Programação orientada a objetos- 2022/2023
ISEC – Instituto Superior de Engenharia de Coimbra

Trabalho prático

Professores:

João António Pereira Almeida Durães

António Augusto Nunes Godinho

Realizado por:

2021110042 - Jorge Ricardo Marques Duarte

2021146383 - João Alexandre Caravela Marques

Introdução

Este relatório tem o objetivo de apresentar o trabalho prático implementado em C++ que simula uma reserva natural e os animais que nela habitam.

O utilizador pode interagir com o simulador através de uma interface de modo texto, enviando comandos para alterar aspectos da simulação e dos animais. A interface deve mostrar a reserva e seu conteúdo, permitindo que o utilizador visualize os animais e suas características em detalhes.

Além disso, o programa permite que o utilizador desloque a área visível da reserva para ver outras partes. O relatório contém explicações das classes principais, estruturas de pastas, decisões tomadas e dificuldades e soluções encontradas no desenvolvimento deste trabalho prático.

Estrutura de pastas

Na pasta *docs,* temos todos os objetivos, dúvidas, relatórios e outros ficheiros relacionados com o trabalho prático.

Na pasta *commands*, temos implementada toda a lógica relacionada a comandos e validações.

Na pasta *models*, é onde temos todos os modelos utilizados pela aplicação, desde tipos de comida e animais a classes utilizadas como estrutura base para a interface visual.

Na pasta *services* é onde está presente a base da aplicação, sendo no ficheiro GameService.cpp que definimos qual o tamanho da nossa reserva natural, assim como a leitura do ficheiro de constantes e também a inicialização para a recepção dos comandos por parte do utilizador.

Na pasta *userInterface*, está contida a lógica de apresentação visual do nosso jogo, isto é, desde a informação sobre os animais, assim como a apresentação da reserva natural com todas as entidades existentes.

Por último, a pasta *utils* apenas contém funções utilitárias que podem ser reutilizadas por outras classes ou funções.

Classes principais da aplicação

GameService

```
class GameService {
private:
    Game game;
    void defineNaturalReserveConfiguration();
    void initializeMatrix();
    void readConstantsFile();
public:
    void initialize();
};
```

A classe *GameService* é responsável por inicializar o jogo, definir a configuração da reserva natural e inicializar a matriz de células da reserva. Ela possui uma variável do tipo Game, que é a classe principal do jogo, e um método privado chamado *readConstantsFile*, que é usado para ler os valores de configuração do arquivo de constantes. O método *initialize* é usado para inicializar o jogo, enquanto o método *defineNaturalReserveConfiguration* é usado para definir a configuração da reserva e o método *initializeMatrix* é usado para inicializar a matriz de células da reserva.

Game

```
class Game {
public:
    std::vector<std::vector<MatrixCell>> matrix = {};
    Node<Animal> *animals = NULL;
    Node<Food> *foods = NULL;
    GameConfiguration configuration;
    std::vector<SaveGame*> savedGames = {};

    int matrixGetNumRows();
    int matrixGetNumColumns();
    void addAnimal(Animal* animal);
    void addFood(Food* food);
    Node<Food>* findFoodNode(int id);
    Node<Animal>* findAnimalNode(int id);

    void addAnimalToTheMatrix(Animal* animal);
    void addAnimalToTheList(Animal* animal);
    void addFoodToTheMatrix(Food* food);
    void addFoodToTheList(Food* food);

    void displayAnimals();
    void displayFoods();
    std::vector<MatrixCell> getMatrixCellsByArea(int i, Position position);
};
```

A classe *Game* representa o jogo em si. Possui uma matriz de células da reserva, uma lista de animais e uma lista de alimentos presentes na reserva. Ela também possui uma variável do tipo *GameConfiguration*, que armazena as configurações do jogo, e um vetor de ponteiros para objetos do tipo SaveGame, que são usados para salvar o progresso

do jogo. A classe possui métodos para adicionar animais e alimentos à reserva, para encontrar nós de animais ou alimentos na lista pelo ID e para adicionar animais ou alimentos à matriz e às listas. A classe também possui métodos para exibir os animais e os alimentos presentes na reserva e para obter células da matriz em uma determinada área.

MatrixCell

```
class MatrixCell {
public:
    Position position = { col: 0, row: 0};
    Node<Animal> *animals = NULL;
    Node<Food> *foods = NULL;
    MatrixCell(int col, int line);
};
```

A classe *MatrixCell* representa uma célula da matriz da reserva. Possui uma posição na matriz, bem como uma lista de animais e uma lista de alimentos presentes nessa célula.

GameConfiguration

```
const static double DISPLAY_AREA_PERCENTAGE = 0.8;
const static double MAX_ARGUMENTS_SIZE = 5;

class GameConfiguration {
public:
    Position screenPosition = {  col: 0, row: 0 };
    MatrixSize size = {  h: 0 , w: 0 };
    void setMatrixSize(int height, int length);
    void displayMatrixSize();
    void moveScreenDisplayPosition(std::string direction, std::string target);
};
```

A classe *GameConfiguration* armazena as configurações do jogo, incluindo a posição atual do ecrã de visualização da reserva e o tamanho da matriz da reserva. Ela possui um método para definir o tamanho da matriz da reserva e um método para exibir o tamanho da matriz da reserva. Além disso, possui um método para mover a posição de visualização do ecrã da reserva para cima, para baixo, para a esquerda ou para a direita.

Animal

```
class Animal {
private:
    virtual void move(Game* game);
    virtual void move(Game* game, Position position);
    virtual void reproduce(Game* game);
    virtual void eat(Game* game, Food* food);
    virtual void fight(Game* game, Animal* animal);
    virtual void verifications(Game* game);
public:
    Animal();
    static int configMaxHP;
    static int configMaxIterations;
    int id;
    int maxHP;
    int currentHP;
    int currentIterations = 0;
    int maxIterations;
    int hunger = 0;;
    int reprodutionCounter = 0;
    char identifier;
    int progenitorId = -1;
    std::string identifierEmoji;
    Position position = { col: 0, row: 0};
    int weight;
    void feed(int nutritivePoints, int toxicityPoints);
    void display();
    void defineRandomPosition(int maxRow,int maxLine);
    void definePosition(int row, int line);
    void displayPosition();
    void setPosition(int row, int column);
    void setCurrentHP(int value);

    virtual void do_iteration(Game* game);
};
```

A classe *Animal* representa um animal presente na reserva. Possui um ID único, um identificador, uma posição na matriz da reserva, um peso, uma contagem de iterações atuais e máxima e uma contagem de reprodução. Além disso, tem uma contagem de fome e um valor de HP máximo e atual. A classe possui métodos para exibir os detalhes do animal, para alimentá-lo, para definir uma posição aleatória na matriz da reserva e para definir uma posição específica na matriz da reserva. Ela também possui métodos virtuais para mover o animal na reserva, para que ele se reproduza, para que ele coma um alimento específico, para que ele lute com outro animal e para fazer as verificações necessárias na iteração do animal. Assim como a classe Food, a classe Animal possui um método virtual chamado *do_iteration*, que é usado para fazer a iteração do animal na simulação e pode ser sobrescrito por subclasses da classe Animal.

Food

```
class Food {
private:
    virtual void reproduce(Game* game);
    virtual void verifications(Game* game);
public:
    static int configMaxIterations;
    int currentIterations = 0;
    int maxIterations;
    int id;
    char identifier;
    std::string identifierEmoji;
    Position position = { col: 0, row: 0};
    int nutritiveValue;
    int toxicity;
    std::list<int> smells;
    Food();
    void display();
    void defineRandomPosition(int maxRow,int maxLine);
    void definePosition(int row, int line);
    virtual void do_iteration(Game* game);

};
```

A classe *Food* representa um alimento presente na reserva. Possui uma ID única, um identificador, uma posição na matriz da reserva, um valor nutritivo e um valor de toxicidade. Além disso, tem uma lista de odores e uma contagem de iterações atuais e máxima, que determinam quanto tempo o alimento vai permanecer na reserva antes de desaparecer. A classe possui métodos para exibir os detalhes do alimento, para definir uma posição aleatória na matriz da reserva e para definir uma posição específica na matriz da reserva. Também possui um método virtual chamado *do_iteration*, que é usado para fazer a iteração do alimento na simulação e pode ser sobrescrito por subclasses da classe *Food*.

Decisões

Animal mistério

A entidade por definir escolhida como animal mistério foi a raposa. Este animal tem as seguintes características:
Visualiza alimentos do tipo vegetal que neste caso é o alimento mistério amora.
A raposa visualiza animais e comida até 6 posições de distância em área.
A velocidade da raposa é de 3 posições sempre e em caso de encontrar outro animal, fica com medo e tenta fugir dele.
A cada instância é acrescentada um valor de fome, a não ser que a raposa ingira uma amora, redefinindo o valor da fome para zero.
A sua reprodução é uma chance de 40% a cada 15 instâncias de jogo. Se a raposa se conseguir reproduzir, surgira uma nova raposa até 10 posições de distancia.

Comida mistério

Para a comida mistério escolhemos a amora, esta comida só é percepcionada pelas raposas, sendo a raposa o animal mistério.
O seu valor nutritivo é de 15 valores, não tem toxicidade, é identificada pela letra ‘a’, e o seu cheiro é vegetal.

Matriz bidimensional de ponteiros

De forma a apresentar todos os animais na matriz, decidimos que a estrutura do jogo seria uma matriz bidimensional de *MatrixCells*, em que cada célula tem uma lista de ponteiros para animais e outra para comidas. Sabemos que apenas é permitida 1 comida por cada célula e o programa, neste momento, tem as validações necessárias para esta situação. Apenas é uma lista porque achamos o projeto bastante interessante e que seria mais engraçado permitir no futuro este tipo de acontecimento. Sendo que cada célula é uma lista de ponteiros, o movimento dos animais na reserva é efetuado através da remoção do ponteiro de uma dada posição e a adição deste ponteiro noutra célula.

Bolsa marsupial do Canguru

Para a implementação da bolsa marsupial do canguru, tendo em conta que o animal enquanto se encontra dentro da bolsa do progenitor não é visível na matriz, este é removido apenas da matriz, ou seja o seu ponteiro é apagado e desta forma não é visível para os outros animais. O controlo das iterações em que este pode ficar dentro da bolsa é feito através da propriedade *onMarsupial* e *onMarsupialInstants*, controlando assim os limites em que este pode ficar escondido.

Comando slide

O comando slide foi implementado sendo possível a visualização de 80% da reserva do jogo, isto significa que podemos andar para cima, lado e baixo na reserva. A área não visível é apresentada com cardinais(#), dando a perceção que não estamos a ver o que está a acontecer durante o jogo naquela dada área.

Biblioteca ncurses

Uma das decisões que tivemos de tomar durante o desenvolvimento deste projeto acabou por ser não utilizar a biblioteca ncurses. Dado que já tínhamos bastante código implementado na primeira meta, decidimos não utilizar a biblioteca sendo que iria ser necessário refazer muito código associado a apresentação das respostas aos comandos e interfaces visuais no projeto.

Gestão de comandos

No programa, precisamos de gerir os comandos a serem executados através da interface visual. Esta necessidade levou à criação da classe Command. A classe *Command* é uma classe que contém métodos estáticos e uma lista de comandos permitidos.

Esta classe verifica se os comandos inseridos são válidos ou não e após estas verificações é chamado o método *handleCommands*, que é responsável por chamar a classe correta para lidar com o comando específico.

O factory pattern é implementado no método *handleCommands*, que cria uma instância da classe correta para lidar com o comando. A classe *DefaultCommand* é uma classe base para todos os comandos específicos que estendem dela, e essas classes específicas sobrescrevem o método *execute* para realizar ações específicas para cada comando. Por exemplo, a classe *AnimalCommand* estende DefaultCommand e possui um método *execute* que adiciona animais à reserva natural.

Aplicação de testes

A criação de vários ficheiros para simular diferentes ambientes na reserva natural foi uma escolha muito útil durante o desenvolvimento e teste da aplicação. Isso permitiu simular diferentes cenários e casos de uso de forma rápida e precisa, sem precisar adicionar manualmente os animais e alimentos todas as vezes que inicializamos a aplicação.

Estes ficheiros estão contidos dentro da pasta resources da aplicação e foram utilizados para as várias simulações e testes durante a criação do programa.

Histórico de controlo dos requisitos implementados

Meta 1 – 27 de novembro (100%)

- Requisitos - funcionalidades para a meta 1:
    - Leitura de ficheiros
    - Ficheiro de comandos
    - Ficheiro de constantes.txt
    - Construção da reserva.
    - A representação da reserva irá ser melhorada na meta 2.
- Definição do conceito de Animal.
    - Não é preciso considerar as variações inerentes às espécies
    - imitar-se aquilo que é genérico e comum a todos.
- Definição do conceito de Alimento.
- Representação visual da reserva e conteúdo incluído nesta meta.
- Inclui-se aqui a questão de ver apenas a área visível da reserva.
- Implementação da leitura e validação de todos os comandos, seja por teclado, ou seja por leitura do ficheiro de comandos.
- Base projeto para leitura e validação dos comandos
    - Os comandos não farão ainda nada, mas devem ser já interpretados e validados
    - incluindo a sintaxe de todos os parâmetros que tenham.
    - Implementar os comandos para:
        - anim
        - visanim
        - info
        - deslizar a área visível para o lado/cima/abaixo
        - executar comandos em ficheiro (que são também validados), e terminar.
- projeto já deverá estar devidamente organizado em .h e .cpp

Meta 2 – 08 de Janeiro

- Implementação de todos os comandos básicos
    - exec_command_nofood
    - exec_command_feed
    - exec_command_feedid
    - exec_command_see
    - Exec_command_killid
    - Exec_command_kill
    - Exec_command_empty
    - Exec_command_animal
    - Exec_command_food
- Remover todos os animais fictícios da inicialização do código
- Criação de arquivos de teste de comando com diferentes ambientes
- Adição possibilidade de vários animais na mesma coluna de linha
- Modificações na classe alimento e animal
- Adição de suporte para todos os tipos S e V no ficheiro de constantes.txt
- Adição de memória de jogo através de comandos store e restore.
- Verificação global do projeto
- Adição de validação para números inteiros na configuração dos tamanhos da reserva
- Adição de validações nos comandos que pendem linha / coluna se são inputs válidos dado o tamanho configurado da reserva.
- Melhorias no comando slide
- Alteração da visualização da área visível
- Verificação das propriedades da classe Animal e Alimento e alterações sobre as mesmas.
- Reunião de decisão das principais funções necessárias para as interações dos animais
- Implementação do comando da iteração , ou seja, comando execute_command_n (n).
- Implementação de cada animal e os seus comportamentos:
    - Raposa
    - Lobo
    - Canguru
    - Coelho
    - Ovelha
- Implementação das principais funções necessárias para todas as interações dos alimentos
    - Cenoura
    - Carne
    - Mirtilos
    - Corpo
    - Relva
- Última verificação global do projeto
- Adição de limitações nos alimentos para ser apenas um permitido por posição

Dificuldades e resoluções

Uma dificuldade encontrada durante o desenvolvimento deste trabalho foi perceber que a matriz que representa a reserva natural deveria ser tratada como um torus, ou seja, quando um animal saía por um lado da matriz, ele surgia do outro lado. Isso exigiu a implementação de verificações e correções de posição específicas para garantir que os animais se movimentam corretamente na matriz e não saíssem da reserva. No entanto, esta dificuldade tornou-se um desafio interessante e gratificante de resolver. Ao perceber a necessidade de tratar a matriz como um torus, foi possível implementar uma lógica específica para garantir que os animais se movimentam corretamente e continuam dentro da reserva. Isto exigiu um pensamento lógico aprofundado e a implementação de código bem estruturado, o que foi uma ótima aprendizagem. Além disto, a resolução deste problema contribuiu significativamente para o funcionamento correto da simulação e para a experiência do utilizador final. Portanto, apesar de inicialmente ser uma dificuldade, tratar a matriz como um torus acabou sendo um desafio gratificante e uma ótima oportunidade de aprendizagem.

Conclusão

Ao concluir este trabalho prático de programação orientada a objetos, é possível ver como a abordagem de orientação a objetos foi útil para organizar o código e garantir a reutilização de código e a coesão das classes. A criação de classes com responsabilidades bem definidas e a utilização de herança e polimorfismo permitiram a implementação de um simulador de reserva natural de forma clara e modular. Além disso, a implementação de métodos virtuais permitiu a criação de comportamentos genéricos e a personalização desses comportamentos em subclasses específicas.

Outras vantagens da programação orientada a objetos incluem o encapsulamento de dados, que protege os atributos das classes de acesso indevido, e a abstração de funcionalidades, que permite ocultar detalhes de implementação das classes e fornecer uma interface simplificada para o utilizador. Ao aplicar esses conceitos no trabalho prático, foi possível aprender a criar uma estrutura de código mais organizada e legível, o que foi fundamental para garantir a qualidade e a eficiência do programa. Em resumo, a utilização de programação orientada a objetos foi fundamental para o sucesso deste trabalho prático e trouxe muitas vantagens e aprendizagens valiosas durante o seu desenvolvimento.