JEU SOGO

RAPPORT DE TRAVAIL TPI | SERRA IVAN | 01.04.2020

CANDIDAT À L'EXAMEN

Serra Ivan
ivan.serra@ceff.ch

SUPÉRIEUR PROFESSIONNEL

Stalder Joachim
joachim.stalder@ceff.ch
032 942 43 34 / 078 758 36 98

Expert responsable
Rebetez Benoît
062 837 46 70 / 079 827 89
benoit@rebetez.org
Expert accompagnant

Beuchat José
079 794 13 11
jose.beuchat@bluewin.ch

DIFFÉRENTS RENSEIGNEMENTS

Date de remise	1 avril 2020
Salle	BD77i, salle informatique
Classe	ICT3_PMA

VERSION SUCCINCTE DU RAPPORT

INTRODUCTION

On m'a demandé dans le cadre d'un « Travail Productif Individuel », de réaliser un projet d'un « Jeu Sogo ». Le but est de recréer le jeu qui est une sorte de puissance 4 en 3D sous l'environnement de Unity et de pouvoir jouer contre un autre joueur et ensuite contre une IA.

OBJECTIF DU PROJET

Le projet consiste à réaliser le jeu du Sogo avec une IA.

Le premier objectif est de créer une interface graphique 3D grâce à Unity du jeu afin de pouvoir visualiser tous les endroits du plateau de jeu. Le design m'a été laissé libre tant que l'on comprend. Le plateau de jeu est une grille 4x4x4. On doit pouvoir voir la bille se déplacer jusqu'en bas de la tige lorsqu'elle est ajoutée. On ne peut pas empiler plus de 4 billes sur une tige et bien évidemment chaque joueur doit jouer chacun son tour.

La deuxième implémentation demandée, c'est que l'on puisse jouer contre un autre joueur humain. On doit pouvoir déterminer le gagnant, donc voir si quelqu'un aligne 4 billes verticalement, horizontalement ou en diagonal.

Puis on rajoute une IA avec la possibilité de pouvoir jouer avec différents niveaux de difficultés et avec des coups aléatoires.

RÉALISATION

Pour la réalisation de l'interface, j'ai utilisé les différentes formes 3D d'Unity pour visualiser le choix qu'a été fait et pour pouvoir tourner autour du plateau. Ensuite, les fonctions pour l'ajout de la bille et pour que la tige n'ait pas plus que 4 billes, ont été implémentées.

Puis, toutes les fonctionnalités pour pouvoir jouer contre un joueur humain ont été ajoutées, c'est-à-dire, chacun joue à son tour avec la méthode qui permet de calculer si quelqu'un a gagné.

L'IA a été faite avec l'algorithme minimax et ensuite avec l'algorithme alpha-beta, sans oublier une méthode permettant de compter les points. Puis, j'ai utilisé la classe *Random*, qui est une classe gérée par Visual Studio qui m'a permis de faire un système de niveaux de difficulté et de coups aléatoires avec l'IA.

CONCLUSION

Le projet est fonctionnel dans sa totalité et le cahier des charges est rempli, même si que pendant le projet, j'ai eu pas mal de problèmes dans l'implémentation de l'IA. Des améliorations pourraient toujours être faites dans l'IA car c'est très dur d'avoir une IA « parfaite ». En revanche, mon code est dynamique et optimisé, pour montrer cela une version Beta a été créée pour que mon plateau puisse se redimensionner toujours en fonctionnant.

Table des matières

1 INTRODUCTION

1.1 SITUATION DE DÉPART

Cette application est réalisée dans le cadre d'un « Travail productif individuel ». Ce projet n'est pas la continuité d'un autre projet, il a été créé depuis le début jusqu'à la fin. Donc, il faut imaginer que le projet a une logique propre à la mienne.

1.2 OBJECTIFS DU PROJETS

Le projet consiste à créer le jeu de Sogo avec une IA. Cette IA doit être capable de jouer contre un joueur de manière la plus efficiente possible.

Les différents objectifs du cahier des charges sont les suivants :

- Réaliser une interface graphique
- Réaliser un plateau de jeu avec une grille 4x4x4
- Pouvoir jouer chacun à son tour
- Pouvoir jouer contre un autre joueur humain
- Déterminer le gagnant
- Pouvoir jouer contre une IA
- Permettre à l'utilisateur de choisir le niveau de difficulté
- Ajouter des coups aléatoires dans les coups de l'IA

1.3 HORAIRE

Le projet se déroule sur sept semaines, il commence le mercredi 19 février 2020 et se conclus le mercredi 1 avril 2020. 80 heures et 35 minutes sont à disposition pour l'élaboration du projet et ce temps est réparti entre les lundis et les mercredis. Chaque lundi, 4 heures et 35 minutes sont disponibles, tandis que pour chaque mercredi, 7 heures et 35 minutes sont à disposition.

1.4 PLANNING

Le planning a été fait pour que chaque case sur Excel corresponde à 15 minutes. Et les cases bleues sont les tâches prévues, les cases oranges sont les tâches effectives et les cases vertes sont les tâches permanentes. Un temps total est écrit pour définir combien de temps a été mis au total pour une tâche.

2 ASPECTS TECHNIQUES ET TECHNOLOGIQUES

2.1 UNITY

Unity a été utilisé dans ce projet, car il a été imposé. Il faut savoir que j'ai utilisé grandement Unity pendant mon temps libre, donc j'ai pas mal d'aisance avec ce programme. Quand on travaille avec Unity dans de la 3D on travaille essentiellement avec des vecteurs qui reviennent à des math assez poussés. La version d'Unity utilisée dans ce projet est 2019.2.5f1.

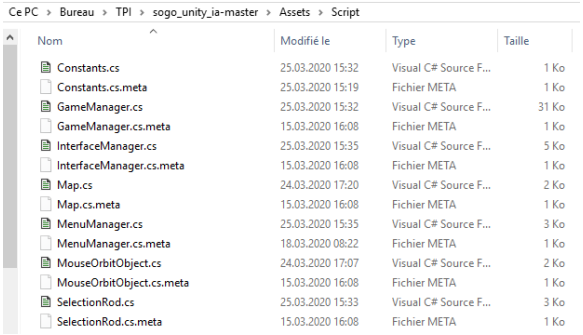
Logo de Unity

2.2 VISUAL STUDIO

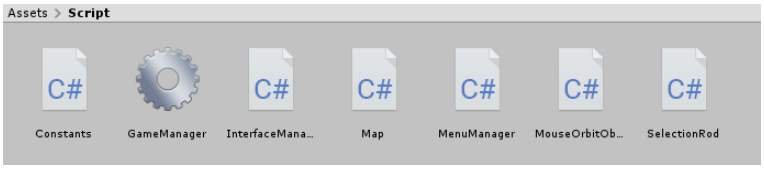
Visual Studio est utilisée car c'est l'IDE qui m'est imposé permet de coder sur Unity et de le lier pour pouvoir *Debug*. La version qui est utilisée est la 16.3.3. Le langage de programmation utilisé est du C#.

2.3 VISUALISATION DES FICHIERS

Pour voir le code effectué, les fichiers se situent à un endroit qui est *Assets→Script* :



Visualisations dans l'explorer de Windows



Visualisation dans Unity

Je l'indique car si on n'a pas l'habitude d'Unity, on ne sait pas où ça se trouve.

2.4 CAPACITÉ

J'ai fait auparavant une IA de morpion qui m'as permis d'avoir une base de comment faire l'IA, mais pour le Sogo, c'est totalement différent, l'algorithme pour compter les points est beaucoup plus complexe. C'est pourquoi, j'allais plus ou moins dans l'inconnu, mais pour contrer cela, j'ai fait en sorte de me donner un peu de temps pour prendre connaissance d'algorithmes semblables à ceux-là comme le puissance 4.

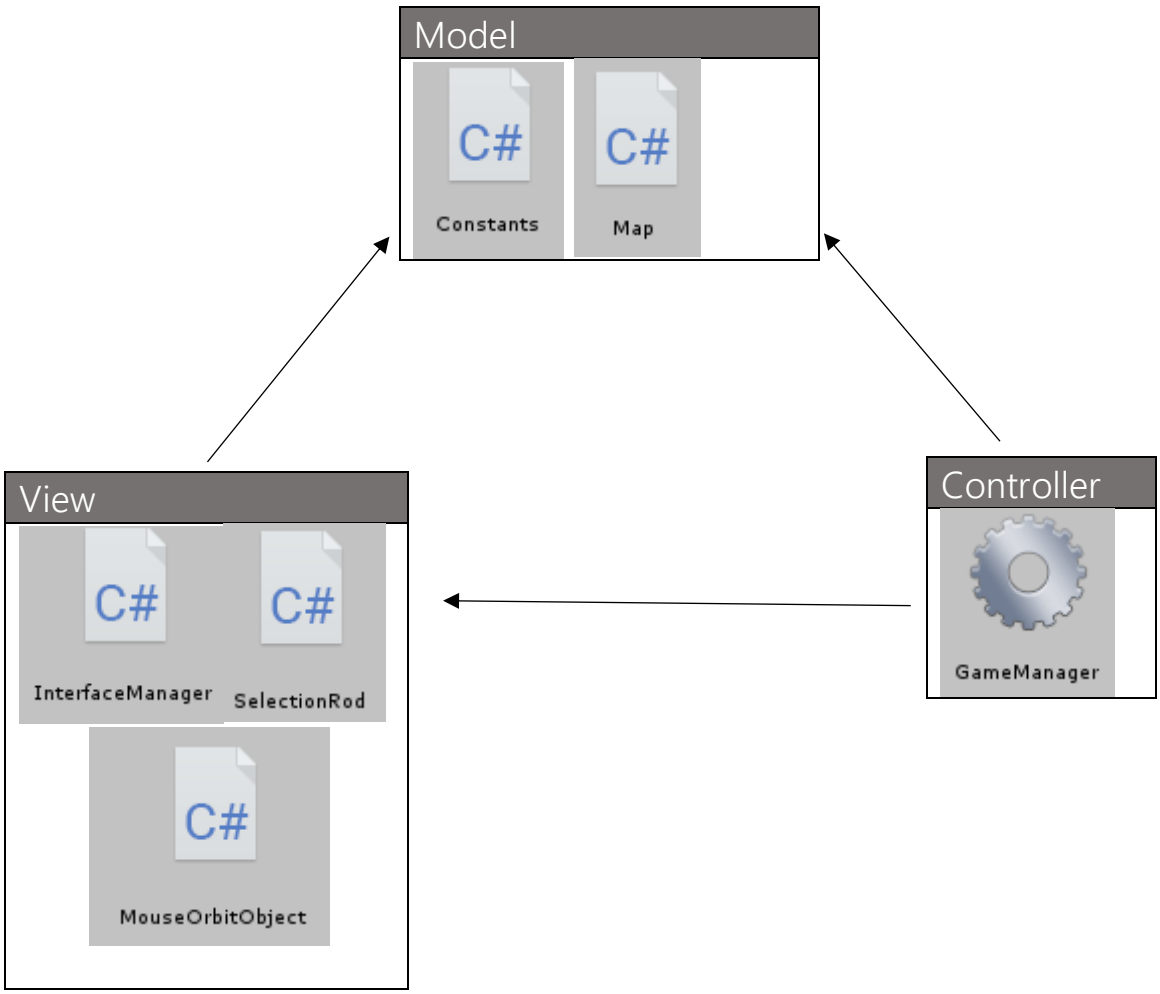
2.5 LANCEMENT JEU

Veuillez, pour avoir la meilleure condition possible, lancer le .exe en fenêtré **1024x768**. Pour poser des billes, il faut faire clique gauche en haut des tiges et pour bouger, maintenir clique droit puis bouger la souris.

3 RÉALISATION

3.1 ARCHITECTURE MVC

Une architecture MVC a été faite afin d'avoir quelque chose de propre et de bien fait. Le modèle ressemble à cela :



Architecture MVC de mon projet

Dans ce schéma, nous avons le *Model* qui est la *Data* de notre jeu, c'est pourquoi on voit qu'il y a les constantes et ma classe *Map* qui est le tableau dans lequel toutes mes valeurs sont stockées. Nous avons le *View* qui sera tout ce qui est en rapport avec le graphique qui aura besoin de prendre les « *data* » donc le *Model*. Pour finir, nous avons le *Controller* qu'aura besoin du *View* et du *Model* pour faire tous les calculs complexes et qui aura l'intelligence artificielle.

3.2 MENU

3.2.1 But

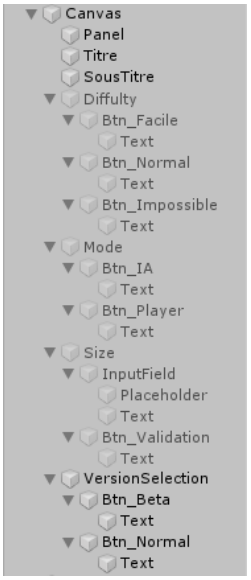
Le but est de pouvoir avoir une interface aidant l'utilisateur à mettre les options qu'il souhaite dans le jeu.

3.2.2 Réalisation

Pour le réaliser, ce que j'ai fait c'est d'afficher ou pas certains boutons selon ce que l'utilisateur choisit. Pour le faire, en tout premier, j'ai ajouté les boutons dans le *Canvas*[1] puis j'ai mis un fond blanc qui est le *Panel*[2]. J'ai ajouté un texte *Titre* et *SousTitre*. A droite, on peut voir comment la hiérarchie est faite sur Unity. Voici le code d'un des boutons, vu qu'ils se ressemblent tous, si on comprend celui-là, on comprend le reste :

```
/// <summary>Méthode permettant de jouer contre une l'IA.</summary>
0 références
public void IA()
{
    SousTitre.text = "Choissisez la difficulté de l'IA!"; 1
    Mode = Constants.IA; 2
    gmoMode.SetActive(false);
    gmoDifficulty.SetActive(true); 3
}
```

Extrait de code pour le boutons IA



Hiérarchie des Composant sur Unity

Un seul *Script* gère le tout et c'est le *MenuManager*. C'est là, où il y aura toutes les méthodes des boutons. En premier lieu, on modifie le *SousTitre*[1]. Puis on met dans une variable le mode avec la constante pour éviter les erreurs d'écriture[2], pour finir, on active les prochains boutons et on désactive les boutons actifs[3].
Ces valeurs sont stockées dans le *MenuManager* qui aura la fonction *DontDestroyOnload*, qui va faire que si je change la scène du jeu, ce script ne sera pas détruit et je pourrai conserver ces valeurs et les utiliser pour que les options choisies de mon utilisateur soient respectées.
J'ai aussi implémenté un bouton pause marchant sur le même principe d'afficher et désafficher les boutons, mais le code se situe dans le script *InterfaceMananger*. Il a les fonctionnalités de recommencer une partie ou de retourner au menu principal.

3.2.3 Test du code

J'ai voulu tester mon code pour que les valeurs qui lui sont envoyées soient les bonnes et que l'utilisateur ne puisse pas, par inadvertance, créer des erreurs dans mon application. J'ai placé des *Try catch* pour être sûr que mon jeu continue de tourner, même si j'obtiens une erreur pour des raisons diverses et variées, en mettant des valeurs par défaut dans ces variables.

[1] Un Canvas est une zone où toutes les UI doivent être posées
[2] Un Panel est une zone rectangulaire avec un *Background*

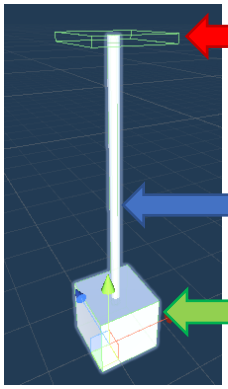
3.3 INTERFACE

3.3.1 Le but

Le but est d'avoir une interface qui sera le plateau de jeu avec lequel l'utilisateur pourra voir le jeu.

3.3.2 Réalisation

Au départ j'étais parti pour que mon plateau soit déjà dessiné sur Unity dans le sens manuellement, mais ensuite j'ai eu l'idée de l'initier grâce à une *Prefab*[3] que j'ai créé et qui ressemble à cela :



C'est une « zone » de clique en forme de cube aplati avec le graphique invisible servant à avoir la zone de clique avec un *Material*[4] vert-transparent et qui possède le script *SelectionRod*.

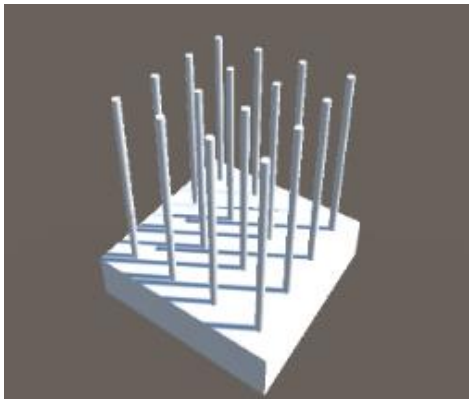
C'est une tige faite à partir d'un cylindre qui a subi un étirement et qui n'as pas de *Collider*, donc les objets lui passent à travers comme la bille.

C'est un cube de Unity permettant d'avoir le bas du plateau et qu'il a un *Collider* qui ne permet pas que la bille tombe.

Visualisation de la structure à l'aide de Unity

Maintenant pour créer notre plateau, on va, dès que la scène est chargée, lancer une méthode *Init* qui se situe dans le script *InterfaceManager* qui va initialiser tout comme il le faut. Concrètement, ce qu'elle fait c'est parcourir avec deux boucles toute la taille du plateau en initialisant chaque fois une de ces *Prefabs* qu'on a vu avant. Aussi, je transmets la position et d'autres données dans le script *SelectionRod* de la zone qu'on clique pour l'utiliser plus tard.

En les mettant les uns après les autres, on obtient ce résultat :



Visualisation du plateau

[3] Une *prefab* est une structure préfète que l'on peut initialiser et qui aura les mêmes propriétés.
[4] Un *material* dans Unity est quelque chose qui permet de mettre de la couleur.

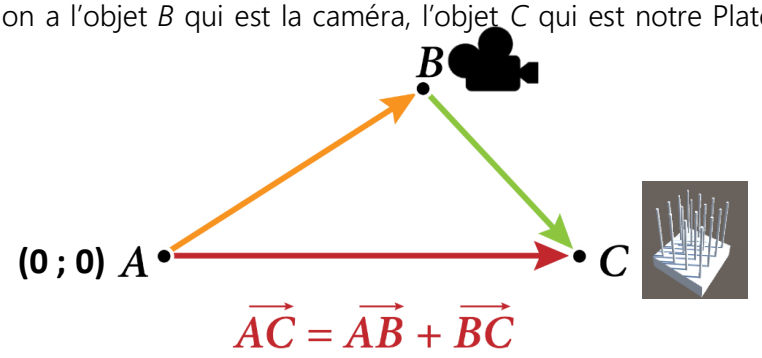
3.4 FONCTION POUR TOURNER AUTOUR

3.4.1 Le but

Le but est d'avoir un moyen, afin de pouvoir voir tout le plateau efficacement et simplement, qu'il soit assez intuitif pour que l'utilisateur puisse le manipuler aisément.

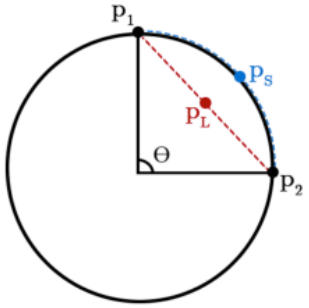
3.4.2 Réalisation

Pour la réalisation de cela, j'ai eu l'idée de faire que, quand on clique droit avec la souris on puisse tourner sur l'axe horizontal du plateau. Pour cela on devra faire une rotation de la caméra[5] pour qu'elle « orbite » autour du plateau. Mais tout d'abord on doit faire que la caméra regarde le plateau, pour cela on va utiliser le théorème de Chasles :
Pour l'expliquer, on a l'objet *B* qui est la caméra, l'objet *C* qui est notre Plateau et *A* qui est le



Scéma du théorème de Chasles

point d'origine (0 ; 0). Dans ce cas, ce qui nous intéresse c'est le Vecteur $\overrightarrow{BC}$ qu'on obtient grâce à ce théorème: $\overrightarrow{AC}$-$\overrightarrow{AB}$=$\overrightarrow{BC}$.
Ces cordonnées sont nécessaires pour que notre caméra regarde le plateau. Par contre, pour que je puisse « orbiter » autour de mon plateau, je vais utiliser la méthode préfète d'Unity *Slerp*, qui va me faire une interpolation sphérique polaire quand je lui donne la valeur de la caméra et sa position finale calculée en fonction de comment la souris bouge dans l'axe X. Avec ces données, il va me retourner l'orbite sphérique comme cela:



Scéma qui explique la méthode Slerp de Unity

Pour finir je vais l'activer que si mon utilisateur clique droit. Autre précision, tout le code se trouve dans le script *MouseOrbitObject*.
On obtient donc comme résultat final, un moyen de se déplacer autour du plateau, mais il faut bien imaginer qu'il y a plusieurs solutions et j'ai fait le choix de celle-ci, car c'était une des plus simples à reproduire pour gagner du temps et cela me permettrais de me concentrer plutôt sur l'IA.

[5] Dans Unity la caméra est ce que l'utilisateur va pouvoir voir quand le jeu se lance

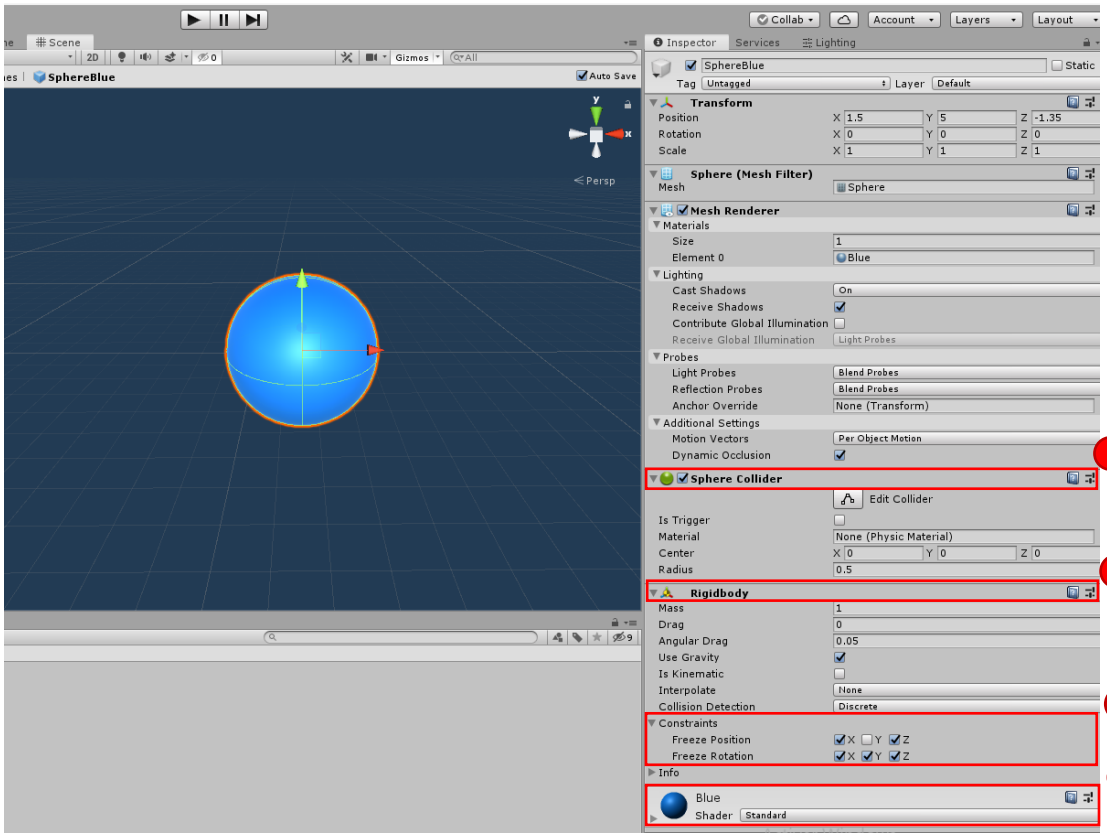
3.5 INSTANCIATION DES BILLES

3.5.1 Le but

Le but est de pouvoir initier une bille de couleur quand le joueur clique sur mes « zones » vertes que vous avez pu voir avant.

3.5.2 Réalisation

Comme vous avez pu le voir avant, j'ai mis un cube vert que j'appelle « zone » qui va m'aider à instancier ma bille. Pour cela, je vais afficher le graphique de la zone verte à l'utilisateur quand il survole avec la souris ce carré et ensuite, si la souris quitte cette zone, je vais cacher le graphique à nouveau. Ces méthodes se trouvent dans *SelectionRod*. Mais si l'utilisateur clique sur cette zone, alors on va faire apparaître une bille qui sera une *Prefab*.



Composants de la bille bleu

Cette bille aura les composants suivants un *RigidyBody* et un *Collider*, ce qui fera que la bille aura une gravité à laquelle j'ai contraint ses mouvements pour qu'elle puisse que descendre ou monter. Vu que j'ai indiqué que la tige ne recevait aucune collision de personne, alors la bille créée avec un *Material* de couleur bleu, ne pourra que tomber jusqu'à toucher le cube tout en bas.

Je répète le même processus, mais cette fois, pour la bille rouge.

Je vais mettre à chacune de mes zones le script *SelectionRod* où le but et la logique sont les suivants; c'est lui qui s'occupera d'afficher, ou pas, sa zone en fonction de la souris et si le joueur clique, il va demander à mon *GameManager* quelle bille doit-il poser, en fonction du tour à qui c'est de jouer. Et pour savoir à qui c'est de jouer, c'est juste une *Boolean* qui s'intervertit chaque fois qu'on pose une bille. Dans une variable sont stockées le nombre de fois qu'il y a eu une bille

qu'a été posée pour que si ce chiffre atteint 4, alors ma zone ne s'affichera plus, empêchant le joueur de poser plus que 4 billes. Pour finir, il va envoyer la position de la bille au *GameManager*.

3.6 FONCTIONNEMENT DE LA MAP

3.6.1 But

Le but va être de trouver un moyen pour pouvoir utiliser ces billes dans notre plateau pour diverses raisons.

3.6.2 Réalisation

Maintenant que nous avons vu tout ce qui était graphique, on va s'attarder sur le fonctionnement du jeu.
En tout premier, je représente mon plateau de manière à ce que je puisse l'utiliser plus tard à travers du code. Ce que j'ai fait est une classe que j'ai appelé *Map* qui contient un tableau à trois dimensions de *String* dans lequel chacune des dimensions représenteront un axe dans le jeu.

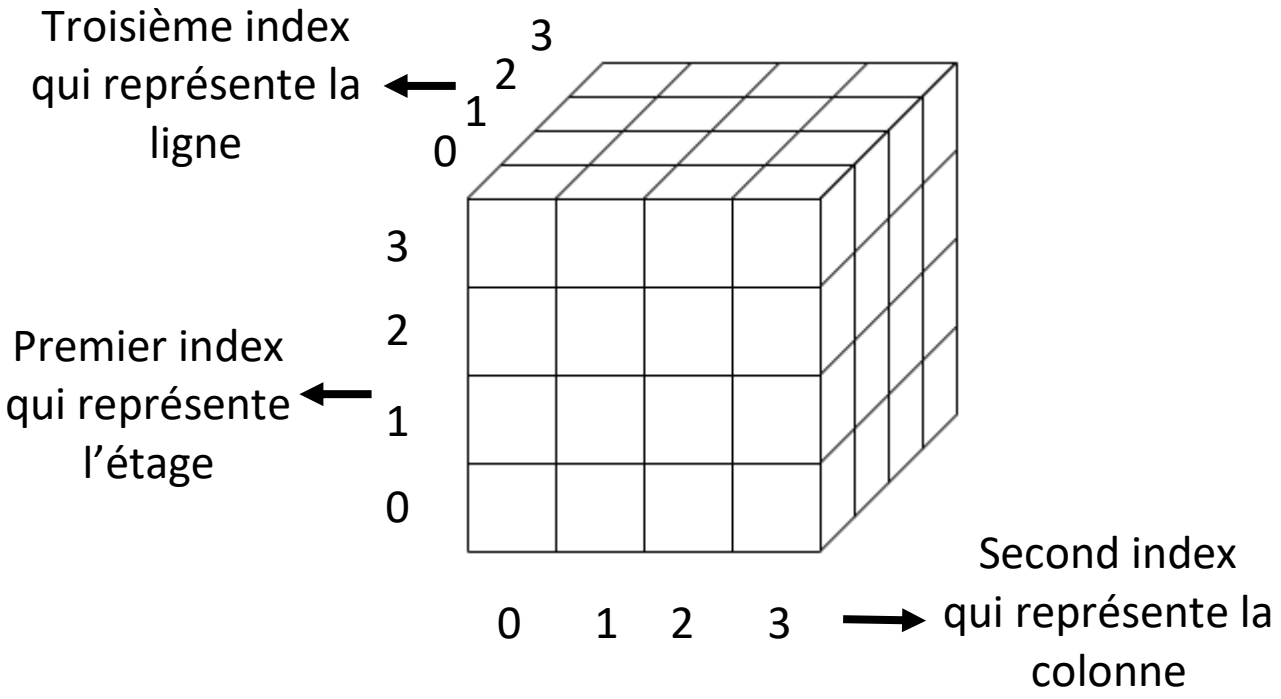


Schéma représentant les index dans mon tableau

Il faut savoir que mes valeurs *String* sont représentées comme ceci :

- *"Blue"* → c'est pour représenter une bille bleue
- *"Red"* → c'est pour représenter une bille rouge
- *"."* → c'est pour représenter qu'il n'y a aucune bille dessus

Quand mon jeu se lance, j'initialise avec des valeurs qui ne représentent aucune bille, et pour les remplir, je prends la donnée de ma zone qui me transmet la position de la bille qu'a été posée à l'endroit souhaité dans ma *Map*. C'est mon GameManager qui s'occupe de cela grâce à la méthode *SetBallInArray*.

3.7 CONDITION DE VICTOIRE

3.7.1 Le but

Le but va être de pouvoir avoir un moyen de déterminer si un joueur a gagné.

3.7.2 Idée 1

Ma première idée était de faire une similitude à un de mes anciens projets qu'il lui ressemblait, le morpion. Pour le faire, j'avais un tableau avec toutes les possibilités de gagner et je le comparais à mon résultat. Cependant, je remarque assez vite que dans une puissance 4 3D, il y a bien plus de solutions pour gagner que dans un morpion, c'est pourquoi j'abandonne cette idée-là.

3.7.3 Idée 2

Mon autre idée était de vérifier toutes les lignes, toutes les colonnes de tous les étages et diagonales. En résumé, c'était de vérifier tout sur tout grâce à des boucles pour voir s'il y avait 4 billes qui se suivent. Donc, le code devait parcourir à chaque clic tout mon tableau et devait vérifier quel joueur jouait. Quand je l'ai fait, tout marchait, cependant on constate que ce bout de code ne contrôle qu'une direction, par exemple; que les lignes, mais aussi il devait passer à travers 3 boucles par direction. C'est pourquoi je me suis dit qu'il devait y avoir mieux pour optimiser la chose, car sinon j'allais me retrouver avec un code beaucoup trop gros, et vérifier tout prendrait beaucoup de temps pour la machine à calculer.

3.7.4 Idée 3

Pour solutionner le problème de l'idée d'avant, j'ai fait en sorte de calculer que la partie qui m'intéresse, ce qui veut dire, que je calcule au moment où l'on pose la bille, et que la ligne, la colonne ou l'étage soient concernés. Si on schématise cela, on dit que le joueur bleu pose une bille, ma méthode va donc calculer tout ce qui se trouve sur la ligne de la bille posée et il va vérifier combien de billes de la même couleur se trouvent devant lui et combien s'en trouvent derrière, tout en ne pas dépassant mon tableau. Donc, si on pose la bille au début de la ligne ou à la fin de la ligne, ma méthode s'adaptera pour calculer le nombre de billes sur la ligne et me retourner s'il y a un résultat gagnant de 4. Puis, on fait de même dans les autres directions comme montré sur cette vue 2D simplifiée (normalement dans le jeu c'est en 3D donc cela part dans toutes les directions sans exception, mais pour la lisibilité je l'ai montré en 2D) :

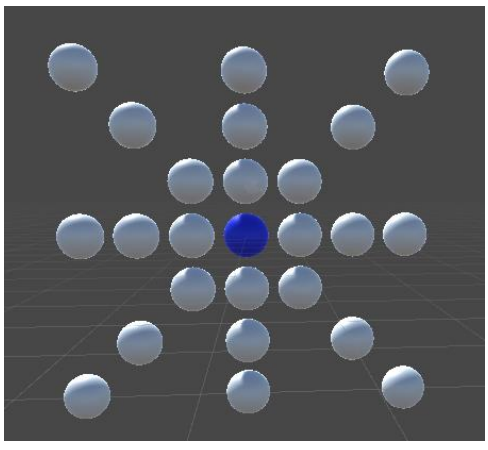


Schéma représentant la vérification en vue 2D

3.7.5 Réalisation

Pour réaliser cela, au départ ma méthode de vérification parcourrait la ligne de la bille posée, tout d'abord il regardait les billes posées devant, puis celles posées derrière, et si on détectait une bille adverse, on pouvait quitter la boucle en renvoyant une *Boolean false* mais si on en détectait 4 qui se suivaient on pouvait quitter la boucle en renvoyant *Boolean true*. Donc, on avait une méthode qui calculait les billes sur la ligne de la bille posée. Puis pour les diagonales, je ne savais pas trop comment m'approcher, j'ai réfléchi pendant un moment puis j'ai *copié coller* la méthode des lignes et changé quelques paramètres, etc. Avec cela, j'ai réussi à calculer une diagonale, mais, du coup, j'ai compris qu'il me fallait une méthode pour chaque direction et, si je faisais cela, je me retrouvais avec au moins 1000 lignes de codes pour faire la vérification, et moi, qu'avais comme but d'optimiser, ne m'arrangeait pas trop. C'est pourquoi, j'ai remarqué les similitudes de mon code et j'en ai conclu que c'est la manière dont je parcours qui est différente à chaque fois. Donc, j'ai ajouté un paramètre à ma méthode qui est la manière de parcourir avec ma boucle, et voilà comment ça marche.

Quand je parcours la boucle j'ajoute +1 à chaque fois où il y a besoin, par exemple si je dois parcourir ma ligne j'ajoute +1 à ma ligne, mais pas aux autres, et si je parcours ma colonne, j'ajoute +1 qu'à la colonne, etc. pour avoir un quelque chose du genre :

```
for (int forward = 0; forward < 4; forward++)
{
    if (Grid.tabMap[floor, column, row + forward] == Player)
    {
        streak++;
    }
}
```

Bout de code pour montrer comment parcourir la ligne

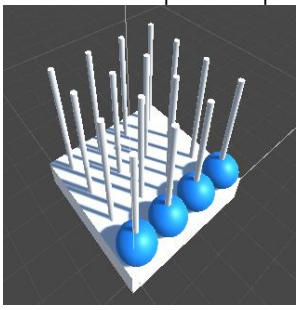
→ Parcours en fonction de la ligne, je regarde si la bille est égale à celle du joueur et j'ajoute 1 à ma série de bille (*streak*)

```
for (int forward = 0; forward < 4; forward++)
{
    if (Grid.tabMap[floor, column+forward, row] == Player)
    {
        streak++;
    }
}
```

Bout de code pour montrer comment parcourir la colonne

→ Parcours en fonction de la colonne, je regarde si la bille est égale à celle du joueur et j'ajoute 1 à ma série de bille (*streak*)

Donc, j'ai eu l'idée d'ajouter un multiplicateur à mon *Forward* par 1, par 0 ou par -1. Ça veut dire que si c'est multiplié par 1, alors les chiffres restent le même, donc il fait son +1, d'autre part, si c'est 0, alors mon *Forward* multiplie par 0, ce qui donne 0, donc mon *Forward* n'ajoute pas de chiffre et, finalement, si c'est –1, il avance à l'envers. Si nous prenons par exemple une ligne, et je note les cordonnées des billes qui sont posées pour la valider, les résultats sont les suivants :

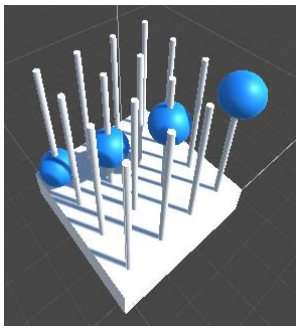


Visualisationd de la ligne gagnante

Etage	Colonne	Ligne
0	0	0
0	0	1
0	0	2
0	0	3
0	**0**	**+1**

Tableau m'aidant pour le multiplicateur

Avec ça, on remarque que, seule la ligne augmente de 1 et que les autres n'augmentent pas, donc mon multiplicateur est de : 0 ; 0 ; 1.
Mais maintenant prenons une diagonale qui sera plus complexe :



Visualisation de la diagonale gagnante

Etage	Colonne	Ligne
0	3	0
1	2	1
2	1	2
3	0	3
+1	-1	+1

Tableau m'aidant pour le multiplicateur

Avec cette méthode, on remarque que mon étage augmente de 1 mais ma colonne redescend de 1 et ma ligne augmente de 1, donc pour cette diagonale, le multiplicateur sera de : 1 ; -1 ; 1.

Je place tous mes multiplicateurs concernant les lignes, colonnes, diagonales etc. dans un tableau qui s'appelle *ArrayMultiplicatorForward* se situant dans le script *GameManager*, que je parcours par derrière pour appeler la même méthode avec ces différents multiplicateurs à l'aide de la méthode *CheckGameWin* se situant aussi dans *GameManager*, il aura comme effet de parcourir tout ce dont j'ai besoin. Dans la méthode *CheckDirection* se situant toujours dans *GameManager*, je vais mettre les valeurs dans un dictionnaire (*IDictionnarry*) de *String* et I*nt*, qui permettra de mieux comprendre le code et, surtout, d'inverser ses valeurs pour réaliser le parcours dans l'autre sens, car on les parcourrait à l'avant et maintenant, il faut voir ce qu'il se trouve derrière, pour cela je refais de nouveau une boucle pour le parcourir comme mentionné, mais avec les nouveaux multiplicateurs inversés.

Et du coup, mon code ressemblera à cela pour vérifier si la bille est égale à celle du joueur :

```
if (Grid.tabMap[
    floor + (forward * listDirectionMultiplicator["floor"]),
    column + (forward * listDirectionMultiplicator["column"]),
    row + (forward * listDirectionMultiplicator["row"])
    ]
    == Player)
```

Bout de code de la méthode CheckDirection

Avec cette méthode on obtient un moyen, de façon assez optimisé et dynamique, mais complexe, pour vérifier si le joueur gagne.

3.7.6 Test du code

J'ai testé cela en essayant toutes les façons de gagner, j'ai bien vérifié que si on faisait cette diagonale ou l'autre que cela me valide que le joueur a gagné.
C'était long mais je n'ai pas vraiment trouvé une solution plus rapide pour pouvoir vérifier que tout marche bien.

3.8 ALGORYTHME MINIMAX

3.8.1 But

Le but de cet algorithme est que l'IA fasse le meilleur coup possible dans tous les cas.

3.8.2 Connaissance

En tout premier j'ai essayé de prendre connaissance d'algorithme minimax de puissance 4 pour voir comment il fonctionnait, pour voir la différence qu'il y avait du morpion que j'avais fait.

3.8.3 Concept

Je vais en tout premier, expliquer comment l'algorithme fonctionne. L'IA va essayer de prévoir à l'avance un certain nombre de coups pour déterminer quel est le meilleur choix possible. Pour faire cela, on va d'abord donner un score à ses choix qui sont tout simplement des nombres entiers, et on va partir du principe que les scores positifs seront pour l'IA, qui jouera avec les billes de couleur rouge, et les scores négatifs seront pour le joueur avec les billes de couleur bleu.

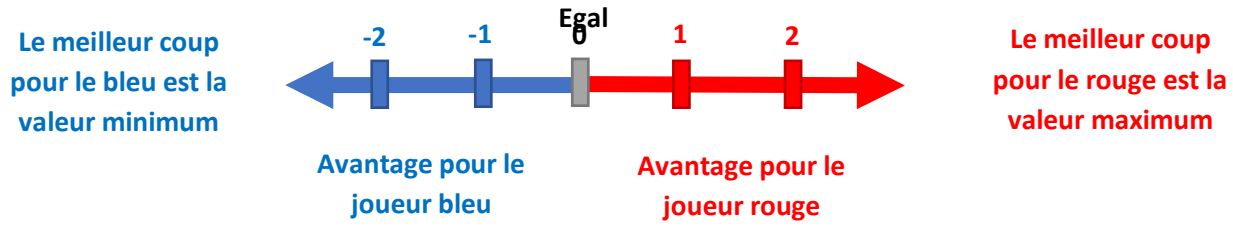
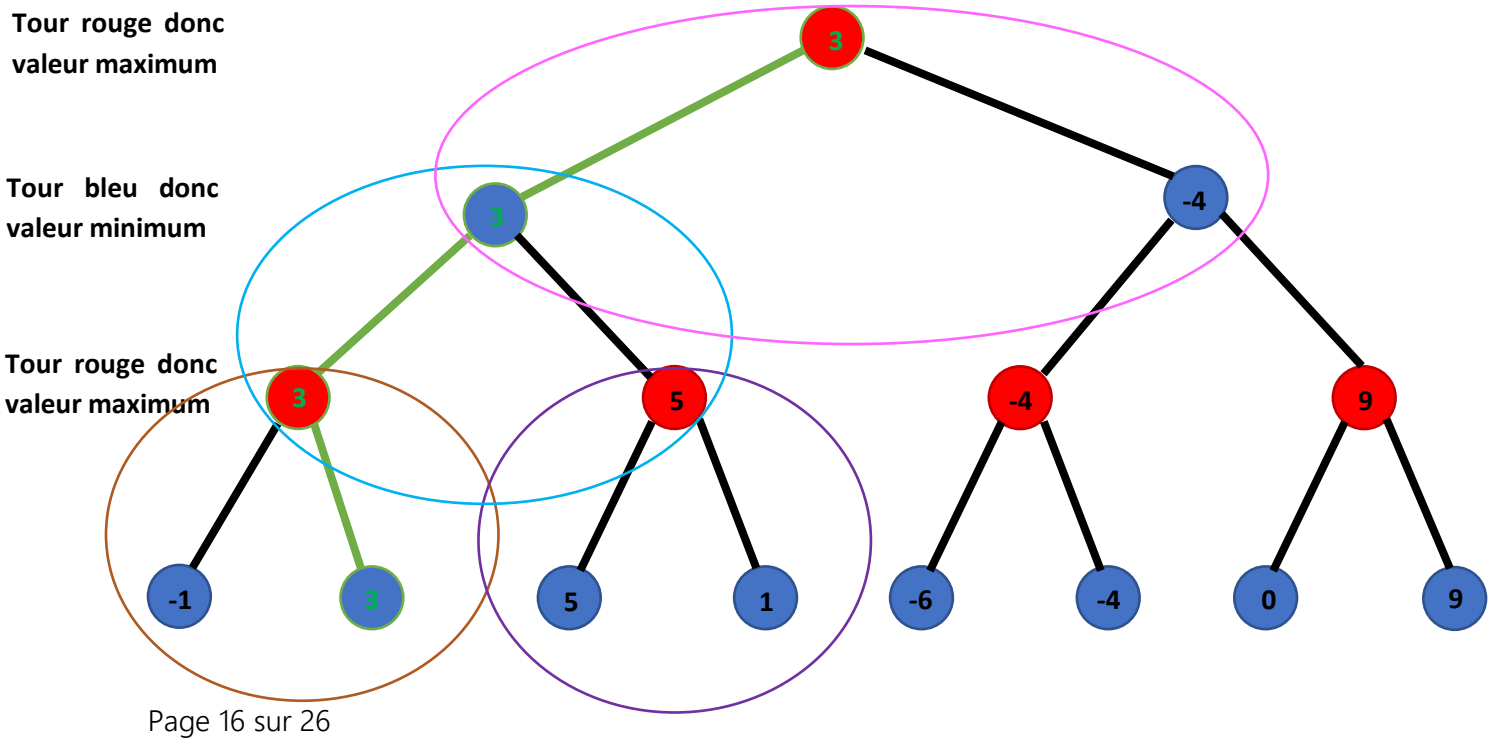


Schéma montrant comment le score fonctionne

Grâce à ce schéma, on comprend donc que pour trouver le meilleur coup du rouge, il faudra trouver la valeur maximum des scores, et pour le joueur bleu il faudra la valeur minimum des scores. Maintenant qu'on a compris cela, nous allons refaire un schéma plus complexe dans lequel on va indiquer le score de chaque coup et on va voir comment l'algorithme trouve la meilleure solution.

Schéma montrant l'algorithme minimax.



Si nous appliquons ce qu'on a dit avant, alors on commence par le bas, nous avons obtenu un score de -1 et 3, le rouge demande la valeur la plus grande donc c'est 3 qu'est pris. On peut donc écrire 3 dans la pastille rouge. A droite nous avons en bas 5 et 1, c'est le rouge donc la valeur maximum est 5. On écrit 5 sur la pastille rouge. Maintenant c'est le tour bleu qu'est demandé au-dessus, donc ça implique que la valeur qui sera prise sera la plus petite. La valeur minimum entre 3 et 5 est 3. C'est donc 3 qu'est à nouveau retenu. Puis, on fait la même chose pour les autres branches du schéma et on arrive avec le coup qui devrait être joué du coté rouge qui doit être le maximum donc entre 3 et -4, donc c'est 3. L'IA va déterminer à travers toutes ces branches que le meilleur coup serait 3.

3.8.4 Réalisation

Une chose importante à tenir en compte, c'est que je n'ai pas inventé cet algorithme je l'ai pris sur internet (https://en.wikipedia.org/wiki/Minimax). Cependant, je vais expliquer les quelques modifications que j'ai apporté à cet algorithme qui se situe dans la méthode *Minimax* qui est dans le script *GameManager*.

J'ai arrêté mon algorithme à la couche 3 ce qui veut dire qu'elle calcule que 3 coups à l'avance, si je fais plus c'est trop lent.
Une autre modification, c'est que, à ma toute première couche, chaque fois je regarde si un de mes joueurs gagne car, si c'est le cas, il n'y a pas besoin d'aller plus loin, on doit urgemment bloquer ou gagner. Ça me permet juste de gagner du temps dans la réflexion de l'IA et d'être sûr qu'il me rapporte la meilleure réponse.
Pour optimiser encore plus la chose, je vais tester que les billes soient positionnées à l'étage le plus bas (le carrée vert dans le schéma), car c'est inutile et impossible de mettre une bille à l'étage 3 si aucune bille est en-dessous. Ce qui donne cela :

Endroit où l'on peut poser des billes

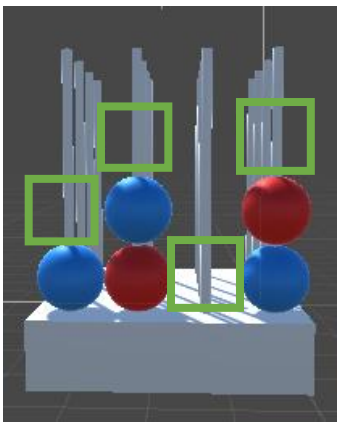


Schéma représentant où les bille peuvent être posées

C'est une modification que j'ai apporté à l'algorithme minimax pour le rendre plus optimisé et efficace, mais il faut toujours savoir que c'est un choix que j'ai fait car je trouvais l'IA lente.

3.8.5 Test du code

Pour vérifier le code, cela a été plus complexe car il fallait voir au *Debug* tout le code et regarder pourquoi il avait ses valeurs. En plus de cela, j'ai *Print* certaines données pour savoir ce qu'elles m'affichaient.

3.9 COMPTAGE DES POINTS

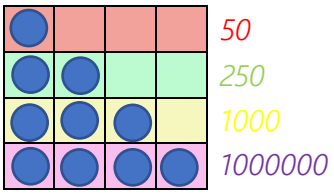
3.9.1 But

Le but est de pouvoir compter les points de chaque joueur afin que l'IA puisse choisir le meilleur coup. Il faut savoir que chacun créera un comptage de points différent selon ses points de vue, c'est pourquoi j'expliquerai mes choix.

3.9.2 Concept

Le concept du comptage il m'a été expliqué par Mr. Mathez pour un morpion car j'étais sensé l'avoir vu en classe. Ensuite, j'ai fait en sorte de l'adapter pour qu'il marche avec mon jeu. Le principe est très simple, il faut donner les points totaux que vaut ce plateau, pour les points, c'est moi qui ai choisi le nombre, mais on peut choisir ce que l'on veut, voici la façon qu'a été faite[6] :

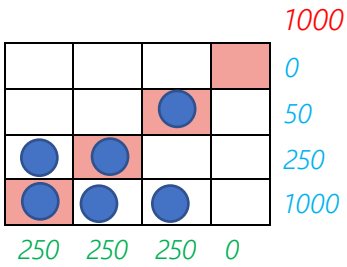
- Si une bille est déjà sur la ligne, on ajoute 50 points, s'il y en a 2, alors on donne plus, dans mon cas 250 et s'il y en a trois, encore plus : 1000. Mais s'il y en a 4, alors on doit lui attribuer un chiffre inimaginable car ça veut dire qu'il a gagné et que rien n'est meilleur. Il faut prendre en compte que c'est la même chose pour les colonnes et les diagonales.



Vision 2D du comptage de points par ligne

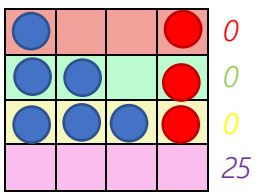
- On a les points d'une ligne, mais pour compter les points de chaque partie, il faut ajouter aussi ceux de chaque colonne et de chaque diagonale, donc ce qu'on fait c'est le total de tout pour obtenir notre score.

Vision 2D du comptage de points de toutes les directions



Total = 1000+0+50+250+1000+250+250+250+0 =3050 Points

- S'il y a déjà une bille d'un joueur adverse, la ligne vaut 0, car ça ne sert plus à rien de vouloir jouer là, vu qu'il n'y a aucun moyen pour gagner. Et si la ligne est vide, ça vaut quelque point, dans mon cas 25, car il y a de l'espoir de pouvoir faire une série.



Vision 2D du comptage de points avec adversaire et vide

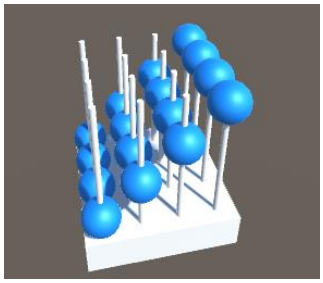
[6] Les schémas ont une vue 2D car c'est plus simple d'expliquer, mais il faut l'imaginer que ça marche pour la 3D

- Autre chose que j'ai ajouté c'est que les points se cumulent dans le sens où 2 billes qui se suivent faisant 250 points, elles vont aussi prendre les points d'une bille seule, donc 50. Le total serait finalement 250+50=300. Cela permet d'éviter que trop de petits scores de 50 arrivent à être meilleurs qu'un score de 250.

Ces points sont attribués au joueur rouge et au joueur bleu, le résultat sera la différence entre le score rouge et le score bleu, si l'IA est rouge.

3.9.3 Réalisation

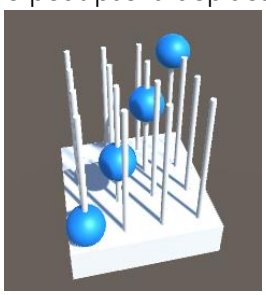
En tout premier ce que je vais faire c'est calculer le résultat des deux joueurs. Pour cela je vais faire appel à une méthode que j'ai créée qui s'appelle *CheckScore* se situant dans le script *GameManager*. Cette méthode ressemble beaucoup à celle pour calculer si quelqu'un a gagné ou pas. Cette fois à la place de calculer que la ligne, diagonale, colonne où la bille a été posée, on doit calculer toutes les billes qui se suivent sur toutes les lignes, diagonales, colonnes. Car maintenant on va donner les points de la partie, donc il faudra tout parcourir dans mon plateau et vérifier les règles qu'on a définie avant.
Si on prend par exemple une diagonale, on va calculer toutes les diagonales qui peuvent s'afficher partout dans le plateau.



Visualisation de toutes les mêmes diagonales qui font gagner

Pour cela, j'ai repris le code que j'avais fait pour la condition de victoire et j'ai créé la méthode *CheckScoreDirectionAll* se situant dans le script *GameManager*. J'ai ajouté des boucles afin de pouvoir parcourir tout et, à la place d'ajouter seulement une série à mon *Streak*, j'ajoute également des points. J'applique de nouveau des multiplicateurs pour faire que ça soit une méthode qui vérifie tout.
Si on reprend l'exemple d'avant on aura donc ma boucle d'avant qui vérifie la première diagonale puis ensuite j'ajoute une boucle qui ajoute 1 à ma ligne et, du coup, ma diagonale décale, et cela jusqu'à atteindre la fin de mon plateau.

C'est différent pour chaque vérification, par exemple pour une diagonale traversant le plateau, on aura que d'une boucle car on ne peut pas la déplacer sur le côté ou vers le haut :



Visualisation d'une autre diagonale

Et d'autres vérifications comme les lignes qui nécessiteront d'avoir 3 boucles, une pour bouger de côté et une autre pour faire l'étage d'au-dessus, mais aussi, par exemple pour la vérification des billes qui se cumulent en étage, les boucles doivent bouger en colonnes et lignes, pas en étages.

Donc je conclue au minimum on aurait besoin de 3 boucles avec mes multiplicateurs que je regarderai de quelle façon bouger. Pour optimiser, si mes multiplicateurs qui bougent de côté valent 0, alors je sors de la boucle qui bouge de côté, ce qui aura comme effet de ne pas perdre de temps à parcourir une boucle inutile.

Finalement on se retrouve avec ces 3 boucles :

```
for (int moveUp = 0; moveUp < lenght; moveUp++)
{
    for (int moveSlide = 0; moveSlide < lenght; moveSlide++)
    {
        //série de bille cumulé
        int streak = 0;
        //score par ligne
        int ScoreStreak = 0;
        //cvariable pour compter le nombre de cases vide
        int Vide = 0;
        //boucle parcourant une direction d'un côté
        for (int forward = 0; forward < lenght; forward++)
        {
```

Bout de code de CheckScoreDirectionAll pour montrer les boucles

Puis dans le *If* on se retrouve avec cela :

```
if (InputGrid.tabMap[
    StartIntFloor + (forward * listDirectionMultiplicator["floor"]) + moveSlide * listDirectionMultiplicator["MoveSlideFloor"] + moveUp * listDirectionMultiplicator["MoveUpFloor"],
    StartIntColumn + (forward * listDirectionMultiplicator["column"]) + moveSlide * listDirectionMultiplicator["MoveSlideColumn"] + moveUp * listDirectionMultiplicator["MoveUpColumn"],
    StartIntRow + (forward * listDirectionMultiplicator["row"]) + moveSlide * listDirectionMultiplicator["MoveSlideRow"] + moveUp * listDirectionMultiplicator["MoveUpRow"]
   ]
   == (Player))
{
```

Bout de code de CheckScoreDirectionAll pour montrer le If

Petit détail que j'ai ajouté, c'est que la profondeur affecte le score dans le sens que plus on est profond, moins c'est bien, car le but est, quand même, que l'IA fasse le meilleur coup le plus vite possible avant que l'adversaire réagisse.

3.10 ALGORITHME ALPHA-BETA

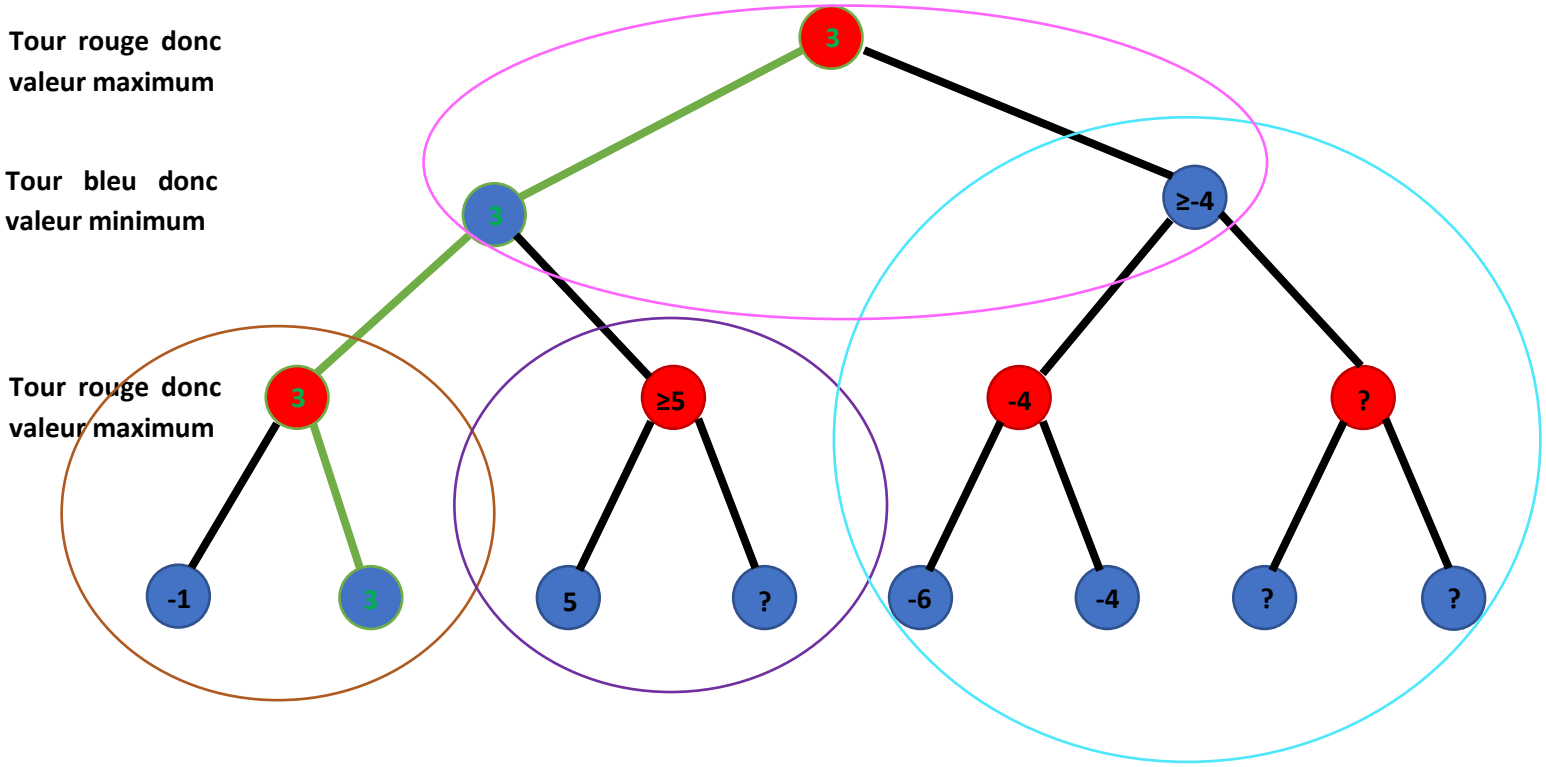
3.10.1 But

Le but avec cet algorithme est d'optimiser la réflexion de l'IA car il est très lent pour calculer un coup.

3.10.2 Concept

Du coup, pour optimiser cela, on va essayer de supprimer les calculs inutiles du minimax. Pour faire cela, je vais montrer comment ça se passe grâce à un schéma.

Schéma de l'algorithme alpha-beta



Comme vous avez pu le remarquer, certaines valeurs sont laissées inconnues « *?* ». Cela veut dire qu'on ne les a pas calculés, car c'était inutile. La raison est logique. On a calculé les valeurs à gauche et on a obtenu 3. Maintenant on calcule la branche de droite on obtient **≥5.** La raison est que pour faire un coup rouge, on est d'accord qu'on prendra la valeur la plus grande. Le chiffre inconnu, pour pouvoir être bon, doit être plus grand que l'autre score, donc 5. Donc le coup rouge est au minimum de 5. Maintenant le coup d'après est bleu, donc le score doit être le plus petit. On a 3 et de l'autre côté on a **≥**5, donc par logique 3 sera toujours le plus petit, donc celui qui sera choisi, et vous avez remarqué qu'a aucun moment on a eu besoin de savoir combien valait le chiffre inconnu.

Si on refait la même chose à droite. On fait les calculs et on obtient **≥-4, car comme on l'a dit, le score bleu doit être le plus petit possible, donc dans notre cas, plus petit ou égal à -4.** Le prochain score est pour le rouge, donc le maximum. On a 3 et **≥-4**, donc le plus grand, quoi qu'il arrive, reste 3, donc on n'a pas eu besoin de calculer les 3 autres inconnues « *?* ».

Cette méthode ne s'applique pas à chaque fois, cela dépend de comment sont placés les scores. Mais, il s'avère possible qu'il doit, quand même, tout calculer.

3.10.3 Réalisation

Pour la réalisation de cela, j'ai appliqué un code que j'ai trouvé sur internet (https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning). Je n'ai pas fait de modification, il marchait tel quel. L'unique chose que j'ai ajouté, c'est un moyen de *Break* dans les boucles *For* . Il se situe dans le la méthode Minimax qui est dans le script GameManager.

3.10.4 Test du code

J'ai testé grâce au *Debug* pour vérifier que tout marche bien. J'ai aussi remarqué que mon IA prenait considérablement moins de temps pour la réflexion.

3.11 COUP ALÉATOIRE

3.11.1 But

Le but est de ne pas prévoir l'IA, car elle fait souvent les mêmes coups donc en ajoutant de l'aléatoire tout en restant autant forte.

3.11.2 Réalisation

Pour faire cela c'était simple, j'ai mis les meilleurs coups qui avaient les mêmes scores de l'IA dans une liste et je n'ai fait qu'un *Random* dans cette liste, pour après définir le choix que l'AI ferait.
Ceci a été fait pour le *Max*, mais c'est la même chose pour le *Min* :

```
List<int> RandomScore = new List<int>();
for (int i = 0; i < scores.Count; i++)
{
    if (scores[i] == scores.Max())
    {
        RandomScore.Add(i);
    }
}
int MaxScoreIndex = RandomScore[random.Next(RandomScore.Count)];
choice = moves[MaxScoreIndex];
```

Code permettant l'aléatoire des coups pour l'IA

3.11.3 Test du code

Pour tester le code, j'ai fait quelques parties contre l'IA et j'ai surveillé, quand je pose la même bille à la même place, qu'elle fasse quelque chose de différent.

3.12 NIVEAU DE L'IA

L'objectif est de pouvoir sélectionner un niveau de difficulté à l'IA.

3.12.1 Idée 1

Une idée est de faire par rapport au nombre de coups à l'avance que l'IA peut prévoir. Mais j'ai trouvé cela compliqué à appliquer, dans mon programme vu que mon IA ne calcule pas plus loin que 3 coups à l'avance, donc j'ai préféré ne pas le faire avec cela.

3.12.2 Idée 2

La deuxième idée est d'ajouter de l'aléatoire dans le score, comme ça, elle a des risques de faire des erreurs. J'ai choisi cela, car c'était simple d'appliquer et on peut régler notre niveau comme on le souhaite.

3.12.3 Réalisation

Pour réaliser cela, j'ai pris donc la donnée de l'utilisateur, quand il choisit le niveau et, en fonction du niveau, je soustrais au score un nombre aléatoire de score, ce qui donne ceci :

```
//selon la difficulté on met de l'aléatoire dans les poin
if (difficulty == Constants.EASY)
{
    ScoreAlly -= random.Next(10000000);
}
else if (difficulty == Constants.NORMAL)
{
    ScoreAlly -= random.Next(200000);

}
```

Code permettant de faire le niveau d'avoir un niveau de difficulté

3.12.4 Test du code

J'ai vérifié que si je mets en mode facile quelques fois, l'IA ne me bloque pas.

3.13 DYNAMISME

3.13.1 But

Le but est de prouver que mon code est dynamique.

3.13.2 Réalisation

Vu que ce n'était pas demandé et qu'il y avait quelques problèmes, Mr. Stalder m'a plutôt proposé de l'insérer dans une Version Beta. J'ai mis un bouton pour la version beta. On obtient donc l'option de choisir la grandeur du plateau, donc si ont choisi 3, on a un 3x3x3 et on gagne à 3 billes cumulé. Pour faire cela je n'ai presque rien eu besoin de changer mon code, vu que je l'avais prévu pour cela, j'ai ajouté cette fonction pour montrer le dynamisme de mon code.

3.14 PROBLEMES RENCONTRÉES

3.14.1 Introduction

Pendant la réalisation de ce projet j'ai rencontré seulement un gros problème. Il était lié au *minimax* et le calcul des scores. J'ai remarqué que l'IA faisait n'importe quoi, au départ elle essayait de bloquer et elle refusait de gagner. En inversant les joueurs dans le code, mon IA maintenant faisait l'inverse, elle préférait gagner, mais maintenant, elle ignorait les coups de l'adversaire, ce qui était logique, mais je me suis dit que, peut-être, j'avais inversé les joueurs.

3.14.2 Solution

Enfaite, quand je vérifiais si on était à la couche 3 pour retourner le score, je retournais le score du joueur contraire. Car je vérifiais cela au début de mon minimax, quand je retourne au tour d'avant dans le récursif. Pour solutionner cela, j'ai donc fait vérifier si l'autre joueur gagnait, comme ça, j'obtenais celui du bon joueur.
J'ai fait un petit schéma pour mieux comprendre la solution

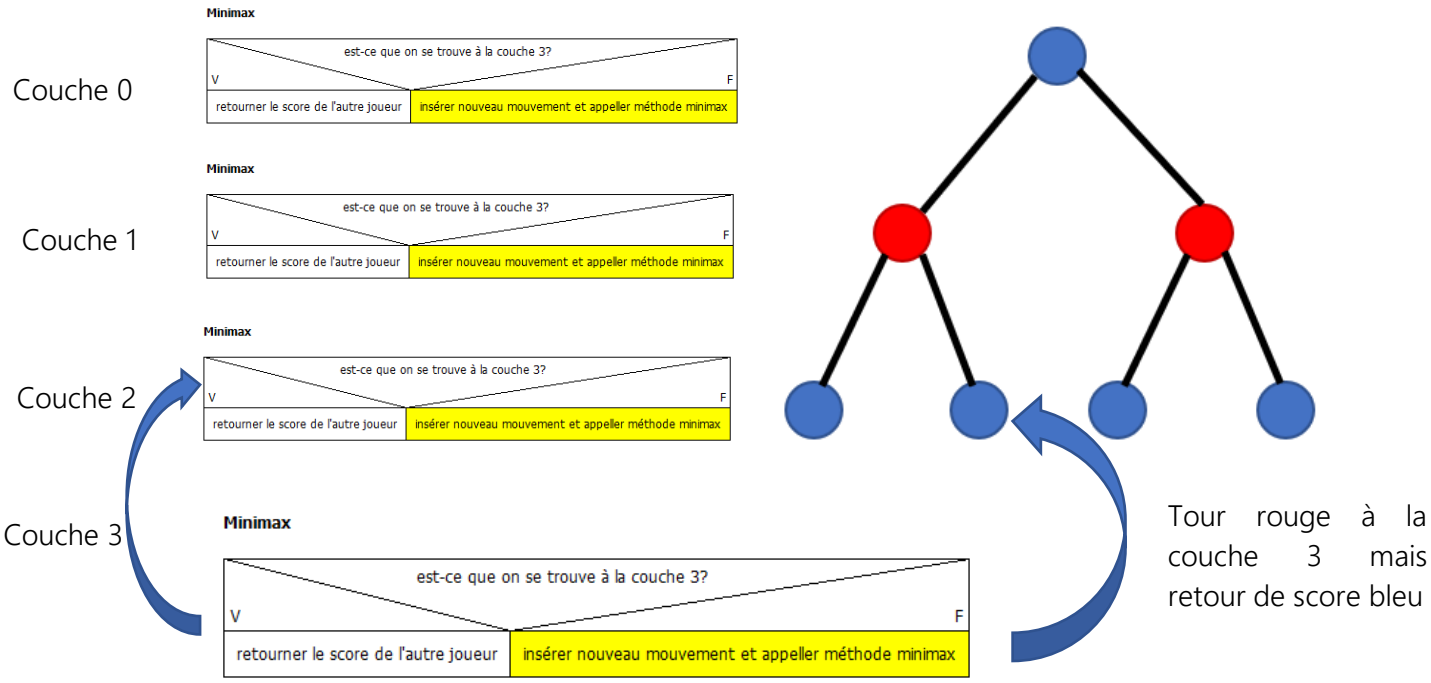


Schéma montrant la solution

4 CONCLUSION

4.1 BILAN

Au final, je suis assez fier de ce que j'ai réussi à faire. Tout mon cahier des charges a été effectué. J'ai réussi, malgré ce gros problème, à finir à temps et j'ai, même, pu améliorer l'IA en ajoutant certaines fonctionnalités.

4.2 PLANNING

Dans mon planning, les heures ne jouent pas trop, car je m'attaquais à un thème plus ou moins inconnu, et j'avais eu des problèmes, et d'autres choses, que je pensais qu'allaient me prendre plus de temps que, finalement, j'en ai pris moins.

4.3 AMÉLIORATION

J'aurais bien aimé pouvoir améliorer ma version Beta, car le problème était, qu'avec les différentes grandeurs, la camera ne se plaçait pas bien et les tiges avaient une hauteur disproportionnées, trop petites ou trop grandes. Autre chose à améliorer, ça serait l'IA, car je sais que mon IA n'est pas parfaite et, je me dis, que si elle arrivait à calculer plus de coups à l'avance sans perdre trop de temps, elle arriverait vraiment à être imbattable.

4.4 AVIS PERSONNEL

Ce qui m'a extrêmement plu c'est qu'au final j'étais assez libre, et Unity est une application que j'apprécie beaucoup et que j'utilise pendant mon temps libre. Ce TPI était pour moi un vrai challenge, car il est très complexe à faire et aussi à expliquer, mais j'aime bien les challenges, donc j'ai donné le plus que je pouvais pour faire ce TPI. Je pense que la partie la plus compliqué de mon TPI était l'IA, à cause de la difficulté de trouver les erreurs et, comme déjà dit avant, la documentation, car c'est dur d'expliquer quelque chose de si complexe.

5 ANNEXES

- Cahier des charges
- Horaire détaillé
- Planification
- Journal de travail

6 SOURCE

- Stackoverflow
  https://stackoverflow.com/
- Unity Documentation
  https://docs.unity3d.com/Manual/index.html
- Algorithme m'ayant aidé
  https://www.codeproject.com/articles/43622/solve-tic-tac-toe-with-the-minimax-algorithm
- Vidéo YouTube m'ayant aidé réaliser le fait de faire tourner grâce à la souris
  https://www.youtube.com/watch?v=xcn7hz7J7sI

7 REMERCIEMENTS

- M. Mathez
- M. Brechbühler
- Grand merci à Joachim Stalder pour m'avoir supporté tout au long de ce projet.