**BOW MAN**

Documentation technique

**Auteur :** CUREAU Melvin

**Date :** 22/07/2024

Ce document comporte X pages :

* Page 1 : Page de garde ; Sommaire
* Page 2 : Introduction ; Architecture ;
* Page 3 : Choix techniques ; Structure de données
* Page 4 : Structure de données
* Page 5 : Références ; Conclusion

# **Sommaire**

Introduction

Installation

Architecture du Projet

Description des classes principales

GameScene

Archer

AIArcher

Arrow

Obstacle

PauseScene

OptionsScene

EndGameScene

Gestion des événements

Gestion des collisions

Système de tir

Fonctionnement de l’ia

Conclusion

Annexes

Ressources

Références

# **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**.

# **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
   1. 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/

├── **gameLogic**/

│ ├── archer.py

│ ├── arrow.py

│ ├── obstacle.py  
│ ├── serverStatus.py

│ └── AIArcher.py

├── **network**/

│ ├── gameScene.py

│ ├── pauseScene.py

│ ├── optionsScene.py

│ └── endGameScene.py  
├── creditsScene.py  
├── customizationScene.py

├── endGameScene.py

├── gameScene.py

├── optionsScene.py

├── pauseScene.py

├── mainScene.py

# **Description des 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.

### Attributs :

* **screen** : L'écran de jeu.
* **width**, **height** : Dimensions de l'écran.
* **clock** : Horloge pour gérer les FPS.
* **background\_img** : Image de fond du jeu.
* **archer\_left, archer\_right** : Instances des archers (joueur et IA).
* **camera\_x** : Position de la caméra.
* **paused** : État du jeu (en pause ou non).
* **turn** : Tour actuel (gauche ou droite).
* **arrows** : Liste des flèches en jeu.
* **obstacle** : Instance de l'obstacle.

### Méthodes :

* **\_\_init\_\_(self, screen, settings)** : Initialisation de la scène de jeu.
* **handle\_events(self)** : Gestion des événements utilisateur.
* **player\_shoot(self)** : Gestion du tir du joueur.
* **ai\_shoot(self)** : Gestion du tir de l'IA.
* **check\_collisions(self)** : Vérification des collisions entre les flèches et les archers.
* **update(self)** : Mise à jour de l'état du jeu.
* **draw(self)** : Affichage des éléments graphiques.
* **run(self)** : Boucle principale du jeu.

## **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.

### Attributs :

* **image** : Image de l'archer.
* **rect** : Rectangle de collision de l'archer.
* **screen** : Écran de jeu.

shoot\_power, shoot\_angle, previous\_shots : Similaires à ceux de Archer.

### Méthodes :

* **\_\_init\_\_(self, image, x, y, screen)** : Initialisation de l'IA.
* **update(self)** : Mise à jour de l'IA (décision de tir).
* **shoot(self)** : Gestion du tir de l'IA.
* **draw(self, camera\_x)** : Affichage de l'IA.

## **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.

# **Gestion des événements**

La gestion des événements est essentielle pour interagir avec le jeu. Dans gameScene.py, la méthode handle\_events(self) capture et traite les événements comme les pressions de touches, les mouvements de la souris, et les clics.

## Extrait du code : handle\_events

def handle\_events(self):

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit()

            sys.exit()

        elif event.type == pygame.KEYDOWN:

            if event.key == pygame.K\_ESCAPE:

                self.paused = not self.paused

            elif event.key == pygame.K\_a and not self.paused:

                self.player\_shoot()

            elif event.key == pygame.K\_o:

                self.obstacle = Obstacle(self.screen, self.scene\_width // 2 - 100, self.scene\_height - 150, 200, 800)

            elif event.key == pygame.K\_UP:

                self.shoot\_angle += 1

                if self.shoot\_angle > 90:

                    self.shoot\_angle = 90

            elif event.key == pygame.K\_DOWN:

                self.shoot\_angle -= 1

                if self.shoot\_angle < 0:

                    self.shoot\_angle = 0

            elif event.key == pygame.K\_RIGHT:

                self.shoot\_power += 1

                if self.shoot\_power > 100:

                    self.shoot\_power = 100

            elif event.key == pygame.K\_LEFT:

                self.shoot\_power -= 1

                if self.shoot\_power < 1:

                    self.shoot\_power = 1

        elif event.type == pygame.MOUSEBUTTONDOWN:

            if event.button == 1 and not self.paused:

                self.move\_camera\_right = not self.move\_camera\_right

            elif self.paused:

                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()

                    pygame.quit()

                    sys.exit()

                elif action == "quit":

                    pygame.quit()

                    sys.exit()

## Explications:

* Quitter le Jeu : Lorsque l'événement pygame.QUIT est capturé (fermeture de la fenêtre), le jeu se termine proprement.
* Pause : K\_ESCAPE permet de mettre en pause et de reprendre le jeu.
* Tirer une Flèche : K\_a tire une flèche si le jeu n'est pas en pause.
* Créer un Obstacle : K\_o crée un nouvel obstacle.
* Ajuster l'Angle et la Puissance du Tir : Les flèches directionnelles modifient self.shoot\_angle et self.shoot\_power.
* Gestion des Clics de Souris : Permet de bouger la caméra ou de naviguer dans le menu de pause.

# **Gestion des collisions**

La méthode check\_collisions(self) vérifie les collisions entre les flèches et les archers. Cette méthode est cruciale pour déterminer les impacts des flèches et réduire la santé des archers en conséquence.

## **Extrait du code : check\_collisions**

def check\_collisions(self):

    for arrow in self.arrows:

        if arrow.x < self.archer\_left.rect.right and arrow.x > self.archer\_left.rect.left and arrow.y < self.archer\_left.rect.bottom and arrow.y > self.archer\_left.rect.top:

            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()

        elif arrow.x < self.archer\_right.rect.right and arrow.x > self.archer\_right.rect.left and arrow.y < self.archer\_right.rect.bottom and arrow.y > self.archer\_right.rect.top:

            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()

## **Explications:**

* Vérification des Coordonnées : Pour chaque flèche, on vérifie si ses coordonnées se trouvent à l'intérieur des coordonnées de l'archer.
* Réduction de la Santé : Si une flèche touche un archer, sa santé est réduite du montant des dégâts de la flèche.
* Suppression de la Flèche : Une fois la flèche ayant touché un archer, elle est supprimée de la liste self.arrows.
* Fin du Jeu : Si la santé de l'archer atteint 0, la scène de fin de jeu est lancée.

# **Système de tir**

Le système de tir est géré par les méthodes player\_shoot(self) et ai\_shoot(self). Le tir du joueur est déclenché par la touche ‘a’, tandis que le tir de l’IA est déclenché automatiquement selon une logique définie dans le fichier AIArcher.py.

## **Extrait du code player\_shoot**

def player\_shoot(self):

    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\_left.rect.right, self.archer\_left.rect.centery, x\_velocity, y\_velocity)

        self.arrows.append(new\_arrow)

        self.turn = 'right'

    elif self.turn == 'right':

        new\_arrow = Arrow(self.screen, self.archer\_right.rect.left, self.archer\_right.rect.centery, -x\_velocity, y\_velocity)

        self.arrows.append(new\_arrow)

        self.turn = 'left'

## **Explications:**

* Calcul des Composantes de Vitesse : La vitesse en x et y est calculée en utilisant la puissance de tir et l'angle de tir.
* Création de la Flèche : Une nouvelle flèche est créée avec les paramètres calculés.
* Changement de Tour : Le tour change après chaque tir.

## **Extrait de code ai\_shoot :**

def ai\_shoot(self):

    if random.random() < 0.05:  # Taux de tir aléatoire

        self.shoot()

## **Explications:**

* Probabilité de Tir : L'IA tire avec une certaine probabilité à chaque frame.
* Tir de l'IA : La méthode shoot est appelée pour gérer le tir réel.

# **Fonctionnement de l’IA**

L’IA utilise la classe AIArcher pour tirer automatiquement des flèches lorsque c’est son tour. La méthode update(self) de AIArcher décide quand tirer en fonction d’un taux de probabilité défini.

## **Extrait de code AIArcher :**

import random

import math

from gameLogic.arrow import Arrow

class AIArcher:

    def \_\_init\_\_(self, image, x, y, screen):

        self.image = image

        self.rect = self.image.get\_rect(topleft=(x, y))

        self.screen = screen

        self.shoot\_power = 20

        self.shoot\_angle = 45

        self.previous\_shots = []

    def update(self):

        # Tirer un coup aléatoire ou basé sur la logique

        if random.random() < 0.05:  # Taux de tir aléatoire

            self.shoot()

    def shoot(self):

        angle = self.shoot\_angle

        power = self.shoot\_power

        # Tirer un coup

        x\_velocity = power \* math.cos(math.radians(angle))

        y\_velocity = -power \* math.sin(math.radians(angle))

        # Créer une flèche avec ces paramètres

        new\_arrow = Arrow(self.screen, self.rect.right, self.rect.centery, x\_velocity, y\_velocity)

        self.screen.arrows.append(new\_arrow)  # Ajouter la flèche à la liste des flèches du jeu

        # Simuler la vérification si le tir a été trop court ou trop long

        if self.previous\_shots:

            last\_shot = self.previous\_shots[-1]

            if last\_shot['distance'] > 0:  # Juste un exemple, la vraie logique dépendra des collisions

                # Ajuster la précision en fonction des résultats précédents

                if last\_shot['distance'] > 0:

                    self.shoot\_power -= 1

                elif last\_shot['distance'] < 0:

                    self.shoot\_power += 1

        # Conserver les informations du tir

        self.previous\_shots.append({'distance': random.randint(-10, 10)})  # Exemple de distance aléatoire

    def draw(self, camera\_x):

        self.screen.blit(self.image, (self.rect.x - camera\_x, self.rect.y))

## **Explications:**

* Probabilité de Tir : L'IA décide de tirer avec une certaine probabilité à chaque frame.
* Calcul des Composantes de Vitesse : La vitesse en x et y est calculée de manière similaire à celle du joueur.
* Ajustement Basé sur les Tirs Précédents : L'IA ajuste la puissance de tir en fonction des résultats des tirs précédents pour améliorer la précision.
* Stockage des Résultats des Tirs : Les résultats des tirs précédents sont stockés pour référence future.

# **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