

Programação Orientada a Objetos

ISEC – Instituto Superior de Engenharia de Coimbra

**Trabalho prático – Meta 2**

2021110042 - Jorge Ricardo Marques Duarte

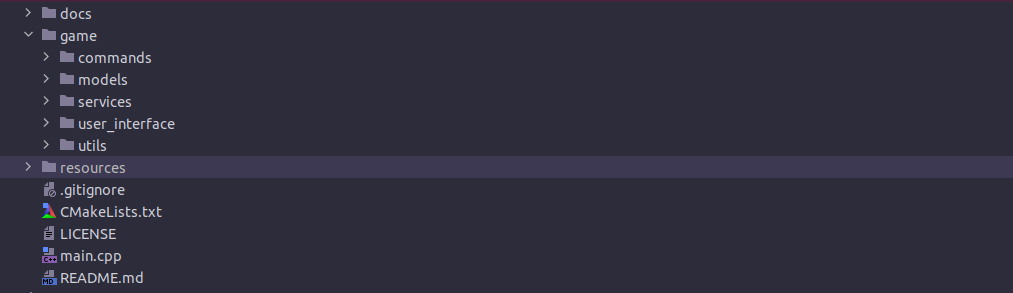
2021146383 - João Alexandre Caravela Marques

**Introdução**

Neste trabalho prático pretende-se aplicar os conhecimentos adquiridos durante as aulas teóricas e práticas da disciplina construindo em C++ um simulador de uma reserva natural povoada por diversos animais.

Este relatório vem com o intuito de demonstrar as diversas decisões tomadas ao longo do desenvolvimento, tanto a nível de estruturas usadas, assim como organização e estrutura da aplicação.

**Estrutura de pastas**

 Na pasta *docs,* temos todos os documentos do projeto, ou seja, todos os objetivos, perguntas e relatórios que foram realizados durante o desenvolvimento.

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

Na pasta *docs*, temos documentos e objetivos do projeto, estando estes seperados por metas.

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 contemos a base da nossa 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 receção dos comandos por parte do utilizador.

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

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

**Classes principais da aplicação**

Como descrito na nossa estruturas de pastas, a nossa pasta principal é a services, contendo toda a lógica principal da aplicação. Nesta pasta contemos a classe GameService contendo classe Game e funções principais privadas para definição de critérios do jogo.

Observe a imagem seguinte referente a classe GameService:



A estrutura Game como podemos visualizar está presente dentro do nosso GameService. Esta estrutura contem todas as informações referentes a reserva natural, como a matriz sendo um vetor de vetores representando a nossa reserva natural, uma lista da classe Animal, outra lista da classe Food, uma classe referente a configurações do jogo e uma lista de estados de jogo a serem guardados em memória.

Observe a imagem seguinte referente a classe Game:



Outra classe importante que contemos na nossa aplicação é a classe MatrixCell que contem os ponteiros de animais ou comidas na respetiva célula de jogo.

O objetivo das listas de ponteiros de animais e comidas é sabermos em que posição de encontra o animal ou comida no tabuleiro. Sendo ponteiro facilita a alteração dos dados do mesmo, sendo que podemos ir diretamente mudar os valores destas entidades.

Isto acaba por trazer uma enorme vantagem a nível de gestão do jogo. Da mesma forma podemos alterar um animal diretamente na lista de animais e sabemos que este será alterado também dentro da célula de jogo.

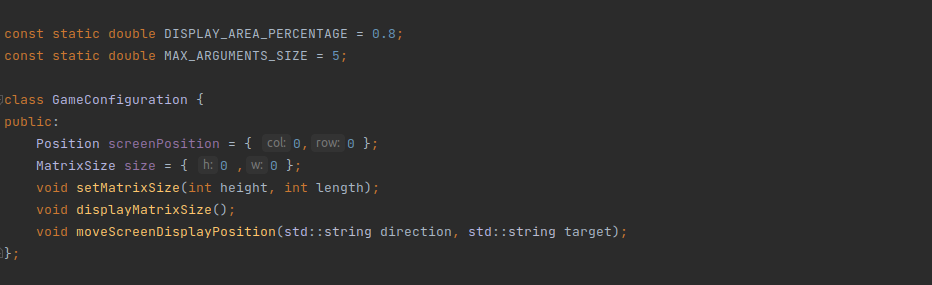
A movimentação dos animais na matriz é feita removendo o ponteiro do animal na célula atual, e adicionando na posição desejada.

Observe a imagem seguinte referente a classe MatrixCell:



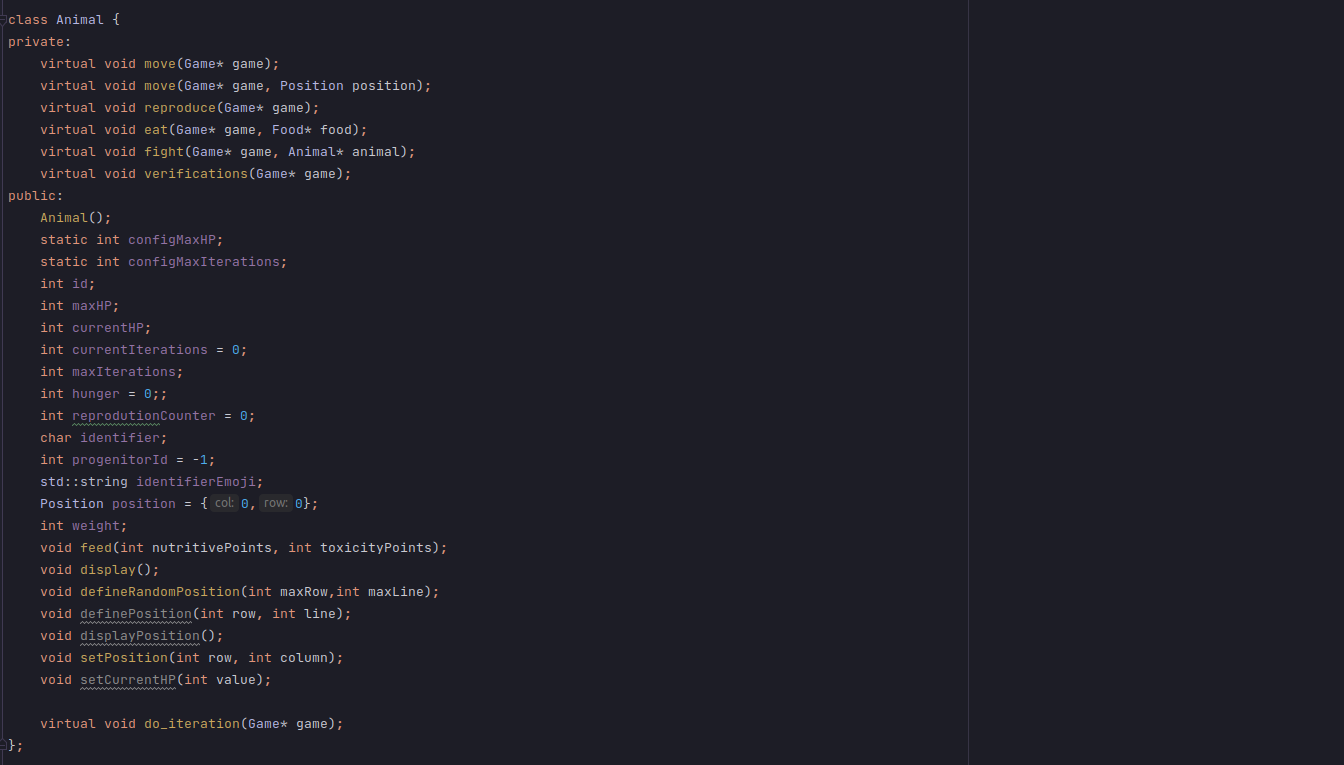
Ainda dentro do Game, podemos observar a classe GameConfiguration, sendo a classe responsável por tudo que tenha a ver com configurações do nosso jogo. A sua estrutura contem o tamanho da nossa reserva natural, assim como uma posição de visualização do ecrã. Definimos como limite de área visível 80% do tamanho do jogo quando criado numa variável global constante. Nesta classe é possível modificar a área de jogo visível através da função moveScreenDisplayPosition.

Observe a imagem seguinte referente a classe GameConfiguration:



Para a criação de cada animal foi necessário criar uma classe com funções e propriedades base para todas as que a fossem herdar.

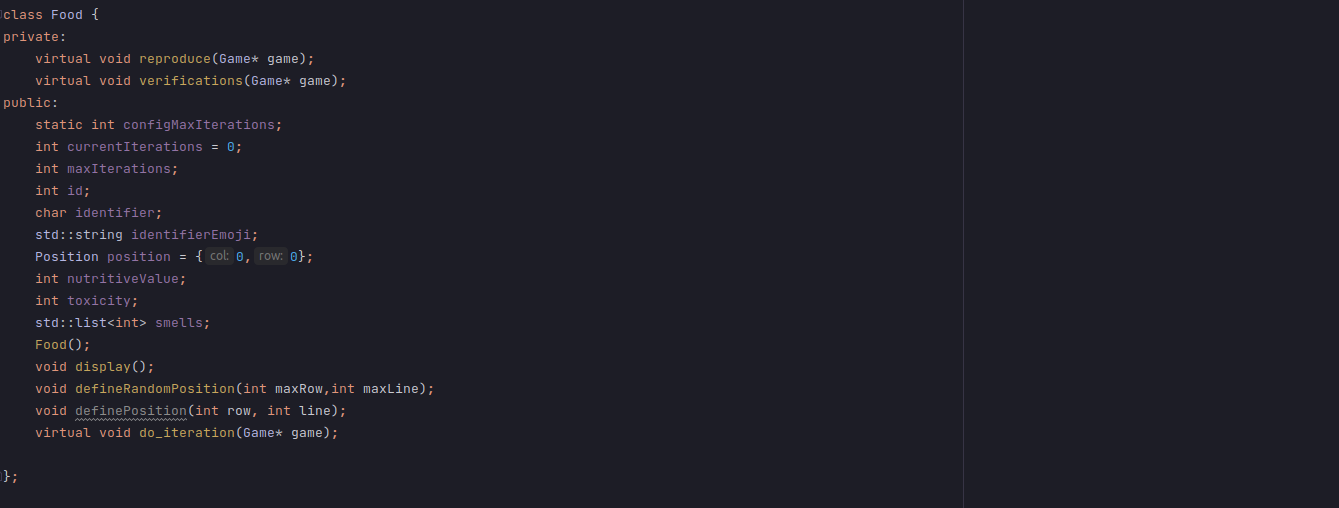
Com isto surgiu a classe Animal, observe a seguinte imagem:

Como podemos observar, esta classe tem propriedades como contadores de reprodução, identificadores, id do progenitor, posição, entre outras características.

Além disto, definimos funções base para serem reescritas pelas que a herdassem. Tentamos ter estas funções virtuais totalmente puras, mas sendo que estávamos a utilizar a classe Animal sem ponteiros em outras situações no código, acabamos por não refazer o necessário para que isto fosse possível. O id do animal é atribuído no construtor do respetivo animal, em que é gerado o novo id através da classe Entity, que é uma classe com a variável static id representando assim efetivamente um único identificador para cada animal ou comida.

Para a criação de cada comida foi necessário a criação também de uma classe Food, contendo várias propriedades utilizadas por todos os tipos de comida. Estas propriedades são o valor nutricional, toxicidade, duração de vida, entre outras.

Observe a classe Food na imagem que se segue:



**Decisões tomadas**

**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 distancia 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 coma uma amora, redefinindo o valor de 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 percecionada pelas raposas, sendo a raposa o animal mistério.

O seu valor nutritivo é de 15 valores, não tem toxidade, 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 permitido 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, 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**

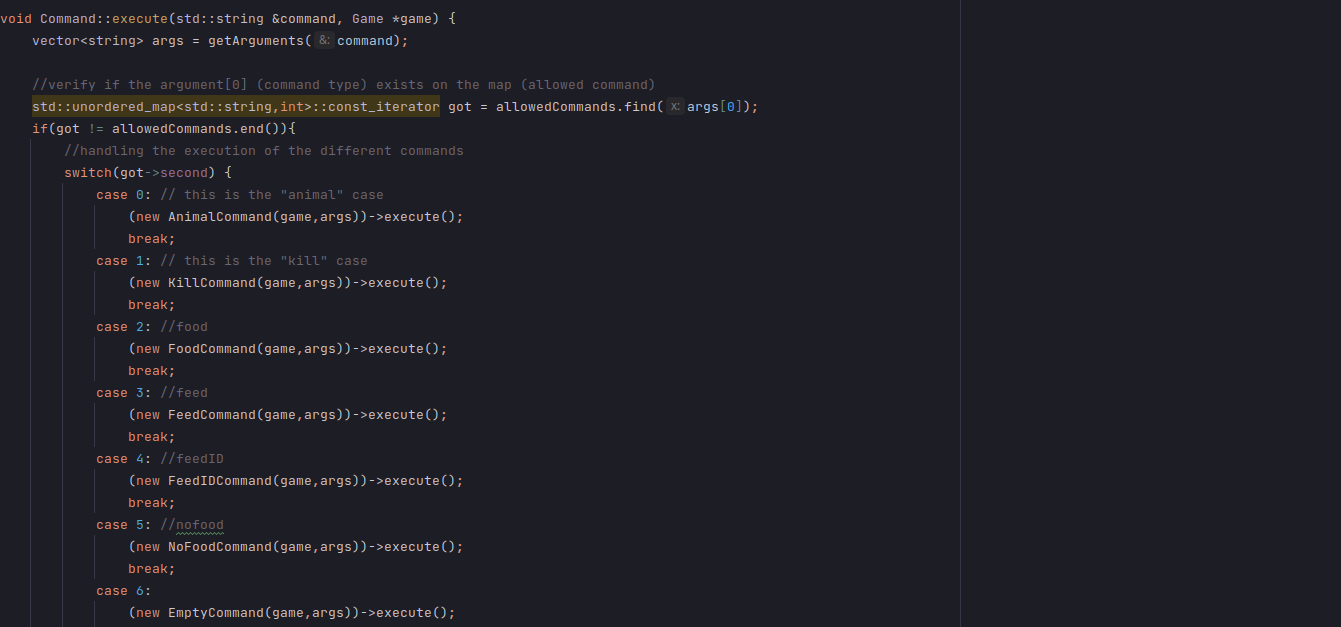
Uma das decisões que tivemos de tomar durante o desenvolvimento deste projeto acabou por ser não utilizar a biblioteca ncurses. Dado que já tínhamos bastante código implementado na primeira meta, decidimos não utilizar a biblioteca sendo que iria ser necessário refazer muito código associado a apresentação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

**Gestão de comandos**

Na primeira meta, tinhamos todos os comandos no mesmo ficheiro cpp, entretanto decidimos utilizar o Factory pattern, e temos a criação de cada comando especifico a sua implementação. Isto acabou por tornar o código mais organizado e estruturado.

A invocação dos comandos é feito na classe Command, sendo esta a classe responsável por validar os argumentos e alguns parâmetros essenciais para a invocação da classe desejada. A função execute, trata de saber qual o comando a ser executado, passando assim os dados importantes para o comando.

Observe a função execute na imagem seguinte:

Como podemos observar neste excerto de código, cada comando é invocado se este existir uma lista de comandos permitidos.

Cada comando especifico estende da classe DefaultCommand, que contem algumas funções necessárias a serem implementadas.

**Testes aplicados durante o desenvolvimento**

Durante todo o desenvolvimento do projeto, achamos necessário realizar testes e criar ambientes base para verificar se as implementações estavam de acordo com os requisitos. Sendo que cada vez que era necessário criar um ambiente executar uma série de comandos, criamos vários ficheiros de cenários de exemplo, que se tornaram bastante úteis com a utilização do comando load.

Isto tornou o processo de desenvolvimento mais rápido sendo que os testes eram mais rápidos de executar. Todos os cenários criados estão dentro da pasta resources, contendo vários ficheiros .txt para os vários ambientes.

**Lista de requisitos implementados**

**Meta 1 – 27 de novembro**

- Requisitos - funcionalidades para a meta 1:  
 - [x] Leitura de ficheiros  
 - [x] Ficheiro de comandos  
 - [x] Ficheiro de constantes.txt  
 - [x] Construção da reserva.   
 - [x] A representação da reserva irá ser melhorada com matéria dada  
 posteriormente e agora só se pretende algo que possa ser representado no ecrã.  
   
 - [x] Definição do conceito de Animal.   
 - [x] Não é preciso considerar as variações inerentes às espécies  
 e deve mesmo limitar-se aquilo que é genérico e comum a todos. (verificar again)  
 - [x] Definição do conceito de Alimento.   
 - [x] Idem focar apenas o que é comum a todos os alimentos. (verificar again)  
 - [x] Representação visual da reserva e conteúdo incluído nesta meta.   
 - [x] Inclui-se aqui a questão de  
 ver apenas a área visível da reserva.  
   
 - [x] Implementação da leitura e validação de todos os comandos, seja por teclado, ou seja por  
 leitura do ficheiro de comandos.   
 - [x] Base projeto para leitura e validação dos comandos  
 - [x] Os comandos não farão ainda nada, mas devem ser já  
 interpretados e validados  
 - [x] incluindo a sintaxe de todos os parâmetros que tenham.  
 - [x] Implementar os comandos para:   
 - [x] ver animais  
 - [x] anim  
 - [x] visanim  
 - [x] info  
 - [x] deslizar a área visível para o lado/cima/abaixo  
 - [x] executar comandos em ficheiro (que são também validados), e terminar.  
 - [x] projeto já deverá estar devidamente organizado em .h e .cpp

**Meta 2 – 08 de Janeiro**

- [x] implementa os comandos básicos

- [x] exec\_command\_nofood

- [x] exec\_command\_feed

- [x] exec\_command\_feedid

- [x] exec\_command\_see

- [x] exec\_command\_killid

- [x] exec\_command\_kill

- [x] exec\_command\_empty

- [x] exec\_command\_animal

- [x] exec\_command\_food

- [x] remove todos os animais fictícios da inicialização do código

- [x] criar arquivos de teste de comando com diferentes ambientes

- [x] vários animais na mesma coluna de linha

- [x] diferentes tipos de alimentos

- [x] diferentes tipos de animais

- [X] implementar os tipos de gestão S e V para todos os animais e alimentos (constantes.txt)

- [x] Tipos S

- [X] Tipos V

- [x] criar uma maneira de armazenar toda a reserva de memória do jogo executada pelo utilizador

- [x] implementa o exec\_command\_store

- [x] implementa o comando exec\_command\_restore

- [x] verificação global do projeto

- [x] adicionar validação para números inteiros na configuração do tamanho da reserva

- [x] adicionar validações nos comandos que pedem linha/colunas (verificar se a linha/coluna é válida)

- [x] fazer melhorias no comando slide

- [x] mudar a exibição da área visível

- [x] verificar as propriedades dos animais/alimentos e corrigir se necessário

- [x] ter uma reunião para decidir as principais funções necessárias para as interações dos animais

- [x] implementa a função "daily" dentro do execute\_command\_n

- [x] finalize o comando execute\_command\_n

- [x] implementar as principais funções necessárias para todas as interações dos animais

- [x] raposa

- [x] lobo

- [x] canguru

- [x] coelho

- [x] ovelhas

- [x] implementam as principais funções necessárias para todas as interações dos alimentos

- [x] cenoura

- [x] carne

- [x] mirtilos

- [x] corpo

- [x] grama

- [ ] última verificação global do projeto

- [x] apenas um alimento por posição deve ser permitido

**Dificuldades e resoluções**

A maior dificuldade encontrada durante o desenvolvimento do projeto foi a verificação da área visível por parte dos animais em área sendo que estes estando na posição 0,0 é possível ver o canto inferior direito do tabuleiro.

Após uma investigação demos conta que o que queríamos representar no trabalho era um toro, ou seja, um donut, sendo que o animal consegue percecionar tudo o que está a sua volta e que as pontas estão conectadas.

A outra dificuldade foi relacionada com imports do projeto tendo problemas com dependências circulares na classe Animal. A classe animal continha a necessidade de receber o ponteiro para o jogo, sendo que seriam feitas alterações na matriz e apercebemo-nos que bastava declarar que aquele header file utilizava uma estrutura Game sendo o import na classe .cpp sempre que necessário. Esta descoberta tornou-se bastante útil e facilitou o desenvolvimento do projeto e a remoção de um ficheiro com ponteiro para o jogo declarado no inicio da aplicação que só seria utilizado na implementação de cada animal com o intuito de ter acesso a matriz.