Rapport projet S6 : Tower Defense

Yannis Chappet-Juan, César Larragueta
Corentin Perdrizet, Joris Rousere

Avril 2023

FIGURE 1 – Image tirée de la page du projet

Table des matières

1 Introduction

Le genre Tower Defense constitue une catégorie spécifique de jeux vidéo stratégiques. Dans ces jeux, le joueur est chargé de la protection d'un territoire, qui est constamment menacé par des assauts répétés d'ennemis obstinés. Ces derniers parcourent souvent une série de sentiers définis, menant à un point spécifique du territoire, généralement identifié comme une zone vulnérable. Pour contrecarrer les avancées ennemies, le joueur se doit d'établir des tours de défense qui, par divers moyens - tels que l'élimination, le blocage ou le ralentissement des adversaires - permettent de maintenir la sécurité du territoire. Les jeux de Tower Defense offrent une grande diversité, se distinguant par leurs paysages, personnages, mécaniques de mouvement et possibilités d'amélioration ou de modification des différentes entités.

L'objectif de ce projet est de concevoir et développer un jeu dans ce genre, en utilisant le langage de programmation TypeScript et en adoptant une approche de programmation fonctionnelle.

Cette approche a été privilégiée pour ses nombreux avantages, tels que la stablité et la clarté du code et l'utilisation de fonctions pures.

2 Définition du problème et analyse des besoins

Nous voulons créer un jeu de tower defense, il existe différents besoins :

1. Le jeu doit offrir une variété de tours avec des capacités distinctes afin d'offrir des stratégies différentes aux joueurs.
2. Les adversaires doivent suivre un ou plusieurs chemins définis vers un point vulnérable, et chaque adversaire doit avoir une résistance et une vitesse spécifique.
3. Le joueur doit pouvoir placer et améliorer les tours durant le jeu.
4. Le jeu doit inclure plusieurs niveaux de difficulté et des scénarios variés pour maintenir l'intérêt des joueurs.

2.1 Décomposition du problème en sous-problèmes (acteurs, phases de jeu, moteur de jeu)

La base du jeu est un univers en 2D dans lequel des acteurs interagissent entre eux. Pour gérer ces interactions, nous avons décomposé le problème en prenant en compte une portée limitée des acteurs pour leurs actions. Ils peuvent donc seulement interagir avec les autres acteurs situés à proximité. Par exemple, une tour peut tirer sur ses ennemis dans un rayon défini autour d'elle.

Le jeu est structuré en différentes phases (telles que la phase de déplacement qui fait bouger les ennemis). Chaque phase de jeu recueille les intentions des acteurs (par exemple, l'intention des ennemis de se déplacer) et produit également une intention globale. Le moteur de jeu est responsable de récupérer cette intention collective et de la résoudre, c'est-à-dire de générer un nouvel état du monde sans conflits. Pour ce faire, il peut être amené à supprimer des acteurs, et ainsi, il détient le contrôle sur le déroulement du jeu.

Cette approche modulaire de la conception du jeu offre une flexibilité considérable, facilitant l'introduction de nouveaux acteurs et de nouvelles phases de jeu. Si l'on souhaite ajouter un nouvel acteur, il suffit de définir son comportement et ses interactions avec les autres acteurs existants. De la même manière, l'introduction d'une nouvelle phase de jeu nécessite simplement la définition de ses effets sur l'état global du jeu et sur les intentions des acteurs.

La modularité et la décomposition du jeu en phases distinctes permettent de créer des structures de code plus lisibles, maintenables et évolutives. Chaque élément du jeu peut être développé, testé et débogué indépendamment des autres, ce qui facilite grandement le processus de développement.

Cette approche s'aligne bien avec les principes de la programmation fonctionnelle. En effet, chaque phase du jeu peut être vue comme une fonction qui prend l'état actuel du jeu et produit un nouvel état. De plus, les intentions des acteurs sont également traitées comme des données immuables qui sont transformées en un nouvel état du monde par le moteur de jeu. Cette transparence référentielle, où chaque fonction produit le même résultat pour les mêmes entrées, facilite le raisonnement sur le code et permet d'éviter de nombreux bugs courants dans le développement de jeux.

3 Conception et modélisation

La décomposition de notre jeu étant désormais établie, nous avons procédé à l'implémentation de types spécifiques pour représenter divers éléments clés de notre jeu. Ces éléments comprennent entre autres le monde, les acteurs, les phases de jeu et le moteur de jeu.

3.1 Conception du monde de jeu

Dans le but de structurer notre monde, nous avons établi plusieurs types distincts, chacun ayant un rôle spécifique dans le schéma global.

Tout d'abord, nous avons défini le type **WorldSize**. Ce type caractérise la taille du monde, en termes de largeur et de hauteur. Voici la déclaration de ce type :

```
type WorldSize = {width: number; height: number};
```

Ensuite, nous avons introduit le type **Point**. Ce type permet de localiser précisément un point dans notre monde en deux dimensions grâce à ses coordonnées x et y :

```
type Point = {x: number; y: number};
```

Ces deux types sont ensuite utilisés pour définir le type **World**, qui représente l'intégralité de notre univers. Ce monde est composé de sa taille, de scores, ainsi que d'un ensemble d'acteurs et de paysages :

```
type World = {
    size: WorldSize;
    score: Score;
    actors: Actor[];
    landscapes: Landscape[];
};
```

Le monde contient un tableau d'acteurs, dont nous détaillerons la définition ultérieurement, ainsi qu'un tableau de paysages. Un paysage est défini par son nom (qui permet de savoir à quel type de paysage il appartient, comme la terre, un chemin, une rivière, etc.), une heuristique (particulièrement utile pour gérer les chemins et le mouvement des ennemis), une position dans le monde et une donnée tile. Cette dernière représente une fonction de rendu que nous pouvons appeler pour afficher le paysage, que ce soit en console, en HTML, ou dans un autre format si défini.

```
type Landscape = {
    name: string;
    heuristic: number;
    pos: Point;
    tile: Tile;
};
```

L'avantage de définir les types WorldSize et Point réside dans la modularité qu'ils apportent à notre jeu. Par exemple, si nous voulions étendre notre jeu à trois dimensions, nous pourrions facilement ajouter un attribut z au type **WorldSize**. De la même manière, le type **Point** pourrait être amélioré pour gérer cette nouvelle dimension. Cette approche assure une grande flexibilité à notre structure de données, ouvrant ainsi la porte à des évolutions futures.

3.2 Conception des acteurs et de leurs comportements

Dans le cadre de notre jeu, un acteur est un élément clé qui interagit activement avec l'environnement du monde. Chaque acteur a une position spécifique, représentée par un type **Point**, un certain nombre de points de vie, une fonction de rendu, et un ensemble d'actions qu'il peut réaliser. Par exemple, un acteur ennemi pourrait avoir l'action de se déplacer. Voici comment nous avons défini le type **Actor** :

```
type Actor = {
    pos: W.Point;
    hp: number;
    tile: W.Tile;
    actions: Actions;
};
```

Chaque action est une fonction qui prend en entrée un acteur et le monde, puis renvoie un nouvel acteur avec des caractéristiques modifiées. Ceci est illustré par le type **Action** suivant :

```
    type Action = (actor: Actor, world: W.World) => Actor;
```

Pour faciliter l'appel des actions, nous avons établi un type Actions qui associe à chaque action une clé de type string. De cette façon, nous pouvons simplement utiliser la clé pour invoquer l'action correspondante :

```
    type Actions = {
        [key: string]: Action;
    };
```

Cette structuration et typologie des acteurs et de leurs actions permettent une grande flexibilité dans la conception du jeu. Elle offre la possibilité de définir une variété d'acteurs avec des comportements distincts, tout en maintenant un code propre et organisé. De plus, l'utilisation de fonctions pour les actions s'inscrit dans notre approche de programmation fonctionnelle, favorisant l'immuabilité des données et simplifiant le débogage.

3.3 Conception des phases de jeu

Le type **Phase** joue un rôle crucial dans la gestion des différentes phases du jeu. Une phase est une fonction qui prend un monde en entrée et renvoie un tableau d'acteurs avec des caractéristiques potentiellement modifiées. C'est la définition de notre type **Phase** :

```
type Phase = (world: W.World) => A.Actor[];
```

Le choix d'un tableau d'acteurs comme valeur de retour, plutôt que le monde lui-même, est stratégique. En effet, une phase ne doit produire qu'une "intention", un état souhaité pour les acteurs sans imposer de modifications directes sur le monde. C'est le moteur de jeu qui, par la suite, prendra en compte ces intentions pour actualiser l'état du monde.

Cette approche confère un haut degré de modularité à notre conception, puisque chaque phase peut être définie indépendamment des autres. De plus, cela renforce notre adhérence à la programmation fonctionnelle, en séparant clairement la définition de l'intention (par les phases) et son application effective (par le moteur de jeu).

3.4 Conception du moteur de jeu

Le moteur de jeu, dans notre conception, est défini comme un "résolveur de phase" (PhaseSolver). Son rôle est d'analyser les intentions exprimées par les acteurs à l'issue d'une phase, de les comparer à l'état actuel du monde et de résoudre les éventuels conflits. Ce processus peut entraîner la mise à jour des positions des acteurs, l'application des effets de certaines actions, ou encore la suppression des acteurs qui n'ont plus de points de vie.

La définition du type PhaseSolver est la suivante :

```
type PhaseSolver = (updatedActors: A.Actor[], world: W.World) => W.World;
```

Cette fonction prend en entrée un tableau d'acteurs (updatedActors), qui représente l'état souhaité des acteurs après une phase, et le monde actuel (world). Elle renvoie un nouveau monde (W.World) qui reflète le résultat de la résolution des conflits entre les intentions des acteurs et l'état actuel du monde.

Cette approche préserve la pureté des fonctions en séparant clairement l'expression des intentions (à travers les phases) et leur réalisation effective (via le PhaseSolver). Cette séparation des préoccupations favorise une conception modulaire et flexible, qui est l'un des principaux avantages de la programmation fonctionnelle. Nous avons aussi utilisé le type suivant :

```
    type PhaseMap = {
      [key: string]: PhaseSolver;
    };
```

Ce type, qui associe une clé de type "string" à une fonction de résolution de phase, offre une grande flexibilité pour la gestion des phases de jeu. En effet, grâce à cette structure, il est possible d'appeler les fonctions de résolution de manière dynamique en utilisant simplement le nom de la phase comme clé. Cette approche facilite également l'ajout de nouvelles phases, qui peuvent être intégrées au système en définissant simplement une nouvelle clé et la fonction de résolution correspondante.

4 Mise en oeuvre

L'implémentation du jeu étant faite, nous pouvons développer l'articulation des différents fichiers entre eux : toutes les implémentations nous permettent de pouvoir utiliser les méthodes natives en TypeScript par exemple concat/push ou map/filter sur les tableaux.

La nature du projet nous laissant une grande liberté créative quant aux caractéristiques de notre jeu, nous avons été amenés à faire certains choix :

- Les *Landscapes* peuvent être :
  - Des *Path* correspondant au chemin emprunté par les ennemis pour aller de leur point d'apparition à leur objectif
  - Des *Grass* correspondant à des éléments de paysages impracticables
- Les *Acteurs* peuvent être :
  - Des *Mobs* : ce sont les ennemis du joueur, ils empruntent les *Path* pour traverser le monde, ils possèdent des caractéristiques de vitesse de déplacement et un certain nombre de points de vie,
  - Des *Tours*, pouvant être activées en échange de monnaie virtuelle, à certains emplacements prédéfinis du monde, elles ont des caractéristiques de portée,(*c.a.d* la distance maximale du *Mob* qu'elles peuvent toucher) et de dégats qu'elles causent aux *Mobs*,
  - Des *Cactus*, qui peuvent être placés de manière temporelle sur les *Path* dans le but de ralentir les *Mobs*.

Au début du développement, notre objectif premier était de créer un chemin aléatoire entre le point d'entrée et le point de sortie du monde du jeu. Nous avons effectué des tests avec différentes tailles de monde et avec des tours placées de manière aléatoire comme suit :

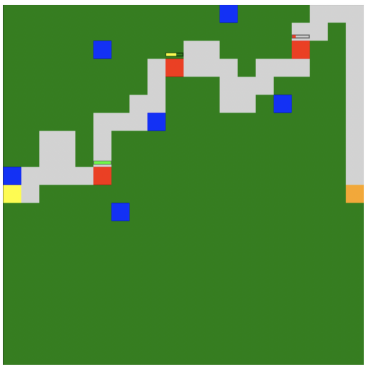

FIGURE 2 – Monde avec placement des tours aléatoires

Sur cette affichage, les *mobs* sont représenté par des carrés rouges, ils doivent se déplacer de l'entrée (carré jaune) à la sortie (carré orange) en essayant de survivre aux attaques des *towers* représentées en bleues.
Cependant, nous avons constaté que les parties jouées de cette façon étaient généralement peu satisfaisantes.
En effet, la disposition aléatoire des tours ne permettait pas toujours une défense efficace contre les *mobs*, et le chemin aléatoire créé pouvait parfois être trop facile ou trop difficile à défendre.

On observe en effet que la configuration de la carte ci dessus est avantageuse car les tours sont placées à proximité chemin, ce qui leur permet de couvrir efficacement la totalité du parcours emprunté par les *mobs*. De plus, le tracé du chemin est également défavorable pour les *mobs* de part sa longueur, exposant les mobs plus longtemps aux tirs ennemis. On remarque en revanche que sur la carte de droite, les tours sont situées à une distance plus grande du chemin, ce qui réduit leur efficacité et leur couverture du parcours, rendant le jeu plus difficile pour le joueur.

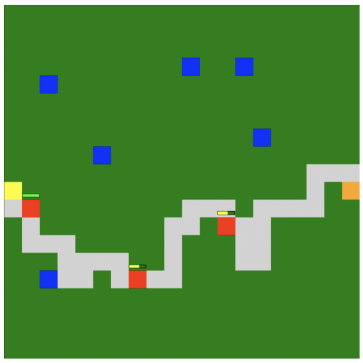

FIGURE 3 – Monde avec placement des tours aléatoires inefficaces

Suite à cette constatation, nous avons décidé de prendre une approche différente. Nous avons opté pour une carte de jeu prédéfinie, avec des emplacements de tours spécifiques à construire par

le joueur. Cette approche se rapproche davantage de celle des jeux de tower defense traditionnels, où le joueur doit réfléchir et planifier sa défense en fonction d'une carte donnée.

Cela a non seulement rendu le jeu plus stratégique, mais a également amélioré l'équilibre du jeu. Le joueur peut désormais se concentrer sur la construction de tours aux emplacements définis et la gestion de ses ressources, plutôt que de compter sur la chance pour avoir une disposition de tours favorable. Cette décision de design a grandement amélioré l'expérience de jeu, en rendant chaque partie à la fois prévisible et stimulante.

Par ailleurs, nous avons créé un système de monnaie : Le joueur démarre la partie avec une certaine quantité de monnaie, il en gagne à l'élimination de chaque mob et il peut payer pour activer des tourelles. Tous les éléments précedemment cités sont visibles sur la figure ci dessous. Le joueur est doté de 10 *Vies*, il en perd une à chaque fois qu'un *Mob* parvient à traverser le monde, la partie se termine lorsque le joueur a perdu toutes ses vies.

Dans l'illustration ci-dessous, notre interface HTML est présentée. Les tours actives sont marquées en bleu tandis que les tours inactives, qui peuvent être construites, sont en gris. Les ennemis, appelés "mobs", sont indiqués en rouge avec une barre de vie affichée pour chacun d'eux. L'entrée du parcours est représentée en jaune et la sortie en orange.

Pour démarrer la partie, l'utilisateur doit cliquer sur le bouton "play". Pour la construction des tours, il faut d'abord appuyer sur le bouton "build" et ensuite cliquer sur une tour inactive. De même, pour placer un cactus sur le chemin, il faut d'abord appuyer sur "build", puis sélectionner l'emplacement souhaité sur le chemin.

Il convient de noter que la construction d'une tour nécessite un investissement de 25 unités de monnaie virtuelle, tandis que le placement d'un cactus coûte 10 unités. Ces fonctionnalités offrent au joueur une variété d'options stratégiques tout au long de la partie.

Il est important de préciser que la partie débute initialement avec une tour déjà activée. Cela est conçu pour apporter une aide initiale au joueur en début de partie.

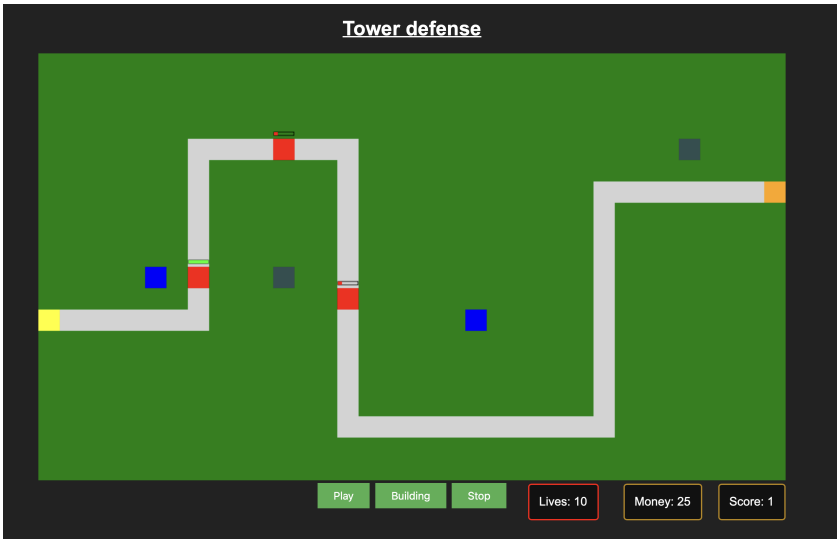

FIGURE 4 – Affichage HTML d'un monde

4.1 Développement en TypeScript

4.1.1 Couche des acteurs

Dans la couche la plus profonde, les acteurs détiennent leurs propres domaines d'action individuels et sont majoritairement caractérisés par leurs interactions (actions) avec d'autres acteurs. Ainsi, la création d'un acteur se résume à la déclaration suivante : const actor = ..., actions : ....

Chaque acteur est donc une entité distincte, pourvue de ses propres actions. Cela est avantageux dans la mesure où il suffit de spécifier des actions distinctes pour les faire évoluer, les modifier, etc.

Le contexte du jeu nous a poussés à concevoir plusieurs "catégories" d'acteurs : les monstres (mobs), les tours, les cactus (qui bloquent le passage et infligent des dégâts si on s'approche trop près). Chaque "catégorie" d'acteur détient des actions unifiées. Toutefois, l'atout réside dans le fait que si nous souhaitons, par exemple, avoir des monstres qui se déplacent lentement et d'autres qui se déplacent rapidement, il suffit de leur attribuer une action de déplacement différente. Il en va de même pour les tours, certaines possédant un rayon d'action plus ou moins étendu.

Dans le contexte de la programmation fonctionnelle, une action génère un nouvel acteur dont les caractéristiques sont déterminées par l'action elle-même. Prenons l'exemple d'une action de "tir" d'une tour, dont l'objectif est de localiser, dans son rayon d'action, un monstre à attaquer. La fonction utilise une méthode "filter", native de TypeScript, pour récupérer les acteurs se trouvant autour de la tour :

```
1  const actorsInRadius = world.actors.filter((actor: Actor) => {
2    const distance = W.distance(tower.pos, actor.pos);
3    return distance <= radius && actor.actions && actor.actions.move !==
     undefined;
4  });
```

Si un tel monstre existe, la fonction renvoie une copie du monstre qui se trouve le plus loin sur le chemin, mais dont les points de vie ont été réduits :

```
1    const actorDamaged = findActorMostToTheEnd(actorsInRadius);
2    return {
3      ...actorDamaged,
4      hp: actorDamaged.hp - damage,
5    };
```

Ainsi, l'action renvoie une intention, en l'occurrence l'intention d'infliger des dégâts au monstre se trouvant dans son cercle d'action. Toutes les actions sont construites sur le même principe : move pour les monstres, "spawnTower", "shoot", "hit" pour les cactus, etc.

4.1.2 Couche des phases

La couche des phases est quant à elle d'un niveau supérieur, car elle synthétise les intentions individuelles des acteurs pour produire une intention collective. Cela est aisé à coder puisqu'une phase se contente d'invoquer les actions des acteurs du monde, sans rien d'autre. L'avantage est qu'il y a un appel uniforme sur tous les "types" d'acteurs, alors que chaque acteur possède ses actions individuelles. On comprend mieux ici l'intérêt des actions dans le type Actor puisque si nous avions choisi de créer des fonctions externes à appliquer sur un acteur, durant une phase de jeu, nous aurions été contraints de savoir quelles actions correspondent à quels acteurs. Alors qu'ici, il suffit de les invoquer directement. Nous devons calculer une intention car les conflits surviennent lorsque tous les acteurs ont pu exécuter leurs actions. Il est donc intéressant de dissocier cette intention de la résolution effective des conflits dans le monde. Prenons l'exemple de la phase "move", la fonction effectue une opération de "map" sur les acteurs de type "monstres" (mobs) dans le monde et applique l'action "move".

```
1    return world.actors.map((actor: A.Actor) => {
2      if (actor.actions && actor.actions.move !== undefined){
3        return A.move(actor, world);
4      }
5      else{
6        return actor;
7      }
8  });
```

Cela signifie que la fonction parcourt tous les acteurs de type "mobs" dans le monde et applique l'action "move" à chaque acteur. Le résultat est un nouveau tableau d'acteurs "mobs", chacun ayant été modifié par l'application de l'action "move". Les différentes actions existantes incluent "move", "target" qui utilise l'action "shoot" prise comme exemple précédemment, "hitcactus", "spawnMob", etc. Chacune de ces actions représente une intention d'un acteur et peut être invoquée individuellement pour modifier l'état d'un acteur particulier.

4.1.3 Couche du moteur

Le moteur de jeu résout les phases, analysant ainsi le tableau d'acteurs fourni par une phase puis résolvant les conflits. C'est lui qui détient la décision finale et qui va mettre à jour le monde, le score associé, etc. Il détecte les ennemis qui atteignent l'arrivée, les ennemis qui n'ont plus de vie et qui doivent donc disparaître, etc. C'est également lui qui décide de l'ordre des phases. Pour les appeler facilement, nous avons utilisé les clés de type chaîne de caractères pour créer des tableaux de phases. Ce tableau est actualisé en fonction du score du joueur pour faire apparaître les ennemis plus ou moins rapidement, les faire bouger plus ou moins rapidement. Voici l'exemple d'un solveur de phase qui, à partir du nom de la phase, permet de résoudre et donc de renvoyer un nouveau monde cohérent où la phase a été appliquée. Cela illustre la manière dont le moteur de jeu gère les différentes phases et maintient l'état du monde cohérent et à jour.

```
1     const resolvePhaseFunction = phaseMap[phaseName];
2     const actors = phaseFunction(world);
3     if (resolvePhaseFunction){
4       return resolvePhaseFunction(actors, world);
5     }
```

Ici, phaseName représente une chaîne de caractères définissant la phase. Il est donc facile d'appeler une phase à résoudre en utilisant simplement son nom. Le moteur décide aussi de l'ordre des phases, par exemple l'odre suivant :

```
1     const basePhaseSequence: string[] = ['spawn', 'move', 'cactus', 'move',
       'cactus', 'move']
```

L'utilisation de chaînes de caractères pour représenter les phases offre une flexibilité appréciable. En effet, cela permet de modifier simplement l'ordre des phases à résoudre et donc d'ajuster ce dernier en fonction du score. C'est un moyen efficace et flexible de contrôler le déroulement du jeu.

4.1.4 Interaction des joueurs

Le jeu offre plusieurs possibilités d'interaction avec le joueur, notamment pour faire apparaître une tour ou placer un cactus. Ces interactions peuvent se faire aussi bien via la console qu'à travers une interface HTML sur une page web. Cela donne au joueur différentes options pour interagir avec le jeu, augmentant ainsi l'accessibilité et l'engagement. Nous présentons la manière dont le jeu gère les interactions avec le joueur dans deux modes différents : le mode console et le mode HTML.

Dans le mode HTML, nous récupérons les boutons de l'interface utilisateur et nous leur attachons des gestionnaires d'événements qui déclenchent les callbacks correspondants.

```
1     const playButton = document.getElementById("playButton");
2     playButton.addEventListener("click", callbacks.play);
```

Dans le mode console, nous utilisons le module Node.js readline pour lire les entrées de l'utilisateur. Ces entrées déclenchent également les callbacks correspondants. La fonction setupInteraction détermine le mode d'interaction en fonction de l'argument mode et appelle la fonction appropriée pour gérer les interactions.

```
1     export const setupInteraction = (mode: Mode, callbacks: Record<string,
      () => void>) => {...};
```

La conception du système d'interaction est fondée sur le principe de la généricité, afin de garantir l'indépendance du mode d'affichage.

Dans le mode HTML, les interactions sont gérées par l'écoute d'événements sur les éléments du Document Object Model (DOM). Par exemple, pour le bouton "play", l'événement "click" est attaché pour déclencher la fonction de rappel "play". Dans le mode console, le module Node.js readline est utilisé pour lire les entrées de l'utilisateur à partir du terminal. Par exemple, quand l'utilisateur tape une ligne, un événement 'line' est déclenché et la fonction de rappel correspondante est invoquée.

4.2 Utilisation de l'architecture de fichiers et des outils spécifiques (npm, Makefile)

L'architecture de notre projet est organisée de manière à faciliter le développement et la maintenance du code. Chaque composant du jeu est défini dans un fichier distinct, ce qui rend le code plus lisible et plus facile à gérer. Les fichiers sont structurés de manière logique, en fonction de la

fonctionnalité qu'ils implémentent. Par exemple, les définitions de type pour les acteurs, le monde et les paysages sont toutes regroupées dans des fichiers spécifiques.

Pour la gestion et l'automatisation des tâches de développement, nous utilisons deux outils puissants : npm et Makefile.

npm (Node Package Manager) est un gestionnaire de paquets pour le langage de programmation JavaScript. Il nous permet de gérer facilement les dépendances de notre projet, c'est-à-dire les bibliothèques et les modules dont notre code a besoin pour fonctionner. Grâce à npm, nous pouvons installer, mettre à jour et supprimer ces dépendances de manière simple et efficace. De plus, npm nous permet également de gérer des scripts personnalisés pour automatiser certaines tâches de développement, comme la compilation ou le test du code.

D'autre part, nous utilisons Makefile pour automatiser davantage de tâches. Un Makefile est un fichier qui spécifie un ensemble de règles pour construire une ou plusieurs cibles. Ces règles décrivent comment une cible (généralement un fichier binaire) peut être générée à partir de sources (généralement des fichiers de code source). Avec Makefile, nous pouvons facilement compiler notre projet, exécuter des tests ou générer de la documentation. Il nous suffit d'exécuter une seule commande pour effectuer ces tâches, ce qui rend le développement plus rapide et plus efficace.

Pour favoriser une collaboration efficace, nous utilisons Git, un système de contrôle de version répandu et puissant. Git nous permet de suivre et de gérer les modifications de notre code source au fil du temps. Il offre une flexibilité inégalée pour le travail en équipe, permettant à plusieurs développeurs de travailler simultanément sur des parties différentes du projet sans conflit. Les modifications peuvent être révisées, fusionnées et annulées si nécessaire, ce qui augmente la fiabilité et la traçabilité de notre code.

Le langage de programmation utilisé pour notre projet est TypeScript. TypeScript est un sur-ensemble de JavaScript qui ajoute des types statiques optionnels. Cela améliore la lisibilité et la robustesse du code en permettant la détection précoce des erreurs. En outre, TypeScript offre une meilleure autocomplétion et une meilleure assistance à la navigation dans le code, ce qui augmente la productivité des développeurs. Pour que notre code TypeScript puisse être exécuté dans un navigateur ou dans tout autre environnement JavaScript, il doit d'abord être transcompilé en JavaScript. Cette opération est réalisée par le compilateur TypeScript (TSC).

Enfin, pour assurer la qualité de notre code, nous avons utilisé ESLint, un linter populaire pour JavaScript et TypeScript. ESLint vérifie notre code pour détecter les erreurs de syntaxe, les bugs potentiels et les violations des conventions de codage que nous avons définies. Cela nous aide à maintenir un style de code cohérent et à prévenir les erreurs avant qu'elles ne se produisent.

En combinant ces outils et méthodes, nous avons pu mettre en place un environnement de développement solide, favorisant l'efficacité, la qualité du code, et facilitant la collaboration au sein de notre équipe.

4.3 Organisation du travail

Au début du projet, nous avons effectué une analyse approfondie des exigences et des fonctionnalités du jeu. Ensuite, nous avons divisé les différentes fonctionnalités en tâches plus petites et identifié les dépendances entre elles. Pour ensuite hierarchiser ces tâches selon leur importance et classer leur ordre d'execution dans une *TODO-LIST*.

Quant à la répartition du travail entre les différents membres du groupe, nous avons opté pour une approche collaborative basée sur la technique du pair programming. (*Le pair programming est une collaboration simultanée de deux programmeurs sur un même code*). Cette méthode de travail nous a permis de répartir efficacement les tâches et d'encourager une forte collaboration entre les membres du groupe, car nous avons pu partager nos connaissances et nos idées, et les erreurs ont donc été détectées et corrigées rapidement.

Ce projet a donc été l'occasion pour nous de découvrir des techniques de developpement logiciel utilisées dans le monde du travail. Pour assurer la cohérence et la propreté du code, nous nous sommmes accordés pour utiliser la convention de codage *Camelcase*, et d'utiliser l'anglais pour tous nos noms de variables, structures etc...

4.4 Arborescence de fichiers

4.4.1 Les différents fichiers du projet

— Le repetertoire *dist/* contient les fichiers *javascipt*, qui sont crées à la compilation des fichiers *typescript* du repertoire *src/*.
— Dans *tst/* son codés tous les tests des fonctions du jeu.

— Le repertoire *src/* contient tous les fichiers sources en *typescript* du jeu, visibles sur la **Figure 5**.

4.4.2 Dépendances entre ces fichiers

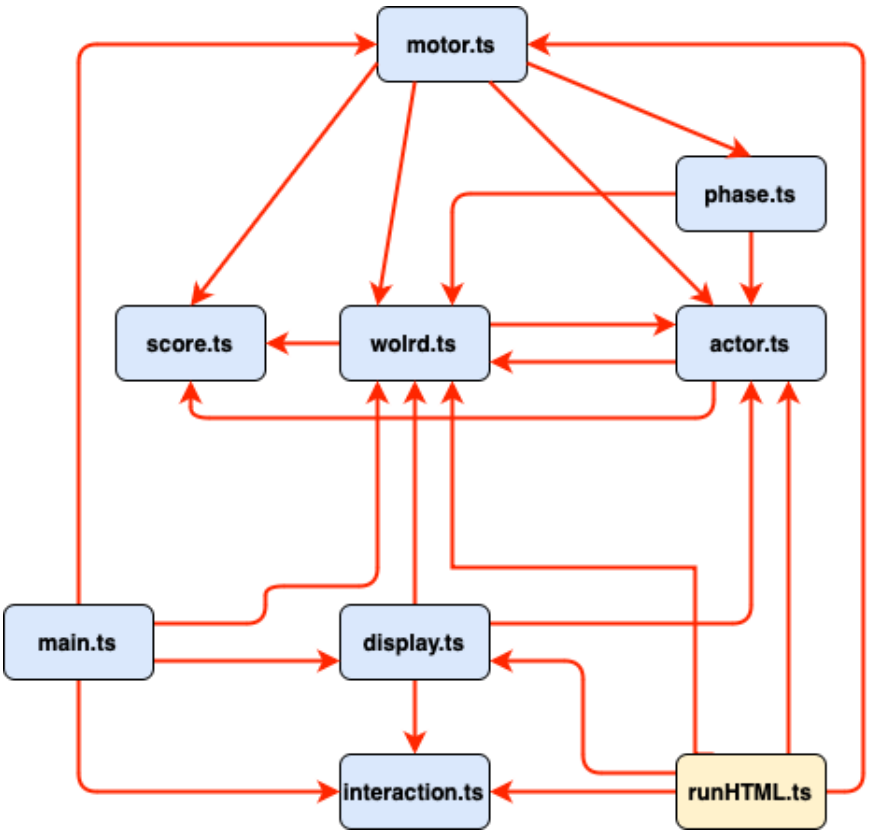

FIGURE 5 – Graphe de dépendances entre les fichiers du repertoire *src/*

5 Tests et validation

Dans cette section, nous aborderons l'aspect crucial de tout projet de développement logiciel : la validation et les tests. Une fois les fonctionnalités implémentées, il est essentiel de s'assurer que le système fonctionne comme prévu. Les tests sont un moyen incontournable de vérifier la qualité du code, de se prévenir de bugs et de garantir la fiabilité du logiciel. Ils permettent également de s'assurer que le logiciel continue de fonctionner correctement à mesure que des modifications sont apportées, en identifiant rapidement les régressions potentielles. Nous discuterons ici des différentes méthodes de test que nous avons utilisées, des outils que nous avons employés pour faciliter cette tâche, et de la manière dont ces tests ont contribué à la robustesse et à la fiabilité de notre jeu de Tower Defense.

5.1 Utilisation de Jest et description des tests

Dans notre projet, nous avons utilisé Jest pour la gestion des tests unitaires. Jest est un framework de tests en JavaScript développé qui offre une grande flexibilité et facilite l'écriture de scénarios de tests.

Chaque fichier TypeScript possède son propre fichier de test respectant le nommage fichier.test.ts. Pour exécuter ces tests, nous utilisons la commande make test qui est définie dans notre Makefile. Cette commande déclenche la commande npm test qui à son tour lance Jest pour exécuter tous les tests définis dans notre projet.

5.1.1 Tests unitaires

Nous avons réalisé principalement des tests unitaires dans le but de valider nos fonctions une par une. Pour chaque test, nous avons essayé de respecter les scénarios suivants : tests des cas

erreurs (throw new Error), vérifier que la fonction retourne bien ce qui est demandé et également des tests sur des données non prévues ou plus complexes.

Prenons par exemple les tests pour la fonction move. Cette fonction est responsable du déplacement d'un acteur dans le monde du jeu. Pour vérifier son bon fonctionnement, nous avons écrit plusieurs scénarios de tests.

Dans le premier scénario, nous testons si un acteur se déplace vers le chemin le plus proche. Nous créons un acteur en position (2,2) dans un monde contenant une série de paysages formant un chemin. Lorsque nous appelons la fonction move avec cet acteur et ce monde, nous nous attendons à ce que la nouvelle position de l'acteur soit (3,2), qui est le point le plus proche sur le chemin.

Le deuxième scénario teste le comportement de la fonction move lorsque l'acteur est sur la position de sortie. Nous créons un acteur sur la position de sortie (4,4) et nous nous attendons à ce que sa position reste inchangée après l'appel à la fonction move.

Le troisième scénario teste le cas où un autre acteur occupe déjà la position vers laquelle l'acteur essaye de se déplacer. Dans ce cas, nous créons un monde où un autre acteur est déjà présent sur la position (3,2). Lorsque nous appelons la fonction move avec notre acteur initial et ce monde, nous nous attendons à ce que la position de notre acteur reste inchangée.

Ces tests garantissent que la fonction move fonctionne correctement dans diverses situations, ce qui assure la robustesse de notre logique de jeu.

Durant notre travail, nous avons eu à plusieurs moments des tests qui n'étaient pas validés, ce qui nous a permis de corriger des erreurs rapidement.

6 Conclusion

Au cours de ce projet, notre tâche était de concevoir et de développer un jeu de "Tower Defense", en mettant à profit la programmation fonctionnelle et en utilisant le langage TypeScript. Notre principal objectif était de créer un jeu fonctionnel et engageant à partir de cette approche de programmation.

Pour atteindre cet objectif, nous avons adopté une architecture modulaire, en définissant explicitement différents types et interfaces pour représenter les divers éléments constitutifs du jeu, tels que le monde, les acteurs et les paysages. Nous avons également structuré notre jeu en plusieurs étapes ou phases (actions individuelles, phases de jeu, moteur de jeu), permettant ainsi une décomposition claire de notre objectif initial.

Cette approche a non seulement rendu notre code plus lisible et compréhensible, mais a aussi facilité les éventuelles modifications et extensions futures du jeu. En effet, grâce à cette structure bien définie, l'intégration de nouvelles fonctionnalités ou modifications peut être réalisée de manière plus fluide et organisée.

Tout au long de ce projet, nous avons constamment visé à maintenir une approche générique, ce qui nous a parfois confrontés à certaines difficultés. En particulier, la fonction de rendu, qui avait pour but d'afficher une représentation visuelle de notre jeu sur n'importe quel système (que ce soit la console ou le HTML), a présenté des défis particuliers. Néanmoins, nous pensons avoir réussi à préserver cette généricité dans presque toutes les fonctions du jeu.

6.1 Limites et potentiel d'évolution du projet

Une possible évolution de ce projet pourrait être l'introduction de nouvelles phases de jeu, telles que l'ajout de la capacité de soin (heal), ainsi que l'introduction de nouveaux acteurs dans l'univers du jeu. Bien que nous n'ayons pas eu le temps de concrétiser ces extensions durant le délai imparti pour le projet, la structure mise en place le permet aisément.

Nous avons rencontré quelques difficultés concernant l'heuristique mise en œuvre pour les paysages de type "chemin" (path). Cette heuristique pourrait être optimisée pour permettre des déplacements plus cohérents sur des chemins de grande largeur. Ceci met en lumière le fait que, même si notre jeu est fonctionnel et jouable en l'état, il y a toujours place à l'amélioration et à l'expansion.

6.2 Apports du projet en termes d'apprentissage et d'expérience

Ce projet a été une occasion précieuse d'apprentissage et de développement de nouvelles compétences. En premier lieu, il nous a permis d'approfondir notre compréhension du langage TypeScript et de ses applications. Il a également renforcé notre maîtrise de la programmation fonctionnelle, pratique à laquelle nous n'étions pas habitués et qui présente pourtant ses avantages.

La création de ce jeu nous a également permis de comprendre l'importance de la conception préalable et de la planification. La définition claire des différentes structures et types de données a été essentielle pour la gestion efficace de la complexité du code. Cette expérience a renforcé notre appréciation de la valeur d'une architecture modulaire et bien définie.

Par ailleurs, ce projet nous a permis d'approfondir notre connaissance des tests logiciels. L'utilisation de la bibliothèque Jest pour la création de tests unitaires, fonctionnels et d'intégration a été une étape cruciale pour garantir la qualité du code et la bonne fonctionnalité du jeu.

Enfin, l'utilisation d'outils tels que Git pour la gestion des versions, ainsi que npm et Makefile pour la gestion des dépendances et l'automatisation des tâches, a été une expérience enrichissante. Ces outils sont très utilisés dans le monde du développement logiciel, et cette expérience pratique nous a donné une meilleure compréhension de leur utilité.

En somme, ce projet nous a permis d'acquérir des compétences importantes en matière de développement logiciel, d'apprendre à travailler efficacement en équipe et de nous familiariser avec des outils et des pratiques industriels courants.