Paulo Ruan Oliveira Barbosa

Gustavo Mitsuo Fernandes Valente Takeda

Relatório do Projeto de Programação Funcional: Game Engine com C++ e ECS

Fortaleza - CE

2024

Introdução

Este relatório descreve os requisitos funcionais e não funcionais do projeto de desenvolvimento de uma game engine utilizando C++ e ECS (Entity Component System). O projeto foi desenvolvido como trabalho final da disciplina de Programação Funcional, sob orientação do Professor Samuel Lincoln Magalhaes Barrocas.

Requisitos Funcionais

Renderização Gráfica

Descrição: A engine deve ser capaz de renderizar objetos 2D e 3D em tempo real.

Implementação: O sistema de renderização utiliza shaders e suporta efeitos visuais avançados.

Gestão de Recursos

Descrição: Deve ser possível carregar, descarregar e gerenciar eficientemente recursos como texturas, modelos 3D, áudio e scripts durante a execução do jogo.

Implementação: O sistema de gestão de recursos utiliza técnicas eficientes para minimizar o tempo de carregamento e maximizar o desempenho durante a execução do jogo.

Sistema de Colisão

Descrição: Implementar um sistema de detecção e resposta de colisão entre objetos no ambiente do jogo.

Implementação: Utilização de algoritmos eficientes para detecção de colisões entre entidades no jogo.

Entrada de Usuário

Descrição: Capturar e processar inputs do usuário, como teclado, mouse e controle, para interação com o jogo.

Implementação: Utilização de bibliotecas de entrada para detectar e processar inputs do usuário.

Sistema de Partículas

Descrição: Capacidade de gerar e manipular sistemas de partículas para efeitos visuais dinâmicos.

Implementação: Implementação de um sistema de partículas flexível e configurável para criação de efeitos visuais dinâmicos no jogo.

Requisitos Não Funcionais

Performance

Descrição: Garantir que a engine seja capaz de manter uma taxa de quadros (FPS) estável, mesmo em ambientes com muitos objetos e efeitos visuais.

Implementação: Otimização do código para maximizar o desempenho da engine, utilizando técnicas de programação eficientes e algoritmos otimizados.

Portabilidade

Descrição: Desenvolver a engine de forma a ser facilmente portável para diferentes plataformas.

Implementação: Utilização de bibliotecas e padrões de programação que permitam a portabilidade da engine para diferentes sistemas operacionais e plataformas de hardware.

Documentação e Comentários de Código

Descrição: Todos os componentes da engine devem ser adequadamente documentados e comentados para facilitar a manutenção e compreensão do código.

Implementação: Inclusão de comentários explicativos e documentação detalhada para todos os componentes da engine, seguindo as melhores práticas de programação.

Escalabilidade

Descrição: Projetar a engine para ser escalável, permitindo que ela possa lidar com jogos de diferentes tamanhos e complexidades sem comprometer o desempenho.

Implementação: Utilização de um sistema de arquitetura modular e flexível que permita a adição de novas funcionalidades e acomode jogos de diferentes complexidades.

Usabilidade

Descrição: Criar uma interface de programação (API) intuitiva e fácil de usar para os desenvolvedores de jogos que utilizarão a engine.

Implementação: Projeto de uma API clara e bem documentada que facilite o desenvolvimento de jogos utilizando a engine, com suporte a recursos como autocompletar e sugestões de código.

Conclusão

Este relatório descreve os requisitos funcionais e não funcionais do projeto de desenvolvimento da game engine utilizando C++ e ECS. O projeto está em

conformidade com os requisitos estabelecidos e busca alcançar um equilíbrio entre funcionalidade e desempenho.

O código-fonte pode ser encontrado no repositório do GitHub: Link para o Repositório

Código com comentários

Main.cpp

```
#include <Base/Game.hpp>
#include <MessageBus/MessageBus.h>

// Estrutura de mensagem para comunicação
struct MSG_TYPE_1
{
  int i;
};

// Função regular para subscrição de mensagens
void RegularFunctionSubscriber( MSG_TYPE_1 msg )
{
  std::cout << "RegularFunctionSubscriber " << msg.i << std::endl;
}

// Classe que define um functor para subscrição de mensagens
class FunctorSubscriber
{
public:
  void operator()( MSG_TYPE_1 msg ) { std::cout << "FunctorSubscriber " << msg.i << std::endl;
}
};

int main()
{
  MSG_TYPE_1 msg1 = { 10 };  // Inicializa uma mensagem com o valor 10

  FunctorSubscriber functorSubscriber;  // Instancia um objeto functor

  // Subscrição de função regular
  SubscriberHandle handle1 = MsgBus<>::Subscribe< MSG_TYPE_1 >(RegularFunctionSubscriber);

  // Subscrição de functor
  SubscriberHandle handle2 = MsgBus<>::Subscribe< MSG_TYPE_1 >( functorSubscriber );

  // Subscrição de função lambda
  SubscriberHandle handle3 = MsgBus<>::Subscribe< MSG_TYPE_1 >( [](MSG_TYPE_1 msg)
                                                                { std::cout<< "Lambda Subscriber
" << msg.i << std::endl; } );

  // Publicação de mensagem de forma bloqueante
  MsgBus<>::PublishBlocking( msg1 );
  std::cout << std::endl;

  // Publicação de mensagem de forma assíncrona
  MsgBus<>::PublishAsync( msg1 );
  std::cout << std::endl;

  // Cancelamento da subscrição de functor
```

```
  MsgBus<>::UnSubscribe( handle2 );
  std::cout << std::endl;

  // Publicação de mensagem de forma bloqueante novamente
  MsgBus<>::PublishBlocking( msg1 );
  std::cout << std::endl;

  // Publicação de mensagem de forma assíncrona novamente
  MsgBus<>::PublishAsync( msg1 );
  std::cout << std::endl;

  // Inicializa e executa o jogo
  Game game;
  game.init();
  game.run();

  return 0;
}
```

Main.cpp comentado detalhadamente

1. Estrutura de Mensagem (MSG_TYPE_1):

Define uma estrutura simples contendo um único inteiro. Esta estrutura é usada para a comunicação através do barramento de mensagens (MessageBus).

2. Função Regular para Subscrição (RegularFunctionSubscriber):

Função que imprime o valor do campo i da estrutura MSG_TYPE_1. É utilizada para exemplificar a subscrição de uma função regular ao barramento de mensagens.

3. Classe Functor para Subscrição (FunctorSubscriber):

Define um functor (objeto que pode ser chamado como uma função) que também imprime o valor do campo i da estrutura MSG_TYPE_1. Demonstra a subscrição de objetos functors ao barramento de mensagens.

4. Função main():

Instanciação da mensagem: Cria uma instância de MSG_TYPE_1 com valor 10.

Subscrição de diferentes tipos de funções:

Função regular.

Functor.

Função lambda.

Publicação de mensagens:

Publicação bloqueante e assíncrona de mensagens.

Cancelamento de subscrição (exemplo com functor).

Re-publicação para mostrar o efeito do cancelamento de subscrição.

Inicialização e execução do jogo:

Demonstra a inicialização (init()) e execução (run()) do objeto Game.

Conceitos de Programação Funcional Utilizados:

1. Função Lambda de Alta Ordem:

Subscrição de função lambda: SubscriberHandle handle3 = MsgBus<>::Subscribe< MSG_TYPE_1 >([](MSG_TYPE_1 msg) { std::cout<< "Lambda Subscriber " << msg.i << std::endl; });

ECS.cpp e ECS.hpp respectivamente

```
#include <ECS/ECS.hpp>

// Método para criar uma nova entidade no mundo
EntityID World::NewEntity()
{
  // Se houver entidades livres (anteriormente destruídas), reutilize uma delas
  if(!free_entites.empty())
  {
    EntityIndex new_index = free_entites.back();  // Pega o último índice disponível
    free_entites.pop_back();  // Remove o índice da lista de entidades livres
    EntityID newID = CreateEntityId(new_index, GetEntityVersion(entities[new_index].id));  // 
Cria um novo ID de entidade com a versão atual
    entities[new_index].id = newID;  // Atualiza o ID da entidade
    return entities[new_index].id;  // Retorna o novo ID
  }

  // Caso contrário, cria uma nova entidade adicionando ao vetor de entidades
  entities.push_back({CreateEntityId(EntityIndex(entities.size()), 0), ComponentMask()});
  return entities.back().id;  // Retorna o ID da nova entidade
}

// Método para destruir uma entidade no mundo
void World::DestroyEntity(EntityID id)
{
  // Cria um novo ID para a entidade com um índice inválido (-1) e a versão incrementada
  EntityID newID = CreateEntityId(EntityIndex(-1), GetEntityVersion(id + 1));
  entities[GetEntityIndex(id)].id = newID;  // Atualiza o ID da entidade destruída
  entities[GetEntityIndex(id)].mask.reset();  // Reseta a máscara de componentes da entidade
  free_entites.push_back(GetEntityIndex(id));  // Adiciona o índice da entidade à lista de 
entidades livres
}
```

ECS.cpp comentado detalhadamente

1. Método NewEntity():

Objetivo: Criar uma nova entidade no mundo do jogo.

1.1 Reutilização de Entidades:

Se houver entidades previamente destruídas (armazenadas em free_entites), reutiliza uma dessas entidades, atribuindo um novo ID e removendo-a da lista de entidades livres.

1.2 Criação de Novas Entidades:

Se não houver entidades livres, cria uma nova entidade, adicionando-a ao vetor entities com um novo ID e uma máscara de componentes vazia (ComponentMask).

2. Método DestroyEntity():

Objetivo: Destruir uma entidade existente no mundo do jogo.

Destruição Lógica:

2.1 Cria um novo ID para a entidade, marcando-a como inválida (índice -1) e incrementando a versão para diferenciar de IDs anteriores.

2.2 Reseta a máscara de componentes da entidade, removendo todos os componentes associados.

2.3 Adiciona o índice da entidade destruída à lista de entidades livres (free_entites), para reutilização futura.

3. Pontos Importantes:

Gestão de Entidades:

3.1 O sistema de gerenciamento de entidades permite a reutilização eficiente de IDs de entidades destruídas, evitando a fragmentação de IDs e mantendo o desempenho.

3. Máscara de Componentes (ComponentMask):

As máscaras de componentes são usadas para rastrear quais componentes estão associados a cada entidade, facilitando operações típicas de um sistema ECS (Entity Component System).

```
#pragma once
#include "entt/entity/fwd.hpp"
#include <bitset>
#include <cstddef>
#include <cstdint>
#include <vector>
#include <entt/entt.hpp>

extern int s_componentCounter; // Contador global para IDs de componentes
typedef uint64_t EntityID; // Definição de tipo para IDs de entidades
const int MAX_COMPONENTS = 32; // Número máximo de componentes
using ComponentMask = std::bitset<MAX_COMPONENTS>; // Máscara de bits para componentes
using ECSWorld = entt::registry; // Definição do tipo para o registro ECS usando a biblioteca EnTT

typedef uint64_t EntityIndex; // Definição de tipo para o índice da entidade
typedef uint64_t EntityVersion; // Definição de tipo para a versão da entidade

// Criação de um ID de entidade combinando o índice e a versão
inline EntityID CreateEntityId(EntityIndex index, EntityVersion version)
{
  return ((EntityID)index << 32) | ((EntityID)version);
}
```

```
// Extração do índice da entidade a partir do ID
inline EntityIndex GetEntityIndex(EntityID id)
{
  return id >> 32;
}

// Extração da versão da entidade a partir do ID
inline EntityVersion GetEntityVersion(EntityID id)
{
  return (EntityVersion)id;
}

// Verificação se o ID da entidade é válido
inline bool IsEntityValid(EntityID id)
{
  return (id >> 32) != EntityIndex(-1);
}

// Definição de um ID de entidade inválido
#define INVALID_ENTITY CreateEntityId(EntityIndex(-1), 0)

// Função template para obter o ID de um componente
template <class T>
int GetId()
{
  static int s_componentID = s_componentCounter++;
  return s_componentID;
}

// Estrutura para o pool de componentes
struct ComponentPool
{
  ComponentPool(size_t element_size)
  {
    this->element_size = element_size;
    pData = new char[element_size * MAX_COMPONENTS];
  }

  ~ComponentPool()
  {
    delete[] pData;
  }

  inline void *get(size_t index)
  {
    return pData + index * element_size;
  }

  char *pData = nullptr;
  size_t element_size = 0;
};
```

```
// Estrutura para representar o mundo ECS
struct World
{
  struct EntityDesc
  {
    EntityID id;
    ComponentMask mask;
  };

  std::vector<EntityDesc> entities; // Vetor de entidades
  std::vector<EntityIndex> free_entites; // Vetor de índices de entidades livres
  std::vector<ComponentPool*> componentPools; // Vetor de pools de componentes

  // Método para criar uma nova entidade
  EntityID NewEntity();
  // Método para destruir uma entidade
  void DestroyEntity(EntityID id);

  // Método template para atribuir um componente a uma entidade
  template <typename T>
  void AssignComponent(EntityID id)
  {
    int componentID = GetId<T>();
    entities[GetEntityIndex(id)].mask.set(componentID);
  }

  // Método template para atribuir e inicializar um componente a uma entidade
  template <typename T>
  T *AssignComponent(EntityID id)
  {
    int componentId = GetId<T>();

    if(componentPools.size() <= componentId)
    {
      componentPools.resize(componentId + 1, nullptr);
    }

    if(componentPools[componentId] == nullptr)
    {
      componentPools[componentId] = new ComponentPool(sizeof(T));
    }

    T *pComponent = new (componentPools[componentId]->get(GetEntityIndex(id))) T();

    entities[GetEntityIndex(id)].mask.set(componentId);
    return pComponent;
  }

  // Método template para obter um componente de uma entidade
  template <typename T>
  T *get(EntityID id)
```

```
  {
    int componentId = GetId<T>();
    if(!entities[GetEntityIndex(id)].mask.test(componentId))
    {
      return nullptr;
    }

    T *pComponent = static_cast<T*>(componentPools[componentId]->get(GetEntityIndex(id)));
    return pComponent;
  }
};
```

ECS.hpp comentado detalhadamente

1. Definições de Tipos e Constantes:

1.1 EntityID, EntityIndex, EntityVersion: Tipos definidos para representar IDs, índices e versões de entidades.

1.2 ComponentMask: Máscara de bits usada para rastrear componentes associados a uma entidade.

ECSWorld: Tipo para o registro ECS usando a biblioteca EnTT.

2. Funções Inline:

2.1 CreateEntityId, GetEntityIndex, GetEntityVersion: Funções para manipulação e extração de partes do ID da entidade.

2.2 IsEntityValid: Verifica se um ID de entidade é válido.

3. Função Template GetId:

3.1 Gera um ID único para cada tipo de componente, utilizando um contador global s_componentCounter.

4. Estrutura ComponentPool:

4.1 Armazena dados de componentes de maneira contígua na memória, facilitando o acesso rápido e eficiente.

5. Estrutura World:

5.1 Contém o vetor de entidades (entities), índices de entidades livres (free_entites) e pools de componentes (componentPools).

5.2 Método NewEntity: Cria uma nova entidade, reutilizando índices de entidades destruídas quando possível.

5.3 Método DestroyEntity: Marca uma entidade como destruída e a adiciona à lista de entidades livres.

5.4 Métodos Templates AssignComponent e get: Gerenciam a atribuição e recuperação de componentes para entidades.

Conceitos de Programação Funcional Aplicáveis:

Funções Lambda: Pode ser utilizado para operações específicas em componentes ou entidades, especialmente em operações de sistema.

Alto Nível de Abstração: O uso de templates para componentes permite um alto nível de reutilização e generalização do código.

Imutabilidade: A imutabilidade pode ser aplicada nos componentes ou sistemas ECS para garantir segurança e consistência de dados.

Game.cpp e Game.hpp respectivamente

```cpp
#include "raylib.h"
#include <Base/Game.hpp>

// Inicializa o jogo
void Game::init()
{
  // Verifica se o arquivo "ini.lua" existe
  if(!file_exists("ini.lua"))
  {
    // Define as dimensões da janela padrão
    windowW = d_windowW;
    windowH = d_windowH;
  }

  // Inicializa a janela
  InitWindow(windowW, windowH, "Game");

  // Configuração da câmera
  cam = {0};
  cam.position = (Vector3){10.0f, 10.0f, 10.0f};
  cam.target = (Vector3){0.0f, 0.0f, 0.0f};
  cam.up = (Vector3){0.0f, 1.0f, 0.0f};
  cam.fovy = 45.0f;
  cam.projection = CAMERA_PERSPECTIVE;

  // Verifica se a janela foi inicializada corretamente
  if(IsWindowReady())
  {
    is_running = true;
  }

  // Define a tecla de saída para nula
  SetExitKey(KEY_NULL);

  // Inicializa as bibliotecas do Lua
  g_lstate.open_libraries(sol::lib::base, sol::lib::os, sol::lib::utf8, sol::lib::math);

  // Desabilita o cursor
  DisableCursor();

  // Define o FPS alvo
  SetTargetFPS(60);
}

// Manipula as entradas do usuário
void Game::handle_input()
{
  // Se a tecla F2 for pressionada, o jogo para
```

```
  if(IsKeyPressed(KEY_F2))
  {
    is_running = false;
  }
}

// Atualiza o estado do jogo
void Game::update()
{
  // Atualiza a câmera
  UpdateCamera(&cam, CAMERA_FREE);
}

// Desenha na tela
void Game::draw()
{
  BeginDrawing();

  // Limpa o fundo com a cor branca
  ClearBackground(RAYWHITE);

  BeginMode3D(cam);

    // Desenha um cubo azul na posição especificada
    DrawCube(cube_pos, 2.0f, 2.0f, 2.0f, BLUE);

    // Desenha uma grade no plano
    DrawGrid(10, 1.0f);

  EndMode3D();

  EndDrawing();
}

// Loop principal do jogo
void Game::run()
{
  SetTargetFPS(60);

  while(is_running)
  {
    handle_input();
    update();
    draw();
  }
  deinit();
}

// Desinicializa o jogo
void Game::deinit()
{
  // Fecha a janela
```

```
  CloseWindow();
}
```

Game.cpp comentado detalhadamente

1. Método init():

Objetivo: Inicializar o estado do jogo, configurar a janela, a câmera e outras configurações iniciais.

1.1 Verificação de Arquivo: Verifica se o arquivo ini.lua existe para definir as dimensões da janela.

1.2 Configuração da Janela: Utiliza a biblioteca raylib para inicializar a janela com as dimensões especificadas.

1.3 Configuração da Câmera: Define a posição, o alvo, a direção "para cima" e o campo de visão (FOV) da câmera.

1.4 Bibliotecas Lua: Inicializa bibliotecas Lua para scripting dentro do jogo.

1.5 Cursor e FPS: Desabilita o cursor e define o FPS alvo para 60.

2. Método handle_input():

Objetivo: Gerenciar a entrada do usuário.

2.1 Parar o Jogo: Se a tecla F2 for pressionada, o jogo para (is_running é definido como false).

3. Método update():

Objetivo: Atualizar o estado do jogo a cada frame.

3.1 Atualização da Câmera: Utiliza a função UpdateCamera da raylib para atualizar a posição da câmera com o modo CAMERA_FREE.

4. Método draw():

Objetivo: Renderizar o conteúdo na tela.

4.1 Desenho em 3D: Inicia o modo 3D, desenha um cubo azul e uma grade, e depois finaliza o modo 3D.

5. Método run():

Objetivo: Loop principal do jogo.

5.1 Loop do Jogo: Continua chamando handle_input, update e draw enquanto is_running for true.

5.2 Desinicialização: Chama deinit após sair do loop.

6. Método deinit():

Objetivo: Limpar e fechar o jogo corretamente.

6.1 Fechar a Janela: Utiliza CloseWindow da raylib para fechar a janela do jogo.

Pontos Importantes:

Gestão da Janela e Câmera: Utiliza a raylib para configurar e gerenciar a janela e a câmera.

Entradas do Usuário: Trata entradas específicas do usuário, como pressionar a tecla F2 para parar o jogo.

Loop de Jogo: Estrutura básica de um loop de jogo com inicialização, atualização, renderização e desinicialização.

Uso de Lua: Inicializa bibliotecas Lua, sugerindo potencial para scripting no jogo.

```
#pragma once
#include "ECS/ECS.hpp"
#include <filesystem>
#include <raylib.h>
#include <sol/sol.hpp>
#include <string>

// Estrutura que representa o jogo
struct Game
{
  // Métodos principais do ciclo de vida do jogo
  void run();
  void init();
  void setup();
  void handle_input();
  void update();
  void draw();
  void deinit();

  // Estado Lua
  sol::state g_lstate;

  // Constantes para a largura e altura da janela padrão
  static constexpr int d_windowW = 720;
  static constexpr int d_windowH = 480;

  // Câmera 3D e posição do cubo
  Camera3D cam;
  Vector3 cube_pos = {0.0f, 0.0f, 0.0f};

  // Variáveis de resolução da janela
  int windowW;
  int windowH;
  int vResolutionW;
  int vResolutionH;

  // Flags de depuração e execução
  bool debug = false;
  bool is_running = false;

  // Mundo ECS
  ECSWorld world;
};
```

```cpp
// Função inline para verificar a existência de um arquivo
inline bool file_exists(const std::string &name)
{
  return std::filesystem::exists(name);
}
```

Game.hpp comentado detalhadamente

1. Headers Incluídos:

1.1 ECS/ECS.hpp: Possivelmente define as estruturas e funcionalidades do sistema de entidades e componentes (ECS).

1.2 filesystem: Usado para operações de sistema de arquivos, como verificar se um arquivo existe.

1.3 raylib.h: Biblioteca de gráficos usada para renderização e manipulação de janelas e entradas.

1.4 sol/sol.hpp: Biblioteca para integração Lua/C++.

1.5 string: Biblioteca padrão para manipulação de strings.

1.6 Pragma Once: Garantia de que o header será incluído apenas uma vez na compilação.

2. Estrutura Game:

Métodos:

2.1 run(): Método principal que executa o loop do jogo.

2.2 init(): Inicializa o jogo, configurando a janela, a câmera e outros elementos.

2.3 setup(): Método sugerido para configuração adicional (não implementado no Game.cpp fornecido).

2.4 handle_input(): Gerencia as entradas do usuário.

2.5 update(): Atualiza o estado do jogo.

2.6 draw(): Renderiza o conteúdo na tela.

2.7 deinit(): Limpa os recursos e fecha o jogo.

2.8 Estado Lua: sol::state g_lstate para gerenciar o estado do Lua no jogo.

2.9 Constantes de Janela: d_windowW e d_windowH definem a largura e altura padrão da janela.

2.10 Câmera 3D: cam para a câmera e cube_pos para a posição do cubo a ser desenhado.

2.11 Resolução da Janela: windowW, windowH, vResolutionW, vResolutionH armazenam a resolução atual da janela.

3. Flags:

3.1 debug: Usado para habilitar/desabilitar o modo de depuração.

3.2 is_running: Indica se o jogo está em execução.

3.3 Mundo ECS: ECSWorld world gerencia as entidades e componentes do jogo.

4. Função Inline file_exists:

Objetivo: Verifica se um arquivo existe no sistema de arquivos.

4.1 Implementação: Utiliza a função std::filesystem::exists para checar a existência do arquivo.

Pontos Importantes:

Estrutura Clara: A estrutura Game está bem definida com métodos claros para cada etapa do ciclo de vida do jogo.

Uso de Bibliotecas: Integração de várias bibliotecas (raylib, sol, std::filesystem) para fornecer funcionalidades essenciais.

Flags de Controle: Variáveis debug e is_running permitem controlar o estado do jogo e o modo de depuração.

ECS: Integração com um sistema de entidades e componentes (ECSWorld) para gerenciar a lógica do jogo de forma modular.

Game.cpp e Game.hpp respectivamente

```cpp
#include <Allocators/GenArena.hpp>
#include <cstdint>

    // Método para alocar um novo índice geracional
    GenIndex GenerationalIndexAllocator::allocate()
    {
     if(m_free_indices.size() > 0)
     {
       // Reutilizar um índice livre se disponível
       uint64_t index = m_free_indices.back();
       m_free_indices.pop_back();

       // Incrementar a geração e marcar como ativo
       m_entries[index].generation += 1;
       m_entries[index].is_alive = true;

       return {index, m_entries[index].generation};
     }
     else
     {
       // Adicionar um novo índice se não houver livre
       m_entries.push_back({true, 0});
       return {static_cast<uint64_t>(m_entries.size()) - 1, 0};
     }
    }

    // Método para desalocar um índice geracional
    void GenerationalIndexAllocator::deallocate(GenIndex index)
    {
```

```
        if(is_alive(index))
        {
          // Marcar o índice como inativo e adicioná-lo à lista de livres
          m_entries[index.index].is_alive = false;
          m_free_indices.push_back(index.index);
        }
       }

       // Método para verificar se um índice está ativo
       bool GenerationalIndexAllocator::is_alive(GenIndex index) const
       {
        return index.index < m_entries.size() &&
               m_entries[index.index].generation == index.generation &&
               m_entries[index.index].is_alive;
}
```

GenArena.cpp comentado detalhadamente

1. **Inclusões e Definições:**

   Headers Incluídos:

   `Allocators/GenArena.hpp`: Provavelmente define a classe `GenerationalIndexAllocator` e a estrutura `GenIndex`.

   `cstdint`: Biblioteca padrão C++ para tipos de inteiros de tamanho fixo (`uint64_t`).

2. **Método `allocate`:**

   Objetivo: Alocar um novo índice geracional.

   Reutilização de Índices:

   Se houver índices livres (`m_free_indices`), reutiliza o último índice da lista.

   Incrementa a geração do índice reutilizado e marca como ativo (`is_alive = true`).

   Novo Índice:

   Se não houver índices livres, adiciona um novo índice à lista (`m_entries`).

Retorna o novo índice com geração 0.

3. **Método `deallocate`:**

   Objetivo: Desalocar um índice geracional.

   Verificação de Atividade:

   Verifica se o índice está ativo usando `is_alive`.

   Se ativo, marca como inativo (`is_alive = false`) e adiciona o índice à lista de livres (`m_free_indices`).

4. **Método `is_alive`:**

   Objetivo: Verificar se um índice está ativo.

   Verificação:

   Verifica se o índice está dentro do tamanho da lista de entradas (`m_entries`).

   Compara a geração do índice fornecido com a geração armazenada.

   Verifica se o índice está marcado como ativo (`is_alive`).

Pontos Importantes:

Reutilização Eficiente: O método `allocate` reutiliza índices desalocados para evitar desperdício de memória.

Incremento de Geração: A cada reutilização de um índice, sua geração é incrementada para distinguir entre alocações diferentes do mesmo índice.

Verificação de Atividade: O método `is_alive` garante que apenas índices válidos e ativos sejam utilizados, prevenindo erros de acesso.

```cpp
#pragma once
#include <cstdint>
#include <cstdlib>
#include <functional>
#include <optional>
#include <tuple>
#include <vector>

// Estrutura para representar um índice geracional
struct GenIndex
{
  uint64_t index = 0;
  uint64_t generation = 0;
```

```
};

// Estrutura para alocador de índice geracional
struct GenerationalIndexAllocator
{
  // Estrutura para representar uma entrada no alocador
  struct AllocatorEntry
  {
    bool is_alive = false;
    uint64_t generation = 0;
  };

  // Vetor de entradas do alocador
  std::vector<AllocatorEntry> m_entries;
  // Vetor de índices livres
  std::vector<uint64_t> m_free_indices;

  // Métodos para alocar, desalocar e verificar se um índice está ativo
  GenIndex allocate();
  void deallocate(GenIndex index);
  bool is_alive(GenIndex index) const;
};

// Template para um array de índices geracionais
template <typename T>
struct GenIndexArray
{
  // Estrutura para representar uma entrada no array
  struct Entry
  {
    uint64_t generation;
    T value;
  };

  // Vetor de entradas opcionais
  std::vector<std::optional<Entry>> m_entries;

  // Método para definir um valor no array
  void set(GenIndex index, T value)
  {
    // Aumentar o tamanho do vetor se necessário
    while(m_entries.size() <= index.index)
    {
      m_entries.push_back(std::nullopt);
    }

    uint64_t prev_gen = 0;

    // Obter a geração anterior, se existir
    if(auto prev_entry = m_entries[index.index])
    {
      prev_gen = prev_entry->generation;
```

```
    }

    // Se a geração anterior for maior, sair com erro
    if(prev_gen > index.generation)
    {
      exit(1);
    }

    // Definir o novo valor com a geração correspondente
    m_entries[index.index] = std::optional<Entry>{{index.generation, value}};
  }

  // Método para remover um valor do array
  void remove(GenIndex index)
  {
    if(index.index < m_entries.size())
    {
      m_entries[index.index] = std::nullopt;
    }
  }

  // Método para obter um valor do array
  T *get(GenIndex index)
  {
    if(index.index >= m_entries.size())
    {
      return nullptr;
    }

    if(auto &entry = m_entries[index.index])
    {
      if(entry->generation == index.generation)
      {
        return &entry->value;
      }
    }

    return nullptr;
  }

  // Método constante para obter um valor do array
  const T *get(GenIndex index) const
  {
    return const_cast<const T*>(const_cast<GenIndexArray*>(this)->get(index));
  }

  // Método para obter todos os índices válidos
  std::vector<GenIndex> get_all_valid_indices(const GenerationalIndexAllocator &allocator)
const
  {
    std::vector<GenIndex> result;
```

```
    for(int i = 0; i < m_entries.size(); ++i)
    {
      const auto &entry = m_entries[i];
      if(!entry)
      {
        continue;
      }

      GenIndex index = {i, entry->generation};

      if(allocator.is_alive(index))
      {
        result.push_back(index);
      }
    }

    return result;
  }

  // Método para obter a primeira entrada válida
  std::optional<std::tuple<GenIndex, std::reference_wrapper<const T>>>
get_first_valid_entry(const GenerationalIndexAllocator &allocator) const
  {
    for(auto i = 0; i < m_entries.size(); ++i)
    {
      const auto &entry = m_entries[i];
      if(!entry)
      {
        continue;
      }

      GenIndex index = {i, entry->generation};

      if(allocator.is_alive(index))
      {
        return std::make_tuple(index, std::ref(entry->value));
      }
    }

    return std::nullopt;
  }
};
```

GenArena.hpp comentado detalhadamente

1. Definições e Estruturas:

GenIndex:

Representa um índice geracional com campos `index` e `generation`.

`GenerationalIndexAllocator`:

Gerencia alocação e desalocação de índices geracionais.

Utiliza `AllocatorEntry` para armazenar se um índice está ativo (`is_alive`) e sua geração.

`GenIndexArray`:

Template que gerencia um array de índices geracionais para qualquer tipo `T`.

Utiliza `Entry` para armazenar valores com suas respectivas gerações.

2. Métodos do `GenerationalIndexAllocator`:

`allocate`:

Aloca um novo índice geracional reutilizando índices livres quando possível.

Incrementa a geração ao reutilizar um índice.

`deallocate`:

Marca um índice como inativo e o adiciona à lista de índices livres.

`is_alive`:

Verifica se um índice está ativo comparando o índice, a geração e o estado `is_alive`.

3. Métodos do `GenIndexArray`:

`set`:

Define um valor no array garantindo que a geração seja válida.

Aumenta o vetor de entradas conforme necessário.

`remove`:

Remove um valor do array definindo a entrada como `std::nullopt`.

`get`:

Retorna um ponteiro para o valor correspondente ao índice e geração fornecidos, ou `nullptr` se inválido.

`get_all_valid_indices`:

Retorna todos os índices válidos, verificando com o `GenerationalIndexAllocator`.

`get_first_valid_entry`:

Retorna a primeira entrada válida como uma tupla contendo o índice e uma referência ao valor.

Pontos Importantes:

- **Gerenciamento de Memória**: Uso de índices livres para reutilizar memória de maneira eficiente.
- **Segurança**: Verificações de geração garantem que apenas entradas válidas sejam acessadas.
- **Uso de `std::optional`**: Permite a presença ou ausência de valores de forma explícita.