I320

Jeu de balle

Christopher Ristic

Ristic Christopher – FID1

ETML / 1004 LAUSANNERue de Sébeillon 12 1004 Lausanne

Table des matières

[1 Introduction 3](#_Toc192245842)

[1.1 Objectif 3](#_Toc192245843)

[2 Début du raisonnement POO 3](#_Toc192245844)

[2.1 Classes 3](#_Toc192245845)

[2.2 Interfaces 3](#_Toc192245846)

[2.3 Pattern matching 3](#_Toc192245847)

[2.4 Polymorphisme 4](#_Toc192245848)

[3 Game 4](#_Toc192245849)

[4 Game Manager 4](#_Toc192245850)

[5 player 4](#_Toc192245851)

[6 Building 4](#_Toc192245852)

[6.1 Affichage 4](#_Toc192245853)

[6.2 Collision et dégâts 5](#_Toc192245854)

[7 Ball 5](#_Toc192245855)

[7.1 Affichage 5](#_Toc192245856)

[7.2 Simulation 5](#_Toc192245857)

[8 Audio 5](#_Toc192245858)

[8.1 Autre solution 5](#_Toc192245859)

[9 Animation 6](#_Toc192245860)

[10 Structure du code 7](#_Toc192245861)

[11 Conclusion 7](#_Toc192245862)

[11.1 Conclusion personnelle 8](#_Toc192245863)

[Table des illustrations 8](#_Toc192245864)

# 

# Introduction

## Objectif

Le projet consiste à créer un jeu avec 2 personnages, 2 bâtiments, un score, des points de vies, et une balle qui est lancée par les 2 joueurs.

Si la balle touche le bâtiment adverse, le bâtiment perds un carré et commence à perdre sa forme rectangulaire.

Si la balle touche le joueur il perd un point de vie, et le joueur qui a tiré gagne un point de vie. Les joueurs peuvent tirer la balle en choisissant un angle et la puissance du tir.

Le jeu se termine lorsqu’un bâtiment est complètement détruit ou qu’un joueur n’a plus de point de vie.

La taille de l’écran de la console sera de 150x40.

Le jeu est codé en langage C# sur Visual Studio 2022, .NET version 4.7.2.

# Début du raisonnement POO

L’idée est de représenter chaque objet du jeu avec une classe. Puis des interfaces pour définir des comportements communs aux classes qui en héritent.

## Classes

J’utilise les classes pour représenter les points suivants :

* Joueur
* Bâtiment
* Balle
* Manager du jeu
* L’ensemble du jeu

## Interfaces

Les interfaces vont me permettre de poser des prototypes de méthode. J’utilise les interfaces pour différents comportements commun des objets :

|  |  |
| --- | --- |
| Fonction | Nom du fichier |
| Entrer en collision | ICollidable.cs |
| Être dommageable | IDamageable.cs |
| Être mis à jour | IUpdatable.cs |

## Pattern matching

Depuis C# 7.0, le mot-clé "is**"** permet non seulement de vérifier le type, mais également d'extraire la valeur convertie dans une variable si la vérification est réussie. Si la condition est vraie, la valeur contenue dans **damageData** est convertie en entier et assignée à la variable **damageAmount**.

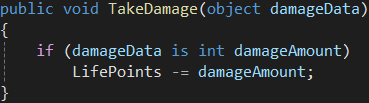


Figure 1: Pattern matching

On trouve ce mot-clé dans certaines conditions que j'ai appliqué notamment un exemple qui se trouve dans la méthode **TakeDamage** voir Figure 1: Pattern matching.

## Polymorphisme

Comme il y a des classes différentes et que je passe par des interfaces pour définir des comportements communs, je vais utiliser des **listes** de la **bibliothèque** **System.Collection.Generic** et ces listes ont un type polymorphe grâce à l’héritage des classes sur les interfaces.

# Game

La classe game va me permettre de contenir l’ensemble du jeu, c’est-à-dire qu’on va pouvoir coordonner le jeu dans son ensemble. On va pouvoir y créer les objets et ils seront atteignable dans le programme principal. Cette classe va aussi gérer le début et fin du jeu, les tours et la simulation de la balle.

# Game Manager

De manière générale un game manager est utilisé pour gérer la logique du jeu et son état. On peut y retrouver des contrôles de niveaux, la gestion de score, sauvegarde et autres événements.

Pour le projet, on va l’utiliser pour gérer :

* La collision entre la balle et un bâtiment
* Le score
* Le système d’angle de tir
* Animation de fin de jeu

# player

Le joueur sera représenté avec la forme suivante :

Il possède plusieurs paramètres tels que : point de vie, score, une position, la couleur de sa balle, et la forme qui le représente.

Si la balle le touche il perd des points de vie et le joueur adverse en gagne.

# Building

## Affichage

Un bâtiment est affiché et composé de plusieurs et même caractère comme celui-ci : ■

Et sa forme complète :

## Collision et dégâts

La classe hérite des 3 Interfaces car on sait que le bâtiment doit pouvoir être touché et détruit par la balle, et son affichage doit être mis à jour après avoir pris des dégâts.

Pour déterminer quelle case a touché la balle, on a une variable de type vecteur en 2D, qui sera la position de la case haut gauche du bâtiment. Un tableau de Boolean représente si la case est intacte ou non. Ainsi on peut calculer la bonne case du bâtiment touché en faisant des décalages.

# Ball

## Affichage

La balle s’affiche sous cette forme-ci :

La méthode qui permet la mise à jour de la position de la balle est accédé depuis une liste d’objet « updatable » que contient Game.cs. L’affichage ce fait avec la méthode Display.

## Simulation

L’équation est basée sur une forme simplifiée de la méthode Euler : Mouvement rectiligne uniformément accéléré (MRUA).

Pour calculer la position de la balle on utilise la formule suivante :

* Position = Position actuelle + Vitesse \* Temps

Concernant la vitesse de la balle on trouve :

* Vitesse verticale = Vitesse actuelle + Gravite \* Temps

La gravité est définie par une constante à 0,5.

# Audio

Pour améliorer l’expérience utilisateur, il y a plusieurs sons dans le jeu. J’utilise la classe **SoundPlayer** qui appartient à la bibliothèque **System.Media**. Elle permet de jouer un fichier audio à la fois. Si une lecture est en cours, elle sera coupée et la suivante sera jouée.

Il faut faire attention de l’utiliser via le mot-clé **using**, qui fait automatiquement appel au garbage collector. Toutefois, si l’on crée un lecteur sans using, il ne faut pas oublier d’utiliser la méthode **Dispose()** avant de créer un nouveau lecteur pour éviter une potentielle fuite de mémoire.

## Autre solution

Dans le cas où l’on voudrait jouer plusieurs sons en même temps, et avoir une musique d’ambiance, il faut utiliser la bibliothèque **WMPLib**. On peut la rajouter dans les références de la manière suviante (voir Figure 2: Windows Media Player) :

**Gestionnaire de références** > **Onglet COM** > **Windows Media Player**

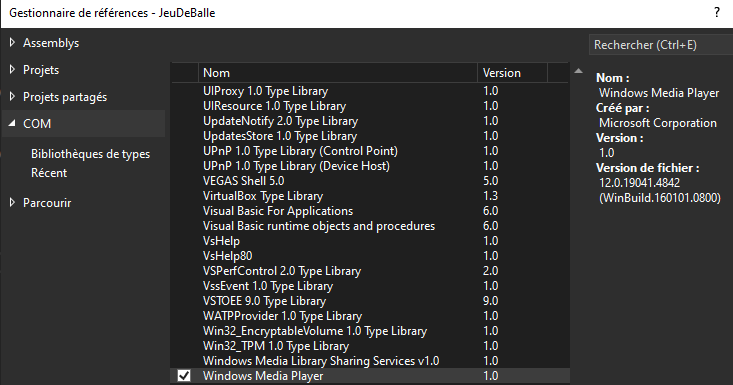


Figure 2: Windows Media Player

# Animation

Je voulais avoir une animation qui est active pendant toute la partie au niveau des points de vie. J’ai fait un essai avec un processus séparé qui s’est avéré peu concluant, notamment du fait que le curseur avait du mal à suivre. Le fait d’avoir l’animation et la balle qui se déplace en même temps, faisait que parfois une des deux actions n’apparaissait pas au bon endroit. (Le code est en commentaire dans le programme principal)

J’ai opté finalement pour un simple affichage pour les points de vie et j’ai crée une animation de fin de jeu avec des étoiles qui défilent comment ceci :

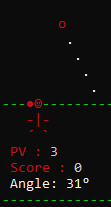


Figure 3: Animation angle de tir



Figure 4: Animation game over

Concernant l’animation pour l’angle de tir, l’animation défile sur les points blancs. Il y a une addition aléatoire qui varie entre -1 et 1 sur l’angle à chaque mise à jour du calcul.

Quant à la puissance du tir, une animation horizontale défile avec un numéro indiquant la puissance. Selon la force, la barre change de couleur.



Figure 5: Animation puissance de tir



Figure 6: Animation (force faible)

# Test unitaire

J’ai créé un test unitaire pour la classe **Building** nommé **UnitTestBuilding**, qui me permet de tester la méthode **DestroyBlock.** Comme ma classe est visible qu’en interne du projet dû au mot-clé **internal**, je dois ajouter ceci **avant le namespace**:

[assembly: InternalsVisibleTo("UnitTestBuilding")]

De plus, il faut importer la bibliothèque **System.Runtime.CompilerServices** pour pouvoir utiliser le constructeur InternalsVisibleTo.

Avant de pouvoir utiliser la classe, il faut ajouter le projet du jeu de balle dans les références du test unitaire en cochant le projet.

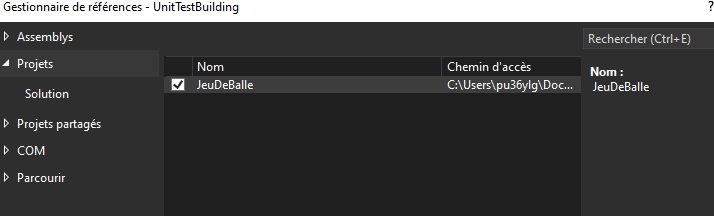


Figure 7: Cocher le projet (référence)

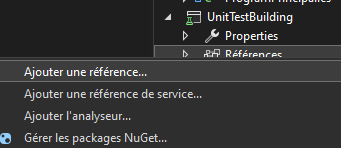


Figure 8: Ajouter le projet (références)

Le test consiste à savoir si j’ai une valeur négative dans les coordonnées de la manière suivante :   
Assert.ThrowsException<ArgumentOutOfRangeException>(()=>building.DestroyBlock(-1, -1));

Comme j’ai passé des valeurs négative dans les paramètres, mon exception est levée.

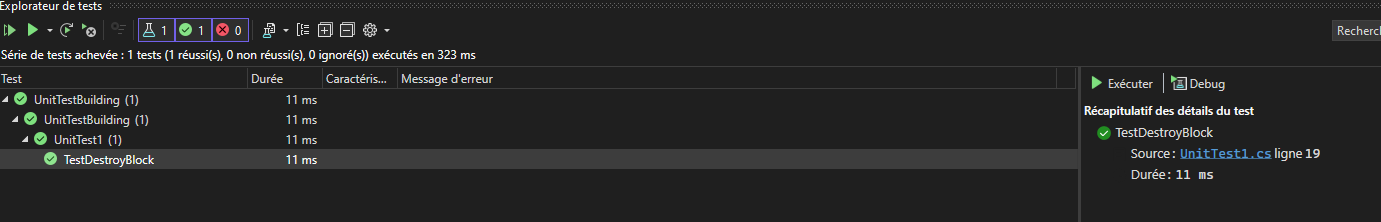


Figure 9: Test unitaire, exception levée

# Structure du code

La structure des interfaces  :

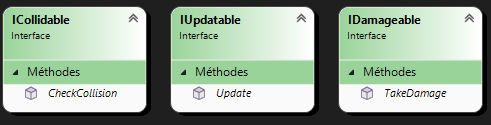


Figure 10: Structure des interfaces

Et les structure des classes :

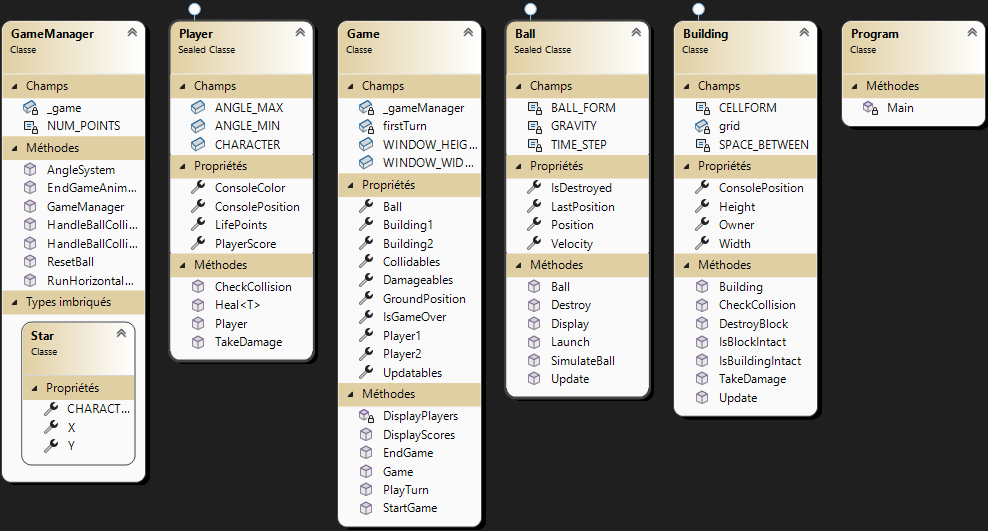


Figure 11: Structure des classes

# Conclusion

Ce projet de jeu en C# est une tout à fait dans le sujet de la programmation orientée objet et de l’application de concepts avancés tels que le polymorphisme, le pattern matching et la gestion des interfaces. En structurant le jeu à travers des classes bien définies et en utilisant des interfaces pour standardiser les comportements communs, le code gagne en modularité, en lisibilité et en maintenabilité.

Le Game Manager joue un rôle clé en orchestrant la gestion du score, les collisions, la logique du jeu et des animations. De même, la gestion des tirs via un système d’angle et de puissance apporte une dimension stratégique au gameplay.

L’ajout de l’audio, avec la gestion des sons via SoundPlayer enrichit l’expérience utilisateur et l’immersion dans le jeu. La représentation des bâtiments et des joueurs, ainsi que leur gestion des dégâts, contribuent à un mécanisme de jeu équilibré où chaque action impacte directement le score.

En conclusion, ce projet illustre parfaitement comment combiner concepts théoriques et implémentation pratique pour concevoir un jeu fonctionnel et structuré. Il pourrait être amélioré en intégrant WMPlib pour les sons, et aussi une fonction pour affronter un adversaire contrôlé par l’ordinateur. Ces améliorations permettraient d’enrichir l’expérience du jeu.

## Conclusion personnelle

Ce projet a été un petit défi, qui m'a aidé à améliorer mes connaissances en C# et j'ai découvert plusieurs choses tel que les interfaces et le pattern matching. Sans doute, mon solide bagage dans différents langages de programmation tel que le C++ m'a aidé à partir directement sur un concept adapté.

Je me suis aussi inspiré de mon expérience C# sur Unity où j'ai découvert une autre manière de programmer en orienté objet en 3D, qui va plus loin que la console 2D et ses limites.

Je trouve dommage que la formation ne pousse pas la programmation vers un moteur de jeu tels que, Unity ou Unreal Engine, et qu'on reste uniquement dans la console (old school). C'est ce qu'on attendrait d'une formation de développeur de nos jours.

# Table des illustrations

[Figure 1: Pattern matching 3](file:///C:\Users\pu36ylg\Documents\GitHub\JeuDeBalle\Christopher_Ristic_Rapport.docx#_Toc192245834)

[Figure 2: Windows Media Player 6](file:///C:\Users\pu36ylg\Documents\GitHub\JeuDeBalle\Christopher_Ristic_Rapport.docx#_Toc192245835)

[Figure 3: Animation angle de tir 6](file:///C:\Users\pu36ylg\Documents\GitHub\JeuDeBalle\Christopher_Ristic_Rapport.docx#_Toc192245836)

[Figure 4: Animation game over 6](file:///C:\Users\pu36ylg\Documents\GitHub\JeuDeBalle\Christopher_Ristic_Rapport.docx#_Toc192245837)

[Figure 5: Animation (force faible) 6](file:///C:\Users\pu36ylg\Documents\GitHub\JeuDeBalle\Christopher_Ristic_Rapport.docx#_Toc192245838)

[Figure 6: Animation puissance de tir 6](file:///C:\Users\pu36ylg\Documents\GitHub\JeuDeBalle\Christopher_Ristic_Rapport.docx#_Toc192245839)

[Figure 7: Structure des interfaces 7](file:///C:\Users\pu36ylg\Documents\GitHub\JeuDeBalle\Christopher_Ristic_Rapport.docx#_Toc192245840)

[Figure 8: Structure des classes 7](file:///C:\Users\pu36ylg\Documents\GitHub\JeuDeBalle\Christopher_Ristic_Rapport.docx#_Toc192245841)