Universidade Federal de Santa Catarina (UFSC)

Campus Reitor João David Ferreira Lima

Departamento de Informática e Estatística

Bacharelado em Ciência da Computação

Disciplina INE 5404 - Programação Orientada a Objetos II - 2022.2

Prof. Jônata Tyska Carvalho e Prof. Mateus Grellert

09/12/2022

Integrantes do Grupo:

André Amaral Rocco

Lucas Cardoso Soares

Vinicius de Campos Pereira

Ismael Coral Hoepers Heinzelmann

Gustavo Konescki Fuhr

Relatório Final Grupo 3 2022.2

Jogo *Bowbound*

Introdução

Para o projeto final da disciplina, o grupo produziu um jogo 2D de plataforma utilizando o PyGame, biblioteca para criação de jogos em Python.

O jogo consiste em um jogador capaz de efetuar disparos com uma crossbow (besta), pular e atravessar obstáculos acertando, concomitantemente, os alvos para que a porta de saída do nível seja liberada. Ademais, os disparos resultam em um knockback (ou seja, o jogador sofre um impulso contrário à direção do tiro), mecânica fundamental para a jogabilidade, que possibilita que o jogador utilize seus disparos para se movimentar pelos níveis além de acertar os alvos.

O desenvolvimento do jogo foi baseado nos conceitos da orientação a objetos, em conjunto com a utilização dos conceitos S.O.L.I.D. e diferentes padrões de projeto.

Principais conceitos e técnicas utilizados

Diversos conceitos lecionados no decorrer da disciplina foram utilizados na construção de *Bowbound*. Dentre esses, alguns dos mais notáveis são:

1. Abstração, Herança, Encapsulamento e Polimorfismo
2. Interface Gráfica com o Usuário (GUI)
3. Tratamento de exceções
4. Padrões de Projeto
    a. Criacionais: Singleton
    b. Estruturais: Data Access Object (DAO)
    c. Comportamentais: States

Abstração, Herança, Encapsulamento e Polimorfismo

A abstração é percebida por todo o projeto. As classes abstraem, dentro do possível, o seu funcionamento por meio dos métodos de interface. Esses métodos buscam mascarar as responsabilidades das classes, ou seja, por mais complexo que seja o funcionamento interno das mesmas, o uso de seus métodos é simples e compreensível.

O projeto utiliza diversas classes abstratas que definem como agem as classes que as herdam. A classe *State*, por exemplo, implementa 3 métodos abstratos que moldam o comportamento dos diferentes estados: os métodos *update_actions()*, *update()*, *render()*. É importante que esses métodos sejam abstratos para que seja forçada a implementação dos mesmos. Isso porque a classe *Game*, que controla estados, baseia-se nesses métodos para o funcionamento do padrão de projetos *States*.

```python
class State(ABC):
    def __init__(self, game, actions_dict: dict):
        self._game = game
        self.__prev_state = None

        self._actions = actions_dict

    @abstractmethod
    def update_actions(self, event):
        # Deve conferir o evento e possivelmente atualizar um atributo de ações
        pass

    @abstractmethod
    def update(self, delta_time) -> None:
        # Irá implementar as atualizações dos elementos do estado
        # É possível entrar em outro estado dentro deste método
        # Deve-se importart o estado desejado, instanciar e chamar o método enter_state()
        pass

    @abstractmethod
    def render(self, display_surface) -> None:
        # Irá renderizar na tela os elementos atualizados em update()
        # Na maioria dos casos, deve-se limpar no início do método usando display_surface.fill((0, 0, 0))
        pass

    def restart_actions(self):
        # Essa função é chamada quando há uma troca estados (para não gerar conflitos) de inputs antigos
        # Reinicia o dicionário salvo em actions para False em todos seus valores
        for key in self._actions:
            self._actions[key] = False
```

No exemplo acima, a herança tem ligação direta com os conceitos de abstração. A herança é utilizada buscando a não repetição do código e para padronizar o comportamento das classes com funcionalidades semelhantes. Já a implementação interna dos métodos que são herdados são abstraídas para aquelas entidades que a utilizam.

Outro exemplo de código que explicita essa aplicação da abstração é o seguinte trecho implementado na classe *Level*:

```python
def __update_arrows(self):
    for arrow in self.__moving_arrows:
        # Aplica o deslocamento na flecha e trata a colisão da flecha
        # Possívelmente irá alterar o atributo stuck da flecha
        arrow.update(self.__level_tiles)
```

No exemplo, o método *update_arrows()* dessa classe chama o método *update()* de todos os diferentes tipos de flechas da lista *moving_arrows*. O nível não tem ciência da implementação interna deste *update()* mas sabe que o método atualiza as flechas.

O encapsulamento foi utilizado em todas as classes ao privar ou proteger atributos e métodos, visando um código seguro. Todos os métodos utilizados apenas internamente pelas classes foram tornados privados e apenas os métodos de interface foram definidos como públicos. Da mesma forma, os atributos das classes estão privados e podem apenas ser acessados (externamente) através de seus *Getters* e *Setters*.

O polimorfismo foi aplicado em diversas partes do código. Por exemplo: a classe abstrata *States* contém o método *restart_actions()* com uma implementação padrão para o seu funcionamento.

```
class State(ABC):
    def restart_actions(self):
        # Essa função é chamada quando há uma troca estados (para não gerar conflitos) de inputs antigos
        # Reinicia o dicionário salvo em actions para False em todos seus valores
        for key in self._actions:
            self._actions[key] = False
```

As classes que herdam *States* podem utilizar essa implementação padrão do método. Entretanto, algumas das classes sobrepõem essa implementação. O estado *LevelPlaying* é o estado que precisa buscar o maior desempenho possível. Por esse motivo, esse estado sobrepõe *restart_actions()* da seguinte maneira:

```
class LevelPlaying(State):
    def restart_actions(self):
        self._actions = {'esc': False, 'restart': False,
                         'up': False, 'down': False, 'left': False, 'right': False,
                         'mouse_left': False, 'mouse_right': False}
```

Interface Gráfica com o Usuário (GUI)

A interface gráfica do jogo é baseada em eventos na qual, dentro dos menus, o usuário poderá escolher o botão que deseja clicar. Ao clicar em um botão, o evento "dispara" e realiza alguma operação ou o jogo procede para outro estado. O próximo estado, possivelmente, também conterá outros botões, repetindo o ciclo.

Tratamento de exceções

O tratamento de exceções foi utilizado, em sua maioria, nas classes DAO. Na classe *LevelDAO*, foram criadas duas exceções, as quais são: *NivelJaExisteExecption*, que levanta a exceção se o usuário tenta adicionar um nível que já existe, e o *NivelNãoExisteExecption*, que levanta a exceção se o usuário tenta remover um nível que não existe.

O tratamento das exceções ocorre durante a utilização dessas classes: a classe responsável importação de mapas (*MapImport*) utiliza o método para a persistência de novos mapas implementado em *LevelDAO* e trata possíveis exceções levantadas durante essa utilização. Se a importação for concluída com sucesso (não houve levantamento de exceção), *MapImport* mostra ao usuário uma mensagem de sucesso. Caso tenha sido levantada alguma exceção durante a importação, *MapImport* mostra ao usuário uma mensagem informando o não sucesso da operação.

Padrões de Projeto

1. Singleton: Precisávamos que houvesse uma classe que tivesse apenas uma instância e que tivesse acesso global a ela, para o carregamento de imagens, fontes e mapas. Por

consequência, criamos a classe *Assets* que herda da classe abstrata *Singleton* para fazer o carregamento desses itens.

2. Data Access Object: Utilizamos o padrão estrutural DAO para implementar operações de persistência e carregamento em arquivos *.json*. As duas classes herdam da classe abstrata *DAO* são: *ScoreDAO* e *LevelDAO*. A classe *ScoreDAO* salva e carrega os scores (nome do jogador, tempo) de cada nível. Já a classe *LevelDAO* salva e remove níveis.
3. State: Para o comportamento do jogo utilizamos o padrão estrutural State. Cada estado do jogo herda a classe *State*. Essas classes devem implementar os métodos abstratos: *update()* (o qual implementa as atualizações para o próximo estado), o método *update_actions()* (que confere e atualiza um atributo de ações) e o método *render()* (que renderiza os elementos do estado na tela). Ademais, utilizamos uma pilha de estados (*state_stack*) para o melhor funcionamento nas transições entre menus e para, principalmente, implementarmos eficientemente o funcionamento da transição entre o estado em que se está jogando o nível e o estado em que o nível está pausado.

Divisão das Responsabilidades

Integrante	**Responsabilidade**
André	Criou a física de movimento do jogador e os algoritmos de colisão. Criou as classes utilizadas como botões e texto. Implementou o estado abstrato *State* que molda o padrão de projeto baseado em estados no projeto, além de ter ajudado na criação das classes de alguns dos estados (estado *ImportMap*, *TitleScreen*, *LevelPlaying*). Fez as texturas, os sprites e as animações utilizados no jogo. Junto com o Gustavo e o Lucas, criou a classe *Timer* que é utilizada como cronômetro no jogo. Junto com o Lucas, desenvolveu a classe abstrata para as flechas (*Arrow*) e os diferentes tipos de flecha (heranças de *Arrow*).
Lucas	Implementou a mecânica das flechas (hold factor, knockback, velocidade mínima e colisão das flechas) e implementação do timer do jogo. Criou o estado *LevelPlaying* que controla os níveis e as transições de níveis. Em conjunto, trabalhou na definição e implementação da base do padrão de projeto States no jogo e participou da criação da tela do menu principal (*TitleScreen*).
Ismael	Implementou a persistência através do *abstractDAO* e *LevelDAO* (para a persistência de níveis). Criou a planilha para geração de mapas e a classe estática *TileMapUtility*, que possui métodos para importação de mapas a partir da planilha de criação de mapas, e também tem como função associar estruturas de construção dos níveis de maneira procedural, permitindo que os mapas possuam suas devidas estruturas adequadas. Também adaptou algumas funções auxiliares para melhorar a usabilidade. Realizou a criação de diversos mapas.
Gustavo	Criou métodos que controlam o funcionamento do nível. Em conjunto com André, desenvolveu o *Timer*, a classe que cronometra o tempo de cada nível. Criou o *ScoreDAO*, classe que faz a persistência dos scores pelo tempo. Criou o protótipo da classe *ScoreController*, classe que faz o controle dos scores entre o *ScoreDAO* e as outras classes. Criou o estado inicial para a digitação do nome do jogador

	(*InputName*). Fez parte da criação de alguns estados como: *LevelSelector, HighScores, ArrowHelpScreen*. Ajudou na criação de níveis.
Vinícius	Implementou as bases da classe abstrata de estados e implementou a classe principal *Game*. Implementou a rotação das imagens da flecha. Participou da criação de níveis. Participou da criação dos elementos de GUI. Criou o estado *stateHelpScreen* (estado que traz as informações de controle do jogo e funcionamento do jogo).

Principais dificuldades encontradas na construção do projeto

Uma das principais dificuldades encontrada ao decorrer do desenvolvimento do projeto teve relação com a maneira na qual, inicialmente, havia sido implementada a classe abstrata *Singleton*. Em um certo período do desenvolvimento, a equipe percebeu uma grande lentidão que estavam ocorrendo as troca dos estados do jogo. Inicialmente imaginávamos que o jogo estava começando a ultrapassar as limitações do PyGame. Entretanto, comparando com outros jogos desenvolvidos utilizando a mesma biblioteca, vimos ser possível desenvolver jogos com as mesmas funcionalidades sem enfrentar problemas de desempenho.

Após uma extensa busca pelo problema, a equipe descobriu o seguinte: a maneira na qual a classe abstrata *Singleton* (que é classe pai de *Assets*) estava sendo implementada atuava corretamente como uma classe singleton: impedia que fosse criada uma nova instância de qualquer uma de suas heranças fosse criada quando *Assets* era instanciada. Entretanto, da maneira na qual estava sendo implementada, o método *__init__()* das heranças era chamado todas às vezes que a herança era atribuída. Essa chamada, recarregava os recursos do jogo (todas as texturas e imagens) toda a vez que a instância já existente da classe singleton *Assets* fosse chamada, e fazia com que todos os recursos do jogo fossem carregados novamente nas transições entre os estados (pois os estados requisitam as instância de *Assets* quando são criados).

Houve também uma dificuldade inicial do grupo em definir os tipos de relacionamento mais complexos entre algumas classes no diagrama UML. Através da consulta com os professores e da pesquisa baseada nas fontes que foram disponibilizadas, conseguimos perceber os diferentes tipos de relacionamentos que deviam ser definidos no Diagrama de Classes do projeto.