Universidade de Brasília

Instituto de Ciências Exatas

Departamento de Ciência da Computação

O Motor de Jogos 'Bloss1'

Gustavo Tomás de Paula

PROJETO FINAL DE CURSO

BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Brasília

2024

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

O Motor de Jogos 'Bloss1'

Gustavo Tomás de Paula

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Ricardo Lopes de Queiroz

Brasília
2024

FICHA CATALOGRÁFICA

Tomás de Paula, Gustavo.
O Motor de Jogos 'Bloss1' / Gustavo Tomás de Paula; orientador Ricardo Lopes de Queiroz. -- Brasília, 2024.
102 p.

Projeto Final de Curso (Bacharelado em Ciências da Computação) -- Universidade de Brasília, 2024.

1. Motor de Jogos. 2. Motor de Jogos 3D. 3. Computação Gráfica. I. Lopes de Queiroz, Ricardo, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

O Motor de Jogos 'Bloss1'

Gustavo Tomás de Paula

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação.

Trabalho aprovado. Brasília, 31 de julho de 2024:

Prof. Dr. Ricardo Lopes de Queiroz,
UnB/IE/CIC
Orientador

Prof.a Dr.a Carla Denise Castanho,
UnB/IE/CIC
Examinador interno

Prof.a Dr.a Edna Dias Canedo,
UnB/IE/CIC
Examinador interno

Eu dedico este trabalho àqueles que me deram a base
para construir o meu futuro

Agradecimentos

Eu agradeço a minha família, amigos, colegas e professores. Obrigado pela oportunidade, apoio, conhecimento e inspiração para realizar este trabalho.

“To play a wrong note is insignificant;
to play without passion is inexcusable.”
(Ludwig van Beethoven)

Resumo

Com a evolução do *hardware* e técnicas de renderização no campo de computação gráfica e simulação física, os limites para criar experiências engajantes e completas é quase inexistente. Entretanto, criar mundos virtuais em jogos se mostra uma tarefa não trivial e extremamente custosa, especialmente em um ambiente 3D que é consideravelmente mais complexo de representar do que um ambiente 2D. Para solucionar esse problema, surgiu o conceito de um motor de jogos, cujo propósito é facilitar e automatizar diversas etapas no desenvolvimento de jogos. Este documento propõe um novo motor de jogos 3D (*engine*) Bloss1. Dentre as funcionalidades mais importantes, destacam-se o motor de renderização OpenGL, simulação física de corpos rígidos com integração de forças e o áudio espacial. O desenvolvimento e teste de estágios também é facilitado pelo editor, que permite configurar diversos sistemas em tempo de execução. Por fim, foi feito um jogo exemplo utilizando a *engine* para demonstrar o potencial da Bloss1.

Palavras-chave: Motor de Jogos. Motor de Jogos 3D. Computação Gráfica.

Abstract

With the evolution of hardware, rendering techniques and physics simulation, the limits to build an engaging experience are next to none. However, creating virtual worlds in games has been shown to be a difficult and costly endeavour, even more so in a 3D environment which is considerably more complex to represent than its 2D counterpart. To solve this problem, the concept of game engines emerged to streamline and facilitate several steps of the game development pipeline. This document details the features and inner workings of the 3D game engine Bloss1. The OpenGL renderer, physics simulation with fixed timestep and spacial audio are some of its key features. The development and testing of games is streamlined by the editor, allowing several scene configurations to be changed during runtime. A game was developed using the engine to demonstrate all of its features.

Keywords: Game Engine. 3D Game Engine. Computer Graphics.

Lista de figuras

Lista de abreviaturas e siglas

AAA	*Triple A*
API	*Application Programmable Interface*
CSM	*Cascaded Shadow Mapping*
ECS	*Entity Component System*
FOSS	*Free and Open Source*
FXAA	*Fast Approximate Antialiasing*
GPU	*Graphics Processing Unit*
LERP	*Linear Interpolation*
OpenGL	*Open Graphics Library*
PBR	*Physically Based Rendering*
RNG	*Random Number Generator*
SO	Sistema Operacional
STL	*Standard Template Library*

Sumário

1 Introdução

Quando a criação de jogos eletrônicos estava em sua infância, os jogos criados eram simples: mundo bidimensional, movimentação restrita e hardware limitado (Gregory, 2015, p.856). Com a evolução do conhecimento de engenharia de software e do hardware, as limitações para criação de jogos foram reduzidas e, com isso, jogos começaram a ser maiores e mais complexos. Agora, jogos são softwares multidisciplinares complexos que envolvem diversas áreas de conhecimento, como áudio, simulação física e computação gráfica em tempo real, dentre outras.

Diante desse grande aumento de complexidade, se tornou inconveniente desenvolver um jogo sem o auxílio de uma ferramenta para automatizar e abstrair diversas funcionalidades necessárias para criação de jogos, principalmente aspectos de mais baixo nível como sincronização, gerenciamento de memória, gerenciamento de *assets* e compatibilidade para diferentes plataformas. Jogos 3D em particular são muito mais complexos que jogos 2D, pois necessitam de uma base matemática mais potente para representar e organizar um mundo virtual.

O tempo de desenvolvimento também se tornou uma inviabilidade. Se antes os jogos mais complexos eram criados em algumas semanas, agora prazos podem se estender por meses ou até anos, mesmo com uma equipe de centenas de desenvolvedores (Gregory, 2015, p.856).

Para facilitar o desenvolvimento desses programas surgiu, então, um software específico para criação de jogos: o motor de jogos[1]. Também chamado de *game engine*, um motor de jogos pode ser comparado a um sistema operacional (SO). Em suma, um SO é responsável por gerenciar os recursos de uma máquina e criar uma camada de abstração para o usuário (Tanenbaum, 2015, p.3). Analogamente, uma *game engine* é responsável pelo gerenciamento de diversos recursos necessários para o funcionamento de um jogo e pela criação de uma abstração para o usuário, facilitando a integração de usuários que não possuem, necessariamente, conhecimentos de programação aprofundados em todos os aspectos necessários para construir um jogo.

Algumas *engines* surgiram ao longo dos anos para solucionar os problemas apresentados. Soluções comerciais como Unreal Engine[2] e CryEngine[3] são adequadas para criação de jogos AAA (*Triple A* - jogos que são grandes e complexos. Geralmente, é preciso de equipes de centenas de desenvolvedores para realizar esses projetos). Unity[4] possui uma extensa

[1] Exemplos de motores de jogos são apresentados nas *footnotes* 2, 3 e 4

[2] Disponível em: https://www.unrealengine.com/

[3] Disponível em: https://www.cryengine.com/

[4] Disponível em: https://unity.com/

asset store e um bom suporte para plataformas *mobile*. Mais recentemente, Godot[5] surgiu como alternativa livre e de código aberto (*FOSS - Free and Open Source*.). Entretanto, cada uma dessas *engines* possui desvantagens. É preciso pagar *royalties* para jogos produzidos com Unreal, CryEngine e Unity. A Godot é livre para uso comercial e utiliza um sistema de nós (Nodes) que é intuitivo e fácil de usar para organizar uma cena. Embora seja fácil de usar, esse sistema de nós não é tão performático quanto um ECS (*Entity Component System* - é uma forma eficiente para armazenar uma cena em um jogo) e, por isso, não é possível criar um mundo extenso como na Unreal sem perda de performance (a documentação da Godot[6] apresenta algumas formas para melhorar a performance). O objetivo deste trabalho é propor uma nova *engine*, denominada Bloss1, que possua código aberto e ofereça um bom desempenho em termos de performance computacional. Ou seja, uma alternativa que fornece diversas técnicas de otimização, inclusive um ECS orientado a dados.

A Bloss1 oferece diversas funcionalidades para facilitar o desenvolvimento de jogos, como editor com interface gráfica, renderizador OpenGL moderno, simulação física de corpos rígidos e áudio espacial, embora ainda seja necessário algum conhecimento de programação na linguagem c++ para apreciar todo o potencial da *engine*.

Para demonstrar o uso da Bloss1, foi criado um jogo piloto 3D de tiro em terceira pessoa, chamado F90. Esse jogo utiliza todas as funcionalidades e sistemas oferecidos pela *engine*, como gerenciamento de estágios, os sistemas de pós-processamento e renderização, o ECS e o sistema de física.

Este documento está estruturado da seguinte maneira: o Capítulo 2 apresenta os termos e conceitos utilizados neste trabalho. O Capítulo 3 apresenta a estrutura/organização da *engine* em alto nível. O Capítulo 4 detalha os subsistemas apresentados no Capítulo 3 e o Capítulo 5 explica a implementação do ECS em detalhes. Por fim, o Capítulo 6 apresenta o jogo criado utilizando a *engine* e o Capítulo 7 as conclusões do trabalho.

[5] Disponível em: https://godotengine.org/

[6] Fonte: https://docs.godotengine.org/en/stable/tutorials/performance/index.html

2 Conceitos Básicos

Neste capítulo serão apresentados os conceitos mais relevantes no âmbito de desenvolvimento de jogos e de motores de jogos. Também serão apresentados alguns problemas comuns e como *engines* procuram solucionar esses problemas.

2.1 Componentes de uma *Engine*

Um motor de jogos é um pedaço de *software* complexo, composto por diversos sistemas que gerenciam diferentes tipos de dados. As seções seguintes apresentam alguns desses sistemas e as estruturas utilizadas para armazenar esses dados.

Subsistemas

Uma *engine* geralmente é composta por diversos subsistemas que se encarregam por gerenciar e atualizar diferentes áreas. A *engine* de *Doom*, por exemplo, possui um subsistema para renderização 3D, um subsistema para tratar colisões e um subsistema para áudio, dentre outros (Gregory, 2015, p.11).

A separação de sistemas diferentes é importante para facilitar a manutenção e diminuir a ocorrência de *bugs*. Além disso, o código do motor deve ser completamente separado do código do jogo desenvolvido. Idealmente, uma *engine* deve sofrer pouca ou nenhuma modificação para implementar qualquer jogo desenvolvido com tal *engine*.

Game Loop

O *game loop* é uma forma de estruturar a execução de uma aplicação. Consiste em um laço de repetição que é executado até uma interrupção ocorrer, geralmente pelo usuário (Gregory, 2015, p.340). Dentro desse *loop*, subsistemas que precisam ser atualizados periodicamente são executados em uma ordem específica. O processamento de *input*, por exemplo, é o primeiro sistema a ser executado. Em seguida, é executado o sistema de física e, por último, a renderização é efetuada. A Figura 2.1 ilustra um exemplo de *game loop*.

```
while (running)
{
    // 1. Calculate delta time (dt)
    dt = curr_time - last_time;

    // 2. Update current stage
    stage.update(dt)
    {
        // 2.1. Update the registered systems
        for (system in systems)
        {
            system.update(dt);
        }
    };

    // 3. Update the editor
    editor.update(ecs, dt);

    // 4. Update the window (swap buffers)
    window.update();
}
```

Figura 2.1 – Exemplo de um *game loop*.

Entity Component System - ECS

Um dos problemas que *engines* procuram solucionar é a representação de mundos virtuais. Geralmente, objetos em um mundo (jogador, inimigos, modelos, etc) são representados por um objeto em um modelo orientado a objetos. Essa representação, embora útil, se torna inconveniente com o aumento de complexidade. Uma classe que representa o jogador, por exemplo, cresce com a quantidade de interações com o mundo e mais métodos e atributos são criados para cada interação (Fabian, 2018, p.85).

Para tornar o gerenciamento de um mundo com, possivelmente, centenas de interações diferentes para cada objeto, foi criado o padrão de *design Entity Component System* (ECS). A ideia desse padrão é representar objetos (chamados entidades) por meio de componentes e um identificador. Em uma forma orientada a dados, cada componente é composto apenas por dados, geralmente uma *struct*. A posição de uma entidade *Transform*, por exemplo, pode ser representada por uma tripla x, y, z. Os sistemas são responsáveis por determinar a lógica e atualizar cada componente (Fabian, 2018, p.91). Um sistema *rendering_system*, por exemplo, pode ser encarregado de renderizar um componente *Model* na posição dada pelo componente *Transform*.

Além de ser uma forma flexível de representar um mundo, o modelo ECS possui a

vantagem de ser orientado a dados e, por isso, é extremamente performático. Isso porque um sistema é responsável por atualizar todos os componentes com o mesmo tipo e, geralmente, os componentes são eficientemente armazenados em um *sparse array* (Fabian, 2018, p.94). Essa forma de armazenar componentes com o mesmo tipo em posições adjacentes também aumenta a probabilidade da ocorrência de *cache hits* (Tanenbaum, 2015, p.25).

Um exemplo de modelo ECS que utiliza *hash tables* para armazenar componentes pode ser observado no Código 2.1.

Código 2.1 – Tabelas de componentes (ecs.hpp).

```
// Table of components
std::map<u32, str> names;
std::map<u32, std::unique_ptr<Transform>> transforms;
std::map<u32, std::unique_ptr<ModelComponent>> models;
std::map<u32, std::unique_ptr<DirectionalLight>> dir_lights;
std::map<u32, std::unique_ptr<PointLight>> point_lights;
std::map<u32, std::unique_ptr<PhysicsObject>> physics_objects;
std::map<u32, std::unique_ptr<Collider>> colliders;
...
```

2.2 Gráficos

A parte gráfica de uma *engine* é, possivelmente, a mais importante dentre todas as outras. Isso porque jogos são, em grande parte, experiências visuais e as características para criação desses visuais são limitadas pelo renderizador implementado. Um renderizador deve ser eficiente e permitir a renderização de cenas complexas em tempo real.

A seguir, serão apresentados alguns conceitos que devem ser considerados no *design* de uma *engine*.

API Gráfica

A API gráfica é responsável por estabelecer uma conexão entre a aplicação e a GPU (*Graphics Processing Unit* - geralmente é uma placa de vídeo). A OpenGL[1], por exemplo, fornece funções para definir uma área de desenho na tela (*glViewport*), funções para desenhar nessa área (*glDrawArrays*) e funções para colorir essa área com uma cor específica (*glClearColor*) (Segal, 2022). Também é preciso notar que o processo de renderização é computacionalmente custoso (Akenine-Möller, 2018, p.29). Por isso, a escolha de uma API gráfica eficiente é essencial para uma *game engine*. DirectX[2] e Vulkan[3] são outros exemplos de APIs extensamente difundidas no ambiente comercial.

[1] Disponível em: https://opengl.org/

[2] Disponível em: https://devblogs.microsoft.com/directx/

[3] Disponível em: https://www.vulkan.org/

Shaders

Shaders são programas que descrevem como os dados enviados para GPU devem ser processados em cada estágio da *pipeline* gráfica. Na OpenGL, apenas os *shaders* de vértice e *pixel* (também chamado de fragmento) devem ser implementados, os estágios restantes são opcionais. Os Códigos 2.3 e 2.4 ilustram um código de vértice e fragmento, respectivamente.

Pipeline Gráfica

O objetivo da *pipeline* é renderizar uma imagem bidimensional por meio de uma câmera virtual (Akenine-Möller, 2018, p.11). A *pipeline* é uma sequência de diversos estágios, mas os mais relevantes no contexto de motores de jogos são os estágios da aplicação, *vertex* e *pixel*. Os outros estágios são opcionais e são implementados pela API gráfica (OpenGL).

Aplicação

O estágio da aplicação é responsável por definir quais dados, como vértices e texturas, serão enviados para a GPU e como esses dados devem ser processados (Akenine-Möller, 2018, p.13). Os vértices, por exemplo, podem ser armazenados em *buffers* e o processamento desses vértices pode ser especificado por meio de *shaders*.

Câmera

A câmera determina quais objetos na cena devem ser renderizados. Ela é representada por duas matrizes: a *view matrix* e a *projection matrix*. A *view matrix* utiliza informações sobre posição e rotação para determinar a localização e orientação da câmera no mundo. A *projection matrix* utiliza os valores de *zoom*, altura e largura da tela, *near* e *far* para determinar os limites da câmera.

Modelos

Modelos tridimensionais são representados por meio de triângulos que, por sua vez, são formados por três vértices. Em código, cada modelo é formado por uma lista de vértices e um *buffer* para armazenar esses vértices. O Código 2.2 ilustra uma lista de vértices que forma um triângulo.

Código 2.2 – Vértices de um triângulo.

```
1 vec3[] vertices = {
2     vec3(0.0f, 0.0f, 0.0f),  // Canto Inferior Esquerdo
3     vec3(1.0f, 0.0f, 0.0f),  // Canto Inferior Direito
4     vec3(0.5f, 1.0f, 0.0f)   // Topo Centro
5 };
```

Vertex Stage

Vértices são enviados para a *pipeline* por meio de *buffers* que armazenam uma lista de vértices. Eles são, então, processados por um *vertex shader*. O Código 2.3 ilustra esse *shader*. Nesse código, as matrizes da câmera (linhas 5 e 6) são utilizadas, juntamente com a posição do vértice (linha 3) para projetar o vértice em um espaço bidimensional. A matriz de modelo

(linha 7) também é utilizada para aplicar uma transformação em um modelo (translação, rotação, mudança de escala).

Código 2.3 – *Vertex Shader* (base_color.vs).

```
1  #version 460 core
2
3  layout(location = 0) in vec3 position;
4
5  uniform mat4 projection;
6  uniform mat4 view;
7  uniform mat4 model;
8
9  void main() {
10     gl_Position = projection * view * model * vec4(position, 1.0);
11 }
```

Pixel Stage

Existem outros estágios antes e depois do *vertex stage*, mas não são tão relevantes neste contexto. O último estágio que é preciso implementar é o *pixel stage*. Esse é um dos últimos estágios, onde os dados processados pelos estágios anteriores são enviados para o *pixel shader*, indicado no Código 2.4. Nesse exemplo, o valor de cada *pixel* é definido pela cor indicada pelo usuário (linha 5).

Código 2.4 – *Fragment (Pixel) Shader* (base_color.fs).

```
1  #version 460 core
2
3  layout(location = 0) out vec4 FragColor;
4
5  uniform vec3 color;
6
7  void main() {
8      FragColor = vec4(color, 1.0);
9  }
```

Com esses passos definidos, a pipeline gráfica está completa e pode ser executada. A API gráfica se encarrega de executar e processar os estágios da *pipeline* e produzir a imagem com os dados e *shaders* fornecidos.

3 Arquitetura da Engine Bloss 1

A arquitetura da *engine* foi inspirada na estrutura elaborada por Jason Gregory (Gregory, 2015, p.231). A *engine* é composta por subsistemas que devem ser inicializados/destruídos em ordem e são gerenciados por uma classe especial *Game*. Cada subsistema é responsável por gerenciar uma funcionalidade específica da *engine*, como a janela, o renderizador e o editor. O Código 3.1 mostra a inicialização dos subsistemas.

Código 3.1 – Construtor de *Game* (game.cpp).

```
1  Game::Game(const str &title, const u32 &width, const u32 &height)
2  {
3      // Create game instance
4      if (instance != nullptr) throw std::runtime_error("there can be only
           one instance of game");
5
6      instance = this;
7
8      // Create the window
9      window = std::unique_ptr<Window>(Window::create(title, width, height));
10     window->set_event_callback(BIND_EVENT_FN(Game::on_event));
11
12     // Create the renderer
13     renderer = std::unique_ptr<Renderer>(Renderer::create());
14     renderer->initialize();
15
16 // Create the editor
17 #if !defined(_RELEASE)
18     editor = std::make_unique<Editor>(*window.get());
19 #endif
20
21     // Create the audio engine
22     audio_engine = std::unique_ptr<AudioEngine>(AudioEngine::create());
23
24     // Create RNG engine
25     random_engine = std::make_unique<Random>();
26
27     // Register callbacks
28     EventSystem::register_callback<WindowCloseEvent>
29         (BIND_EVENT_FN(Game::on_window_close));
30     EventSystem::register_callback<WindowResizeEvent>
31         (BIND_EVENT_FN(Game::on_window_resize));
32     EventSystem::register_callback<KeyPressEvent>
33         (BIND_EVENT_FN(Game::on_key_press));
34     EventSystem::register_callback<MouseScrollEvent>
35         (BIND_EVENT_FN(Game::on_mouse_scroll));
36 }
```

Com os subsistemas inicializados (linhas 9-35), a engine está pronta para executar o *game loop* (Gregory, 2015, p.239) que é o método *run* na classe *Game*. O loop é encarregado de atualizar o estágio registrado até que não haja mais estágios para executar ou a pedido do usuário. A Figura 3.1 mostra o pseudocódigo do *loop* implementado pela Bloss1. A Seção A.1 detalha o *game loop*.

```
while (running)
{
    // 1. Calculate delta time (dt)
    dt = curr_time - last_time;

    // 2. Update current stage
    stage.update(dt)
    {
        // 2.1. Update the registered systems
        for (system in systems)
        {
            system.update(dt);
        }
    };

    // 3. Update the editor
    editor.update(ecs, dt);

    // 4. Update the window (swap buffers)
    window.update();
}
```

Figura 3.1 – Pseudocódigo do *game loop*.

Quando a aplicação sair do *game loop*, os subsistemas são encerrados e toda memória alocada é liberada. Como os subsistemas são criados com *smart pointers* (Whitney, 2021b) eles são destruídos automaticamente após *Game* ser destruído.

3.1 Estágios

Estágios podem ser interpretados como níveis ou fases em um jogo. Na Bloss1, fica a cargo do usuário definir quais sistemas do ECS (Capítulo 5) serão utilizados em cada estágio, além de carregar qualquer *asset* necessários por meio do *parser* de cena (Seção 4.9). O Código 3.2 ilustra a classe base *Stage*.

Código 3.2 – Classe base de um estágio (stage.hpp).

```
1  class Stage
2  {
3      public:
4          virtual ~Stage()
5          {
6          }
7
8          virtual void start() = 0;
9          virtual void update(f32 dt) = 0;
10
11         std::unique_ptr<ECS> ecs;
12 };
```

A classe base é extremamente simples. Como o funcionamento fica a cargo do usuário, esse design foi implementado para oferecer maior flexibilidade. Apenas os métodos *start* e *update* (linhas 8 e 9) devem ser implementados pela subclasse.

Código 3.3 – Inicialização de um estágio (test_stage.cpp).

```
1  void TestStage::start()
2  {
3      // Create the ECS
4      ecs = std::unique_ptr<ECS>(new ECS());
5
6      // Add systems in order of execution
7      ecs->add_system(physics_system);
8      ecs->add_system(state_machine_system);
9      ecs->add_system(camera_system);
10     ecs->add_system(animation_system);
11     ecs->add_system(render_system_forward);
12     ecs->add_system(cleanup_system);
13
14     // Load entities from file
15     SceneParser::parse_scene(*ecs,
           "bloss1/assets/scenes/test_stage.bloss");
16
17     auto &renderer = Game::get().get_renderer();
18     if (renderer.get_shadow_map() == nullptr)
19     {
20         renderer.create_skybox(
21             "bloss1/assets/textures/satara_night_no_lamps_4k.hdr",
22             AppConfig::skybox_config.skybox_resolution,
23             AppConfig::skybox_config.irradiance_resolution,
24             AppConfig::skybox_config.brdf_resolution,
25             AppConfig::skybox_config.prefilter_resolution,
26             AppConfig::skybox_config.max_mip_levels);
27
28         renderer.create_shadow_map(*ecs);
29         renderer.create_post_processing_passes(*ecs);
30     }
31
```

```
32      // Load configurations from file
33      SceneParser::parse_scene(*ecs,
            "bloss1/assets/scenes/bloss_config.bcfg");
34  }
```

O Código 3.3 mostra como inicializar um estágio. Primeiro, é criado um novo ECS (linha 4) e em seguida os sistemas necessários são registrados em ordem de execução (linhas 7-12). Uma cena é carregada (linha 15) e em seguida a *skybox* (linhas 20-26), as sombras e os passes de pós processamento (linha 28 e 29). As configurações para cada passe são obtidas de um arquivo de configuração (linha 33).

O funcionamento de um estágio (também chamado de lógica) fica a cargo do usuário. Os sistemas registrados na inicialização devem ser atualizados a cada *frame* e qualquer lógica extra também pode ser implementada nesse passo. O Código 3.4 mostra um ciclo de atualização desse mesmo estágio.

Código 3.4 – Atualização de um estágio (test_stage.cpp).

```
1  void TestStage::update(f32 dt)
2  {
3      BLS_PROFILE_SCOPE("update");
4
5      // Exit the stage
6      if (Input::is_key_pressed(KEY_ESCAPE))
7      {
8          Game::get().change_stage(nullptr);
9          return;
10     }
11
12     // Update all systems in registration order
13     auto &systems = ecs->systems;
14     for (const auto &system : systems) system(*ecs, dt);
15 }
```

Nesse estágio, os sistemas registrados na inicialização são atualizados em ordem de registro (linhas 13-14). Caso a tecla *ESC* seja pressionada, a aplicação é encerrada (linhas 6-10).

4 Subsistemas

Os subsistemas são responsáveis por realizar diversas funções na Bloss1. Podem ser implementados por um padrão de *Singleton* (Gregory, 2015, p.232) para garantir que exista apenas uma instância de um subsistema e que esse subsistema seja inicializado apenas uma vez. Na Bloss1, a classe *Game* se encarrega de inicializar/destruir os subsistemas na ordem correta (Capítulo 3). As seções seguintes detalham o funcionamento de cada subsistema.

4.1 Janela

A janela é responsável pelo tratamento de eventos do usuário com a aplicação (pressionamento de teclas, clique do mouse, redimensionamento de janela, dentre outros) e por apresentar os *frames* gerados pelo renderizador. O funcionamento da janela é apresentado em detalhes no Apêndice A.8.

4.2 Sistema de Input

O sistema de entrada é responsável por gerenciar eventos de janela relacionados ao uso de periféricos (mouse, teclado e controle). Esse sistema é implementado por meio da classe *Input*. Em grande parte, esse sistema serve como *wrapper* para as funções nativas da glfw.

Código 4.1 – Classe GlfwInput (input.hpp).

```
1  class GlfwInput : public Input
2  {
3      protected:
4          bool is_key_pressed_native(i32 keycode) override;
5          bool is_mouse_button_pressed_native(i32 button) override;
6          bool is_joystick_button_pressed_native(i32 joystick, i32 button)
               override;
7          f32 get_joystick_axis_value_native(i32 joystick, i32 axis)
               override;
8          std::pair<f32, f32> get_mouse_position_native() override;
9          f32 get_mouse_x_native() override;
10         f32 get_mouse_y_native() override;
11 };
```

O Código 4.1 mostra a assinatura da classe GlfwInput. Como dito anteriormente, o funcionamento desses métodos é em grande parte um *wrapper*. O Código 4.2 mostra a implementação do método *is_key_pressed_native* que utiliza a função da glfw *glfwGetKey* para verificar se uma tecla foi pressionada.

Código 4.2 – Implementação de métodos do input (input.cpp).

```
bool GlfwInput::is_key_pressed_native(i32 keycode)
{
    auto native_window = static_cast<GLFWwindow
       *>(Game::get().get_window().get_native_window());
    auto state = glfwGetKey(native_window, keycode);
    return state == GLFW_PRESS || state == GLFW_REPEAT;
}
```

Os outros métodos dessa classe seguem um padrão similar e podem ser analisados em detalhe no Apêndice A.9.

4.3 Sistema de Eventos

A classe EventSystem é encarregada de gerenciar eventos. Para isso, ela precisa registrar callbacks para cada tipo de evento e chamá-los quando os eventos ocorrerem. A relação entre a janela, o sistema de eventos e Game é um pouco complexa, mas pode ser abreviada da seguinte forma: a janela registra (*Window::set_event_callback*) um callback de Game (*Game::on_event*) que é chamado quando um evento ocorre. Além disso, os callbacks de game (*Game::on_window_close, Game::on_window_resize, etc*) são registrados no sistema de eventos (*EventSystem::register_callback*). Dessa forma, quando um evento ocorre, a janela chama o callback de Game. Game, por sua vez, detecta o tipo de evento e chama todos os callbacks registrados em EventSystem que possuem o mesmo tipo de evento.

Esse sistema pode parecer desnecessariamente complexo, mas a ideia por trás é que qualquer classe possa registrar um callback no sistema de eventos sem ter conhecimento da implementação da janela. Esses callbacks são, então, chamados em ordem de registro. O Código 4.3 mostra os trechos de Código relevantes.

Código 4.3 – Sistema de eventos (event.hpp).

```
Game::Game(...)
{
    // Create the window
    window = std::unique_ptr<Window>(Window::create(title, width, height));
    window->set_event_callback(BIND_EVENT_FN(Game::on_event));

    ...

    // Register callbacks
    EventSystem::register_callback<WindowCloseEvent
    (BIND_EVENT_FN(Game::on_window_close));
    EventSystem::register_callback<WindowResizeEvent>
    (BIND_EVENT_FN(Game::on_window_resize));
    EventSystem::register_callback<KeyPressEvent>
    (BIND_EVENT_FN(Game::on_key_press));
```

```
16      EventSystem::register_callback<MouseScrollEvent>
17      (BIND_EVENT_FN(Game::on_mouse_scroll));
18  }
19
20  ...
21
22  void Game::on_event(Event &event)
23  {
24      if (typeid(event) == typeid(WindowCloseEvent))
25          EventSystem::fire_event(static_cast<const WindowCloseEvent
                &>(event));
26
27      else if (typeid(event) == typeid(WindowResizeEvent))
28          EventSystem::fire_event(static_cast<const WindowResizeEvent
                &>(event));
29
30      else if (typeid(event) == typeid(KeyPressEvent))
31          EventSystem::fire_event(static_cast<const KeyPressEvent &>(event));
32
33      else if (typeid(event) == typeid(MouseScrollEvent))
34          EventSystem::fire_event(static_cast<const MouseScrollEvent
                &>(event));
35
36      else if (typeid(event) == typeid(MouseMoveEvent))
37          EventSystem::fire_event(static_cast<const MouseMoveEvent
                &>(event));
38
39      else
40          LOG_ERROR("invalid event type");
41  }
```

4.4 Renderizador

O renderizador pode ser considerado como o sistema mais relevante da engine, pois sua função principal é desenhar a cena. Isso inclui modelos, texturas, iluminação, partículas e texto. O renderizador foi implementado como uma API de modo retido (Radich; Satran, 2019) para maior performance e utiliza OpenGL 4.6 (Segal, 2022) e glew[1] como *backend* gráfico.

Código 4.4 – Classe Renderer (renderer.hpp).

```
1   class Renderer
2   {
3       public:
4           virtual ~Renderer()
5           {
6           }
```

[1] Disponível em: https://glew.sourceforge.net/

```
7
8           virtual void initialize() = 0;
9
10          virtual void set_viewport(u32 x, u32 y, u32 width, u32 height) = 0;
11          virtual void set_debug_mode(bool active) = 0;
12          virtual void set_blending(bool active) = 0;
13          virtual void set_face_culling(bool active) = 0;
14          virtual void set_tesselation_patches(u32 patches) = 0;
15
16          virtual void clear_color(const vec4 &color) = 0;
17          virtual void clear() = 0;
18          virtual void draw_indexed(RenderingMode mode, u32 count, const
               void *indices = 0) = 0;
19          virtual void draw_arrays(RenderingMode mode, u32 count) = 0;
20
21          virtual void create_skybox(const str &file,
22                                     const u32 skybox_resolution,
23                                     const u32 irradiance_resolution,
24                                     const u32 brdf_resolution,
25                                     const u32 prefilter_resolution,
26                                     const u32 max_mip_levels) = 0;
27
28          virtual void create_shadow_map(ECS &ecs) = 0;
29
30          virtual void create_height_map(
31              u32 width, u32 height, u32 min_tess_level, u32 max_tess_level,
                   f32 min_distance, f32 max_distance) = 0;
32
33          virtual void create_post_processing_passes(ECS &ecs) = 0;
34
35          virtual std::map<str, std::shared_ptr<Shader>> &get_shaders() = 0;
36          virtual std::vector<std::pair<str, std::shared_ptr<Texture>>>
               &get_textures() = 0;
37          virtual std::unique_ptr<FrameBuffer> &get_gbuffer() = 0;
38          virtual std::unique_ptr<Skybox> &get_skybox() = 0;
39          virtual std::unique_ptr<Quad> &get_rendering_quad() = 0;
40          virtual std::unique_ptr<ShadowMap> &get_shadow_map() = 0;
41          virtual std::unique_ptr<HeightMap> &get_height_map() = 0;
42          virtual std::unique_ptr<PostProcessingSystem>
               &get_post_processing() = 0;
43
44          // Must be implemented by the platform
45          static Renderer *create();
46  };
```

O Código 4.4 mostra a interface do renderizador. Todos os métodos devem ser implementados pela subclasse. Em grande parte, o renderizador é um *wrapper* para funções OpenGL. O Apêndice A.2 apresenta o renderizador OpenGL em detalhes.

4.4.1 Shaders

Bloss1 suporta compilar e executar shaders nativos da OpenGL (.glsl). Para isso, existe uma classe Shader que se encarrega de ler, compilar e executar esses shaders. Em grande parte, essa classe é um wrapper para funções da OpenGL que comunicam com shaders. O Apêndice A.5 mostra a classe Shader completa.

4.4.2 Sombras

Bloss1 implementa sombras utilizando *Cascaded Shadow Mapping* (CSM) (Árbócz, 2021), o que permite obter um nível de detalhamento inversamente proporcional à distância das sombras em relação à câmera. Essa variação é essencial para criar sombras bem detalhadas em qualquer distância. Se as sombras fossem criadas com uma única resolução, sombras de longe seriam bem detalhadas e sombras de perto não teriam resolução suficiente, acarretando em um resultado indesejável e no desperdício de recursos de processamento. As Figuras 4.1, 4.2 e 4.3 mostram sombras geradas pelo CSM com distância vertical de 50, 100 e 200 unidades, respectivamente.

Figura 4.1 – Sombra gerada pelo CSM com distância vertical de 50 unidades.

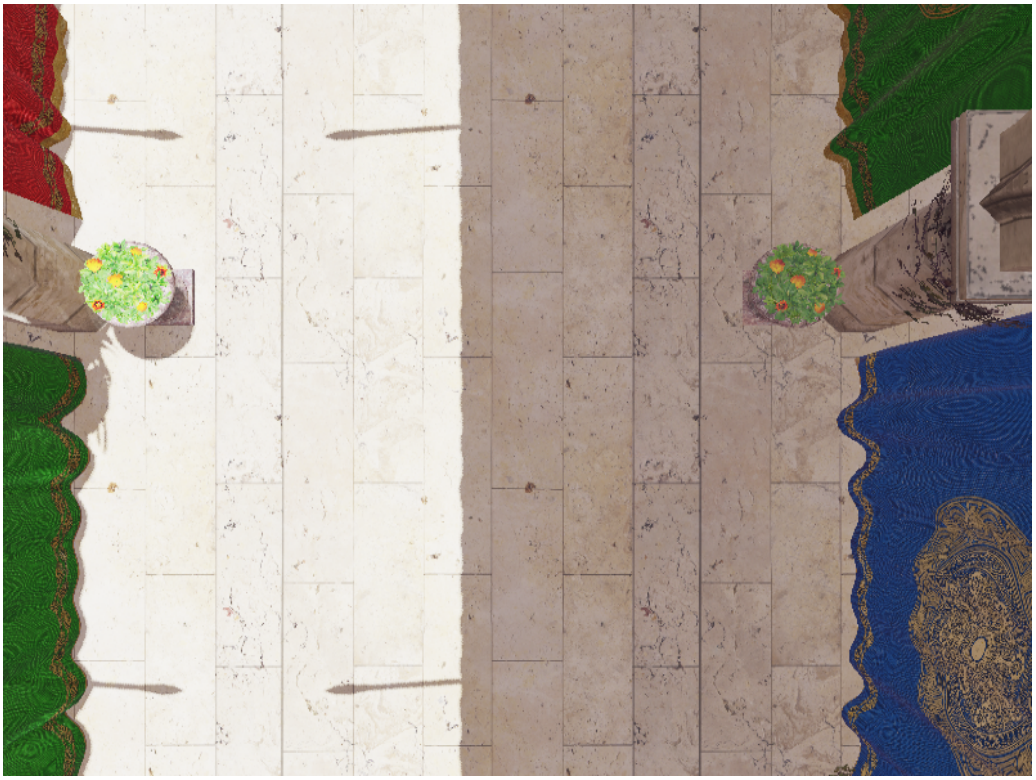


Figura 4.2 – Sombra gerada pelo CSM com distância vertical de 100 unidades.

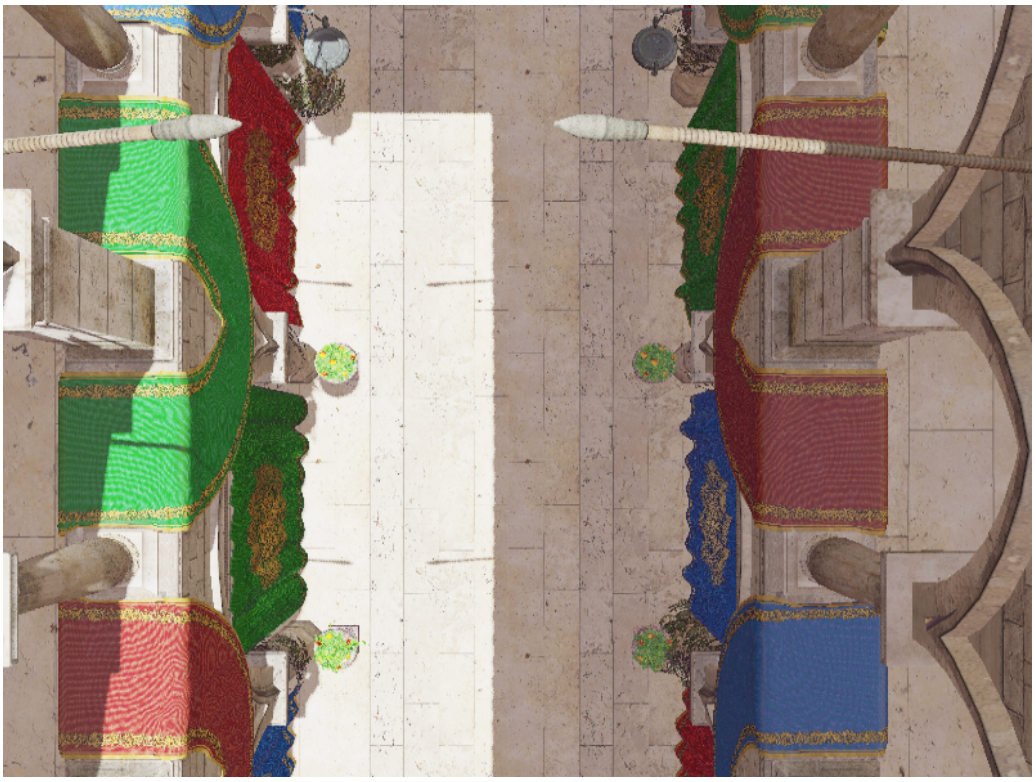


Figura 4.3 – Sombra gerada pelo CSM com distância vertical de 200 unidades.

Nestas figuras é possível observar que o nível de detalhe das sombras se mantém

constante mesmo com variação da distância. O Apêndice A.6 apresenta a classe *ShadowMap* em detalhes.

4.4.3 Skybox

É possível carregar arquivos .hdr como skyboxes (Akenine-Möller, 2018, p.547). Bloss1 também é capaz de gerar mapas de iluminação a partir dessas texturas (DeVries, 2017a) (DeVries, 2017b). A Figura 4.4 mostra a skybox gerada a partir da textura Satara Night.

Figura 4.4 – Detalhe da skybox gerada a partir da textura Satara Night.

4.4.4 Pós-processamento

Figura 4.5 – Detalhe do painel do sistema de pós processamento.

O sistema de pós-processamento permite aplicar efeitos de pós-processamento (*RenderPasses*) sequencialmente. O usuário pode definir quais e em qual ordem os efeitos são aplicados.

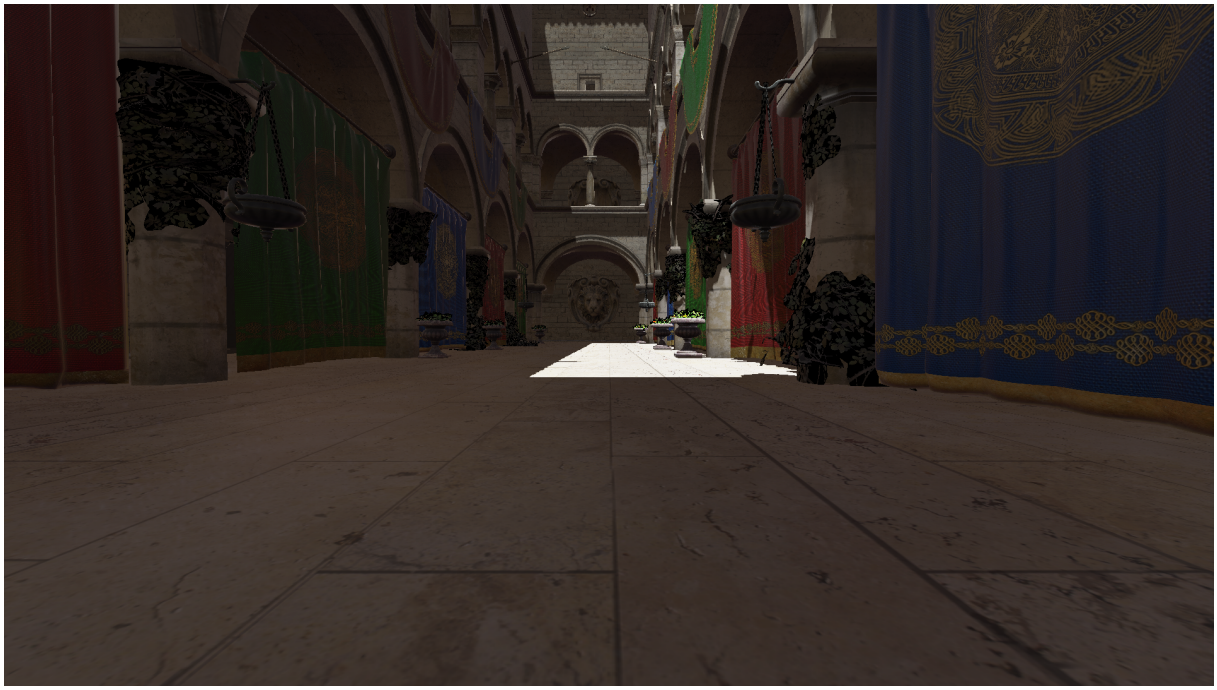


Figura 4.6 – Cena renderizada sem efeitos de pós-processamento (*BasePass*).

A Figura 4.6 mostra uma cena sem efeitos (cena base) para comparação com os outros efeitos. Bloss1 oferece nove efeitos diferentes:

- **FXAAPass –** Fast approximate antialiasing (Gregory, 2015, p.509). Diminui, sutilmente, o efeito de serrilhamento (*aliasing*) decorrente do processo de renderização. É de baixo custo, por isso é recomendado sempre ativar esse efeito. A Figura 4.7 mostra a cena base com FXAA ativo.

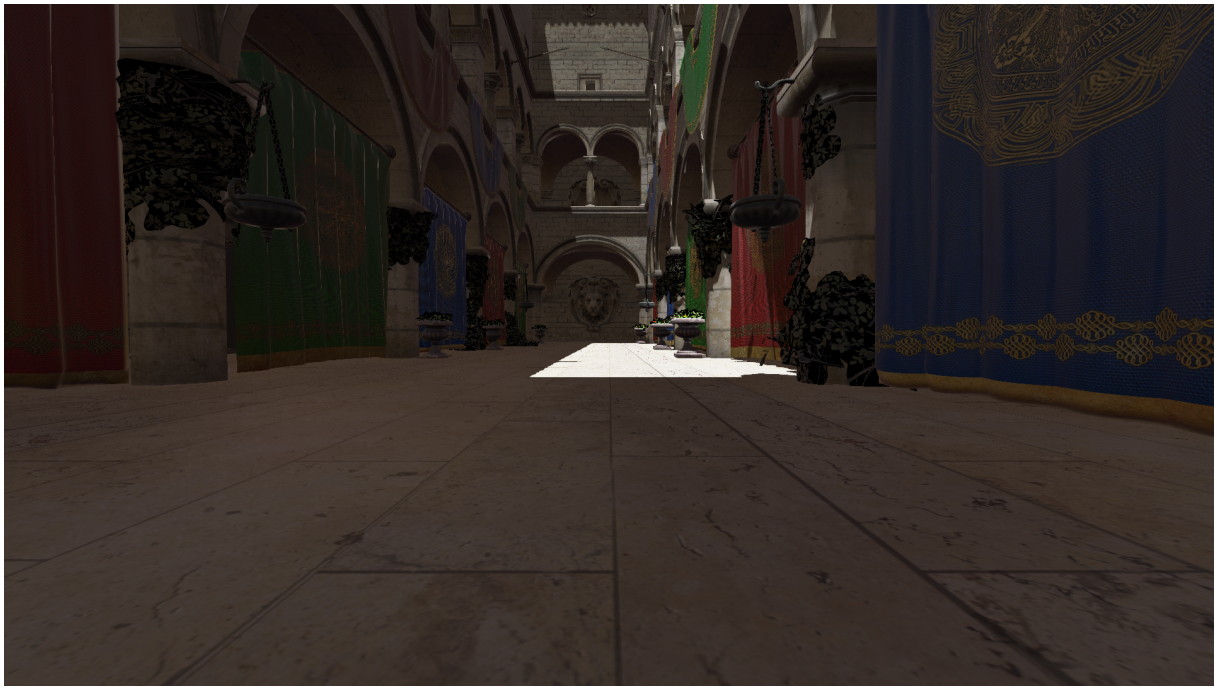


Figura 4.7 – Cena base renderizada com fxaa ativo.

- **BloomPass –** *Bloom* é um efeito causado pela luz ao atravessar a lente de uma câmera (Akenine-Möller, 2018, p.524). Esse efeito causa uma área que é intensamente iluminada a ’vazar’ nas regiões adjacentes, embaçando essas regiões. Na Bloss1, os parâmetros *spread*, *threshold* e *amount* controlam o tamanho da área adjacente ’vazada’, a intensidade mínima para ocorrer o efeito e a quantidade de embaçamento aplicada na Imagem. A Figura 4.8 mostra a cena base renderizada com esse efeito.

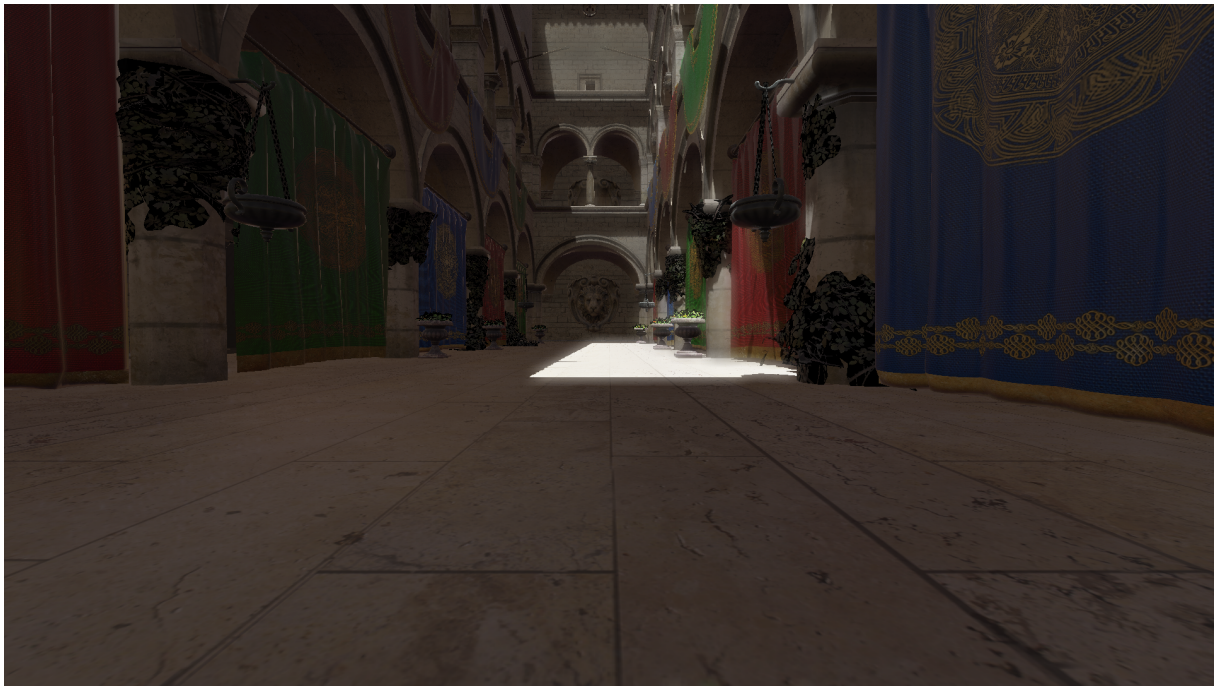


Figura 4.8 – Cena base renderizada com o efeito *bloom*.

- **FogPass –** O efeito de *fog* é um efeito de névoa que se prolonga entre uma distância mínima e máxima em relação à posição da câmera, ofuscando a visão (Akenine-Möller, 2018, p.600). A Figura 4.9 mostra a cena base renderizada com esse efeito.

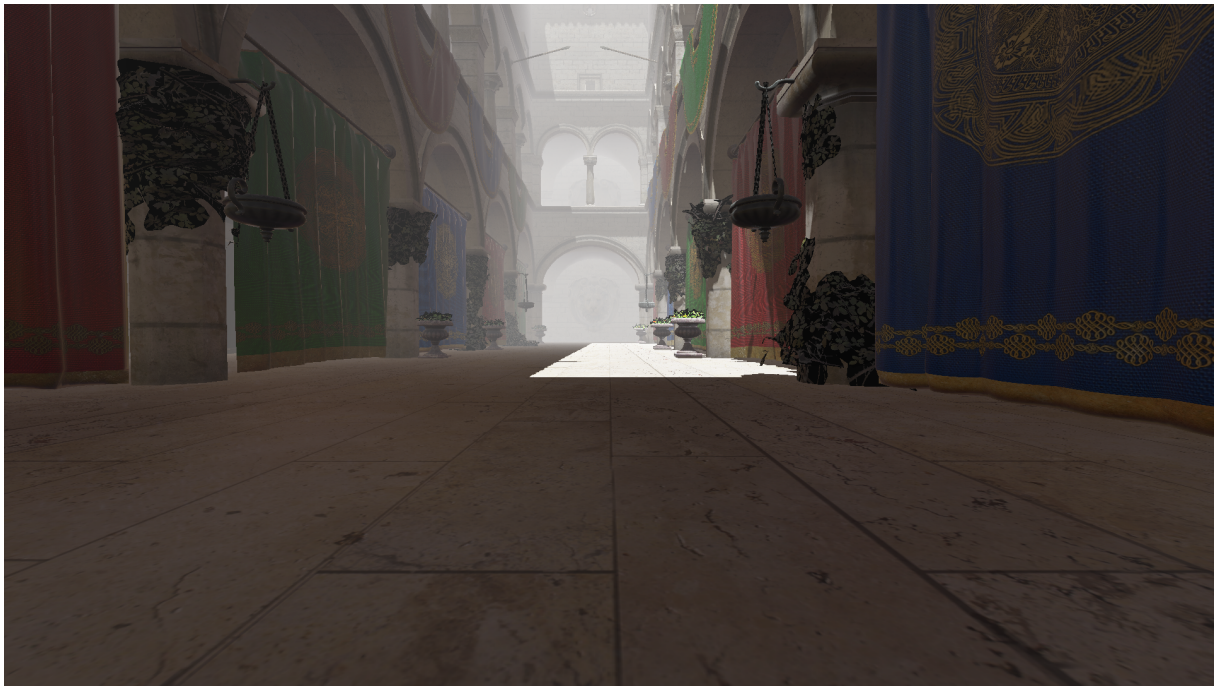


Figura 4.9 – Cena base renderizada com o efeito *fog*.

- **SharpenPass –** Esse efeito aumenta o contraste entre pixels, especialmente nas bordas

de um objeto (veja esse link). Quando usado com baixos valores, pode aumentar o realismo de uma cena. A Figura 4.10 mostra a cena base renderizada com esse efeito (exagerado para melhor visualização).

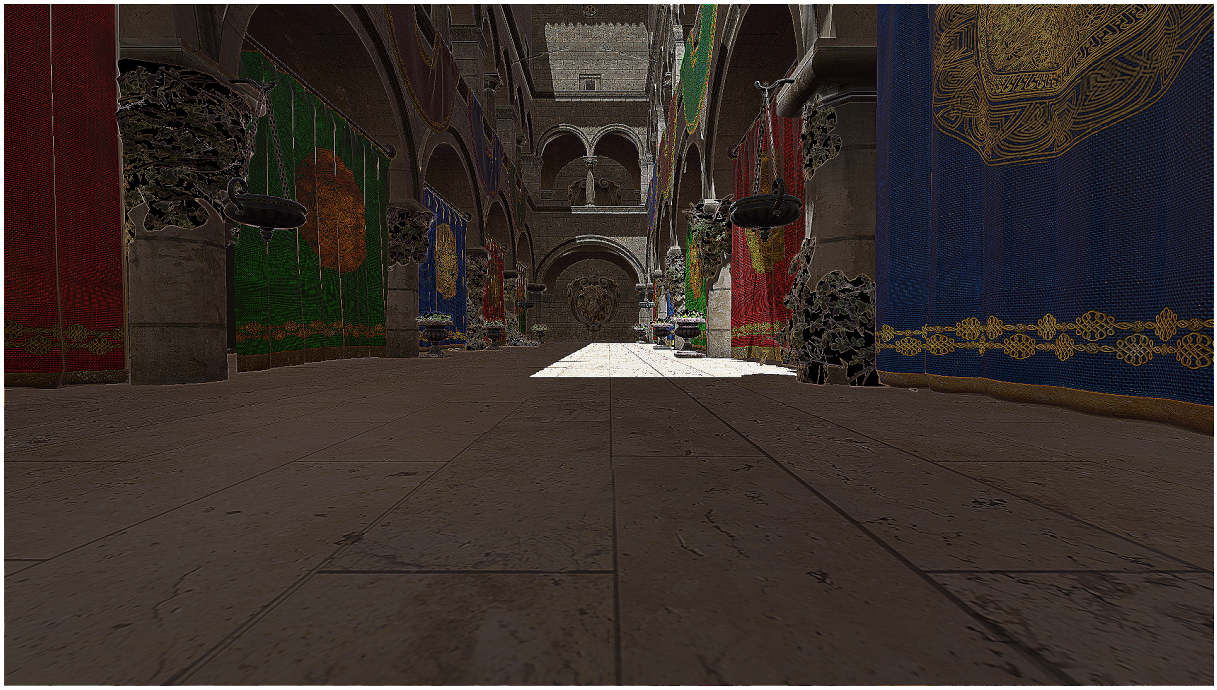


Figura 4.10 – Cena base renderizada com o efeito *sharpen*.

- **PosterizationPass –** *Posterization* é o efeito que reduz o número de gradiantes em uma Imagem, causando uma transição abrupta entre tons de cores (veja esse link). A Figura 4.11 mostra a cena base renderizada com 10 níveis de posterização.

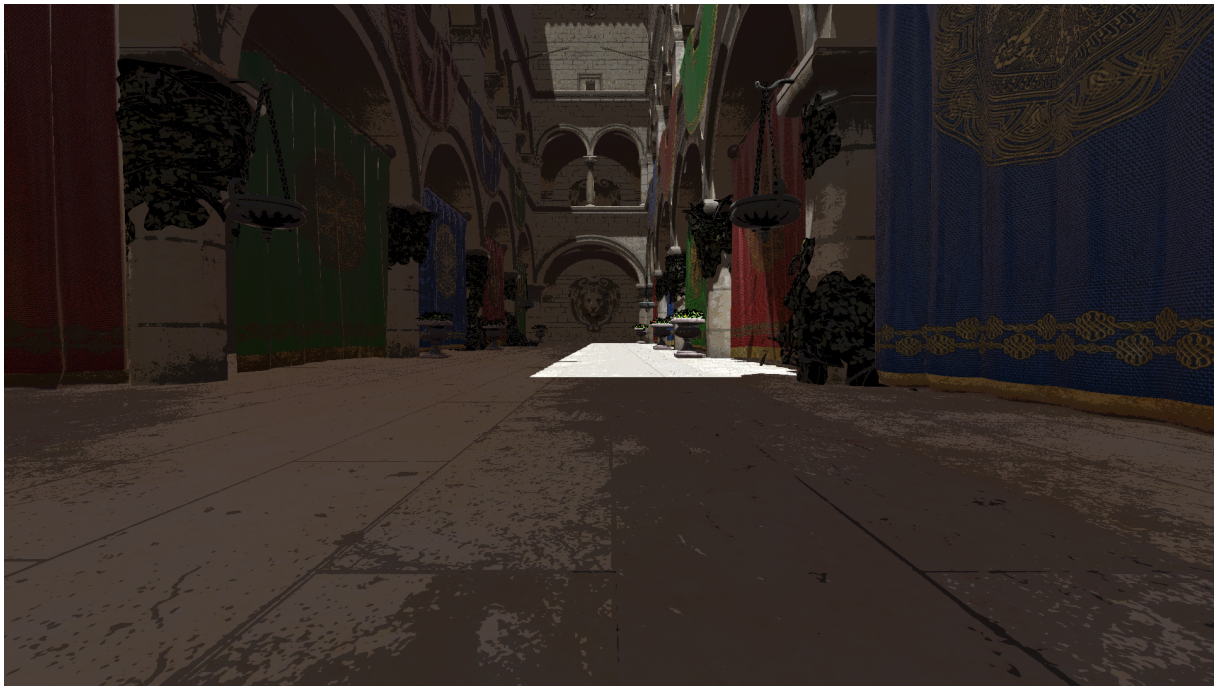


Figura 4.11 – Cena base renderizada com o efeito *posterization*.

- **PixelizationPass –** A pixelização é um processo que reduz a resolução de uma Imagem. Pode ser usado para causar um efeito retrô. Na Bloss1, o nível de redução pode ser controlado pelo pixel size. A Figura 4.12 mostra a cena base renderizada com pixel size 4.

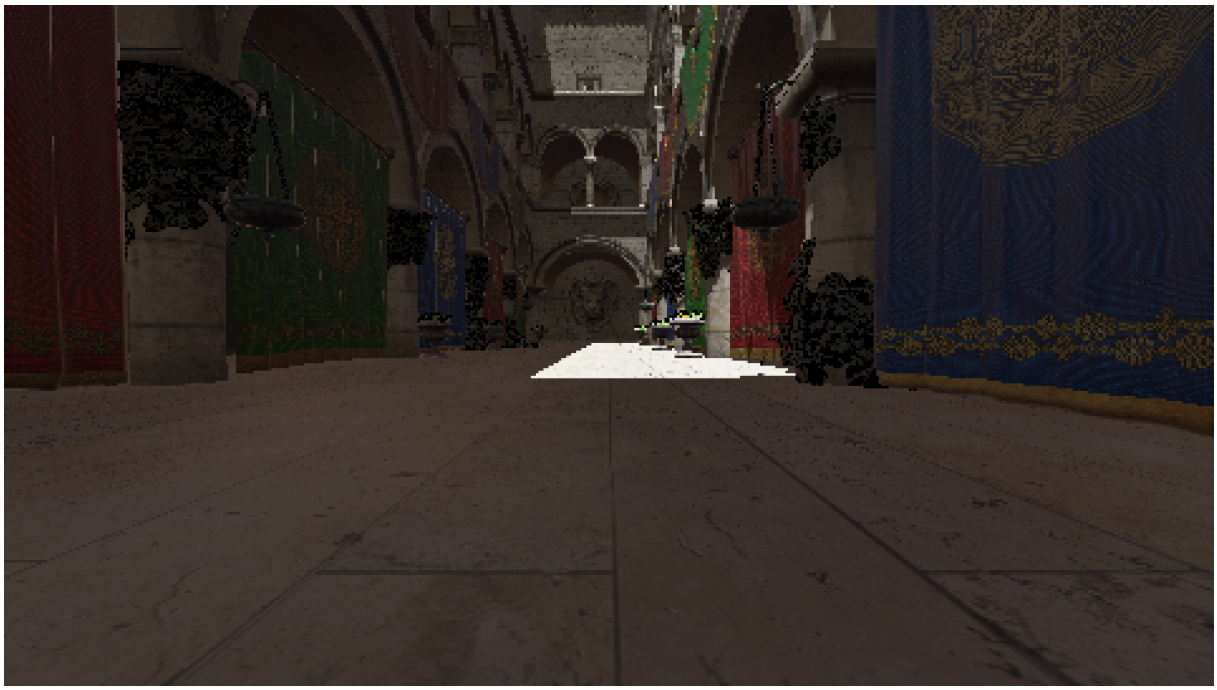


Figura 4.12 – Cena base renderizada com o efeito pixelização.

- **OutlinePass –** Bloss1 implementa o algoritmo de Sobel para detecção de bordas (Akenine-Möller, 2018, p.660). O parâmetro de *threshold* controla os limites para um pixel ser considerado uma borda e o parâmetro de cor controla a cor da borda. A Figura 4.13 mostra a cena base renderizada com threshold 0.8 e borda amarela.

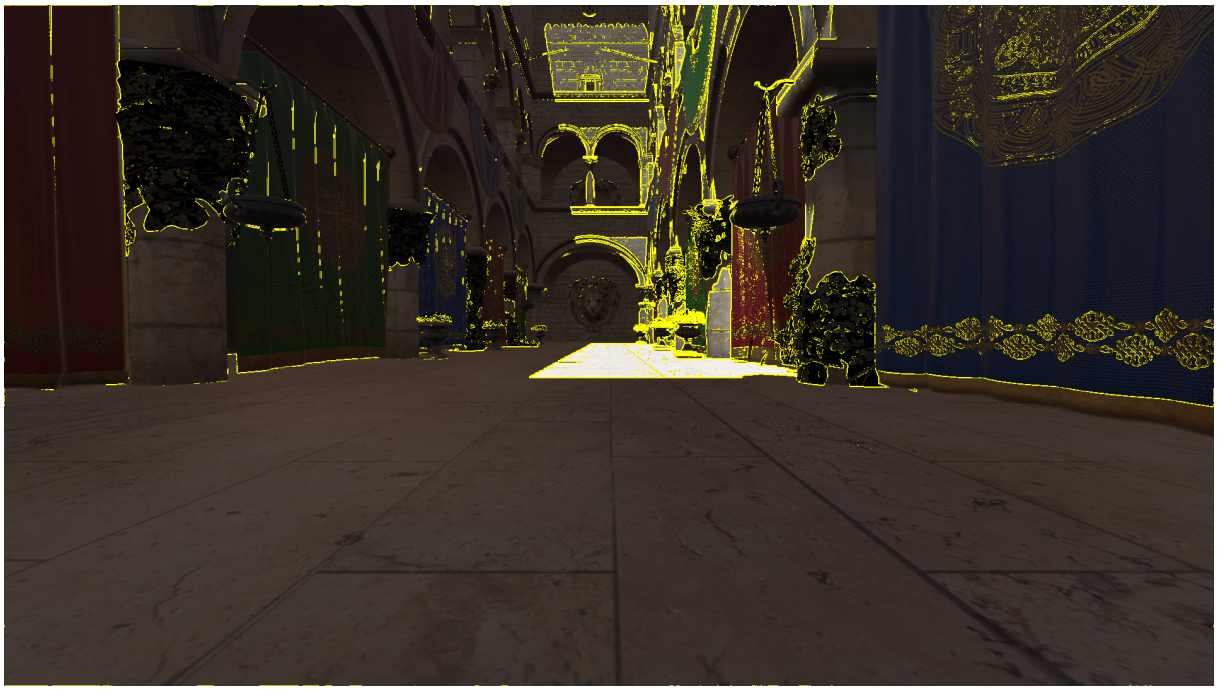


Figura 4.13 – Cena base renderizada com o efeito *outline*.

- **VignettePass –** *Vignette* é um efeito de escurecimento das bordas (Gregory, 2015, p.540). Pode ser usado para criar uma ambientação mais dramática ou para focar o olhar em uma região da tela. Na Bloss1, o parâmetro de *lens radius* controla o tamanho da vignette e *lens feathering* controla a transição entre a vignette e a cena. A Figura 4.14 mostra a cena base renderizada com *lens radius* 0.6 e *lens feathering* 0.35.

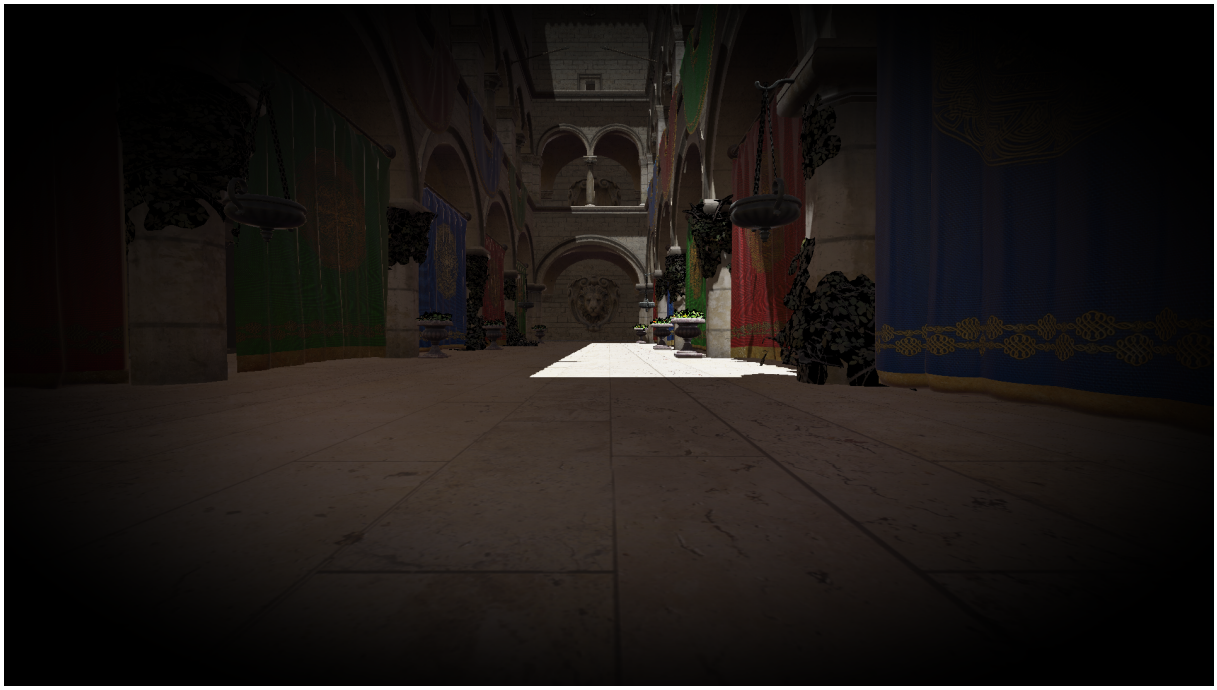


Figura 4.14 – Cena base renderizada com o efeito *vignette*.

- **KuwaharaPass –** O filtro de Kuwahara (Kyprianidis *et al.*, 2010) causa uma suavização da Imagem, o que provoca um efeito similar a uma pintura. A intensidade desse efeito pode ser controlada por meio do parâmetro *radius*. A Figura 4.15 mostra a cena base com o filtro aplicado e raio 4.

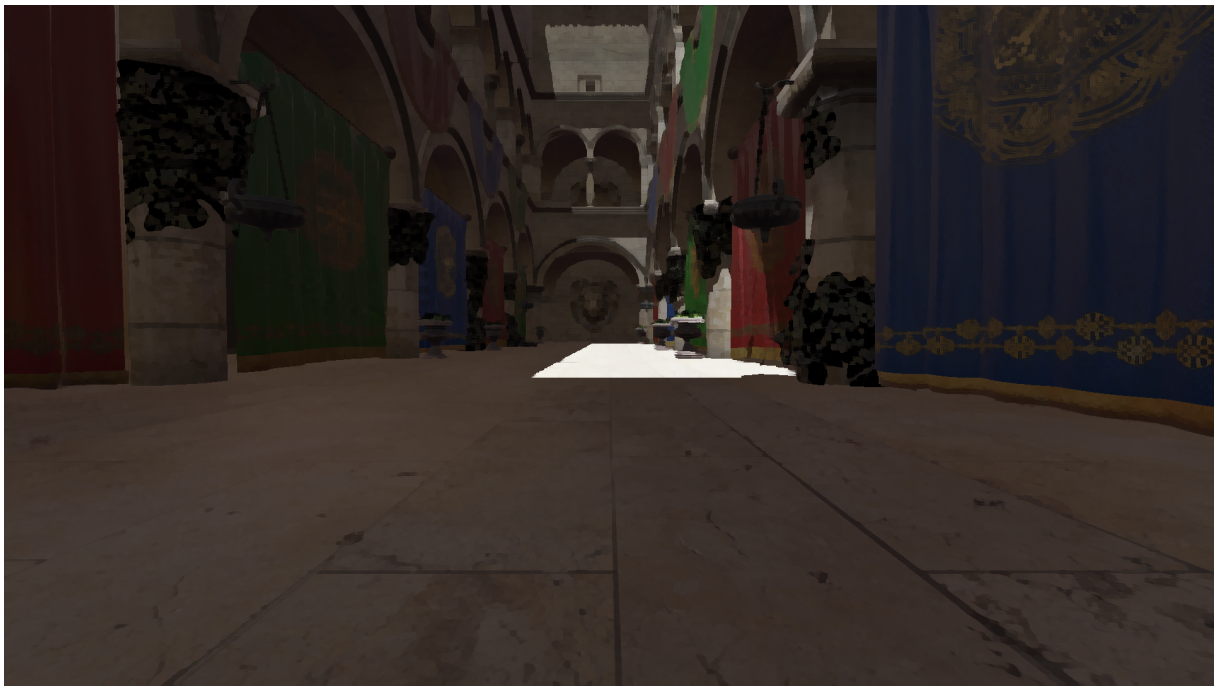


Figura 4.15 – Cena base renderizada com o efeito kuwahara.

4.5 Editor

O editor é responsável por criar uma interface gráfica de fácil uso para o usuário. Com ele é possível alterar valores de variáveis em tempo real e navegar pela cena. A engine utiliza a versão 1.90 da biblioteca ImGui (Cornut, 2024) para criar uma interface gráfica. A Figura 4.16 mostra uma visão geral do editor.

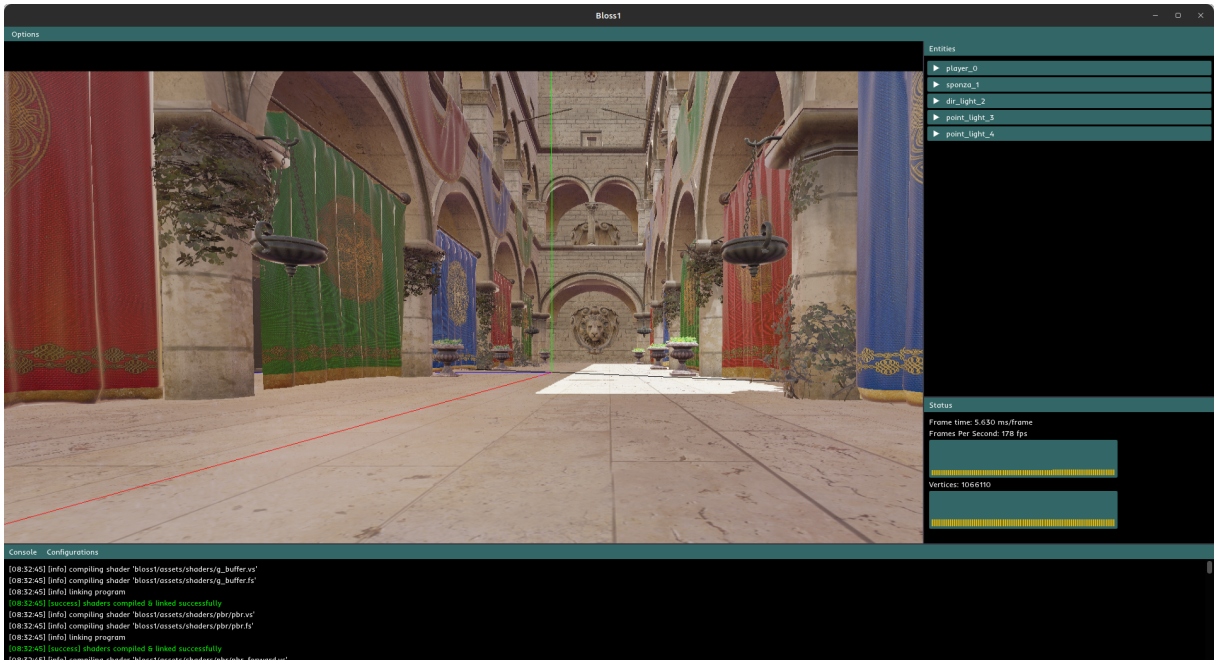


Figura 4.16 – Editor da Bloss1.

O editor é dividido em diversas janelas. Cada janela permite configurar e editar parâmetros diferentes da cena:

Console

O console mostra mensagens (*logs*) após operações relevantes, como carregamento de *assets* e *shaders*. Os *logs* possuem diferentes níveis, como *debug*, *success*, *warning* e *error*.

Entidades

A aba de entidades mostra todas as entidades da cena e seus componentes. Os parâmetros que podem ser modificados variam para cada componente. O componente de transformação, por exemplo, possui todos os valores de translação, rotação e escala ajustáveis.

Configurações

A janela de configurações permite alterar outros parâmetros da cena, como a *skybox*, os filtros de pós-processamento e *gizmos* de *debug*.

Status

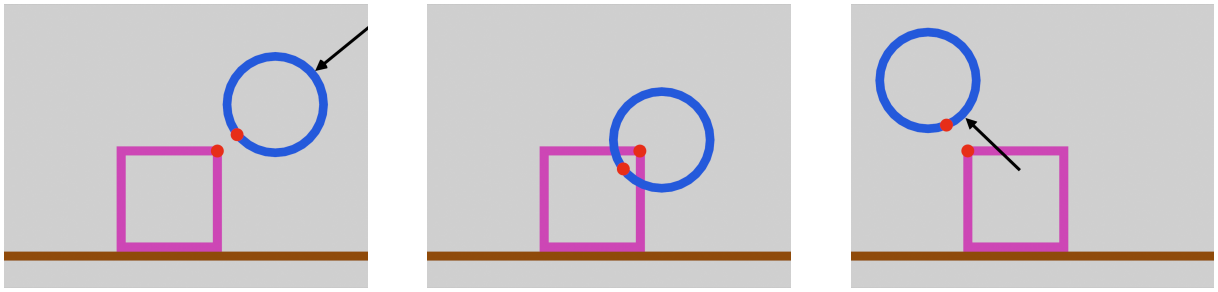
A aba de status mostra informações relevantes para análise de performance, como o tempo para renderizar um *frame* e o número de vértices renderizados.

Opções

A aba de opções abre um menu para salvar a configuração e/ou cena atual em um arquivo de texto.

4.6 Motor de Física

O motor de física é responsável por integrar forças e calcular a velocidade resultante em corpos rígidos. A *engine* utiliza *timestep* fixo, de forma que o renderizador e o motor de física são independentes um do outro (Fiedler, 2004a). A Figura 4.17 ilustra o processamento de uma colisão.



(a) Esfera se movendo em direção a uma caixa. A seta representa o sentido da velocidade.

(b) Colisão entre a esfera e a caixa com os pontos de contato em destaque.

(c) Esfera se movendo com a colisão corrigida - o sentido da velocidade foi alterado.

Figura 4.17 – Diagrama de uma colisão entre uma esfera e uma caixa (visão lateral).

O sistema de física pode ser dividido em três partes: integração de forças, teste de colisão e resolução de colisão.

Integração de forças

A primeira parte consiste em aplicar forças em um objeto por meio da integração de Euler semi explícita (Fiedler, 2004b).

Código 4.5 – Integração de forças (physics_system.cpp).

```
1  for (auto &[id, object] : objects)
2  {
3      object->mass = clamp(object->mass, MIN_MASS, MAX_MASS);
4
5      // Do not apply forces to immovable ojbects
6      if (!colliders[id]->immovable)
7      {
8          // Apply forces
9          object->force += vec3(0.0f, object->mass * -GRAVITY, 0.0f);
10         object->velocity += (object->force / object->mass) * dt;
11
```

```
12          // Apply deceleration
13          object->velocity.x = apply_deceleration(object->velocity.x,
                DECELERATION, object->mass, dt);
14          object->velocity.y = apply_deceleration(object->velocity.y,
                DECELERATION, object->mass, dt);
15          object->velocity.z = apply_deceleration(object->velocity.z,
                DECELERATION, object->mass, dt);
16
17          object->velocity = clamp(object->velocity,
                -object->terminal_velocity, object->terminal_velocity);
18          transforms[id]->position += object->velocity * dt;
19      }
20
21      // Reset forces
22      object->force = vec3(0.0f);
23  }
```

O Código 4.5 mostra o processo para calcular a posição resultante de um objeto a partir das forças aplicadas a ele. Primeiro, é calculada a força exercida pela gravidade (linha 8). Em seguida, é feito o cálculo da velocidade (linha 10) que pode ser influenciada pelas forças de atrito (linhas 13-15). A velocidade final é mantida dentro dos valores terminais para aquele objeto (linha 17). Por fim, é possível utilizar o valor da velocidade no cálculo da posição (linha 18). As forças são zeradas ao final desse processo (linha 22).

Teste de colisão

A segunda parte consiste em verificar se uma colisão entre dois objetos ocorre. Objetos podem ser um de dois tipos distintos: caixa ou esfera. Então, é preciso verificar se ocorre uma colisão entre esfera/esfera, esfera/caixa ou caixa/caixa (ver Apêndice A.4). Os algoritmos variam para cada caso, mas o resultado de um teste de colisão entre dois objetos A e B é sempre uma *struct Collision* (Código 4.6) que armazena o ponto na superfície de A mais próximo do objeto B e o ponto na superfície de B mais próximo do objeto A (pontos vermelhos ilustrado na Figura 4.17). Com esses dois pontos é possível calcular o *overlap* entre dois objetos. Se houver *overlap*, é preciso reposicionar os objetos. Caso contrário, não se faz nada.

Código 4.6 – Struct Collision (physics_system.cpp).

```
1  struct Collision
2  {
3          vec3 point_a;
4          vec3 point_b;
5          bool has_collision;
6  };
```

Resolução de colisão

Se há colisão, ela deve ser resolvida. Como os pontos de contato já foram calculados no passo anterior, a única operação a ser feita é mover os objetos no sentido contrário ao vetor de penetração. O Código 4.7 mostra a resolução de colisão entre dois objetos.

Código 4.7 – Resolução de colisão (physics_system.cpp).

```
1  void solve_collision(ECS &ecs, u32 id_a, u32 id_b, Collision collision)
2  {
3      vec3 delta = collision.point_a - collision.point_b;
4      f32 dist = length(delta);
5      vec3 normal = delta / dist;
6
7      auto collider_a = ecs.colliders[id_a].get();
8      auto collider_b = ecs.colliders[id_b].get();
9
10     auto trans_a = ecs.transforms[id_a].get();
11     auto trans_b = ecs.transforms[id_b].get();
12
13     auto displacement_a = normal * dist * 0.5f;
14     auto displacement_b = normal * dist * 0.5f;
15
16     if (!collider_a->immovable)
17     {
18         auto object_a = ecs.physics_objects[id_a].get();
19         object_a->velocity = object_a->velocity - (dot(object_a->velocity,
               normal) * normal);
20     }
21
22     else
23     {
24         displacement_a = vec3(0.0f);
25         displacement_b *= 2.0f;
26     }
27
28     if (!collider_b->immovable)
29     {
30         auto object_b = ecs.physics_objects[id_b].get();
31         object_b->velocity = object_b->velocity - (dot(object_b->velocity,
               normal) * normal);
32     }
33
34     else
35     {
36         displacement_a *= 2.0f;
37         displacement_b = vec3(0.0f);
38     }
39
40     trans_a->position += displacement_a;
41     trans_b->position -= displacement_b;
42 }
```

Primeiro, é feito o cálculo do vetor de penetração, ou seja, quanto de um objeto está dentro do outro (linha 3). O vetor normal representando o sentido do vetor de penetração também é calculado (linhas 4-5).

Por padrão, ambos os objetos em colisão são, na mesma proporção, movidos no sentido

contrário à normal (linhas 13-14 e 40-41). Além disso, uma força contrária à normal também é aplicada na velocidade do objeto (18-19), provocando um efeito de *bounce* (ilustrado na Figura 4.17c) após a colisão. Se um objeto for imóvel, sua posição permanece inalterada (linha 25). Feito isso, o processo de física é encerrado.

4.7 Motor de Áudio

O objetivo de um motor de áudio é gerenciar e reproduzir áudios, podendo ainda aplicar efeitos diversos (eco, *doppler*, reverberação, etc). A *engine* utiliza a versão 20200207 da biblioteca Soloud (Komppa, 2013) para carregar e reproduzir áudios em 3D. O Código 4.8 mostra a classe *AudioEngine*.

Código 4.8 – Motor de Áudio (audio_engine.hpp).

```
1  class AudioEngine
2  {
3      public:
4          virtual ~AudioEngine()
5          {
6          }
7
8          virtual void load(const str &name, const str &path, bool looping =
               false) = 0;
9          virtual void play(const str &name,
10                           const vec3 &position = vec3(0.0f),
11                           const vec3 &velocity = vec3(0.0f),
12                           f32 volume = 1.0f) = 0;
13         virtual void play_dist(const str &name,
14                                const vec3 &position = vec3(0.0f),
15                                const vec3 &velocity = vec3(0.0f),
16                                f32 distance_from_source = 0.0f,
17                                const f32 max_dist = 1000.0f) = 0;
18         virtual void stop(const str &name) = 0;
19         virtual void stop_all() = 0;
20         virtual void fade_to(const str &name, const f32 volume, const f64
               time) = 0;
21         virtual void set_echo_filter(const str &name, f32 delay, f32
               decay) = 0;
22
23         static AudioEngine *create();
24 };
```

Como a *AudioEngine* é em grande parte um *wrapper* para SoLoud, o funcionamento pode ser melhor explorado na documentação da SoLoud (Komppa, 2013).

4.8 Gerador de Números Aleatórios

O gerador de números randômicos (*Random Number Generator* - RNG) é responsável por gerar números aleatórios em um dado intervalo. O RNG da engine é um *wrapper* do gerador de números mt19937 da stl do c++ e pode gerar números em ponto flutuante ou inteiros em um intervalo definido pelo usuário. O Código 4.9 mostra os métodos para gerar números em ponto flutuante (linhas 10-18) e inteiros (linhas 20-23).

Código 4.9 – Gerador de números aleatórios (random.cpp).

```
1  Random::Random()
2  {
3      random_engine.seed(std::random_device()());
4  }
5
6  Random::~Random()
7  {
8  }
9
10 f32 Random::get_float(f32 begin, f32 end)
11 {
12     assert(begin <= end);
13
14     // Number is a value between 0 and 1
15     f32 number =
16         static_cast<f32>(distribution(random_engine)) /
               static_cast<f32>(std::numeric_limits<uint_fast32_t>::max());
17     return begin + (end - begin) * number;
18 }
19
20 i32 Random::get_int(i32 begin, i32 end)
21 {
22     return static_cast<i32>(get_float(static_cast<f32>(begin),
           static_cast<f32>(end)));
23 }
```

4.9 Parser de Cena

O parser de cena (*SceneParser*) é capaz de ler e escrever arquivos de texto contendo a descrição de entidades em uma cena (.bloss) ou as configurações do sistema de pós-processamento (.bcfg). Entidades são definidas por colchetes e, em seguida, os componentes entre chaves. As configurações são definidas pelos sinais maior que, menor que e, em seguida, os passes de pós-processamento. O Código 4.10 mostra a estrutura desses arquivos.

Código 4.10 – Formato dos arquivos .bloss e .bcfg (bloss_format.txt).

```
1  // Format:
```

```
[entity_name]
{
    component_type1: component_arg1, component_arg2, component_arg3, ... ;
    component_type2: component_arg1, component_arg2, component_arg3, ... ;
    ...
}

// Configuration:
<cfg_name>
{
    cfg_1: arg_1, arg_2, ... ;
    cfg_2: arg_1, arg_2, ... ;
}
```

O parser lê um arquivo sequencialmente, linha a linha, mas não verifica a corretude dos dados. Se um valor errado for inserido (como um caractere em um vetor de ponto flutuante) ocorrerá erro de *runtime*. Se os dados inseridos forem corretos, os componentes serão criados e armazenados no ECS indicado. O Código 4.11 mostra a entidade *player* do arquivo *debug.bloss*.

Código 4.11 – Entidade *player* (debug.bloss).

```
[player]
{
  model: bloss1/assets/models/f90/f90.fbx, 0;
  transform: (122.802246, 7.000000, -8.566934), (-1.230041, -180.0,
      0.000000), (5.000000, 5.000000, 5.000000);
  physics_object: (-0.000277, 0.000000, 0.000542), (70.000000, 70.000000,
      70.000000), (0.000000, 0.000000, 0.000000), 5.000000;
  collider: box, (4.000000, 4.500000, 4.000000), (0.000000, -4.300000,
      0.000000), 0, 2, 15;
  camera: (10.000000, 5.000000, 25.000000), (0.000000, 1.000000,
      0.000000), 34.999996, 1.000000, 10000.000000, 7.500000;
  camera_controller: (250000.0, 350000.0, 250000.0), 10.0;
  hitpoints: 100.0;
  particle_system: sphere, 0, 10, 0.1, (122.802246, 7.000000, -8.566934),
      10.0;
}
```

4.10 Gerenciadores

Gerenciadores, também chamados de managers, são responsáveis por gerenciar o ciclo de vida de alguns recursos persistentes como texturas, modelos, fontes, áudios e *shaders*. Esses *managers* utilizam um padrão de *Singleton* (Gregory, 2015, p.232), e verificam se um recurso requisitado já está carregado por via de um *map* (Whitney, 2021a). Se o recurso estiver no *map*, um *shared pointer* para esse recurso é retornado. Caso contrário, o recurso é criado e depois retornado. Por meio desse sistema, é garantido que não há dados duplicados e que um

recurso novo é carregado do disco apenas uma vez. O Código 4.12 mostra o funcionamento do gerenciador de modelos.

Código 4.12 – Gerenciador de modelos (model_manager.cpp).

```
1  void ModelManager::load(const str &name, std::shared_ptr<Model> model)
2  {
3      models[name] = model;
4  }
5
6  std::shared_ptr<Model> ModelManager::get_model(const str &name)
7  {
8      if (exists(name))
9          return models[name];
10
11     else
12         throw std::runtime_error("model '" + name + "' doesn't exist");
13 }
14
15 bool ModelManager::exists(const str &name)
16 {
17     return models.count(name) > 0;
18 }
19
20 ModelManager &ModelManager::get()
21 {
22     static ModelManager instance;
23     return instance;
24 }
```

Um recurso é inserido na tabela de modelos por meio do método *load* (linhas 1-4). O carregamento de um recurso do disco é feito pela classe Model (Seção 5.2.6). O método *get_model* (linhas 6-13) retorna um ponteiro para o modelo requisitado se existir, ou cria uma exceção se não existir. O método *exists* (linhas 15-18) verifica se um modelo está na tabela e o método *get* (linhas 20-24) retorna uma referência para a única instância do gerenciador.

5 Entity Component System - ECS

O Sistema de componente e entidade (em inglês *Entity Component System*) é uma forma flexível de representar um mundo virtual. O ECS da engine foi feito de uma forma orientada a dados (Fabian, 2018, p.83) para maior performance. Nessa implementação, sistemas são funções e entidades são representadas por um identificador (inteiro sem sinal). Os componentes são representados apenas por dados e são armazenados em tabelas.

As tabelas de componentes utilizam *smart_pointers* (Whitney, 2021b) extensivamente, o que garante a liberação da memória alocada quando um par id-componente é removido da tabela. O Código do ECS pode ser observado em sua íntegra no Apêndice A.3.

5.1 Sistemas

Um sistema é uma função que possui a seguinte assinatura:

Código 5.1 – Assinatura de um sistema (ecs.hpp).

```
1 // System: the logic bits
2 typedef void (*System)(ECS &ecs, f32 dt);
```

O primeiro argumento do Código 5.1 dá acesso aos componentes e o segundo é para eventuais cálculos atrelados ao tempo de um *frame*. Em algumas implementações, é feito uma *query* para filtrar entidades que possuem todos os componentes necessários (por exemplo, entidades que possuem os componentes de transformação e colisor).

Esse método garante que apenas entidades válidas são processadas por um sistema, mas o custo de uma busca aumenta muito com o número de filtros. Por isso, a filosofia adotada por Bloss1 não utiliza filtros. As entidades válidas para cada sistema são definidas implicitamente (uma entidade que é renderizada no sistema de renderização **precisa** de um componente de transformação além do componente de modelo).

5.1.1 Sistemas de Renderização

O sistema de renderização é responsável por desenhar uma cena com o shader especificado. Além disso, também é responsável por desenhar a interface de usuário (UI), textos e colisores (debug). Dois sistemas auxiliares (*render_system_forward* e *render_system_deferred*) são responsáveis por renderizar o restante da cena (sombras, *skybox*, terreno, pós-processamento) e atualizar o sistema de partículas.

O fluxo de execução para renderizar uma cena pode variar de acordo com a filosofia de renderização adotada (*forward* ou *deferred*), mas um modelo simples pode ser observado no Código 5.2.

Código 5.2 – Pseudocódigo para renderizar uma cena com *forward shading*.

```
// Reset the viewport
renderer.clear_color({0.0f, 0.0f, 0.0f, 1.0f});
renderer.clear();
renderer.set_viewport(0, 0, window_width, window_height);

// Bind the shaders and set uniforms
shader.bind();
shader.set_uniform4("projection", projection);
shader.set_uniform4("view", view);

// Bind vertex arrays
mesh.vertex_array.bind();

// Draw the mesh
renderer.draw_indexed(RenderingMode::Triangles, mesh.indices.size());
```

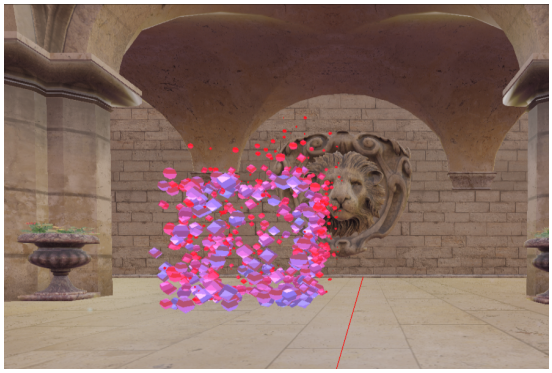
O Apêndice A.7 explora os modos de renderização em detalhe.

5.1.2 Sistema de Partículas

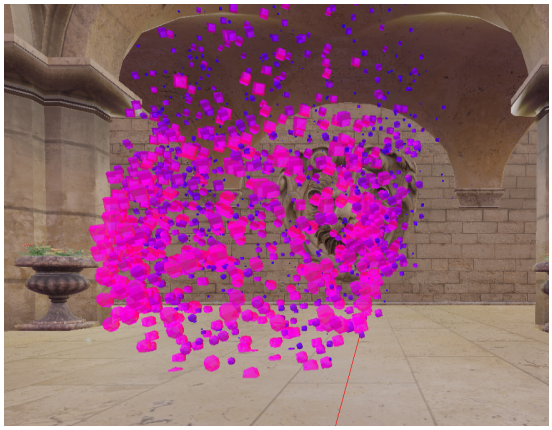
O sistema de partículas é um sistema especial no sentido em que não é chamado diretamente pela aplicação, mas por outro sistema - o sistema de renderização. Isso é necessário porque a ordem em que objetos são renderizados é diferente em um *forward renderer* e *deferred renderer*.

Dito isso, o sistema de partículas é, essencialmente, composto por duas partes: o emissor e as partículas. O emissor define o tipo da partícula que vai ser emitida (2D ou 3D) e onde uma partícula vai ser emitida. Na Bloss1, há dois tipos de emissores: caixa e esfera. O emissor de caixa emite partículas na superfície de uma caixa com as dimensões definidas e o emissor de esfera faz o mesmo na superfície de uma esfera com o raio dado. A partícula é uma struct que define o estado de uma partícula: posição, velocidade, cor, rotação e tempo de vida são alguns desses parâmetros.

A cada update o emissor atualiza o estado de todas as partículas, remove aquelas com tempo de vida expirado e renderiza aquelas que ainda estiverem ativas. As Figuras 5.1a e 5.1b mostram os emissores em formato de caixa e esfera, respectivamente.



(a) Emissor de partículas com formato de caixa.



(b) Emissor de partículas com formato de esfera.

Figura 5.1 – Emissores de partículas.

Um aspecto importante do sistema de partículas é o uso da memória. Para armazenar as partículas, utiliza-se uma *particle pool*, implementada por um *vector*. Criar ou deletar uma partícula dessa pool seria extremamente custoso (*vectors* são alocados na *heap* e um sistema de partículas pode ter milhares ou até centenas de milhares de partículas ativas). Por isso, o sistema implementado aloca espaço para todas as partículas durante a inicialização. Quando uma partícula atinge seu tempo de vida, essa partícula é reciclada. Não há deleção ou alocação na *pool* de partículas. Desse modo, também é garantido que quando um sistema de partículas é criado não ocorrerá erros de alocação ou falta de memória durante a execução da aplicação.

5.1.3 Sistema de Física

O sistema de física é responsável por aplicar forças em corpos rígidos e resolver eventuais colisões. O funcionamento é discutido em detalhe na Seção 4.6.

5.1.4 Sistema de Animação

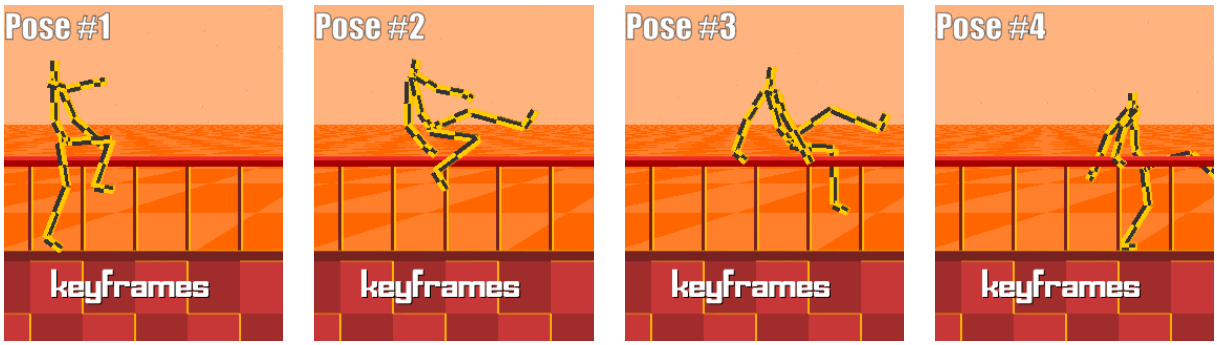


Figura 5.2 – Keyframes de uma animação.

O sistema de animação é responsável por atualizar as transformações de um modelo. Para isso, é feita a interpolação entre a transformação atual e a do próximo *frame*. A Figura 5.2[1] ilustra uma sequência de keyframes (também chamado de poses). Um relógio marca a duração de cada *frame*. O Código 5.3 detalha o funcionamento do sistema.

Código 5.3 – Sistema de animação.

```
1  auto &animations = ecs.transform_animations;
2  auto &transforms = ecs.transforms;
3  auto &timers = ecs.timers;
4  for (const auto &[id, animation] : animations)
5  {
6      auto &key_frames = animation->key_frames;
7      auto &curr_frame_idx = animation->curr_frame_idx;
8      auto curr_frame = key_frames[curr_frame_idx];
9      auto transform = transforms[id].get();
10     auto timer = timers[id].get();
11
12     // Set transform as curr frame transform
13     if (timer->time < curr_frame.duration)
14     {
15         f32 interpolation_factor = dt / (curr_frame.duration -
               timer->time);
16         transform->position = mix(transform->position,
               curr_frame.transform.position, interpolation_factor);
17         transform->rotation = mix(transform->rotation,
               curr_frame.transform.rotation, interpolation_factor);
18         transform->scale = mix(transform->scale,
               curr_frame.transform.scale, interpolation_factor);
19     }
20
21     // Update curr frame when frame duration ends
22     timer->time += dt;
23     if (timer->time > curr_frame.duration)
24     {
25         timer->time = 0.0f;
26         curr_frame_idx = (curr_frame_idx + 1) % key_frames.size();
27     }
28 }
```

As linhas 12-19 calculam a transformação atual de acordo com o tempo de frame restante. Quando o tempo expirar, o próximo frame é iniciado (linhas 22-27). Por padrão, quando todos os *frames* de uma animação são percorridos a animação é reiniciada.

[1] Fonte: https://learnopengl.com/img/guest/2020/skeletal_animation/poses.gif

5.1.5 Sistema de Câmera

O sistema de câmera atualiza as matrizes de projeção (Akenine-Möller, 2018, p.92) de todos as entidades que possuem câmera. O Código 5.4 mostra o sistema de câmera.

Código 5.4 – Sistema de câmera.

```
void camera_system(ECS &ecs, f32 dt)
{
    auto width = static_cast<f32>(Game::get().get_window().get_width());
    auto height = static_cast<f32>(Game::get().get_window().get_height());

    // Update all cameras
    auto &cameras = ecs.cameras;
    auto &transforms = ecs.transforms;
    for (const auto &[id, camera] : cameras)
    {
        auto transform = transforms[id].get();

        auto world_up = camera->world_up;
        auto target_offset = camera->target_offset;
        auto target_zoom = camera->target_zoom;

        auto target_position = transform->position;
        auto target_yaw = transform->rotation.y;
        auto target_pitch = transform->rotation.x;

        // Update target values
        auto target_front = vec3(cos(radians(target_yaw)) *
           cos(radians(target_pitch)),
                                 sin(radians(target_pitch)),
                                 sin(radians(target_yaw)) *
                                    cos(radians(target_pitch)));
        target_front = normalize(target_front);

        auto target_right = normalize(cross(target_front, world_up));
        auto target_up = normalize(cross(target_right, target_front));

        target_position = target_position + target_front *
           (-target_offset.z);
        target_position = target_position + target_up * target_offset.y;
        target_position = target_position + target_right * target_offset.x;

        auto &cam_position = camera->position;
        auto &cam_front = camera->front;
        auto &cam_up = camera->up;
        auto &cam_zoom = camera->zoom;
        auto &cam_near = camera->near;
        auto &cam_far = camera->far;

        auto &view_matrix = camera->view_matrix;
        auto &projection_matrix = camera->projection_matrix;

```

```
44          cam_position = target_position;
45          cam_front = target_front;
46          cam_up = target_up;
47          cam_zoom = target_zoom;
48
49          // Update view and projection matrices
50          view_matrix = look_at(cam_position, cam_position + cam_front,
                camera->up);
51          projection_matrix = perspective(radians(cam_zoom), width / height,
                cam_near, cam_far);
52      }
53  }
```

Nesse sistema, a câmera deve se alinhar com a posição e rotação da entidade (*target*) com um *offset* definido. Para isso, é preciso calcular os três vetores que representam a orientação da *target*: *foward vector*, *side vector* e *up vector* (linhas 22-28). Em seguida, é preciso aplicar o *offset* ao longo desses vetores (linhas 30-32).

O resultado desse processo fornece a posição e rotação da câmera, o que permite calcular as matrizes de projeção (linhas 41-51). Os métodos *look_at* e *perspective* são fornecidos pela biblioteca glm.

5.1.6 Sistema de Controle do Jogador

O sistema de controle do jogador é um exemplo de sistema que realiza lógicas complexas. Esse sistema é responsável por atualizar as transformações, a câmera, as animações, a máquina de estados (estado atual) e criar projéteis. Esse sistema não faz parte da engine em si. Seu propósito é demonstrar ao usuário como integrar funcionalidades específicas para um jogo.

5.1.7 Sistema de Limpeza

O sistema de limpeza deleta todas as entidades armazenadas na fila de exclusão. Uma entidade não deve ser deletada por nenhum sistema além desse. Caso uma entidade tenha expirado, deve ser apenas incluída na fila de exclusão. Esse *delay* para deletar uma entidade ocorre para evitar bugs (uma entidade usada pelo sistema de animação foi deletada no sistema de controle do jogador, por exemplo). Dessa forma, o sistema de exclusão é sempre o primeiro (ou último) sistema a ser chamado.

5.2 Componentes

Os componentes são estruturas que armazenam apenas dados. Qualquer lógica (podendo ou não envolver componentes diferentes) é incluída nos sistemas. O Código 5.5 mostra a classe base de um componente.

Código 5.5 – Interface de um componente (components.hpp).

```
// Component: contain the data
class Component
{
    public:
        virtual ~Component()
        {
        }
};
```

5.2.1 Transformação

O componente de transformação representa a posição, rotação e escala de um objeto em um mundo 3D. Essas informações são essenciais para calcular a matriz de transformação de um objeto. Também podem ser usadas para guardar posição e direção de uma luz direcional e posição de luzes em geral. A Figura 5.3 mostra o componente.

```
class Transform : public Component
{
    public:
        Transform(const vec3 &position = vec3(0.0f),
                  const vec3 &rotation = vec3(0.0f),
                  const vec3 &scale = vec3(1.0f))
            : position(position), rotation(rotation), scale(scale)
        {
        }

        vec3 position;
        vec3 rotation;
        vec3 scale;
};
```

Figura 5.3 – Componente de transformação.

5.2.2 Câmera

Uma câmera é responsável por controlar como o mundo é 'visto'. Para isso, é necessário calcular matrizes de projeção (Gregory, 2015, p.479). Opcionalmente, também é possível calcular um *offset* e *zoom* a partir de certo ponto para câmeras em terceira pessoa. A Figura 5.4 ilustra o *frustum* da câmera (Gregory, 2015, p.217). Na Bloss1, o eixo X é representado em vermelho, o eixo Y em verde e o eixo Z em azul.

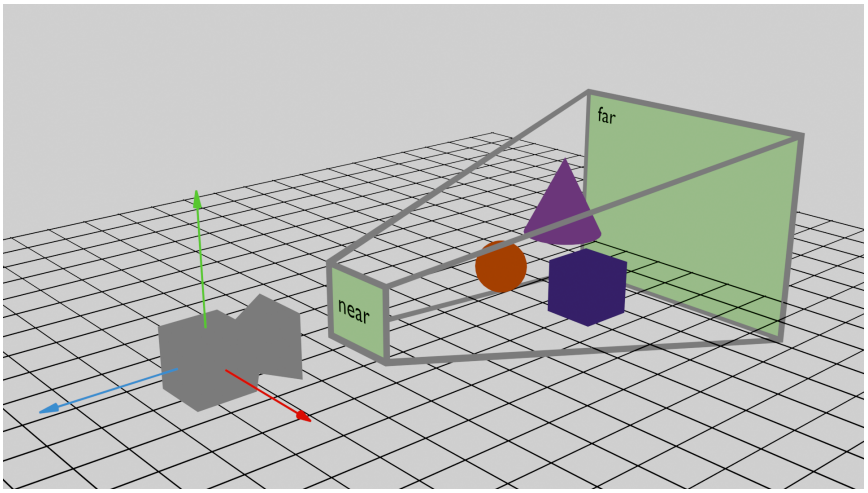


Figura 5.4 – Diagrama de uma câmera.

5.2.3 Controlador da Câmera

O nome desse componente é questão de debate. Embora o controlador também controle a câmera de uma entidade, ele não controla apenas a câmera, mas o comportamento de uma entidade. Dito isso, a câmera em um vácuo é estática. Para movê-la, utiliza-se o controlador. Esse componente é responsável por armazenar informação sobre a rotação atual, velocidade e sensibilidade dos controles. A rotação é armazenada nesse componente (além da rotação da câmera) caso o usuário queira usar *lerp* (Gregory, 2015, p.181) (o uso de *lerp* com o sistema de física ativo não é recomendado).

5.2.4 Luz Pontual

Uma luz pontual é representada por um ponto que emite raios de luz em todas as direções a partir de sua posição. Os parâmetros de cor de uma luz pontual podem variar dependendo do modelo de iluminação adotado, mas geralmente pode ser descrito por três parâmetros: cor ambiente, cor difusa e cor especular. A cor ambiente simula os raios de luz que são refletidos em um ambiente (iluminação indireta), tornando-o mais claro. A cor especular é aplicada em pontos de maior incidência da luz (onde a luz 'bate' mais forte). Por fim, a cor difusa é a cor de luz que ilumina uma superfície. A Figura 5.5[2] ilustra o funcionamento de uma luz pontual.

[2] Fonte: https://learnopengl.com/Lighting/Light-casters

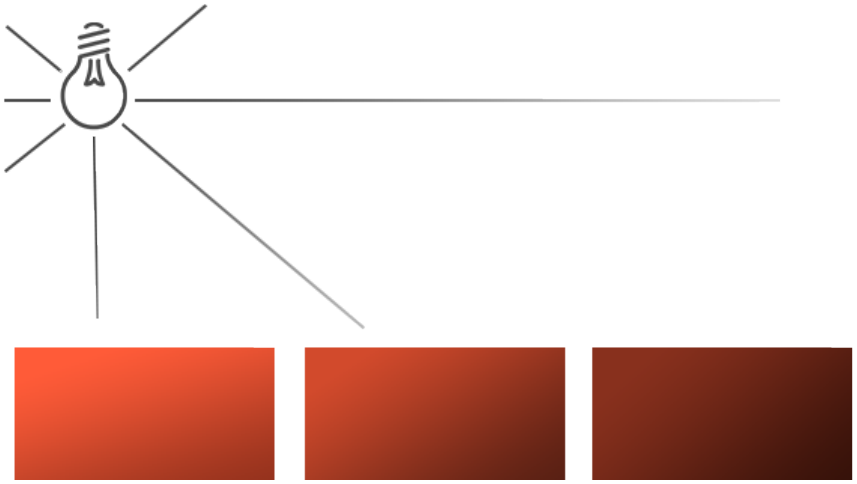


Figura 5.5 – Diagrama de uma luz pontual.

Os parâmetros *constant*, *linear* e *quadractic* são valores para calcular a atenuação da luz de acordo com a distância em um modelo *Phong* (Gregory, 2015, p.471). Em um modelo PBR (Akenine-Möller, 2018, p.293) a atenuação é calculada de acordo com a lei do inverso do quadrado (Akenine-Möller, 2018, p.111-112). Os Códigos 5.6 e 5.7 mostram os shaders de fragmento para cada modelo.

Código 5.6 – Atenuação da luz pontual em um modelo *Phong* (phong.fs).

```
// Input from vertex shader
in VS_OUT {
    vec2 TexCoords;
    vec3 Normal;
    mat3 TBN;
    vec3 FragPos;
    vec4 FragPosLightSpace;
} fs_in;

// PointLight
struct PointLight {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float constant;
    float linear;
    float quadratic;
};

...

// Attenuation
float distance = length(pointLight.position - fs_in.FragPos);
float attenuation = 1.0 / (pointLight.constant + pointLight.linear *
    distance + pointLight.quadratic * (distance * distance));
```

Código 5.7 – Atenuação da luz pontual em um modelo PBR (pbr.fs).

```
// Lights
const int numberOfLights = 16;
struct Lights {

    // Point lights
    vec3 pointLightColors[numberOfLights];
    vec3 pointLightPositions[numberOfLights];

    // Directional lights
    vec3 dirLightDirections[numberOfLights];
    vec3 dirLightColors[numberOfLights];
};

...

// Attenuation according to the inverse square law
float distance = length(lights.pointLightPositions[i] - FragPos);
float attenuation = 1.0 / (distance * distance);
```

5.2.5 Luz Direcional

A luz direcional é similar a luz pontual (cor ambiente, difusa e especular), mas os raios de luz simulam aqueles emitidos pelo Sol. Isso quer dizer que os raios emitidos são paralelos entre si e possuem a mesma direção. Além disso, essa luz não possui atenuação como a luz pontual. A Figura 5.6[3] apresenta um diagrama de uma luz direcional.

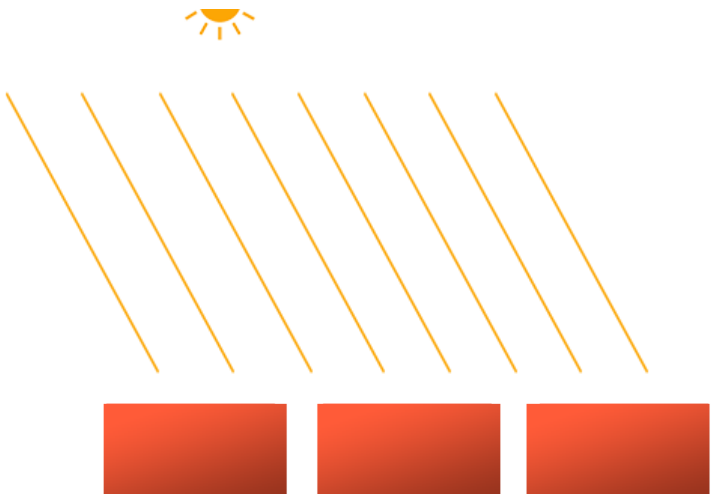


Figura 5.6 – Diagrama de uma luz direcional.

[3] Fonte: https://learnopengl.com/Lighting/Light-casters

5.2.6 Modelo

Um modelo (*Model*) é uma representação de um objeto 3D. Ele é composto por vértices, ossos, animações e texturas. Como um modelo é um componente complexo para ser representado por um ECS, o *ModelComponent* foi criado como *wrapper* para a classe *Model*.

A classe Model utiliza a biblioteca assimp para carregar modelos de diversos formatos. Informações sobre vértices, texturas, ossos e animações são, então, convertidas para um formato interno da Bloss1. A Figura 5.7 ilustra o modelo de uma caixa.

Figura 5.7 – Modelo *planter_box*.

5.2.7 Animador

O animador é responsável por atualizar animações esqueletais (Kushwah, 2020) de um modelo a cada frame. Também é possível transicionar entre animações (*cross_fade*) (Gregory, 2015, p.578). Essa funcionalidade é especialmente útil para complementar uma máquina de estados (Seção 5.2.14). A transição entre a animação estacionária e de andar, por exemplo, é extremamente comum e também é implementada no jogo F90 (Figura 6.1).

5.2.8 Objeto Físico

O *PhysicsObject* representa um corpo rígido em uma simulação de forças. É composto por velocidade, velocidade terminal, força e massa. Esse componente pode fazer parte do componente de transformação. Entretanto, nem toda entidade que possui transformação é um objeto físico e, por isso, foi feita essa separação.

5.2.9 Colisor

O colisor representa o volume de uma caixa ou esfera. Seu propósito é detectar quando um objeto entra em seu volume (*trigger*). Além disso, um colisor pode ser imóvel e possuir um offset relativo à posição da entidade. Isso é preciso porque o colisor serve como componente complementar ao objeto físico (objetos físicos não possuem forma) e objetos como o chão ou paredes devem ser imóveis. A Figura 5.8 mostra um colisor esférico em contato com o chão.

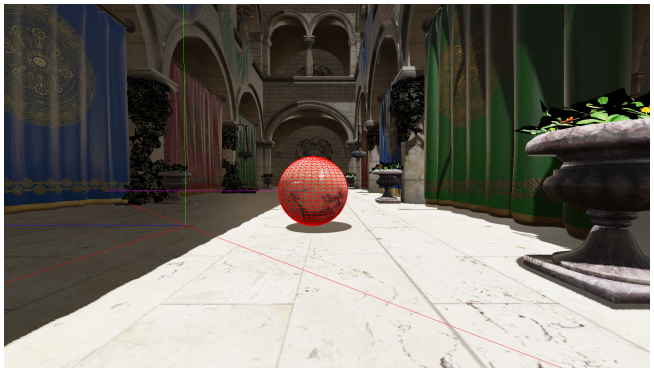


Figura 5.8 – Colisor em formato de esfera.

5.2.10 Relógio

O relógio (*Timer*, Código 5.8) é um simples contador de tempo. Uma variável de ponto flutuante armazena os incrementos de *dt* para contar os segundos. A contagem também pode ser decrescente, a decisão é do usuário.

Código 5.8 – Componente *Timer*.

```
class Timer : public Component
{
    public:
        Timer(f32 time = 0.0f) : time(time)
        {
        }

        f32 time;
};
```

5.2.11 Som

O som (*Sound*, Código 5.9) representa um áudio. Para carregar um som, é preciso do arquivo, nome e volume. Além disso, é possível escolher se o áudio é tocado em *loop* e se é para ser tocado assim que o carregamento terminar.

Código 5.9 – Componente *Sound*.

```
class Sound : public Component
{
    public:
        Sound(const str &file, const str &name, f32 volume, bool play_now,
            bool looping)
            : file(file), name(name), volume(volume), play_now(play_now),
                looping(looping)
        {
        }
```

```
8
9           str file;
10          str name;
11          f32 volume;
12          bool play_now;
13          bool looping;
14  };
```

5.2.12 Texto

O texto (*Text*) é um componente que representa uma string de caracteres. Possui uma fonte, o texto a ser escrito, uma cor, posição e tamanho. A classe *Font* é uma classe auxiliar para carregar arquivos de fontes e renderizar o texto com as características definidas. A Bloss1 utiliza a biblioteca freetype para carregar os arquivos de fontes. A Figura 5.9 mostra um texto renderizado com a fonte Marske.ttf.

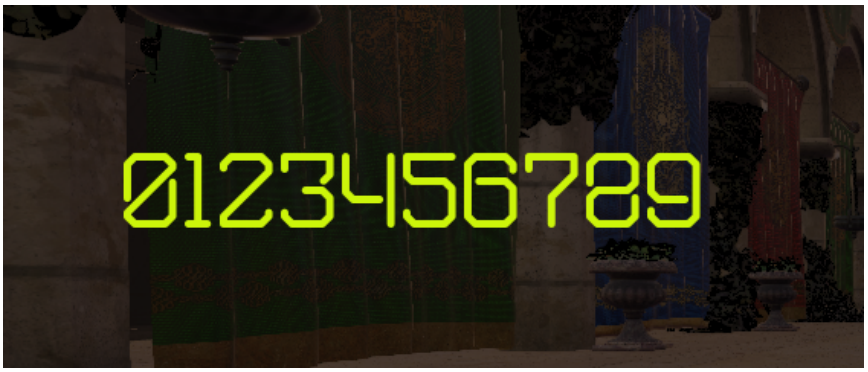


Figura 5.9 – Texto que utiliza a fonte Marske.ttf.

É importante destacar que um componente de texto é sempre renderizado por cima da cena e não possui rotação. É comparável a um *overlay*.

5.2.13 Transformação para Animação

Na Bloss1, uma sequência de key frames forma uma animação (Gregory, 2015, p.558). Cada key frame possui uma transformação e uma duração. O componente de *TransformAnimation* armazena uma sequência de key frames que são interpolados sequencialmente. O funcionamento é detalhado no sistema de animação (Seção 5.1.4).

5.2.14 Máquina de Estados

A máquina de estados armazena o estado atual de uma entidade. Essa máquina é utilizada para controlar o estado de uma entidade. Em particular, é útil para transicionar entre animações (Gregory, 2015, p.621). Um estado possui três transições: *enter* que é chamado

quando a máquina entra em um estado; *update* que atualiza as animações da entidade e *exit* que encerra qualquer operação pendente.

Código 5.10 – Classe base State (state_machine.hpp).

```
1  class State
2  {
3      public:
4          virtual void enter(ECS &ecs, u32 id, const str &state);
5          virtual void update(ECS &ecs, u32 id, f32 dt);
6          virtual void exit();
7
8      protected:
9          SkeletalAnimation *last_animation = nullptr;
10         SkeletalAnimation *curr_animation = nullptr;
11 };
```

O Código 5.10 mostra a classe base State com os métodos *enter*, *update* e *exit*. A implementação desses métodos fica a cargo do usuário. Ponteiros para as animações são membros protegidos para permitir acesso pela subclasse.

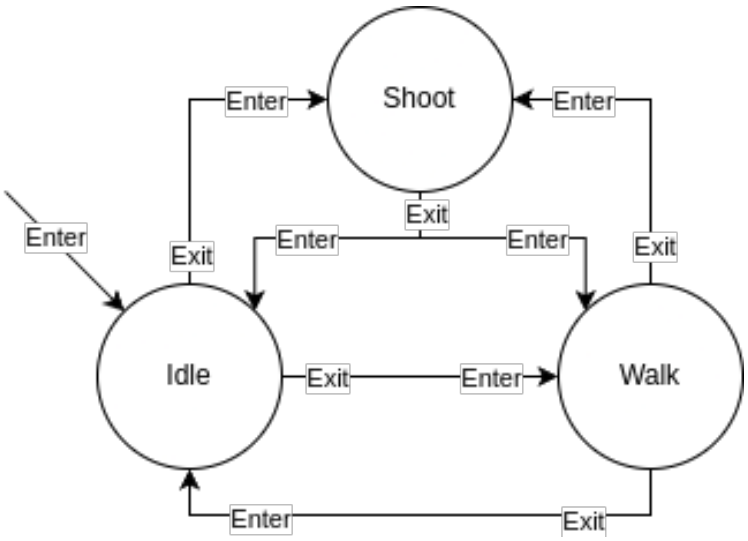


Figura 5.10 – Máquina de estados de um jogador.

A Figura 5.10 mostra o funcionamento de uma máquina de estados para uma entidade de jogador. Nessa máquina de estados existem três estados possíveis: *Idle*, *Walk* e *Shoot*. O início do funcionamento é indicado pela seta *Enter* no estado *Idle*. A transição para outros estados varia de acordo com as ações do jogador. Se o jogador move o *joystick* esquerdo do controle, por exemplo, o personagem se move e a máquina de estados transita do estado *Idle* para *Walk*.

5.2.15 Partículas

Partículas (*ParticleSystem*, Código 5.11) é um *wrapper* para o emissor de partículas. Ele armazena o número de partículas emitidas e o tempo para emitir cada partícula. O funcionamento é apresentado em detalhes na Seção 5.1.2.

Código 5.11 – Componente *ParticleSystem*.

```
1  class ParticleSystem : public Component
2  {
3      public:
4          ParticleSystem(Emitter *emitter, u32 particles_to_be_emitted = 50,
               f32 time_to_emit = 0.1f)
5              : particles_to_be_emitted(particles_to_be_emitted),
                   time_to_emit(time_to_emit)
6          {
7              this->emitter = std::unique_ptr<Emitter>(emitter);
8          }
9
10         std::unique_ptr<Emitter> emitter;
11         u32 particles_to_be_emitted;
12         f32 time_to_emit;
13 };
```

6 Jogo Piloto

Foi criado um protótipo de um jogo, denominado F90, com o objetivo de demonstrar as características da Bloss1. F90 é um jogo de tiro em terceira pessoa onde o jogador controla um *mecha* (F90) e o objetivo é derrotar o inimigo (Ophanim). Para isso, o jogador deve atirar projéteis no inimigo e procurar desviar dos ataques do Ophanim. Se o jogador tiver seus pontos de vida zerados, indicado pelo número azul na parte inferior da tela, ele perde. Se o inimigo tiver seus pontos de vida zerados, indicado pelo número bege no topo da tela, o jogador vence. A Figura 6.1 ilustra a *gameplay* do jogo.

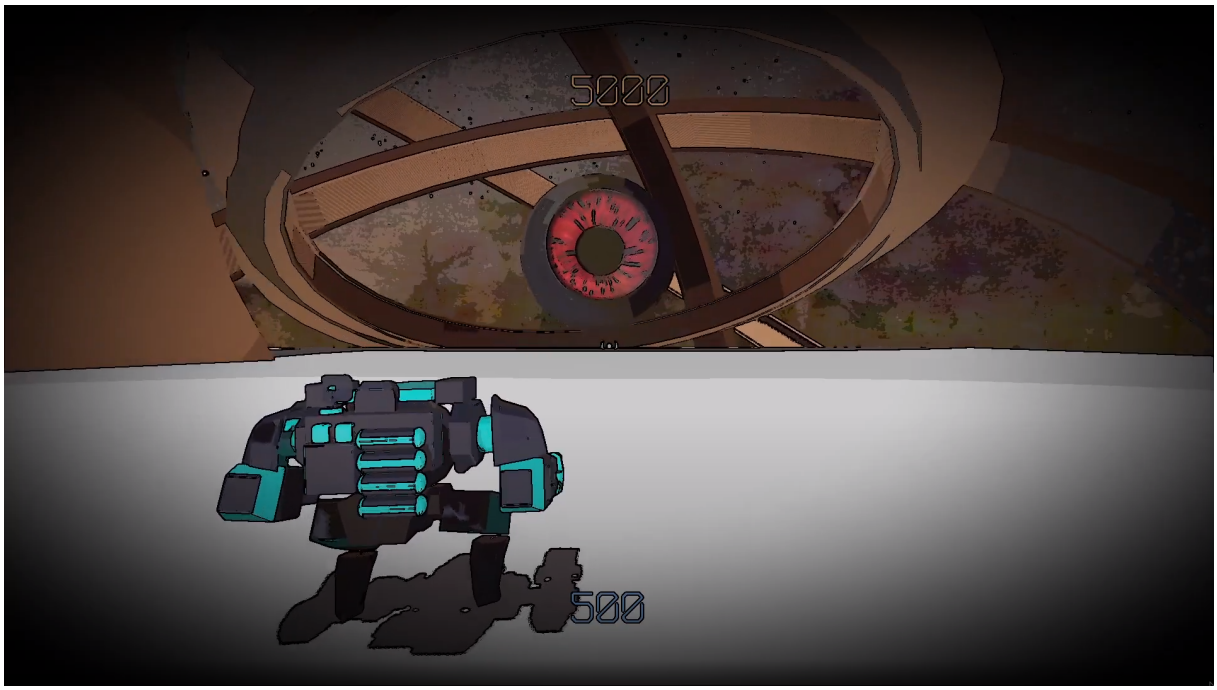


Figura 6.1 – F90 - jogo feito utilizando o motor de jogos Bloss1.

F90 utiliza todos os recursos ofertados pela engine, como iluminação global e diversos filtros de pós-processamento, física independente do tempo de renderização e áudio espacial. Componentes do ECS como *Projectile* e *BulletLandingIndicator* foram criados especificamente para uso no jogo e foram integrados sem dificuldade por meio dos sistemas *projectile_system* e *bullet_indicator_system*. As cenas criadas pelo editor (*menu.bloss* e *main_stage.bloss*) são carregadas durante a inicialização dos estágios e permitem fácil configuração de valores sem necessidade de recompilar o programa.

O Código 6.1 ilustra a inicialização do estágio principal com os sistemas da *engine* e do usuário acrescentados nas linhas 7-17. Os estágios e configurações criados para esse jogo são carregados nas linhas 20 e 28.

Código 6.1 – Inicialização do MainStage (main_stage.cpp).

```
void MainStage::start()
{
    // Create the ECS
    ecs = std::unique_ptr<ECS>(new ECS());

    // Add systems in order of execution
    ecs->add_system(player_controller_system);
    ecs->add_system(ophanim_controller_system);
    ecs->add_system(physics_system);
    ecs->add_system(bullet_indicator_system);
    ecs->add_system(projectile_system);
    ecs->add_system(state_machine_system);
    ecs->add_system(camera_system);
    ecs->add_system(animation_system);
    ecs->add_system(render_system_forward);
    ecs->add_system(sound_system);
    ecs->add_system(cleanup_system);

    // Load entities from file
    SceneParser::parse_scene(*ecs,
        "bloss1/assets/scenes/main_stage.bloss");

    auto &renderer = Game::get().get_renderer();
    if (renderer.get_height_map()) renderer.get_height_map().reset();
    if (!renderer.get_shadow_map()) renderer.create_shadow_map(*ecs);
    if (!renderer.get_post_processing())
        renderer.create_post_processing_passes(*ecs);

    // Load configurations from file
    SceneParser::parse_scene(*ecs,
        "bloss1/assets/scenes/bloss_config.bcfg");
}
```

O componente *BulletLandingIndicator* marca uma entidade como alvo e indica onde um projétil irá atingir essa entidade. Possui uma duração antes de ser extinguido e pode ser rotacionado em relação à posição do alvo. O código para esse componente pode ser analisado no Código 6.2.

Código 6.2 – Componente *BulletLandingIndicator* (components.hpp).

```
class BulletLandingIndicator : public Component
{
    public:
        BulletLandingIndicator(u32 target_id, u32 sender_id, const vec3
            &offset, const vec3 &rotation, f32 duration)
            : target_id(target_id), sender_id(sender_id), offset(offset),
                rotation(rotation), duration(duration)
        {
        }

```

```
9           u32 target_id, sender_id;
10          vec3 offset;
11          vec3 rotation;
12          f32 duration;
13  };
```

O componente *Projectile* representa um projétil. Esse projétil possui um tempo de vida e explode quando esse tempo se esgota. O dano, tempo de vida, raio e duração da explosão são configuráveis. O Código 6.3 ilustra esse componente.

Código 6.3 – Componente *Projectile* (components.hpp).

```
1   class Projectile : public Component
2   {
3       public:
4           Projectile(u32 sender_id,
5                      f32 damage = 5.0f,
6                      f32 explosion_radius = 10.0f,
7                      f32 explosion_duration = 1.0f,
8                      f32 time_to_live = 5.0f)
9               : sender_id(sender_id),
10                damage(damage),
11                explosion_radius(explosion_radius),
12                explosion_duration(explosion_duration),
13                time_to_live(time_to_live)
14          {
15          }
16
17          u32 sender_id;
18          f32 damage;
19          f32 explosion_radius;
20          f32 explosion_duration;
21          f32 time_to_live;
22  };
```

Para ilustrar a implementação dos sistemas do ECS, o Código 6.4 apresenta o sistema do *bullet_indicator*. O objetivo desse sistema é atualizar os componentes *BulletLandingIndicator*. Para isso, é feito o cálculo da posição do indicador, de acordo com a posição da entidade alvo (linhas 5-17). Um emissor de partículas também é utilizado para indicar visualmente a posição do alvo (linhas 19-23). Quando o tempo de vida do indicador expirar, a entidade é deletada e um projétil é lançado em direção ao alvo (linhas 25-40).

Código 6.4 – Sistema *bullet_indicator_system* (bullet_indicator_system.cpp).

```
1   void bullet_indicator_system(ECS& ecs, f32 dt)
2   {
3       for (const auto& [id, bullet_indicator] : ecs.bullet_indicators)
4       {
5           const auto& timer = ecs.timers[id];
```

```
6           const auto& target_transform =
                ecs.transforms[bullet_indicator->target_id];
7
8           vec3 target_pos = target_transform->position;
9           target_pos.y -= target_transform->scale.y;  // Moves a bit closer
                to the base of the target
10
11          // Recalculate offseted position from target
12          auto model_mat = mat4(1.0f);
13          model_mat = glm::translate(model_mat, target_pos);
14          model_mat = glm::rotate(model_mat, bullet_indicator->rotation.x,
                vec3(1.0f, 0.0f, 0.0f));
15          model_mat = glm::rotate(model_mat, bullet_indicator->rotation.y,
                vec3(0.0f, 1.0f, 0.0f));
16          model_mat = glm::rotate(model_mat, bullet_indicator->rotation.z,
                vec3(0.0f, 0.0f, 1.0f));
17          model_mat = glm::translate(model_mat, bullet_indicator->offset);
18
19          const vec3 indicator_position = vec3(model_mat[3]);
20
21          const auto& particle_sys = ecs.particle_systems[id];
22          const auto& emitter = particle_sys->emitter;
23          emitter->set_center(indicator_position);
24
25          timer->time += dt;
26          if (timer->time >= bullet_indicator->duration)
27          {
28              ecs.mark_for_deletion(id);
29
30              // Ophanim ID
31              if (bullet_indicator->sender_id != 1) return;
32
33              auto bullet_pos = indicator_position;
34              bullet_pos.y = 120.0f;
35
36              auto bullet_transform = Transform(bullet_pos, vec3(90.0f,
                    0.0f, 0.0f), vec3(20.0f));
37              auto bullet_object =
38                  PhysicsObject(vec3(0.0f), vec3(10000.0f), vec3(0.0f,
                        -1.0f, 0.0f) * 200000.0f, 15.0f);
39              bullet(ecs, bullet_transform, bullet_object, 1, 2.0f, 10.0f,
                    1.0f);
40          }
41      }
42  }
```

Por fim, um vídeo completo da gameplay pode ser acessado no youtube[1]. O código do jogo e da engine pode ser acessado no github[2].

[1] Disponível em: https://www.youtube.com/watch?v=Gb9bmYP_0Yg

[2] Disponível em: https://github.com/gustavo-tomas/Bloss1/

7 Conclusões

A Bloss1 demonstra realizar as tarefas necessárias para um motor de jogos moderno. Sua biblioteca de ferramentas permite a criação de mundos completos com visuais elegantes, simulação física consistente e áudios imersivos, como demonstrado pelo jogo exemplo. O ECS foi implementado de uma maneira orientada a dados, o que permite grande flexibilidade e performance para a criação de objetos. O editor, além de ser completamente configurável, proporciona conforto e velocidade para montar qualquer cena que o usuário desejar. O renderizador é performático e utiliza técnicas modernas para renderizar milhões de triângulos por *frame*. O motor de áudio permite reprodução de áudios espaciais e disponibiliza filtros para obter o resultado desejado. Ainda assim, é preciso destacar algumas limitações da Bloss1.

Processamento em Paralelo

A primeira limitação diz respeito ao uso de *multithreading*. *Engines* modernas procuram utilizar todo o potencial de processadores multinucleados por meio do uso de *threads*. Duas otimizações possíveis com o uso de *threads* são *streaming* de dados e o *Fork and Join approach* (Gregory, 2015, p.369).

A ideia por trás de *streaming* é carregar diversos *assets* (como texturas e modelos) em uma *thread* diferente da *main thread* (onde é executado o *game loop*). Dessa forma, *assets* podem ser carregados durante um jogo sem precisar pausar a execução.

O *Fork and Join approach*, por sua vez, consiste em paralelizar uma tarefa em tarefas menores (*Fork*) e depois juntar os resultados quando as tarefas terminarem (*Join*). Um exemplo de uso é durante a interpolação de poses esqueletais (Gregory, 2015, p.369).

Bloss1 é um motor *single-threaded* e, por isso, não é possível realizar *streaming* de dados. Uma fase deve ter todos os seus *assets* carregados antes de iniciar sua execução. Além disso, não é possível dividir uma tarefa utilizando o método *Fork and Join*, limitando a quantidade de tarefas que podem ser executadas sem causar uma grande perda de performance.

Batch Rendering

O renderizador, embora eficiente, não implementa uma técnica essencial que aumenta a performance do renderizador, principalmente em cenas com muitos triângulos, chamado *batch rendering* (Gregory, 2015, p.920). Essa técnica consiste em desenhar todos os triângulos de uma cena utilizando o menor número de *draw calls* possível. Isso porque *draw calls* são funções com custo de execução muito alto, por isso é essencial reduzir o número dessas chamadas. Além disso, placas gráficas se beneficiam muito de paralelismo, então enviar a maior quantidade de dados em uma só chamada para a GPU auxilia o uso de 100% dos recursos da placa.

Motor de Áudio

Como dito na Seção 4.7, o motor de áudio da Bloss1 é em grande parte um *wrapper* da biblioteca SoLoud, mas possui poucos filtros, além do áudio espacial. A combinação de diferentes filtros também não é suportada, limitando a construção de efeitos mais complexos. Os recursos de sincronização e *threading* também não foram implementados, e é um detalhe a ser considerado em um contexto com processamento em paralelo.

7.1 Trabalhos Futuros

Dentre os possíveis trabalhos futuros, sugere-se:

A modificação da *engine* para suportar uma arquitetura com processamento em paralelo: mudanças como o *streaming* de *assets* em uma *background thread*, a execução de sistemas independentes em *threads* separadas e o uso de uma API gráfica que seja mais adequada para execução em paralelo (por exemplo, Vulkan).

A modificação do renderizador para aumentar a eficiência: mudanças como o suporte para *batch rendering*, inclusive para texto e partículas.

A modificação do motor de áudio: melhorar e ampliar o suporte para filtros e permitir a combinação de diferentes filtros.

Por fim, se dá destaque a um outro trabalho que está sendo desenvolvido baseado na Bloss1, chamado Magnolia (Figura 7.1). O objetivo dessa *engine* é ser uma versão melhorada da Bloss1, com um editor mais potente, processamento em paralelo e renderização mais eficiente. A API de renderização utilizada é Vulkan, que possui recursos para renderização eficiente e é mais adequada para processamento em paralelo, embora seja consideravelmente mais complicada de usar. O projeto pode ser acessado no github[1].

[1] Disponível em: https://github.com/gustavo-tomas/Magnolia

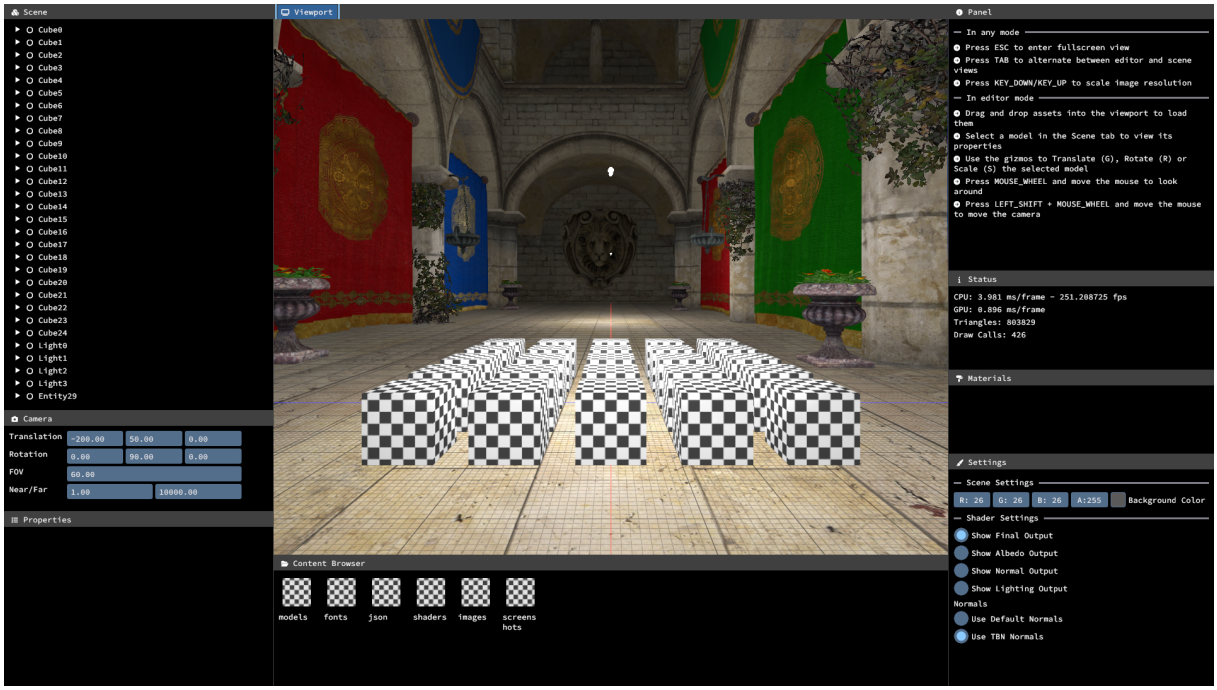


Figura 7.1 – Motor de Jogos Magnolia (v0.4.0).

Referências

AKENINE-MÖLLER, T. **Real-Time Rendering**. 4. ed. 6000 Broken Sound Parkway NW, Suite 300: CRC Press, 2018. Foi utilizada a versão eBook - PDF desse livro. ISBN 9781138627000. Citado nas pp. 17, 18, 30, 32, 33, 36, 50, 54 e 72.

ÁRBÓCZ, M. **Cascaded Shadow Mapping**. 2021. Online. Disponível em: https://learnopengl.com/Guest-Articles/2021/CSM. Citado nas pp. 28 e 88.

CORNUT, O. **Imgui Wiki**. 2024. Online. Disponível em: https://github.com/ocornut/imgui/wiki. Citado na p. 38.

DEVRIES, J. **Deferred Shading**. 2015. Online. Disponível em: https://learnopengl.com/Advanced-Lighting/Deferred-Shading. Citado na p. 72.

DEVRIES, J. **Diffuse irradiance**. 2017. Online. Disponível em: https://learnopengl.com/PBR/IBL/Diffuse-irradiance. Citado na p. 30.

DEVRIES, J. **Specular IBL**. 2017. Online. Disponível em: https://learnopengl.com/PBR/IBL/Specular-IBL. Citado na p. 30.

FABIAN, R. **Data-Oriented Design**. 2018. Online. Foi utilizada a versão pdf desse livro. Citado nas pp. 16, 17 e 46.

FIEDLER, T. G. **Fix Your Timestep!** 2004. Online. Disponível em: https://gafferongames.com/post/fix_your_timestep/. Citado na p. 39.

FIEDLER, T. G. **Integration Basics**. 2004. Online. Disponível em: https://gafferongames.com/post/integration_basics/. Citado na p. 39.

GREGORY, J. **Game Engine Architecture**. 2. ed. 6000 Broken Sound Parkway NW, Suite 300: CRC Press, 2015. Foi utilizada a versão eBook - PDF desse livro. ISBN 9781466560062. Citado nas pp. 13, 15, 20, 21, 24, 31, 36, 44, 52, 53, 54, 56, 58, 65 e 80.

KOMPPA, J. **SoLoud**. 2013. Online. Disponível em: https://solhsa.com/soloud/. Citado na p. 42.

KUSHWAH, A. S. **Skeletal Animation**. 2020. Online. Disponível em: https://learnopengl.com/Guest-Articles/2020/Skeletal-Animation. Citado na p. 56.

KYPRIANIDIS, J. E.; SEMMO, A.; KANG, H.; DöLLNER, J. Anisotropic Kuwahara Filtering with Polynomial Weighting Functions. *In*: COLLOMOSSE, J.; GRIMSTEAD, I. (Ed.). **Theory and Practice of Computer Graphics**. [*S.l.*]: The Eurographics Association, 2010. ISBN 978-3-905673-75-3. Foi utilizada a versão do autor desse trabalho. Citado na p. 37.

PAONE, J. **Tessellation Chapter I: Rendering Terrain using Height Maps**. 2021. Online. Disponível em: https://learnopengl.com/Guest-Articles/2021/Tessellation/Height-map. Citado na p. 72.

RADICH, Q.; SATRAN, M. **Retained Mode Versus Immediate Mode**. 2019. Online. Disponível em: https://learn.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode. Citado na p. 26.

SEGAL, M. **OpenGL 4.6 (Core Profile)**. 2. ed. [*S.l.*], 2022. Foi utilizada a versão PDF desse manual. Citado nas pp. 17, 26 e 72.

TANENBAUM, A. S. **Modern Operating Systems**. 4. ed. Upper Saddle River, New Jersey, 07458: Pearson, 2015. Foi utilizada a versão eBook - PDF desse livro. ISBN 9780133591620. Citado nas pp. 13 e 17.

WHITNEY, T. **map Class**. 2021. Online. Disponível em: https://learn.microsoft.com/en-us/cpp/standard-library/map-class?view=msvc-170. Citado na p. 44.

WHITNEY, T. **Smart pointers (Modern C++)**. 2021. Online. Disponível em: https://learn.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-170. Citado nas pp. 21 e 46.

Apêndices

Apêndice A – Códigos de Programação

A.1 Códigos da Classe *Game*

O Código A.1 mostra o *game loop* da Bloss1. Um estágio inicial *MenuStage* é registrado antes de iniciar o *loop* (linha 6). Em seguida, o estágio atual é atualizado até o jogo acabar (*stage* é nulo). As macros *BLS_PROFILE_BEGIN_SESSION* (linha 3) e *BLS_PROFILE_END_SESSION* (linha 51) são funções para medir performance e são removidas no modo *release*.

Código A.1 – *Game loop* (game.cpp).

```
1  void Game::run()
2  {
3      BLS_PROFILE_BEGIN_SESSION("MainLoop", "profile/runtime.json");
4
5      // Register initial stage
6      change_stage(new MenuStage());
7
8      // Time variation
9      last_time = window->get_time();
10     current_time = 0;
11     dt = 0;
12
13     set_target_fps(0);
14
15     window_open = true;
16     minimized = false;
17
18     // The game loop
19     while (stage && window_open)
20     {
21         // Don't render if the application is minimized
22         if (minimized)
23         {
24             window->update();
25             continue;
26         }
27
28         // Calculate dt
29         current_time = window->get_time();
30         dt = clamp(static_cast<f32>(current_time - last_time), 0.0f, 0.1f);
31         last_time = current_time;
32
33         // Update running stage
34         stage->update(dt);
35
36         if (!stage) break;
37
```

```
38              // Update editor
39 #if !defined(_RELEASE)
40         if (stage->ecs != nullptr) editor->update(*stage->ecs, dt);
41 #endif
42
43         // Update window
44         window->update();
45
46         // Sleep to match target spf
47         f64 elapsed = window->get_time() - last_time;
48         if (target_spf - elapsed > 0.0) window->sleep(target_spf -
               elapsed);
49     }
50
51     BLS_PROFILE_END_SESSION();
52 }
```

A variação de tempo (*delta time*) é calculada no início de cada *frame* (linha 29-31). Após um estágio ser atualizado, a janela é atualizada: *polling* de eventos e apresentação do *frame* renderizado (linha 44). Opcionalmente, é possível configurar um valor específico de *frames* por segundo utilizando o método *set_target_fps*.

A.2 Códigos do Renderizador

O renderizador deve ser implementado por uma subclasse. O Código A.2 mostra o a subclasse *OpenGLRenderer*. A inicialização consiste em ativar algumas funcionalidades da OpenGL, como teste de profundidade e *culling* (linhas 3-18). Em seguida, *shaders* que são utilizados extensamente por outras partes da *engine* são criados (linhas 24-40). Um *buffer* para geometria (DeVries, 2015) também é criado (linha 43), juntamente com as texturas para o sistema de *shading* em modo deferido (linhas 45-60). As linhas 66-68 criam uma textura específica para o jogo. As linhas 62-68 criam um *render buffer* (Segal, 2022, p.31) e terminam a inicialização do sistema de *shading* deferido. Por fim, a linha 71 cria um quadrilátero que serve como alvo de renderização e a linha 74 inicializa o sistema de pós processamento.

Os métodos nas linhas 77-145 encapsulam funções da OpenGL. O funcionamento dessas funções pode ser melhor explorado na documentação oficial (Segal, 2022).

Os métodos *create_** são encarregados de criar uma *skybox* (Seção 4.4.3) (linhas 147-157), um *shadow map* (Akenine-Möller, 2018, p.234) (linhas 159-169), um *height map* (Paone, 2021) para criação de terreno (linhas 171-176), e, por fim, os passes do sistema de pós-processamento (linhas 178-204). A criação desses sistemas não pode ser feita durante o processo de inicialização porque eles possuem dependências de outras partes do Código. A criação do *shadow map*, por exemplo, é feita a partir da posição da câmera, que só é criada na inicialização do ECS, depois do renderizador.

Código A.2 – Renderizador OpenGL (renderer.cpp).

```
void OpenGLRenderer::initialize()
{
    // Enable depth testing
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    // Enable stencil testing
    glEnable(GL_STENCIL_TEST);
    glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

    // Cull triangles which normal is not towards the camera
    glEnable(GL_CULL_FACE);

    // Remove cubemap seams
    glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);

    // Gamma correction (done by the shaders)
    glDisable(GL_FRAMEBUFFER_SRGB);

    auto &window = Game::get().get_window();
    auto width = window.get_width();
    auto height = window.get_height();

    // Create shaders
    // ----------------

    // Geometry buffer shader
    shaders["g_buffer"] =
        Shader::create("g_buffer", "bloss1/assets/shaders/g_buffer.vs",
            "bloss1/assets/shaders/g_buffer.fs");

    // PBR shader
    shaders["pbr"] = Shader::create("pbr",
       "bloss1/assets/shaders/pbr/pbr.vs",
       "bloss1/assets/shaders/pbr/pbr.fs");

    // PBR shader
    shaders["f_pbr"] = Shader::create(
        "f_pbr", "bloss1/assets/shaders/pbr/pbr_forward.vs",
            "bloss1/assets/shaders/pbr/pbr_forward.fs");

    // Debug shader
    shaders["color"] = Shader::create(
        "color", "bloss1/assets/shaders/test/base_color.vs",
            "bloss1/assets/shaders/test/base_color.fs");

    // Create g_buffer framebuffer
    g_buffer = std::unique_ptr<FrameBuffer>(FrameBuffer::create());

    // Create and attach framebuffer textures
    std::vector<str> texture_names = {"position", "normal", "albedo",
```

```
        "arm", "emissive", "depth"};
47      for (const auto &name : texture_names)
48      {
49          auto texture = Texture::create(width,
50                                         height,
51                                         ImageFormat::RGBA32F,
52                                         TextureParameter::Repeat,
53                                         TextureParameter::Repeat,
54                                         TextureParameter::Nearest,
55                                         TextureParameter::Nearest);
56
57          textures.push_back({name, texture});
58          g_buffer->attach_texture(texture.get());
59      }
60      g_buffer->draw();
61
62      // Create and attach depth buffer
63      render_buffer =
            std::unique_ptr<RenderBuffer>(RenderBuffer::create(width, height,
            AttachmentType::Depth));
64      render_buffer->bind();
65
66      // Check if framebuffer is complete
67      if (!g_buffer->check()) throw std::runtime_error("framebuffer is not
            complete");
68      g_buffer->unbind();
69
70      // Create a quad for rendering
71      quad = std::make_unique<Quad>(*this);
72
73      // Create post processing system
74      post_processing = std::make_unique<PostProcessingSystem>(width,
            height);
75  }
76
77  void OpenGLRenderer::set_viewport(u32 x, u32 y, u32 width, u32 height)
78  {
79      glViewport(x, y, width, height);
80  }
81
82  void OpenGLRenderer::set_debug_mode(bool active)
83  {
84      // Wireframe mode
85      if (active)
86      {
87          glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
88          glDisable(GL_CULL_FACE);
89      }
90
91      // Normal mode
92      else
93      {
94          glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

```
95          glEnable(GL_CULL_FACE);
96      }
97  }
98
99  void OpenGLRenderer::set_blending(bool active)
100 {
101     if (active)
102     {
103         glEnable(GL_BLEND);
104         glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
105     }
106
107     else
108         glDisable(GL_BLEND);
109 }
110
111 void OpenGLRenderer::set_face_culling(bool active)
112 {
113     if (active)
114         glEnable(GL_CULL_FACE);
115
116     else
117         glDisable(GL_CULL_FACE);
118 }
119
120 void OpenGLRenderer::set_tesselation_patches(u32 patches)
121 {
122     glPatchParameteri(GL_PATCH_VERTICES, patches);
123 }
124
125 void OpenGLRenderer::clear_color(const vec4 &color)
126 {
127     glClearColor(color.r, color.g, color.b, color.a);
128 }
129
130 void OpenGLRenderer::clear()
131 {
132     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
            GL_STENCIL_BUFFER_BIT);
133 }
134
135 void OpenGLRenderer::draw_indexed(RenderingMode mode, u32 count, const
        void *indices)
136 {
137     auto opengl_mode = convert_to_opengl_rendering_mode(mode);
138     glDrawElements(opengl_mode, count, GL_UNSIGNED_INT, indices);
139 }
140
141 void OpenGLRenderer::draw_arrays(RenderingMode mode, u32 count)
142 {
143     auto opengl_mode = convert_to_opengl_rendering_mode(mode);
144     glDrawArrays(opengl_mode, 0, count);
145 }
```

```
146
147 void OpenGLRenderer::create_skybox(const str &file,
148                                     const u32 skybox_resolution,
149                                     const u32 irradiance_resolution,
150                                     const u32 brdf_resolution,
151                                     const u32 prefilter_resolution,
152                                     const u32 max_mip_levels)
153 {
154     skybox.reset();
155     skybox = std::unique_ptr<Skybox>(Skybox::create(
156         file, skybox_resolution, irradiance_resolution, brdf_resolution,
            prefilter_resolution, max_mip_levels));
157 }
158
159 void OpenGLRenderer::create_shadow_map(ECS &ecs)
160 {
161     // Create shadow map
162     for (const auto &[id, dir_light] : ecs.dir_lights)
163     {
164         const auto &transform = ecs.transforms[id];
165         auto dir = transform->rotation;
166         dir.y *= -1.0f;
167         shadow_map = std::make_unique<ShadowMap>(*ecs.cameras[0],
            normalize(dir));
168     }
169 }
170
171 void OpenGLRenderer::create_height_map(
172     u32 width, u32 height, u32 min_tess_level, u32 max_tess_level, f32
        min_distance, f32 max_distance)
173 {
174     height_map =
175         std::make_unique<HeightMap>(width, height, min_tess_level,
            max_tess_level, min_distance, max_distance);
176 }
177
178 void OpenGLRenderer::create_post_processing_passes(ECS &ecs)
179 {
180     auto &window = Game::get().get_window();
181     auto &camera = ecs.cameras[0];
182
183     auto width = window.get_width();
184     auto height = window.get_height();
185
186     u32 pass_position = 1;
187     post_processing->add_pass(new FXAAPass(width, height),
        pass_position++);
188     post_processing->add_pass(new BloomPass(width, height, 5, 7.0f, 0.4f,
        0.325f), pass_position++);
189
190     post_processing->add_pass(new FogPass(width,
191                                           height,
192                                           vec3(0.0f),
```

```
193                                         vec2(camera->far / 3.0f,
                                                camera->far / 2.0f),
194                                         camera->position,
195                                         textures[0].second.get()),
196                                 pass_position++);
197
198     post_processing->add_pass(new SharpenPass(width, height, 0.05f),
            pass_position++);
199     post_processing->add_pass(new PosterizationPass(width, height, 8.0f),
            pass_position++);
200     post_processing->add_pass(new PixelizationPass(width, height, 4),
            pass_position++);
201     post_processing->add_pass(new OutlinePass(width, height, vec3(0.0f),
            0.8f), pass_position++);
202     post_processing->add_pass(new VignettePass(width, height, 0.85f,
            0.45f), pass_position++);
203     post_processing->add_pass(new KuwaharaPass(width, height, 5),
            pass_position++);
204 }
```

A.3 Códigos do ECS

Essa classe armazena todas as entidades e sistemas de um ECS. Sistemas são responsáveis por determinar a lógica e relacionamento entre componentes. Componentes são apenas dados, por isso é preciso que os sistemas determinem o significado desses dados.

O ID para uma nova entidade é obtido de uma *pool*. Quando a entidade é deletada, esse ID é inserido de volta na *pool*. As tabelas de componentes mapeiam o ID de cada entidade para seus respectivos componentes. Nessa organização, cada entidade possui no máximo um componente de cada tipo.

O Código A.3 apresenta a classe ECS. Durante a inicialização, os IDs disponíveis são registrados (linhas 8-11). Para criar uma entidade, é preciso chamar *get_id*, que retira o id da *pool* e o retorna. Sistemas podem ser adicionados utilizando *add_system*. A remoção de entidades/sistemas segue um padrão similar com *clear_systems* e *erase_entity*.

Código A.3 – Classe ECS (ecs.hpp).

```
1  // System: the logic bits
2  typedef void (*System)(ECS &ecs, f32 dt);
3
4  // ECS: container of the systems and entities
5  class ECS
6  {
7      public:
8          ECS(u32 max_entity_id = MAX_ENTITY_ID) :
               max_entity_id(max_entity_id)
9          {
```

```
10              for (u32 id = 0; id <= max_entity_id; id++)
                    available_ids.insert(id);
11          }
12
13          ~ECS()
14          {
15          }
16
17          // Return a new id (create a new entity)
18          u32 get_id()
19          {
20              if (available_ids.empty()) throw std::runtime_error("no
                    available ids left");
21
22              u32 id = *available_ids.begin();
23              available_ids.erase(id);
24
25              return id;
26          }
27
28          // Register a system
29          void add_system(System system)
30          {
31              systems.push_back(system);
32          }
33
34          // Clear all systems
35          void clear_systems()
36          {
37              systems.clear();
38          }
39
40          void mark_for_deletion(u32 id)
41          {
42              deletion_queue.push(id);
43          }
44
45          // Erase all the components of an entity
46          void erase_entity(u32 id)
47          {
48              if (id > max_entity_id) throw std::runtime_error("tried to
                    delete invalid id: " + to_str(id));
49
50              names.erase(id);
51              transforms.erase(id);
52              models.erase(id);
53              dir_lights.erase(id);
54              point_lights.erase(id);
55              physics_objects.erase(id);
56              colliders.erase(id);
57              transform_animations.erase(id);
58              timers.erase(id);
59              cameras.erase(id);
```

```
60              camera_controllers.erase(id);
61              texts.erase(id);
62              sounds.erase(id);
63              state_machines.erase(id);
64              projectiles.erase(id);
65              particle_systems.erase(id);
66              bullet_indicators.erase(id);
67              hitpoints.erase(id);
68
69              available_ids.insert(id);
70          }
71
72          // Registered systems
73          std::vector<System> systems;
74
75          // Table of components
76          std::map<u32, str> names;
77          std::map<u32, std::unique_ptr<Transform>> transforms;
78          std::map<u32, std::unique_ptr<ModelComponent>> models;
79          std::map<u32, std::unique_ptr<DirectionalLight>> dir_lights;
80          std::map<u32, std::unique_ptr<PointLight>> point_lights;
81          std::map<u32, std::unique_ptr<PhysicsObject>> physics_objects;
82          std::map<u32, std::unique_ptr<Collider>> colliders;
83          std::map<u32, std::unique_ptr<TransformAnimation>>
                transform_animations;
84          std::map<u32, std::unique_ptr<Timer>> timers;
85          std::map<u32, std::unique_ptr<Camera>> cameras;
86          std::map<u32, std::unique_ptr<CameraController>>
                camera_controllers;
87          std::map<u32, std::unique_ptr<Text>> texts;
88          std::map<u32, std::map<str, std::unique_ptr<Sound>>> sounds;
89          std::map<u32, std::unique_ptr<StateMachine>> state_machines;
90          std::map<u32, std::unique_ptr<Projectile>> projectiles;
91          std::map<u32, std::unique_ptr<ParticleSystem>> particle_systems;
92          std::map<u32, std::unique_ptr<BulletLandingIndicator>>
                bullet_indicators;
93          std::map<u32, f32> hitpoints;
94
95          std::queue<u32> deletion_queue;
96
97      private:
98          // Entities IDs
99          u32 max_entity_id;
100         std::set<u32> available_ids;
101 };
```

A.4 Códigos de Detecção de Colisão Entre Diferentes Colisores

O teste de colisão calcula o ponto na superfície de cada colisor mais perto um do outro. Feito isso, é possível calcular a interseção entre dois objetos (vetor de penetração) e a partir dessa interseção é possível mover os objetos em direções contrárias.

Para maioria dos casos, esse tipo de resolução é suficiente. Entretanto, esses algoritmos não levam em conta interseções em que um objeto está completamente dentro de outro. Além disso, mesmo utilizando *timestep* fixo, em escalas extremas de velocidade objetos podem sofrer o fenômeno de *tunneling* (Gregory, 2015, p.673).

Por decorrência dessas limitações, a criação de jogos de tiro com projéteis velozes e/ou movimentação rápida não é indicada.

Os Códigos A.4, A.5 e A.6 mostram os três tipos de colisão suportados pela Bloss1: caixa/esfera, esfera/esfera e caixa/caixa.

Código A.4 – Teste de colisão entre Caixa e Esfera.

```
1  // Box v. Sphere
2  if (collider_a->type == Collider::ColliderType::Box && collider_b->type ==
       Collider::ColliderType::Sphere)
3  {
4      const auto *col_a = static_cast<BoxCollider *>(collider_a);
5      const auto *col_b = static_cast<SphereCollider *>(collider_b);
6
7      // Point where the box 'begins'
8      vec3 min_aabb = trans_a.position - col_a->dimensions;
9
10     // Point where the box 'ends'
11     vec3 max_aabb = trans_a.position + col_a->dimensions;
12
13     // 1) find point 'pbox' on box the closest to the sphere centre.
14     vec3 closest_point_aabb;
15
16     // For each coordinate axis, if the point coordinate value is
17     // outside box, clamp it to the box, else keep it as is
18     for (u32 i = 0; i < 3; i++) closest_point_aabb[i] =
           clamp(trans_b.position[i], min_aabb[i], max_aabb[i]);
19
20     // 2) if 'pbox' is outside the sphere no collision.
21     f32 dist_aabb_to_sphere = distance(closest_point_aabb,
           trans_b.position);
22     if (dist_aabb_to_sphere < col_b->radius)
23     {
24         // 3) find point 'pshpere' on sphere surface the closest to point
               'pbox'.
25         vec3 closest_point_sphere =
26             trans_b.position + normalize(closest_point_aabb -
                   trans_b.position) * col_b->radius;
```

```
27
28          if (length(closest_point_sphere - closest_point_aabb) > 0.0f)
29          {
30              collision.point_a = closest_point_sphere;
31              collision.point_b = closest_point_aabb;
32              collision.has_collision = true;
33          }
34      }
35
36      return collision;
37  }
```

Código A.5 – Teste de colisão entre Esfera e Esfera.

```
1   // Sphere v. Sphere
2   else if (collider_a->type == Collider::ColliderType::Sphere &&
3            collider_b->type == Collider::ColliderType::Sphere)
4   {
5       const auto *col_a = static_cast<SphereCollider *>(collider_a);
6       const auto *col_b = static_cast<SphereCollider *>(collider_b);
7
8       // Insert tolerance to avoid equal points
9       const f32 TOL = 0.002;
10      f32 dist = distance(trans_a.position, trans_b.position);
11      if (dist < col_a->radius + col_b->radius - TOL)
12      {
13          // SphereA closest point to SphereB
14          vec3 vector_to_center_b = normalize(trans_b.position -
                trans_a.position);
15          vec3 vector_to_center_a = normalize(trans_a.position -
                trans_b.position);
16          vec3 closest_point_sphere_a = trans_b.position +
                vector_to_center_a * col_b->radius;
17          vec3 closest_point_sphere_b = trans_a.position +
                vector_to_center_b * col_a->radius;
18
19          if (length(closest_point_sphere_a - closest_point_sphere_b) > 0.0f)
20          {
21              collision.point_a = closest_point_sphere_a;
22              collision.point_b = closest_point_sphere_b;
23              collision.has_collision = true;
24          }
25      }
26
27      return collision;
28  }
```

Código A.6 – Teste de colisão entre Caixa e Caixa.

```
1   // Box v. Box
2   else if (collider_a->type == Collider::ColliderType::Box &&
        collider_b->type == Collider::ColliderType::Box)
```

```
3  {
4      const auto *col_a = static_cast<BoxCollider *>(collider_a);
5      const auto *col_b = static_cast<BoxCollider *>(collider_b);
6
7      // Point where the box 'begins'
8      vec3 min_aabb_a = trans_a.position - col_a->dimensions;
9      vec3 min_aabb_b = trans_b.position - col_b->dimensions;
10
11     // Point where the box 'ends'
12     vec3 max_aabb_a = trans_a.position + col_a->dimensions;
13     vec3 max_aabb_b = trans_b.position + col_b->dimensions;
14
15     bool intersecting =
16         (min_aabb_a.x <= max_aabb_b.x && max_aabb_a.x >= min_aabb_b.x &&
               min_aabb_a.y <= max_aabb_b.y &&
17          max_aabb_a.y >= min_aabb_b.y && min_aabb_a.z <= max_aabb_b.z &&
                max_aabb_a.z >= min_aabb_b.z);
18
19     if (intersecting)
20     {
21         vec3 overlap;
22         overlap.x = max(0.0f, min(max_aabb_a.x, max_aabb_b.x) -
               max(min_aabb_a.x, min_aabb_b.x));
23         overlap.y = max(0.0f, min(max_aabb_a.y, max_aabb_b.y) -
               max(min_aabb_a.y, min_aabb_b.y));
24         overlap.z = max(0.0f, min(max_aabb_a.z, max_aabb_b.z) -
               max(min_aabb_a.z, min_aabb_b.z));
25
26         vec3 penetration_depth = vec3(0.0f);
27         if (overlap.x < overlap.y && overlap.x < overlap.z)
28         {
29             if (min_aabb_a.x < min_aabb_b.x)
30             {
31                 max_aabb_a.x -= overlap.x;
32                 min_aabb_b.x += overlap.x;
33                 penetration_depth.x = -overlap.x;
34             }
35             else
36             {
37                 min_aabb_a.x += overlap.x;
38                 max_aabb_b.x -= overlap.x;
39                 penetration_depth.x = overlap.x;
40             }
41         }
42
43         else if (overlap.y < overlap.z)
44         {
45             if (min_aabb_a.y < min_aabb_b.y)
46             {
47                 max_aabb_a.y -= overlap.y;
48                 min_aabb_b.y += overlap.y;
49                 penetration_depth.y = -overlap.y;
50             }
```

```
51              else
52              {
53                  min_aabb_a.y += overlap.y;
54                  max_aabb_b.y -= overlap.y;
55                  penetration_depth.y = overlap.y;
56              }
57          }
58
59          else
60          {
61              if (min_aabb_a.z < min_aabb_b.z)
62              {
63                  max_aabb_a.z -= overlap.z;
64                  min_aabb_b.z += overlap.z;
65                  penetration_depth.z = -overlap.z;
66              }
67              else
68              {
69                  min_aabb_a.z += overlap.z;
70                  max_aabb_b.z -= overlap.z;
71                  penetration_depth.z = +overlap.z;
72              }
73          }
74
75          if (length(penetration_depth) > 0.0f)
76          {
77              collision.point_a = penetration_depth;
78              collision.point_b = vec3(0.0f);
79              collision.has_collision = true;
80          }
81      }
82
83      return collision;
84  }
```

A.5 Códigos da Classe *Shader*

A classe *Shader* é em grande parte um *wrapper* para funções da OpenGL (Código A.7). Essa classe se encarrega de ler e compilar um arquivo *glsl* do disco (linhas 18-105). Em um programa típico, apenas os *shaders* de vértice e fragmento são necessários para uma compilação bem sucedida. As etapas restantes são opcionais.

Para usar um shader, antes é preciso chamar o método *bind* (linha 148). Após isso, é possível enviar dados de vários tipos em formato de *uniforms* utilizando os métodos *set_** (linhas 158-193).

Código A.7 – Classe *OpenGLShader* (shader.cpp).

```
1  #include "renderer/opengl/shader.hpp"
```

```
2
3  #include <GL/glew.h>  // Include glew before glfw
4
5  #include "GLFW/glfw3.h"
6  #include "core/logger.hpp"
7
8  #define MAX_SHADER_ID 100000007
9
10 namespace bls
11 {
12     OpenGLShader::OpenGLShader(const str &vertex_path,
13                                const str &fragment_path,
14                                const str &geometry_path,
15                                const str &tess_ctrl_path,
16                                const str &tess_eval_path)
17     {
18         // Create the shaders
19         GLuint vertex_shader_id = glCreateShader(GL_VERTEX_SHADER);
20         GLuint fragment_shader_id = glCreateShader(GL_FRAGMENT_SHADER);
21         GLuint geometry_shader_id = MAX_SHADER_ID;
22         GLuint tess_ctrl_shader_id = MAX_SHADER_ID;
23         GLuint tess_eval_shader_id = MAX_SHADER_ID;
24
25         if (geometry_path != "") geometry_shader_id =
               glCreateShader(GL_GEOMETRY_SHADER);
26         if (tess_ctrl_path != "") tess_ctrl_shader_id =
               glCreateShader(GL_TESS_CONTROL_SHADER);
27         if (tess_eval_path != "") tess_eval_shader_id =
               glCreateShader(GL_TESS_EVALUATION_SHADER);
28
29         str vertex_shader_code, fragment_shader_code, geometry_shader_code;
30         str tess_ctrl_shader_code, tess_eval_shader_code;
31
32         // Read the Vertex Shader code from the file
33         try
34         {
35             vertex_shader_code = get_code_from_file(vertex_path);
36             fragment_shader_code = get_code_from_file(fragment_path);
37
38             if (geometry_path != "") geometry_shader_code =
                   get_code_from_file(geometry_path);
39             if (tess_ctrl_path != "") tess_ctrl_shader_code =
                   get_code_from_file(tess_ctrl_path);
40             if (tess_eval_path != "") tess_eval_shader_code =
                   get_code_from_file(tess_eval_path);
41         }
42
43         catch (...)
44         {
45             LOG_ERROR("error when reading files '%s' and '%s'",
                   vertex_path.c_str(), fragment_path.c_str());
46         }
47
```

```
48          // Compile shaders
49          compile_shader(vertex_path, vertex_shader_code, vertex_shader_id);
50          compile_shader(fragment_path, fragment_shader_code,
                fragment_shader_id);
51
52          if (geometry_path != "" && geometry_shader_id != MAX_SHADER_ID)
53              compile_shader(geometry_path, geometry_shader_code,
                    geometry_shader_id);
54
55          if (tess_ctrl_path != "" && tess_ctrl_shader_id != MAX_SHADER_ID)
56              compile_shader(tess_ctrl_path, tess_ctrl_shader_code,
                    tess_ctrl_shader_id);
57
58          if (tess_eval_path != "" && tess_eval_shader_id != MAX_SHADER_ID)
59              compile_shader(tess_eval_path, tess_eval_shader_code,
                    tess_eval_shader_id);
60
61          // Link the program
62          LOG_INFO("linking program");
63          id = glCreateProgram();
64
65          glAttachShader(id, vertex_shader_id);
66          glAttachShader(id, fragment_shader_id);
67
68          if (geometry_path != "") glAttachShader(id, geometry_shader_id);
69          if (tess_ctrl_path != "") glAttachShader(id, tess_ctrl_shader_id);
70          if (tess_eval_path != "") glAttachShader(id, tess_eval_shader_id);
71
72          glLinkProgram(id);
73
74          GLint result = GL_FALSE;
75          int log_length = 0;
76
77          // Check the program
78          glGetProgramiv(id, GL_LINK_STATUS, &result);
79          glGetProgramiv(id, GL_INFO_LOG_LENGTH, &log_length);
80          if (log_length > 0)
81          {
82              std::vector<char> error_message(log_length + 1);
83              glGetProgramInfoLog(id, log_length, NULL, &error_message[0]);
84
85              LOG_ERROR("linking program");
86              LOG_ERROR("error when linking shader: '%s'",
                    vertex_path.c_str());
87              LOG_ERROR("error when linking program: '%s'",
                    error_message.data());
88
89              throw std::runtime_error("failed to link program");
90          }
91
92          glDetachShader(id, vertex_shader_id);
93          glDetachShader(id, fragment_shader_id);
94
```

```
95          if (geometry_path != "") glDetachShader(id, geometry_shader_id);
96          if (tess_ctrl_path != "") glDetachShader(id, tess_ctrl_shader_id);
97          if (tess_eval_path != "") glDetachShader(id, tess_eval_shader_id);
98
99          glDeleteShader(vertex_shader_id);
100         glDeleteShader(fragment_shader_id);
101
102         if (geometry_path != "") glDeleteShader(geometry_shader_id);
103         if (tess_ctrl_path != "") glDeleteShader(tess_ctrl_shader_id);
104         if (tess_eval_path != "") glDeleteShader(tess_eval_shader_id);
105
106         LOG_SUCCESS("shaders compiled & linked successfully");
107     }
108
109     void OpenGLShader::compile_shader(const str &path, const str &code,
        u32 ID)
110     {
111         GLint result = GL_FALSE;
112         int log_length = 0;
113
114         // Compile shader
115         LOG_INFO("compiling shader '%s'", path.c_str());
116
117         char const *sourcePtr = code.c_str();
118         glShaderSource(ID, 1, &sourcePtr, NULL);
119         glCompileShader(ID);
120
121         // Check shader
122         glGetShaderiv(ID, GL_COMPILE_STATUS, &result);
123         glGetShaderiv(ID, GL_INFO_LOG_LENGTH, &log_length);
124
125         if (log_length > 0)
126         {
127             std::vector<char> error_message(log_length + 1);
128             glGetShaderInfoLog(ID, log_length, NULL, &error_message[0]);
129
130             LOG_ERROR("error when compiling '%s': %s", path.c_str(),
                error_message.data());
131         }
132     }
133
134     str OpenGLShader::get_code_from_file(const str &path)
135     {
136         str code = "";
137         std::stringstream sstr;
138         std::ifstream stream;
139
140         stream.open(path);
141         sstr << stream.rdbuf();
142         code = sstr.str();
143         stream.close();
144
145         return code;
```

```
146     }
147
148     void OpenGLShader::bind()
149     {
150         glUseProgram(id);
151     }
152
153     void OpenGLShader::unbind()
154     {
155         glUseProgram(0);
156     }
157
158     void OpenGLShader::set_uniform1(const str &name, bool value)
159     {
160         glUniform1i(glGetUniformLocation(id, name.c_str()), (u32)value);
161     }
162
163     void OpenGLShader::set_uniform1(const str &name, u32 value)
164     {
165         glUniform1i(glGetUniformLocation(id, name.c_str()), value);
166     }
167
168     void OpenGLShader::set_uniform1(const str &name, f32 value)
169     {
170         glUniform1f(glGetUniformLocation(id, name.c_str()), value);
171     }
172
173     void OpenGLShader::set_uniform2(const str &name, const vec2 &vector)
174     {
175         glUniform2f(glGetUniformLocation(id, name.c_str()), vector.x,
                vector.y);
176     }
177
178     void OpenGLShader::set_uniform3(const str &name, const vec3 &vector)
179     {
180         glUniform3f(glGetUniformLocation(id, name.c_str()), vector.x,
                vector.y, vector.z);
181     }
182
183     void OpenGLShader::set_uniform3(const str &name, const mat3 &matrix)
184     {
185         glUniformMatrix3fv(glGetUniformLocation(id, name.c_str()), 1,
                GL_FALSE, value_ptr(matrix));
186     }
187
188     void OpenGLShader::set_uniform4(const str &name, const vec4 &vector)
189     {
190         glUniform4f(glGetUniformLocation(id, name.c_str()), vector.x,
                vector.y, vector.z, vector.w);
191     }
192
193     void OpenGLShader::set_uniform4(const str &name, const mat4 &matrix)
194     {
```

```
195         glUniformMatrix4fv(glGetUniformLocation(id, name.c_str()), 1,
                GL_FALSE, value_ptr(matrix));
196     }
197
198     vec3 OpenGLShader::get_uniform3(const str &name)
199     {
200         vec3 vector;
201         glGetUniformfv(id, glGetUniformLocation(id, name.c_str()),
                value_ptr(vector));
202         return vector;
203     }
204
205     mat4 OpenGLShader::get_uniform4(const str &name)
206     {
207         mat4 matrix;
208         glGetUniformfv(id, glGetUniformLocation(id, name.c_str()),
                value_ptr(matrix));
209         return matrix;
210     }
211 };  // namespace bls
```

A.6 Códigos da Classe *ShadowMap*

O processo do CSM consiste em renderizar um *shadow map* para cada nível do mapa (na Bloss1, existem 5 níveis de sombras). O Código A.8 mostra a classe *ShadowMap*.

As linhas 5-9 determinam a quantidade de níveis de sombras e o valor de cada nível. As linhas 10-65 configuram o *depth_map* necessário para calcular a distância de objetos em relação à câmera, ou seja, a profundidade de cada objeto.

O método *bind* (linhas 75-92) é encarregado de enviar dados das *light_space_matrices* (Árbócz, 2021) para os *shaders* inicializados no construtor. Para calcular essas matrizes, é utilizado um método auxiliar get_light_space_matrices (linha 228-243).

Após a cena ser renderizada utilizando o *shadow_map_depth* (linhas 145-148), o *shadow map* está devidamente preenchido e pronto para ser usado. Utilizando o método *bind_maps* (linhas 101-112) é possível enviar os *maps* preenchidos para os *shaders* e renderizá-los em uma cena.

Código A.8 – Classe *ShadowMap* (shadow_map.cpp).

```
1 ShadowMap::ShadowMap(const Camera &camera, const vec3 &light_dir)
2 {
3     this->light_dir = light_dir;
4
5     u32 zoom_factor = 10.0f;
6     shadow_cascade_levels = {camera.far / (5.0f * zoom_factor),
7                              camera.far / (2.5f * zoom_factor),
8                              camera.far / (1.0f * zoom_factor),
```

```
9                               camera.far / (0.2f * zoom_factor)};
10      depth_map_resolution = 4096;
11
12      shadow_map_depth = Shader::create(
13          "shadow_map_depth",
14          "bloss1/assets/shaders/shadow_map_depth.vs",
15          "bloss1/assets/shaders/shadow_map_depth.fs",
16          "bloss1/assets/shaders/shadow_map_depth.gs");
17
18      debug_depth = Shader::create(
19          "debug_depth", "bloss1/assets/shaders/test/debug_depth.vs",
                "bloss1/assets/shaders/test/debug_depth.fs");
20
21      debug_depth->bind();
22      debug_depth->set_uniform1("depthMap", 0U);
23
24      debug_quad = std::make_unique<Quad>(Game::get().get_renderer());
25
26      // Create light depth buffers
27      glGenFramebuffers(1, &light_FBO);
28
29      glGenTextures(1, &light_depth_maps);
30      glBindTexture(GL_TEXTURE_2D_ARRAY, light_depth_maps);
31      glTexImage3D(GL_TEXTURE_2D_ARRAY,
32                   0,
33                   GL_DEPTH_COMPONENT32F,
34                   depth_map_resolution,
35                   depth_map_resolution,
36                   static_cast<i32>(shadow_cascade_levels.size()) + 1,
37                   0,
38                   GL_DEPTH_COMPONENT,
39                   GL_FLOAT,
40                   nullptr);
41
42      glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER,
            GL_NEAREST);
43      glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER,
            GL_NEAREST);
44      glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_S,
            GL_CLAMP_TO_BORDER);
45      glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_T,
            GL_CLAMP_TO_BORDER);
46
47      constexpr f32 bordercolor[] = {1.0f, 1.0f, 1.0f, 1.0f};
48      glTexParameterfv(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_BORDER_COLOR,
            bordercolor);
49
50      glBindFramebuffer(GL_FRAMEBUFFER, light_FBO);
51      glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
            light_depth_maps, 0);
52      glDrawBuffer(GL_NONE);
53      glReadBuffer(GL_NONE);
54
```

```
55      int status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
56      if (status != GL_FRAMEBUFFER_COMPLETE) throw
            std::runtime_error("lightFBO framebuffer is not complete");
57
58      glBindFramebuffer(GL_FRAMEBUFFER, 0);
59
60      // Configure UBO
61      glGenBuffers(1, &matrices_UBO);
62      glBindBuffer(GL_UNIFORM_BUFFER, matrices_UBO);
63      glBufferData(GL_UNIFORM_BUFFER, sizeof(glm::mat4x4) * 16, nullptr,
            GL_STATIC_DRAW);
64      glBindBufferBase(GL_UNIFORM_BUFFER, 0, matrices_UBO);
65      glBindBuffer(GL_UNIFORM_BUFFER, 0);
66  }
67
68  ShadowMap::~ShadowMap()
69  {
70      glDeleteFramebuffers(1, &light_FBO);
71      glDeleteBuffers(1, &matrices_UBO);
72      glDeleteTextures(1, &light_depth_maps);
73  }
74
75  void ShadowMap::bind(const Camera &camera)
76  {
77      // UBO setup
78      const auto lightMatrices = get_light_space_matrices(camera);
79
80      glBindBuffer(GL_UNIFORM_BUFFER, matrices_UBO);
81      for (size_t i = 0; i < lightMatrices.size(); i++)
82          glBufferSubData(GL_UNIFORM_BUFFER, i * sizeof(glm::mat4x4),
                sizeof(glm::mat4x4), &lightMatrices[i]);
83
84      glBindBuffer(GL_UNIFORM_BUFFER, 0);
85
86      // Render depth of scene to texture (from light's perspective)
87      shadow_map_depth->bind();
88      glBindFramebuffer(GL_FRAMEBUFFER, light_FBO);
89      glViewport(0, 0, depth_map_resolution, depth_map_resolution);
90      glClear(GL_DEPTH_BUFFER_BIT);
91      glCullFace(GL_FRONT);  // peter panning (dont forget to reset after
            render)
92  }
93
94  void ShadowMap::unbind()
95  {
96      glCullFace(GL_BACK);
97      glBindFramebuffer(GL_FRAMEBUFFER, 0);
98      shadow_map_depth->unbind();
99  }
100
101 void ShadowMap::bind_maps(Shader &shader, u32 slot)
102 {
103     // Set shadow map uniforms
```

```
104     shader.set_uniform1("textures.shadowMap", slot);
105     glActiveTexture(GL_TEXTURE0 + slot);
106     glBindTexture(GL_TEXTURE_2D_ARRAY, light_depth_maps);
107 
108     shader.set_uniform1("cascadeCount",
            static_cast<u32>(shadow_cascade_levels.size()));
109 
110     for (u64 i = 0; i < shadow_cascade_levels.size(); i++)
111         shader.set_uniform1("cascadePlaneDistances[" + to_str(i) + "]",
                shadow_cascade_levels[i]);
112 }
113 
114 u32 layer = 0;
115 bool released = true;
116 void ShadowMap::render_debug()
117 {
118     if (Input::is_key_pressed(KEY_SPACE) && released)
119     {
120         layer = (layer + 1) % 5;
121         released = false;
122     }
123 
124     if (!Input::is_key_pressed(KEY_SPACE)) released = true;
125 
126     // Render Depth map for visual debugging
127     debug_depth->bind();
128     debug_depth->set_uniform1("layer", layer);
129 
130     glActiveTexture(GL_TEXTURE0);
131     glBindTexture(GL_TEXTURE_2D_ARRAY, light_depth_maps);
132     debug_quad->render();
133 }
134 
135 void ShadowMap::set_light_dir(const vec3 &light_dir)
136 {
137     this->light_dir = light_dir;
138 }
139 
140 vec3 ShadowMap::get_light_dir()
141 {
142     return light_dir;
143 }
144 
145 Shader &ShadowMap::get_shadow_depth_shader() const
146 {
147     return *shadow_map_depth;
148 }
149 
150 std::vector<vec4> ShadowMap::get_frustum_corners_world_space(const mat4
        &projview)
151 {
152     const mat4 inv = inverse(projview);
153 
```

```
154     std::vector<vec4> frustum_corners;
155     for (u32 x = 0; x < 2; ++x)
156     {
157         for (u32 y = 0; y < 2; ++y)
158         {
159             for (u32 z = 0; z < 2; ++z)
160             {
161                 const vec4 pt = inv * vec4(2.0f * x - 1.0f, 2.0f * y -
                        1.0f, 2.0f * z - 1.0f, 1.0f);
162                 frustum_corners.push_back(pt / pt.w);
163             }
164         }
165     }
166 
167     return frustum_corners;
168 }
169 
170 std::vector<vec4> ShadowMap::get_frustum_corners_world_space(const mat4
        &proj, const mat4 &view)
171 {
172     return get_frustum_corners_world_space(proj * view);
173 }
174 
175 mat4 ShadowMap::get_light_space_matrix(const Camera &camera, f32 near, f32
        far)
176 {
177     auto &window = Game::get().get_window();
178     f32 width = window.get_width();
179     f32 height = window.get_height();
180 
181     const mat4 proj = perspective(radians(camera.zoom), width / height,
            near, far);
182 
183     const auto corners = get_frustum_corners_world_space(proj,
            camera.view_matrix);
184 
185     vec3 center = vec3(0.0f);
186     for (const vec4 &v : corners) center += vec3(v);
187 
188     center /= corners.size();
189 
190     const mat4 lightView = lookAt(center + light_dir, center, vec3(0.0f,
            1.0f, 0.0f));
191 
192     f32 minX = std::numeric_limits<f32>::max();
193     f32 maxX = std::numeric_limits<f32>::lowest();
194     f32 minY = std::numeric_limits<f32>::max();
195     f32 maxY = std::numeric_limits<f32>::lowest();
196     f32 minZ = std::numeric_limits<f32>::max();
197     f32 maxZ = std::numeric_limits<f32>::lowest();
198 
199     for (const vec4 &v : corners)
200     {
```

```
201         const vec4 trf = lightView * v;
202         minX = min(minX, trf.x);
203         maxX = max(maxX, trf.x);
204         minY = min(minY, trf.y);
205         maxY = max(maxY, trf.y);
206         minZ = min(minZ, trf.z);
207         maxZ = max(maxZ, trf.z);
208     }
209
210     // Tune this parameter according to the scene
211     constexpr f32 zMult = 10.0f;
212     if (minZ < 0)
213         minZ *= zMult;
214
215     else
216         minZ /= zMult;
217
218     if (maxZ < 0)
219         maxZ /= zMult;
220
221     else
222         maxZ *= zMult;
223
224     const mat4 lightProjection = glm::ortho(minX, maxX, minY, maxY, minZ,
        maxZ);
225     return lightProjection * lightView;
226 }
227
228 std::vector<mat4> ShadowMap::get_light_space_matrices(const Camera &camera)
229 {
230     std::vector<mat4> ret;
231     for (size_t i = 0; i < shadow_cascade_levels.size() + 1; i++)
232     {
233         if (i == 0)
234             ret.push_back(get_light_space_matrix(camera, camera.near,
                shadow_cascade_levels[i]));
235
236         else if (i < shadow_cascade_levels.size())
237             ret.push_back(get_light_space_matrix(camera,
                shadow_cascade_levels[i - 1], shadow_cascade_levels[i]));
238
239         else
240             ret.push_back(get_light_space_matrix(camera,
                shadow_cascade_levels[i - 1], camera.far));
241     }
242     return ret;
243 }
```

A.7 Códigos do Sistema de Renderização

Embora os sistemas de renderização *forward* e *deferred* funcionem de formas diferentes, existem funções comuns aos dois sistemas. O Código A.9 mostra a função *render_scene* que se encarrega de calcular as transformações e renderizar modelos.

Na linha 7, as matrizes de ossos são reiniciadas nos *shaders*. Caso um modelo possua animador (e por consequência um esqueleto), essas matrizes são sobrescritas pelas matrizes de ossos do modelo. Feito isso, são calculadas as matrizes de transformação do modelo. Seus valores são, então, enviados para o *shader*. O último passo antes de realizar a *draw call* é enviar as texturas. Cada textura que o modelo possui é enviada para o *shader* de fragmentos. Por fim, os vértices de cada modelo são desenhados.

Código A.9 – Função *render_scene* (render_system.cpp).

```
1  void render_scene(ECS &ecs, Shader &shader, Renderer &renderer)
2  {
3      // Render all entities
4      for (const auto &[id, model] : ecs.models)
5      {
6          // Reset bone matrices
7          for (u32 i = 0; i < MAX_BONE_MATRICES; i++)
8              shader.set_uniform4("finalBonesMatrices[" + to_str(i) + "]",
                   mat4(1.0f));
9
10         // Update animators
11         auto animator = model->model->animator.get();
12         if (animator)
13         {
14             // Update bone matrices
15             auto bone_matrices = animator->get_final_bone_matrices();
16             for (u32 i = 0; i < bone_matrices.size(); i++)
17                 shader.set_uniform4("finalBonesMatrices[" + to_str(i) +
                       "]", bone_matrices[i]);
18         }
19
20         // Remember: scale -> rotate -> translate
21         auto transform = ecs.transforms[id].get();
22         auto model_matrix = mat4(1.0f);
23
24         // Translate
25         model_matrix = translate(model_matrix, transform->position);
26
27         // Rotate
28         model_matrix = rotate(model_matrix,
               radians(transform->rotation.x), vec3(1.0f, 0.0f, 0.0f));
29         model_matrix = rotate(model_matrix,
               radians(transform->rotation.y), vec3(0.0f, 1.0f, 0.0f));
30         model_matrix = rotate(model_matrix,
               radians(transform->rotation.z), vec3(0.0f, 0.0f, 1.0f));
31
```

```
32          // Scale
33          model_matrix = scale(model_matrix, transform->scale);
34
35          // Bind and update data to shader
36          shader.set_uniform4("model", model_matrix);
37
38          // Render the model
39          for (const auto &mesh : model->model->meshes)
40          {
41              // Bind textures
42              for (u32 i = 0; i < mesh->textures.size(); i++)
43              {
44                  auto texture = mesh->textures[i];
45
46                  str type_name;
47                  auto type = texture->get_type();
48
49                  switch (type)
50                  {
51                      case TextureType::Diffuse:
52                          type_name = "diffuse";
53                          break;
54                      case TextureType::Specular:
55                          type_name = "specular";
56                          break;
57                      case TextureType::Normal:
58                          type_name = "normal";
59                          break;
60                      case TextureType::Metalness:
61                          type_name = "metalness";
62                          break;
63                      case TextureType::Roughness:
64                          type_name = "roughness";
65                          break;
66                      case TextureType::AmbientOcclusion:
67                          type_name = "ao";
68                          break;
69                      case TextureType::Emissive:
70                          type_name = "emissive";
71                          break;
72
73                      default:
74                          LOG_ERROR("invalid texture type");
75                          break;
76                  }
77
78                  shader.set_uniform1("material." + type_name, i);
79                  texture->bind(i);  // Offset the active samplers in the
                        frag shader
80              }
81
82              mesh->vao->bind();
83              renderer.draw_indexed(RenderingMode::Triangles,
```

```
                mesh->indices.size());
84              mesh->vao->unbind();
85
86              // Update stats
87              AppStats::vertices += mesh->vertices.size();
88          }
89      }
90  }
```

A.8 Códigos da Janela

A classe *Window* (Código A.10) é uma classe abstrata que representa uma janela. A implementação em si é feita pela subclasse *GlfwWindow* (Código A.11) - a Bloss1 utiliza a biblioteca glfw[1] para criar sua janela e glew[2] para inicializar a OpenGL. A ordem de inicialização nesse caso é importante, por isso *glew* é inicializada aqui.

Código A.10 – Classe *Window* (window.hpp).

```
1   class Window
2   {
3       public:
4           using EventCallback = std::function<void(Event &)>;
5
6           virtual ~Window()
7           {
8           }
9
10          virtual void update() = 0;
11          virtual void sleep(f64 seconds) = 0;
12
13          virtual u32 get_width() const = 0;
14          virtual u32 get_height() const = 0;
15
16          virtual void *get_native_window() const = 0;
17          virtual f64 get_time() const = 0;
18
19          virtual void set_event_callback(const EventCallback &callback) = 0;
20
21          // Must be implemented by the platform
22          static Window *create(const str &title, const u32 &width, const
                u32 &height);
23  };
```

No Código A.10 os métodos virtuais puros devem ser implementados pela subclasse. O método *create* (linha 22) é encarregado de criar uma janela *glfw* e retornar um ponteiro

[1] Disponível em: https://www.glfw.org/

[2] Disponível em: https://glew.sourceforge.net/

para a classe criada. O Código A.11 mostra a subclasse *GlfwWindow*.

Código A.11 – Classe *GlfwWindow* (window.cpp).

```
GlfwWindow::GlfwWindow(const str &title, const u32 &width, const u32
    &height)
{
    window_data.title = title;
    window_data.width = width;
    window_data.height = height;

    // Initialize GLFW
    glewExperimental = true;
    if (!glfwInit())
        throw std::runtime_error("failed to initialize GLFW");

    // GLFW window hints
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);  // OpenGL Version 4.6
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
    glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, 1);  // Debug
    // glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GLFW_TRUE); // Apple
        killed OpenGL support

    // GLFW window and context
    native_window = glfwCreateWindow(width, height, title.c_str(), NULL,
        NULL);
    if (!native_window)
    {
        glfwTerminate();
        throw std::runtime_error("failed to create GLFW native_window");
    }

    // Initialize GLEW
    glfwMakeContextCurrent(native_window);
    glewExperimental = true;
    if (glewInit() != GLEW_OK)
        throw std::runtime_error("failed to initialize GLEW");

    // Get native_window dimensions
    glfwGetWindowSize(native_window,
                      const_cast<i32 *>(reinterpret_cast<const i32
                          *>(&width)),
                      const_cast<i32 *>(reinterpret_cast<const i32
                          *>(&height)));

// Set debug callbacks
#if defined(_DEBUG)
    if (GLEW_ARB_debug_output)
    {
        glDebugMessageCallbackARB(&debug_callback, NULL);
        glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS_ARB);
    }
```

```
45
46      else
47          LOG_WARNING("ARB Debug extension not supported");
48  #endif
49
50      // Disable sticky keys
51      glfwSetInputMode(native_window, GLFW_STICKY_KEYS, GLFW_FALSE);
52
53      // Hide the mouse and enable unlimited movement
54      // glfwSetInputMode(native_window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
55
56      // Disable VSync
57      glfwSwapInterval(0);
58
59      // Set callbacks
60      glfwSetWindowUserPointer(native_window, &window_data);
61
62      // Window close callback
63      glfwSetWindowCloseCallback(
64          native_window,
65          [](GLFWwindow *window)
66          {
67              auto &window_data =
68                  *reinterpret_cast<WindowData
                        *>(glfwGetWindowUserPointer(window));
69              WindowCloseEvent event = {};
70              window_data.event_callback(event);
71          });
72
73      // Resize callback
74      glfwSetFramebufferSizeCallback(
75          native_window,
76          [](GLFWwindow *window, i32 width, i32 height)
77          {
78              auto &window_data =
79                  *reinterpret_cast<WindowData
                        *>(glfwGetWindowUserPointer(window));
80              window_data.width = width;
81              window_data.height = height;
82
83              WindowResizeEvent event = {(u32)width, (u32)height};
84              window_data.event_callback(event);
85          });
86
87      // Key callback
88      glfwSetKeyCallback(
89          native_window,
90          [](GLFWwindow *window, i32 key, i32, i32 action, i32)
91          {
92              // Key press
93              if (action == GLFW_PRESS || action == GLFW_REPEAT)
94              {
95                  auto &window_data =
```

```
96                      *reinterpret_cast<WindowData
                            *>(glfwGetWindowUserPointer(window));
97                  KeyPressEvent event = {(u32)key};
98                  window_data.event_callback(event);
99              }
100         });
101
102     // Mouse callback
103     glfwSetCursorPosCallback(
104         native_window,
105         [](GLFWwindow *window, f64 x_position, f64 y_position)
106         {
107             auto &window_data =
108                  *reinterpret_cast<WindowData
                        *>(glfwGetWindowUserPointer(window));
109             MouseMoveEvent event = {x_position, y_position};
110             window_data.event_callback(event);
111         });
112
113     // Scroll callback
114     glfwSetScrollCallback(
115         native_window,
116         [](GLFWwindow *window, f64 x_offset, f64 y_offset)
117         {
118             auto &window_data = *reinterpret_cast<WindowData
                    *>(glfwGetWindowUserPointer(window));
119             MouseScrollEvent event = {x_offset, y_offset};
120             window_data.event_callback(event);
121         });
122 }
123
124 GlfwWindow::~GlfwWindow()
125 {
126     glfwTerminate();
127 }
128
129 void GlfwWindow::update()
130 {
131     // Poll events
132     glfwPollEvents();
133
134     // Swap buffers
135     glfwSwapBuffers(native_window);
136 }
137
138 void GlfwWindow::sleep(f64 seconds)
139 {
140     std::this_thread::sleep_for(
141         std::chrono::microseconds(static_cast<i64>(seconds * 1'000'000)));
142 }
143
144 void GlfwWindow::set_event_callback(const EventCallback &callback)
145 {
```

```
146     window_data.event_callback = callback;
147 }
148
149 u32 GlfwWindow::get_width() const
150 {
151     return window_data.width;
152 }
153
154 u32 GlfwWindow::get_height() const
155 {
156     return window_data.height;
157 }
158
159 f64 GlfwWindow::get_time() const
160 {
161     return glfwGetTime();
162 }
163
164 void *GlfwWindow::get_native_window() const
165 {
166     return native_window;
167 }
```

Primeiro, é feito a inicialização das bibliotecas *glfw* e *glew* (linhas 8-34). *Callbacks* de *debug* também são ativados caso o modo de execução seja _DEBUG (linhas 39-48). Alguns outros parâmetros como *sticky keys* e *vertical sync* são desativados (linhas 51-57). As linhas 60-121 configuram os *callbacks* para que a aplicação receba os eventos de janela emitidos pela *glfw*.

O método *update* (linha 129) é encarregado de emitir eventos (linha 132) e trocar os *buffers* da janela (linha 135). Esse método deve ser chamado ao final do *game loop*, após o término da execução do renderizador.

Por fim, os métodos restantes são *getters* (linhas 149-167) e *setters* (linhas 144-147) para diversos membros da classe.

A.9 Códigos do Gerenciador de *Input*

Os métodos da classe *GlfwInput* (Código A.12) utilizam funções da *glfw* para retornar o estado de teclas, botões do mouse e do controle. As linhas 1-6 verificam o estado das teclas. O estado do mouse é verificado nas linhas 8-13 e a posição do mouse nas linhas 41-60. Os botões e valores dos *joysticks* são calculados nas linhas 15-39. O funcionamento das funções *glfw* pode ser explorado na documentação oficial[3].

Código A.12 – Classe *GlfwInput* (input.cpp).

[3] Disponível em: https://www.glfw.org/docs/latest/input_guide.html

```
bool GlfwInput::is_key_pressed_native(i32 keycode)
{
    auto native_window = static_cast<GLFWwindow
        *>(Game::get().get_window().get_native_window());
    auto state = glfwGetKey(native_window, keycode);
    return state == GLFW_PRESS || state == GLFW_REPEAT;
}

bool GlfwInput::is_mouse_button_pressed_native(i32 button)
{
    auto native_window = static_cast<GLFWwindow
        *>(Game::get().get_window().get_native_window());
    auto state = glfwGetMouseButton(native_window, button);
    return state == GLFW_PRESS;
}

bool GlfwInput::is_joystick_button_pressed_native(i32 joystick, i32 button)
{
    bool state = false;
    if (glfwJoystickPresent(joystick))
    {
        i32 button_count;
        auto buttons = glfwGetJoystickButtons(joystick, &button_count);
        state = buttons[button];
    }
    return state == GLFW_PRESS;
}

f32 GlfwInput::get_joystick_axis_value_native(i32 joystick, i32 axis)
{
    f32 value = 0.0f;
    if (axis == GAMEPAD_AXIS_LEFT_TRIGGER || axis ==
        GAMEPAD_AXIS_RIGHT_TRIGGER) value = -1.0f;

    if (glfwJoystickPresent(joystick))
    {
        i32 axes_count;
        auto axes = glfwGetJoystickAxes(joystick, &axes_count);
        value = axes[axis];
    }
    return value;
}

std::pair<f32, f32> GlfwInput::get_mouse_position_native()
{
    auto window = static_cast<GLFWwindow
        *>(Game::get().get_window().get_native_window());
    f64 x_pos, y_pos;
    glfwGetCursorPos(window, &x_pos, &y_pos);

    return {(f32)x_pos, (f32)y_pos};
}
```

```
49
50 f32 GlfwInput::get_mouse_x_native()
51 {
52     auto [x, y] = get_mouse_position_native();
53     return x;
54 }
55
56 f32 GlfwInput::get_mouse_y_native()
57 {
58     auto [x, y] = get_mouse_position_native();
59     return y;
60 }
```