UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

AUGUSTO BORANGA
RODRIGO FRANZOI SCROFERNEKER

Implementação de um Tower Defense em Swift - Entrega parcial

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2017

SUMÁRIO

1 INTRODUÇÃO

Escolhemos para implementar o problema do jogo Tower Defense.

Tower Defense é um subgênero de jogos de estratégia, onde o jogador deve defender seu território (que varia de acordo com a temática do jogo) contra o ataque de inimigos, podendo utilizar um montante inicial para adquirir elementos do jogo que o ajudem na defesa.

Nosso código-fonte (além de zipado junto com este relatório) encontra-se no seguinte repositório:

`https://github.com/gutoboranga/tower-defense`

2 PARADIGMA

Para a primeira etapa do trabalho, optamos por implementar o jogo usando Orientação a Objetos devido a conhecimento prévio de ambos integrantes do grupo.

Acreditamos que será mais fácil implementar o jogo no paradigma funcional como segunda etapa ao invés da primeira, pois agora temos um conhecimento maior acerca da linguagem e das regras do próprio jogo.

3 REQUISITOS

A seguir está a lista de requisitos para o paradigma de Orientação a Objetos.

Para cada requisito cumprido na implementação, há um exemplo retirado do nosso código-fonte.

3.1 Classes

Especificar e utilizar classes (utilitárias ou para representar as estruturas de dados utilizadas pelo programa).

Utilizamos diversas classes no projeto inteiro, seguindo o padrão MVC com leves alterações devido à presença do conceito de GameScene, trazido pela biblioteca de jogos SpriteKit.

Exemplo: classe `Projectile`, que representa um projétil atirado pelas torres do jogo.

```
class Projectile:  StandardBlock {
    ...
}
```

3.2 Encapsulamento

Fazer uso de encapsulamento e proteção dos atributos, com os devidos métodos de manipulação (setters/getters) ou propriedades de acesso, em especial com validação dos valores (parâmetros) para que estejam dentro do esperado ou gerem exceções caso contrário.

Exemplo:

Praticamos onde possível (e necessária) a proteção dos atributos. Por exemplo, na classe `LifeBar`, que representa a porcentagem de vida dos personagens do jogo (e seu elemento visual, no formato de uma barra que muda de acordo com o valor associado), há as seguintes funções de get e set.

```
public func getLife() -> Double {
    return self.actualLife
```

```
}

public func setLife(newValue:  Double) {
    self.actualLife = newValue
}
```

O objetivo aqui é proteger o atributo privado `actualLife`, que representa o quanto de vida o objeto que possui este `LifeBar` ainda possui dentro do jogo.

3.3 Construtores-Padrão

Especificação e uso de construtores-padrão para a inicialização dos atributos e, sempre que possível, de construtores alternativos.

Na maioria das classes implementadas, criamos apenas um construtor - com parâmetros - pois não encontramos necessidade de criar construtores padrão, uma vez que em Swift pode-se inicializar os atributos de uma classe junto com sua declaração. Ao adotar esta prática, os atributos já terão valores default na criação do objeto.

Na classe `Enemy`, temos um construtor padrão que inicializa os elementos que ainda não haviam sido inicializados na declaração e chamamos o método `init` da classe pai (super.init(parâmetros)).

3.4 Destrutores

Especificação e uso de destrutores (ou métodos de finalização), quando necessário.

Não houve necessidade de criar destrutores nas classes implementadas pois o framework utilizado lida com a desalocação de memória de maneira competente.

3.5 Espaços de nome

Organizar o código em espaços de nome diferenciados, conforme a função ou estrutura de cada classe ou módulo de programa.

Não foi implementado.

3.6 Herança

Usar mecanismo de herança, em especial com a especificação de pelo menos três níveis de hierarquia, sendo pelo menos um deles correspondente a uma classe abstrata, mais genérica, a ser implementada nas classes-filhas.

Exemplo:

Temos a classe `StandardBlock`, que é uma classe abstrata da qual a maioria dos personagens do jogo herda. `StandardBlock` possui apenas um atributo de orientação no espaço e métodos para manipular este atributo.

Dentre suas herdeiras está a classe `Enemy`, que representa o tipo genérico de todos os inimigos presentes no jogo. Esta classe possui atributos e métodos comuns a todas as especificações de inimigos.

Por sua vez, `Enemy` possui algumas classes herdeiras que especializam e personalizam o conceito genérico de inimigo, com implementações específicas de funções genéricas de acordo com as características deste personagem. Por exemplo, temos a classe `AstronautEnemy`, que representa o personagem astronauta.

3.7 Polimorfismo por inclusão

Utilizar polimorfismo por inclusão (variável ou coleção genérica manipulando entidades de classes filhas, chamando métodos ou funções específicas correspondentes).

Como citado anteriormente na seção Herança, temos a classe `Enemy` representando a classe genérica dos inimigos no jogo. Ela possui alguns métodos que devem ser sobrecarregados pelas suas herdeiras, de acordo com as características destas.

Por exemplo, o método `getDamageValue()`, que retorna um Double representando o valor do dano que este inimigo causa ao atingir o castelo do jogador. Para a classe `AstronautEnemy` (que no âmbito do jogo representa um ser humano), esta função é implementada de forma a retornar o valor 0.7, enquanto que na classe `RoverEnemy` (que representa um veículo de exploração espacial, portanto mais forte do que um ser humano), a implementação desta função retorna o valor 2.0.

3.8 Polimorfismo paramétrico

Usar polimorfismo paramétrico através da especificação de *algoritmo* (método ou função genérico) utilizando o recurso oferecido pela linguagem (i.e., generics, templates ou similar) e da especificação de *estrutura de dados* genérica utilizando o recurso oferecido pela linguagem.

Exemplo:

Implementamos uma função que previne erros na física da colisão dos elementos do jogo. Esta função recebe dois parâmetros de qualquer tipo e retorna um booleano que indica se seu tipo (mais genérico) é o mesmo.

```
func sameType<T,E>(one :  T, two:  E) -> Bool {
    if type(of:  one) == type(of:  two) {
        return true
    }
    return false
}
```

3.9 Polimorfismo por sobrecarga

Usar polimorfismo por sobrecarga (vale construtores alternativos).

Utilizamos polimorfismo por sobrecarga na classe `Enemy`, onde temos duas implementações (com assinaturas diferentes) para a função `move()`:

- Uma delas é pública e não recebe parâmetros, tendo a seguinte assinatura, portanto:

  `public func move()`

  Esta serve para iniciar o movimento de um inimigo, e é chamada de fora da classe que a implementa, indicando o início do movimento de um inimigo.

- A segunda é privada e recebe como parâmetro uma lista de movimentos (pares ordenados x, y) a serem feitos pelo inimigo em questão. Sua assinatura é a seguinte:

  `private func move(_ newDir :  [[CGFloat]])`

  Nesta função privada, é realizado o movimento até a primeira posição (x, y) da lista

recebida e após, são feitas chamadas recursivas passando como parâmetro a lista sem o primeiro elemento.

3.10 Delegates

Especificar e usar delegates.

Utilizamos o conceito de delegate (em Swift conhecido como protocol) para fazer a comunicação entre os elementos visuais do jogo (View, no padrão MVC) com a parte mais conceitual (Model) sem criar dependências desnecessárias entre estas classes.

Exemplo:

Na classe `Tower` (que modela o conceito de uma torre de defesa dentro do jogo), é declarado o seguinte delegate:

```
protocol TowerDelegate {
    func removeTower(tower :  Tower)
    func upgradeTower(tower:  Tower)
}
```

A classe que implementa este delegate é a `GameScene`. Ela é, portanto, a responsável por tomar as ações necessárias quando a classe `Tower` acionar algum dos métodos declarados neste protocol. Estas ações envolvem chamadas a métodos de outras classes. O uso deste delegate evita que a classe `Tower` fique acoplada a estas outras classes, deixando que a `GameScene` faça a comunicação necessária entre elas.

4 SCREENSHOTS

A seguir estão listados alguns screenshots tirados durante a execução do nosso jogo.

Figura 4.1: Menu inicial

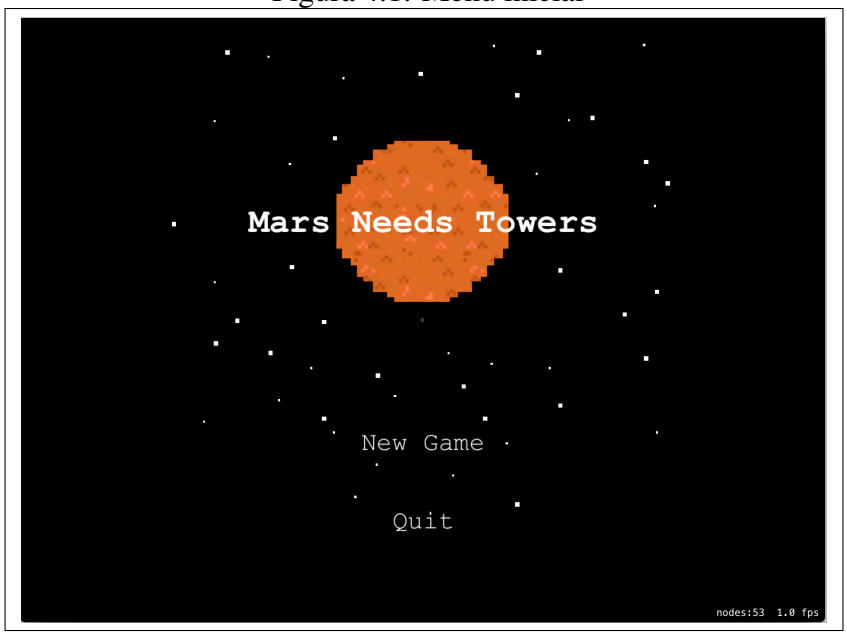


Figura 4.2: Jogo em ação

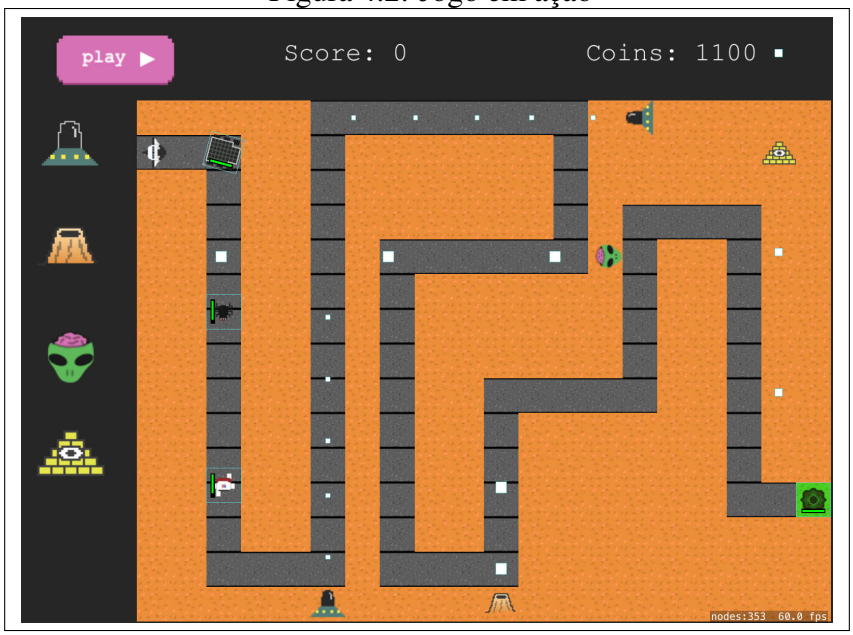


Figura 4.3: Final do jogo: vitória

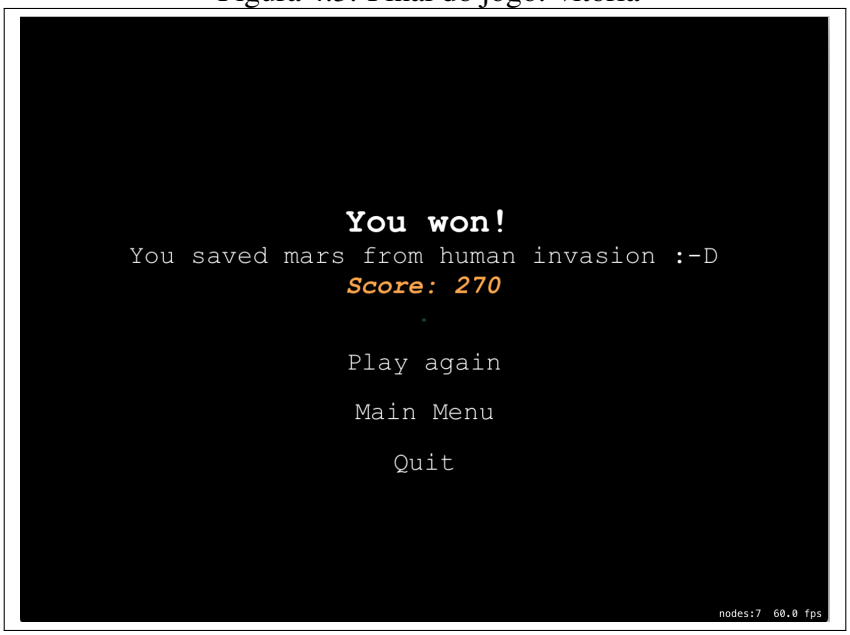
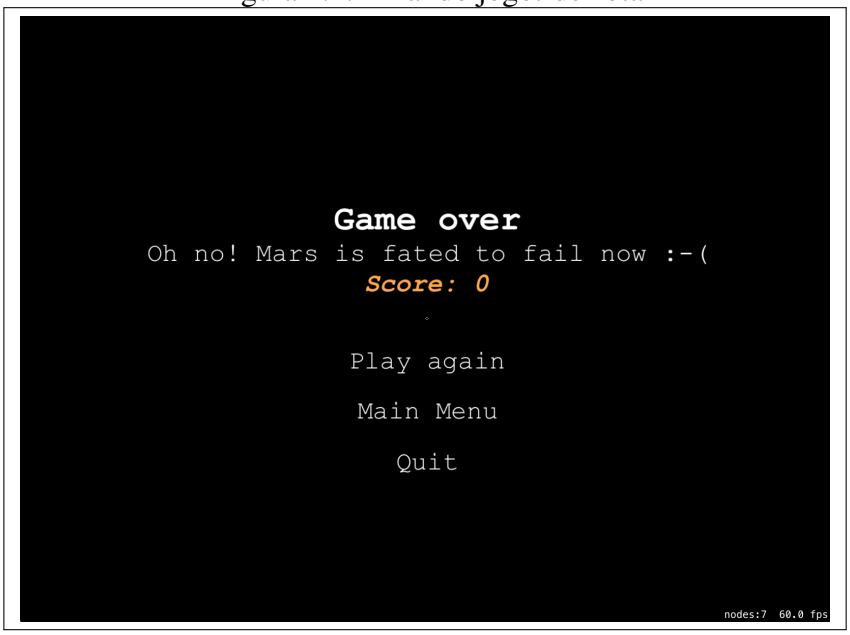


Figura 4.4: Final do jogo: derrota



5 CONCLUSÃO

Ao término deste etapa do trabalho, podemos concluir que utilizar Orientação a Objetos em projetos que envolvem um grande número de entidades e elementos com diferentes características é muito apropriado.

Isto se deve ao fato de este paradigma se basear na criação de tipos abstratos (com seus atributos e métodos próprios), que permite uma maior separação nas responsabilidades de cada módulo do programa, resultando em um código-fonte mais organizado e conciso.

6 REFERÊNCIAS

Acessos feitos em 1/12:

- https://developer.apple.com/swift/
- https://developer.apple.com/spritekit/
- https://www.devmedia.com.br/uso-de-polimorfismo-em-java/26140
- http://nsomar.com/parametric-compile-time-polymorphism-in-swift/
- https://medium.com/@jamesrochabrun/implementing-delegates-in-swift-step-by-step-d3211cbac3ef
- http://www.dca.fee.unicamp.br/cursos/PooJava/metodos/construtor.html
- https://en.wikipedia.org/wiki/Delegation_(object-oriented_programming)