Instituto Superior de Engenharia
Politécnico de Coimbra

Trabalho Prático
Programação Orientada a Objetos
2024/2025

POO

Alexandre Moreira
a2020144214@isec.pt

Licenciatura em Engenharia Informática
ISEC

Coimbra, 29 de Novembro 2024

Índice

Capítulo 1

Introdução

1.1 Introdução

Este trabalho tem como objetivo o desenvolvimento de um simulador que gere interações entre caravanas, itens e outros elementos num mapa dinâmico. O simulador permite modelar e observar o comportamento de diferentes tipos de caravanas em cenários com recursos limitados, eventos imprevisíveis e interações que podem alterar o estado de cada entidade no sistema.

O projeto foi concebido como uma aplicação prática dos conceitos de Programação Orientada a Objetos (POO), com ênfase no uso de herança, polimorfismo, encapsulamento e gerenciamento eficiente de memória em C++. Além disso, o sistema implementa mecanismos de interação entre classes distintas, como *Caravana*, *Item* e *Mapa*, criando uma simulação funcional e modular.

No decorrer do desenvolvimento, foi necessário garantir que o simulador gerenciasse corretamente o estado de cada entidade, incluindo a movimentação autônoma das caravanas, o consumo de recursos, os efeitos de itens encontrados no mapa, e os combates entre caravanas de diferentes tipos. Adicionalmente, foram implementadas funcionalidades que permitem a interação com o usuário, como salvar e carregar estados do sistema e manipular configurações iniciais.

Este relatório está organizado da seguinte forma:

- Na **Seção 2**, são apresentados os requisitos do sistema e os objetivos;
- Na **Seção 3**, descrevemos o design e a arquitetura do projeto, incluindo o diagrama de classes e a estrutura geral do código;
- Na **Seção 4**, detalhamos as principais decisões e implementações feitas ao longo do trabalho;
- Na **Seção 5**, conlsuão.

Capítulo 2

Objetivos

2.1 Objetivos

Este projeto tem como objetivo principal a criação de um simulador que modela as interações entre caravanas, itens e outros elementos dispostos em um mapa dinâmico. A seguir, apresentamos os detalhes do problema e os requisitos que orientaram o desenvolvimento do simulador.

2.1.1 Descrição dos Objetivos

No simulador, várias caravanas percorrem um mapa enquanto enfrentam desafios como escassez de recursos, encontros com itens de efeito variado e combates. Cada tipo de caravana (comercial, militar, secreta, bárbara) possui características e comportamentos únicos que influenciam a simulação. Adicionalmente, eventos como tempestades e itens no mapa adicionam uma camada de complexidade ao sistema.

- **Caravanas**: As caravanas podem se mover de forma autônoma ou manual, consumir água, transportar tripulantes e carga, e podem ser destruídas por eventos ou falta de recursos.
- **Mapa**: O mapa representa o ambiente do simulador, contendo cidades, montanhas e áreas transitáveis. Os itens e caravanas são representados em posições específicas no mapa.

- **Itens**: Os itens têm efeitos distintos quando encontrados por uma caravana, como aumentar a tripulação, paralisá-la temporariamente ou destruí-la.
- **Combate**: Combates entre caravanas ocorrem quando elas estão em posições adjacentes, sendo determinados por regras baseadas no número de tripulantes de cada uma.

2.1.2 Requisitos do Sistema

Para atender aos objetivos do simulador, os seguintes requisitos foram estabelecidos:

- **Gestão de Caravanas**: O sistema deve permitir criar caravanas de diferentes tipos e coordenar seus movimentos, consumo de recursos e interações.
- **Mapeamento do Ambiente**: Deve ser possível representar o mapa com elementos como cidades, montanhas, e posicionar caravanas e itens dinamicamente.
- **Interação com Itens**: Quando uma caravana encontra um item no mapa, o efeito associado ao item deve ser aplicado imediatamente.
- **Simulação de Combate**: O simulador deve gerenciar combates entre caravanas, determinando o vencedor e as consequências para cada uma.
- **Sistema de Eventos**: Eventos como tempestades devem alterar o estado das caravanas e do mapa de forma imprevisível.
- **Interface de Controle**: O simulador deve aceitar comandos do usuário para realizar operações como mover caravanas, exibir informações e salvar estados.

2.1.3 Soluções Propostas

Para resolver o problema descrito, foram adotadas as seguintes estratégias:

- Utilização de Programação Orientada a Objetos (POO) para modelar os principais elementos da simulação, como caravanas, itens e o mapa.
- Implementação de herança e polimorfismo para permitir comportamentos distintos entre os diferentes tipos de caravanas.
- Uso de estruturas de dados como vetores para gerenciar caravanas e itens de forma eficiente.
- Aplicação de algoritmos para interações dinâmicas, como verificações de proximidade entre itens e caravanas e a execução de combates.
- Implementação de um sistema de log detalhado para registrar eventos importantes durante a simulação.

Com base nesses requisitos e soluções propostas, o próximo capítulo apresenta o design e a arquitetura do sistema, detalhando as principais classes e suas interações.

Capítulo 3

Funcionamento

3.1 Design e Arquitetura

Neste capítulo, apresentamos o design e a arquitetura do simulador, detalhando as classes principais, suas interações e como o sistema foi estruturado para atender aos requisitos definidos anteriormente. Este capítulo inclui um diagrama de classes e explicações sobre o papel de cada componente no sistema.

3.1.1 Visão Geral do Sistema

O simulador foi projetado usando os princípios da Programação Orientada a Objetos (POO), com um foco especial em modularidade, reutilização de código e separação de responsabilidades. O sistema é composto por diferentes módulos que interagem entre si, como o módulo de simulação, o mapa, as caravanas e os itens.

O objetivo central do design é garantir que as interações entre as entidades (caravanas, itens e mapa) sejam implementadas de forma clara e extensível. Para isso, a arquitetura utiliza herança e polimorfismo para modelar os diferentes comportamentos das caravanas e itens, permitindo adicionar novos tipos de entidades futuramente com impacto mínimo no restante do sistema.

3.1.2 Classes

Classes principais:

- **Simulador**: Controla o ciclo principal da simulação, gerencia os eventos e coordena as interações entre as caravanas, itens e o mapa.
- **Caravana**: Classe base para representar caravanas. Possui subclasses (*CaravanaComercial*, *CaravanaMilitar*, etc.) que implementam comportamentos específicos.
- **Item**: Representa os itens no mapa, cada um com um tipo e um efeito associado.
- **Mapa**: Responsável por armazenar a representação espacial do simulador, como as posições de caravanas, itens, cidades e montanhas.

3.1.3 Descrição das Classes Principais

Simulador

A classe `Simulador` é o núcleo do sistema, responsável por gerenciar o estado geral da simulação. As principais funções dessa classe incluem:

- Gerenciar as caravanas e itens ativos no mapa.
- Controlar o ciclo principal da simulação, como mover caravanas, processar combates e aplicar efeitos de itens.
- Fornecer uma interface para que o usuário interaja com o sistema, permitindo emitir comandos como *mover caravanas*, *salvar estados* e *carregar mapas*.

Caravana

A classe `Caravana` é uma classe base que define os atributos e comportamentos comuns a todas as caravanas. Suas subclasses (`CaravanaComercial`, `CaravanaMilitar`, etc.) implementam comportamentos específicos, como:

- `CaravanaComercial`: Foca em coletar e transportar itens no mapa.
- `CaravanaMilitar`: Especializada em combates com caravanas bárbaras e proteção de outras caravanas.

Os atributos principais de uma caravana incluem:

- `linha, coluna`: Posição no mapa.
- `tripulacao`: Número de tripulantes.
- `carga`: Quantidade de carga transportada.
- `agua`: Quantidade de água disponível.
- `estadoAutonoma`: Indica se a caravana está se movendo de forma autônoma.

Item

A classe `Item` representa os itens que podem ser encontrados no mapa. Cada item possui:

- Um tipo (`CaixaPandora`, `ArcaTesouro`, etc.).
- Um efeito específico que é aplicado à caravana que o encontra.
- Uma posição no mapa (`linha, coluna`).

Os efeitos dos itens são definidos no método `aplicarEfeito`, que é chamado pelo simulador quando uma caravana encontra o item.

Mapa

A classe `Mapa` gerencia a representação espacial da simulação. Suas principais responsabilidades incluem:

- Armazenar a matriz do mapa e suas células.
- Representar cidades, montanhas, itens e caravanas no mapa.
- Fornecer métodos para verificar posições adjacentes e atualizar células do mapa.

3.1.4 Interações Entre as Classes

As classes interagem entre si da seguinte forma:

- O `Simulador` coordena a interação entre `Caravana`, `Item` e `Mapa`.
- A `Caravana` consulta o `Mapa` para determinar sua posição e interagir com itens.
- O `Item` aplica efeitos diretamente na `Caravana` ao ser encontrado.
- O `Mapa` atualiza a posição das caravanas e itens conforme o simulador avança.

Capítulo 4

Implementação

4.1 Implementação

Neste capítulo, detalhamos as principais decisões e implementações realizadas ao longo do desenvolvimento do simulador. Explicamos como os requisitos e o design foram traduzidos em código funcional, abordando as funcionalidades principais e os desafios encontrados durante a implementação.

4.1.1 Estrutura do Projeto

O projeto foi implementado em C++ e estruturado em módulos separados, cada um representando um componente essencial do simulador. A seguir, apresentamos os diretórios e arquivos principais:

- **Simulador/Simulador.cpp e Simulador.h**: Gerencia o ciclo principal da simulação, interações entre caravanas, itens e eventos.
- **Caravanas/Caravana.cpp e Caravana.h**: Define a classe base `Caravana` e suas subclasses (`CaravanaComercial`, `CaravanaMilitar`, etc.).
- **Itens/Item.cpp e Item.h**: Define os itens e seus efeitos no simulador.
- **Mapa/Mapa.cpp e Mapa.h**: Implementa a representação do mapa e suas funcionalidades, como verificar adjacências e atualizar posições.

- **Buffer/Buffer.cpp e Buffer.h**: Implementa o sistema de buffer para armazenar o estado visual do simulador.
- **Main.cpp**: Ponto de entrada do programa, responsável por carregar as configurações iniciais e iniciar o simulador.

4.1.2 Classe `Buffer`

A classe `Buffer` foi implementada para gerenciar a exibição do simulador, armazenando o estado visual em uma matriz dinâmica de caracteres. Ela permite que o estado do mapa seja atualizado a cada turno e possibilita salvar e carregar estados visuais para análise futura.

Estrutura da Classe

A classe `Buffer` utiliza um `std::unique_ptr<char[]>` para gerenciar a matriz bidimensional dinamicamente. O construtor inicializa o tamanho do buffer com base nas dimensões do mapa, e métodos adicionais foram implementados para manipular os dados.

Atributos Principais:

- `int linhas, colunas`: Dimensões do buffer.
- `std::unique_ptr<char[]> buffer`: Armazena o estado visual como um array unidimensional.

Métodos Principais:

- `void definirCelula(int linha, int coluna, char valor)`: Atualiza o valor de uma célula no buffer.
- `char obterCelula(int linha, int coluna) const`: Retorna o valor de uma célula específica.
- `void imprimir() const`: Imprime o estado atual do buffer.

Exemplo de Código

A seguir, apresentamos a implementação de alguns métodos da classe `Buffer`:

```
Buffer::Buffer(int linhas, int colunas)
    : linhas(linhas), colunas(colunas),
      buffer(std::make_unique<char[]>(linhas * colunas)) {
    std::fill(buffer.get(), buffer.get() + linhas * colunas, ' ');
}

void Buffer::definirCelula(int linha, int coluna, char valor) {
    buffer[linha * colunas + coluna] = valor;
}

char Buffer::obterCelula(int linha, int coluna) const {
    return buffer[linha * colunas + coluna];
}

void Buffer::imprimir() const {
    for (int i = 0; i < linhas; ++i) {
        for (int j = 0; j < colunas; ++j) {
            std::cout << buffer[i * colunas + j];
        }
        std::cout << std::endl;
    }
}
```

Funcionalidade de Salvar e Carregar Estados: O `Buffer` também permite salvar estados visuais para análise futura, integrando-se com o `Simulador`. Utilizamos um `std::map` para associar estados a nomes fornecidos pelo usuário, permitindo armazenar múltiplos snapshots do mapa.

—

4.1.3 Desafios Encontrados

Gerenciamento de Dependências Circulares

Durante o desenvolvimento, enfrentamos desafios relacionados a dependências circulares, especialmente entre as classes `Caravana`, `Item` e `Mapa`. Isso ocorreu porque algumas classes referenciavam diretamente outras em seus cabeçalhos, causando ciclos de inclusão.

Gerenciamento de Memória Dinâmica

Outro desafio foi garantir que todos os objetos alocados dinamicamente (como `Caravana` e `Item`) fossem liberados corretamente ao longo da simulação. Para resolver isso, utilizamos:

- `std::unique_ptr`: Gerenciou automaticamente a memória do `Buffer`.
- Liberação manual com `delete`: Para objetos armazenados em `std::vector<Caravana*>` e `std::vector<Item*>`.

Exemplo de liberação de memória no vetor de caravanas:

```
for (auto it = caravanas.begin(); it != caravanas.end(); ) {
    if ((*it)->estaSemTripulantes()) {
        delete *it; // Libera a memória da caravana
        it = caravanas.erase(it); // Remove do vetor
    } else {
        ++it;
    }
}
```

Capítulo 5

Conculsão

5.1 Conclusão

O desenvolvimento do simulador atendeu aos requisitos definidos inicialmente, permitindo modelar de forma eficiente as interações entre caravanas, itens e o mapa. A arquitetura baseada em Programação Orientada a Objetos provou ser adequada para o problema, possibilitando a reutilização de código e a expansão do sistema com novos tipos de entidades e funcionalidades.

Os testes realizados validaram as principais funcionalidades do simulador, incluindo:

- Movimentação autónoma e manual das caravanas, com comportamentos específicos para cada tipo;
- Interação dinâmica com itens no mapa, aplicando os efeitos definidos para cada tipo de item;
- Gestão de combates entre caravanas, com regras claras e resultados consistentes;
- Representação e manipulação do estado visual do simulador, utilizando a classe `Buffer` para salvar e carregar estados.

Apesar do sucesso na implementação, alguns desafios técnicos foram enfrentados, como o gerenciamento de dependências circulares e a liberação

correta de memória dinâmica. Esses problemas foram solucionados com o uso de boas práticas de programação, incluindo declarações antecipadas (*forward declarations*), o uso de `std::unique_ptr`, e o gerenciamento manual de memória quando necessário.

Os resultados obtidos mostraram que o simulador é funcional e estável, sendo capaz de representar eventos complexos como combates, interações com itens e eventos ambientais (ex.: tempestades).

Concluímos que o simulador atingiu seus objetivos principais, aplicando conceitos avançados de Programação Orientada a Objetos e algoritmos dinâmicos em C++.