O RISCO DO RABISCO: A JORNADA DAS CORES

Alunos:
Arthur João Lourenço,
Bernardo Borges Sandoval,
Otávio Wada,
Victor do Valle Cunha.
Participação Especial:
André Hanazaki Peroni,
Vicente Tavares Alves Ferreira.

Introdução:

O Risco do Rabisco: a Jornada das cores é um jogo de plataforma, estilo Mario. Conta com 5 fases e um chefe numa fase exclusiva. O Jogo foi programado pelos quatro alunos, porém o projeto como um todo foi feito por seis pessoas. André Hanazaki Peroni, graduando de música na UDESC fez a música que é ouvida durante o jogo, e Vicente Tavares Alves Ferreira, ex graduando de artes visuais na UDESC fez a vasta maioria da arte do jogo, além de ajudar em toda a parte criativa. Apesar dessa ajuda externa, nenhum dos dois participantes sabe programar, ou sequer viu o código, a única ajuda foi na direção de arte.

Principais conceitos e técnicas de POO II:

Abstração, Herança, Composição e Agregação:

Os conceitos pilares de OO são utilizados em todos o projeto, herança, em especial é utilizado por praticamente toda a classe. O Exemplo mais notável disso, vem das três classes que atuam como o tronco da árvore de herança, do qual, a maior parte das outras classes herda.

Estático, é a classe que dá corpo a tudo que é visto durante o jogo em si, todos os inimigos, poderes, e até o jogador herdam direta ou indiretamente essa classe.

```python
class Estatico():
    """Base para qualquer objeto fisico no mapa..."""
    def __init__(self, nome: str, x: int, y: int, altura: int, largura: int, imagem: str, cor=(0, 0, 0)):
        self.__nome = nome
        self.__x = x
        self.__y = y
        self.__largura = largura
        self.__altura = altura
        self.__corpo = pygame.Rect(x, y, largura, altura)
        self.__imagem = imagem
        self.__cor = cor
        try:
            self.__sprite = Sprite(imagem)
        except FileNotFoundError:
            self.__sprite = []  # nao possui sprite
```

Movel é uma extensão de Estático, que acrescenta métodos e atributos que permitem movimento

```python
class Movel(Estatico):
    """Expande no estatico e adiciona a acapacidade de mexer..."""
    def __init__(self, nome: str, x: int, y: int, altura: int, largura: int, limite_vel: int, imagem: str,
                 cor=(0, 0, 0)):
        super().__init__(nome, x, y, altura, largura, imagem, cor)
        self.escala_tempo = 1.0
        self.__velx = 0
        self.__vely = 0
        self.__limite_vel = limite_vel
        self.__face = 1
```

Entidade herda móveis, e garante propriedades como vida, dano por contato e animação. Ela é majoritariamente usada para criação de inimigos.

```python
class Entidade(Movel):
    "Expande no movel, adicionando dano de contato e animacoes"
    def __init__(self, nome: str, x: int, y: int, altura: int, largura: int, limiteVel: int, vida: int,
                 dano_contato: int, imagem: str, cor, frames: int, fogo = False):
        super().__init__(nome, x, y, altura, largura, limiteVel, imagem, cor)
        self.__vida = vida
        self.__dano_contato = dano_contato
        self.__a_prova_de_fogo = fogo
        self.__frames = frames
```

Polimorfia é o segundo pilar mais usado, uma vez que a maioria das classes que herdam móvel sobre escrevem o funcionamento do método mover, além de algumas alterarem a de atualização.

Padrões de Projeto:

Decorator:

Usamos o Padrão Decorator para facilitar a administração dos mapas, por exemplo. O Decorador "instanciável" salva classes com o Decorator numa lista, que futuramente é usada para instanciar objetos a partir de um dicionário que contém os mapas do jogo. Um método em Mapa consegue usar as próprias classes da lista e criar os objetos usados.

```python
#Decorator que indica o que a classe pode ser instanciada no mapa
def instanciavel(classe):
    classes_instanciaveis.append(classe)
    return classe
```

```python
@instanciavel
class Atirador(Inimigo):...

@instanciavel
class Saltante(Inimigo):...

@instanciavel
class Gelatina(Inimigo):...
```

Template:

O Padrão Template é usado no método atualizar de Movel. chama os métodos mover e renderizar, além de realizar outras execuções. A maioria das classes que herdam Movel tem sua própria definição de mover, mas não de renderizar, e são atualizadas pelos métodos atuais da classe mãe.

```
def atualizar(self, tela, mapa, dimensoes_tela):
    "Governa movimento, assim como seu movimento em tempo distorcido"
    if self.escala_tempo != mapa.escala_tempo:
        self.escala_tempo += max(min(mapa.escala_tempo - self.escala_tempo, 0.05), -0.05)
    self.mover(dimensoes_tela, mapa)
    self.corpo = pygame.Rect(self.x, self.y, self.largura, self.altura)
    if mapa.campo_visivel.colliderect(self.corpo):
        self.renderizar(tela, mapa)
    return False
```

DAO:

A classe Dao foi utilizada no jogo para implementar persistência. No jogo, é possível ter múltiplos saves, que são persistentes, e ter diferentes configurações, como volume de música e resolução da tela, que também são persistentes. A leitura dos mapas vem de um arquivo que sempre é lido pela DAO ao iniciar o jogo.

```
class DAO:
    """Classe responsavel por da
    def __init__(self):
        self.carregar_configs()
        self.carregar_saves()
        self.carregar_mapas()
```

Divisão de Responsabilidades:

As responsabilidades se revezaram entre tarefas bem definidas para cada um dos integrantes do grupo e a liberdade de cada um para implementar o que era possível dentro da sua semana, o que fez com que muitas tarefas fossem começadas por um participante é finalizada e/ou corrigida por outro. Para isso, era postado em grupo do telegram o progresso do jogo em termos de implementação e correção de problemas que surgiam durante as semanas.

Porém, destaca-se algumas tarefas realizadas por cada um dos integrantes.

Nome	**Responsabilidade**
Arthur João Lourenço	Implementação da maior parte dos coletáveis, funcionamento da Paleta (armazenamento de poder extra), física do jogador, criação do inimigo atirador e dos projéteis de todos os inimigos que atiram fogo. Construção do rei das cores (ênfase nos punhos), elementos do HUD, poder invisível (azul) e design da fase 5.

Bernardo Borges Sandoval	Implementação do método colisão, sprites, maioria dos inimigos do jogo (bolota, espinhento, saltante, gelatina). Coleta de poder, parte da construção do rei das cores (ênfase na troca de fases), poder do dash (vermelho) design das fases 1, 2, 3 e 4.
Otávio Wada	Construção de todas as interfaces gráficas do menu, persistência, mecânicas de manipulação temporal, poder temporal (roxo), e de aceleração (verde) [não utilizado], finalização do poder de fogo (laranja).
Victor do Valle Cunha	Implementação da música, do dano entre jogador e bolota, da classe chão, inicio do poder vermelho, implementação do inimigo voador [não utilizado] e do poder dos clones (marrom) [não utilizado].

Principais dificuldades encontradas:

Uma das principais dificuldades do projeto foi seguir os princípios S.O.L.I.D. em algumas ocasiões. A função de colisão passou por três iterações diferentes. Na primeira versão o acoplamento era enorme, o jogador verificava o tipo do objeto que colidia para administrar o que deveria acontecer. A segunda era menos acoplada mas dependia de uma estrutura de dados esquisita e a administração da colisão era dividida entre a classe colidida e a que colide. Na terceira interação, todas as entidades receberam dois novos métodos. Uma para quando ela é colidida pelo jogador e outra quando é colidida por qualquer outra coisa. Assim, uma vez que há uma colisão a entidade simplesmente chama esse método.

Um dos problemas com o código é o uso de encapsulamento próprio e respeito a atributos privados. Muitos atributos privados simplesmente tem o método get e set sem filtro, o que torna aquele atributo público na prática. Esse não foi o caso com todos os atributos, mas definitivamente com mais do que gostaríamos.

Em algumas fases do jogo, o jogo roda mais lentamente e o fps diminui temporariamente, não conseguimos encontrar exatamente a razão pela qual isso acontece, apesar de que otimizamos o carregamento dos mapas e tentamos reduzir o *garbage collection* efetivamente apagando os dados inimigos e projéteis quando morriam ou colidiram.

Houve um problema com o acesso de arquivos. O Pycharm buscava arquivos a partir do mesmo diretório dos roteiros, enquanto o Vscode buscava a partir da pasta principal do projeto. Por causa disso, a pasta com sprites, e arquivos de saves, mapas, música, etc… aparecem múltiplas vezes no projeto.

Absolutamente tudo do rei das cores foi implementado nas últimas 48 horas. Não planejamos colocá-lo, mas o jogo estava pronto e ficamos animados. A luta ficou completa, porém não é intuitiva, e o jogador pode não saber o que fazer, faltou polimento, Há uma manual na pasta “extra” junto com o arquivo do jogo, explicando o que deve ser feito.

Apesar de tudo, o jogo fluiu bem, e houve pouca coisa que queríamos implementar e não conseguimos, o grupo ficou extremamente satisfeito com o resultado.