IPB
ENSEIRB
MATMECA
BORDEAUX

Rapport de Projet : Tower Defense

équipe : projetss6-tower-19048

Auteurs : Youri CHANCRIN, Théo DESCOMPS, Samuel KHALIFA, Jean-Baptiste MARTINEZ

Responsable de projet : David Renault

Enseignant référent : Frédéric Herbreteau

Première année, filière informatique
Date : 12 mai 2023

Version actuelle du document : finale

Table des matières

1 Sujet

L'objectif de ce projet est la réalisation d'un jeu de type tower defense.
Ce genre peut se résumer en une seule phrase :

Des attaquants envahissent une zone défendue par des défenseurs.

1.1 Contraintes

La programmation fonctionnelle est le paradigme mis en avant. Nous évitons donc au maximum la programmation impérative ou encore la programmation orienté objet. L'exécutable est réalisé en TypeScript, une surcouche de JavaScript que nous utilisons pour typer le contenu de notre programme.

1.2 Theme

Nous avons choisi de créer un jeu de tower defense autour du thème du Pastafarisme, en faisant attention à éviter toute forme de violence. Selon cette religion, la température de la Terre est inversement proportionnelle au nombre de pirates qu'elle contient. Cette problématique est aujourd'hui d'une importance majeure, et nous voulons sensibiliser les joueurs aux causes climatiques. L'objectif du jeu est donc de répandre le Pastafarisme, prêché par des pirates qui correspondent aux défenseurs de notre jeu, et de défendre le Monstre Spaghetti Volant, notre sauveur.

2 Analyse du sujet

Notre description du jeu, en partie 1, nous permet d'en tirer les éléments fondamentaux.
Parmi ces éléments se trouvent les entités, comme les attaquants, mais aussi les défenseurs. Bien qu'elles soient antagonistes, ces entités ont des similarités, notamment dans le traitement de leurs actions.
Nous décidons donc de les regrouper sous un même nom, les **acteurs**.

2.1 Entité Acteur

Les acteurs forment un ensemble d'entités aux comportements et états variés, qui intéragissent entre elles.
Le type acteur est très générique et peut représenter beaucoup de choses. Pour garder une consistence entre nos acteurs, on définit des archétypes :

— Spaghetti Monster : Le gentil monstre spaghetti que les défenseurs doivent protéger.
— Good Guy : Les défenseurs.
— Ignorant : Un type d'attaquant.
— Ignorance Spreader : Un autre type d'attaquant.
— Spawner : Un point d'apparition pour les attaquants.
— Ground : Un bout de chemin sur lequel les attaquants se déplacent.

Les types Good Guy, Ignorant et Ignorance Spreader représentent les défenseurs et les attaquants. Ils découlent directement de la description du jeu. Cependant, nos acteurs comportent aussi un type Spawner et Ground. Ces types ne peuvent ni attaquer ni être attaqués, mais ils peuvent interagir avec le reste des entités, ce qui justifie qu'ils soient aussi considérés comme des acteurs.
En effet, Spawner peut être vu comme un attaquant immortel, et Ground un défenseur qui contraint le déplacement des attaquants. Passons aux interactions entre entités.

Chaque acteur a besoin de disposer d'un ensemble de comportements qui puisse évoluer au cours de la partie, comme pour simuler des effets incapacitants ou stimulants.
Pour que ces interactions prennent effet, nos entités ont également besoin d'un état interne.
Les états internes des entités comportent toujours :
— Une **position**
— Un **type** d'entité

Avec la diversification des acteurs vient le problème de la spécialisation qui se traduit par l'ajout d'attributs *spécifiques* à chaque acteur.

Par exemple, les acteurs béligérants ont besoins d'attributs spécifiques tels que des points de croyance et une limite que ces points peuvent atteindre.

On pourrait rajouter tous les attributs nécessaires à tous les comportements à tous les acteurs. Bien que cela permettrait un typage plus précis et moins d'erreurs de programmation, cette pratique n'a pas été retenue pour trois raisons :

— Premièrement, cela entraînerait une modification sur le type acteur à chaque nouvel attribut ajouté, et entraînerai une modification de la signatures de certaines fonctions auxiliaires comme le constructeur.
— Deuxièmement, le type acteur aurait toujours plus d'états possibles, et deviendrait de plus en plus complexe et difficile à comprendre et gérer.
— Troisièmement, les erreurs de programmation liées à l'utilisation de mauvais noms de variable dans les dictionnaires de type "any" (quelconque) peuvent être évitées en passant par l'utilisation de getters et setters.

Nous avons donc séparé la responsabilité des attributs dépendants de certains comportements dans un champs **externalProps**, qui est un dictionnaire de type quelconque, muni de getters et setters. Ce typage lui permet de contenir tout type d'attribut.

Nos acteurs sont ainsi parés à toute éventualité d'évolution. Cependant, bien qu'ils disposent de comportements pouvant évoluer, ils ne peuvent modifier leurs états internes directement. Nous avons besoin de quelque chose d'ordre supérieur pour gérer leurs interactions. Quelque chose comme un système de phase.

2.2 Système de phase

Nos acteurs effectuent des actions qui ont des conséquences. Cela implique qu'il y a un avant et un après aux actions. Pour chaque type d'action, on crée une phase. Elle agit comme un ordonnanceur. Elle obtient les intentions de chaque acteur pour l'action qu'elle traite, et applique les modifications, créant une nouvelle liste d'acteur.

Nommons un tour de jeu comme une exécution de l'ensemble de nos phases.

Au cours d'un tour de jeu, les phases doivent permettre aux acteurs de réaliser les actions suivantes :

— Se déplacer
— Attaquer un autre acteur
— Soigner un autre acteur
— Faire apparaitre des nouveaux acteurs
— Se retirer du jeu

Dans un soucis de cohérence et d'équité entre les acteurs, leurs intentions sont traitées de manière synchronisées au cours des phases. Si ce n'était pas le cas, l'ordre dans lequel les acteurs jouent pourrait influer sur le résultat du tour de jeu. Par exemple, si un acteur joue dans sa phase d'attaque, inflige des dégats à un ennemi, puis joue sa phase de déplacements, s'eloignant et privant l'ennemi d'une juste riposte.

Ceci ne constitue qu'un choix de jeu, et le découpage d'un tour de jeu en une liste de phase permet de moduler le déroulement d'un tour. Il en résulte qu'il est très simple d'ajouter des phases de jeu, ou de changer l'ordre dans lequel les acteurs agissent.

Enfin, il peut arriver que les intentions des acteurs soient incompatibles. Par exemple, si deux acteurs ayant des collisions veulent aller au même endroit. Chaque phase doit pouvoir résoudre les conflits qu'elle génère.

Nous disposons donc d'acteurs, qui peuvent désormais intéragir entre eux avec une grande modularité.

Il ne nous manque plus qu'une seule notion issue de notre description du jeu en partie 1, celle de zone, dans laquelle nos acteurs devraient rester. Cela rejoint l'idée de limiter certaines actions des acteurs, en l'occurence, on veut interdire les acteurs de quitter la zone de jeu, autrement appelée Monde.

2.3 Monde

Un monde est un ensemble de coordonnées sur lesquelles nos entités peuvent exister.
Le monde est utilisé pour au moins deux besoins qui sont :

— Vérifier la cohérence des positions des acteurs.
— Créer le chemin que les attanquants devront parcourir au cours de leur invasion sur la zone de jeu.

Avec ces trois élements que sont les acteurs, les phases et le monde, on va pouvoir créer des acteurs de différents types, effectuant diverses actions, et ce dans une zone définie.
La cohérence des positions dans le monde mène à un résultat similaire à celui de la figure 1.

(a) Monde de taille 10 x 10.

(b) Monde de taille 20 x 20.

FIGURE 1 – Différents affichages du jeu réalisé.

La figure 1 contient deux mondes, générés aléatoirement : un monde de hauteur et de largeur de 10 cases (a), et un autre de hauteur et largeur de 20 cases (b).

3 Réalisation

Cette partie concerne des implémentations particulièrement intéressantes répondant aux besoins énoncés dans la section 2. Une première problématique à résoudre était celle de la génération de chemin dans le jeu. Cette génération respecte certaines règles, mais implique également l'aléatoire.

3.1 Génération aléatoire de chemin

Nous résolvons le problème de génération aléatoire de chemin en deux étapes. D'abord, nous positionnons des points de passage, ensuite nous les connectons pour former le chemin final.

3.1.1 Génération des points de passage

Un point de passage est une position par laquelle le chemin généré doit passer.
Les points de passage sont générés en considérant une direction à prendre le long d'un axe du monde. Par exemple, il peuvent être générés de manière à ce que chaque nouveau point de passage ait une coordonnée supérieure sur l'axe des abscisses au point de passage précédent. Les premiers points de passage se situent sur les acteurs de type Spawner, et les derniers points de passages sur les acteurs de type Spaghetti Monster.
Pour que les chemins ne soient jamais trop courts, nous contraignons les spawners à être générés sur un bord du plateau de jeu opposé au bord sur lequel seront générés les monstres en spaghetti. Les spawners sont donc répartis sur un axe (qu'on appellera "axe des spawners") dont la direction est perpendiculaire à la direction selon laquelle les points de passages sont générés. Les points de passage sont numérotés dans l'ordre croissant en fonction de leur coordonnée sur l'axe perpendiculaire à l'axe des spawners.

Cette numérotation définit l'ordre dans lequel ils doivent être parcourus, comme le présente la Figure 2.

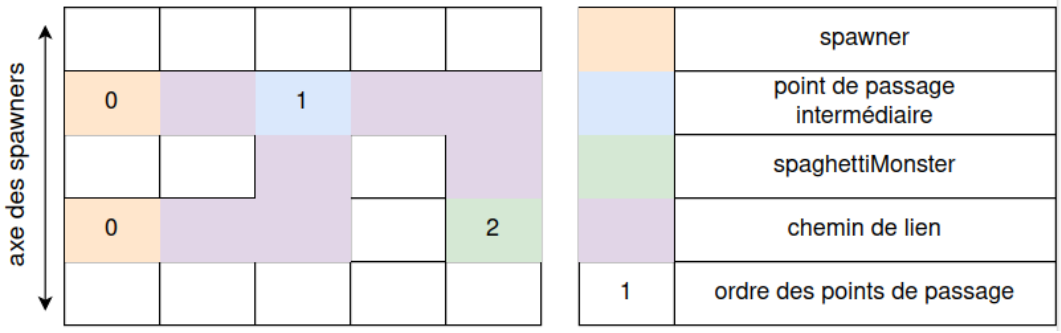

FIGURE 2 – Exemple de génération aléatoire de chemin dans un monde de taille 5 x 5.

Il reste maintenant à lier ces points de passage pour former un ou plusieurs chemins menant des spawners aux monstres en spaghetti.

3.1.2 Liaison des points de passage

Nous voyons dans la Figure 2 que des chemins de lien ont été générés. Ces chemins sont générés d'une manière très simple qui consiste à relier chaque point de passage de numérotation n aux points de passage de numérotation $n-1$, en commençant par placer des cases de chemin le long d'un axe (ici des abscisses) puis le long de l'autre axe (ici des ordonnées).

La fonction reliant deux points de passage est une fonction récursive terminale prenant en paramètres une position initiale et une position finale à relier. À chaque appel récursif, il suffit de modifier la position initiale en l'ayant déplacée de une case vers la case finale, et ce jusqu'à ce que la case initiale corresponde à la case finale.

Il est facile de se convaincre que de cette manière, n'importe quel chemin peut être généré, pour peu qu'on place correctement les points de passage nécessaires à sa génération. Cette fonction de liaison de chemin, une fois les points de passage placés, étant déterministe, les attaquants peuvent utiliser cette même fonction pour se déplacer correctement sur le chemin. Pour une défense optimale, le joueur a tout intérêt à placer ses défenseurs près de ce chemin.

3.2 Joueur automatique

Généralement, dans un jeu de Tower Defense, la répartition des défenseurs dans le monde est confiée au joueur. Jouer un coup consiste à placer un défenseur à une position donnée. Dans le cas du développement de notre jeu, ce placement est automatique et peu donc être vu comme faisant partie de la phase de spawn. Nous séparons tout de même ces deux phases dans un soucis de modularité. Le joueur automatique est implémenté comme étant un acteur de type "player", possédant une action "play".

3.2.1 Joueur intelligent

Le joueur que nous avons implémenté diversifie ses coups à l'aide de deux algorithmes. L'algorithme utilisé en priorité consiste à jouer des coups aléatoires proches des cases de chemin, en priorisant celles qui sont le moins nombreuses dans leur rangée. Une rangée étant dans la même direction que l'axe des spawners.

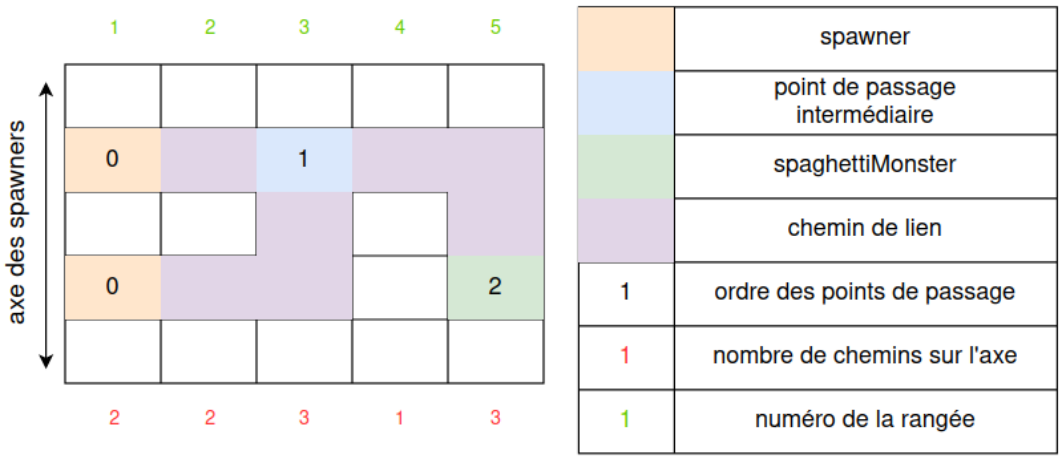

FIGURE 3 – Exemple de cases de chemin par rangéee dans un monde de taille 5 x 5.

Dans l'exemple de la Figure 3, les coups seront d'abord joués dans la 4ème rangée, puis dans les deux premières, et enfin dans les rangées 3 et 5. Cet algorithme utilise un générateur de nombre aléatoires basé sur une implémentation pure récursive terminale du mélange de Fisher-Yates, qui termine.

L'implémentation de cet algorithme nous permet de passer en paramètres une portée dans laquelle les coups peuvent être joués, autour des cases. Une fonction de distance, permettant de vérifier si la portée est respectée, doit également être précisée. Par exemple, la distance euclidienne peut être utilisée, et fournirait une portée circulaire.

Lorsque ce premier algorithme a épuisé ses possibilités, le joueur se met à jouer des coups valides aléatoires.

3.2.2 Joueur aléatoire

Ce deuxième algorithme énumère les positions libres, les mélange en utilisant la même implémentation du mélange de Fisher-Yates, et les parcourt jusqu'à trouver une position libre sur laquelle il peut jouer.

Il est évidemment possible d'utiliser ces deux algorithmes indépendamment, pour créer des joueurs aux comportements diversifiés. L'implémentation du joueur intelligent pourrait alors être légèrement améliorée afin d'augmenter progressivement sa portée et finir par atteindre toutes les cases du plateau.

Une autre manière de créer des joueurs aux comportements diversifiés est d'utiliser le concept de décorateur fonctionnel.

3.3 Diversification dynamique des comportements

En exécutant le jeu, vous remarquerez que les attaquants ne se déplacent pas tous à la même vitesse. Et pourtant, ils utilisent tous la même fonction de déplacement ! Le comportement de celle-ci est "simplement" augmenté en utilisant le principe de décorateur fonctionnel.

3.3.1 Choix d'implémentation des décorateurs, ou générateurs d'action

Nos décorateurs fonctionnel **purs** permettent, de manière définitive comme dynamique, de modifier les comportements des entitées du jeu. Ils permettent de ne pas avoir à écrire une nouvelle fonction, ou à passer de nombreux paramètres supplémentaires à toutes les fonctions de comportement, à chaque fois que l'on souhaite modifier un comportement.

En excluant les implémentations qui ne respectent pas le principe de pureté, au moins deux possibilités d'implémentation de ces décorateurs demeurent :

— Modifier le type de retour des actions (une action décrivant un comportement) afin qu'elles retournent, en plus de leur résultat, la prochaine fonction à utiliser pour cette même action. Cette implémentation complexifie la création des actions, et rend leur type de retour beaucoup moins évident. Ce n'est pas la solution retenue.

— Séparer les actions des **générateurs d'action**, qui seront utilisés pour mettre à jour les actions après les avoir exécutées. Dans cette implémentation, les actions restent inchangées, mais une nouvelle variable s'ajoute au type Actor, qui est un objet contenant pour chaque action son générateur, qui est une fonction retournant la prochaine action à utiliser, ainsi que le nouveau générateur, à l'état potentiellement différent, qui sera utilisé une fois la prochaine action exécutée. C'est la solution retenue.

Pour ralentir le déplacement d'un attaquant, il suffit de décorer son action de déplacement afin qu'elle ne soit éxecutée qu'une fois sur deux lorsqu'elle est appelée.

3.3.2 Exemple concret

En suivant le principe de généricité et de citoyenneté de première classe, nous avons programmé un décorateur, dont le fonctionnement est détaillé en Figure 4.

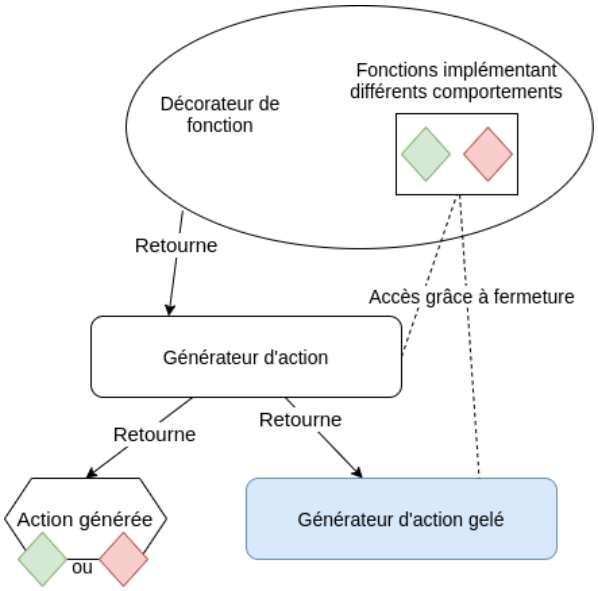

FIGURE 4 – Décomposition d'un décorateur retournant alternativement une des fonctions données.

Ce décorateur prend en paramètres une fonction à décorer, la fonction à exécuter lorsque la fonction décorée ne doit pas être appelée, et une valeur définissant tous les combien de tours la fonction décorée doit être exécutée. On peut se servir des actions par défaut déjà programmées, telles que le déplacement de 0 case, qui revient à ne pas se déplacer, pour définir la fonction par défaut.

Ce décorateur retourne l'exécution d'une fonction interne, qui est un générateur d'action, en suivant le principe de fermeture. Ce générateur d'action retourne l'appel **gelé** d'un autre générateur, dont les paramètres auront été mis à jour, et l'action générée, qui définira le prochain comportement de l'entité concernée. Le gel du générateur retourné permet d'éviter un appel récursif infini.

Cette fonction est typée de manière récursive, et un type ActionGenerators décrivant les générateurs d'action qu'un acteur doit contenir est créé, et basé sur le type ActorActions qui définissait déjà les types des actions des acteurs.

3.3.3 Mise à jour des actions et des générateurs

Pour que cette implémentation fonctionne, il faut que le générateur et l'action soient mis à jour à chaque exécution de l'action. Ceci n'est pas automatiquement réalisé par l'action, et c'est le système de **phases** qui nous permet de mettre en place et synchroniser cette mise à jour. Les phases permettent de mettre à jour les acteurs en fonction de leurs intentions. Cela suppose que l'action de l'acteur concernant la phase en question, de déplacement par exemple, a été exécutée et a donné son résultat. La fonction d'éxécution propre à la phase, et retournant le nouvel acteur en prenant en compte les intentions

retournées par l'action de l'acteur, en profite donc pour retourner un acteur dont le générateur et l'action on tous les deux été mis à jour.

3.3.4 Pour aller plus loin

En suivant cette implémentation des phases et des générateurs d'action, un acteur n'est pas le seul à pouvoir modifier son propre comportement. N'importe quelle phase pourrait traîter les données qu'elle reçoit de manière à ce qu'un acteur modifie le comportement d'un autre acteur. Par exemple, les défenseurs pourraient ralentir les attaquants, en précisant dans le retour de leur action, le décorateur devant être utilisé sur les actions des attaquants affectés.

On pourrait imaginer d'autres scénarios qui complexifient légèrement le programme : la décoration des décorateurs. En effet, un attaquant pourrait grâce à un décorateur ne se déplacer qu'une fois tous les deux tours. Un défenseur paralyse cet attaquand pendant 3 tours, qui ne peut alors pas se déplacer. L'action décorée doit donc être encore décorée. On doit donc pouvoir créer plusieurs niveaux de décoration, et dans notre implémentation, il faudrait alors pouvoir décorer non pas l'action, mais le générateur d'actions. Cela n'a pas été implémenté, mais nous immaginons qu'il faudrait alors que les générateurs d'action retournent :

- Le prochain générateur d'actions 'parent' à appeler.
- L'état du générateur d'action imbriqué (qui peut aussi être retourné comme générateur d'action 'parent' lorsque le générateur d'action 'parent' effectue sa dernière exécution).
- L'action générée (sur plusieurs niveaux de décoration, cette action correspondrait à la fonction par défaut du générateur 'parent' ou le résultat du générateur imbriquée, dont l'état devra alors être mis à jour).

Cette structuration permettrait plusieurs niveaux de décoration.

4 Conclusion

Les efforts de pureté et modularité effectués dans ce projet nous permettent de facilement tester et maintenir chaque fonction, sans toujours devoir créer tout un environnement d'exécution pour celles-ci, et sans se préoccuper des changements de l'état de l'environnement.

Les inconvénients sont parfois la complexification d'écriture des fonctions, notamment des décorateurs fonctionnels purs.

La pureté n'est pas compatible avec l'aléatoire. On pourrait argumenter sur le fait que l'aléatoire n'existe pas et qu'on est sur un système pseudo-aléatoire. Cependant, une vrai piste d'amélioration vers plus de pureté serait plutôt que les fonctions se servant de l'aléatoire prennent en paramètre cette fonction aléatoire, ou bien même directement son résultat. On pourrait alors tester ces fonctions de manière précise et exhaustive.

Les inconvénients du paradigme de programmation utilisé comprennent également les réductions de performance, notre programme ne souffre cependant pas de ce problème. De plus, dans un jeu du type tower defense, le délais entre les tours de jeu combiné à de la programmation asynchrone peut permettre de rendre ce problème invisible.