BowMasters

Une image contenant capture d’écran

Le contenu généré par l’IA peut être incorrect.

Maël Naudet – FID1

Lausanne

40 périodes

Patrick Chenaux

Table des matières

[1 Spécifications 3](#_Toc192407589)

[1.1 Titre 3](#_Toc192407590)

[1.2 Description 3](#_Toc192407591)

[1.3 Matériel et logiciels à disposition 3](#_Toc192407592)

[1.4 Prérequis 3](#_Toc192407593)

[1.5 Cahier des charges 3](#_Toc192407594)

[1.5.1 Fonctionnalités requises (du point de vue du développeur) 3](#_Toc192407595)

[1.5.2 Fonctionnalités requises (du point de vue de l’utilisateur) 3](#_Toc192407596)

[1.6 Livraison 4](#_Toc192407597)

[1.7 Validation et conditions de réussite 4](#_Toc192407598)

[2 Réalisation 4](#_Toc192407599)

[2.1 Dossier de Réalisation 4](#_Toc192407600)

[2.1.1 Configuration de la fenêtre du jeu 4](#_Toc192407601)

[2.1.2 Gestion des positions 4](#_Toc192407602)

[2.1.3 Gestion des Hitboxes 4](#_Toc192407603)

[2.1.4 Création des joueurs 4](#_Toc192407604)

[2.1.5 Création des tours 5](#_Toc192407605)

[2.1.6 Création de la balle 5](#_Toc192407606)

[2.1.7 Gérer l’angle de la balle 5](#_Toc192407607)

[2.1.8 Gérer la vitesse de la balle 5](#_Toc192407608)

[2.1.9 Gérer les sons 6](#_Toc192407609)

[2.1.10 Logique du jeu 6](#_Toc192407610)

[3 Tests 7](#_Toc192407611)

[3.1 Dossier des tests 7](#_Toc192407612)

[4 Conclusion 7](#_Toc192407613)

[4.1 Bilan des fonctionnalités demandées 7](#_Toc192407614)

[4.2 Bilan de la planification 7](#_Toc192407615)

[4.3 Bilan personnel 7](#_Toc192407616)

[5 Divers 7](#_Toc192407617)

[5.1 Webographie 7](#_Toc192407618)

[6 Annexes 7](#_Toc192407619)

# Spécifications

## Titre

BowMasters – Projet C# pour le module 320

## Description

Dans une application console, 2 joueurs, séparés par 2 grandes tours s’affrontent en se lançant des balles chacun à leur tour. Pour gagner la partie, il suffit de toucher 3 fois le joueur adverse.

## Matériel et logiciels à disposition

* Accès à Internet
* Visual Studio

## Prérequis

* Avoir suivi le module I319
* Suivre le module I320

## Cahier des charges

### Fonctionnalités requises (du point de vue du développeur)

* 2 joueurs
* 2 tours
* Une balle lancée par un joueur
* Une fonctionnalité de choix d’angle
* Une fonctionnalité de choix de puissance de tir
* Un score
* Des sons
* Un diagramme de classe
* Un rapport
* Une documentation sur le code

### Fonctionnalités requises (du point de vue de l’utilisateur)

* Appuyer sur espace pour obtenir un angle
* Maintenir espace pour choisir une puissance de tir

## Livraison

* Le code
* Un diagramme de classe
* Un rapport
* Une documentation sur le code

## Validation et conditions de réussite

* Compréhension du travail
* Possibilité de transmettre le travail à une personne extérieure pour le terminer, le corriger ou le compléter
* Etat de fonctionnement du produit livré

# Réalisation

## Dossier de Réalisation

### Configuration de la fenêtre du jeu

Le projet a démarré avec la création de la fenêtre de jeu et sa modification. Pour cela, j’ai créé une classe Config avec les paramètres de grandeurs de la hauteur et de la largeur de la fenêtre et j’ai créé une méthode statique qui permet de changer de la taille de la fenêtre ainsi que la zone de mémoire tampon d’écran (pour éviter de pouvoir scroller dans la console).

### Gestion des positions

Pour gérer les positions des différents objets que je vais utiliser, j’ai créé une classe *PositionByte* qui a comme attributs la position X et Y de type byte dans l’écran directement.

### Gestion des Hitboxes

Comme certains objets ont besoin d’une Hitbox (zone de collision), j’ai créé une classe « *Hitbox* » qui contient une liste de positions qui forment un rectangle composant la Hitbox de l’objet. Finalement, ce n’est que la classe Player qui l’utilise étant donné que pour gérer la collision de la tour, j’ai directement utilisé la position de ses pièces.

L’explication sur comment j’ai utilisé ces différents attributs sera donnée plus tard.

### Création des joueurs

Les joueurs de la classe *Player* ont plusieurs propriétés comme la gestion de leur vie, leur position dans la grille de jeu, leur modèle et leur hitbox.

Ils peuvent être affichés avec la méthode Display(), peuvent prendre des dégâts avec la méthode TakeDamage(byte damage) et peuvent afficher leurs informations directement au-dessus d’eux avec leur nombre de point de vie, leur score et la barre de progression d’un lancer de balle.

### Création des tours

Pour faire la tour, j’ai d’abord créé les pièces avec la classe *TowerPiece* qui la composent en leur donnant un modèle et une position. J’ai permis de les afficher et de les enlever également. Ensuite, j’ai créé une classe *Tower* qui est en fait une liste de TowerPiece qui peut s’afficher avec une Hitbox et une Position.

### Création de la balle

Pour lancer la balle, il fallait d’abord faire une classe statique *Balistic* qui calcule le mouvement balistique d’un objet sur l’axe x et y en fonction du temps, de la vélocité initiale, de la position initiale de l’objet et de l’angle.

Ensuite, j’ai implémenté la balle en elle-même. La balle possède tous les attributs nécessaires à sa manipulation dans l’espace grâce aux méthodes de la classe *Balistic* donc une position et une vitesse initiale et un angle ainsi qu’une couleur aléatoire définie dès sa création grâce à une méthode dans une classe statique *Custom* qui retourne aléatoirement une couleur affichable dans une console windows.

Pour cela, plusieurs méthodes ont été implémentées comme celle qui permet de mettre à jour la position de la balle en fonction du temps, celle pour afficher la balle et celle pour l’effacer. Tout ça dans des try catch afin de faire en sorte que la balle ne se retrouve pas en dehors des limites de la fenêtre.

### Gérer l’angle de la balle

Pour pouvoir gérer l’angle de la balle, plusieurs points s’affichent et le joueur n’a cas appuyer sur espace pour bloquer un angle. Pour faire cela, j’ai dû parcourir différents angles, 0 à 90 ou 90 à 180 en fonction de la direction du tir, faire en sorte d’afficher un point au bon endroit et dès que l’utilisateur appuie sur espace, le dernier point affiché est bloqué et le dernier angle parcouru est affiché.

### Gérer la vitesse de la balle

Pour gérer la vitesse de la balle, le joueur doit appuyer sur espace pour faire progresser une barre et relâcher pour bloquer une vitesse. La vitesse retournée est dans tous les cas une valeur entre 0 et 50, qu’importe le temps maximum donné en paramètre. Au début, la barre de progression se remplissait toute seule et il suffisait d’appuyer sur espace pour l’arrêter mais ce n’est pas ce qui était demandé ; il faut maintenir la barre espace pour augmenter la vitesse. Pour ce faire, j’ai dû utiliser la méthode GetAsyncKeyState sur la barre espace qui permet de savoir en temps réel si une touche est appuyée afin de ne pas saturer le buffer. Et tant que cette touche est appuyée, le temps maintenu augmente avec un seuil maximum. Une fois la touche relâchée, la vitesse est retournée sous la forme d’un pourcentage entre le temps pressé et le temps maximum multiplié par 50 (comme ça je peux changer le temps maximum et toujours avoir des valeurs entre 0 et 50).

### Gérer les sons

Pour pouvoir faire les sons j’ai utilisé la classe implémentée *SoundPlayer* qui permet de lancer un son en mode console. Au début, j’ai simplement lancé un son quand je voulais qu’il se lance mais un temps de latence posait un problème. Alors, j’ai créé une classe *SoundEffect* qui possède un dictionnaire avec les différents fichiers sons ainsi que le nom qui les définit, une méthode pour pouvoir précharger les différents sons et les stocker dans le dictionnaire et une méthode pour lire le son. Maintenant, il suffit de précharger le son et de le lire afin d’avoir beaucoup moins de temps de latence qu’avant.

### Logique du jeu

La logique de mon jeu est gérée par la classe *BowGame.* Cette classe initialise le jeu, fais tourner la boucle du jeu et affiche le message de fin. Pour créer le jeu, il suffit de lui donner 2 joueurs et 2 tours, les deux gérés par un struct (*Players* et *Towers*) qui permet de ne pas pouvoir lancer un jeu sans le nombre requis d’objets.

#### Initialisation

L’initialisation du jeu affiche les joueurs, les tour et précharge les sons utilisés dans le jeu.

#### Boucle du jeu

La boucle du jeu suit un schéma simple. Le joueur 1 doit jouer alors il donne un angle et une vitesse grâce aux outils expliqués précédemment à sa balle et une méthode qui permet de le faire tout seul implémentée dans cette classe. La balle est lancée, un son de lancer est joué, et traverse le terrain tant que la balle ne touche ni une tour, ni le joueur en face et bien évidemment si elle sort du terrain (la balle peut tout de même passer au-dessus de la hauteur maximum de la fenêtre, elle retombera simplement un peu plus tard).

Quand la balle touche une tour ou un joueur, un son spécifique à l’objet touché est également joué. Et une fois que la balle a terminé sa course, c’est au tour du joueur 2.

Pour pouvoir gérer la collision de la balle sur un objet, des méthodes qui checkent en permanence si la balle touche la tour alliée, la tour ennemie ou encore le joueur ennemi ont été implémentées en utilisant notamment la hitBox des joueurs.

Si la balle touche un joueur, l’un de ses points de vie est retiré est si un des joueurs ne possède plus de point de vie, un son pour signifier qu’un joueur est mort est lancé et la boucle s’arrête.

#### Fin du jeu

Pour le message de fin de jeu, une petite musique est lancée avec un message qui félicite simplement le joueur encore en vie tandis que le perdant est dessiné, à terre.

## Utilisation de l’IA

Je n’ai pas beaucoup utilisé l’IA lors de mon projet. Principalement pour des aspects graphiques comme pour la disposition des points de l’angle de tir qui pour finir n’a même pas marché ou encore comment faire un bonhomme stickman à terre.

J’ai cependant aussi utilisé l’IA dans le but de savoir comment utiliser la méthode GetAsyncKeyState() ou encore pour savoir comment faire en sorte de précharger les sons du jeu où elle m’a proposé d’utiliser un dictionnaire, concept que je connaissais déjà grâce à python et que j’ai décidé d’adopter pour mon jeu.

Je l’ai aussi utilisé quand je devais modifier entièrement mon code pour simplement ajouter une chose qui se répète et ne pas avoir à tout refaire. Je n’ai pas d’exemple dans mon projet mais je suis certain de l’avoir fait.

# Tests

## Dossier des tests

Test de la balistique : Deux tests ont été réalisés afin de savoir si la balistique a bien été implémentée dans mon jeu. Ils consistent à voir que la position X et Y d’un objet lancé à un angle de 45 degrés, à 20 mètres par seconde depuis un certain point (en 0, 0) soit bel et bien correct après 5 secondes.

Les tests étant positifs, j’en conclus que j’ai bien implémenté la balistique dans mon jeu.

# Conclusion

## Bilan personnel

Ce projet m’a permis d’en apprendre bien plus sur comment sont fait les moteurs de jeu avec notamment une petite approche sur la physique dans la création d’un jeu. Mais surtout, il m’a permis de mettre en pratique les différents concepts de la programmation orientée objet appris dans le cadre du cours I320.

# Divers

## Webographie

* Balistique : [Youtube](https://www.youtube.com/watch?v=m4d9cFufv9I&pp=ygUTYmFsaXN0aXF1ZSBwaHlzaXF1ZQ%3D%3D)
* SoundEffect
  + Musique de victoire : [Youtube](https://www.youtube.com/watch?v=lcJH8JtgZoE)
  + Son de mort : [Youtube](https://www.youtube.com/watch?v=glN0W8WogK8)
  + Son de coup pris : [Youtube](https://www.youtube.com/watch?v=RMkKsZL3_uc)
  + Son de lancer : [Youtube](https://www.youtube.com/watch?v=fm3963W8x2I)
  + Son de coup sur la tour : [Youtube](https://www.youtube.com/watch?v=7JHysNDcZVM)

# Annexes

* Code de mon projet
* Diagramme de classe (dans le dossier du code)
* Sons (dans le dossier du code)
* Documentation du code