGL.

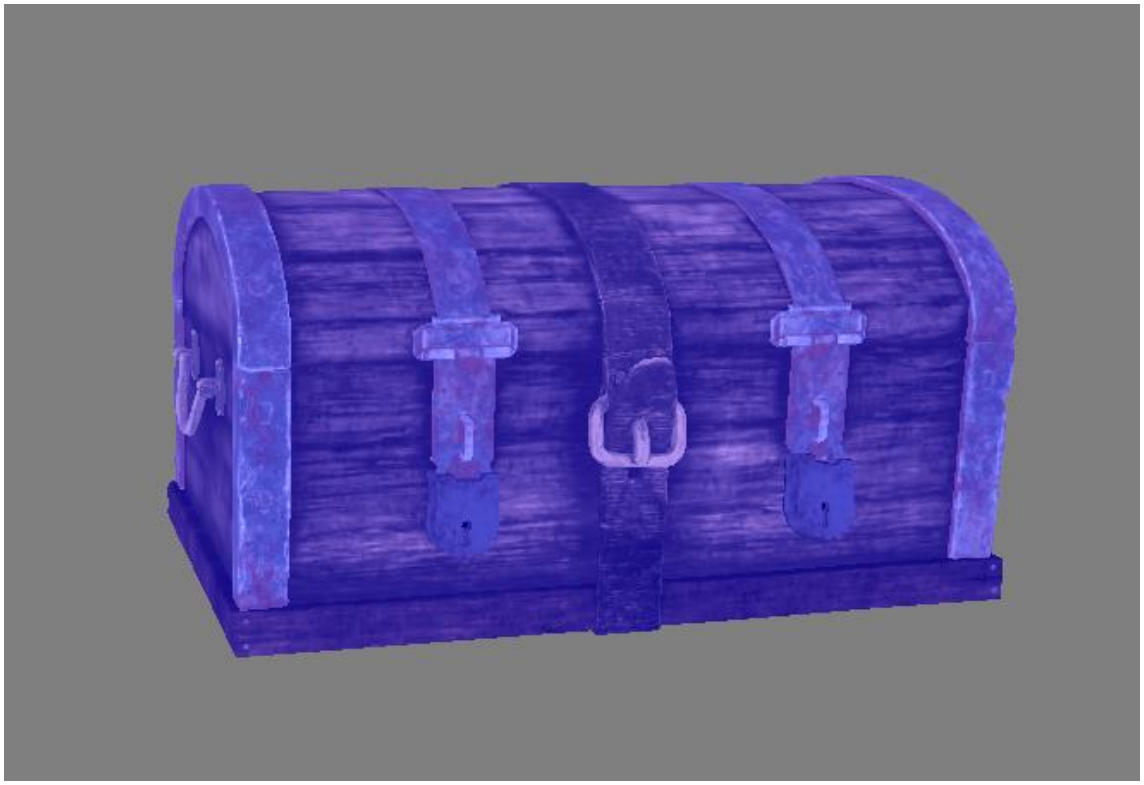

Figura 8 - Simples filtro azul aplicado como pós-processamento

Uma vasta variedade de técnicas podem ser aplicadas através deste *shader*, como a alteração de cores, aplicação de texturas (Figura 7) e técnicas de pós-processamento (Figura 8) como exemplos.

O desenvolvedor tem acesso a esse estágio na *engine* pelo *fragment shader* de terminação. fs e adicionando-o ao *shader* programa ser compilado e logo após a compilação pode ser adicionado ao objeto. Também pode-se optar pelo *shader* já existente e implementar nele alterações.

4 Funcionalidades Desenvolvidas

Para desenvolver as funcionalidades da *engine* diversas bibliotecas externas foram utilizadas e serão explicadas de acordo com a funcionalidade que foi desenvolvida junto a ela.

4.1 I/O

GLFW, segundo sua própria definição, é uma biblioteca multiplataforma para OpenGL, OpenGL ES e Vulkan. Ela providencia uma API simples que cria janelas, contextos e interfaces, recebendo entradas e eventos. É através dessa biblioteca que toda a abstração é feita permitindo que, independentemente do sistema operacional, todos os processos fossem resolvidos da mesma maneira.

4.1.1 Janela

A criação da janela em que a *engine* é executada é feita através da função 'glfwCreateWindow', que recebe como parâmetros principais a resolução desejada e o título da janela. O *looping* principal da *engine* é executado enquanto o retorno da função 'glfwWindowShouldCLose', que recebe como parâmetro a própria janela, retornar o valor falso.

4.1.2 Teclado

```
4  typedef struct {
5      bool pressed;
6  } Key;
7
8  typedef struct {
9      Key keys[GLFW_KEY_LAST];
10 } Keyboard;
```

Figura 9 - Estruturas para armazenar informações de uma tecla e do teclado

Para lidar com as entradas do teclado duas estruturas foram criadas. A primeira para armazenar a informação se a tecla está sendo pressionada chamada de *key* e a segunda sendo um array de *key* representando cada tecla do teclado chamada de keyboard. Ambas as estruturas podem ser observadas na Figura 9.

O uso da constante GLFW_KEY_LAST é necessário para que todas as teclas suportadas pela biblioteca tenham sua posição específica dentro da estrutura. Dessa forma também a leitura de uma tecla em específica na array é feito com uma complexidade constante O(1) e para verificar se uma tecla está sendo pressionada basta checar se a propriedade *pressed* dentro da posição específica da tecla na estrutura é verdadeira (Figura 10).

```
if (state.window->keyboard.keys[GLFW_KEY_0].pressed) {
    state.config_mode = true;
}
```

Figura 10 - Verificação de tecla pressionada

Por fim, para que as teclas pressionadas alterem seu estado é utilizada uma função de *callback* 'glfwSetKeyCallback', que recebe como parâmetros a janela em que a função será aplicada e a função que será executada em cada *callback*. A função que será executada precisa conter alguns parâmetros e esses são um ponteiro da janela, uma *key*, um *scancode*, uma *action* e um *mods*. Nem todos os parâmetros estão sendo utilizados pela *engine* em sua primeira versão, entretanto já estão disponíveis para melhoras.

```
switch(action) {
    case GLFW_PRESS:
        keyboard.keys[key].pressed = true;
        break;
    case GLFW_RELEASE:
        keyboard.keys[key].pressed = false;
        break;
    default:
        break;
}
```

Figura 11 - Switch que altera o estado das keys

Quando a função de *callback* é executada ela ativa um *switch* de que caso a *action* seja igual GLFW_PRESS, então a propriedade *pressed* da *key* é alterada para *true*, caso seja igual a GLFW_RELEASE, então a propriedade é alterada para *false* (Figura 11).

4.1.3 Mouse

Para lidar com a entrada do mouse uma nova função de *callback* é utilizada chamada de 'glfwSetCursorPosCallback', que recebe como parâmetro a janela onde o *callback* será aplicado e a função que será executada. A função precisa ter os parâmetros: um ponteiro para a janela, uma posição x e uma posição y.

```
typedef struct {
    double x;
    double y;
    double z;
    double last_x;
    double last_y;
    int on_screen;
    bool first_mouse;
} Mouse;
```

Figura 12 - Estrutura de informações do mouse

Uma estrutura também foi criada para armazenar as propriedades do mouse que, assim como a estrutura do teclado, não faz uso de sua total capacidade na primeira versão da *engine*, que, na função executada pelo *callback*, somente muda a posição do mouse que será utilizada pela câmera.

4.2 UI

Uma interface gráfica foi implementada para ter controle sobre a *engine* e para isso foi utilizada uma biblioteca chamada Dear ImGui. Essa biblioteca possui diversos componentes (Figura 13) que permitem uma fácil integração e habilita uma interação rápida com a *engine*.

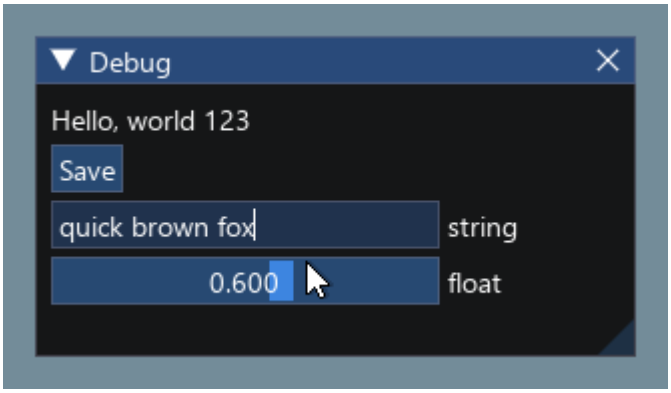

Figura 13 - ImGui componentes

A primeira versão da *engine* possui interface gráfica para alterar configurações no Renderer, na Scene e em cada um dos modelos carregados, além de exibir diversas informações para propósitos de debug (Figuras 14 e 15).

Os valores alterados pela interface gráfica são atualizados nos seus respectivos destinos a cada chamada de atualização do *looping* principal e passados adiante para

serem renderizados com os valores atualizados através dos *uniforms* que serão explicados na próxima sessão.

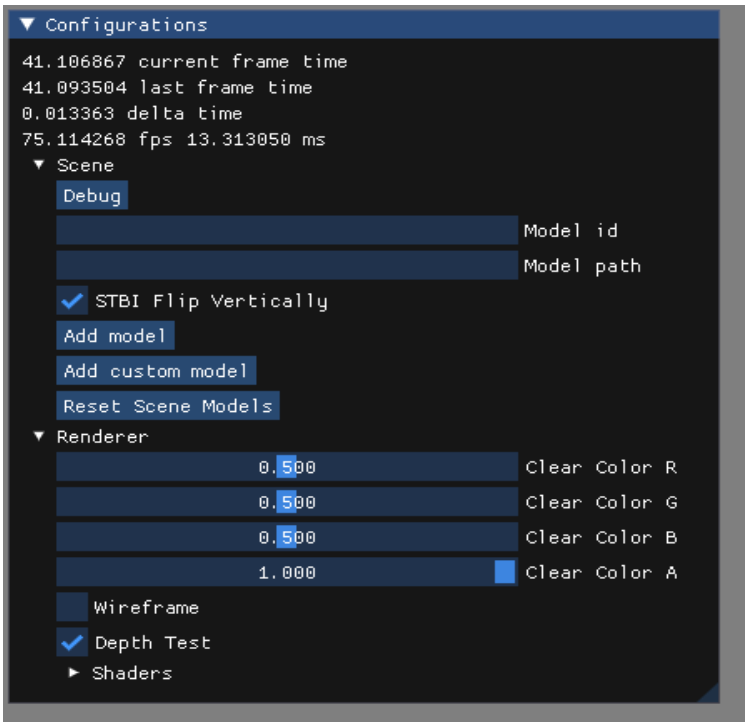

Figura 14 - UI para alteração do Renderer e da Scene

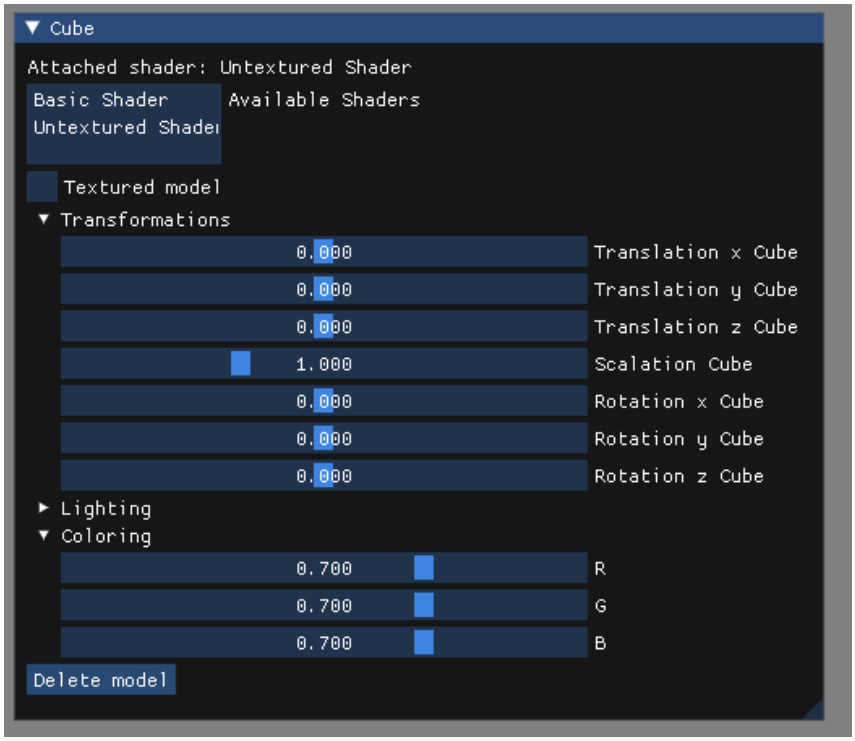

Figura 15 - UI para alteração das propriedades de cada modelo

4.3 Carregamento de modelos 3D

A *engine* suporta o carregamento de modelos 3D através de uma biblioteca chamada Assimp. A biblioteca lida com o processamento de *meshes, normals*, tangentes, bastando ao desenvolvedor armazenar as *meshes* processadas em uma estrutura para que seja processada e renderizada. Em sua primeira versão a *engine* suporta o carregamento de modelos com arquivos de texturas únicos.

```
1
2  struct Vertex {
3      glm::vec3 Position;
4      glm::vec3 Normal;
5      glm::vec2 TexCoords;
6      glm::vec3 Tangent;
7      glm::vec3 Bitangent;
8      int m_BoneIDs[MAX_BONE_INFLUENCE];
9      int m_Weights[MAX_BONE_INFLUENCE];
10 };
11
```

Figura 16 - Estrutura de um vértice da classe Mesh

Mesh é o conjunto de vértices, com suas posições, *normals*, coordenadas de texturas, não se limitando somente à essas informações, e representa um modelo 3D. Para que fossem carregados e posteriormente processados, a estrutura que pode ser observada na Figura 16 foi criada e faz parte de uma classe *Mesh* que possui um vetor com vértices, índices e texturas.

Para carregar um modelo, deve-se selecionar o caminho até o arquivo de vértices, alguns exemplos foram providenciados na pasta *models* da *engine*, e dar uma identificação para esse modelo, que será utilizada para criar os componentes da interface gráfica e designar o *shader* que será utilizado na renderização. Uma vez adicionado, o modelo terá seu próprio componente de interface gráfica onde o desenvolvedor poderá realizar alterações nos valores do mesmo (Figura 15).

Existem vários formatos de modelos 3D que são exportados por diversos *software* de modelagem, sendo assim o suporte a todos não foi possível na primeira versão da *engine*. Mesmo utilizando uma biblioteca externa para auxiliar no carregamento de modelos, alguns ajustes ainda precisam ser feitos para suportar diferentes formatos de arquivos, sendo assim a *engine* foi extensivamente testada apenas com formatos .obj. Formatos .obj possuem consigo um arquivo de materiais (.mtl) que carregam consigo a

identificação não só do tipo da textura, mas também o nome do arquivo que será utilizado como textura, sendo fundamental para o carregamento de texturas junto ao modelo.

4.3.1 Carregamento de texturas

A biblioteca stb_image é bastante conhecida para o carregamento de imagens em aplicações C++ e sendo assim foi a biblioteca escolhida para o carregamento das texturas do modelo. A biblioteca é bem simples de usar e com o uso de uma única função "stbi_load" já carrega a imagem.

Após ser carregada, a textura precisa ser ativada em um dos canais de textura que a OpenGL disponibiliza utilizando a função "glActiveTexture". A função recebe como parâmetro um *enumerator* indicando qual unidade será ativada, sendo identificadas como GL_TEXTUREi, onde i é o número do canal indo de 0 à constante GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS menos um.

Para que os *shaders* tenham acesso às texturas carregadas um meio de comunicação entre os estágios totalmente acessíveis da pipeline e a aplicação precisa ser estabelecido, assim informações como texturas e transformações podem ser aplicadas ao modelo, esse meio de acesso de informações são os *uniforms* (também denominados *constant buffers* na API gráfica Direct3D). Utilizando os *uniforms* para enviar a unidade de textura ativa é possível assim acessá-las para aplicar ao modelo, onde a linguagem de *shaders* própria da OpenGL a GLSL (*OpenGL Shading Language*), possui uma função chamada *texture*, que recebe a textura carregada e a coordenada da textura como parâmetros, e aplica a textura em sua coordenada correspondente ao modelo.

4.4 Câmera

A câmera é parte fundamental em uma *engine* de renderização, uma vez que é utilizada para visualizar um mesmo lugar por diferentes pontos de vistas, para ter controle sobre a cena e até mesmo para cálculos avançados de iluminação.

A primeira versão da *engine* contém uma câmera em primeira pessoa estilo FPS (*first person shooter*) que foi desenvolvida seguindo as transformações de Euler, *pitch*, *yaw* (em algumas fontes chamado de *head*) e *roll*, sendo esse último não utilizado na implementação. As transformações *pitch* e *yaw* representam respectivamente a rotação da câmera sobre o eixo x e sobre o eixo y, e formalmente a transformação completa é representada pela seguinte fórmula:

$$E(h,p,r) = Rz\,(r)\,Rx\,(p)\,Ry\,(h),$$

onde E é a concatenação das rotações sendo essas a rotação *roll* sobre o eixo z, *pitch* sobre o eixo x e *yaw* (*head*) sobre o eixo y. Para que tais rotações sejam ativadas, a cada chamada de *update* da *engine,* o *offset* do mouse é calculado através das já mencionadas *callbacks*, sendo assim possível atualizar os valores da matriz 'lookat', que determina para onde a câmera estará apontando.

A câmera também possui transformações ligadas às teclas W, A, S, D que, quando pressionadas, atualizam o vetor de posição da câmera que é utilizado para realizar o cálculo da função 'lookat', atualizando a posição da câmera.

5 Compilação

O processo de compilação pode ser particularmente desafiador, uma vez que precisam ser compiladas cada uma das bibliotecas externas utilizadas e o código fonte da *engine*, realizar o *link* das bibliotecas com a *engine* sendo que preferencialmente o processo aconteça em múltiplos sistemas operacionais da mesma forma. Todos esses estágios são necessários por se tratar de um projeto em desenvolvimento, em uma aplicação consolidada a compilação aconteceria de formas diferentes.

A compilação das bibliotecas e do código fonte da *engine* pode ser feito de diversas maneiras, tais como manualmente, através de scripts ou através de *software* que automatizam esse processo. A opção escolhida para a *engine* foi utilizar o *software* CMake que além de ser suportado por todas as bibliotecas, faz uma compilação rápida e simples, possibilitando re-compilar somente o que foi alterado no código fonte quando necessário. O *software* também atende ao requisito de suportar múltiplos sistemas operacionais, sendo assim possível realizar a compilação da mesma forma independente do sistema operacional e dos *softwares* utilizados para realizar o desenvolvimento.

5.1 CMake

O processo de compilação do *software* funciona com base em um arquivo .txt que precisa ter o nome de CMakeLists (CMakeLists.txt), onde é possível adicionar as bibliotecas a serem compiladas de maneira recursiva, e realizar o *link* das mesmas com a

engine. Após esse arquivo ser executado, o CMake constrói arquivos de *build*, para compilar cada uma das bibliotecas e o código fonte.

O *software* suporta tanto compilações pela sua interface gráfica quanto por linha de comando, funcionando também como um CLI (*Comand Line Interface*). Entretanto no processo de compilação da *engine* foi optado o uso somente de comandos para que houvesse controle sobre todas as opções que o CMake suporta, uma vez que a interface gráfica possui muitas opções de configurações e podem gerar conflitos com o sistema operacional no qual está sendo executado.

5.2 Pipeline de compilação da *Engine*

Para iniciar a *pipeline* de compilação da *engine*, foi desenvolvido um pequeno *script* para executar o primeiro comando CMake, para montar os arquivos de *build* que precisam ser compilados. O *script* possui a opção de compilar ou não as bibliotecas externas, sendo que essas precisam ser compiladas somente uma vez, o *script* se faz necessário novamente somente em casos de adição de novos arquivos com dependência nas bibliotecas externas por ser responsável pelo *link*.

Após o *script* ser executado um novo diretório (*build*) é criado com os arquivos de *build* onde de fato toda a compilação será executada e utilizando o *software* CMake, o uso de somente um comando é necessário ***make***. A primeira execução desse comando é responsável por realizar a compilação de todas as bibliotecas externas e do código fonte da *engine*, as execuções seguintes são responsáveis somente pela compilação do código fonte que foi alterado ou adicionado, tornando assim um processo mais rápido de compilação.

Resultados

Utilizando bibliotecas de código aberto e de pouca manipulação a nível de sistema operacional, conhecimento em matemática e em computação, foi possível obter uma *engine* de renderização que pode servir de base para *software* mais específicos que possam vir a ser desenvolvidos a partir dela, e até mesmo alocada a uma aplicação já existente.

As funcionalidades desenvolvidas podem ser consideradas como base de qualquer sistema que utilize renderização 3D e foram desenvolvidas para serem facilmente alteradas e expandidas. A câmera por exemplo pode ser facilmente transformada para o

estilo MOBA (movimentação da câmera ao encostar o cursor nas bordas da janela) ou Photoshop (clicar, segurar e arrastar para mover a câmera), sem muito conhecimento de como todos os cálculos para posicionamento da câmera funciona.

Embora muitas melhorias ainda possam ser feitas, o funcionamento da *engine* acontece como planejado no seu estágio inicial de desenvolvimento, seguindo os mesmos princípios de desenvolvimento desde o princípio e se baseando em fontes sólidas para implementação de conceitos matemáticos. A maioria das bibliotecas utilizadas podem ser facilmente implementadas do zero se necessário para que haja uma melhor optimização ou para que funcionem de uma maneira específica se necessário, que inclusive, é o recomendado para *software* mais robustos. Sendo assim a *engine* cumpre o seu propósito de ser um ponto de partida em comum para diversos tipos de *software*, independente de plataformas de desenvolvimento ou de sistemas operacionais e seu completo funcionamento pode ser observado na Figura 17.

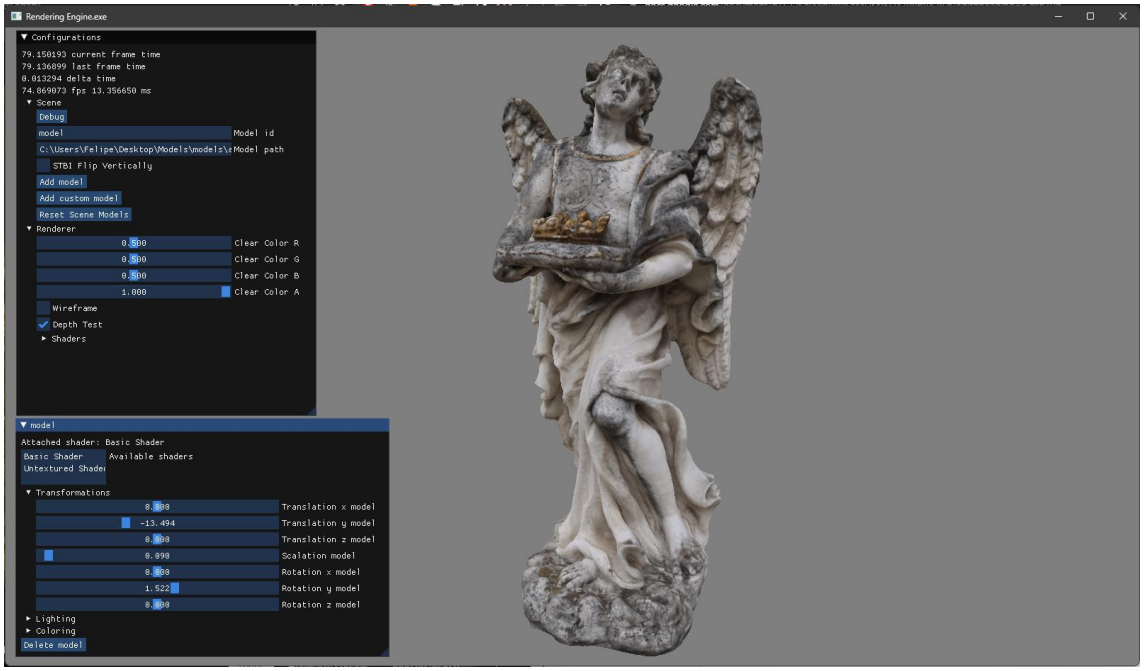

Figura 17 - Funcionamento completo do motor de renderização

Durante o desenvolvimento todas as funcionalidades foram testadas nativamente no Windows 11 e também utilizando uma máquina virtual com a última versão LTS (*long-term support*) disponível do sistema operacional Ubuntu (22.04.1). Apesar da instalação e configuração dos *softwares* necessários serem feitos de maneiras diferentes, tudo funcionou conforme esperado, da compilação a todas as funcionalidades desenvolvidas.

Considerações Finais

Diante dos resultados obtidos é possível concluir que, utilizando tecnologias para o desenvolvimento de *software* em baixo nível, com acesso não só a pipeline de renderização mas ao *software* como um todo, e bibliotecas que sigam a mesma mentalidade de desenvolvimento, é possível criar um ponto de partida em comum para o desenvolvimento de diversos *software* complexos e que precisam passar pela pipeline de renderização para atingir seu objetivo final e que ainda possam ser desenvolvidos/executados em diversos sistemas operacionais. Ainda que todas as bibliotecas possuam código aberto e sejam totalmente acessíveis para os desenvolvedores, é bastante comum que sejam desenvolvidos procedimentos próprios para carregar modelos 3D por exemplo, uma vez que diferentes *softwares* desenvolvidos irão utilizar diferentes estruturas em seus modelos 3D, sendo assim é comum que bibliotecas semelhantes sejam desenvolvidas completamente do zero para que os desenvolvedores tenham mais controle sobre toda a aplicação.

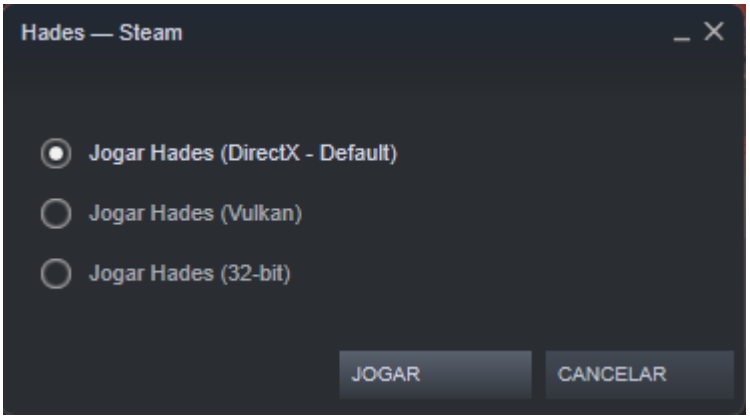

Figura 18 - Suporte a múltiplas APIs gráficas
Fonte: Jogo eletrônico **Hades** na Steam

Futuramente há a pretensão de modernizar a *engine*, adicionando suporte para múltiplas APIs gráficas, como mostra a Figura 18 o menu de inicialização do jogo Hades, sendo essas Vulkan para múltiplos sistemas operacionais e DirectX quando executada somente em um sistema operacional Windows. O autor planeja também desenvolver um *software* específico de estilização utilizando modelos 3D e shaders a partir da *engine* desenvolvida. Além de objetivos específicos, o autor também planeja continuar atualizando e melhorando a *engine* de forma geral.

Referências

1. FUTURE MARKET INSIGHTS. **Computer Graphics Market**, 2022. Market Insights on Computer Graphics covering sales outlook, demand forecast & up-to-date key trends. Disponível em: Computer Graphics Market Size, Industry Share & Trends – 2032. Acesso em: 19 dez. 2022.

2. KESSENICH, John; SELLERS, Graham; SHREINER, Dave. **OpenGL Programming Guide.** 8. ed. Boston: Addison-Wesley Professional, 2016.

3. LUNA, Frank. **Introduction to 3D game programming with directX 12. Mercury Learning & Information.** United States: Herndon VA, 2016.

4. AKENINE-MÖLLER, Tomas *et al*. **Real-Time Rendering**. 4. ed. Boca Raton: CRC Press, 2018.

5. STATISTA. **J.Clement**, 2021. Number of Linux games available on Steam worldwide as of January 2018. Disponível em https://www.statista.com/statistics/761434/number-linux-games-steam/. Acesso em: 19 dez. 2022.

6. STATISTA. **J.Clement**, 2022. Number of games released on Steam worldwide from 2004 to 2022. Disponível em https://www.statista.com/statistics/552623/number-games-released-steam/. Acesso em: 19 dez. 2022.

7. PAVEL, Zemcik; PAVEL, Tisnovsky; ADAM, Herout. Particle rendering pipeline. *In*: SPRING CONFERENCE ON COMPUTER GRAPHICS (SCCG '03), 19. **Proceedings[…],** 2003, Association for Computing Machinery, New York, NY, USA, p. 165-170. Disponível em: https://doi.org/10.1145/984952.984979. Acesso em: 19 dez. 2022.

8. LAAKSONEN, Jarno. **OpenGL Rendering Pipeline**. 2017. Disponível em: https://www.theseus.fi/bitstream/handle/10024/140206/Laaksonen_Jarno.pdf;jsessionid=15E1C034FCDD5A27D56A024015274973?sequence=1. Acessado em: 10 maio 2023.