Conception agile de projets informatiques et génie logiciel

KARFA Hichem - KOLLI Lucas

27/11/2022

Image réaliser avec l'IA Midjounney

Contents

1 Présentation globale du projet

1.1 Comment fonctionne le jeu

Ce projet java prend la forme d'un jeu de balle au prisonnier. Il se joue à 2 joueurs sur le même clavier. Chaque joueur est accompagné de 4 autres personnages contrôlés informatiquement (appelé Bots). Lorsqu'un joueur à la balle, il tir vers l'équipe adverse. S'il touche un adversaire, celui-ci perd un points de vie. Lorsqu'un joueur n'a plus de point de vie au cours de la partie, il perd et disparait. La partie s'arrête lorsqu'un des joueurs principaux est éliminé. Si le tir est raté, la balle revient au joueur principal adverse et la partie continue. Un joueur peut effectuer une passe en visant un de ses allié. Passivement, la balle va de plus en plus vite, il faut donc essayer d'éliminer rapidement les bots avant d'affronter le joueur principal en face.

1.2 Comportement de l'IA

Il existe plusieurs niveaux de difficulté du jeu. On peut par exemple changer le mode de déplacement des bots. Il existe pour l'instant 2 modes de déplacement. Un mode normal, ou chaque bots se déplace régulièrement de gauche à droite dans un espace donné. Dans ce mode de déplacement, il y a des feintes. Elles permetent aux bots de changer brusquement de direction. Il est possible de rendre le jeu plus compliqué en augmentant le taux de feinte (donc en diminuant la variable feinte initialisé à 50). L'autre mode de déplacement des bots permet aux deux joueurs de contrôler leurs bots. Chaque bot se déplace de la même manière que le joueur principal, on bouge donc toute l'équipe d'un coup à l'image d'un baby-foot. Les bots tirs lorsqu'ils ont la balle, avec un angle aléatoire entre 70° et 190°.

1.3 Améliorations future

Concernant les changements et amélioration que nous aurions aimé faire, il y a un système de parade. Le joueur devrait appuyer sur une touche au moment ou il se ferait toucher avec un timing bien précis, afin de ne pas perdre de point de vie. Cela nous donnerai un levier supplémentaire afin de gérer la difficulté du jeu en ajoutant aux bots cette fonctionnalité qui serait plus ou moins précise.
Nous aimerion aussi rajouter plusieurs scènes, comme par exemple le choix de la difficulté et une scène de fin de partie avec un récapitulatif de la partie (pourcentage de tirs réussis, nombre de passes etc...)

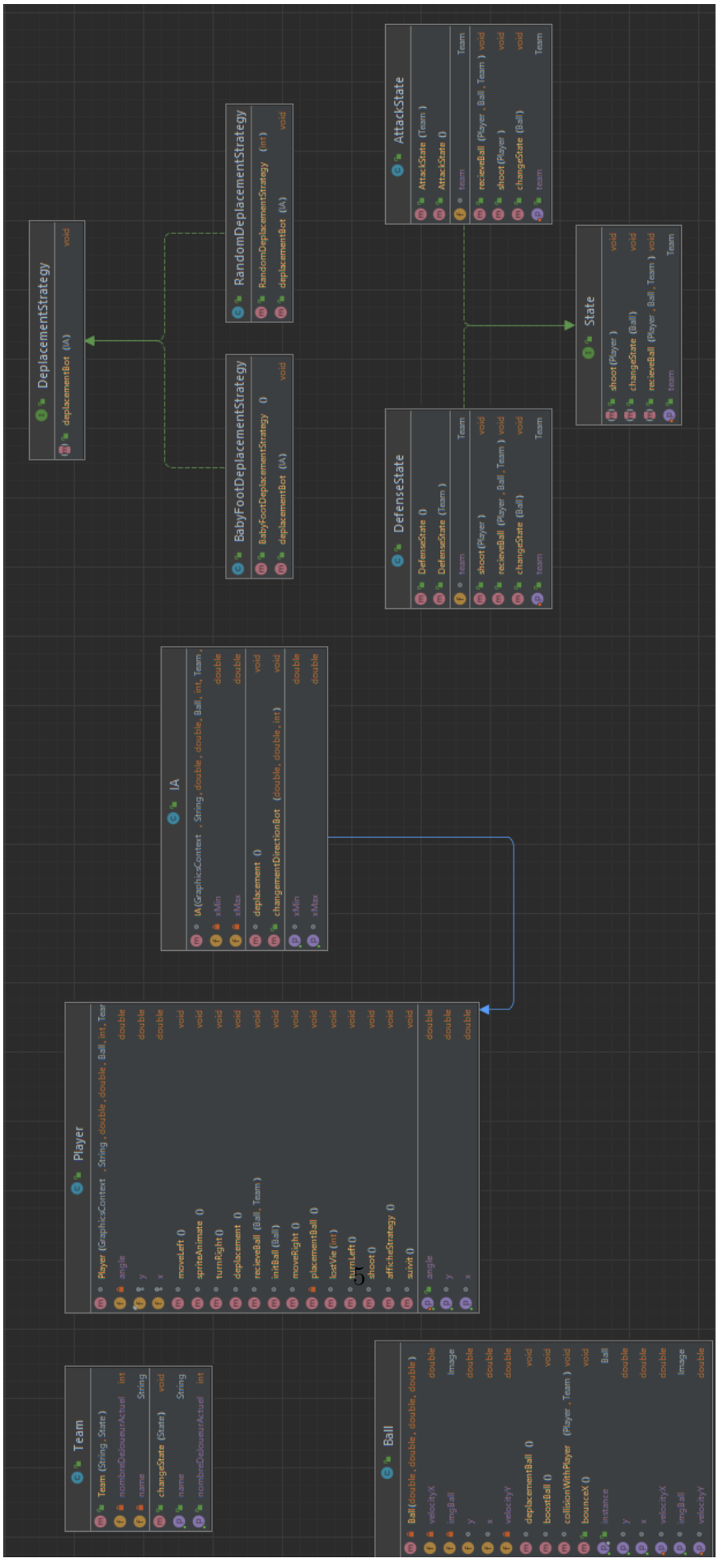
2 Motivation et explications des choix

2.1 Description rapide des classes

Nous allons tout d'abord décrire les classes :

- App : Créer la scène et lance l'application
- Field : S'occupe de gérer tout le jeu avec la boucle principale
- Display : Gère les affichages, des joueurs, des labels.
- LabelGestion : Construction des labels.
- Sprite : Animation des joueurs et de leur mort.
- Player : Caractéristiques des joueurs ainsi que les actions liées aux joueurs.
- IA : hérite de Player, afin de faire le même travail mais avec les bots.
- Ball : Classe qui a pour objectif de s'occuper à part entière de la balle.
- Team : Utilise les joueurs et les bots pour former les équipes. Il s'agit d'un tableau de joueurs.
- DéplacementStrategy : L'IA de déplacement, il y a une classe qui implémente celle-ci par mode de déplacement (deux ici).
- State : État des joueurs, afin de décider les actions à faire. Deux classes qui implémente celle-ci pour l'état attaque et l'état défense.

2.2 Choix d'architecture et diagramme



Afin de ne pas perdre trop de temps, et par manque de connaissances du concept, nous avons décidé de conserver les principes de bases de MVC sans pour autant le suivre à la lettre. Cela aurait pu nous être très bénéfique. En effet, répartir nos classes avec un but bien précis permettrait une bien meilleure structuration de nos classes. Comme vu au-dessus, l'architecture de notre application est assez complexe. Il y a énormément de dépendances entre les classes. Grace au MVC, nous aurions pu réduire grandement ces dépendances. Les classes que nous avons crées fonctionnerait bien dans un MCV, car le travail de ces classes ressemble à ce patron de conception. Premierement, les classes d'affichage pourrait être dans les Vues. La classe Field serait le contrôleur et les classes de création (Player, IA, Ball etc) le modèle. C'est un patron de conception qui s'adapte bien à ce genre de projet et qui aurait été un gain de temps. En revanche, d'autre patron de conception nous ont aidé à structurer notre application.

2.3 Designs Patterns

2.3.1 Designs Patterns utilisés

Afin d'avoir un code plus rigide, modulable et pour respecter un maximum le principe SOLI, nous avons structuré l'application en trois patron de conception. D : un lié à la création, qui est Singleton et deux autres pour le comportement. Il s'agit du design patterns stratégie et état.

2.3.2 Explication des designs patterns

- Singleton

  Nous avons utilisé Singleton pour la classe balle et afficheur. L'objectif étant de pourvoir une seule instance par classe. Cela permet de garantir l'unicité de cette instance et d'y obtenir un accès globale. En effet, nous avons besoin d'un seul objet pour la balle et un seul objet pour gérer l'afficheur. D'autres classes auraient aussi pu utiliser ce design pattern, comme par exemple le terrain de jeu (Field). Afin de rendre le jeu plus flexible (en vue d'un ajout de terrain par exemple), nous n'avons pas fait ce choix.

- Stratégie

  Nous avons remarqué que le comportement lié au différent type de déplacement de bots était assez similaire : il faisait exactement la même chose (indiquer au bot gauche et droite) mais d'une manière différente. En effet, le déplacement aléatoire indique au bot une direction, susceptible de changer lors d'une feinte.

Celui en baby-foot indique une direction, qui est la même que le joueur. Dans les deux cas, on indique au bot une direction. Ces déplacements sont susceptibles de changer à l'avenir, et il est tout à fait possible d'en rajouter. C'est pour cela que nous avons utilisé le design pattern stratégie. Nous avons crée une interface "Deplacement strategy". Cette classe sera implémentée par des classes concrète de déplacement, par exemple "Baby-foot strategy". Chaque bot se verra assigner en attribut un objet stratégie pour lui indiquer quel déplacement effectuer. Lors de l'exécution. Chaque bot se déplacera automatiquement en appelant la classe qui lui a été attribuée. Cela permet de respecter le principe d' "Ouvert/Fermé". D'autres stratégies peuvent être implémentés sans modifier le contexte initial, en rajoutant une classe.

- État

  La gestion du tir de la perte de point de vie et de la réception de la balle fût particulièrement complexe car nous devions tester à chaque fois plusieurs paramètres (le jouer a-t-il la balle, quelle équipe tir, combien de point de vie à le joueur etc). le code était donc difficilement maintenable et pas approprié pour de futur améliorations. Pour résoudre tous ces problèmes nous avons utilisé le design pattern Etat. Chaque équipe se voit attribué un état. Pour l'instant, il y a l'état Attaque et l'état défense. Lorsqu'un joueur tir avec l'état attaquant, il déclenche le déplacement de la balle. Cependant, lorsqu'il reçoit la balle, il récupère cette balle (pour les passes entre joueurs). Les défenseurs quant à eux ont aussi une fonction pour tirer qui pourrait être utilisé pour parer (voir Futures améliorations au début). Ils ont aussi une fonction pour recevoir la balle qui déclenche une perte de point de vie. Chaque fois qu'un joueur est éliminé/touché/la balle sort du terrain, chaque équipe change d'état et cela permet au contexte d'execution de se lancer sans connaître les paramètres des joueurs. Tout comme stratégie, l'implémentation de ce patron de conception se fait via une interface (State) qui implémente différentes classes. Ces différents états connaissent donc les autres états, ce qui fait la différence avec stratégie (chaque état se connait mutuellement). Chaque équipe se fait donc attribué un état. Comme dit précédemment, ce design pattern évite l'utilisation de gros blosc conditionnel, tout en respectant le principe "Ouvert/Fermé" ainsi que le principe de responsabilité unique.