

UNIVERSITÉ DE MONTPELLIER  
MASTER 1 - IMAGINE

---

# Swing

Jeu vidéo de réflexion

---

RAPPORT DE PROJET T.E.R.  
PROJET INFORMATIQUE — HAI823I

**Étudiants :**

M. Florentin DENIS  
M. Marius JENIN  
M. Khélian LARVET  
M. Benjamin PRE

**Année :** 2021 - 2022

**Encadrant :**

Mme. Noura FARAJ



*Nous tenons à remercier notre encadrante Mme. Noura FARAJ pour son accompagnement, sa bienveillance et ses conseils précieux tout au long de ce projet qui nous ont permis de le porter à son état actuel. Nous remercions également Pascal LARVET et Julie BADIA, pour leur relecture attentive de ce rapport.*

# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1 Organisation du projet</b>	<b>4</b>
1.1 Attribution et planification des tâches . . . . .	4
1.2 Technologies utilisées . . . . .	5
<b>2 Rapport technique</b>	<b>6</b>
2.1 Grille et état de jeu . . . . .	6
2.1.1 Tableau de prédition . . . . .	7
2.1.2 Tableau de jeu . . . . .	8
2.2 Implémentation des balles . . . . .	9
2.2.1 Balles normales (sans pouvoir) . . . . .	9
2.2.2 Balles spéciales (avec pouvoir) . . . . .	10
2.3 Implémentation des règles . . . . .	13
2.3.1 Les alignements . . . . .	13
2.3.2 Les empilements . . . . .	14
2.3.3 Les balancements . . . . .	14
2.4 Niveau, Score et Gestion de la difficulté . . . . .	16
2.4.1 Niveau . . . . .	16
2.4.2 Incitation à la prise de décision rapide . . . . .	16
2.4.3 Calcul du score . . . . .	17
2.4.4 Apparition des balles spéciales . . . . .	17
2.5 Gestion des animations au niveau des balles . . . . .	19
2.6 Ajout des bruitages et musiques . . . . .	23
2.6.1 Bruitages . . . . .	23
2.6.2 Musiques . . . . .	23
2.7 Gestionnaire des contrôles . . . . .	24
2.8 Mise en place des paramètres . . . . .	27
2.8.1 Paramètres . . . . .	28
2.8.2 Langues . . . . .	29
2.9 Interfaçage graphique avec Unity . . . . .	32
2.9.1 Modélisation 3D . . . . .	32
2.9.2 Shaders dynamiques . . . . .	33
2.9.3 Effets visuels . . . . .	36
2.9.4 Création des menus . . . . .	38
2.9.5 Affichage tête haute (ou HUD) . . . . .	42
2.10 Implémentation de l'écran partagé . . . . .	43
2.11 Classement des meilleurs scores . . . . .	45
2.11.1 Coté serveur . . . . .	45
2.11.2 Coté client . . . . .	47
<b>3 Articles</b>	<b>49</b>
3.1 Étude des techniques d'interaction pour les environnements 3D interactifs . . . . .	49
3.2 Animation contrôlée des sprites vidéo . . . . .	52
3.3 Détection de collision voxel . . . . .	54
3.4 Génération des niveaux à partir de vidéos de jeu . . . . .	56

<b>4 Bilan et difficultés rencontrées</b>	<b>57</b>
4.1 Avancement du projet . . . . .	57
4.2 Changements majeurs . . . . .	57
4.3 Difficultés rencontrées . . . . .	57
<b>Conclusion</b>	<b>59</b>
<b>Annexes</b>	<b>60</b>
<b>Bibliographie</b>	<b>62</b>

# Introduction

Dans le cadre du module de projet informatique du second semestre de M1 IMAGINE, nous avons développé un jeu nommé Swing dont le logiciel et le code sont disponibles en ligne aux adresses suivantes :

**Jeu compilé :** <https://m1.flareden.fr/TER/>

**Dépôt de code :** <https://github.com/KhelianL/Swing>

Swing est un jeu d'arcade et de réflexion se jouant sur une grille contenant 4 bascules (donc 8 colonnes). Chaque colonne possède une hauteur maximale de 10 balles. Si l'une des colonnes dépasse cette limite, la partie se termine. Le joueur devra donc faire un maximum de point avec plusieurs méthodes dont l'empilement, l'alignement et les basculements que nous présenterons dans ce rapport.

Notre groupe est composé de quatre personnes, Florentin DENIS, Marius JENIN, Khélian LARVET, Benjamin PRE et nous sommes encadrés par Mme. Noura FARAJ. La réalisation du projet s'est déroulée du 30 janvier au 23 mai 2022.

## Motivations du projet

Notre première motivation sur ce projet a été la possibilité de manipuler le moteur de jeu Unity et de se familiariser avec. Ce moteur est gratuit et très répandu dans la communauté des créateurs de jeu permettant de créer des jeux rapidement avec les diverses fonctionnalités mises à disposition.

La perception des joueurs sur un jeu a également été un point-clé sur notre motivation. En effet il est difficile de définir ce qui peut produire de l'amusement chez un joueur. Cette notion étant subjective, nous avons essayé de donner l'effet d'un jeu facile à prendre en main mais difficile à maîtriser. Ainsi notre jeu étant basé sur la réflexion nous avons trouvé judicieux de récompenser la prise de décision rapide pour favoriser le dynamisme du jeu.

De plus, ce projet avait également pour objectif la rénovation d'un ancien jeu nommé "Marble Master" (1997) qui, jusqu'à cette date, n'a toujours pas été modernisé de manière sérieuse. Ainsi nous voulions retranscrire à travers ce projet un maximum des sensations que nous pouvions obtenir dans le jeu d'origine.

Ce projet nous semblait donc parfait pour apprendre les rudiments du moteur de jeu Unity et comprendre comment faire un jeu attrayant et dynamique.

## Objectifs du projet et cahier des charges

Notre objectif principal a donc été de créer un jeu d'arcade et de réflexion sur Unity dans le langage C#. Plusieurs objectifs ont été fixés pour mener à bien ce projet :

- Créer une interface en deux dimensions au sein d'un environnement en trois dimensions.
- Implémenter les différents aspects du jeu tels que les grilles, les entités, et les règles.
- Gérer les animations au niveau des balles
- Ajouter des menus afin de naviguer convenablement dans le jeu.
- Ajuster la difficulté du jeu pour produire une courbe d'apprentissage.
- Ajouter des menus afin de naviguer convenablement dans le jeu.
- Créer un système multijoueur écran partagé voire en ligne avec tableau des scores.

# Chapitre 1

## Organisation du projet

### 1.1 Attribution et planification des tâches

Nous nous sommes mis d'accord, selon les préférences de chacun, sur la répartition des tâches. Nous avons donc défini des phases en fonction de l'avancement de notre projet :

La première phase du projet s'est basée sur l'apprentissage et la prise en main du moteur Unity. Nous avons donc mis en place notre projet Unity sur cette période mais nous avons également recherché une structure permettant d'implémenter facilement toutes les méthodes nécessaires au bon fonctionnement de notre jeu (cf. Annexe F).

Une fois la mise en place du projet terminée, nous sommes passés à la seconde phase du projet consistant à implémenter notre structure. Nous nous sommes donc répartis les tâches suivantes :

- **Marius JENIN** : Implémentation de la règle de balancement des balles et des animations avec ajout des bruitages.
- **Benjamin PRE** : Implémentation des règles d'alignement et d'empilement.
- **Khélian LARVET** : Implémentation de la génération des balles normales (sans pouvoir) et modélisation des balles spéciales (avec pouvoir).
- **Florentin DENIS** : Modélisation de la scène, implémentation des interactions utilisateur, création des menus et affichage tête haute (HUD).

Cette phase a été cruciale car elle représentait le noyau du jeu et il était impératif qu'aucun élément ne puisse produire un quelconque problème au sein du jeu.

Enfin, nous avons peaufiné notre projet en y ajoutant des sons, effets visuels, des menus ainsi qu'un système multijoueur.

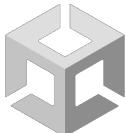
- **Marius JENIN** : Gestion de la difficulté et du score et débogage et améliorations des balances, des animations et des balles de pouvoirs.
- **Benjamin PRE** : Amélioration des règles d'alignement et d'empilement.
- **Khélian LARVET** : Ajout des effets visuels et implémentation des actions pour les balles spéciales avec une amélioration de leur modèle.
- **Florentin DENIS** : Implémentation du multijoueur, création des paramètres, mise en place des classements et finalisation des menus et affichage tête haute (HUD).

Une fois par semaine en utilisant Discord, nous avons pu faire le point sur notre avancement personnel, et par la même occasion, organiser les différents rendez-vous avec notre encadrant. Discord nous permettait également de garder une trace écrite de nos recherches et des points abordés lors des rendez-vous.

## 1.2 Technologies utilisées

**Discord :**

Logiciel de messagerie instantanée gratuite. Il nous permet de communiquer au sein de l'équipe, de s'entraider, et d'avoir une trace de nos discussions.

**Unity :**

Moteur de jeu multiplateforme avec licence gratuite. Il nous offre toutes les fonctionnalités de base pour la création d'un jeu. Il nous permet également de produire un exécutable multi-plateforme et multi-architecture.

**Plastic SCM :**

Service d'hébergement et de gestion de développement informatique en ligne. Il nous permet de garder une trace de chaque version de fichiers en ligne, lors du développement du projet Unity.

**C# :**

Langage de programmation orientée objet, il est compilé dans un langage intermédiaire et ensuite interprété par une machine virtuelle .NET (dérivé du C++). Il nous est imposé dans la mesure où nous utilisons Unity.

**Node.js :**

Plateforme logicielle JavaScript permettant la création d'application événementielle, tel que des serveurs web permettant une forte montée en charge.

**MySQL :**

C'est un serveur de bases de données relationnelles distribué sous licence libre GNU. Nous pouvons le voir comme un lieu de stockage des données qui seront ensuite récupérées via des requêtes SQL.

# Chapitre 2

## Rapport technique

### 2.1 Grille et état de jeu

La grille de jeu est contenue dans notre classe `GameZone`, qui est la classe principale de notre jeu. C'est ici que sont stockés les différents éléments de notre jeu tels que le joueur, notre animateur, les tableaux, la boucle de jeu, etc. La classe `GameZone` utilise une autre classe nommée `GameState` qui initialise et conserve toutes les variables décrivant l'état de notre jeu :

```
1 public class GameState
2 {
3     // Static bounds
4     private static int SCORE_BALL = 50;
5     public static int MULTIPLICATOR_MAX = 4;
6     public static int MULTIPLICATOR_MIN = 1;
7     private static float TIME_MAX = 5.0f;
8     public static int BALL_BY_LEVEL = 30;
9     private static int INIT_LEVEL = 3;
10    private static int INIT_MULTIPLICATOR = 1;
11    private static float TIMING_BETWEEN_GAMEOVER_ROWS = 0.1f;
12
13    // Variables
14    private float _gameDuration;
15    private int _level;
16    private long _score;
17    private int _multiplicator;
18    private int _nbBallDrop;
19    private int _nbBallBeforeLevelUp;
20    private int _countPowerUp;
21    private bool _gameOver;
22    private bool _gameOverComputing;
23    private int _gameOverRow;
24    private float _timingGameOver;
25    private float _time;
26    private string _name;
27    private SpecialBall _nextPu;
28    private NormalBall _levelBall;
29    [...]
30 }
```

Ainsi, notre jeu est facilement adaptable et modulable en modifiant l'ensemble de ces variables. Toutes ces variables sont ensuite utilisées dans diverses méthodes pour produire des événements. Ces événements sont décrits séquentiellement dans la boucle de jeu contenue dans la classe `GameZone` qui s'exécute à intervalle régulier et permettant de vérifier l'ensemble des règles du jeu :

```

1 // GameZone.cs
2 public void ComputeGameZone(float deltaT)
3 {
4     // Mise à jour visuel du multiplicateur
5     if (_gameState.UpdateMultiplicator(deltaT))
6         UpdateMultiplicatorLight();
7
8     // Test si l'animateur a échoué (Erreur pour une balle "Out of bounds")
9     if (Animator.Animate(deltaT) && !_gameState.GameOverComputing)
10        _gameState.StartGameOver();
11
12    // Mise à jour des poids et calcul des balancements
13    UpdateWeightSwings();
14    if (ComputeSwing() && !_gameState.GameOverComputing)
15        _gameState.StartGameOver();
16
17    // Si le jeu n'est pas terminé
18    if (!_gameState.GameOverComputing)
19    {
20        ComputePowerUps();           // Application des actions des balles spéciales
21        ComputeAlignment();        // Calcul des alignements
22        ComputeStack();           // Calcul des empilements
23
24        // Vérification de la hauteur maximale et de l'état du jeu
25        if (!IsLastRowEmpty() && !_gameState.GameOverComputing)
26            _gameState.StartGameOver();
27    }
28
29    // Animation de game over
30    if (_gameState.GameOverComputing)
31    {
32        // Explode the GameZone
33        if (_gameState.UpdateGameOver(deltaT))
34            ExplodeStepGameZone();
35    }
36
37    // Rafraîchissement des effets visuels
38    UpdateEffect();
39 }

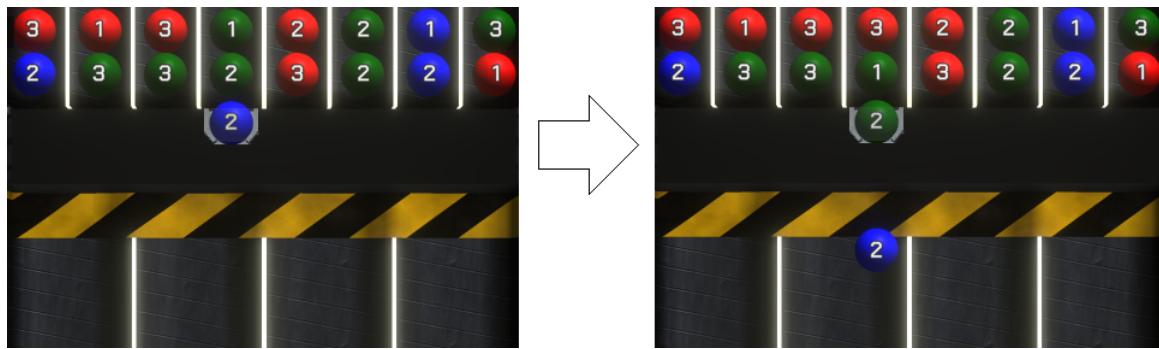
```

Comme énoncé précédemment, la classe **GameZone** contient deux tableaux de conteneurs de balles : un contenant les balles à venir (le tableau de prédiction avec la classe **PredictionZone**) et l'autre contenant les balles en jeu (le tableau de jeu avec la classe **PlaygroundZone**). Nous allons voir en détails comment marchent ces tableaux :

### 2.1.1 Tableau de prédiction

Le tableau de prédiction est un tableau indiquant les balles à venir. C'est ici que le joueur aura la possibilité de récupérer des balles pour les placer dans le tableau de jeu.

Nous avons ainsi assigné une action permettant au joueur de placer la balle qui possède dans le tableau de jeu, et par la même occasion de récupérer celle au-dessus de lui depuis le tableau de prédiction.



Exemple : Le joueur pose sa balle dans une colonne tout en prenant une nouvelle au-dessus de lui

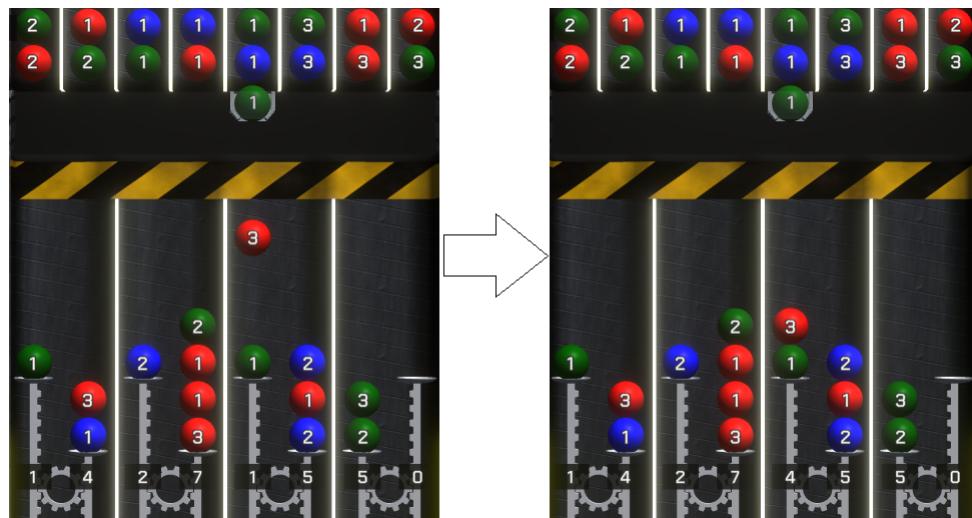
Lorsqu'une balle quitte la zone de prédiction, une nouvelle balle doit être générée pour la remplacer. Pour ce faire, nous faisons appel à l'usine à balle (**BallFactory**) pour qu'elle nous fournisse la balle adéquate selon l'état du jeu. Nous définirons plus tard dans ce rapport comment nous définissons les balles à générer.

Les compteurs de niveau et de balle spéciale sont ensuite augmentés, et si  $n$  balles ont été posées depuis le dernier passage de niveau, le niveau est incrémenté de 1 (avec  $n$  fixé à 30).

### 2.1.2 Tableau de jeu

Le tableau de jeu contient la zone où le joueur placera ses balles. Il s'agit également d'un tableau de conteneur de balle. Lorsqu'une balle est lâchée par le joueur, l'animateur s'occupe de celle-ci jusqu'à ce qu'elle tombe au-dessus d'une autre balle ou d'une balance. Il la met ensuite dans le tableau de jeu à la position trouvée. Nous détaillerons cette partie dans la section 2.5.

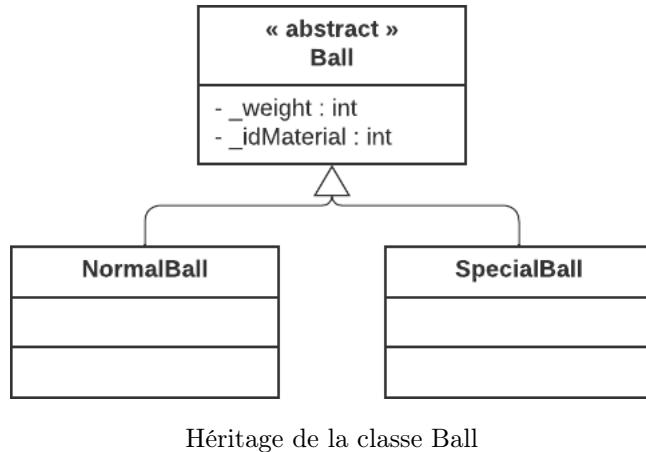
Dans l'image ci-dessous, la balle étant en train de tomber est toujours prise en charge par l'animateur, mais est placée dans un conteneur du tableau dès qu'elle se trouve au-dessus d'une autre balle.



Exemple : La balle rouge avec un poids de trois est envoyée dans le tableau de jeu

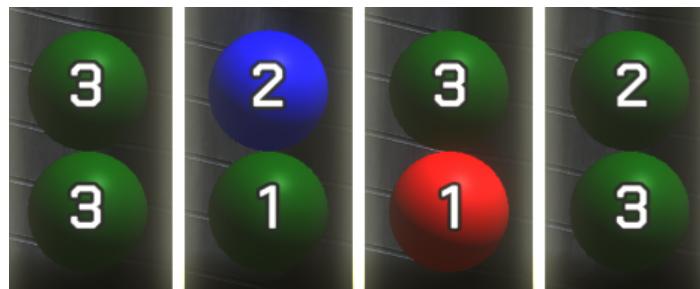
## 2.2 Implémentation des balles

Les "Balles" sont le noyau de notre jeu. Il en existe deux types que nous allons détailler ci-dessous :



### 2.2.1 Balles normales (sans pouvoir)

Les balles normales sont caractérisées par une **couleur** représentée par l'attribut **idMaterial** qui relie un entier à une couleur bien définie, et un numéro indiquant leur **poids** représenté par l'attribut **weight**.



Exemple de balles normales

Ces informations sont choisies aléatoirement selon le niveau actuel du joueur par notre usine à balle (*BallFactory*) de la manière suivante :

$$\text{Couleur} = \text{Aléatoire}(0, \text{Niveau})$$

$$\text{Poids} = \text{Aléatoire}(1, \text{Niveau} + 1)$$

Ces informations sont ensuite utilisées pour déterminer des comportements dans le tableau de jeu selon les 3 règles suivantes :

- **Un alignement** horizontal de 3 balles ou plus de même couleur implique leur élimination.
- **Un empilement** vertical de 5 balles ou plus de même couleur implique leur fusion en une seule balle.
- **Un basculement** de poids sur une balance provoque le déplacement de la balle la plus haute sur la colonne le plus légère de la différence de poids provoquée.

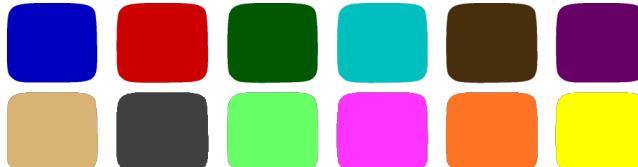
Nous détaillerons plus en profondeur ces règles dans la prochaine section.

Comme énoncé précédemment, le niveau du jeu s'incrémentera toutes les **n** balles posées par le joueur, avec **n** défini à 30 actuellement. Cette incrémentation implique l'ajout d'une nouvelle couleur avec l'attribut

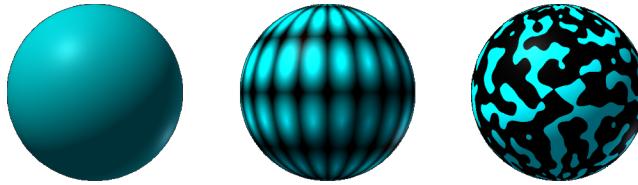
`idMaterial` et pose ainsi un problème de clarté avec notre première règle. En effet il faut pouvoir distinguer avec aisance les couleurs de chaque balle pour former des alignements.

Nous nous sommes donc posés la question suivante : *Comment produire des couleurs voire des motifs qui soient uniques et rapidement distinguables ?*

Pour pallier ce problème, nous avons défini arbitrairement un ensemble de 12 couleurs distinguables. Ces couleurs seront ensuite utilisées sur 3 types de motifs (uniforme, quadrillé, sinueux) permettant de produire jusqu'à 36 balles différentes.



Couleurs distinguables



Types de motifs en utilisant la couleur cyan

Ces motifs ont été créés en utilisant des "ShaderGraph" permettant la création de shaders dynamiques. Nous verrons plus en détail comment nous avons créé nos shaders dans la section 2.9.

- Le motif quadrillé est composé d'un diagramme de Voronoï avec un angle de 0, ce qui forme des cellules équidistantes les unes des autres.
- Le motif sinueux est composé d'une texture procédurale de bruit blanc sur lequel nous avons appliqué un seuillage.

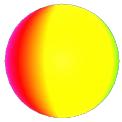
Toutes nouvelles combinaisons seraient superflues car au fur et à mesure que le niveau du joueur augmente, de nouvelles couleurs apparaissent et il devient de plus en plus difficile de créer des alignements dus à la grande variété de couleurs. De plus il est très complexe de définir de nouvelles combinaisons qui soient distinguables rapidement des autres.

Nous avons donc pris la décision que tout attribut `idMaterial` supérieur à 36 impliquerait une couleur noire. Ce qui signifie qu'il est impossible de savoir qu'elle est la couleur d'une balle ce qui force le joueur à utiliser notre troisième règle et les balles spéciales.

### 2.2.2 Balles spéciales (avec pouvoir)

Les balles spéciales n'ont pas de couleur (`idMaterial` fixé à -1) et ont un poids de 0. De plus elles ne sont pas impactées par les règles d'alignement et d'empilement.

Ces balles sont caractérisées par une action qui va impacter le tableau de jeu et un modèle visuellement unique et reconnaissable. Ces balles permettent d'ajouter du dynamisme à notre jeu et à l'heure actuelle nous avons réussi à implémenter vingt balles spéciales :



Joker



Bomb



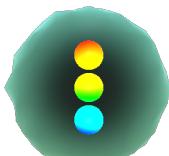
Star



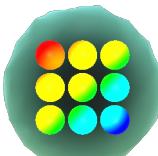
Cutter



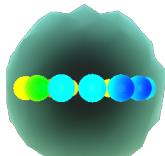
Sting



Color line



Color square



Color prediction



Zap horizontal



Zap diagonal



Joker transform



Bomb transform



Color destroy



Brick transform



Brick



Flash



Flash diagonal



Flash triangle



Brick square



Tower

- **Joker** : peut être remplacée par toutes les couleurs.
- **Bomb** : élimine toutes les balles dans un rayon de trois par trois dès sa collision avec une autre balle. (Les balles éliminées augmentent le score)
- **Star** : Si trois étoiles sont alignées horizontalement, toutes les balles sont éliminées. (Les balles éliminées augmentent le score)
- **Cutter** : élimine toutes les balles d'une colonne avant de s'éliminer lui-même.
- **Sting** : élimine périodiquement les balles se trouvant au-dessus et en dessous de lui.
- **Color line** : dès sa collision avec une colonne, transforme toutes les balles de cette colonne par la balle touchée avant de s'éliminer lui-même.
- **Color square** : dès sa collision avec une balle, transforme toutes les balles dans un rayon de trois par trois par la balle touchée avant de s'éliminer lui-même.
- **Color prediction** : dès sa collision avec une balle, transforme toutes les balles de la dernière ligne du tableau de prédiction par la balle touchée avant de s'éliminer lui-même.

- **Zap horizontal** : dès sa collision, élimine toutes les balles se trouvant sur la ligne de la collision avant de s'éliminer lui-même. (Les balles éliminées augmentent le score)
- **Zap diagonal** : dès sa collision, élimine toutes les balles se trouvant sur les diagonales de la collision avant de s'éliminer lui-même. (Les balles éliminées augmentent le score)
- **Joker transform** : dès sa collision avec une balle, transforme toutes les balles ayant le même attribut `idMaterial` que la balle touchée par des **Joker**. Il s'élimine lui même une fois l'opération terminée.
- **Bomb transform** : dès sa collision avec une balle, transforme toutes les balles ayant le même attribut `idMaterial` que la balle touchée par des **Bomb**. Il s'élimine lui même une fois l'opération terminée.
- **Color destroy** : dès sa collision avec une balle, élimine toutes les balles ayant le même attribut `idMaterial` que la balle touchée. Il s'élimine lui même une fois l'opération terminée. (Les balles éliminées augmentent le score)
- **Brick transform** : dès sa collision avec une balle, transforme toutes les balles ayant le même attribut `idMaterial` que la balle touchée par des **Brick**. Il s'élimine lui même une fois l'opération terminée.
- **Brick** : balle sans effet particulier ce qui permet d'entraver le joueur.
- **Flash** : dès sa collision avec une balle, transforme aléatoirement deux balles par ligne par la balle touchée. Il s'élimine lui-même une fois l'opération terminée.
- **Flash diagonal** : dès sa collision avec une balle, transforme diagonalement toutes les balles par la balle touchée. Il s'élimine lui même une fois l'opération terminée.
- **Flash triangle** : dès sa collision avec une balle, transforme toutes les balles se situant entre les deux diagonales par la balle touchée. Il s'élimine lui-même une fois l'opération terminée.
- **Brick square** : dès sa collision avec une balle, transforme toutes les balles dans un rayon de trois par trois par des **Brick** avant de s'éliminer lui-même.
- **Tower** : dès sa collision, il remplit la colonne en question jusqu'à la limite possible avec des **Bricks**.

Ainsi chaque balle spéciale possède sa propre classe où nous définissons son `GameObject` (ou Prefab) et deux méthodes :

- La méthode `Clone()`, qui nous est très utile pour l'ensemble des pouvoirs à base de transformations permettant de dupliquer une balle avec une certaine classe.
- La méthode `Action()`, qui définit le comportement des balles spéciales lors de leur entrée dans le tableau de jeu.

Diverses autres méthodes ont été implémentées dans le tableau de jeu pour accélérer leur développement notamment avec les fonctions :

- `HasBall()` : Renvoie vrai si une case du tableau de jeu possède une balle.
- `ExplodeBall()` : Permet de supprimer et libérer la mémoire pour une case du tableau de jeu. Cette fonction permet aussi de gérer l'application d'un effet visuel ou le lancement d'un bruitage. Nous développerons cette partie un peu plus tard dans ce rapport.

## 2.3 Implémentation des règles

Les règles du jeu concernent tous les calculs effectués sur le tableau de jeu. Ces règles permettent de déterminer si des balles doivent être éliminées (alignement), fusionnées (empilement) ou envoyées vers d'autres colonnes (balancements). À chaque boucle de jeu, trois fonctions sont appelées pour faire ces calculs.

### 2.3.1 Les alignements

Lorsqu'au minimum 3 balles de la même couleur sont alignées, ces balles doivent être éliminées, et le score doit être incrémenté en fonction du poids de celles-ci. Pour ce faire, nous avons implémenté une fonction qui parcourt toutes les balles du tableau, de gauche à droite et de bas en haut.

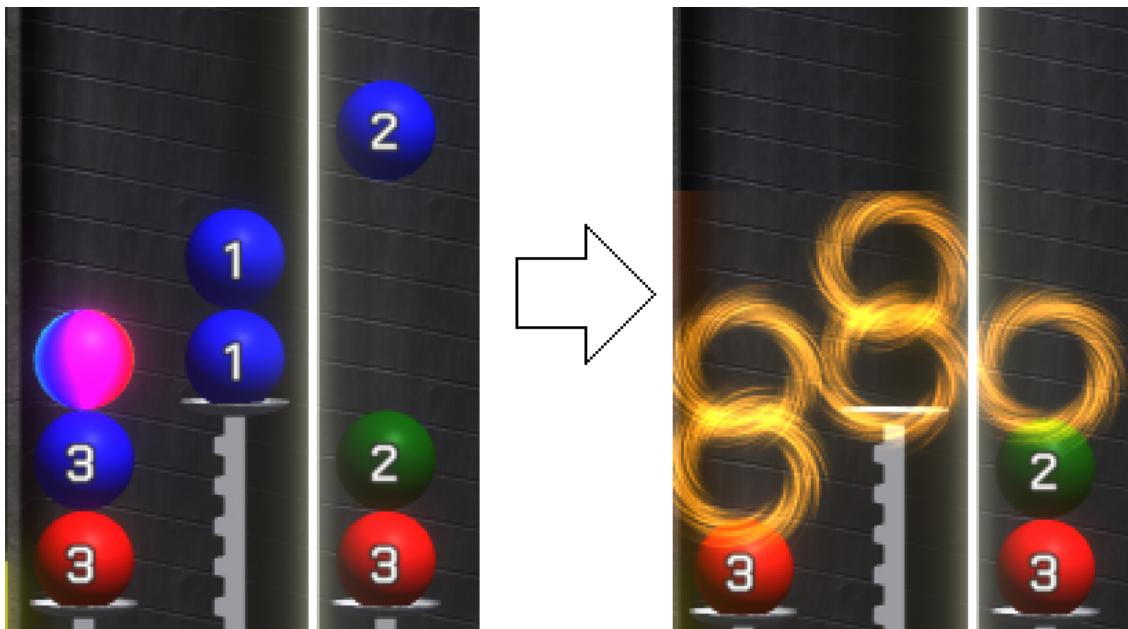
Pour chacune des balles parcourues, on sauvegarde son `idMaterial` afin de connaître sa couleur. Puis on regarde la balle suivante, si elle est de la même couleur ou si c'est un joker alors on continue la recherche. Si la première balle qu'on observe est un joker, on prend la couleur de la suivante. Les coordonnées de ces balles seront ensuite stockées dans une liste pour être détruites plus tard.

En plus de devoir détruire les balles alignées, l'algorithme doit également détruire toutes les balles adjacentes à ces balles si elles ont la même couleur que l'alignement.

Pour ce faire, l'algorithme enregistre dans une liste, les positions des balles alignées qui ont été trouvées. Puis, pour chacune de ces balles, la fonction `checkAdjacency()` analyse leur 4 balles voisines.

Si une des balles voisine est de la même couleur, elle est ajoutée dans la liste de balles à détruire, puis la fonction est de nouveau appelée sur elle.

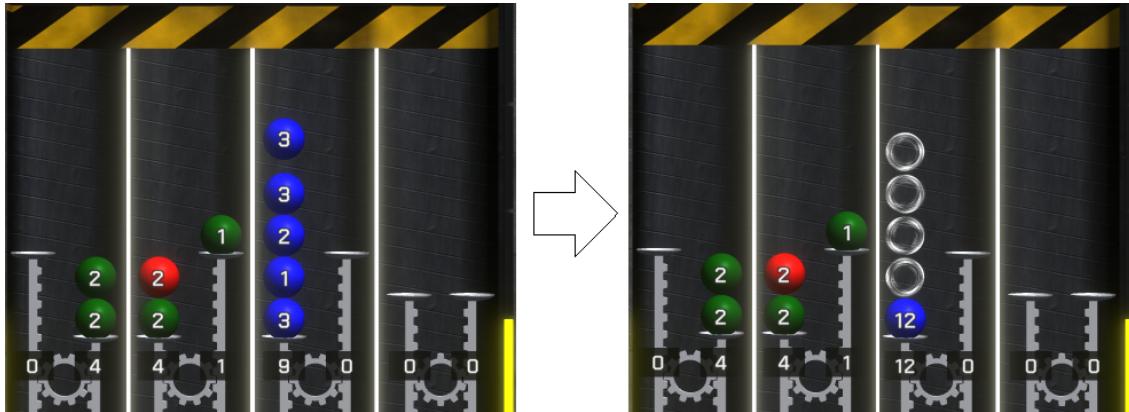
Une fois la liste des positions des balles à éliminer remplie, l'algorithme doit vérifier une dernière chose. En effet, pour une meilleure jouabilité, un alignement ne peut pas être effectué si des balles de la même couleur qu'un alignement sont en train de tomber dans les colonnes concernées par cet alignement. Si cette condition est respectée, toutes les balles de la liste sont effacées de la zone de jeu et le score est augmenté de la valeur adéquate. Enfin, toutes les balles en suspension (dont la case en dessous est vide) sont prises en charge par l'animateur afin de tomber.



Exemple d'un alignement de balles bleues avec un joker

### 2.3.2 Les empilements

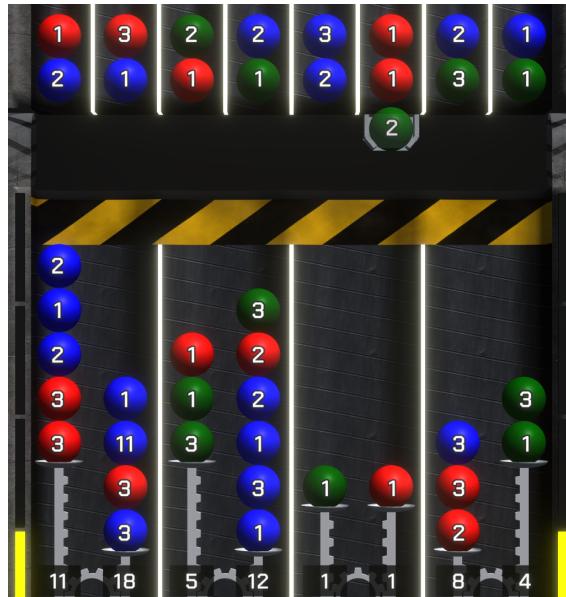
Si le joueur parvient à empiler cinq balles ou plus de la même couleur, elles doivent alors être fusionnées pour ne former qu'une seule balle dont le poids est la somme des poids des balles fusionnées. Ce comportement est pris en charge par une fonction parcourant tout le tableau, puis pour chaque balle, comptant le nombre de balles de la même couleur empilées au-dessus d'elle. Si cinq balles ou plus sont empilées de cette façon, alors on détruit chacune des balles de cet empilement en sommant leur poids pour créer une nouvelle balle, ayant pour poids cette somme.



Exemple d'un empilement de balles bleues

### 2.3.3 Les balancements

Les balancements sont les mécaniques originales de ce jeu. C'est d'ailleurs là d'où vient le nom du jeu (Swing). Les balances sont étroitement liées à la zone de jeu. En effet chaque balance est posée sur 2 colonnes (un plateau par colonne). Une balance est définie par un état possible parmi 3 : équilibrée ou déséquilibrée à gauche ou déséquilibrée à droite.



Exemple de balances déséquilibrées et équilibrées

Pour aider le joueur dans son appréciation des déséquilibrages, nous avons rajouté des compteurs en dessous de chaque plateau des balances pour indiquer la somme des poids actuels de chaque colonne.

Ainsi, les basculements permettent la mécanique la plus amusante de notre jeu : le catapultage de balle. En effet lorsqu'une balle déstabilise une balance, la balle placée au sommet de la colonne la plus lourde (si elle existe) est déplacée de  $n$  colonnes dans la direction de la déstabilisation, avec  $n$  défini par :

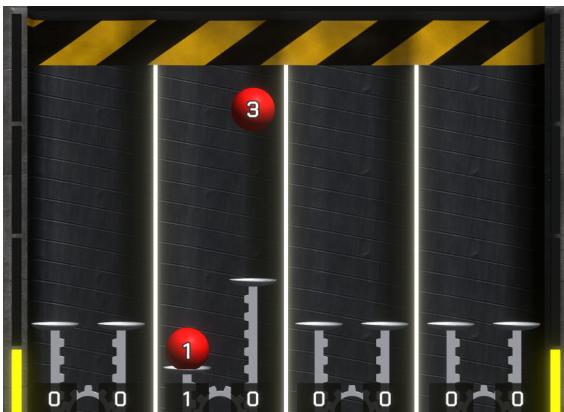
$$n = \text{poids}(\text{Colonne Lourde}) - \text{poids}(\text{Colonne Legere})$$

Si le déséquilibre d'une balance provoque un catapultage en dehors des limites du tableau de jeu, alors la balle continuera de se mouvoir dans un tuyau qui la repositionnera sur la colonne limite du tableau de jeu opposée à son trajet.

Cette transition est également accompagnée d'une transformation en fonction du type de la balle qui sera repositionnée :

- Une balle normale, ou une **Bomb** sera transformée en **Joker**.
- Une balle spéciale autre qu'une **Bomb** sera transformée en **Bomb**.

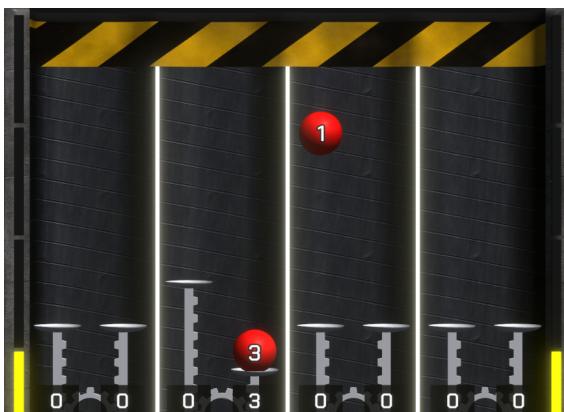
Cette mécanique permet de créer des balles de pouvoir et aide les joueurs autant au début qu'à la fin du jeu. Voici un petit exemple du déséquilibre des balances :



1) Une balle avec un poids de 3 tombe sur une balance.



2) La balance est déséquilibrée et catapulte la balle de poids de 1 vers la droite.



3) La balle avec un poids de 1 se déplace de (3-1) colonnes.



4) La balle avec un poids de 1 déséquilibre une balance vide.

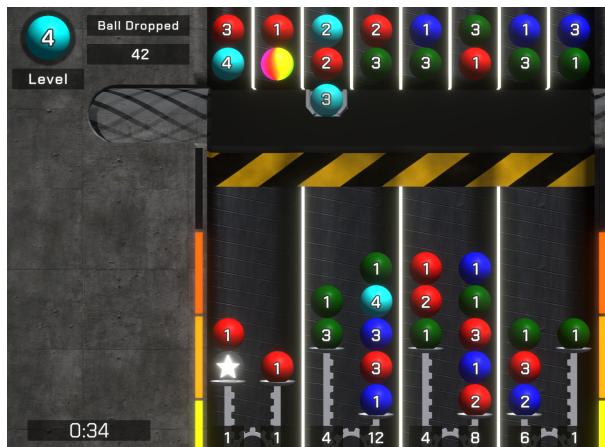
## 2.4 Niveau, Score et Gestion de la difficulté

### 2.4.1 Niveau

Au début d'une partie, le joueur commence au niveau 3. Comme énoncé précédemment, ce niveau désigne le poids maximal d'une balle et le nombre total de couleurs possibles.

Ainsi au niveau 3, il y a 3 couleurs et des balles de poids 1, 2 ou 3. Nous avons fixé notre variable de passage de niveau à 30, ce qui signifie que toutes les 30 balles lâchées on passe au niveau suivant. Nous avons également rajouté une particularité lors d'un passage de niveau avec l'apparition d'une balle spéciale **Star** qui ne peut apparaître qu'à cette occasion.

Il n'y a pas de limite de niveau car théoriquement le jeu est infini. Néanmoins la gestion de la difficulté empêche de jouer indéfiniment, car la difficulté augmente progressivement.



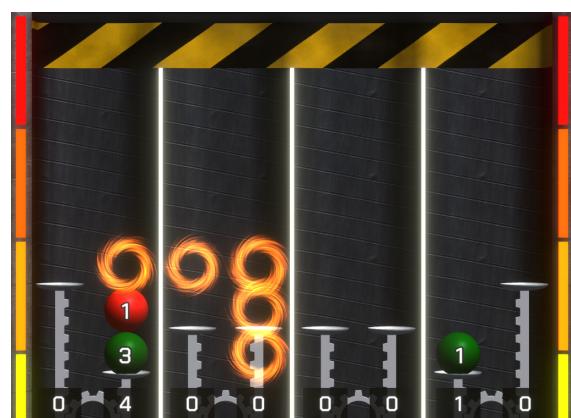
Exemple de jeu au niveau 4



Exemple de jeu au niveau 17

### 2.4.2 Incitation à la prise de décision rapide

Afin d'ajouter la vitesse comme un facteur de victoire, le jeu comporte un multiplicateur. Un multiplicateur est au minimum à 1 et au maximum à 4. À chaque fois que le joueur provoque un alignement de balles, ce multiplicateur est remis à 4 et toutes les 5 secondes il décroît de 1. Ce multiplicateur va jouer un rôle dans le calcul du score et le taux d'apparition des balles spéciales.



Multiplicateur à 1 et à 4 après l'alignement de balles (indiqué par les barres lumineuses sur les cotés)

### 2.4.3 Calcul du score

Le score du joueur ne peut qu'augmenter. Il faut le voir comme un compteur de score de jeu d'arcade. Après que le joueur ait effectué un alignement ou explosé des balles avec une balle spéciale, qui génère des points de score, on additionne le poids de toutes les balles concernées. Le score augmente alors de cette manière :

$$\text{score} = \text{score} + 50 * \text{poids} * \text{multiplicateur}$$

### 2.4.4 Apparition des balles spéciales

Le taux d'apparition des balles spéciales est très important dans notre jeu. En effet c'est ce qui va gérer la difficulté du jeu et donc le rythme et la durée d'une partie. Il faut également prendre en compte que la difficulté augmente naturellement avec la montée des niveaux qui incrémente le poids des balles et le nombre de couleurs possible. Ce taux d'apparition dépend du niveau de la partie et du multiplicateur.

Pour chaque palier de 2 niveaux jusqu'au niveau 20, des probabilités ont été fixées pour chaque balle de pouvoir. Ainsi les balles de pouvoir les plus impactantes ou les plus handicapantes n'arrivent qu'à partir d'un certain niveau et de manière régulée. Nous avons alors créé cette table de probabilités :

PU \ Levels	3	4 & 5	6 & 7	8 & 9	10 & 11	12 & 13	14 & 15	16 & 17	18 & 19	20 +
Joker	50	50	25	20	10	5	0	0	0	0
Bomb	50	50	25	20	10	8	7	7	7	6
ZapH	0	0	25	20	10	9	7	7	7	6
ZapDiag	0	0	25	20	10	9	7	7	7	6
Cutter	0	0	0	20	10	8	7	7	7	6
PlasmaNoTriangle	0	0	0	0	13	8	5	3	0	0
PlasmaEmptyTriangle	0	0	0	0	13	11	8	5	3	0
BrickSquare	0	0	0	0	13	10	7	7	6	4
Sting	0	0	0	0	11	10	8	6	6	6
TransformBrick	0	0	0	0	0	11	8	8	8	6
TransformDestroy	0	0	0	0	0	11	9	9	8	6
TransformJoker	0	0	0	0	0	0	9	9	8	5
CopyLine	0	0	0	0	0	0	9	8	7	6
CopySquare	0	0	0	0	0	0	9	8	7	6
CopyPrediction	0	0	0	0	0	0	0	9	9	8
TransformBomb	0	0	0	0	0	0	0	0	10	10
BrickTower	0	0	0	0	0	0	0	0	0	9
PlasmaFullTriangle	0	0	0	0	0	0	0	0	0	10
Total	100	100	100	100	100	100	100	100	100	100

Table définissant les probabilités de chaque balle de pouvoir

Après avoir défini des probabilités pour choisir la balle de pouvoir, il nous fallait définir à quel moment la balle choisie allait apparaître. Nous avons donc mis en place un compteur indiquant le nombre de balle à poser avant l'apparition de la balle en question. Ce compteur est défini par la table suivante et sera impacté par le multiplicateur qui permettra de réduire le compteur et d'ajouter de la stratégie à notre jeu :

Levels	3	4 & 5	6 & 7	8 & 9	10 & 11	12 & 13	14 & 15	16 & 17	18 & 19
Count	ballByLevel + 8	8	8	7	7	7	6	6	5

Table définissant le compteur de la prochaine balle de pouvoir pour chaque niveau

Ensuite nous ajoutons le complément à 4 du multiplicateur (4 - multiplicateur) pour que la vitesse de jeu ait un intérêt dans la génération des balles de pouvoir.

Ensuite pour les niveaux au-dessus de 19 nous avons des fonctions définissant le compteur afin que l'on puisse jouer théoriquement indéfiniment en faisant évoluer la difficulté. Nous voulions qu'après le niveau 19 le nombre de balle de pouvoir continue à augmenter progressivement (donc que le compteur diminue) jusqu'à un niveau donné (le niveau 25 pour la version finale). Après ce niveau le nombre de balle de pouvoir diminue petit à petit. Pour ces fonctions nous utilisons ces variables :

*multiplicator* : Multiplicateur (valeur entre 1 et 4)  
*im* : Importance du multiplicateur (dans version finale : 2)  
*startLvl* : Niveau à partir duquel la fonction est définie (dans version finale : 20)  
*mediumLvl* : Niveau à partir duquel la fonction augmente (dans version finale : 25)  
*valStartLvl* : Valeur du niveau précédent le *startLvl* (dans version finale : 9)  
*minValue* : Valeur minimale atteinte au *mediumLvl* (dans version finale : 3)  
*actualLevel* : Niveau dont nous voulons connaître le compteur

$$m = \frac{\text{multiplicator} - 1}{3} - 1$$

La fonction que nous utilisons entre le niveau de départ et le niveau intermédiaire :

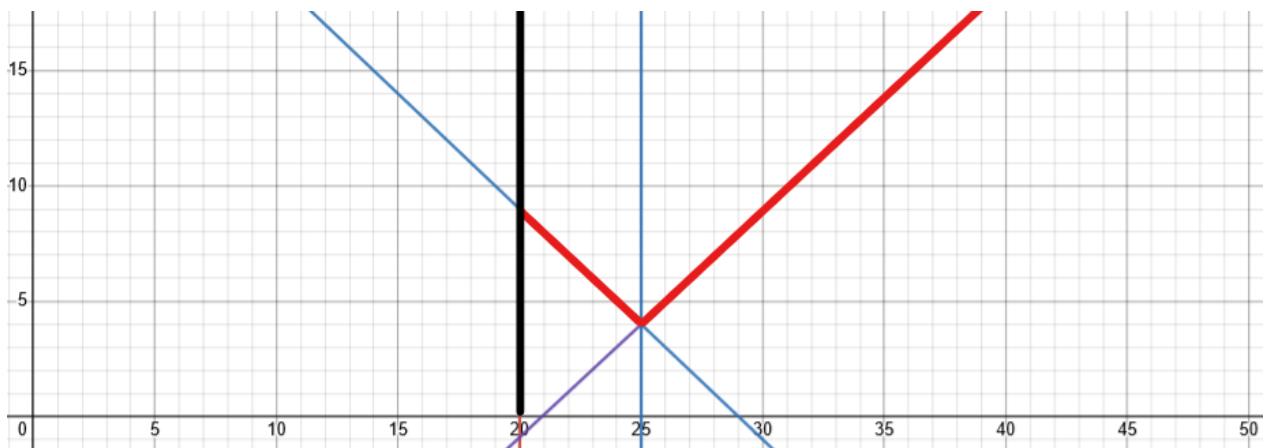
$$\text{compteur} = \frac{\text{valStartLvl} - (\text{minValue} - m * \text{im})}{\text{startLvl} - \text{mediumLvl}} * (\text{actualLevel} - \text{mediumLvl}) + \text{minValue} - m * \text{im}$$

La fonction que nous utilisons à partir du niveau intermédiaire avec un paramètre supplémentaire :

*tilt* : Inclinaison voulue de la fonction à partir du niveau intermédiaire (dans version finale : 0.7)

$$\text{compteur} = \frac{(\text{valStartLvl} - \text{minValue}) * \text{tilt}}{\text{mediumLvl} - \text{startLvl}} * (\text{actualLevel} - \text{mediumLvl}) + \text{minValue} - m * \text{im}$$

En combinant ces 2 fonctions et en les visualisant on obtient ce graphique :



Courbes finales des fonctions du compteur du taux d'apparition pour les niveaux supérieurs à 19

Tous ces paramétrages et ces fonctions ont été implémentées par la suite dans l'usine à balles (*BallFactory*). Ainsi nous gérions la difficulté en contrôlant l'aléatoire de la génération de balles.

## 2.5 Gestion des animations au niveau des balles

Afin que les balances puissent catapulter des balles et que les balles puissent tomber dans chacune des colonnes, nous avons créé un gestionnaire d'animation : l'Animateur.

L'animateur permet d'alterner les animations affectées à des conteneurs de balles flottants (non-fixés). Il existe 3 types d'animations de mouvement : vers le haut, vers un coté (gauche ou droite) et vers le bas. Ces animations s'enchaînent toujours dans un ordre précis lors d'un catapultage :

haut → coté → bas

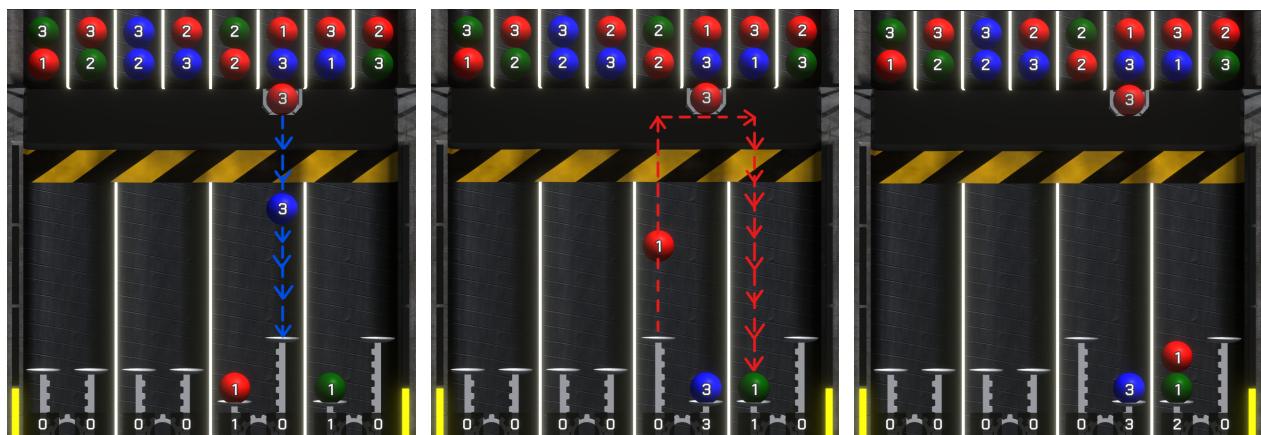
Nous pouvons aussi faire tomber une balle directement dans une colonne, ce qui sera directement une animation du type "bas".

Lors d'un catapultage on renseigne à l'animateur dans quelle direction et de combien de cases devra se déplacer la balle ciblée. Celui-ci va se charger de faire monter la balle (animation de type "haut") d'une hauteur prédefinie.

Ensuite il va changer le type de l'animation pour que la balle aille vers un coté (animation de type "coté") de la distance renseignée.

Enfin l'animateur va de nouveau changer le type de l'animation pour que la balle tombe (animation de type "bas"). Cependant cette-fois l'animateur ne la fait se déplacer que d'une case et recommence l'opération si la balle n'a rien en-dessous d'elle. Lorsque celle-ci ne peut plus tomber elle est retirée du conteneur de balle flottant pour être ajoutée dans la zone de jeu.

**Exemple de catapultage de balle avec les détails de l'animateur**



L'animateur déplace récursivement la balle bleue vers le bas

L'animateur déplace la balle rouge vers le haut en une seule fois, vers le coté en une seule fois et vers le bas en plusieurs fois

Les balles sont arrivées à destination et replacées dans la zone de jeu

Voici la fonction appelée dans l'animateur pour faire tomber une balle comme la balle bleue dans l'exemple précédent. Elle ajoute un nouveau conteneur flottant avec les bons paramètres à la liste des conteneurs de balles flottantes.

```

1  public void AddDropingBall(Ball b, Vector2 position, float nbUp = 0)
2  {
3      Vector2Int posRounded = new Vector2Int((int) Math.Round(position.x), (int)
4          Math.Round(position.y - 1));
5
6      if (_gameZone.IsPositionFree(posRounded))
7      {
8          _floatingBallsContainers.Add(
9              new FloatingBallContainer(ball: b,
10                 state: StateAnim.StateDownDeplBall, position: position, nbSide: 0, nbUp:
11                     nbUp, directionSide: Direction.DirectionLeft,
12                     directionUp: Direction.DirectionDown));
13
14      }
}

```

Voici la fonction appelée dans l'animateur pour catapulter une balle comme la balle rouge dans l'exemple précédent. Elle ajoute également un nouveau conteneur flottant à la liste des conteneurs de balles flottants mais pour une balle catapultée cette fois-ci.

```

1  public void AddFlyingBall(Ball b, float nbUp, float nbSide, Vector2 position,
2      → SwingAnimator.Direction dirHorizontal)
3  {
4      _floatingBallsContainers.Add(
5          new FloatingBallContainer(ball: b,
6              state: StateAnim.StateUpDeplBall, position: position, nbSide: nbSide,
7                  nbUp: nbUp + GameZone.SpacingFlyingBallPlayground, directionSide:
8                      dirHorizontal, directionUp: Direction.DirectionUp));
9
10 }

```

La mise à jour d'une animation de balle (`BallAnimation`) se fait de la manière suivante.

```

1  public void Actualize(float deltaTime, GameObject go)
2  {
3      float timingDepl = Math.Min(_timing, deltaTime);
4      // multDir : -1 if Left or Down / 1 if Right or Up
5      float multDir = _direction == SwingAnimator.Direction.DirectionRight || _direction
6          == SwingAnimator.Direction.DirectionUp ? 1 : -1;
7      if (_vertical == SwingAnimator.Orientation.Vertical)
8      {
9          go.transform.Translate(0, multDir * timingDepl * SwingAnimator.VerticalSpeed *
10              GameZone.SizeBall, 0);
11     }
12     else
13     {
14         go.transform.Translate(multDir * timingDepl * SwingAnimator.HorizontalSpeed *
15             GameZone.SpacingBall, 0, 0);
16     }
17
18     // On décrémente le timing
19     _timing -= timingDepl;
20 }
```

L'animateur va mettre à jour toutes les animations de balles (`BallAnimation`) et va les transformer si elles sont terminées.

```

1  public bool Animate(float deltaTime)
2  {
3      bool gameOver = false;
4      int sizeFloatingBall = _floatingBallsContainers.Count;
5      List<FloatingBallContainer> toRemove = new List<FloatingBallContainer>();
6
7      for (int i = 0; i < sizeFloatingBall; i++)
8      {
9          FloatingBallContainer floatingBall = _floatingBallsContainers[i];
10         if (!floatingBall.Paused)
11         {
12             // Mise à jour des animations
13             floatingBall.Animation.Actualize(deltaTime, floatingBall.ContainerObject);
14         }
15         if (floatingBall.Animation.IsFinished())
16         {
17             // Changement d'état de l'animation (haut + cote + bas)
18             NextAnimation(floatingBall, toRemove);
19         }
20     }
21
22     // On rajoute les balles dont l'animation est finie dans la zone de jeu
23     [...]
24
25     return gameOver;
26 }
```

Beaucoup de problèmes ont découlé du fait que nous sortions des balles de la zone de jeu pour les remettre dedans plus tard. Il a donc fallu garder en mémoire de manière discrète des mouvements continus de balles pour savoir à n'importe quel moment quelles balles dans la zone de jeu pouvaient tomber ou non. Ces vérifications nous ont servi lors des animations mais également lors des balancements et des actions de certaines balles de pouvoirs (notamment la **Tower**).

L'animateur est le noyau de la dynamique du jeu. Rien ne bouge sans passer par cet animateur qui met à jour à chaque image tous les conteneurs de balles flottantes. Cela a été un point important de la création du "gameplay".

## 2.6 Ajout des bruitages et musiques

### 2.6.1 Bruitages

Notre jeu propose des bruitages simples pour accompagner les différents événements qui peuvent survenir lors d'une partie. Ainsi, lorsqu'une balle est lâchée ou lorsqu'une balance est déséquilibrée un certain son est joué.

Pour ce faire nous avons mis en place une classe Singleton nommée `AudioManager`. Ainsi elle possède une instance unique et différentes méthodes qui seront appelées lors d'actions particulières pour jouer un son :

```

1  public class AudioManager
2  {
3      public static float SOUND_VOLUME = 1.0f;
4      private static AudioManager INSTANCE;
5
6      private AudioManager() {}
7
8      public static AudioManager GetInstance()
9      {
10         return (INSTANCE is null ? new AudioManager() : INSTANCE);
11     }
12
13     public void PlayLandingBallSound(GameObject gameObject)
14     {
15         AudioSource.PlayClipAtPoint(Resources.Load("Audio/LandingBall") as
16             → AudioClip,gameObject.transform.position,SOUND_VOLUME);
17     }
18
19     [...]
}

```

### 2.6.2 Musiques

Nous avons également ajouté des musiques d'ambiances qui sont jouées en boucle de manière aléatoire parmi une liste lorsque nous sommes à l'intérieur du jeu. À l'instar des bruitages, une classe singleton permet de gérer cet aspect, nommée `GameMusicManager` :

```

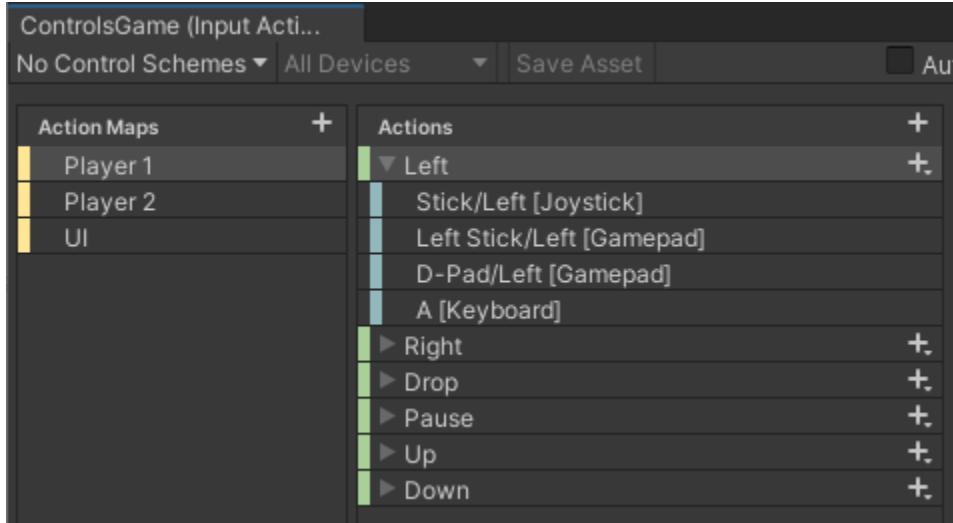
1  public class GameMusicManager : MonoBehaviour
2  {
3      private static GameMusicManager _instance;
4      public AudioClip[] tracks;
5      public AudioSource source;
6      private float _volume = 1.0f;
7
8      [...]
9
}

```

Ainsi un élément  `AudioSource` peut via la méthode `Play()`, jouer des sons de la classe  `AudioClip`. Les  `AudioSource` possèdent également un attribut de classe permettant de maîtriser leur volume.

## 2.7 Gestionnaire des contrôles

Pour permettre l'interaction entre le joueur et le jeu nous avons utilisé la librairie `Input System V1.3` de Unity à la place du `Input Manager` proposé par défaut afin de pouvoir prendre en compte plus facilement la gestion de plusieurs périphériques, tel qu'un clavier ou une manette.



Interface de Input System

Nous pouvons voir sur l'image que nous avons 2 joueurs : "Player 1" et "Player 2", qui peuvent tous deux effectuer des actions similaires :

- **Left et Right** : Permet de déplacer l'objet du joueur de gauche à droite dans la zone de jeu.
- **Drop** : Permet au joueur de lancer la balle qu'il tient dans la zone de jeu et d'en récupérer une depuis la zone de prédiction.
- **Pause** : Permet au joueur de mettre en pause la partie.
- **Up et Down** : Permet de se déplacer dans les menus Pause et Gameover.

Nous pouvons aussi remarquer des contrôles pour "UI" qui permettent la prise en charge de la souris sur les menus, et du bouton "entrée".

Afin de récupérer les touches pressées pour les appliquer au jeu, nous avons créé une classe `SwingInput`, implémentant deux interfaces : `ControlsGame.IPlayer1Actions` et `ControlsGame.IPlayer2Actions`, respectivement, les callbacks du joueur 1 et du joueur 2.

Ces deux interfaces contiennent les actions listées plus hauts, et permettent d'agir sur la classe `SwingInput`. Cette classe contient deux sous classes :

- **GlobalInputState** : Liste les états des actions globales, tels que l'appui sur le bouton de pause.
- **PlayerInputState** : Liste les états des actions d'un joueur, tels que droite, gauche ou lancer une balle.

```

1  private InputDevice _p1Controller = null;
2  private InputDevice _p2Controller = null;
3  private GlobalInputState _global;
4  private PlayerInputState _stateP1;
5  private PlayerInputState _stateP2;
6
7  public SwingInput()
8  {
9      _p1Controller = null;
10     _p2Controller = null;
11
12     ControlsGame cg = new ControlsGame();
13     cg.Player1.SetCallbacks(this);
14     cg.Player1.Enable();
15     cg.Player2.SetCallbacks(this);
16     cg.Player2.Enable();
17
18     _stateP1 = new PlayerInputState();
19     _stateP2 = new PlayerInputState();
20     _global = new GlobalInputState();
21 }
```

SwingInput a donc plusieurs attributs de classe :

- Deux InputDevice correspondant aux appareils de saisie que les joueurs utilisent
- Un GlobalInputState pour les actions telles que mettre en pause
- Deux PlayerInputState correspondant aux actions de chacun des deux joueurs possibles

Nous avons également créé un ControlsGame() qui correspond à la classe de notre Input System, pour pouvoir configurer les Callbacks et les activer.

```

1 void ControlsGame.IPlayer1Actions.OnDrop(InputAction.CallbackContext context)
2 {
3     bool same = true;
4     if (context.control.device.deviceId > 10)
5     {
6         if (_p1Controller == null && context.control.device != _p2Controller)
7         {
8             _p1Controller = context.control.device;
9         }
10        else if (_p1Controller != context.control.device)
11        {
12            same = false;
13        }
14    }
15
16    if (context.started && same)
17    {
18        _stateP1.drop = true;
19    }
20    else if (context.canceled && same)
21    {
22        _stateP1.drop = false;
23    }
24 }
```

La méthode `OnDrop()` correspond à l'action de lancer une balle dans la zone de jeu à partir du joueur 1. Dans un premier temps nous vérifions quel contrôleur (clavier ou manette) demande l'action. Si notre joueur n'a aucun périphérique (dû à une déconnexion par exemple), ou que ce même contrôleur n'est pas attribué, nous lui attribuons un périphérique. Sinon si notre joueur a déjà un périphérique, et qu'il ne s'agit pas de celui d'un autre joueur, alors nous passons le statut de l'attribut `drop` à vrai ce qui permet d'effectuer l'action dans le jeu.

Les actions sont demandées, à chaque image, au niveau de la boucle de jeu `SwingLoop`, qui est la boucle principale de notre jeu.

```

1 void Update()
2 {
3     [...]
4     List<Game.Action> globalActions = _inputV2.GetGlobalActions();
5     for (int i = 0, max = globalActions.Count; i < max; i++)
6     {
7         if (globalActions[i] == Game.Action.Pause)
8         {
9             pauseAsked = true;
10        }
11    }
12    [...]
13    for (int i = 0; i < _nbPlayer && !_isPause; i++)
14    {
15        if (!_games[i].IsGameOver())
16        {
17            List<Game.Action> actionList = _inputV2.GetPlayerActions(i);
18            if (_nbPlayer == 1)
19            {
20                actionList.AddRange(_inputV2.GetPlayerActions(1));
21            }
22            _games[i].Update(Time.deltaTime, actionList);
23        }
24    }
25    [...]
26 }
27 }
```

Lors de la boucle de mise à jour (appelée à chaque image), nous récupérons dans un premier temps les actions globales, afin de savoir si un joueur a mis pause.

Puis, s'il n'y a pas pause, nous récupérons pour chaque joueur ses actions avant de les envoyer à la partie du joueur concerné. S'il n'y a qu'un seul joueur, les commandes du joueur 1 et du joueur 2 lui sont attribuées.

## 2.8 Mise en place des paramètres

Pour permettre au jeu de s'exécuter des ordinateurs avec différentes configurations, nous avons décidé de mettre en place quelques options. Ces options permettent de gérer les aspects graphiques, sonores, et linguistiques de notre jeu avec également un système de niveaux de détails afin d'optimiser le rendu du jeu.

Les paramètres sont stockés au format JSON dans le même répertoire que l'exécutable. Si le fichier de sauvegarde est supprimé, nous le régénérerons avec les valeurs par défauts.

Pour faire cela nous avons créé plusieurs classes :

- **SettingsMenuScript** : correspond au script relié au menu visuel des paramètres en jeu. Il permet la modification des valeurs par l'utilisateur et leur application.
- **ApplySettingsScript** : correspond au script permettant la lecture des paramètres et la sauvegarde de ceux-ci.
- **SettingsData** : permet de stocker les paramètres. Cette classe possède une sous classe **Data** ayant comme attributs les différents paramètres de notre jeu. Ainsi nous donnons comme attributs deux **Data** : une pour les paramètres déjà appliqués, et une pour les paramètres modifiés mais pas encore appliqués.
- **ISettingsCallback** : interface permettant de mettre en place un système de callbacks afin de fermer le menu de paramètre quand cela est demandé dans le script.

```

1  private void ReadFromFile()
2  {
3      bool readSuccess = false;
4      if (System.IO.File.Exists(_settingsPath))
5      {
6          string data = System.IO.File.ReadAllText(_settingsPath);
7          if (data.Trim().Length > 0)
8          {
9              SettingsData.SavedSettings = JsonUtility.FromJson<SettingsData.Data>(data);
10             readSuccess = true;
11         }
12     }
13     if (!readSuccess)
14     {
15         SettingsData.SavedSettings.msaa = 2;
16         SettingsData.SavedSettings.renderScale = 1.0f;
17         SettingsData.SavedSettings.vSync = 1;
18         SettingsData.SavedSettings.musicVolume = 1.0f;
19         SettingsData.SavedSettings.soundEffectVolume = 1.0f;
20         SettingsData.SavedSettings.langue = "fr";
21     }
22 }
```

Voici la fonction permettant de lire depuis le fichier de configuration : `ReadFromFile()`.

Dans un premier temps nous tentons de lire le fichier, si nous réussissons nous venons appliquer les données JSON récupérées aux paramètres. Sinon, nous donnons aux paramètres les valeurs par défaut.

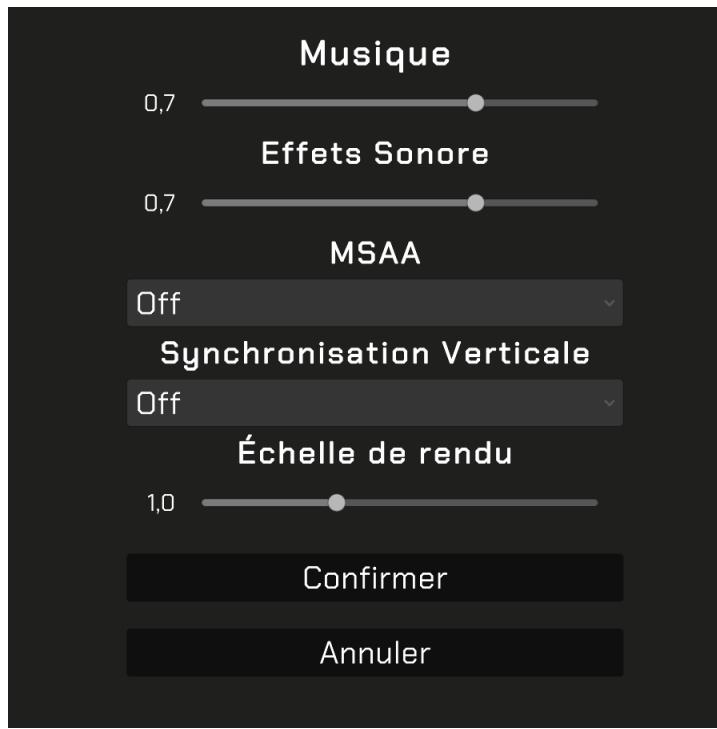
```

1 public void SaveToFile()
2 {
3     string json = JsonUtility.ToJson(SettingsData.SavedSettings);
4     System.IO.FileInfo file = new System.IO.FileInfo(_settingsPath);
5     System.IO.File.WriteAllText(file.FullName, json);
6 }
```

La fonction `SaveToFile()` quant à elle, permet de sauvegarder les paramètres dans le fichier de configuration.

### 2.8.1 Paramètres

Comme énoncé précédemment, plusieurs paramètres ont été implémentés et nous allons voir en détails leurs utilités :



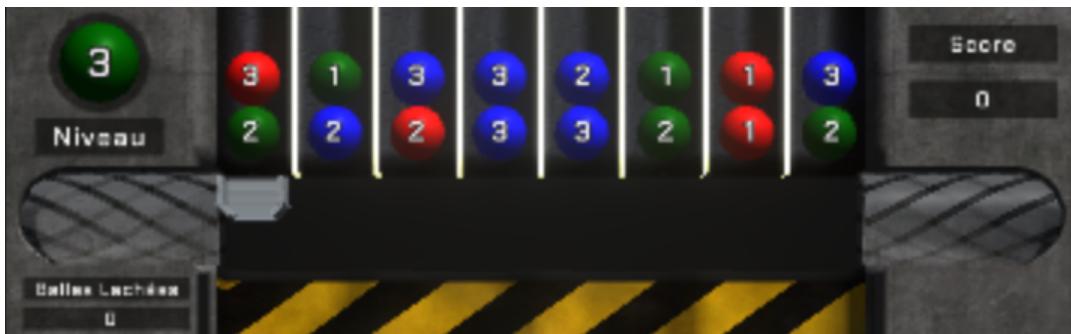
Menu de Paramètres sur le jeu

- **Musique** et **Effets sonore** : Permet de modifier le volume de la musique et des effets sonores des balles. Lorsque ce paramètre est modifié, nous offrons à l'utilisateur un aperçu du son qu'il aura avec la valeur actuelle. Le volume est donc actualisé en temps réel.
- **Anticrénelage MSAA** : Permet de modifier la configuration du MSAA (Multisample anti-aliasing) afin de réduire l'effet d'escalier à l'affichage des modèles 3D. Ce paramètre permet donc de rendre le jeu plus esthétique. Cette technologie s'applique directement sur le pipeline graphique, et non pas sur la caméra comme pourrait le faire le FXAA (Fast approximate anti-aliasing).
- **Synchronisation verticale** : Elle permet d'éviter les effets de déchirement lors de mouvements trop rapides sur l'écran (affichage d'une partie de l'écran décalé par rapport à l'autre).

- **Échelle de rendu** : allant de 0.5 à 2.0, ce paramètre permet de modifier la résolution de rendu de l'image, avant l'affichage sur l'écran. Par exemple sur un écran en 1920x1080, si nous mettons l'échelle à 0.5, l'image sera rendue en 960x540 avant d'être affiché. Ceci permet une grande amélioration des performances, mais une grosse perte en qualité. Au contraire à l'échelle 2.0 l'image sera rendue en 3840x2160, permettant de rendre le jeu très légèrement plus agréable, mais impactera grandement les performances.



Échelle de rendu à 1.0

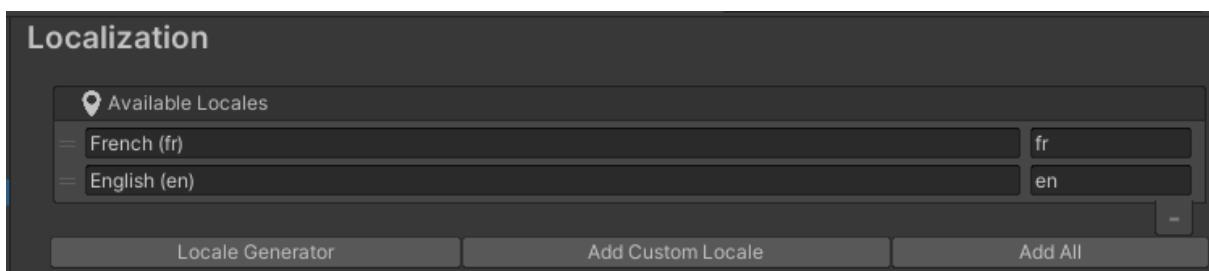


Échelle de rendu à 0.5

### 2.8.2 Langues

Nous avons décidé de mettre en place deux langues sur notre jeu : française et anglaise. Pour cela nous avons utilisé le système **Localization** de Unity, permettant de mettre en place facilement et rapidement le système de langue, et de les relier aux différents éléments graphiques.

Pour cela nous avons dû ajouter les deux langues aux paramètres du projet.



Menu Localization de Project Settings

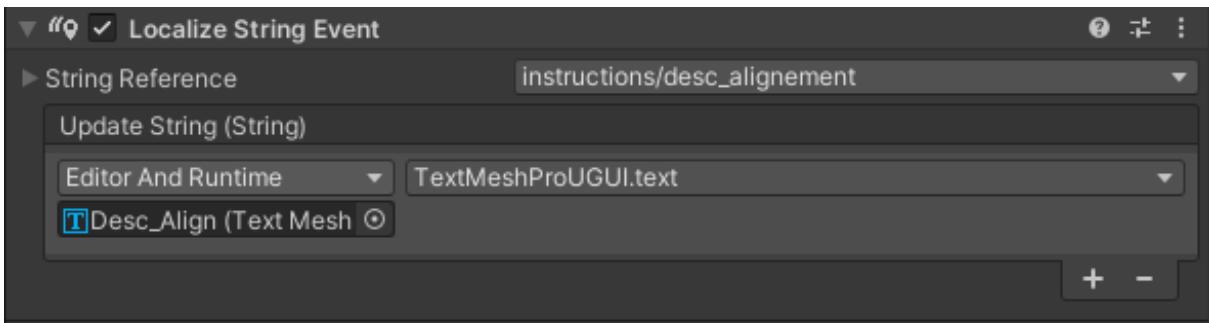
Après cela nous avons pu créer différentes tables de traduction, permettant de mettre en place des traductions dans un tableau de clés avec multiples valeurs.



Menu Localization Table de Assets Management

Pour les relier au jeu nous avons utilisé plusieurs méthodes :

- Pour les textes simples nous avons utilisé le système intégré à l'inspecteur d'Unity.
- Pour les textes plus complexes, tels que la combinaison de plusieurs phrases, ou l'ajout d'un nombre au sein d'une phrase, nous avons été contraint de récupérer dans le code les différents textes traduits et de les combiner.



Attribut de Localization dans l'inspecteur Unity pour le texte d'instruction des alignements

```

1 LocalizedString lsPlayer = new LocalizedString("Local", "joueur");
2 LocalizedString lsWin = new LocalizedString("Local", "gagne");
3 canvas.transform.Find("InfoPartieDuo/Winner_Value")
   .GetComponent<TMPRO.TextMeshProUGUI>().text = lsPlayer.GetLocalizedString() + " 1 "
   + lsWin.GetLocalizedString();

```

Nous pouvons voir que dans un premier temps nous allons chercher deux chaînes de caractères traduites dans la table "Local" avant de venir les placer sur un champ de texte nommé "Winner\_Value", en rajoutant entre les deux le numéro du vainqueur, ici le joueur 1.

Pour ce qui est du changement de langue, nous utilisons un bouton avec le drapeau de la langue actuelle.



Langue française

Langue Anglaise

```
1 public void ChangeLanguage()
2 {
3     if(LocalizationSettings.SelectedLocale.Identifier.Code
4         .Trim().ToLower().Equals("fr"))
5     {
6         LocalizationSettings.SelectedLocale = Locale.CreateLocale(new
7             → LocaleIdentifier(SystemLanguage.English));
8         SettingsData.SavedSettings.langue = "en";
9     }
10    else
11    {
12        LocalizationSettings.SelectedLocale = Locale.CreateLocale(new
13            → LocaleIdentifier(SystemLanguage.French));
14        SettingsData.SavedSettings.langue = "fr";
15    }
16    menuSettings.transform.Find("SettingsScript")
17        .GetComponent<ApplySettingsScript>().SaveToFile();
18 }
```

Cette fonction est appelée lorsque nous appuyons sur le drapeau pour changer de langue. Pour effectuer un changement de langue, nous récupérons la langue actuelle et si elle est égale à la chaîne "fr" (pour français) nous la changeons en anglais.

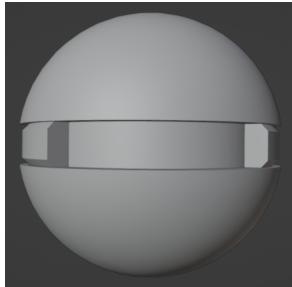
Autrement si la chaîne est égale à "en", nous changeons la langue en français.

Enfin, nous sauvegardons l'option de la langue dans le fichier de paramètres, afin de conserver la langue au prochain démarrage du jeu.

## 2.9 Interfaçage graphique avec Unity

### 2.9.1 Modélisation 3D

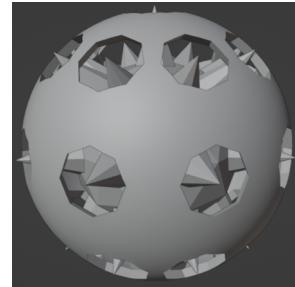
Le jeu Swing est avant tout en jeu en deux dimensions, mais malheureusement aucun des membres de notre groupe ne sait dessiner correctement. Nous nous sommes donc tournés vers la création des divers éléments de notre jeu en trois dimension avec Blender.



Modèle ZapHorizontal



Modèle Cutter



Modèle Sting

Nous avons donc modélisé de cette manière toutes les balles spéciales, le modèle du joueur ainsi que le décor. Nous avons ensuite importé nos modèles dans Unity en utilisant le format de fichier FBX qui est très facilement reconnaissable par Unity.

Une fois les modèles importés, nous avons créé pour chacun de nos modèles un "Prefab" (ou élément préfabriqué) qui permet de stocker plusieurs informations telles que :

- Un animator provenant de Unity.
- Un matériau permettant de gérer les différentes composantes : ambiante, diffuse et spéculaire.
- D'autres modèles dans le cas des objets composites.

Ces éléments préfabriqués correspondront aux objets finaux représentés dans notre jeu.

Ainsi les "Prefab" peuvent être stockés dans des attributs de classe `GameObject` pouvant ainsi la représentation d'un objet particulier pour une classe donnée. Voici un exemple d'instanciation avec la classe `BrickBall` :

```

1  public class BrickBall : SpecialBall
2  {
3      public BrickBall() : base()
4      {
5          this.BallObject = GameObject.Instantiate(Resources.Load("Prefabs/PU_BrickBall",
6              typeof(GameObject))) as GameObject;
7      }
8  }

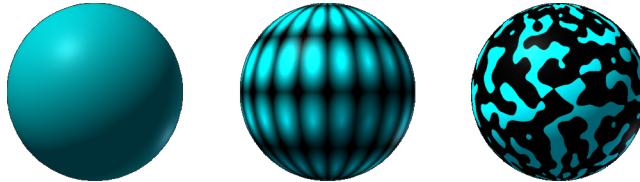
```

## 2.9.2 Shaders dynamiques

Dans la continuité de notre recherche sur la conception des balles uniques et facilement distinguables, nous avons découvert la possibilité de développer des shaders dynamiques avec ce qu'on appelle des **ShaderGraph**.

Ces shaders sont accompagnés par un système de noeud permettant de simplifier leur création. Ainsi nous avons développé plusieurs shaders pour les balles normales et spéciales :

Nous l'avions rapidement abordé dans la section sur **L'implémentation des balles** et nous allons voir plus en détail comment nous avons réalisé ces différentes textures.



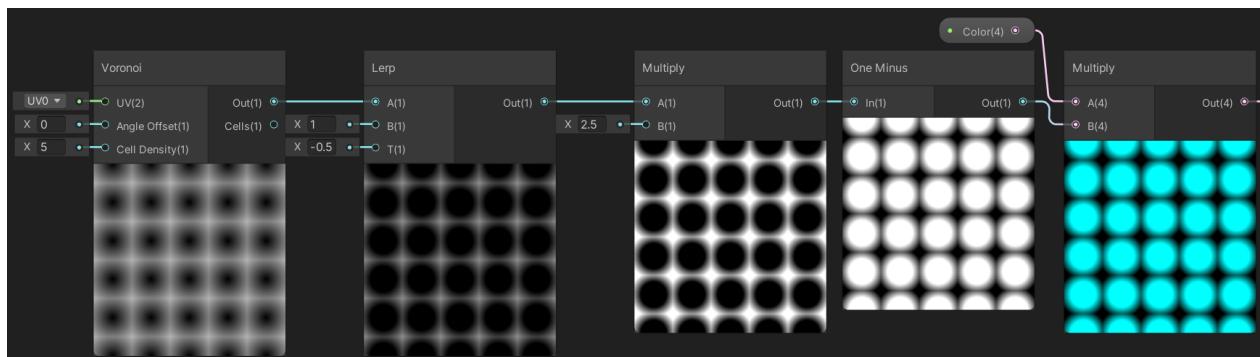
Les types de motifs avec la couleur cyan

### Shaders pour les balles normales

Notre premier shader fut réalisé avec une texture Voronoï ayant une équidistance entre chacune de ses cellules (**Angle offset = 0**). Nous avons ensuite fait en sorte que chaque cellule soit plus noircie et contrastée en utilisant une interpolation linéaire (**Lerp** pour "linear interpolation") et en multipliant les valeurs par environ 2.

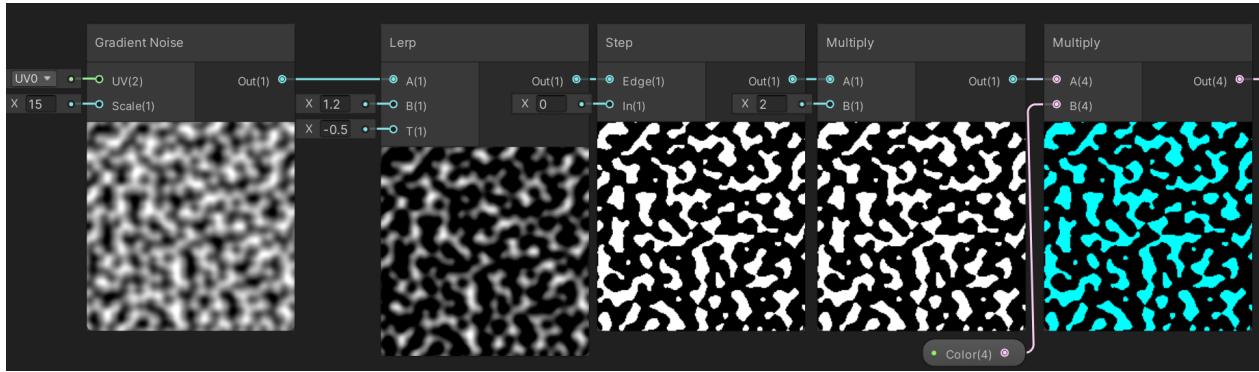
Enfin nous avons inversé les valeurs de la texture avec une fonction **One Minus** ce qui a pour effet de rendre les éléments blancs en noir et inversement. La texture résultante peut finalement être multipliée par une couleur permettant de colorer uniquement les éléments blancs.

Nous avons également appliqué un effet "carrelage" pour dupliquer notre texture sur l'axe Y pour qu'elle soit plus visible sur une sphère.



Création du shader quadrillé (Voronoi)

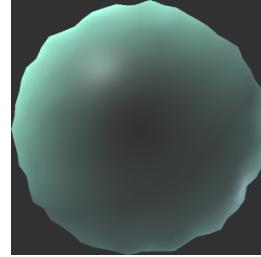
Notre second shader pour les balles normales fût relativement simple. En effet, nous avons utilisé une texture **Gradient Noise** qui, avec une interpolation linéaire (**Lerp**) et un seuillage (**Step**), nous a donné une texture sinuuse. Cette texture était parfaite dans la mesure où elle était asymétrique et distinguable de notre premier shader.



Création du shader sineux (GradientNoise)

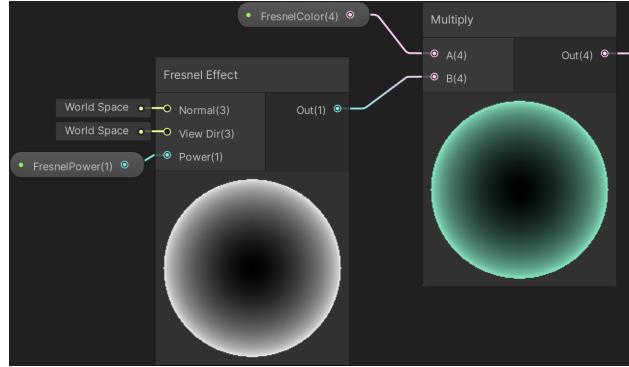
### Shaders pour les balles spéciales

Nous avons également développé des shaders pour les balles spéciales permettant de mieux les distinguer des balles normales avec notamment des animations. En voici un exemple :



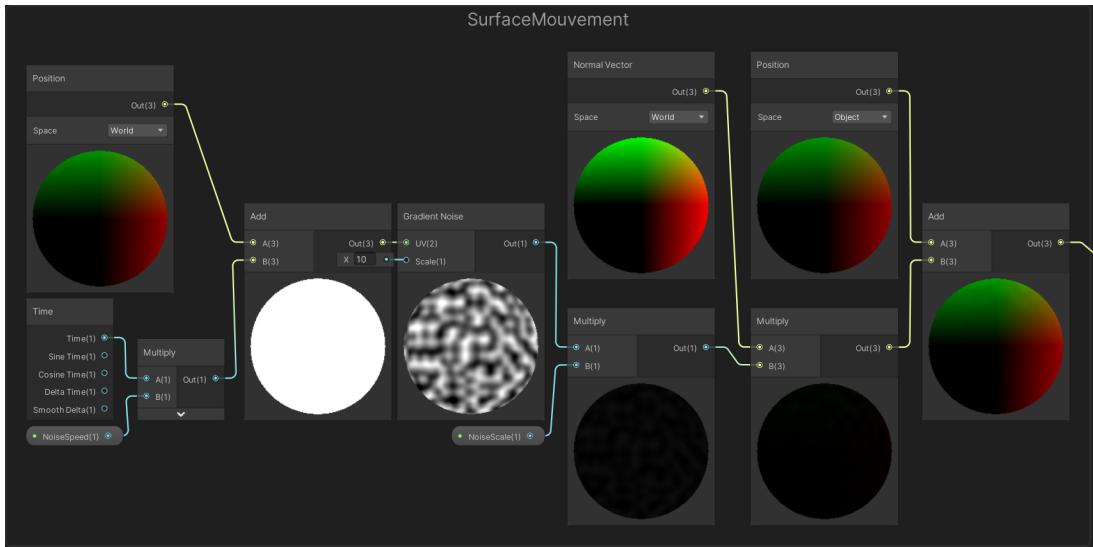
Création d'un shader dynamique "Plasma"

Pour ce faire, nous avons dans un premier temps utilisé l'effet de Fresnel permettant d'affecter une couleur brillante sur les bords de l'objet.



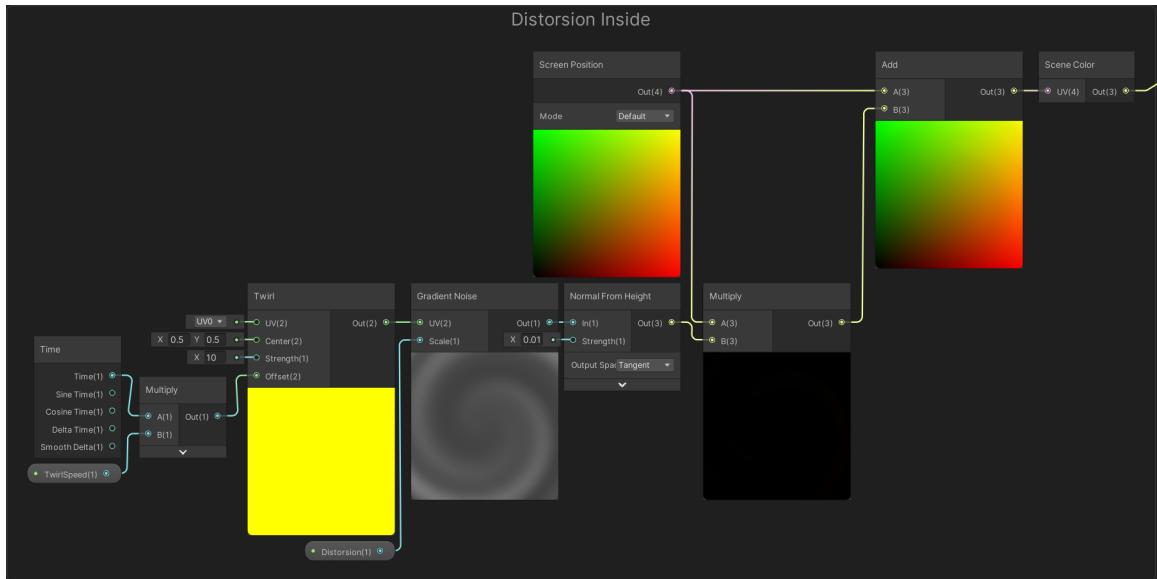
Effet de Fresnel

Puis pour produire des bords mouvementés, nous avons utilisé un **Gradient Noise** avec comme paramètre la position de chaque point de notre objet en fonction du temps. Nous avons ensuite multiplié ce gradient par la normale du vecteur en y ajoutant la position du point initial, ce qui a pour effet de déformer l'objet selon la texture en fonction du temps.



Décalage des points selon le Gradient Noise et leur normale

Enfin nous avons rajouté un effet de distorsion pour tout élément se trouvant derrière ce shader. Nous avons donc d'abord créé la distorsion avec une fonction Twirl appliquée à un Gradient Noise en fonction du temps. Puis en multipliant les normales créées par la distorsion par la position de la caméra dans l'espace monde, nous avons réussi à produire l'effet de distorsion voulu.

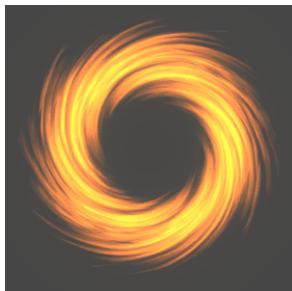


Création de la distorsion des objets se trouvant derrière le shader

### 2.9.3 Effets visuels

Pour apporter davantage d'informations aux utilisateurs concernant les actions s'opérant dans le tableau de jeu, nous avons ajouté des effets visuels.

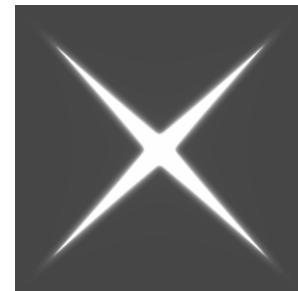
Ces effets visuels sont caractérisés par des systèmes de particules générés par Unity et peuvent être paramétrés sur plusieurs aspects (émission, forme, vitesse, couleur, etc).



Effet visuel d'un alignement



Effet visuel d'une explosion



Effet visuel d'un ZapDiagonal

Ainsi, nous avons créé un ensemble d'effets visuels permettant d'informer sur les événements suivants :

- Lorsqu'un alignement horizontal est effectué.
- Lorsqu'une balle spéciale "Bomb" explose.
- Lorsqu'une balle spéciale "Zap" est déclenchée (horizontal ou diagonal).
- Lorsqu'une transformation s'effectue sur une balle.
- Lorsqu'une balle spéciale s'auto-détruit.

Tous ces effets visuels sont ensuite gérés par une classe nommée **Effect**. Cette classe possède une méthode **Effect()** qui peut être appelée pour un effet particulier listé dans l'énumération **EffectType** :

```

1 // Effect.cs
2 public enum EffectType
3 {
4     NoEffect,
5     BombExplosion,
6     BallExplosion,
7     BallAlign,
8     ZapHorizontal,
9     ZapDiagonal,
10    BallTransform,
11    BallSmoke,
12    StarTransform
13 }
```

```

1  public class Effect
2  {
3      // Attributs
4      private GameObject _go;
5      private List<ParticleSystem> _ps;
6
7      public Effect(EffectType type, Vector3 v)
8      {
9          _ps = new List<ParticleSystem>();
10
11         // Quel est le type d'effet
12         switch (type)
13         {
14             case EffectType.BombExplosion:
15                 _go =
16                     GameObject.Instantiate(Resources.Load("Particles/PS_BombExplosion",
17                     typeof(GameObject)) as GameObject;
18                 _ps.Add(_go.transform.Find("Particle").GetComponent<ParticleSystem>());
19                 break;
20                 [...]
21         }
22
23         // Vérification
24         if (_go != null)
25         {
26             // Exécution de tous les systèmes de particules à la position donnée
27             _go.transform.position = v;
28             for (int i = 0; i < _ps.Count; i++)
29             {
30                 _ps[i].Play();
31             }
32         }
33         [...]
34     }
35 }
```

Elle possède également des attributs `GameObject` et une liste de `ParticleSystem` permettant de retrouver et de stocker avec aisance les systèmes de particules de nos "Prefab".

Ainsi lorsqu'un type d'effet visuel est passé dans la méthode, nous ajoutons les systèmes de particules à exécuter dans la liste de `ParticleSystem` et nous appliquons pour chaque élément de cette liste la méthode `Play()`.

### 2.9.4 Création des menus

Actuellement le jeu possède cinq menus :

- Le menu principal.
- Le menu des paramètres.
- Le menu des instructions.
- Le menu de pause.
- Le menu de fin de partie.

Nous avons déjà précédemment abordé le menu de paramètre dans la sous-section **Paramètres**

#### Menu principal



Menu principal

Nous pouvons observer plusieurs boutons dans le Menu principal et nous allons voir quels sont leurs utilités :

- **Jouer** : ouvre un sous-menu permettant de choisir entre le lancement d'une partie solo ou en multi-joueur.
- **Classement** : ouvre le classement en ligne. Nous y reviendrons plus en détail dans la section 2.10.
- **Instructions** : ouvre le menu d'instructions, que nous verrons un peu plus en détail ci-après.
- **Options** : ouvre le menu des options, que nous avons précédemment vu.
- **Drapeau** : permet de changer la langue du jeu.
- **Quitter** : permet de quitter le jeu.

```

1 public void Solo()
2 {
3     CrossSceneData.Multijoueur = false;
4     CrossSceneData.TransitionMainMenu = true;
5     transitionScript.LoadSceneWithTransition("Game");
6     eventSystem.gameObject.SetActive(false);
7 }
```

Voici la méthode (`Solo()`) exécutée par le bouton qui permet de lancer une partie à un seul joueur. Dans un premier temps nous passons à l'objet statique `CrossSceneData` qui contient les informations sur le jeu, puis nous chargeons la scène en effectuant une transition graphique, et désactivant le système de bouton.

### Menu d'instructions

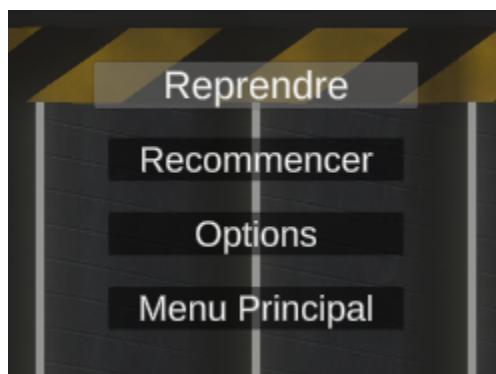


Menu des instructions

Ce menu liste les instructions du jeu, en commençant par les règles d'alignements, d'empilement et de balancement, puis des différentes balles de pouvoir et des malus.

Nous pouvons naviguer dans ce menu avec la molette de la souris, ou grâce aux touches haut et bas du clavier.

### Menu pause



Menu de pause lors d'une partie

Plusieurs boutons sont accessibles pour notre menu de pause :

- **Reprendre** : permet de reprendre la partie en cours.
- **Recommencer** : permet de recommencer une nouvelle partie.
- **Option** : ouvre le menu d'option comme pour le menu principal.
- **Menu Principal** : retourne au menu principal.

### Menu de fin de partie

Lorsqu'une partie se termine, le menu de fin de partie apparaît.



Écran de fin de partie à un joueur

L'enregistrement des scores se fait uniquement dans les parties à un joueur et seulement si le joueur possède un score supérieur à 0. L'utilisateur peut alors choisir de sauvegarder son score en entrant un pseudonyme à trois caractères. Nous avons également mis en place un système de défilement des caractères avec les flèches situées au-dessus et en dessous des caractères. Ce qui permet aux bornes d'arcades avec des contrôles restreint de rentrer des informations alphanumériques.

Nous pouvons aussi observer à droite de l'écran de fin de partie, les dix meilleurs scores enregistrés dans notre base de données. Nous y reviendrons plus en détail dans la section **Classement des meilleurs scores**.

Une fois cette étape passée, nous affichons la possibilité de recommencer une partie ou de revenir au menu principal.



Écran de fin de partie en multijoueur

En multijoueur, l'affichage du menu de fin de partie change un peu en affichant le score effectué par chacun des joueurs, et en affichant lequel des deux est vainqueur en se basant sur le score maximal.

### Transition graphique entre les scènes

Pour permettre au jeu d'avoir le temps de charger toutes les données et de rendre plus fluide le passage entre deux scènes ou à l'ouverture d'un menu ; nous avons décidé de mettre en place deux systèmes de transition.

La première est la transition d'ouverture d'un menu. Elle consiste simplement en un fondu du menu en question au-dessus du jeu ou d'un autre menu. Par exemple, l'affichage du menu pause par-dessus la scène de jeu, quand la pause est demandée ou que nous reprenons la partie.

```

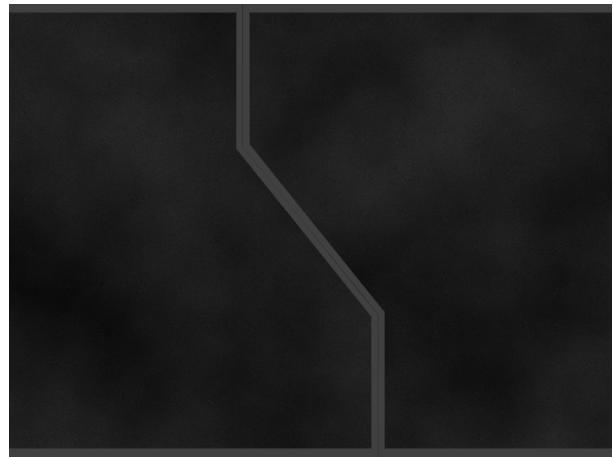
1  if (_isPause && _pauseFade < 1.0f)
2  {
3      _pauseFade += Time.deltaTime * _speedFade;
4      if (_pauseFade < 1.0f)
5      {
6          canvas.GetComponent<CanvasGroup>().alpha = _pauseFade;
7      }
8      else
9      {
10         canvas.GetComponent<CanvasGroup>().alpha = 1.0f;
11         eventSystem.gameObject.SetActive(true);
12     }
13 }
```

Cette partie de code permet de faire la transition entre le menu pause invisible et visible. À chaque image il rend de plus en plus visible le menu, en fonction du temps passé et de la vitesse d'affichage. Une fois que le menu est entièrement visible, nous activons les contrôles du menu.

La deuxième transition est celle du changement de scène, elle consiste en une animation représentant la fermeture d'une porte, quand le changement de scène est demandé, puis la réouvre quand la scène est entièrement chargée.



Début de la transition de fermeture



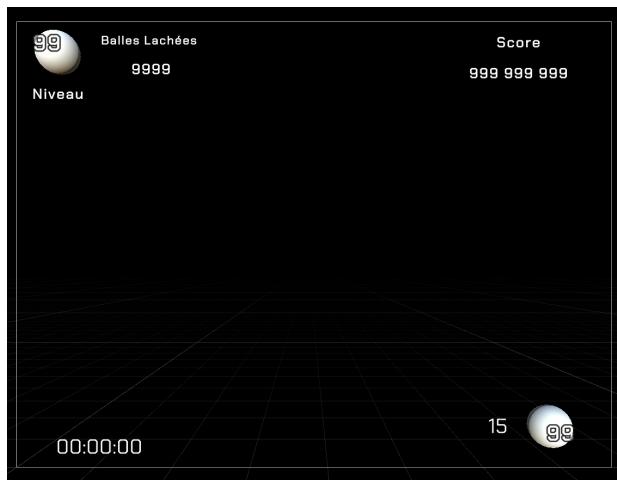
Transition fermée

### 2.9.5 Affichage tête haute (ou HUD)

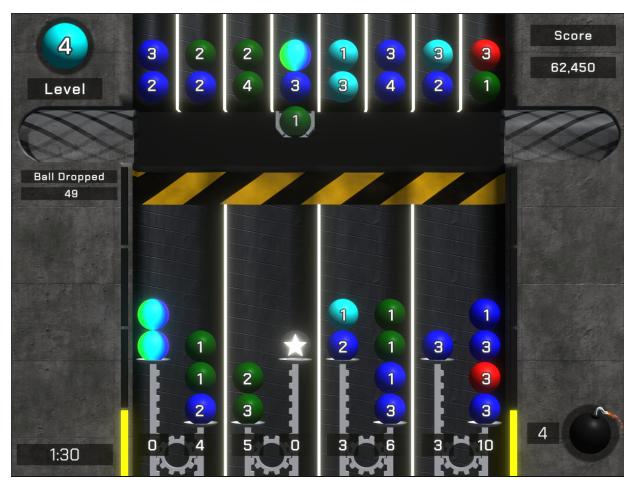
Nous avons décidé de mettre en place un système d'affichage tête haute (ou HUD pour heads-up display en anglais) pour permettre de donner des informations utiles et stratégiques à l'utilisateur.

Pour réaliser cela nous avons utilisé le système de caméra overlay d'Unity. Ce système consiste à venir donner à la caméra principale une ou plusieurs caméras secondaires, qui viennent ajouter à l'écran des informations au-dessus de l'affichage.

Dans le cas de notre projet, la caméra overlay placée sur la caméra du joueur utilise le système de projection orthographique, afin d'avoir un texte toujours plat et à la même dimension, qu'importe la distance ou la taille de l'écran.



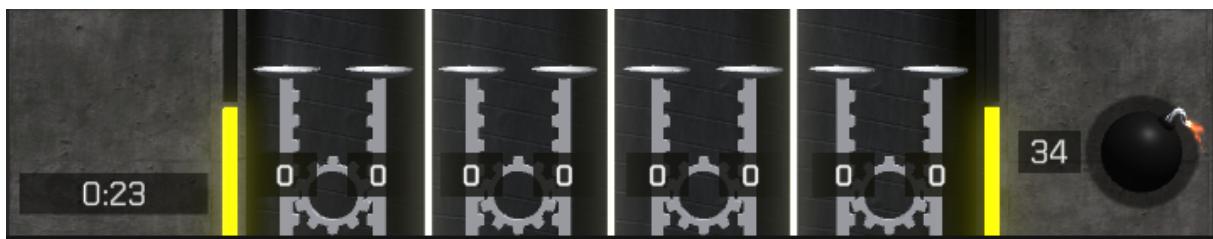
HUD de l'écran de jeu



HUD de l'écran de jeu lors d'une partie en ratio d'écran 4:3

Dans l'image ci-dessus, nous pouvons distinguer plusieurs informations grâce au HUD avec :

- Le niveau actuel, indiquant également le poids maximal possible des balles mais également la dernière couleur ajoutée.
- Le nombre de balles lachées dans le tableau de jeu.
- Le score actuel du joueur.
- Le temps de jeu pour la partie actuelle.
- Le nombre de balles à poser avant que la balle spéciale affichée en bas à droite apparaisse dans le tableau de prédition.



HUD du poids des balances

Une autre information essentielle à la fluidité du jeu est l'affichage du poids total de chaque colonne pour pouvoir effectuer rapidement des choix selon les balancements que nous voulons effectuer.

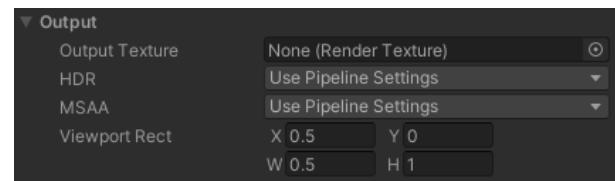
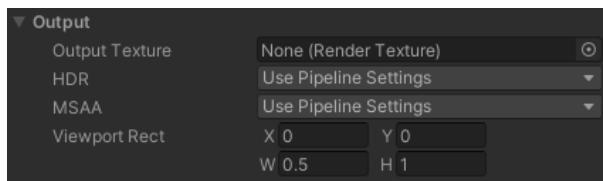
## 2.10 Implémentation de l'écran partagé

Pour pouvoir effectuer des parties à plusieurs, nous avons décidé de mettre en place un système d'écran partagé.

Pour cela nous avons dû utiliser le système d'Unity permettant de rendre une caméra sur une seule partie de l'écran.  
Si pour un écran 1920x1080, nous avons deux caméras ayant un rectangle de vue de taille  $X = 0.5$  et  $Y = 1.0$  (la moitié de 1920, donc 960x1080), et que nous décalons la deuxième caméra de 0.5 en X (donc 960 pixels), nous obtenons un écran partagé en deux caméras distinctes.

Nous pouvons étendre ce principe à n caméras si nous le souhaitons.

Par exemple, si nous avons quatre caméras de taille  $X = 0.5$  et  $Y = 0.5$ , celles de la première ligne, décalées de 0.5 en Y, et celles de droites décalées de 0.5 en X.



Résultat de l'affichage de deux caméras à des distances différentes d'un carré

Pour implémenter cette fonctionnalité dans le jeu, nous avons décidé de créer les caméras dans le script de la création d'un environnement de jeu pour un joueur.

Comme nous l'avons vu plus tôt un joueur possède deux caméras, une principale et une d'overlay pour l'HUD.

Ainsi, chaque joueur possède ses propres caméras. S'il n'y a qu'un seul joueur les caméras ont un rectangle de vue de X = 1 et Y = 1, sinon les caméras sont paramétrées comme vues au-dessus.

```

1  if (multijoueur)
2  {
3      if (idJoueur == 0)
4      {
5          _mainCamera.rect = new Rect(0.0f, 0.0f, 0.5f, 1.0f);
6          _overlayCamera.rect = _mainCamera.rect;
7          _zone.Zone.transform.position = new Vector3(-10.0f, 0.0f, 0.0f);
8          _zone.Zone.name = "Player1";
9      }
10     else if (idJoueur == 1)
11     {
12         _mainCamera.rect = new Rect(0.5f, 0.0f, 0.5f, 1.0f);
13         _overlayCamera.rect = _mainCamera.rect;
14         _zone.Zone.transform.position = new Vector3(10.0f, 0.0f, 0.0f);
15         _zone.Zone.name = "Player2";
16     }
17 }
```

Voici ci-dessus le code permettant de mettre en place l'écran partagé.

D'abord nous vérifions l'identifiant du joueur, s'il est égal à 0, la caméra est à gauche, sinon à droite.

Ensuite nous donnons la taille du rectangle de vue à la caméra principale, puis nous l'attribuons aussi à la caméra d'overlay.

Enfin nous déplaçons la zone de jeu sur un côté pour éviter que les deux terrains soient affichés en même temps chez les deux joueurs.

L'attribution du nom sert essentiellement à l'affichage Unity pour le développement.

## 2.11 Classement des meilleurs scores

Nous avons décidé de mettre en place un système de classement des meilleurs scores pour notre jeu Swing. À la fin d'une partie, diverses informations sont récupérées puis stockées sur une base de données en ligne à l'aide d'une communication client-serveur.

### 2.11.1 Coté serveur

Pour le serveur nous avons décidé d'utiliser Node.js avec le framework "Express.js" afin de créer un service web pouvant accepter bien plus de clients en simultané que PHP. Cela permet également de coupler le service web à une base de données MySQL, afin de pouvoir stocker chacun des scores.

Le service web et la base de données sont hébergés sur notre propre serveur Ubuntu et nous n'avons pas jugé nécessaire l'utilisation du HTTPS pour ce service.

La base de données ne contient qu'une seule table nommée **score** :

swing score	
#	<b>id</b> : int
#	<b>name</b> : varchar(32)
#	<b>score</b> : bigint
#	<b>horodatage</b> : varchar(16)

Table Score via PhpMyAdmin

- **id** est la clé primaire. Elle est auto-incrémentée et stockée sous forme d'un entier.
- **name** correspond au nom de l'utilisateur relié au score. Il est stocké sur 32 caractères et actuellement les noms enregistrés sont limités à seulement 3 caractères, mais nous pouvons toujours modifier cela pour permettre des noms plus longs.
- **score** correspond au score que le joueur a réussi à atteindre. Il est stocké sur un **bigint** correspondant à un **long long int** en c++. (soit  $[-9 * 10^{18}, 9 * 10^{18}]$ ).
- **horodatage** est la date d'enregistrement du score. Elle est stockée sur 16 **char**, au format **AAAAMMJJ-hhmmss** et permet de différencier les scores identiques en les triant du plus ancien au plus récent.

Le service web est exécuté automatiquement au démarrage du serveur à l'aide de PM2 qui est un gestionnaire de processus pour Node.js.

name	mode	pid	uptime	status	cpu	mem
leaderboard	fork	486990	84m	online	0%	53.8mb

Status du service via PM2

Quand le service web démarre, il se connecte dans un premier temps à la base de données MySQL en utilisant des crédentails de connection et les certificats SSH du serveur MySQL.

```

1 var con = mysql.createConnection({
2   host: "localhost", user: "swing_user", password: "?",
3   database: 'swing',
4   ssl: {
5     ca: fs.readFileSync(__dirname + '/certs/ca.pem'),
6     key: fs.readFileSync(__dirname + '/certs/client-key.pem'),
7     cert: fs.readFileSync(__dirname + '/certs/client-cert.pem')
8   }
9 });

```

Avant d'ouvrir le port d'écoute (par défaut fixé à 8000).

```

1 app.listen('8000','0.0.0.0',()=>{
2   console.log("server is listening on 8000 port");
3 });

```

Nous pouvons faire 3 requêtes différentes au service web. Chacune d'entre elles retourne un objet au format JSON et le client reçoit uniquement le nom, le score ainsi que l'horodatage. Les ids des enregistrements restent privés.

- **/GetLeaderboard** : méthode GET, permet de récupérer le classement des scores en entier, sans limite dans la lecture. Le résultat peut être très lourd selon la quantité de scores présents dans la base de données.
- **/Top10** : méthode GET, permet de récupérer les 10 meilleurs scores de la base de données. les données passées sont relativement légères en mémoire.
- **/AddScore** : méthode POST, permet de rajouter un score à la base de données en passant dans le corps de la requête les informations nécessaires telles que le nom, le score et l'horodatage. Elle retourne par défaut les 10 meilleurs scores (top10).  
L'utilisation de la méthode POST, permet d'éviter les possibles ajouts de scores frauduleux si nous connaissons l'URL.

Chaque réponse est incluse dans un objet JSON "data" et la réponse ressemble à : { 'data' : données }

Si le serveur est lancé, nous pouvons rentrer directement l'URL des requêtes avec la méthode GET dans le navigateur et visualiser la réponse au format JSON. (exemple : <http://www.flareden.fr:8000/top10>)

```

    ▼ data:
      ▼ 0:
        Name:      "MAR"
        Score:     208018900
        Horodatage: "2022420_165419"
      ▼ 1:
        Name:      "MAR"
        Score:     114101050
        Horodatage: "2022420_16120"

```

Exemple de réponse JSON

### 2.11.2 Coté client

Pour le client, nous avons créé une classe d'instance **Leaderboard**, qui contient deux structures :

- **ScorePlayer** : Correspond aux données reçues du serveur, contenant nom, score et horodatage.
- **ListScorePlayer** : Correspond au tableau de ScorePlayer envoyé par le serveur.

Mais aussi une interface de callback **ILeaderboardChange**, permettant la mise à jour de l'affichage avec les données reçues.

La classe **Leaderboard** peut réaliser plusieurs choses, telles que l'envoi d'un score, récupérer le top 10, ou encore générer le préfabriqué permettant l'affichage des scores.

```

1  public async void SendScore(ScorePlayer scorePlayer)
2  {
3      Dictionary<string, string> values = new Dictionary<string, string>{
4          { "Name", scorePlayer.Name },
5          { "Score", scorePlayer.Score.ToString() },
6          { "Horodatage", scorePlayer.Horodatage}
7      };
8      FormUrlEncodedContent content = new FormUrlEncodedContent(values);
9
10     HttpClient httpClient = new HttpClient();
11     HttpResponseMessage response = await httpClient.PostAsync(LEADERBOARD_URL +
12         "AddScore/", content);
13     string strResponse = await response.Content.ReadAsStringAsync();
14     httpClient.Dispose();
15
16     ListScorePlayer res = JsonUtility.FromJson<ListScorePlayer>(strResponse);
17     callback.OnTop10Receive(res.data);
}
```

Nous pouvons voir si dessus la fonction asynchrone permettant d'envoyer un score, elle vient dans un premier temps créer un dictionnaire permettant de former le JSON nécessaire au serveur, en y attribuant le nom, le score et l'horodatage, avant de venir l'encoder au bon format pour le transfert. Puis, nous créons une connexion au service avant de venir y envoyer nos données. Enfin, nous récupérons la réponse qui correspond au top10, mis à jour avec le nouveau score si besoin, avant de venir traduire en ListScorePlayer et de l'envoyer au callback pour l'affichage.

Rank	Name	Score
1	MAR	208018900
2	MAR	114101050
3	ABA	73040150
4	GOD	42502400
5	ABA	20924650
6	MAR	8746738
7	ABA	938950
8	FLO	890350
9	FLO	813150
10	BAB	527100

Menu de Classement sur le jeu

```

1 public void askNameValidate(AskNameScript askNameScript, string name)
2 {
3     System.DateTime dt = System.DateTime.Now;
4     string horodatage = dt.Year + "" + dt.Month + "" + dt.Day + "_" + dt.Hour + "" +
5         → dt.Minute + "" + dt.Second;
6
7     Leaderboard lb = Leaderboard.GetInstance();
8     lb.AddListener(this);
9     if (this._askNameP1.transform.Find("AskNameScript").GetComponent<AskNameScript>() ==
10        → askNameScript)
11     {
12         this._askNameP1.gameObject.SetActive(false);
13         lb.SendScore(new Leaderboard.ScorePlayer(name, this._scoreP1, horodatage));
14         this.eventSystem.SetSelectedGameObject(
15             → this.canvas.transform.Find("Buttons/Button_Restart").gameObject);
16     }
17 }
```

Exemple d'appel de SendScore() lorsque l'utilisateur à rentré son nom après un Gameover.

```

1 public void ShowLeaderboard()
2 {
3     Leaderboard.GetInstance().SetListener(this);
4     Leaderboard.GetInstance().FetchTop10();
5     eventSystem.GetComponent<EventSystem>().SetSelectedGameObject(
6         → buttonBackLeaderboard.gameObject);
7     _leaderboardShow = true;
8     leaderboardMenu.SetActive(true);
9 }
```

Exemple d'appel de FetchTop10(), lorsque l'utilisateur demande d'afficher le classement dans le menu principal.

# Chapitre 3

## Articles

### 3.1 Étude des techniques d'interaction pour les environnements 3D interactifs

Article étudié par Florentin DENIS

*A Survey of Interaction Techniques for Interactive 3D Environments par acek Jankowski et Martin Hachet*

Dans un environnement 3D nous pouvons trouver 3 interactions dites universelles :

- Navigation : Déplacement de l'utilisateur ou d'un objet dans le monde.
- Sélection et Manipulation : Sélection d'un objet dans le monde et modification de celui-ci dans l'espace.
- Système de contrôles : Communication entre l'utilisateur réel et le système virtuel.

Dans la navigation l'utilisateur se déplace selon la règle des six dégrées de liberté (6DOF) (trois pour la position et trois pour la rotation).

Les problèmes majeurs de la navigation dans un environnement 3D sont que nous utilisons pratiquement exclusivement des périphériques 2D afin de visualiser ou envoyer des informations, comme par exemple la souris qui ne se déplace que sur les axes X et Y.

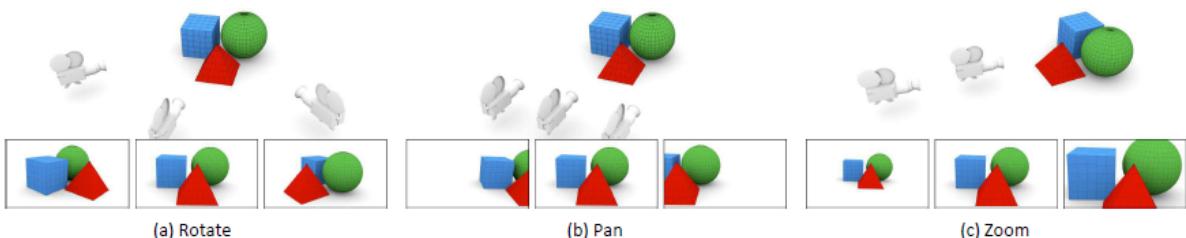
Mais aussi que nous pouvons souhaiter naviguer dans un immense univers 3D ou vouloir visualiser de près un objet très précis.

À cela peut se rajouter un problème qui peut différer selon l'utilisateur : celui de trouver son chemin dans le monde, qui peut être grandement impacté par le champ de vision ou le manque d'information sensitives, comme la stimulation de l'oreille interne.

Pour la navigation nous pouvons trouver quatre familles de mouvements :

- Généraux : Mouvement d'exploration tel que marcher dans un environnement.
- Ciblés : Se déplace autour d'un objet spécifique.
- Coordonnée spécifique : Se déplace en respectant une position et une rotation.
- Trajectoire spécifique : Se déplace sur la position et l'orientation d'une trajectoire, comme un mouvement de caméra au cinéma.

*J. Jankowski & M. Hachet / A Survey of Interaction Techniques for Interactive 3D Environments*



Pour les déplacements généraux plusieurs techniques existent :

- Déplacement, rotation et Zoom : utilise un périphérique pour via un seul point permettre le déplacement, la rotation et le zoom.

Par exemple : appuyer sur le bouton et déplacer la souris pour tourner, la molette pour zoomer ou dézoomer et appuyer sur le bouton gauche en déplaçant la souris pour se déplacer.

- Déplacement avec l'écran : utilisation d'un ou plusieurs points d'appui afin de réaliser les trois tâches vues précédemment.

Par exemple : le mouvement d'un doigt sur l'écran permet le déplacement, un pincement le dézoom, et le déplacement de deux doigts la rotation.

Les déplacements ciblés possèdent aussi plusieurs méthodes :

- Point d'intérêt : déplace l'utilisateur selon l'endroit où il appuie, s'il s'agit d'un objet, le déplacement s'effectue de façon logarithmique. Ainsi, plus nous sommes loin plus la caméra va vite, plus nous nous rapprochons plus la caméra ralentit.

- Dessin de chemin : déplace un élément selon un chemin dessiné d'avance par l'utilisateur (méthode préférée par les utilisateurs mais plus lente).

- Hyperliens et marque-page : permet de se déplacer dans l'environnement grâce à des positions pré enregistrées, cette technique peut provoquer une désorientation mais elle est extrêmement rapide.

- Déplacement par requête : demandé à l'utilisateur où il souhaite aller par des mots, par exemple "Paris".

Le déplacement spécifié quant à lui consiste simplement en la saisie des coordonnées exactes en X, Y et Z, afin de pouvoir déplacer instantanément l'objet ou la caméra à cette même position.

Pour les trajectoires spécifiées, il existe également plusieurs techniques :

- Déplacement guidé / contraint : l'utilisateur peut se déplacer selon un chemin prédéterminé à l'avance.

- Déplacement cinématographique : utilisation de la caméra permettant de suivre un élément en mouvement dans l'environnement selon plusieurs orientations possibles.

- Navigation assistée : système de déplacement avec guide, par exemple, lorsque l'utilisateur arrête de bouger dans l'environnement, nous déplaçons son regard vers un élément important ou une direction spécifique.

- Navigation adaptative : adapte l'environnement aux besoins de l'utilisateur, souvent utilisé pour le web.

Pour résoudre le problème consistant à trouver son chemin, nous pouvons utiliser l'environnement lui-même ou une interface afin de guider l'utilisateur. Mais cela se base essentiellement sur sa capacité à se repérer et à se diriger.

Par exemple, nous pouvons fournir une carte, des panneaux de direction ou encore rendre semi-transparente certaine surface pour pouvoir visualiser l'environnement plus facilement.

Pour tout ce qui est de la sélection, nous pouvons utiliser le système de Ray casting, permettant depuis l'espace 2D de notre écran, de lancer un rayon dans la scène et sélectionner le premier objet rencontré.

Pour la manipulation nous pouvons fournir à l'utilisateur des outils simples pour permettre de modifier les objets sélectionnés, par exemple une touche permettant la rotation sur l'axe X ou le déplacement sur l'axe z.

Nous pouvons aussi utiliser la manipulation par utilisation des doigts, par exemple, nous sélectionnons un objet sur la scène avec un doigt, puis pouvons le déplacer, dimensionner et le tourner avec simplement deux doigts positionnés à côté de l'objet.

Il existe aussi les manipulation pseudo physique permettant de prendre en compte certaines caractéristiques physiques des objets.

Par exemple, un objet peut être déterminé comme posable, avec une surface de pose telle une assiette, un autre objet comme support, qui peut donc recevoir un objet posable, tel que le sol, ou encore un mélange des deux, comme une table.

Dans les systèmes de contrôle, nous pouvons différencier deux paradigmes :

- Le paradigme WIMP : des contrôles utilisant un système de fenêtre standard avec des icônes, des menus, et un système de pointer, tel qu'une souris, les instructions sont faites pour être simple et clair. Beaucoup de jeu de la catégorie **Point and Click** utilise ce système, correspondant parfaitement au style du jeu, mais nous pouvons aussi trouver des jeux tels que World of Warcraft, qui n'est pas un Point and Click.
- Le paradigme Post-WIMP : WIMP étant surtout fait pour les applications 2D, la liaison entre une application 3D pouvait devenir complexe et contre-intuitif pour l'utilisateur, ainsi sont apparus les systèmes de contrôle permettant de prendre en compte les gestes, le suivi de mouvement ou encore la reconnaissance de parole.

Pour savoir si notre interface est utilisable nous avons plusieurs méthodes : la première consiste à parcourir les différents menus avec un scénario ou un raisonnement logique, par exemple via des heuristiques, pour voir si nous arrivons au résultat attendu ou s'il y a eu un problème dans la logique.

Une deuxième consiste simplement à faire essayer l'interface à un utilisateur, et à analyser ses résultats : si l'utilisateur prend beaucoup de temps à comprendre un fonctionnement alors l'interface ou les manipulations sont difficiles et peu utilisables.

Une dernière méthode est l'utilisation de sondage, pour savoir si un utilisateur aime ou non certaine partie de l'application et de l'interface.

De nos jours les périphériques de contrôles et d'affichages évoluent, ainsi nous nous retrouvons avec plus que simplement un clavier et une souris, mais nous pouvons aussi avoir des applications contrôlées par appui, position de la tête ou expression faciale, utilisation de jouet connecté, communication avec le cerveau, ou encore de nouvelles technologies d'affichage tel que les écrans 3D.

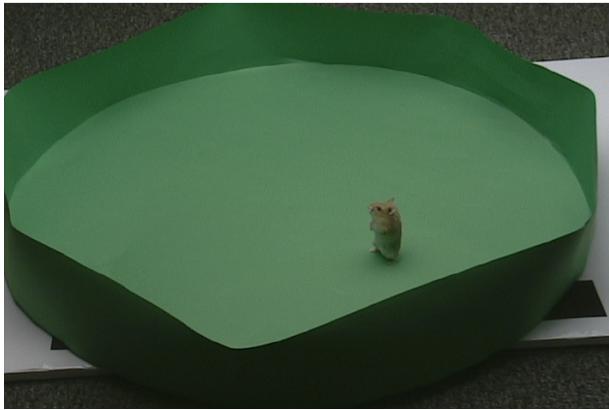
Nous avons utilisé certains éléments de cette étude dans notre projet tels que le déplacement guidé d'objet, l'utilisation de Ray Casting pour les interfaces avec boutons ou encore le test de l'interface par des utilisateurs.

### 3.2 Animation contrôlée des sprites vidéo

Article étudié par Marius JENIN

*Controlled Animation of Video Sprites par Arno Schödl et Irfan A. Essa*

Ce papier scientifique traite de la génération de sprites réalistes et de l'animation de ceux-ci. Le but principal est de récupérer les sprites à partir de sujet réel et de pouvoir par la suite animer pour n'importe quel déplacement. Cela est utile lorsque nous voulons animer des sujets qui ne peuvent pas être contrôlés (comme des animaux par exemple) et que l'animation en 3D est trop coûteuse. Pour expliciter le propos, les auteurs ont pris comme sujets un hamster et une mouche.



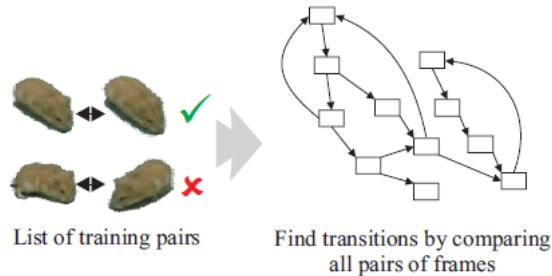
Capture des mouvements des sujets sur fond vert

La première étape consiste à réaliser une capture vidéo (séquence d'images) des sujets. Pour cela, on place le sujet sur un fond vert et on le filme en train de se déplacer.

Pour la seconde étape, il faut créer les sprites à partir de ces images. Tout d'abord, il y a un filtrage des images traitables (par exemple si le hamster est trop proche du bord de la zone de test l'image n'est pas utilisable). Pour récupérer les sprites il faut supprimer l'arrière plan, faire de l'incrustation (possible avec les fonds verts) et corriger la perspective. On peut ensuite copier chaque sprite obtenu et lui appliquer un effet miroir (dans le cas d'un sujet symétrique). On a ainsi la totalité de nos sprites utilisables.

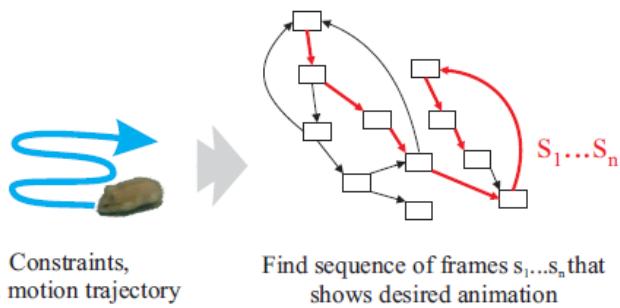
La troisième étape est cruciale. Elle consiste à trouver des paires de sprites qui vont pouvoir être "compatibles". Cela va déterminer si deux sprites peuvent s'enchaîner dans une animation. Pour cela les auteurs ont fait du machine learning et ont entraîné des modèles pour qu'ils puissent eux-mêmes déterminer les paires de sprites compatibles.

La quatrième étape va utiliser ces paires de sprites pour générer un graphe orienté décrivant les enchaînements possibles des sprites. Chaque noeud est un sprite et chaque arête est une transition possible.

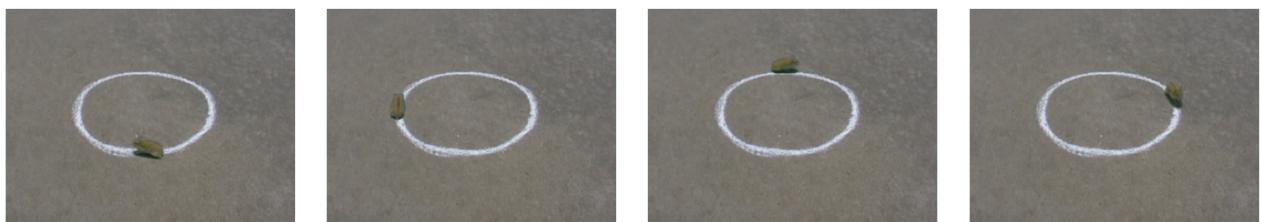


Troisième et quatrième étapes consistant à générer des paires de sprites compatibles et de créer un graphe avec ces derniers

Enfin la cinquième et dernière étape consiste à trouver le bon enchaînement de sprites pour décrire un déplacement donné. Cela est fait en calculant le coût d'une séquence de sprites pour un déplacement et en voulant le minimiser. Il y a tout d'abord le coût de la proximité entre deux sprites. Celui-ci permet d'augmenter la fluidité d'un mouvement. Les auteurs ont décrit d'autres fonctions de coût à ajouter à celui-ci : on retrouve la contrainte sur la position, la contrainte de trajectoire, la contrainte anti-collisions et la contrainte sur l'image.



Cinquième étape consistant à trouver la bonne séquence de sprite à partir d'un déplacement et d'un graphe de sprite



Exemple de résultat pour une trajectoire circulaire pour le hamster

Nous n'avons pas utilisé de méthodes comme celle-ci pour la simple raison que nos seules animations sont réalisées sur des balles sans rotations. Il n'y a pas de raisons à effectuer un processus de la sorte si nos modèles sont inertes et simples.

### 3.3 Détection de collision voxel

Article étudié par Khélian LARVET

*Voxel collision detection for virtual environments par S.M. Lock et D.P.M Wills*

Cette revue scientifique aborde la notion de détection des collisions pour des environnements virtuels avec une nouvelle approche intéressante basée sur des voxels. La plupart des algorithmes actuels sont lents et ne permettent pas de rendre une scène réaliste avec une fréquence d'images stable.

L'efficacité est donc essentielle dans un environnement virtuel, sinon la nature interactive est perdue. Les environnements virtuels sont prometteurs pour développer de nouvelles interactions homme-machine mais pour ce faire, le réalisme perçu par un utilisateur doit être maximisé.

Ainsi, deux caractéristiques principales, que nous considérons comme des acquis doivent être implémentées dans les univers virtuels :

- La **solidité** qui est caractérisée par des lois physiques permettant d'éviter qu'un objet passe à travers un autre. Ce qui rend la simulation plus naturelle.
- La capacité **d'intéragir** avec des objets.

La détection de collision dans un environnement multi-corps se déroule généralement en deux phases :

- La **Narrow Phase**; Cette phase implique la détection de collision par traitement de paires d'objets. Ainsi si deux objets s'interpénètrent, la **Narrow Phase** le signalera et donnera tous les points de collision.
- La **Broad Phase**; Elle consiste à élargir le nombre de paires d'objets qui doivent subir test de collision par traitement de paires.

Ces deux phases doivent pouvoir être exécutées à une fréquence quasi constante et la précision doit être adaptée à l'application et les besoins perceptifs des utilisateurs. Ainsi la détection de collision multi-corps est très dépendante de la capacité à analyser des collisions entre des paires.

Le modèle de corps rigide est actuellement la représentation géométrique la plus couramment utilisée dans la détection des collisions. Mais ce modèle reste très coûteux en calcul et donc lent. En effet, de nombreux cas particuliers existent, nécessitant des traitements complexes.

Plusieurs améliorations ont donc été proposées pour tenter de simplifier les calculs des corps rigides :

- Le "**back-face culling**" permettant de diminuer le nombre de polygones soumis à des tests de collision.
- Les **décompositions spatiales** qui en utilisant une arborescence d'objets permet de localiser les zones de collision avant d'effectuer des tests de collision ("Quadtrees" ou "Octrees").
- L'utilisation des **niveaux de détail** pour réduire la précision des collisions pour des objets situés à grande distance de la caméra.

Toutes les techniques mentionnées ci-dessus utilisent des collisions polygone-polygone, qui complexifient les algorithmes. Cela reste donc inapproprié pour les applications d'environnement virtuel qui exigent des taux de mise à jour élevés pour les objets complexes.

Il faut donc trouver d'autres représentations autres que polygonale pour effectuer des détections de collision. C'est ici que la **représentation basée sur les voxels** entre en jeu.

L'approche basée sur les voxels pour la détection des collisions exploite davantage la mémoire par rapport au processeur. En effet, nous effectuons un ensemble de prétraitement qui seront ensuite stockés dans la mémoire pour éviter de faire peser une charge trop lourde sur les demandes de traitement. Ces prétraitements résident dans le stockage de ce qu'on appelle une "VoxMap" et un "Point Shell".

**Le "Voxmap"** est donc le nom donné au volume de représentation d'un objet qui est un réseau régulier de voxels cubiques en trois dimensions.

Les Voxmaps sont conçues directement pour des objets et permettent de distinguer l'intérieur de l'extérieur d'un objet.

**Le "Point Shell"** est le nom donné à la seconde représentation surfacique d'un objet. Il se compose d'un nombre de points uniformément répartis sur la surface de l'objet. Ces éléments servent à effectuer une recherche dans le "Voxmap" d'un autre objet pendant le test de collision. Théoriquement il est quasi impossible qu'un voxel puisse ne pas être détecté lors d'une collision par les "Point Shell".

Ainsi la détection de collision s'effectue sur des paires d'objets en analysant la "Voxmap" de l'un et les "Point Shell" de l'autre :

On transforme les différents "Point Shell" d'un objet dans l'espace "Voxmap" du second objet. Si un "Point Shell" est contenu dans un voxel et qu'il est marqué comme appartenant à l'objet, alors une collision s'est produite en ce point.

Ainsi via cette représentation, nous supprimons les collisions polygone-polygone grâce à l'utilisation des "Voxmap" pré-traités qui permettent de résoudre la détection de collision par une simple recherche de "Points Shell" dans les voxels d'une "Voxmap".

Cette technique basée sur les voxels est très intéressante mais malheureusement notre projet ne requiert que des mouvements précis qui ne reposent en aucun cas sur de la physique.

### 3.4 Génération des niveaux à partir de vidéos de jeu

Article étudié par Benjamin PRE

*Toward Game Level Generation from Gameplay Videos par Matthew Guzdial et Mark O. Riedl*

Ce quatrième article concerne la génération de niveaux à partir de vidéos de jeu. J'ai trouvé cet article pertinent car pour notre projet, nous n'avions pas eu beaucoup de ressources. Nous nous sommes principalement basé sur d'anciennes vidéos de gameplay du jeu. Ainsi la génération des niveaux de manière procédurale à partir des vidéos aurait pu être intéressante.

L'article présente une solution de machine learning afin de générer des niveaux de jeu permettant au niveau d'être le meilleur possible. Il se base notamment sur le jeu *Super Mario Bros* sur le nombre d'interactions qu'un joueur effectue sur une portion du niveau. En fonction du temps d'interaction d'un joueur sur une portion de niveau, on peut ainsi en déterminer sa difficulté, et agir sur cette portion en conséquence, afin d'augmenter ou de réduire la difficulté de cette section.

Pour notre projet, la solution de l'article n'est pas applicable, puisqu'il se concentre sur la reconnaissance des objets de terrains, qu'on peut par exemple voir sur des jeux de plateforme, et sur leur placement pour varier la difficulté. Cependant, dans le cadre de notre projet, le but n'est pas de savoir où placer des structures, mais de savoir quelle bille générer pour avoir une difficulté adéquate. Comme détaillé dans le rapport, notre projet suit une courbe de difficulté mathématique, et nous n'avons donc pas pu implémenter de méthode similaire à l'article.

# Chapitre 4

## Bilan et difficultés rencontrées

### 4.1 Avancement du projet

À l'heure actuelle, notre application est complètement fonctionnelle avec des composants modulables et suit au mieux le cahier des charges qui a été fixé.

Ainsi nous pouvons proposer aux utilisateurs de jouer seul ou à plusieurs avec plus de vingt balles spéciales et une difficulté ajustée pour récompenser les joueurs qui prennent des décisions rapidement. Les trois règles du jeu original sont également implémentées et plusieurs paramètres sont mis à disposition pour faire en sorte que la plupart des ordinateurs puissent exécuter notre application.

Nous pouvons également changer certaines facettes des règles du jeu très rapidement en changeant quelques paramètres. Cela peut être utile lors d'éventuels ajouts de modes de jeu.

Par manque de temps, nous avons décidé de développer et d'implémenter un tableau de score à la place d'un système d'intelligence artificielle capable de choisir les meilleures actions pour un état de jeu donné.

### 4.2 Changements majeurs

Notre seul et unique changement majeur s'est effectué lors de la décision de la mise en place de la neuvième ligne. En effet, lors de la création du tableau de jeu et des divers éléments, nous avions donné une hauteur fixe de huit.

La neuvième ligne est donc une "fausse" ligne car l'objectif de son implémentation est la possibilité pour que le joueur puisse poser des balles qui vont instantanément disparaître. Comme par exemple lorsqu'une balle provoque la destabilisation d'une balance sur la colonne la plus lourde et ainsi laisse deux emplacements de plus libres sur sa colonne. Ou encore lorsqu'une balle de pouvoir s'exécute et modifie instantanément le tableau de jeu.

Si une balle reste sur cette ligne alors cela signifie que la partie est perdue. Ce changement a été très important dans l'animateur et dans l'implémentation des balances.

### 4.3 Difficultés rencontrées

Notre première difficulté a été la migration soudaine de notre projet de *Unity Collab* à *Plastic SCM*. Cette migration fut imposée par Unity sans notre accord et nous a quelque peu bouleversés car nous avons dû faire preuve d'une rapide adaptation pour comprendre comment marchait ce nouveau système.

La seconde difficulté fut en rapport avec notre classe Animateur. En effet, toute la partie animation est relativement complexe car elle fait le lien entre le monde continu de l'animation des balles et le monde discret de la zone de jeu.

Une autre difficulté a été celle de la compréhension des règles du jeu initial et du séquencement des actions dans la boucle de jeu. Le jeu original étant vieux avec très peu de ressources à jour, nous avons eu des visions très différentes sur les implémentations à réaliser. A force d'observations de sources vidéo du jeu et d'appropriation des règles nous avons finalement réussi à trouver un accord qui se rapproche le plus possible du jeu initial.

Enfin, nous avons dû résoudre le problème de la neuvième ligne. Comme énoncé précédemment, l'ajout de cette ligne a entraîné un gros changement dans notre jeu. Il a fallu revoir complètement certains comportements des balances, de l'animateur, du joueur et de certaines balles de pouvoir.

# Conclusion

Pour conclure, le projet a permis l'enrichissement de notre expérience dans des domaines techniques ainsi que dans le travail d'équipe. En effet, nous avons eu l'occasion d'apprendre à manipuler le moteur Unity et d'acquérir de nouvelles compétences telles que :

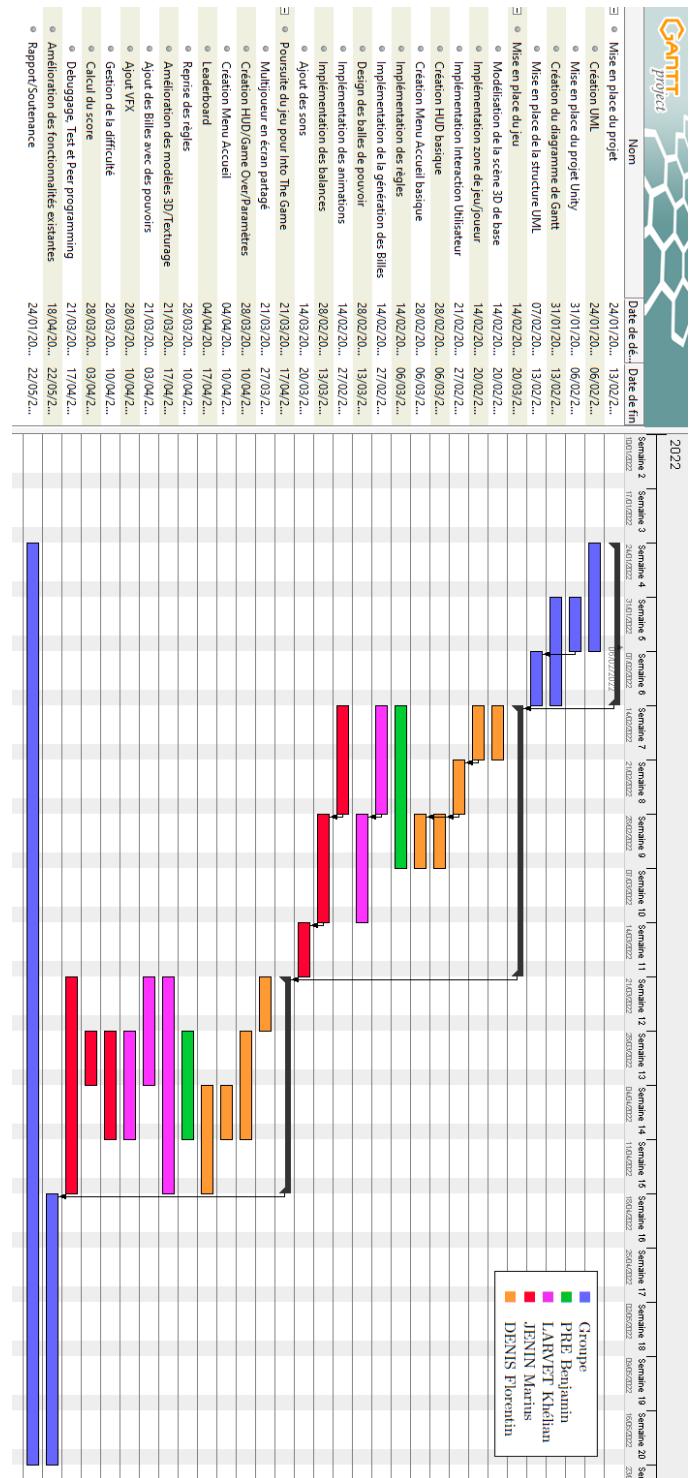
- L'utilisation de script dans le langage C#.
- La création d'effets spéciaux (avec Unity).
- La manipulation des contrôles avec la librairie `Input System` (avec Unity).
- La modélisation de shader dynamique (avec Unity).

Pour travailler en équipe, nous avons utilisé des méthodes agiles, avec la mise en place de réunions hebdomadaires par l'intermédiaire de Discord. De la même façon, notre système de versionnage de fichier, *Unity Collab* puis *Plastic SCM*, nous a aidé pour maintenir une trace de nos recherches et des mises à jour.

Plusieurs améliorations restent possibles sur ce projet afin de le rendre plus complet et attrayant. Par exemple, en mettant en place une intelligence artificielle capable d'affronter un humain, ou en implémentant le système multi-joueur interactif permettant la liaison des tableaux de jeu. Cette dernière amélioration permettrait aux joueurs de s'échanger des balles via la règle de balancement, ce qui pourra engendrer différents modes de multi-joueur tels que coopératif ou duel.

## Annexes

# Diagramme de Gantt



# Bibliographie

- [1] Wikipedia, Swing, 2021  
[https://en.wikipedia.org/wiki/Swing\\_\(video\\_game\)](https://en.wikipedia.org/wiki/Swing_(video_game))
- [2] Youtube, Toutes les règles de Swing, 2021  
<https://www.youtube.com/watch?v=7bVNiktUOsM>
- [3] Youtube, Exemple de partie, 2015  
[https://www.youtube.com/watch?v=\\_F56QHP5Y24](https://www.youtube.com/watch?v=_F56QHP5Y24)
- [4] acek Jankowski, Martin Hachet. A Survey of Interaction Techniques for Interactive 3D Environments. Eurographics 2013 - STAR, May 2013, Girona, Spain. hal-00789413  
[https://seafile.lirmm.fr/d/25e7381938ec400fb33c/files/?p=%2FArticles%2F4\\_A\\_survey\\_of\\_interaction\\_techniques\\_for\\_interactive\\_3D\\_environments.pdf](https://seafile.lirmm.fr/d/25e7381938ec400fb33c/files/?p=%2FArticles%2F4_A_survey_of_interaction_techniques_for_interactive_3D_environments.pdf)
- [5] VoxColliDe: Voxel collision detection for virtual environments, S.M. Lock, D.P.M Wills, Department of Compute Science, University of Hull, UK,  
[https://seafile.lirmm.fr/d/25e7381938ec400fb33c/files/?p=%2FArticles%2F21\\_VoxcolliDe\\_Voxel\\_collision\\_detection\\_for\\_virtual\\_environments.pdf](https://seafile.lirmm.fr/d/25e7381938ec400fb33c/files/?p=%2FArticles%2F21_VoxcolliDe_Voxel_collision_detection_for_virtual_environments.pdf)
- [6] Controlled Animation of Video Sprites, Arno Shödl, Irfan A. Essa, Georgia Institute of TechnologyGVU Center / College of ComputingAtlanta  
[https://seafile.lirmm.fr/d/25e7381938ec400fb33c/files/?p=%2FArticles%2F9\\_Controlled\\_animation\\_of\\_video\\_sprites.pdf](https://seafile.lirmm.fr/d/25e7381938ec400fb33c/files/?p=%2FArticles%2F9_Controlled_animation_of_video_sprites.pdf)
- [7] Toward Game Level Generation from Gameplay Videos, Matthew Guzdial, Mark O. Riedl  
School of Interactive Computing, Georgia Institute of Technology  
<https://arxiv.org/ftp/arxiv/papers/1602/1602.07721.pdf>