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

Este relatório discorre sobre o trabalho final da disciplina de Modelos de Linguagem de Programação, explicando as decisões tomadas e os requisitos cumpridos ao longo do desenvolvimento.

O objetivo deste trabalho é implementar um problema computacional em dois paradigmas diferentes (Orientação a Objetos e Funcional) utilizando uma mesma linguagem.

Escolhemos implementar um jogo do gênero Tower Defense com a linguagem Swift.

Nosso código-fonte encontra-se no seguinte repositório:

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

2 PROBLEMA

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.

2.1 Dinâmica do jogo

Basicamente, há dois elementos essenciais em um jogo do estilo Tower Defense:

- Territórios ou propriedades com certa quantidade de vida (o esgotamento desta implica em fim de jogo), que o jogador deve defender;
- Inimigos (com outra quantidade de vida) atacando os territórios do jogador;

A dinâmica do jogo com estes elementos é de que há "ondas"de ataques dos inimigos ao jogador. Isto é, o ataque ocorre em partes, com pequenas pausas entre eles.

O jogador pode então, entre ou durante as ondas de ataques (isto varia de jogo para jogo), utilizar-se de subterfúgios que atrapalhem a missão dos inimigos (como por exemplo, posicionar obstáculos ou outras estruturas que os ataquem). A aquisição destes equipamentos de defesa custa um certo valor que é decrementado do montante disponível para o jogador.

O objetivo do jogador é sobreviver ao final das N ondas de ataques inimigos.

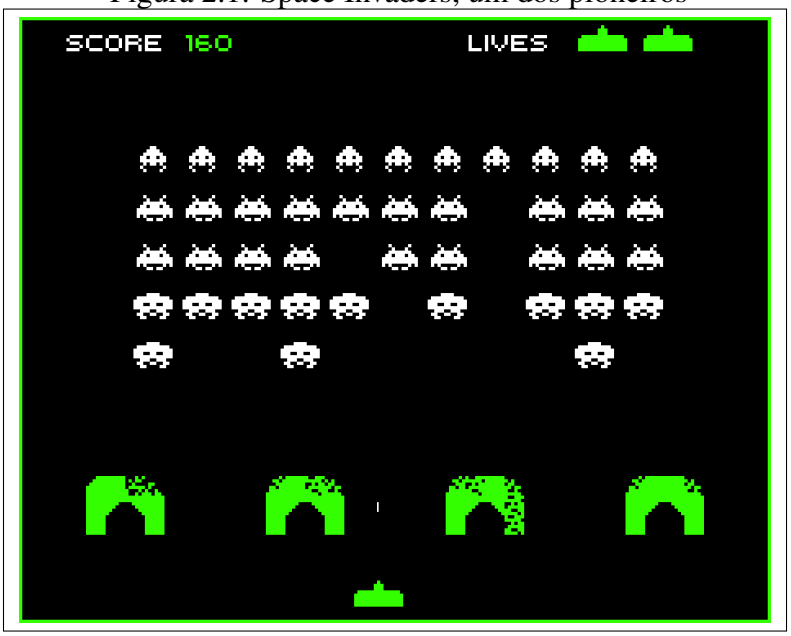
2.2 Historicamente

Os primeiros jogos de Tower Defense datam da década de 1980, considerada a Era de Ouro dos video-games.

No jogo *Space Invaders* - um clássico dos video-games lançado em 1978 -, o jogador deve defender seu território atirando em invasores alienígenas. É tido por uns como um precursor dos jogos de Tower Defense, mas isto é contestado por outros pelo fato de não possuir um elemento fundamental do gênero: a possibilidade de aquisição de elementos extras que auxiliem o jogador na defesa.

Já o jogo *Rampart* - lançado em 1990 -, é amplamente considerado como o jogo que definiu o gênero, por possuir todos os elementos fundamentais deste. Em *Rampart*, as fases de preparação (posicionamento dos itens de defesa), ação (momento em que a base é atacada e o jogador deve defendê-la) e reparação (consertar elementos danificados pelos ataques) são bem distintas.

Figura 2.1: Space Invaders, um dos pioneiros



Fonte: http://fantendo.wikia.com/wiki/File:Space-Invaders.png

Figura 2.2: Rampart, o Tower Defense clássico

Fonte: http://www.wikiwand.com/en/Rampart_(video_game)

3 LINGUAGEM

Swift é uma linguagem de programação multiparadigma que se apresenta como uma linguagem moderna e focada em três aspectos: segurança, performance e suporte à aplicação de design patterns.

Mesmo sendo consideravelmente recente, Swift demonstra uma comunidade de bom tamanho, ficando em 11º nas linguagens mais populares e a 4ª mais amada no site stackoverflow em 2017.

Neste capítulo daremos um panorama a respeito da linguagem adotada, explanando acerca de suas origens e discutindo seus pontos positivos e negativos.

3.1 História

Swift é uma das linguagens de programação mais recentes desenvolvidas no mercado. Foi apresentada em 2014 na WWDC (Worldwide Developers Conference, evento organizado pela Apple para divulgação de novos produtos e features).

Baseada e influenciada por Objective-C, Ruby e Python, se mostrou uma linguagem poderosa e de fácil compreensão (devido principalmente, à sua sintaxe simples).

Inicialmente, Swift era de uso exclusivo por usuários de Mac OS, visto que este era o único sistema operacional habilitado a compilar a linguagem. Em dezembro de 2015, porém, um grande anúncio mudou o jogo: Swift viraria open source, abrindo um leque ainda maior de possibilidades de uso para a linguagem.

3.2 Aspectos técnicos

Swift é uma linguagem multiparadigma, suportando programação funcional e também oferecendo recursos de Orientação a Objetos, como classes e protocolos.

A linguagem possui tipagem estática, o que proporciona maior segurança (garantia dos tipos de dados esperados no código, evitando erros de tipos) e performance (não há gasto de máquina com checagem dos tipos) às aplicações que a utilizam.

Por ter sido construída pela Apple, possui alta integração com Obj-C, outra linguagem utilizada no ambiente da Apple. Assim sendo, o programador que utiliza Swift tem à sua disposição uma série de bibliotecas em Obj-C, como por exemplo *SpriteKit*, a

biblioteca de construção de jogos que utilizaremos no desenvolvimento do trabalho.

Sua sintaxe foi pensada para ser o mais simples e expressiva possível, tornando o código mais fácil de ler e escrever.

3.3 Utilização

Mesmo sendo uma linguagem relativamente recente, Swift agradou a comunidade de desenvolvimento. Na pesquisa do site stackoverflow realizada em 2017, Swift ficou em 4º lugar nas linguagens preferidas pelos usuários.

Suas principais aplicações são:

- **Mobile**: Swift é a linguagem oficial para desenvolvimento de aplicativos para a plataforma iOS. Obj-C também é suportada em iOS, mas o programador mobile é encorajado pela própria Apple a utilizar Swift por esta ser uma linguagem mais recente e simples de usar;

- **Desktop**: Similar à plataforma mobile (citada acima), aplicações para o sistema operacional dos computadores da Apple (macOS, antigamente chamado OS X) também podem ser escritas em Swift;

- **Servidor**: Além disso, Swift também pode ser utilizado para o desenvolvimento de aplicações no lado do servidor. Existem alguns frameworks e toolkits (como o Perfect) que auxiliam o desenvolvedor nesta tarefa.

3.4 Motivo da escolha

Optamos por Swift por dois motivos:

- Ambos os integrantes do grupo tem familiaridade com a linguagem, devido a experiência prévia desenvolvendo aplicativos mobile.

- Swift é uma linguagem muito intuitiva e possui uma variedade de bibliotecas auxiliares disponíveis.

3.5 Análise

A análise crítica será feita ao término do trabalho.

4 IMPLEMENTAÇÃO

4.1 PARADIGMA ORIENTADO A OBJETOS

Na 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.

4.1.1 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.

4.1.1.1 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 {
    ...
}
```

4.1.1.2 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.

4.1.1.3 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)).

4.1.1.4 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.

4.1.1.5 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.

4.1.1.6 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.

4.1.1.7 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.

4.1.1.8 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
}
```

4.1.1.9 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.

4.1.1.10 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.

4.2 PARADIGMA FUNCIONAL

Na segunda etapa do trabalho, implementamos o jogo utilizando o paradigma de Programação Funcional.

4.2.1 Requisitos

A seguir estão listados os requisitos para o paradigma funcional. Para cada requisito cumprido na implementação, há um exemplo retirado do nosso código-fonte.

4.2.1.1 Elementos imutáveis e funções puras

Priorizar o uso de elementos imutáveis e funções puras (por exemplo, sempre precisar manipular listas, criar uma nova e não modificar a original, seja por recursão ou

através de funções de ordem maior).

4.2.1.2 Funções lambda

Especificar e usar funções não nomeadas (ou lambda).

4.2.1.3 Currying

Especificar e usar funções que usem currying.

4.2.1.4 Pattern matching

Especificar funções que utilizem pattern matching ao máximo, na sua definição.

4.2.1.5 Funções de ordem superior próprias

Especificar e usar funções de ordem superior (maior) criadas pelo programador.

4.2.1.6 Funções de ordem superior prontas

Usar funções de ordem maior prontas (p.ex., map, reduce, foldr/foldl ou similares).

4.2.1.7 Funções como elementos de 1ª ordem

Especificar e usar funções como elementos de 1ª ordem.

4.2.1.8 Recursão

Usar recursão como mecanismo de iteração (pelo menos em funções de ordem superior que manipulem listas).

5 SCREENSHOTS

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

Figura 5.1: Menu inicial

Figura 5.2: Jogo em ação

Figura 5.3: Final do jogo: vitória

Figura 5.4: Final do jogo: derrota

6 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.

REFERÊNCIAS

BEST Tower Defense Games of All Time. <http://gameranx.com/features/id/13529/article/best-tower-defense-games/>. Accessado em: 2017-10-28.

HOW to use BibTeX. <http://www.bibtex.org/Using/>. Accessado em: 2017-12-14.

O que é paradigma? - stackoverflow. <https://pt.stackoverflow.com/questions/141624/o-que-%C3%A9-paradigma>. Accessado em: 2017-10-28.

SWIFT - Apple Developer. <https://developer.apple.com/swift/>. Accessado em: 2017-10-28.

TOWER Defense. <https://en.wikipedia.org/wiki/Tower_defense>. Accessado em: 2017-10-28.

WHY Learn Swift. <http://www.bestprogramminglanguagefor.me/why-learn-swift>. Accessado em: 2017-10-28.