

INSTITUTO SUPERIOR DE ENGENHARIA DE COIMBRA

INSTITUTO POLITÉCNICO DE COIMBRA

Licenciatura em Engenharia Informática 2º Ano – 1º
Semestre 2021/2022

Natural Reserve Simulator

Rafael Couto Nº 2019142454

Rafaela Carvalho Nº 2019127935

COIMBRA

7 de janeiro de 2023

Índice

Introdução	5
Implementação	6
Estruturas de dados	6
• Vetor de Células.....	6
• Vetor de Reservas	7
• Vetor de Animais.....	7
Classes	8
• Interface	8
initInterface()	8
showMatrix()	8
commandReader()	9
increaseSimulatedTime()	9
• Reserva.....	9
getId()	10
getSimulatedTime()	10
getCellInfo().....	10
addHistory()	10
animalActions()	10
foodActions().....	11
• Célula	11

getInfo()	11
removeAllEntities()	12
getAnimals().....	12
getFood()	12
removeAnimal().....	12
copyNewAnimal()	12
adicionaRelva()	13
• Animais	13
addFoodHistory().....	13
feed().....	13
getInfo()	14
getMotherID()	14
getBirthInstant()	14
• Alimentos	14
getCheiro()	15
operator= (Alimentos *alimentos)	15
getDuracao()	15
getValorNutritivo()	15
getToxicidade()	15
Comandos	16
Animal.....	16

kill / killid	16
food	16
feed / feedid	17
nofood.....	17
empty	17
see.....	17
info.....	18
n	18
anim	18
visanim.....	18
store.....	19
restore	19
load.....	19
slide	19
clear	19
Decisões tomadas.....	20
Conclusão	21
Anexos.....	22

Introdução

A elaboração deste trabalho prático visa consolidar conhecimentos em linguagem C++, explorados nas aulas teóricas e práticas, criando capacidade de desenvolvimento de aplicações nesta linguagem de programação.

Pretende-se criar um simulador, denominado *Natural Reserve Simulator*, destinado a gestão e desenvolvimento de uma reserva natural. Ao jogador será atribuída uma reserva que o mesmo deve gerir, provando-a e alimentando todos os seres vivos.


Deste modo, será implementada uma classe geral *Reserva* responsável por suportar todo o tipo de dados relativos a cada *célula* e à sua pormenorização.

Ora, cada *Reserva* tem *nLinhas* por *nColunas* e cada unidade será uma *cell* que é descrita por *animais* e *alimento*.

A interação com o jogo processa-se através de comando e ação por parte do jogador.

```

cmake-build-debug — TPPO02223 — 108x38
[*] Welcome to Natural Reserve Simulator!



Simulated Instant: 0

(0,0) |-----| (0,5)
| C: (0,0) | C: (0,1) | C: (0,2) | C: (0,3) | C: (0,4) |
| A: Empty | A: Empty | A: Empty | A: Empty | A: Empty |
| F: Empty | F: Empty | F: Empty | F: Empty | F: Empty |
|-----|
| C: (1,0) | C: (1,1) | C: (1,2) | C: (1,3) | C: (1,4) |
| A: Empty | A: Empty | A: Empty | A: Empty | A: Empty |
| F: Empty | F: Empty | F: Empty | F: Empty | F: Empty |
|-----|
| C: (2,0) | C: (2,1) | C: (2,2) | C: (2,3) | C: (2,4) |
| A: Empty | A: Empty | A: Empty | A: Empty | A: Empty |
| F: Empty | F: Empty | F: Empty | F: Empty | F: Empty |
|-----|
| C: (3,0) | C: (3,1) | C: (3,2) | C: (3,3) | C: (3,4) |
| A: Empty | A: Empty | A: Empty | A: Empty | A: Empty |
| F: Empty | F: Empty | F: Empty | F: Empty | F: Empty |
|-----|
| C: (4,0) | C: (4,1) | C: (4,2) | C: (4,3) | C: (4,4) |
| A: Empty | A: Empty | A: Empty | A: Empty | A: Empty |
| F: Empty | F: Empty | F: Empty | F: Empty | F: Empty |
|-----|
(5,0) |-----| (5,5)

[-] COMMAND: █

```

Implementação

No jogo *Natural Reserve Simulator* existem várias maneiras de interação, desde colocar alimentos aleatoriamente ou especificamente, bem como *animais*.

Permite fazer toda a gestão de mapa e a cada instante vão-se desempenhando ações derivadas das características de cada *animal* e *alimento*

Estruturas de dados

- Vetor de Células

A estrutura de dados escolhida para armazenar os objetos da classe *Célula* foi um *vetor* bidimensional. Este vetor pertence à classe *Reserva* e para ser criado foi utilizado um vetor dentro de outro vetor para que seja possível aceder aos objetos utilizando a notação de coordenada (vetor[x][y]).

```
vector< vector<Cell*>> area;
```

Figura 1 – Vetor bidimensional de células

De seguida foi necessário criar um vetor temporário para armazenar objetos da mesma coluna nesse vetor para posteriormente ser adicionado o vetor temporário ao vetor principal.

Este vetor é criado no construtor da classe *Reserva* quando um objeto deste tipo é criado.

- Vetor de Reservas

O vetor de *Reservas* é responsável por guardar os vários estados de jogo existentes à medida que os instantes de jogo vão avançando. São guardadas novas instâncias de *Reservas* sempre que o utilizador executa o comando *store*.

```
vector<Reserva *> history;
```

Figura 2 - Vetor de Reservas

- Vetor de Animais

De modo que fosse possível armazenar os animais de uma célula criou-se um vetor de animais para que fosse possível armazenar vários objetos do tipo *Animal* na mesma célula.

```
vector<Animais*> animais;
```

Figura 3 - Vetor de Animais

Classes

Cada classe criada é representativa de um domínio do jogo. Os objetos de classes que contém múltiplas informações sobre esse domínio e funções que nos permitem alterar e ou obter várias informações à cerca de um objeto.

- Interface

A classe *Interface* é a classe que suporta toda a lógica de jogo e transpõe para a parte visual.

- Funções:

- `initInterface()`
- `showMatrix()`
- `commandReader()`
- `increaseSimulatedTime()`

`initInterface()`

A função *initInterface()* é responsável por inicializar toda a interface e é quem lança o construtor da classe *Reserva* com os valores necessários.

`showMatrix()`

A função *showMatrix()* é responsável por representar toda a área da classe *Reserva* na janela de visualização.

commandReader()

A função *commandReader()* é responsável por reconhecer e validar todos os comandos para o bom funcionamento do jogo, bem como aplicar comandos vindos de ficheiro .txt externo.

increaseSimulatedTime()

A função *increaseSimulatedTime()* é responsável por incrementar a atual instância de jogo e desempenhar todas as ações e verificações necessárias a cada execução de tempo do nosso jogo.

- Reserva

A classe *Reserva* é a classe principal do jogo. Representa um mapa com várias células e também informações de animais e alimentos.

- Funções:
 - `getId()`
 - `getSimulatedTime()`
 - `getCellInfo()`
 - `addHistory()`
 - `animalActions()`
 - `foodActions()`

getId()

A função *getId()* é responsável por retornar o número de ID incrementado, permitindo aplicar à criação dos elementos da reserva tornando assim cada elemento com um ID único.

getSimulatedTime()

A função *getSimulatedTime()* é responsável por retornar o instante de simulação e é o que sustenta toda a evolução do jogo.

getCellInfo()

A função *getCellInfo()* é responsável por devolver em formato *string* a informação de uma célula de coordenadas X e Y.

addHistory()

A função *addHistory()* é responsável por adicionar ao vetor *history* um objeto do tipo *Reserva*. É neste vetor que ficam armazenados os vários estados de jogo guardados pelo utilizador através do comando *store <name>*.

animalActions()

A função *animalActions()* é responsável por todo o mecanismo de movimentos e ingestão de comida por parte dos animais, bem como outras questões de implementação mencionados no enunciado do trabalho.

foodActions()

A função *foodActions()* é onde todas as ações possíveis de acontecerem a um alimento se desenrolam, desde gerar novos alimentos a remover alimentos que esgotem a sua duração.

- Célula

A classe *Célula* representa uma coordenada no mapa e pode possuir animais e um alimento.

- Funções:

- *getInfo()*
- *removeAllEntities()*
- *getAnimals()*
- *getFood()*
- *removeAnimal()*
- *copyNewAnimal()*
- *removeFood()*
- *adicionaRelva()*

getInfo()

A função *getInfo()* retorna a informação da célula e é usado para representar toda a informação dos elementos presentes na célula. Apresenta informação de coordenadas, animais e alimento existentes. É usada por exemplo com recurso ao comando *see <X><Y>*.

[removeAllEntities\(\)](#)

A função *removeAllEntities()* serve, como o nome indica, para remover todas as entidades, neste caso, remove todos os elementos presentes na célula. É a função que está associada ao comando *empty <X><Y>*.

[getAnimals\(\)](#)

A função *getAnimals()* retorna uma *string* com a informação dos animais presentes na célula.

[getFood\(\)](#)

A função *getFood()* retorna uma *string* com a informação do alimento presente na célula.

[removeAnimal\(\)](#)

A função *removeAnimal()* como o próprio nome indica, é responsável por eliminar um animal em específico da célula em questão. Para o fazer usa como recurso o *ID* do animal. É usado maioritariamente no movimento dos animais e em caso de falecimento.

[copyNewAnimal\(\)](#)

A função *copyNewAnimal()* é responsável por copiar o um objeto da classe *Animal* para outra célula. É usado maioritariamente no movimento dos animais.

adicionaRelva()

A função `adicionaRelva()` é responsável por gerar um novo alimento do tipo *Relva*. É usado para assegurar a condição descrita no enunciado do trabalho.

- Animais

A classe *Animais* representa um animal que pode estar posicionado numa determinada célula da reserva.

- Funções:
 - `addFoodHistory()`
 - `feed()`
 - `getInfo()`
 - `getMotherID()`
 - `getBirthInstant()`

addFoodHistory()

A função `addFoodHistory()` é responsável por tratar do histórico de alimentos. Mais a baixo neste documento, na secção *Decisões Tomadas*, é descrito mais em detalhe a implementação.

feed()

A função `feed()` é responsável por atribuir ao *Animal* os atributos específicos de cada alimento e consequentemente incrementar ou decrementar características do *Animal* como saúde e fome.

getInfo()

A função *getInfo()* retorna a informação detalhada do *Animal* e é usada por exemplo aquando da chamada do comando *see <ID>*.

getMotherID()

A função *getMotherID()* é responsável por retornar o ID do *Animal* mãe que criou o *Canguru*. Trata-se, portanto, de uma função específica da classe derivada *Canguru*. É usada para confirmar se o *Canguru* tem mãe ou não e assim desempenhar diferentes movimentos.

getBirthInstant()

A função *getBirthInstant()* é responsável por retornar o instante em que o *Animal* foi criado e permite fazer cálculos para decrementar as suas características bem como saber quando é que o mesmo falece.

- Alimentos

A classe *Alimentos* representa um alimento que pode ser posicionado em qualquer célula da Reserva.

- Funções:

- *getCheiro()*
- *operator= (Alimentos *alimentos)*
- *getDuracao()*
- *getValorNutritivo()*

- `getToxicidade()`

`getCheiro()`

A função `getCheiro()` é o que nos permite identificar qual é o cheiro do *Alimento* em questão e consequentemente delimitar a busca de cada *Animal* pelo tipo de *Alimento* que deseja consumir.

`operator= (Alimentos *alimentos)`

A função `operator=` foi criada essencialmente como construtor de cópia para servir de auxílio à função `addFoodHistory()` da classe *Animal*, dada a questão especial de implementação.

`getDuracao()`

A função `getDuração()` retorna a duração do *Alimento* de modo a desenrolar as ações possíveis relativas aos *Alimentos*.

`getValorNutritivo()`

A função `getValorNutritivo()` retorna o valor nutritivo do *Alimento* e permite o desenrolar das ações possíveis relativas aos *Alimentos*.

`getToxicidade()`

A função `getToxicidade()` retorna o valor nutritivo do *Alimento* e permite o desenrolar das ações possíveis relativas aos *Alimentos*.

Comandos

Os comandos são utilizados pelo jogador para desempenhar uma série de ações de modo a efetuar operações como criar animais e alimentos.

Animal

Comando “*animal*” – permite criar um *Animal* na *Reserva* através da seguinte formatação:

- animal <especie: c / o / l / g / m> <linha> <coluna>
- animal <especie: c / o / l / g / m> [Posição aleatória]

kill / killid

Comando “*kill ou killid*” – permite matar um *Animal* na *Reserva* através da seguinte formatação:

- kill <linha> <coluna>
- killid <id>

food

Comando “*food*” – permite colocar um *Alimento* na *Reserva* através da seguinte formatação:

- food <tipo: r / t / b / a> <linha> <coluna>
- food <tipo: r / t / b / a> [Posição aleatória]

feed / feedid

Comando “*feed ou feedid*” – permite alimentar diretamente um *Animal* da *Reserva* através da seguinte formatação:

- feed <linha> <coluna> <pontos nutritivos> <pontos de toxicidade>
- feedid <ID> <pontos nutritivos> <pontos de toxicidade>

nofood

Comando “*nofood*” – permite remover um *Alimento* da *Reserva* através da seguinte formatação:

- nofood <linha> <coluna>
- nofood <ID>

empty

Comando “*empty*” – permite o que quer que esteja numa posição da *Reserva* através da seguinte formatação:

- empty <linha> <coluna>

see

Comando “*see*” – permite ver toda a informação de uma posição da *Reserva* através da seguinte formatação:

- see <linha> <coluna>

info

Comando “*info*” – permite ver a informação de um elemento da *Reserva* através da seguinte formatação:

- info <ID>

n

Comando “*n*” – permite avançar instâncias da *Reserva* através da seguinte formatação:

- n [Avança apenas uma instância]
- n <N> [Avança N instâncias com intervalos de 1 segundo]
- n <N><P> [Avança N instâncias com intervalos de P segundos]

anim

Comando “*anim*” – permite ver a informação de todos os *Animais* da *Reserva* através da seguinte formatação:

- anim

visanim

Comando “*visanim*” – permite ver a informação de todos os *Animais* da *Reserva* através da seguinte formatação:

- visanim

store

Comando “*store*” – permite guardar os estados da *Reserva* no seu vetor de histórico através da seguinte formatação:

- store <nome>

restore

Comando “*restore*” – permite recuperar um estado da *Reserva* guardado através de um nome no seu vetor de histórico através da seguinte formatação:

- restore <nome>

load

Comando “*load*” – permite aplicar comandos através de um ficheiro .txt externo aplicando na *Reserva* através da seguinte formatação:

- load <nome-do-ficheiro>

slide

Comando “*slide*” – permite deslizar pela *Reserva* seguindo uma analogia de “janela deslizante” através da seguinte formatação:

- slide <direção: up/down/left/right> <nLinhas/nColunas>

clear

Comando “*clear*” – permite limpar a janela através da seguinte formatação:

- clear

Decisões tomadas

Em relação à parte que visível no ecrã tomamos a decisão de mostrar um tamanho fixo, definido na classe *Reserva* com o nome *viewWindow*.

De modo que esta *viewWindow* seja independente de toda a visualização do tabuleiro de jogo foi criado uma variável *topLeftCorner* que permitirá fazer o cálculo de apresentação das células no intervalo desejado.

Tal como descrito no enunciado o número de linhas e colunas do tabuleiro de jogo é pedido ao utilizador em *runtime* no início da execução do programa.

Foi decidido que o jogo iniciaria sempre a sua área visual no canto superior esquerdo (0,0) e daí em diante é controlado através do utilizador.

Tomamos por iniciativa mais uma vez criar um comando clear, comando que irá ser destinado para limpar tudo o que já foi escrito no terminal para que se possa continuar a usar o simulador sem falta de informação por falta de espaço no terminal.

Como descrito no enunciado do trabalho, não era possível o uso de coleções da *Standard Library* para o armazenamento do histórico de Alimentos ingeridos por um Animal e para tal foi usado um ponteiro do tipo classe *Alimento* que é sempre inicializado a *nullptr* e que cada vez que é necessário adicionar um objeto a este histórico se faz um *realloc* de memória.

Conclusão

Em termos de conclusão, é de salientar a importância deste tipo de trabalhos para o desenvolvimento de capacidades de programação e raciocínio. É deste modo também que foi possível consolidar os conceitos abordados em sala de aula e melhor conhecer a linguagem de C++.

Um dos grandes fatores que torna este trabalho interessante é o facto de ser um jogo interativo e que nos coloca à prova dos nossos conhecimentos e explorar para além do conhecido.

De igual modo o paradigma de programação orientada a objetos é deveras interessante e bastante diferente das linguagens conhecidas até agora.

Anexos

- Reserva.h
- Reserva.cpp
- Cell.h
- Cell.cpp
- Animais.h
- Animais.cpp
- Alimentos.h
- Alimentos.cpp
- Interface.h
- Interface.cpp
- Utils.h
- main.cpp
- constantes.txt
- commands.txt
- POO – 2223 - Relatório.pdf