UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

DEPARTAMENTO DE INFORMÁTICA APLICADA

Desenvolvendo um Tower Defense em Python

Bruno Iochins Grisci - 208151
Jorge Alberto Wagner Filho - 205975
19 de junho de 2013

1 PYTOWER DEFENSE, UM TOWER DEFENSE EM PYTHON

Para o trabalho final da disciplina de Modelos de Linguagens de Programação, escolhemos a opção do jogo de tower defense por conta dos conhecimentos prévios da dupla sobre o tema e vontade de implementar um projeto que tanto pudesse ser utilizado para demonstrar o aprendizado adquirido na disciplina quanto para fins práticos (no caso, jogar o jogo).

1.1 TOWER DEFENSE

No universo dos jogos eletrônicos, os tower defense[1] (defesa de torre) existem como um subgênero dos jogos de estratégia em tempo-real. Embora existam inúmeras versões e adaptações, tradicionalmente consistem de um mapa com uma base estática que é constantemente atacada por hordas de inimigos, os quais percorrem a tela até conseguirem tomá-la. O objetivo do jogador é proteger sua base adicionando unidades imóveis (as "torres") na tela, capazes de impedir o avanço dos inimigos. Detalhes como tipos de inimigos e de torres, traçado do mapa e outras estratégias são características de cada jogo.

1.2 NOSSO JOGO

Para nosso trabalho, decidimos criar uma paródia com o tema Python (a linguagem que escolhemos), por isso definimos os inimigos como pequenas cobras. Ao todo existem seis diferentes cobras que diferem em seu desenho, velocidade, poder de ataque e vida. Estes

inimigos são criados em posições aleatórias dentro de um intervalo definido acima do limite superior do mapa e sempre avançam em linha reta para baixo até encontrarem a base do jogador, representada por uma muralha. Quando a cobra colide com a muralha, passa a atacá-la e também é por ela atacada. Se a base sofrer danos suficientes, pega fogo e o jogo acaba.

Figura 1.1: Tela do jogo

O objetivo do jogador é proteger sua muralha através do posicionamento de torres na tela. Elas são representadas por um obelisco negro que dispara um laser vermelho em uma das cobras que estiver dentro de seu campo de visão (indicado por um círculo vermelho na hora da construção da mesma), causando-lhe dano, e podem ser colocadas em qualquer posição válida do mapa (dentro dos limites da tela e fora da base). As torres são indestrutíveis, sendo a estratégia o melhor posicionamento e evolução das mesmas. Para construi-las, é preciso ter dinheiro suficiente na carteira, obtido ao matar cobras. Quando houver dinheiro o bastante, o jogador pode escolher entre construir uma nova torre (botão esquerdo do mouse), aumentar o poder de ataque das torres que já possui (botão direito do mouse) ou jogar uma bomba que elimina a maioria das cobras na tela (barra de espaço). O custo para a realização destas ações cresce a cada “compra”.

Figura 1.2: Os seis diferentes tipos de cobra

Novas cobras são geradas automaticamente sempre que uma é morta, portanto não existe situação de vitória no jogo. Em vez disso, optamos por uma mecânica de arcade onde o objetivo é acumular o maior número de pontos (no caso, número de cobras mortas) até a base ser destruída pelas cobras. A partida fica mais difícil com o passar do tempo pois o número de inimigos simultâneos cresce a medida que os antigos vão sendo eliminados.

2 Ambiente de trabalho

O trabalho foi implementado em um computador com Windows 7 64 bits, utilizando a linguagem Python em sua versão mais recente, a 3.3[2]. Utilizamos a biblioteca Pygame[3], construída com o propósito de auxiliar a criação de jogos oferecendo métodos para facilitar a manipulação de imagens, áudio e eventos de mouse e teclado. A biblioteca cx_Freeze[4] foi usada para a criação de um script para gerar o executável do jogo, de forma que ele seja jogável mesmo em computadores sem Python e as demais bibliotecas instaladas. A IDE adotada para o projeto foi a PyScripter[5]. Ao programar-se em Python, uma IDE é preferível a um editor de texto padrão por conta da indentação obrigatória.

As imagens base das cobras foram desenhadas utilizando a ferramenta de edição gráfica do Google Drive, bem como esboços iniciais. As demais figuras foram obtidas da pesquisa no Google, sendo o obelisco negro[6] originário do jogo "Empires & Allies" e a muralha[7] uma criação de fãs para a série "Age of Empires". As imagens foram manipuladas com o editor PhotoScape[8].

2.1 A linguagem Python

Python é uma linguagem sem fins lucrativos e relativamente recente, surgiu em 1991 criada pelo holandês Guido van Russum. Trata-se de uma linguagem interpretada de altíssimo nível e sintaxe simplificada, de forma que o trabalho do programador seja facilitado. A ausência de ponto e vírgula como marcador de final de linha e demarcação de blocos por indentação em vez do uso de chaves são um indício dessa filosofia e da preocupação com a legibilidade do código.

De forma bem humorada, a filosofia de Python é ilustrada pelo poema "The Zen of Python", que pode ser lido com o comando "import this".

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python é uma linguagem imperativa orientada a objetos, mas com diversas características de linguagens funcionais, tais como funções anônimas (lambda) e de alta ordem (map, filter), o que permite um pouco mais de flexibilidade na hora de programar. A tipagem é dinâmica e forte, excluindo a necessidade de declarar explicitamente o tipo de uma variável, que é definido pelos valores que a mesma recebe. Por isso uma variável pode trocar de tipo ao longo da execução, o que exige uma maior atenção do programador para não cometer erros e para a leitura do código. Os tipos nativos de dado são str, list, tuple, set, dict, int, float, complex e bool, e novos tipos de dados podem ser criados através da construção de classes.

No controle de fluxo e operadores, Python se assemelha a sintaxe de C, mas com diferenças importantes. Os operadores lógicos são nominais em vez de simbólicos (and no lugar de &&), não há marcadores de fim de linha de código e nem o uso de chaves para definir blocos, que são substituídos pelos dois pontos e pela indentação. Em Python, existe essa característica de utilizar palavras em vez de pontuação para deixar a leitura mais agradável.

A existência de um interpretador interativo permite que se teste instruções da linguagem sem a necessidade de executar um código, o que é bastante útil para o aprendizado. Python é uma linguagem de propósitos gerais. Como é interpretada, normalmente é menos eficiente que uma linguagem compilada.

3 IMPLEMENTAÇÃO DO JOGO

3.1 CLASSES

Por ser uma linguagem orientada a objetos, nada mais natural que a utilização de classes em nosso programa. Criamos classes e subclasses para todas as estruturas do jogo, como as cobras e torres, o que facilitou muito a manipulação das mesmas através de seus métodos no restante do programa e melhorou bastante a legibilidade do código. Existem várias classes no jogo, abaixo duas delas são exibidas como exemplo.

```
class Carteira:
    _dinheiro = 20
    def __init__(self):
        pass
    def getDinheiro(self):
        return self._dinheiro
    def setDinheiro(self, valor):
        self._dinheiro = valor
    def compraLiberada(self, item):
         return (lambda x, y: x >= y)(self._dinheiro, item.getCusto())
    def ganhaDinheiro(self, valor):
        self._dinheiro = self._dinheiro + valor
    def gastaDinheiro(self, item):
        if self.compraLiberada(item):
            self._dinheiro = self._dinheiro - item.getCusto()
```

```
class Base(Componente):
    _TAMANHO_IMAGEM_ORIGINAL = (0,0)
    _dimensao = [880, 186]
    _posicao = [100, 247]
    _FILEPATH = 'wall.png'
    _FILEPATHdestruicao = 'destrocos.png'
    _imagem = pygame.transform.scale(pygame.image.load(_FILEPATH), _dimensao)
    _imagem_destruicao = pygame.transform.scale(pygame.image.load(_FILEPATHdestruicao),
    _vida = 5500 ##Resistencia da base ate ser destruida
    _ataque = 2
    def __init__(self):
        pass
    def getImagemDestruicao(self):
        return self._imagem_destruicao
    def destruida(self):
        _vida = 0
        _ataque = 0
```

Em Python, as classes são definidas através da palavra reservada "class". Carteira implementa a "bolsa de moedas"virtual do jogador, que será utilizada na hora de receber o dinheiro ganho

matando cobras e também quando gastá-lo ao construir novas torres, melhorar seu ataque ou jogar bombas. A classe Base implementa a base do jogador, graficamente representada pela muralha. Note que ela é herdeira da classe Componente, como pode ser visto em

```
class Base(Componente):
```

Falaremos mais sobre isso adiante.

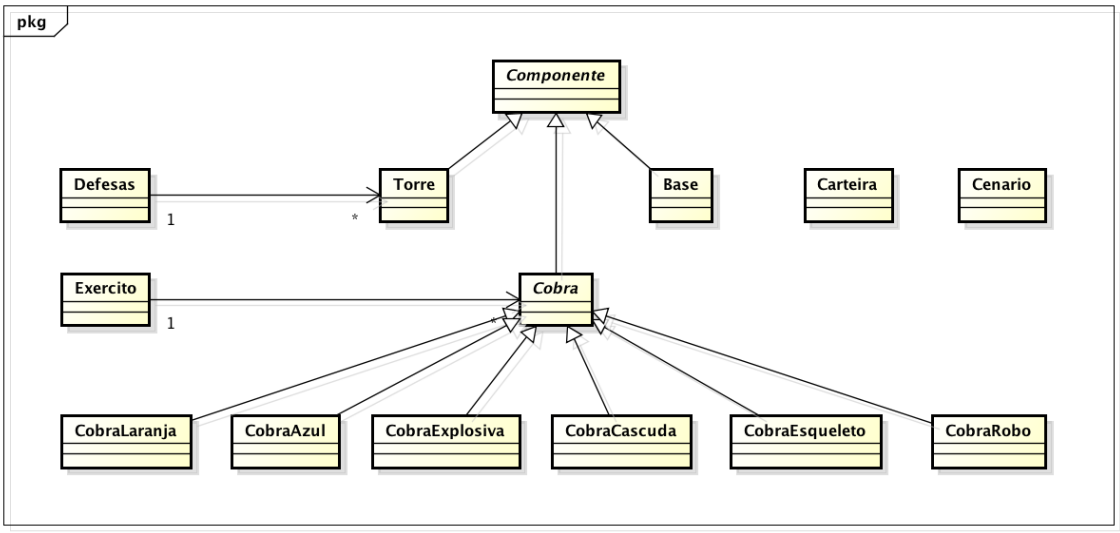

Figura 3.1: Diagrama de classes do jogo simplificado

3.2 Encapsulamento e proteção dos atributos

Não faria muito sentido utilizarmos classes e não aproveitarmos as vantagens proporcionadas por tipos abstratos de dados. Cada classe possui métodos e atributos próprios que nem sempre queremos que sejam visíveis para o "mundo exterior". Em linguagens orientadas a objetos, tradicionalmente, contamos com níveis de privilégio como "public", "protected"e "private"que definem quem pode acessar determinados atributos e métodos. Infelizmente, em Python, não existe uma grande preocupação em limitar este acesso, de forma que tais níveis não estão bem definidos. O que existe é uma convenção que determina que atributos ou métodos cujo nome é iniciado por _ serão protegidos e iniciados por __ serão privados.

Apesar disso, em nosso jogo nos preocupamos em tentar proteger os atributos das classes da melhor maneira possível, utilizando getters e setters sempre que necessário, de forma a aumentar a segurança e padronização do código. Um exemplo desta utilização pode ser visto abaixo na classe Torre, que implementa as torres do jogador. É possível observar que os atributos que devem ser acessíveis são manipulados e lidos através de métodos get e set.

```
class Torre(Componente):
    _ataque = 1
    _TAMANHO_IMAGEM_ORIGINAL = (134,164)
```

```
    _dimensao = [134, 164]
    _coordenada = [375, 655]
    _posInicial = [375, 655]
    _FILEPATH = 'Black_Obelisk.png'
    _imagem = pygame.image.load(_FILEPATH)
    _raiodealcance = 200
    _custo = 10
    _audio = pygame.mixer.Sound('Hammering.wav')
    def __init__(self):
        pass
    def getCoordenadaPonta(self):
        return [self._coordenada[0]+38, self._coordenada[1]+1]
    def getRaio(self):
        return self._raiodealcance
    def distancia(self, alvo):
        return math.sqrt(math.pow(self.getCoordenadaPonta()[0]
                        - alvo.getCoordenadaCentral()[0],2) +
                        math.pow(self.getCoordenadaPonta()[1] -
alvo.getCoordenadaCentral()[1],2))
    def alcancavel(self, alvo):
        if self._raiodealcance >= self.distancia(alvo):
            return True
        else:
            return False
    def getAudio(self):
        return self._audio
    def incAtaque(self):
        if self._ataque < 5:
            self._ataque = self._ataque + 1
```

Um ponto interessante a se observar é que em Python todo método deve obrigatoriamente receber como parâmetro "self", ou seja, o próprio objeto que o está chamando. Também é obrigatório referenciar métodos e atributos dentro das classes utilizando "self". Esse é um contra-ponto a outras linguagens como Java, que não exige a utilização de "this".

3.3 UTILIZAÇÃO DE MÓDULOS

Algo interessante em Python é que a linguagem possui dois níveis de encapsulamento. O primeiro são as classes, que já foram abordadas. Mas, ao contrário de algumas linguagens de programação, não existe a obrigatoriedade de manter uma única classe por arquivo. Pelo contrário, os arquivos de código de Python são chamados módulos e podem conter várias classes. Assim sendo, é possível agrupá-las por características semelhantes, deixando o código mais organizado.

Em nosso jogo criamos cinco módulos:

- Pytower_Defense: módulo principal do programa, contém o laço do jogo e desenha a interface gráfica.
- Mapa: contém elementos gráficos do mapa do jogo.
- Componentes: implementa a classe abstrata Componente utilizada para diversos elementos do jogo, como inimigos e torres.
- Jogador: classes do domínio do jogador, como Torre e Carteira.
- Inimigo: classes do domínio dos inimigos, como as classes de cada tipo de cobra.

Caso queira utilizar uma classe ou função de um módulo em outro, basta importá-lo. Um exemplo é o visto no módulo Pytower_Defense e reproduzido abaixo.

```
import Mapa, Inimigo, Componentes, Jogador
```

3.4 Mecanismos de herança

3.4.1 Classes abstratas e hierarquia

Como foi mencionando anteriormente, criamos uma classe abstrata chamada Componente que contém métodos e atributos que são herdados por diversas outras classes do programa. Por exemplo, métodos de redimensionamento.

```
class Componente:

    _TAMANHO_IMAGEM_ORIGINAL = (0,0)
    _dimensao = [0, 0]
    _vida = 0
    _ataque = 0
    _coordenada = [0, 0]
    _posInicial = [0, 0]
    _FILEPATH = ' '
    _imagem = None
    _valor = 0
    _custo = 0

    #Exemplo de currying simulado em Python
    def redimensiona(self):
        def redimensionaLargura(larg):
            def redimensionaAltura(alt):
                self._altura = alt
                self._largura = larg
            return redimensionaAltura
        return redimensionaLargura
```

```
    def redimensionaProp(self, escala):
        if escala > 0:
            self._dimensao = [math.floor(self._dimensao[0]*escala),
                              math.floor(self._dimensao[1]*escala)]
            self._imagem = pygame.transform.scale(self._imagem, self._dimensao)

    def atualizaVida(self, hp):
        self._vida = hp
    ...
```

Naturalmente, por ser uma classe abstrata, Componente não conta com um inicializador para instanciar objetos. Outra classe abstrata, que é herdeira de Componente, é a classe Cobra, que oferece atributos e métodos genéricos para todos os diferentes tipos de cobra.

```
class Cobra(Componente):

    _TAMANHO_IMAGEM_ORIGINAL = (99,227)
    _dimensao = [99,227]
    _coordenada = [0,0]
    _audiomorte = pygame.mixer.Sound('Bone Crushing.wav')
    _imagem_morte = pygame.image.load('Little dead snake.png')
    _valor = 5

    def andaVertical(self, deslocamento):
        self._coordenada[1] = deslocamento + self._coordenada[1]

    def andaHorizontal(self, deslocamento):
        self._coordenada[0] = deslocamento + self._coordenada[0]

    def getVelocidade(self):
        return self._velocidade

    def getCoordenadaCentral(self):
        return [self._coordenada[0] + math.floor(self._dimensao[0]/2),
                self._coordenada[1] + math.floor(self._dimensao[1]/2)]

    def setVelocidade(self, vel):
        self._velocidade = vel

    def morre(self):
        self._audiomorte.play()
        self._vida = 0
        self._ataque = 0
        self._velocidade = 0
        self._imagem_morte = pygame.transform.scale(self._imagem_morte, self._dimensao)
```

```
        self._imagem = self._imagem_morte
```

Mas e para criar os objetos cobra? Para isso temos classes específicas que herdam de Cobra os métodos e atributos genéricos. Como exemplo, veja a classe CobraLaranja.

```
class CobraLaranja(Cobra):
    _vida = 100
    _velocidade = 5
    _ataque = 1
    _FILEPATH = 'Little orange snake.png'
    _imagem = pygame.image.load(_FILEPATH)
    def __init__(self):
        pass
```

Agora sim temos uma classe que pode ser instanciada, pois temos o construtor

```
    def __init__(self):
        pass
```

3.4.2 POLIMORFISMO POR INCLUSÃO

Uma das vantagens de utilizarmos classes abstratas e herança é podermos definir métodos genéricos que servem para todas as classes herdeiras. Nos exemplos vistos acima, só foi necessário implementar o método "ataca"na classe Componente, e todas as classes filhas já podem chamá-lo. Mas e se uma cobra atacasse de forma diferente? Este é o caso da CobraExplosiva.

```
class CobraExplosiva(Cobra):
    _vida = 100
    _velocidade = 3
    _ataque = 10
    _FILEPATH = 'Little bomb snake.png'
    _FILEPATHexplosao = 'Explosion snake.png'
    _imagem = pygame.image.load(_FILEPATH)
    _imagem_morte = pygame.image.load(_FILEPATHexplosao)
    _audiomorte = pygame.mixer.Sound('Grenade.wav')
    def __init__(self):
        pass
    def ataca(self, alvo):
        alvo.danifica(self._ataque)
        self.morre()
```

Nesse caso o que fazemos é reescrever o método de forma que ele se adeque a ação que a cobra explosiva deve tomar. Essa redefinição foi necessária porque tal inimigo não ataca como os outros, em vez disso ela se explode ao colidir com a muralha do jogador, morrendo no processo mas causando grande dano. Ao chamarmos o método "ataca"para uma instância de CobraExplosiva, será chamada essa versão particular, e não a genérica definida em Componente.

3.5 Polimorfismo paramétrico

Por ser interpretada e usar tipagem dinâmica do estilo Duck typing[20], tentar implementar polimorfismo paramétrico em Python seria apenas gastar tempo recriando algo que a linguagem já oferece naturalmente. Todas as estruturas, como listas e dicionários, já são genéricas e aceitam elementos de diferentes tipos. Esta característica é evidenciada em nosso jogo, onde a mesma estrutura de lista é utilizada para armazenar tanto diferentes tipos de cobra quanto as torres do jogador.

```
class Exercito:
    _horda = []
    def __init__(self):
        pass
    def insere(self, monstro):
        self._horda.append(monstro)
    ...

class Defesas:
    _defesas = []
    _raiodealcance = 200
    _custo = 10
    _custo_da_bomba = 10
    _cordeexplosao = (255, 255, 153) #Pale Canary Yellow (Crayola Canary)
    _audioexplosao = pygame.mixer.Sound('Big Bomb.wav')
    def __init__(self):
        pass
    def insere(self, torre):
        self._defesas.append(torre)
    ...
```

3.6 Polimorfismo por sobrecarga

Infelizmente, em Python, o polimorfismo por sobrecarga não é suportado. Isso acontece porque na linguagem as chamadas de função são genéricas, sem determinação de tipo, tornando a sobrecarga inviável. É algo que faz falta, em nosso trabalho em mais de um momento quisemos criar métodos que aceitassem tipos diferentes de parâmetro, como nos que recebiam coordenadas, nos quais seria interessante poder passar como argumento tanto dois números quanto uma tupla ou uma lista de dois elementos.

3.7 Funções

3.7.1 Funções não nomeadas

Originadas da abstração matemática do cálculo lambda, funções não nomeadas são comuns em linguagens de programação funcionais e estão disponíveis em Python através da palavra reservada "lambda". Um exemplo de uso pode ser visto abaixo.

```
def compraLiberada(self, item):
    return (lambda x, y: x >= y)(self._dinheiro, item.getCusto())
```

3.7.2 Utilizando funções como elementos de primeira ordem e funções de ordem maior

Python permite que passemos funções como parâmetros para outras funções. Dentre as funções de maior ordem (as que recebem funções) nativas da linguagem, destacam-se a map e a filter. Aproveitamos esta característica da linguagem para a implementação da funcionalidade da evolução das torres. Sempre que tem dinheiro disponível, o jogador pode, clicando com o botão direito do mouse, aumentar em um o poder de ataque de suas torres. Conseguimos implementar este recurso criando uma lista a partir da aplicação de uma função de incremento em map, passando a lista de torres como parâmetro.

```
#Aumenta o ataque das torres
if(pygame.mouse.get_pressed()[2] and
carteira.compraLiberada(defesas) and
muralha.estaVivo() and defesas.lista() != []):
    def incrementa(x):
        x.incAtaque()
        return x
    defesas.setDefesas(list(map(incrementa, defesas.lista())))
    carteira.gastaDinheiro(defesas)
    defesas.setCusto(defesas.getCusto() + t.getCusto())
```

Neste trecho de código, incrementa() é a função de primeira ordem e map() a de ordem maior.

3.8 Manipulação de listas

Para o trabalho utilizamos listas diversas vezes. As principais são as das classes Exercito e Defesas que guardam as cobras e torres do jogo, respectivamente. Em geral, utilizamos para iterá-las o "for"do Python que permite uma construção simples e prática.

```
#Ataque das torres
        if muralha.estaVivo():
            for torre in defesas.lista():
                for cobra in horda.lista():
                    if cobra.visivel() and torre.alcancavel(cobra):
                        pygame.draw.line(TELADEJOGO, (255,0,0),
                        (torre.getCoordenadaPonta()),
                        cobra.getCoordenadaCentral(), torre.getAtaque())
                        torre.ataca(cobra)
                        break
```

Um exemplo de manipulação de listas utilizando a função "map"foi mostrado na seção anterior. Adiante será exibido também uma iteração em listas utilizando recursão.

3.9 CURRYING

Linguagens funcionais frequentemente oferecem a utilização de currying, isto é, permitir que uma função com múltiplos parâmetros seja chamada passando um parâmetro de cada vez, retornando uma função que recebe os parâmetros faltantes. Python, apesar de disponibilizar recursos funcionais, não implementa currying. É possível simular sua utilização, contudo, como é mostrado no código abaixo retirado de uma das classes do jogo.

```
#Exemplo de currying simulado em Python
def redimensiona(self):
    def redimensionaLargura(larg):
        def redimensionaAltura(alt):
            self._altura = alt
            self._largura = larg
        return redimensionaAltura
    return redimensionaLargura
```

Esta solução, contudo, não é satisfatória, sendo implementada apenas para indicar a sua viabilidade. Como em Python existe vinculação de parâmetros por valor padrão, uma alternativa caso deseje-se passar apenas uma entrada para a função é atribuir um valor a um ou mais de seus parâmetros, de forma que eles deixam de ser obrigatórios na chamada, assumindo os valores indicados no cabeçalho. No método abaixo, não é necessário fornecer um valor para "dano"ao chamá-lo, ele assume o valor trezentos.

```
def danifica(self, dano=300):
    self._vida = self._vida - dano
```

3.10 PATTERN MATCHING

O uso de casamento de padrões, geralmente visto em linguagens funcionais, é uma solução elegante e sucinta para testes, especialmente em estruturas de dados como árvores e listas. Em Python, não existe uma forma de casamento de padrões como a de Haskell ou ML, embora seja possível utilizar expressões regulares para o tratamento de sequências de caracteres de forma similar. O efeito também pode ser simulado com "ifs"e "elses"encadeados, como visto abaixo.

```
def criaHorda(self):
    for i in range(1,random.randrange(1,5)):
        tipo = random.randrange(1,7)
        if tipo == 1:
            cobra = CobraLaranja()
        else:
            if tipo == 2:
                cobra = CobraAzul()
            else:
                if tipo == 3:
```

```
                cobra = CobraExplosiva()
            else:
                if tipo == 4:
                    cobra = CobraCascuda()
                else:
                    if tipo == 5:
                        cobra = CobraEsqueleto()
                    else:
                        if tipo == 6:
                            cobra = CobraRobo()
        cobra.redimensionaProp(1/2)
        pos_x = random.randrange(cobra.getLargura(), 820 -
        cobra.getLargura())
        pos_y = random.randrange(-5000, -500)
        cobra.setCoordenada(pos_x,pos_y)
        self.insere(cobra)
```

3.11 Recursão como mecanismo de iteração

Quando trabalhamos com listas, uma maneira interessante de percorrê-las é usar a recursão, fazendo uma função chamar a si mesma. Pudemos aproveitar este mecanismo quando implementamos o método para a utilização da bomba no jogo, o bombardeia, que recebe um objeto que contém uma lista que precisa ser iterada. A cada recursão, o primeiro elemento da lista (que é uma cobra, por sinal), é removido, até que a lista fique vazia.

```
def bombardeia(self, alvos):
    head = alvos.lista()[0]
    tail = alvos
    tail.remove(head)
    if head.visivel():
        head.danifica()
    if not tail.vazio():
        self.bombardeia(tail)
        tail.insere(head)
    else:
        self._audioexplosao.play()
```

3.12 Delegates

Usamos delegates para passar uma tarefa de uma classe para outra. Isto pode ser observado no método reduzVelocidade da classe Exercito que apenas chama o método reduzVelocidade da classe Cobra.

```
def reduzVelocidade(self):
    for cobra in self._horda:
```

```
            cobra.reduzVelocidade()

    def reduzVelocidade(self):
        self._velocidade = self._velocidade - 1;
```

4 ANÁLISE CRÍTICA DE PYTHON

Tabela 4.1: Avaliação dos critérios e características de Python

Critério ou característica	Nota (0 a 5)
Simplicidade	5
Ortogonalidade	3
Expressividade	5
Adequabilidade e variedade de estruturas de controle	5
Mecanismos de definição de tipos	5
Suporte a abstração de dados e de processos	5
Modelo de tipos	4
Portabilidade	5
Reusabilidade	5
Suporte e documentação	4
Tamanho de código	5
Generalidade	5
Eficiência	2
Custo	4

Em primeiro lugar, Python é uma linguagem interpretada e, portanto, acaba sendo menos eficiente que as compiladas, embora a existência do compilador de bytecode reduza o tempo de carga da execução. Devido a isso, também, Python é bastante portável, executando em diferentes sistemas operacionais. Em segundo, Python é uma linguagem de muito alto nível, o que a deixa com uma sintaxe simples e fácil de aprender, com várias estruturas próximas da linguagem natural. Quando iteramos uma lista de objetos, por exemplo, podemos usar o laço desta forma:

```
for torre in defesas.lista():
    TELADEJOGO.blit(torre.getImagem(), torre.getCoordenada())
```

Mesmo quem não conhece Python ou programação em geral não deverá ter problemas em entender que está se percorrendo torre por torre a lista de defesas. A isso também se deve o tamanho diminuto dos códigos, já que é bastante expressiva. O código dos módulos do jogo somado não passa de 661 linhas, incluindo os comentários.

A orientação a objetos da linguagem, combinada com algumas características funcionais, também aumenta o poder de expressão da mesma e facilita o uso e reuso de estruturas mais adequadas para diferentes tipos de problemas. Por falar em tipos, o sistema de tipagem

dinâmica é ao mesmo tempo prático e perigoso. Temos mais liberdade para lidar com nossas variáveis e funções, mas ao mesmo tempo a compreensão do código é comprometida por causa da falta de explicitação dos tipos. Nem sempre é trivial olhar para a definição de um método e compreender o que ele recebe como parâmetros, exigindo uma maior atenção do programador.

Voltando à orientação a objetos, ela permite o encapsulamento típico de outras linguagens, mas sem uma proteção verdadeira. Um atributo protegido com "_"como primeiro caractere ainda assim poderá ser acessado de fora da sua classe e classes filhas, demandando disciplina de quem estiver a utilizando. Por isso mesmo é recomendável documentar o código. Em Python, pode-se comentar uma linha de código com o caractere "#". A divisão de blocos através da indentação é interessante por obrigar um fluxo mais organizado, elegante e legível do código. Os programas podem ser documentados com técnicas de Engenharia de Software tradicionais, como diagramas UML.

Apesar destes problemas aparentes, Python mostrou-se uma boa linguagem durante a implementação do Pytower Defense, oferecendo ferramentas úteis e fáceis de programar e aprender.

5 Conclusão

Com este trabalho tivemos a oportunidade não apenas de colocar em prática conceitos vistos em aula como também de analisá-los criticamente em uma linguagem diferente das que estávamos habituados. A implementação bem sucedida do jogo foi facilitada pela aplicação de algumas das características estudadas e construções de Python, mas também enfrentamos problemas ao tentar adicionar todos os requisitos solicitados, pois alguns não são totalmente adequados para o estilo de programação da linguagem ou para as funcionalidades do programa.

Um desafio foi combinar características funcionais e orientação a objetos. Passar como parâmetros para um "filter"um atributo e uma lista de objetos não é tão trivial quanto parece, e mesmo utilizar recursão ou "map"para manipular listas acaba sendo trabalhoso por conta de outras estruturas como o "for"mais adequadas para as funcionalidades do jogo. É, portanto, importante que se conheça bem as opções disponíveis de forma a aproveitá-las de fato na construção dos programas de forma que contribuam positivamente.

Referências

[1] http://en.wikipedia.org/wiki/Tower_defense

[2] http://www.python.org/download/releases/3.0/

[3] http://www.pygame.org/news.html

[4] http://cx-freeze.sourceforge.net/

[5] https://code.google.com/p/pyscripter/

[6] http://empiresandallies.wikia.com/wiki/Black_Obelisk

[7] http://emperor.heavengames.com/gameinfo/buildings/stonewall

[8] http://www.photoscape.org/ps/main/index.php

[9] Albert Sweigart *Making Games with Python & Pygame* 2012.

[10] http://www.ibm.com/developerworks/library/l-prog/

[11] http://osdir.com/ml/python.brasil/2003-03/msg00019.html

[12] http://docs.python.org/2/tutorial/controlflow.html#SECTION006700000000000000000

[13] http://zetcode.com/lang/python/

[14] http://pythonhelp.wordpress.com/2012/05/13/map-reduce-filter-e-lambda/

[15] http://www.async.com.br/projects/python/pnp/node41.html

[16] http://www.geeksbr.com/2012/10/python-gerar-executavel-exe-com-cxfreeze.html

[17] http://stackoverflow.com/questions/6725868/generics-templates-in-python

[18] http://mtomassoli.wordpress.com/2012/03/18/currying-in-python/

[19] http://pt.wikipedia.org/wiki/Python

[20] http://pt.wikipedia.org/wiki/Duck_typing