MASTER INFORMATIQUE

Projet Génie logiciel : Tower Defense

BOULLI Marouan: 22311680,
ESSAM EDWAR AZIZ Steven: 22309059

23 janvier 2024

Table of Contents :

1. Introduction

1.1 Objectif

L'objectif de ce projet est de mettre en application des bonnes pratiques de génie logiciel à travers la réalisation d'un jeu de Tower Defense.

Pour ce faire, des contraintes liées au développement du jeu sont imposées :

- Mettre en place un motif d'architecture logicielle **Modèle Vue Contrôleur** (MVC).
- Inclure des designs patterns (au choix). Nous concernant, nous avons implémenté les design patterns suivant : **Singleton**, **Command** et **Factory**.
- Faire en sorte que l'installation du jeu soit la plus simple possible et fonctionne sur toutes les machines.

Le choix du langage de programmation étant libre, nous choisissons de programmer le jeu en **Python** puisque c'est un langage que nous maîtrisons plus que les autres et qu'il possède un framework nommé **Pygame** facilitant la gestion de l'interface graphique du jeu.

Le jeu est contenu dans un package qui s'installe facilement avec la commande **pip**.

1.2 Tower Defense

Un jeu de tower defense est un genre de jeu vidéo stratégique où l'objectif principal est de défendre un territoire ou une possession en empêchant les ennemis d'atteindre un point spécifique sur la carte. Cela se fait généralement en construisant une variété de structures défensives, souvent appelées "tours".

Le joueur doit stratégiquement placer ces tours sur la carte pour intercepter efficacement les vagues d'ennemis qui suivent un chemin prédéterminé.

Le défi dans ces jeux vient de la gestion des ressources et de la planification stratégique. Les joueurs doivent décider où placer les tours et quand les améliorer, tout en tenant compte du budget limité et des caractéristiques uniques des ennemis.

Avec chaque vague d'ennemis vaincue, le joueur gagne des ressources qui peuvent être utilisées pour construire ou améliorer des tours, rendant le jeu progressivement plus complexe et stimulant. L'objectif est de survivre à autant de vagues que possible, chaque vague étant généralement plus difficile que la précédente.

2. Le jeu

2.1 Installation

La procédure d'installation est relativement simple (une commande à exécuter). Elle est décrite dans le README du dépôt git : https://github.com/Marouan-git/tower_defense.

Le seul prérequis d'installation est d'avoir installé **python** sur sa machine avec le gestionnaire de paquets python **pip** (en général, l'installation de python inclut pip).

2.2 Méthode de packaging

Pour packager et distribuer le jeu de tower defense, la méthode utilisée implique l'utilisation des fichiers pyproject.toml, setup.cfg, et setup.py, en conformité avec les standards modernes de Python pour le packaging.

Le fichier **pyproject.toml** spécifie les outils et les dépendances nécessaires pour la construction du package, ici en utilisant setuptools.

```
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
```

setup.cfg contient les métadonnées du projet telles que le nom, la version, la description, et les dépendances spécifiques (comme Pygame, cf [options]), ainsi que des informations sur les données de package comme les images et les fichiers audio (cf [options.package_data]).

```
[metadata]
name = tower
version = 1.0
description = Tower defense game
long_description = Software engineering project
license = MIT
classifiers =
    Programming Language :: Python :: 3

[options]
zip_safe = false
packages = find:
```

```
install_requires =
    pygame==2.*

[options.package_data]
tower_defense.assets.images = *.png
tower_defense.assets.audio = *.wav
tower.assets.levels = *

[options.entry_points]
console_scripts =
    tower_defense = tower_defense.main:main
```

Enfin, **setup.py** est un script minimaliste qui appelle simplement setup(), permettant l'utilisation d'outils de construction traditionnels.

```
from setuptools import setup
setup()
```

Ces configurations permettent d'installer le jeu

Les avantages de cette méthode sont multiples. D'abord, elle sépare clairement les préoccupations : pyproject.toml pour le système de build, setup.cfg pour les métadonnées et les configurations spécifiques, rendant la gestion du projet plus organisée.

De plus, cette approche est en alignement avec les recommandations actuelles de la communauté Python, favorisant une meilleure compatibilité et facilité d'utilisation.
En utilisant cette méthode, l'installation du jeu se fait simplement avec une commande pip install, rendant le déploiement du jeu facile et accessible pour les utilisateurs.

2.3 Fonctionnement du jeu

Le jeu comporte 10 niveaux. Au fur et à mesure d'avancement dans le jeu, le nombre d'ennemis et leur vitesse augmentent. Si le joueur parvient à compléter le niveau 10, il gagne et la partie est terminée.

Gestion des ressources

Au début de la partie, le joueur démarre avec 100 points de vie et 550 coins (monnaie du jeu). Le prix d'une tour est fixé à 200 coins, il peut donc en placer 2 au début du jeu.

Pour chaque ennemi éliminé, le joueur gagne 1 coin. Pour chaque niveau complété, il gagne 100 coins.

Chaque fois qu'un ennemi atteint le bout du chemin, le joueur perd un point de vie. Lorsque le nombre de points de vie est égal à 0, la partie est terminée et le joueur a perdu.

Gestion des ennemis

Les ennemis apparaissent un par un et parcourent le chemin qui leur est tracé.

Il existe 4 types d'ennemis qui se différencient par leur vitesse de déplacement et leurs points de vie.
Au début du jeu seul les ennemis faibles sont présents, puis au fur et à mesure de l'avancement des niveaux des ennemis plus forts apparaissent.

Gestion des tours

Les tours peuvent être placées n'importe où en dehors du chemin des ennemis. Elles possèdent une portée limitée.
Nous avons tenté d'implémenter une fonctionnalité permettant d'améliorer ces tours au prix de 100 coins (e.g augmenter le nombre de canons et leur portée), malheureusement elle ne fonctionne pas correctement.

3. Design patterns

Si on considère le modèle MVC comme un design pattern, quatre design patterns ont été implémentés dans ce jeu : **MVC**, **Singleton**, **Command** et **Factory**.

3.1 Modèle MVC

Le modèle MVC est un pattern architectural qui sépare une application en trois composants principaux : Modèle, Vue et Contrôleur.

Le Modèle gère les données et la logique métier.
La Vue présente les données (le modèle) à l'utilisateur.
Le Contrôleur traite les entrées de l'utilisateur, manipule les données du modèle, et met à jour la vue.

Avantages :

- Séparation des préoccupations : Chaque composant a une responsabilité claire, ce qui rend le code plus organisé et plus facile à maintenir.
- Réutilisabilité et évolutivité : Le modèle peut être réutilisé et l'application peut évoluer plus facilement.
- Testabilité : Les composants séparés peuvent être testés indépendamment.

Implémentation dans le jeu

Le code principal du jeu est séparé dans trois dossiers différents :
- Model : gère la logique du jeu.
- View : gère l'affichage.
- Controller : gère les interactions utilisateur.

Modèle (Model):

Le dossier Model contient les sous-dossiers Enemy et Turret, ainsi que le fichier world.py.
Enemy et Turret gèrent les données et la logique des ennemis et des tourelles, respectivement.
world.py est responsable de la gestion de l'état global du jeu, comme la carte du jeu, les statistiques du joueur, etc.

Vue (View):

Le dossier View contient button.py, enemy_view.py, turret_view.py, world_view.py.

Ces fichiers gèrent la représentation visuelle des différents éléments du jeu, comme les ennemis, les tourelles, et l'interface utilisateur.

Contrôleur (Controller):

Le fichier game_controller.py dans le dossier Controller agit comme le contrôleur.
Il interprète les actions de l'utilisateur, modifie les données dans le modèle en conséquence, et met à jour la vue.

3.2 Singleton

Le Singleton est un pattern de conception qui garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

Avantages :

- Contrôle d'accès : Une seule instance signifie un contrôle centralisé sur les ressources.

- Économie de mémoire : Une instance unique évite la redondance des données.
- Facilité d'accès : Un point d'accès global simplifie l'accès à l'instance.

Implémentation dans le jeu

```
class GameConstants:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(GameConstants, cls).__new__(cls)
            # Initialize your constants here
            cls.ROWS = 15
            cls.COLS = 15
            cls.TILE_SIZE = 48
            cls.SIDE_PANEL = 300
            cls.SCREEN_WIDTH = cls.TILE_SIZE * cls.COLS
            cls.SCREEN_HEIGHT = cls.TILE_SIZE * cls.ROWS
            cls.FPS = 60
            cls.HEALTH = 100
            cls.MONEY = 550
            cls.TOTAL_LEVELS = 10

            # Enemy constants
            cls.SPAWN_COOLDOWN = 400

            # Turret constants
            cls.TURRET_LEVELS = 4
            cls.BUY_COST = 200
            cls.UPGRADE_COST = 100
            cls.KILL_REWARD = 1
            cls.LEVEL_COMPLETE_REWARD = 100
            cls.ANIMATION_STEPS = 8
            cls.ANIMATION_DELAY = 15
            cls.DAMAGE = 5
        return cls._instance
```

Dans le fichier constants.py, la classe GameConstants est implémentée en tant que Singleton. Ce pattern garantit qu'une seule instance de GameConstants est créée.

La méthode __new__ est utilisée pour vérifier si _instance est None. Si c'est le cas, une nouvelle instance est créée et initialisée avec diverses constantes du jeu. Si _instance n'est pas None, l'instance existante est retournée.

Avantages :

- Uniformité des données : Assure que toutes les parties du jeu accèdent aux mêmes constantes.
- Économie de ressources : Évite la création de multiples instances de constantes, économisant ainsi de la mémoire.

3.3 Command

Le pattern Command encapsule une demande en tant qu'objet, permettant ainsi de paramétrer les clients avec différentes requêtes, files d'attente ou opérations.

Avantages :

- Découplage : Le client est séparé du destinataire de la demande, permettant une plus grande flexibilité dans la conception.
- Extensibilité : De nouvelles commandes peuvent être ajoutées sans modifier le code existant.
- Annulation et rétablissement : Les commandes peuvent être annulées et restaurées facilement.

Implémentation dans le jeu

```
class Command:
    def execute(self):
        pass
```

```
from commands.commands import Command

class UpgradeTurretCommand(Command):
    def __init__(self, world, turret_group, turret_spritesheets,
selected_turret, constants):
        self.world = world
        self.turret_group = turret_group
        self.turret_spritesheets = turret_spritesheets
        self.selected_turret = selected_turret
        self.constants = constants

    def execute(self):
        # Logic to upgrade a turret
        if self.selected_turret and self.world.money >=
```

```
self.constants.UPGRADE_COST:
            self.selected_turret.upgrade(self.turret_spritesheets)
            self.world.money -= self.constants.UPGRADE_COST
```

Dans UpgradeTurretCommand.py (exemple donné ci-dessus), des classes spécifiques de commandes sont créées en héritant de la classe de base Command et en implémentant la méthode execute.

UpgradeTurretCommand gère l'amélioration des tourelles existantes.

Ces classes encapsulent toute la logique nécessaire pour exécuter ces actions spécifiques, et sont appelées par le contrôleur lorsque nécessaire.

3.4 Factory

Le pattern Factory fournit une interface pour créer des objets dans une super-classe, permettant aux sous-classes de modifier le type d'objets qui seront créés.

Avantages :

- Flexibilité : Les sous-classes peuvent redéfinir quelles classes concrètes sont instanciées.
- Encapsulation de la création : Le code client est séparé de la logique de création des objets, le rendant plus propre.
- Clarté : Les noms de méthodes de factory peuvent rendre le code plus intuitif.

Implémentation dans le jeu

```
from abc import ABC, abstractmethod

class TurretFactory(ABC):
    @abstractmethod
    def create_turret(self, tile_x, tile_y, shot_fx, constants):
        pass
```

```
from Model.Turret.turret import Turret
from Factory.TurretFactory import TurretFactory

class BasicTurretFactory(TurretFactory):
    def create_turret(self, turretSpritesheets, tile_x, tile_y, shot_fx,
```

```
constants):
        return Turret(turretSpritesheets, tile_x, tile_y, shot_fx,
constants)
```

TurretFactory.py définit une classe abstraite TurretFactory avec une méthode abstraite create_turret.

BasicTurretFactory dans create_turret.py est une implémentation concrète de cette fabrique. Elle crée et retourne une instance de Turret avec les paramètres spécifiés.

Avantages :

- Flexibilité : Permet l'extension facile du jeu pour inclure différents types de tourelles sans modifier le code existant.
- Séparation des préoccupations : Isoler la logique de création des tourelles du reste du jeu, rendant les modifications plus simples et le code plus propre.

Références

Tutoriels PyGame :

https://www.inspiredpython.com/course/create-tower-defense-game/make-your-own-tower-defense-game-with-pygame

https://www.youtube.com/watch?v=WRuf9iPAXfM

Design patterns :

https://refactoring.guru/design-patterns/python