BOW MAN

Documentation technique

Auteur : CUREAU Melvin

Date : 22/07/2024

Ce document comporte X pages :

Sommaire

Introduction

Bow Man est un jeu de tir à l'arc développé en **Python** utilisant la bibliothèque **Pygame**. Ce document vise à fournir une description complète de l'architecture du projet, des classes principales, de la gestion des événements, du système de tir, et de l'intelligence artificielle (IA) intégrée dans le jeu. Ce jeu se catégorise comme un jeu stratégique pouvant se jouer à maximum 2 joueurs. Le but du jeu est de toucher l'adversaire en tirant une flèche par l'intermédiaire de son arc sans voir son adversaire. Chaque joueur peut tirer une flèche par tour. Une fois le tir effectué le joueur aura un aperçu de la précision de sa flèche et pourra prendre en compte son résultat dans le but d'être plus précis au tour suivi.

Ce document explique les détails techniques de l'implémentation et de la réalisation du projet en **Python** et utilisant la librairie **PyGame**.

Langage et bibliothèque graphique

Langage de Programmation : Python

Python a été choisi pour ce projet en raison de sa simplicité d'utilisation et de sa lisibilité. Il est particulièrement adapté pour le développement rapide de prototypes et pour les projets nécessitant une maintenance aisée. Python possède également un écosystème riche en bibliothèques, facilitant le développement de jeux vidéo.

Bibliothèque Graphique : Pygame

Pygame a été sélectionné comme bibliothèque graphique en raison de sa compatibilité avec Python et de sa simplicité pour gérer les éléments fondamentaux des jeux vidéo, tels que la gestion des entrées utilisateur, le rendu graphique, et la gestion des sons. Pygame est largement utilisé pour les projets de jeux 2D, ce qui en fait un choix naturel pour ce projet.

Installation

Pour installer et exécuter ce jeu, suivez les étapes ci-dessous :

1. Cloner le dépôt :
2. Accéder au répertoire du projet
3. Installer les dépendances
    a. Pip install -r requirements.txt
4. Lancer le jeu

Architecture du projet

La structure des dossiers du projet est organisée comme suit :

bow-man/

├ **assets/**

│ ├ backgrounds/

│ ├ characters/

│ ├ sounds/

├ **docs/**

├ **src/**

│ ├ **gameLogic/**

│ ├ serverStatus.py

│ ├ archer.py

│ ├ arrow.py

│ ├ obstacle.py

│ ├ AIArcher.py

│ ├ **network/**

│ ├ client.py

│ ├ gameState.py

│ ├ server.py

│ ├— customizationScene.py

│ ├— creditsScene.py

│ ├— mainScene.py

│ ├— gameScene.py

│ ├— pauseScene.py

│ ├— optionsScene.py

│ └— endGameScene.py

Structures de Données

Les principales structures de données utilisées dans le jeu incluent :

- **Archer :** Cette classe modélise les personnages principaux du jeu, avec des attributs tels que la position, la santé, et les méthodes pour tirer des flèches.
- **Flèche (Arrow) :** La classe Flèche stocke les informations relatives à chaque flèche, y compris sa position actuelle, sa vitesse, et sa direction.
- **Obstacle :** Cette classe représente les obstacles dans le jeu, avec des méthodes pour détecter les collisions avec les flèches.
- **IA :** L'IA est modélisée comme une classe séparée, gérant les actions automatiques de l'archer contrôlé par l'ordinateur.

Équation de la Trajectoire Balistique

La trajectoire des flèches dans le jeu est modélisée par les équations de la physique classique pour un projectile en mouvement. L'équation générale de la trajectoire balistique est donnée par :

$$y = x \cdot \tan(\theta) - \frac{g \cdot x^2}{2 \cdot v_0^2 \cdot \cos^2(\theta)}$$

où :

- y est la hauteur du projectile,
- x est la distance horizontale parcourue,
- 0 est l'angle de tir,
- g est l'accélération due à la gravité,
- v_0 est la vitesse initiale du projectile.

Ces équations permettent de simuler la trajectoire réaliste des flèches en fonction de l'angle de tir et de la puissance appliquée par le joueur.

Classes principales

GameScene.py

GameScene est la classe principale qui gère la logique de jeu, l'affichage des éléments graphiques, et la gestion des événements.

1. Présentation Générale

La classe **GameScene** est le cœur du jeu Bow Man. Elle gère la scène principale du jeu, en orchestrant les éléments visuels, les interactions du joueur, la gestion de l'intelligence artificielle (IA), les collisions, et les transitions entre différentes scènes du jeu. Cette classe est conçue pour offrir une expérience de jeu fluide, que ce soit en mode solo, multijoueur, ou contre une IA.

2. Construction (‘__init__’)

Le constructeur initialise les éléments essentiels de la scène de jeu. Cela inclut la création des objets graphiques (archers, obstacle, arrière-plan), la gestion de la caméra, l'initialisation des flèches, ainsi que l'intégration de l'IA et du statut du serveur pour le mode multijoueur.

- **Paramètres :**
    - **screen**: Surface de l’écran.
    - **settings**: Dictionnaire contenant les paramètres de configuration du jeu, tels que le mode de jeu, le style des archers, et la couleur des flèches.
- **Éléments Clés :**
    - **Arrière-plan :** Chargement et redimensionnement de l'image d'arrière-plan en fonction de la résolution de la scène.
    - **Archers :** Création des instances des archers gauche et droit avec positionnement dynamique.
    - **Caméra :** Initialisation des paramètres de la caméra pour suivre les flèches pendant leur vol.
    - **Obstacle :** Création optionnelle d'un obstacle au milieu de la scène, selon les paramètres du jeu.
    - **IA :** Configuration de l'IA si le mode de jeu l'exige.

- **Statut du Serveur :** Initialisation du suivi du statut du serveur en mode multijoueur.

3. Gestion des Événements (‘handle_events’)

La méthode **handle_events** est responsable de la gestion des interactions du joueur via les entrées clavier et souris.

- **Interruption du Jeu :** Le jeu peut être mis en pause ou quitté via la touche **ESC**.
- **Tirs de Flèches :** Le joueur peut tirer des flèches à l'aide de la touche **A**, ajuster la puissance avec les touches **gauche/droite**, et modifier l'angle avec les touches haut/bas.
- **IA :** Après chaque tir du joueur, si le mode IA est activé, l'IA est préparée pour tirer automatiquement.

4. Détection des Collisions (‘check_collisions’)

La méthode **check_collisions** détecte les collisions entre les flèches et les autres éléments du jeu (archers, obstacle). Pour chaque flèche, on vérifie si ses coordonnées se trouvent à l’intérieur des coordonnées de l’archer (autrement dit sa zone de collision).

- **Collision avec les Archers :** Lorsqu'une flèche touche un archer, les points de vie de l'archer sont réduits en fonction des dégâts de la flèche. Une fois la flèche ayant touché un archer, elle est supprimée de la liste **self.arrows**. Si les points de vie d'un archer tombent à zéro, le jeu se termine avec une scène de fin.
- **Collision avec l'Obstacle :** Si un obstacle est présent et qu'une flèche le touche, celle-ci est supprimée de la scène sans causer de dégâts supplémentaires.

```
    def check_collisions(self):
        for arrow in self.arrows:
            # Collision avec l'archer gauche
            if self.archer_left.rect.collidepoint(arrow.x, arrow.y):
                self.archer_left.health -= arrow.damage
                self.arrows.remove(arrow)
                if self.archer_left.health <= 0:
                    end_game_menu = EndGameScene(self.screen, "Archer Gauche")
                    end_game_menu.run()

            # Collision avec l'archer droit
            elif self.archer_right.rect.collidepoint(arrow.x, arrow.y):
                self.archer_right.health -= arrow.damage
                self.arrows.remove(arrow)
                if self.archer_right.health <= 0:
```

```
                    end_game_menu = EndGameScene(self.screen, "Archer Droit")
                    end_game_menu.run()

            # Collision avec l'obstacle
            if self.obstacle and self.obstacle.contains_point(arrow.x,
arrow.y):
                self.arrows.remove(arrow)
```

5. Mise à Jour de la Scène (‘update’)

La méthode **update** est appelée à chaque cycle du jeu pour mettre à jour la position des flèches, vérifier les collisions, et actualiser l'état de l'IA si nécessaire.

- **Caméra :** La position de la caméra est ajustée pour suivre la flèche la plus récemment tirée.
- **Flèches :** Chaque flèche est mise à jour selon son angle et sa vitesse.
- **IA :** L'IA est mise à jour en fonction de l'action du joueur, déclenchant potentiellement un tir automatique.

```
    def update(self):
        if self.arrows:
            arrow = self.arrows[-1]
            target_camera_x = arrow.x - self.width // 2

            if target_camera_x < 0:
                target_camera_x = 0
            elif target_camera_x > self.scene_width - self.width:
                target_camera_x = self.scene_width - self.width

            self.camera_x += (target_camera_x - self.camera_x) * 0.1

        for arrow in self.arrows:
            arrow.update()

        self.check_collisions()

        # Mise à jour de l'IA
        if self.settings["play_mode"] == "IA":
            self.IA.update(self.player_has_shot, self.arrows)
            self.player_has_shot = False
```

6. Affichage (‘draw’)

La méthode **draw** gère le rendu graphique de la scène.

- **Arrière-plan :** L'image d'arrière-plan est dessinée en tenant compte de la position de la caméra.

- **Archers et Flèches :** Les archers et les flèches sont dessinés à leur position actuelle.
- **IA :** Si l'IA est active, elle est également dessinée, y compris ses points de vie.
- **Obstacle :** L'obstacle est dessiné s'il est présent.
- **Interface Utilisateur :** Affichage de la puissance, de l'angle de tir, du statut du serveur en multijoueur, et du chronomètre.

7. Boucle Principale du Jeu (‘run’)

La méthode **run** contient la boucle principale du jeu, qui s'exécute tant que le jeu est en cours.

- **Gestion des Événements :** Appelle **handle_events** pour traiter les entrées utilisateur.
- **Mise à Jour :** Si le jeu n'est pas en pause, la méthode **update** est appelée pour mettre à jour la scène.
- **Rendu Graphique :** La méthode **draw** est ensuite appelée pour afficher les éléments mis à jour.
- **Contrôle de la Fréquence d'Image :** La boucle est cadencée à 30 FPS pour assurer une fluidité de jeu.

```
    def run(self):
        running = True
        while running:
            self.handle_events()

            if not self.paused:
                self.update()
                self.draw()
            else:
                self.pause_menu.draw()
                action = self.pause_menu.handle_events()
                if action == "continue":
                    self.paused = False
                elif action == "options":
                    options_menu = OptionsScene(self.screen)
                    options_menu.run()
                elif action == "main_menu":
                    from mainScene import MainMenu
                    main_menu = MainMenu(self.screen)
                    main_menu.run()
```

```
                    running = False
                elif action == "quit":
                    pygame.quit()
                    sys.exit()
```

8. Conclusion

La classe **GameScene** est une pièce maîtresse du jeu **Bow Man**, intégrant et coordonnant les différents éléments du gameplay pour créer une expérience immersive et engageante. Son architecture modulaire permet une grande flexibilité, permettant de gérer différents modes de jeu, de mettre à jour les éléments du jeu en temps réel, et d'assurer un rendu graphique cohérent et attractif.

Archer.py

Archer représente un archer dans le jeu. La classe Archer représente à la fois son affichage visuel à l'écran et la gestion de sa barre de vie. Elle se concentre sur le rendu de l'archer en tenant compte de la position de la caméra et met à jour l'état visuel de la santé de l'archer au fur et à mesure que celle-ci diminue.

1. Présentation Générale

La classe **Archer** est une abstraction simple d'un personnage dans un environnement de jeu 2D utilisant la bibliothèque **Pygame**. Elle gère l'affichage de l'archer à l'écran en fonction de sa position actuelle et de celle de la caméra. De plus, elle inclut une méthode pour dessiner une barre de vie au-dessus de l'archer, qui reflète la santé actuelle du personnage.

2. Constructeur ('__init__')

Le constructeur initialise les paramètres essentiels pour l'affichage de l'archer à l'écran :

- **image** : Sprite représentant l'archer.
- **x, y** : Coordonnées initiales de l'archer sur l'écran. Elles définissent la position en pixels du coin supérieur gauche de l'archer.
- **screen** : Surface de l'écran

En plus de ces paramètres, l'archer possède une propriété **health** (vie), initialisée à 30 points, qui détermine sa barre de vie.

```
class Archer:
```

```
    def __init__(self, image, x, y, screen):
        self.image = image
        self.rect = self.image.get_rect(topleft=(x, y))
        self.screen = screen
        self.health = 30
```

Dans ce code, rect est un objet Rect de Pygame qui enveloppe l'image de l'archer et permet de manipuler facilement sa position, sa taille, et de gérer les collisions.

3. Affichage de l'Archer (‘draw’)

La méthode **Draw** est responsable de l'affichage de l'archer à l'écran. Elle prend en compte la position de la caméra pour ajuster la position de l'archer et assure également l'affichage de la barre de vie.

```
def draw(self, camera_x):
        # archer
        self.screen.blit(self.image, (self.rect.x - camera_x, self.rect.y))
        # barre de vie
        self.draw_health(camera_x)
```

4. Affichage de la Barre de Vie (‘draw_health’)

La méthode **draw_health** dessine une barre de vie au-dessus de l'archer, indiquant visuellement la santé restante du personnage.

```
def draw_health(self, camera_x):
        # Position de la barre de vie
        health_bar_width = 100
        health_bar_height = 10
        health_bar_x = self.rect.x - camera_x + (self.rect.width -
health_bar_width) // 2
        health_bar_y = self.rect.y - 30

        # fond de la barre de vie
        pygame.draw.rect(self.screen, (0, 0, 0), (health_bar_x, health_bar_y,
health_bar_width, health_bar_height))

        # Calculer du pourcentage de vie restant
        health_percentage = max(self.health / 30.0, 0)

        # barre de vie en couleur
        pygame.draw.rect(self.screen, (255, 0, 0), (health_bar_x,
health_bar_y, health_bar_width * health_percentage, health_bar_height))

        # texte de la vie
        font = pygame.font.Font(None, 24)
```

```
        health_text = font.render(f"HP: {self.health}", True, (255, 255, 255))
        self.screen.blit(health_text, (health_bar_x, health_bar_y - 20))
```

Calcul du pourcentage de vie restante : La proportion de la barre de vie à afficher en rouge est calculée en fonction de la santé actuelle de l'archer par rapport à sa santé maximale (30 points).

Barre de vie rouge : Un rectangle rouge est dessiné au-dessus du rectangle noir, dont la longueur est proportionnelle à la santé restante.

5. Conclusion

La classe **Archer** est une implémentation simple mais efficace pour gérer un personnage dans un jeu. Elle permet de dessiner l'archer à l'écran en tenant compte de la position de la caméra, et fournit un retour visuel sur la santé du personnage via une barre de vie. Cette classe est conçue pour s'intégrer facilement dans un système de jeu plus large, où plusieurs archers ou autres personnages peuvent être gérés de manière similaire, avec des mécanismes de collision, d'animation, et d'interaction avec l'environnement.

AIArcher.py

AIArcher représente un archer contrôlé par l'IA. La classe IA est responsable de la gestion du comportement de l'intelligence artificielle dans le jeu. Elle gère la décision des actions de l'IA, le calcul des paramètres de tir, ainsi que l'ajustement de ces paramètres en fonction des tirs précédents.

1. Introduction

La classe **IA** est conçue pour simuler le comportement d'une intelligence artificielle dans un jeu de tir à l'arc. Cette IA est capable de tirer des flèches en ajustant la puissance et l'angle de tir en fonction des résultats précédents. Elle fonctionne de manière autonome pour offrir un défi au joueur humain, en tentant de viser une cible ou de battre l'adversaire humain.

2. Initialisation (‘__init__’)

Le constructeur initialise l'IA avec les paramètres essentiels pour son fonctionnement dans le jeu. Cela inclut l'archer contrôlé par l'IA, l'écran de jeu, l'obstacle, la liste des flèches, et le tour de l'IA (gauche ou droite).

Paramètres :

- **archer**: Instance de la classe Archer, représentant l'archer contrôlé par l'IA.
- **screen**: Surface de l’écran.
- **obstacle**: Instance de la classe Obstacle, représentant un obstacle dans le jeu.
- **arrows**: Liste des flèches présentes dans la scène de jeu.
- **turn**: Indicateur du tour, déterminant si l'IA est à gauche ou à droite de l'écran ("left" ou "right").

Éléments Clés :

- **shoot_power**: Puissance de tir initiale, définie par défaut à 20.
- **shoot_angle:** Angle de tir initial, défini par défaut à 45 degrés.
- **last_shot_distance**: Distance entre le dernier tir de l'IA et la cible visée, utilisée pour ajuster les tirs futurs.
- **target_x, target_y**: Coordonnées cibles aléatoires définies au hasard pour que l'IA tente de les atteindre.
- **initial_shot**: Booléen indiquant si l'IA en est à son premier tir, pour décider de la stratégie de tir.

3. Gestion des Actions de l'IA (‘decide_action’)

La méthode **decide_action** est responsable de la prise de décision de l'IA à chaque tour. Selon que c'est le premier tir ou non, elle décide si l'IA doit effectuer un tir aléatoire ou ajuster ses paramètres de tir en fonction des résultats précédents.

Fonctionnement :

- Si c'est le premier tir (initial_shot est True), l'IA effectue un tir aléatoire via shoot_random.
- Si ce n'est pas le premier tir, l'IA ajuste ses paramètres en utilisant adjust_shooting_parameters, puis effectue un tir.

```
    def decide_action(self):
```

```
        if self.initial_shot:
            self.shoot_random()
            self.initial_shot = False
        else:
            self.adjust_shooting_parameters()
            self.shoot()
```

4. Tir Aléatoire (‘shoot_random’)

La méthode **shoot_random** permet à l'IA d'effectuer un tir avec des paramètres (puissance et angle) choisis aléatoirement. Ce tir est utilisé principalement pour le premier tir de l'IA ou dans des situations où aucun ajustement préalable n'est disponible.

Détails :

- La puissance est choisie aléatoirement entre 10 et 50.
- L'angle est choisi aléatoirement entre 0 et 90 degrés.

```
    def shoot_random(self):
        # Tir aléatoire sans ajustement
        self.shoot_power = random.randint(10, 50)
        self.shoot_angle = random.uniform(0, 90)
        self.shoot()
```

5. Ajustement des Paramètres de Tir (‘adjust_shooting_parameters’)

Cette méthode ajuste la puissance et l'angle de tir de l'IA en fonction de la distance de la dernière flèche par rapport à la cible. Cela permet à l'IA d'améliorer sa précision à chaque tir successif.

Logique d'Ajustement :

- Si la dernière flèche a atterri à moins de 100 unités de la cible, l'IA augmente la puissance et l'angle de tir.
- Si la flèche a atterri à plus de 200 unités de la cible, la puissance et l'angle de tir sont diminués.

```
    def adjust_shooting_parameters(self):
        if self.last_shot_distance is not None:
            # Ajuster la puissance et l'angle en fonction de la distance du
dernier tir
            if self.last_shot_distance < 100:
                self.shoot_power = min(self.shoot_power + 5, 100)
            elif self.last_shot_distance > 200:
                self.shoot_power = max(self.shoot_power - 5, 10)

            if self.last_shot_distance < 100:
```

```
                self.shoot_angle = min(self.shoot_angle + 5, 90)
            elif self.last_shot_distance > 200:
                self.shoot_angle = max(self.shoot_angle - 5, 0)
```

6. Exécution d'un Tir (‘shoot’)

La méthode **shoot** calcule la trajectoire d'une flèche en fonction de la puissance et de l'angle actuels de l'IA, puis la fait tirer par l'archer contrôlé par l'IA. Elle calcule également la distance entre le point d'impact et la cible définie pour l'IA.

Détails :

- Conversion de l'angle en radians pour calculer les composantes de vitesse en x et y.
- Création d'une nouvelle flèche (Arrow) avec les vitesses calculées, tirée dans la direction appropriée selon le tour de l'IA (left ou right).
- Mise à jour de la distance du dernier tir (last_shot_distance) pour les ajustements futurs.

```
    def shoot(self):
        # Tirer une flèche avec les paramètres calculés
        angle_radians = math.radians(self.shoot_angle)
        x_velocity = self.shoot_power * math.cos(angle_radians)
        y_velocity = -self.shoot_power * math.sin(angle_radians)

        if self.turn == 'left':
            new_arrow = Arrow(self.screen, self.archer.rect.right,
self.archer.rect.centery, x_velocity, y_velocity)
        else:
            new_arrow = Arrow(self.screen, self.archer.rect.left,
self.archer.rect.centery, -x_velocity, y_velocity)

        self.arrows.append(new_arrow)

        # Mettre à jour la distance du dernier tir
        self.last_shot_distance = math.hypot(new_arrow.x - self.target_x,
new_arrow.y - self.target_y)
```

7. Mise à Jour de l'IA (‘update’)

La méthode **update** est appelée à chaque cycle du jeu pour permettre à l'IA de décider de son action pour ce tour. Elle encapsule la logique de prise de décision et l'exécution des tirs.

Fonctionnement :

- À chaque mise à jour, decide_action est appelée pour déterminer et exécuter l'action de l'IA.

8. Conclusion

La classe **IA** est une composante clé du jeu, permettant d'introduire un adversaire automatisé capable de s'adapter et d'ajuster ses tirs en fonction des résultats précédents. Grâce à ses mécanismes d'ajustement de la puissance et de l'angle de tir, l'IA peut progressivement affiner sa précision, offrant un défi de plus en plus difficile au joueur humain.

Arrow.py

Arrow représente une flèche dans le jeu. La classe Arrow est essentielle pour représenter les flèches dans le jeu en modélisant leur trajectoire, leur apparence visuelle et leur interaction avec d'autres éléments tels que le sol et les obstacles. Elle simule le mouvement d'une flèche en prenant en compte la gravité et les conditions initiales de vitesse.

1.Présentation Générale

La classe **Arrow** encapsule le comportement d'une flèche lancée dans un espace 2D. Elle gère la physique simple du mouvement balistique sous l'influence de la gravité et met à jour la position de la flèche à chaque frame. En outre, elle permet de dessiner la flèche avec un rendu réalise, y compris la pointe de la flèche.

2.Constructeur ('__init__')

Le constructeur initialise les paramètres essentiels pour la simulation et le rendu d'une flèche :

screen : surface de l'écran
start_x, start_y : coordonées intiales de la flèche au moment de son lancement
x_velocity, y_velocity : composantes horizontale et verticale de la vitesse de la flèche
color : couleur du corps de la flèche
tip_color : couleur de la pointe de la flèche

La flèche est modélisée avec une longueur fixe (30 pixels) et est influencée par la gravité. La position de la flèche est contrainte par un niveau de sol fixe, et la flèche s'arrête lorsqu'elle touche le sol.

3.Mise à jour de la position (‘update’)

La méthode **Update** est responsable de la mise à jour de la position et de l’orientation de la flèche à chaque frame. Elle prend en compte la vitesse actuelle de la flèche et l’influence de la gravité pour déterminer la nouvelle position :

```
def update(self):

        self.x += self.x_velocity
        self.y_velocity += self.gravity
        self.y += self.y_velocity

        # Mise à jour du rectangle pour suivre la position de la flèche
        self.rect.x = self.x
        self.rect.y = self.y

        if self.y >= self.ground_level:
            self.y = self.ground_level
            self.y_velocity = 0
            self.x_velocity = 0

        # Calcul de l'angle de la flèche par rapport à l'horizontale
        self.angle = math.degrees(math.atan2(self.y_velocity,
self.x_velocity))

        # Ajustement de la position du rect en fonction de l'angle et de la
longueur de la flèche
        self.rect = pygame.Rect(self.x, self.y, self.length, 3)
        self.rect.topleft = (self.x - self.length / 2, self.y - 1.5)  #
Ajustement pour centrer le rect
```

Cette méthode effectue plusieurs opérations :

- **Mise à jour de la position** : La flèche se déplace selon les vitesses x_velocity et y_velocity, ajustées par l'accélération due à la gravité.
- **Gestion des collisions avec le sol** : Si la flèche atteint le niveau du sol (ground_level), elle s'arrête.
- **Calcul de l'angle** : L'angle de la flèche est calculé par rapport à l'horizontale en utilisant la fonction atan2, ce qui permet de dessiner la flèche avec la bonne orientation.
- **Mise à jour du rectangle de collision** : Le rect de la flèche est mis à jour pour correspondre à sa nouvelle position et orientation.

4. Dessin de la flèche (‘draw’)

La méthode draw s'occupe du rendu visuel de la flèche à l'écran, en tenant compte de la position de la caméra pour ajuster la position de dessin. Elle dessine la flèche comme une ligne avec une pointe visible, ce qui ajoute un effet visuel réaliste :

```
def draw(self, camera_x):

        # Calcul des coordonnées de l'extrémité de la flèche
        end_x = self.x - self.length * math.cos(math.radians(self.angle))
        end_y = self.y - self.length * math.sin(math.radians(self.angle))

        # ligne principale de la flèche
        pygame.draw.line(self.screen, self.color, (int(self.x - camera_x),
int(self.y)), (int(end_x - camera_x), int(end_y)), 3)

        # coordonnées pour dessiner la pointe de la flèche
        arrow_tip_length = 10  # Longueur de la pointe de la flèche
        arrow_tip_angle = math.radians(30)  # Angle de la pointe de la flèche
par rapport à l'axe de la flèche

        tip1_x = end_x - arrow_tip_length * math.cos(math.radians(self.angle)
+ arrow_tip_angle)
        tip1_y = end_y - arrow_tip_length * math.sin(math.radians(self.angle)
+ arrow_tip_angle)

        tip2_x = end_x - arrow_tip_length * math.cos(math.radians(self.angle)
- arrow_tip_angle)
        tip2_y = end_y - arrow_tip_length * math.sin(math.radians(self.angle)
- arrow_tip_angle)

        # pointe de la flèche
        pygame.draw.polygon(self.screen, self.tip_color, [
            (int(end_x - camera_x), int(end_y)),
            (int(tip1_x - camera_x), int(tip1_y)),
            (int(tip2_x - camera_x), int(tip2_y))
        ])

        # MAJ de la position du rect pour correspondre à la position actuelle
de la flèche
        self.rect.topleft = (self.x - camera_x, self.y)
```

Dans cette méthode :

- **Coordonnées de la flèche** : Les coordonnées de l'extrémité de la flèche sont calculées en fonction de la longueur de la flèche et de son angle.
- **Dessin de la ligne principale** : Une ligne est tracée entre la position actuelle de la flèche et son extrémité calculée.

- **Dessin de la pointe** : Deux segments sont tracés à partir de l'extrémité de la flèche pour former la pointe, en tenant compte d'un angle spécifique pour simuler la forme triangulaire de la pointe.

5. Gestion des Collisions

La classe **Arrow** gère les collisions via la mise à jour du rectangle rect, qui représente la zone de collision de la flèche. Ce rectangle est mis à jour dans chaque appel à update et ajusté pour correspondre à la position actuelle et l'orientation de la flèche. Bien que la gestion des collisions avec des objets comme des obstacles ne soit pas directement implémentée dans cette classe, ce rect peut être utilisé en combinaison avec les méthodes de collision de **Pygame** pour vérifier si la flèche touche un autre objet.

6. Conclusion

La classe **Arrow** combine des éléments de physique simple avec des techniques de rendu pour modéliser le comportement réaliste d'une flèche en vol. Elle assure que la flèche réagit de manière plausible aux forces physiques comme la gravité, tout en offrant des méthodes pour dessiner cette flèche avec précision à l'écran. Cette classe est également conçue pour s'intégrer facilement avec d'autres éléments du jeu, tels que les obstacles et le sol, en utilisant des techniques de gestion de collisions basées sur les rectangles.

Obstacle.py

Obstacle représente un obstacle dans le jeu. La classe Obstacle est une composante clé du système de jeu. En effet elle est responsable de la génération et de la gestion des obstacles dans la scène de jeu. Elle encapsule la logique nécessaire pour créer un obstacle à partir de points générés de manière aléatoire, tout en s'assurant que l'obstacle ait une forme convexe et une position cohérente par rapport au sol de la scène.

1. Présentation générale :

L'objectif de la classe ***Obstacle*** est de créer des obstacles réalistes et dynamiques qui peuvent interagir avec d'autres éléments du jeu, tels que les flèches et les personnages. Les obstacles sont représentés sous forme de polygones irréguliers, générés aléatoirement pour offrir une diversité visuelle et un défi supplémentaire aux joueurs.

2. Constructeur (‘__init__’)

Le constructeur de la classe initialise les paramètres essentiels pour la création d’un obstacle.

- **Screen** : surface de l’écran
- **x, y** : coordonnées du coin supérieur gauche de l’obstacle
- **width, height** : hauteur et largeur de l’obstacle

L’obstacle est toujours positionné de manière à toucher le sol de la scène de jeu. De plus les dimensions minimales de l’obstacle sont fixées à 400 pixels de largeurs et 600 pixels de hauteur.

3. Génération des points aléatoires

L’obstacle est représenté par un polygone constitué de plusieurs points générés aléatoirement. La méthode ***generate_random_points*** crée un certain nombre de points aléatoires (ici 7 points) dans la zone définie par les coordonnées **x ; y ; width et height**

```
def generate_random_points(self):

        num_points = 7  # Nombre de points pour dessiner le rocher
        points = []

        # Génération des points / marges pour assurer une taille minimale
        for _ in range(num_points):
            px = self.x + random.uniform(0, self.width)
            py = self.y + random.uniform(0, self.height)
            points.append((px, py))

        # algorithme de convex hull pour générer une forme convexe
        return self.convex_hull(points)
```

4.Algorithme du Convew Hull

Pour garantir que les points forment une forme convexe (et donc un polygone réaliste et sans enchevêtrement), la méthode ***convex_hull*** est utilisé pour réorganiser les points. L’algorithme choisi ici est le **Grahan Scan**, un algorithme classique pour calculer l’enveloppe convexe d’un ensemble de points en 2D.

L’algorithme commence par trier les points, puis il construit l’enveloppe convexe en parcourant les points pour construire le bas et le haut de l’enveloppe :

```
def convex_hull(self, points):
        # Algorithme de Graham Scan pour trouver le convex hull
```

```
        points = sorted(points)  # Trier les points
        if len(points) <= 1:
            return points

        def cross(o, a, b):
            return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] -
o[0])

        lower = []
        for p in points:
            while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
                lower.pop()
            lower.append(p)

        upper = []
        for p in reversed(points):
            while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
                upper.pop()
            upper.append(p)

        return lower[:-1] + upper[:-1]
```

L'algorithme utilise le produit vectoriel pour déterminer l'orientation des segments de droite formés par les points successifs, assurant ainsi que seuls les points nécessaires pour définir l'enveloppe convexe sont retenus.

5.Méthode de dessin ('draw')

La méthode ***draw*** s'occupe de l'affichage de l'obstacle à l'écran. Elle ajuste d'abord les coordonnées des points en fonction de la position de la caméra, puis dessine le polygone à l'aide de la fonction ***pygame.draw.polygon*** :

```
def draw(self, camera_x):
        # Ajuster les points en fonction de la position de la caméra
        adjusted_points = [(px - camera_x, py) for px, py in self.points]

        # Polygone qui représente le rocher
        pygame.draw.polygon(self.screen, self.color, adjusted_points)
```

6.Vérification des collisions ('contains_point')

Pour déterminer si un point (la pointe de la flèche) est à l'intérieur de l'obstacle, la méthode utilise l'algorithme de **Ray-Casting**. Cet algorithme détermine si un point est à l'intérieur d'un polygone en comptant combien de fois une ligne horizontale partant de

ce point croise les bords du polygone. Un nombre impair de croisements signifie que le point est à l'intérieur :

```
def contains_point(self, x, y):

        # Algorithme de ray-casting pour vérifier si un point est à
l'intérieur du polygone
        num_points = len(self.points)
        inside = False
        p1x, p1y = self.points[0]
        for i in range(num_points + 1):
            p2x, p2y = self.points[i % num_points]
            if y > min(p1y, p2y):
                if y <= max(p1y, p2y):
                    if x <= max(p1x, p2x):
                        if p1y != p2y:
                            xinters = (y - p1y) * (p2x - p1x) / (p2y - p1y) +
p1x
                        if p1x == p2x or x <= xinters:
                            inside = not inside
            p1x, p1y = p2x, p2y
        return inside
```

7. Conclusion

La classe **Obstacle** est une implémentation robuste qui utilise des algorithmes géométriques classiques pour générer des formes d'obstacles réalistes et interagir avec les éléments du jeu. Grâce à l'utilisation d'algorithmes comme le ***Graham Scan*** pour le calcul de l'enveloppe convexe et le ***Ray-Casting*** pour la détection des collisions, cette classe assure que les obstacles sont à la fois visuellement intéressants et fonctionnels dans le cadre du jeu.

PauseScene.py

PauseScene gère l'affichage et la logique du menu de pause.

Méthodes :

- **__init__(self, screen)** : Initialisation de la scène de pause.
- **draw(self)** : Affichage du menu de pause.
- **handle_events(self)** : Gestion des événements du menu de pause.

OptionsScene.py

OptionsScene gère l'affichage et la logique du menu des options.

Méthodes :

- **__init__(self, screen)** : Initialisation de la scène des options.
- **draw(self)** : Affichage du menu des options.
- **handle_events(self)** : Gestion des événements du menu des options.
- **run(self)** : Boucle principale du menu des options.

EndGameScene.py

EndGameScene gère l'affichage et la logique de la scène de fin de jeu.

Méthodes :

- **__init__(self, screen, winner)** : Initialisation de la scène de fin de jeu.
- **draw(self)** : Affichage de la scène de fin de jeu.
- **handle_events(self)** : Gestion des événements de la scène de fin de jeu.
- **run(self)** : Boucle principale de la scène de fin de jeu.

Système de tir

Classe : GameScene

Méthodes : handle_events(self) : Gère les événements utilisateur, y compris le tir du joueur.

Tir du joueur

Lorsque le joueur appuie sur la **touche A**, un angle et une puissance de tir sont utilisés pour créer une flèche.

La direction du tir est déterminée par le tour actuel **(self.turn).**

La flèche est initialisée avec une position, une vitesse **(x_velocity, y_velocity),** et une couleur.

Logique

La flèche tire toujours dans la direction où l'archer est orienté (gauche ou droite).

Après le tir, c'est au tour de l'IA (ou de l'autre joueur si en mode multijoueur ou local).

Si en mode IA, l'IA se prépare à tirer automatiquement après le tir du joueur.

Multijoueur

Le jeu multijoueur est géré via un serveur qui synchronise les actions entre les différents joueurs connectés. Chaque joueur dispose d'une instance de jeu locale, et les informations importantes (telles que les positions des flèches, les tirs, et les collisions) sont transmises au serveur. Le serveur relaye ces informations aux autres joueurs pour maintenir une cohérence de l'état du jeu à travers les différentes sessions. Un protocole de communication spécifique, basé sur TCP/UDP, est utilisé pour minimiser la latence et assurer une expérience de jeu fluide.

Conclusion

Pour résumer, la réalisation de ce projet a été une expérience enrichissante et stimulante. Ce projet m’a permis de mettre en pratique les compétences en programmation que j’ai acquises, tout en découvrant de nouveaux outils de gestion de fichier en Python.

En ce qui concerne l’architecture du projet, j’ai opté pour une structure de code modulaire qui favorise une meilleure organisation et facilite la maintenance, contribuant ainsi à rendre le code plus lisible et simple.

En ce qui concerne les perspectives d’amélioration, il serait intéressant d’explorer la possibilité d’introduire une fonctionnalité de reprise des parties en sauvegardant l’état du jeu dans un fichier .txt, permettant ainsi aux joueurs de reprendre là où ils s’étaient arrêtés.

En fin de compte, je suis satisfait du travail accompli, et je suis convaincu que le jeu Bow Man offre une expérience de jeu à la fois divertissante et stimulante pour les

joueurs de jeux de société. Cette documentation technique fournit une vue d'ensemble complète de la structure et des fonctionnalités du jeu 'Bow Man'. Pour des détails supplémentaires, veuillez-vous référez au code source et aux commentaires incus dans chaque fichier.

Annexes

Ressources

- Pygame documentation : https://www.pygame.org/docs/
- Python Documentation : https://docs.python.org/3/

Références

- X