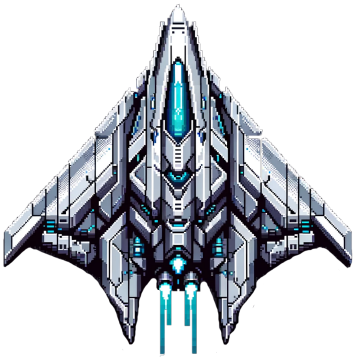
Rapport CPA :
Refonte d'une application de jeu vidéo
Space Conquest

Ewen GLASZIOU (21312544) Julien LE GOFF (21304302)

01 mai 2024



Enseignant : Bình Minh Bùi Xuân

Chargé de TME : Arthur Escriou

M1 STL 2023-2024

Table des matières

1 Introduction

1.1 Membres de l'équipe

— Ewen Glasziou - Chef de projet, Développeur
— Julien Le Goff - Front-end, Développeur

1.2 Objectif du projet

Le projet vise à développer un jeu en 2D avec au moins un aspect complexe, comme par exemples :

— de la physique (collision, gravité)
— de la génération aléatoire (création de niveau aléatoire : labyrinthe, plateforme, ennemi)
— du pathfinding (des éléments de jeu utilisant un algo de pathfinding : Dijkstra, A*, D*)

Notre choix pour *Space Conquest* s'est tourné vers un jeu de stratégie solo en temps réel en 2D (RTS) où le joueur doit comme son nom l'indique stratégiquement capturer avec des vaisseaux spatiaux toutes les planètes d'un système planétaire avant les IA adverses. Ce jeu combine des éléments de tactique, de gestion de ressources et de rapidité d'exécution.

Notre projet nécessite des collisions entre les différents vaisseaux mais également avec les planètes et les obstacles, en plus d'utiliser le pathfinding pour la gestion du déplacement des troupes. Nous pouvons donc dire que nous respectons le cahier des charges avec cette proposition de jeu.

1.3 Analyse fonctionnelle

User Stories :

— En tant que joueur, je souhaite avoir des vaisseaux au fur et à mesure de la partie.
— En tant que joueur, je veux sélectionner mes vaisseaux et les déplacer.
— En tant que joueur, je souhaite attaquer d'autres planètes pour étendre mon territoire.
— En tant que joueur, je veux voir une carte du système planètaire pour planifier mes mouvements stratégiques.

1.4 Illustration avec un wireframe

Le wireframe du jeu a été créé avec l'outil Figma, qui a permis de définir clairement l'interface utilisateur et les interactions principales avant la phase de développement.

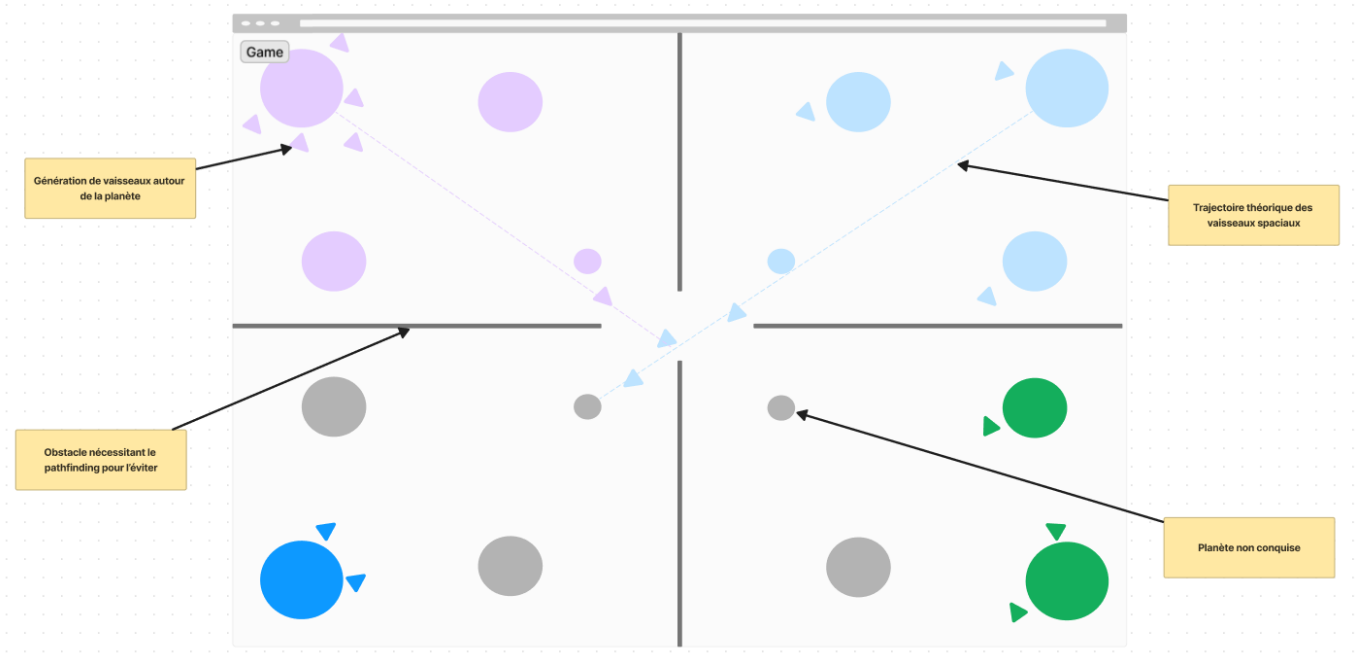


FIGURE 1 – Wireframe de notre jeu avec quelques détails.

Nous retrouvons ainsi la possibilité de diriger des vaisseaux (après une sélection), la gestions des collisions, non seulement entre les vaisseaux eux-mêmes, mais aussi avec les planètes et les obstacles, et la génération de troupes autour des planètes. De plus, la présence d'obstacle nécessite que les intelligences artificielles soient capables de les détecter et de les éviter efficacement.

1.5 Technologies utilisées et justification

Le jeu a été développé en utilisant les technologies suivantes :

— **HTML5 / CSS3 / React** pour le développement frontal.

— **TypeScript** comme moteur de jeu pour gérer le rendu graphique et les interactions.

Justification : L'utilisation de React couplé à TypeScript nous a permis de réutiliser le template du TME7 offrant une certaine robustesse dans son utilisation. Ces technologies sont par ailleurs bien documentés, ce qui réduit le temps d'apprentissage et permet une mise en œuvre rapide de prototypes fonctionnels.

Voici par ailleurs un comparatif des différentes technologies utilisées pour faire des jeux 2D dans le même type que celui demandé :

Technologie	Portabilité	Performance	Contrôle	Bibliothèques	Apprentissage	Documentation	Taille
Java	X	X		X			
React	X	X		X		X	
PyGame	X			X	X	X	
C++		X	X	X			
JavaScript	X			X	X	X	
Vue	X	X		X			X

On peut voir sur ce tableau comparatif que plusieurs technologies sortent du lot tels que React, PyGame, JavaScript et Vue.

Nous avions utilisé au départ une version from scratch intégralement faite en JavaScript mais suite aux conseilles de M. Escriou et pour gagner en performances nous avons réutilisé son

template React en le passant en MVC, tout ceci dans le but d'optimiser les différents calculs et affichages. En effet React bénéficie d'une gestion efficace du Virtual DOM en plus des autres avantages déjà cités, ce qui contribue à des performances fluides, même pour des applications graphiquement riches comme les jeux, ce qui correspond bien à notre cas avec les nombreux vaisseaux spatiaux.

1.6 Bilan de ressources déployées pour le projet

Le tableau suivant résume le nombre de jours-hommes par tâche principale du projet :

Tâche	**Jours-hommes**
Conception initiale	4
Création des wireframes	1
Développement de la logique de jeu	15
Intégration des graphiques	7
Tests et ajustements	10
Préparation du rapport	2
Préparation de la présentation finale (vidéo)	1

Ce tableau donne une vue d'ensemble de l'effort de travail réparti sur les différentes phases du projet. Elle permet notamment de visualiser l'allocation des ressources humaines tout au long du développement du jeu.

2 Partie technique

2.1 Architecture générale

Pour ce projet, l'architecture Modèle-Vue-Contrôleur (MVC) a été mise en œuvre afin de favoriser une gestion efficace du code par une séparation claire des responsabilités. Cette architecture divise le système en trois composants principaux : le Modèle qui gère les données et la logique métier, la Vue qui s'occupe de l'affichage de ces données à l'utilisateur, et le Contrôleur qui agit comme intermédiaire entre le Modèle et la Vue en gérant les interactions utilisateurs et en déclenchant des mises à jour du modèle et des rafraîchissements de la vue. Cette séparation permet de faciliter les modifications et les extensions du système, en particulier pour l'ajout rapide et efficace de nouvelles représentations graphiques, telles que les sprites.

2.1.1 Modèle

Le modèle contient les données de l'application, la logique et les règles du système, dans notre projet, il est divisé en 5 fichiers :

— **model.ts :** Gère les données et les états des planètes, vaisseaux spatiaux, et astéroïdes. Il inclut également la logique de sélection des vaisseaux dans une zone définie.
— **collision.ts :** Pour la gestion des collisions des vaisseaux entre eux et avec les obstacles.
— **eventFunctions.ts :** Définit les actions et interactions possibles pour les utilisateurs, transformant les entrées en événements traitables.
— **pathFinding.ts :** Implémente l'algorithme de recherche de chemin pour naviguer efficacement dans l'environnement de jeu et ses fonctions auxiliaires.

— **ia.ts :** Gère la logique des intelligences artificielles ennemies, définissant leur comportement en fonction du contexte de jeu.

2.1.2 Vue

La vue gère la représentation visuelle des données du modèle :

— **view.ts :** Ce fichier est responsable de l'affichage des éléments graphiques sur l'interface utilisateur. Il actualise la vue en réponse aux modifications des données du modèle, garantissant une synchronisation en temps réel avec l'état du jeu.

2.1.3 Contrôleur

Le contrôleur fait le lien entre l'utilisateur et le système :

— **controller.tsx :** Orchestrer les interactions entre la vue et le modèle. Ce fichier traite les entrées de l'utilisateur, invoque les mises à jour sur le modèle et actualise la vue en conséquence. Il joue un rôle crucial dans la gestion des événements et la délégation des tâches spécifiques comme la sélection et la génération de nouvelles entités.

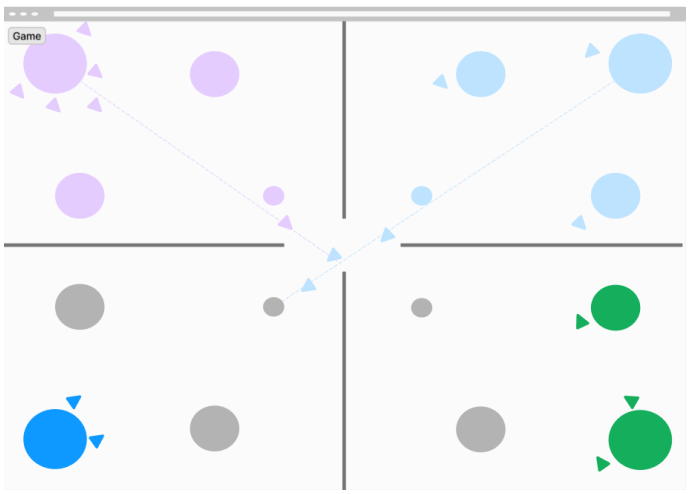
Cette architecture MVC a facilité une division claire et fonctionnelle des éléments du projet, optimisant ainsi la gestion du développement et la maintenance. Elle permet non seulement une meilleure modularité mais aussi une plus grande flexibilité pour les mises à jour futures et l'intégration de nouvelles fonctionnalités, minimisant les interférences entre les différents composants du système. Cette organisation structurelle a donc été essentielle pour accroître l'efficacité du développement et pour soutenir l'évolution continue du projet sans perturbation majeure des fonctionnalités existantes.

2.2 Interfaces

Nous avons dans notre jeu un total de deux vues :

— Celle principale qui permet de jouer.

— Celle de fin affichant si l'on a perdu ou gagné et qui permet de rejouer.



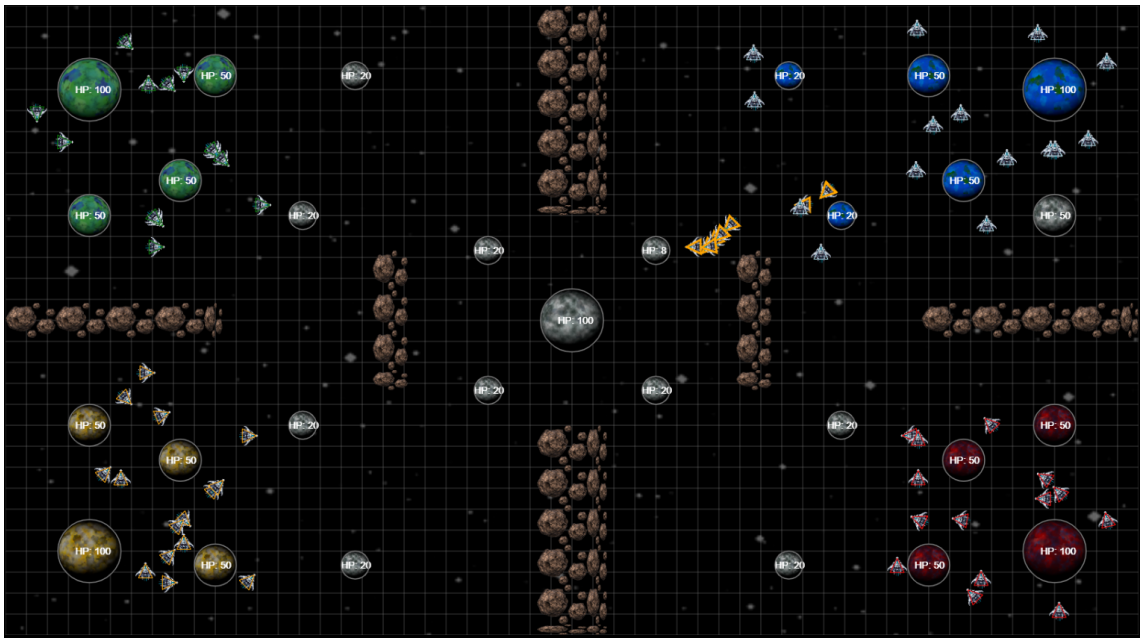


FIGURE 2 – Comparatif entre la maquette et la version finale

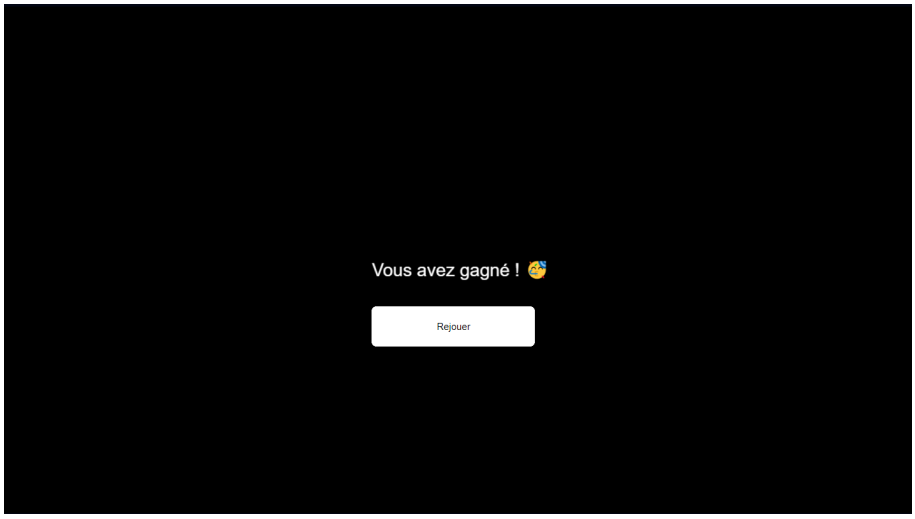


FIGURE 3 – Écran de victoire

2.3 Algorithmes

Concernant les algorithmes de notre jeu, nous les avons séparés en quatre catégories principales :

- **Collision :** Gestion des interactions physiques entre les entités du jeu.
- **Pathfinding :** Calcul des trajectoires optimales pour les déplacements des unités.
- **IA ennemis :** Stratégies de comportement pour les ennemis contrôlés par l'ordinateur.
- **Gestion du jeu** : algorithme utile au fonctionnement du jeu ou pour des fonctionnalités supplémentaire.

2.3.1 Gestion des collisions

La gestion des collisions dans notre jeu est essentielle pour assurer une interaction réaliste et fluide entre les différentes entités du jeu telles que les unités (triangles) et les obstacles (rectangles et cercles). Le système emploie des méthodes variées pour détecter les collisions, en utilisant des algorithmes géométriques pour optimiser les performances et maintenir l'équité du jeu.

Collision entre triangles et cercles La fonction *checkCollisionWithCircle* permet de détecter les collisions entre un triangle et un cercle. Cette vérification se fait en deux étapes : d'abord, une vérification ponctuelle pour chaque sommet du triangle par rapport au cercle, et ensuite, une vérification des segments du triangle avec le cercle. Cette méthode garantit que toutes les interactions possibles sont prises en compte, que ce soit par un point ou un segment touchant le cercle.

Collision entre triangles La fonction *checkCollisionWithTriangle* est utilisée pour détecter les collisions entre deux triangles. Cette fonction vérifie si un point de l'un des triangles est à l'intérieur de l'autre triangle en utilisant la fonction *isPointInsideTriangle*, ce qui permet de détecter les collisions même quand les triangles ne se chevauchent que partiellement.

Collision avec des obstacles rectangulaires Pour les collisions avec des rectangles, qui peuvent représenter des murs ou d'autres obstacles fixes, la fonction *checkCollisionWithRectangle* est mise en œuvre. Elle examine les intersections entre les arêtes du triangle et les arêtes du rectangle, ainsi que la présence de points du triangle à l'intérieur du rectangle. Cette approche est cruciale pour les environnements contenant divers obstacles dispersés sur la carte.

Collisions avec les bords du jeu Enfin, la fonction *checkCollisionWithBorders* est utilisée pour empêcher les triangles de sortir des limites du jeu. Cette fonction vérifie si les points d'un triangle dépassent les dimensions du canvas, ce qui est particulièrement important pour maintenir le jeu dans son espace défini.

En cas de détection de collision, les entités impliquées subissent un rebond, simulant un impact en redirigeant l'entité dans la direction opposée à celle de l'impact. Ce mécanisme est essentiel pour renforcer l'aspect tactique du jeu, où le positionnement et le mouvement sont clés.

2.3.2 Pathfinding

Le système de pathfinding de notre jeu est conçu pour permettre une navigation efficace et réaliste des unités sur le terrain de jeu. La grille utilisée pour le pathfinding reflète l'espace jouable, où chaque cellule représente un possible point de passage, chaque obstacle modifiant la praticabilité des cellules correspondantes.

Choix de la taille des cellules La taille des cellules de la grille a été définie de manière à ce qu'un vaisseau puisse être contenue dans une cellule. Ce choix simplifie grandement la gestion des collisions, car il évite le problème de devoir vérifier plusieurs petites cellules sous-jacentes qu'un vaisseau pourrait chevaucher simultanément (surtout pour les mouvement en diagonal nécessitant que les cellules dans la diagonales opposé sois aussi praticable). En optant pour des cellules plus grandes, nous réduisons la complexité algorithmique et améliorons les performances, surtout compte tenu du nombre élevé de vaisseaux qui peuvent être présents dans *Space Conquest*.

Création de la grille La fonction *createGrid* initialise dans un premier temps, la grille en fonction des dimensions du canvas. Chaque cellule de la grille est initialement marquée comme praticable. Puis dans un second temps, les obstacles qui correspondent aux murs(rectangulaires) et aux planètes(circulaires) sont traités en ajustant la praticabilité des cellules concernées pour refléter les zones non traversables.

Intégration des obstacles Les obstacles sont intégrés dans la grille en marquant les cellules correspondantes comme non praticables. Pour les rectangles, cela est fait en parcourant chaque cellule qui tombe à l'intérieur des coordonnées de l'obstacle. Pour les cercles, le processus est similaire, mais avec une vérification supplémentaire pour s'assurer la cellule n'est pas en contacte avec le cercle.

Algorithme A* L'algorithme de recherche de chemin A* est implémenté dans la fonction *findPath*, qui utilise une grille clonée pour trouver le chemin le moins coûteux d'un point à un autre tout en permettant des modifications spécifiques de praticabilité sans impacter d'autres processus ou trajets. Ce clonage est essentiel pour traiter les cases de manière indépendante lors de chaque recherche de chemin, en particulier lorsque les configurations des obstacles varient entre les trajectoires des différents joueurs ou unités. L'algorithme commence par la cellule de départ, ajoutant les cellules voisines à l'ensemble ouvert si elles sont praticables et n'ont pas déjà été traitées. Chaque cellule garde une trace de son coût de déplacement et de son prédécesseur, permettant de reconstruire le chemin une fois la destination atteinte.

Algorithme 1 : Algorithme A* adapté pour la navigation dans le jeu

Entrée : Grille originale : *grid_original*, point de départ : *start*, cible : *target*, booléen : *isPlayer*
Sortie : Liste de points formant le chemin ou vide si aucun chemin n'est trouvé

```
1  Function A*
2      grid ← cloneGrid(grid_original);
3      Ajuster la praticabilité de la cible si le chemin est pour le joueur, on permet que la
         cible puisse être un mur;
4      Définir la zone correspondant à la cible comme praticable pour la navigation (la
         cible est une planète);
5      openSet ← Créer une liste contenant la cellule de départ (cellules à traitées);
6      closedSet ← Créer un ensemble vide (contiendra les cellules déjà traitées);
7      Initialiser le coût de la cellule de départ à 0;
8      tant que openSet n'est pas vide faire
9          current ← cellule dans openSet avec le coût le plus bas;
10         si current est la cellule de la cible alors
11             retourner reconstructPath(current, target);
12         fin
13         Retirer current de openSet et ajouter à closedSet;
14         neighbors ← getNeighbors(current, grid);
15         pour chaque neighbor dans neighbors faire
16             si neighbor n'est pas praticable ou déjà dans closedSet alors
17                 continuer;
18             fin
19             Calculer le coût temporaire pour neighbor;
20             si neighbor n'est pas dans openSet ou le coût temporaire est inférieur au
                 coût de neighbor alors
21                 Mettre à jour le coût de neighbor;
22                 Définir current comme parent de neighbor;
23                 Ajouter neighbor à openSet si ce n'est pas déjà fait;
24             fin
25         fin
26     fin
27     retourner liste vide (échec de trouver un chemin);
```

Complexité de l'algorithme La complexité temporelle de notre implémentation de l'algorithme A* dépend principalement du nombre de cellules dans la grille et de l'efficacité avec laquelle les voisins sont évalués. Typiquement, la complexité est $O(n \log n)$, où n est le nombre de cellules dans la grille. Cependant, dans notre approche spécifique, où nous adaptons la praticabilité des cellules autour de la cible pour chaque recherche de chemin et clonons la grille pour chaque invocation pour éviter des effets secondaires sur l'état global du jeu, la complexité peut être affectée de manière significative.

En particulier, l'ajustement de la praticabilité autour de la cible, qui implique le calcul de la distance entre chaque cellule et la cible pour déterminer si elle doit être rendue praticable ((n) vérifications), augmente le nombre d'opérations nécessaires à chaque recherche de chemin. De plus, la nécessité de cloner la grille pour chaque recherche de chemin ajoute une surcharge mémoire et computationnelle non négligeable.

En termes de vérification des voisins, la prise en compte des mouvements diagonaux et

la vérification que les cellules adjacentes sont praticables avant de permettre un déplacement diagonal augmentent également la complexité. Chaque cellule nécessite une vérification de praticabilité supplémentaire qui, dans les grilles densément peuplées d'obstacles, peut sensiblement ralentir l'exécution de l'algorithme.

En conséquence, bien que la complexité générale reste $O(n \log n)$ dans le cas idéal, ces facteurs spécifiques à notre implémentation peuvent conduire à des performances inférieures dans des scénarios avec de nombreux obstacles ou de grandes grilles. Cependant bien que la modification de praticabilité soit un coût supplémentaire pour chaque recherche, celui-ci ne varie pas et reste le même pour chaque nouvelle recherche de chemin .

Reconstruction du chemin Après avoir atteint la cellule cible, le chemin est reconstruit à partir de la cellule de fin en remontant à travers les cellules parentes jusqu'à la cellule de départ. La fonction *reconstructPath* est utilisée pour ce processus, garantissant que le chemin final reflète le parcours optimal déterminé par l'algorithme A*. Pour simplifier le chemin, seules les positions des changements de direction sont conservées : si deux cellules consécutives partagent la même orientation (soit horizontale, soit verticale), une seule position est conservée pour cette ligne droite. Cette optimisation réduit le nombre de points dans le chemin final, facilitant ainsi le suivi par les unités sans perdre de précision dans le mouvement.

Cette méthode de reconstruction du chemin analyse les changements de direction entre les cellules pour ne conserver que les points nécessaires au tracé du chemin. Cela permet de réduire la complexité des données de chemin et d'optimiser la navigation des unités dans le jeu (évite des lags du calcule destinations consécutif).

2.3.3 IA ennemis

L'intelligence artificielle des ennemis dans notre jeu est conçue pour offrir une certaine diffculté au joueur, leurs actions s'adaptent en temps réel au fur et à mesure de la partie. Les comportements des ennemis sont gérés par des algorithmes qui évaluent les situations et adaptent une stratégie en prenant des décisions basées sur les configurations de jeu actuelles. Les stratégies implémentées permettent aux IA d'attaquer différentes cibles ennemies en fonction de la partie. De plus, les IA peuvent cibler des planètes non occupées pour étendre leur influence ou diviser leurs forces entre plusieurs cibles, selon le nombre de planètes qu'ils contrôlent déjà.

Les versions précédentes de notre IA incluaient des comportements où l'ennemi ciblait spécifiquement le joueur pour augmenter la difficulté, mais cette stratégie a été modifiée pour rendre le niveau actuel plus accessible. Actuellement, l'IA tend à privilégier l'occupation des planètes inoccupées avant d'attaquer les planètes ennemies, ajustant ses tactiques en fonction du contexte stratégique global du jeu.

Exemple de stratégie de l'IA actuel

- Si le nombre de planètes contrôlées est inférieur à quatre, l'IA cible la planète non habitée la plus proche pour étendre rapidement son territoire.
- Si plus de quatre planètes sont contrôlées, l'IA peut choisir de diviser ses forces entre la planète non habitée la plus proche et la planète ennemie la plus faible et la plus proche, utilisant des calculs pour optimiser l'efficacité de l'attaque basée sur la distance et la santé de la cible.

Ces comportements sont programmés pour s'adapter dynamiquement aux changements dans le jeu, assurant que les ennemis restent un défi compétitif et pertinent tout au long de la partie.

Les algorithmes prennent en compte la distribution géographique des planètes, la santé des cibles potentielles, et la disposition des forces du joueur pour formuler une réponse tactique adéquate.

2.3.4 Gestion du jeu

— **Gestion du mouvement des troupes**

La gestion du mouvement des troupes est cruciale pour la stratégie et la tactique du jeu. Nous utilisons une série de fonctions dédiées à la gestion du déplacement, de l'orientation, et de la réaction aux collisions des unités, spécifiquement les vaisseaux spatiaux sous forme de triangles.

1. **Réorientation et mouvement**
   Les vaisseaux sont continuellement réorientés pour pointer vers leur destination grâce à la fonction *reorientTriangle*. Cette fonction calcule l'angle nécessaire pour aligner chaque vaisseau avec sa trajectoire cible et applique une rotation limitée pour assurer un changement progressif de direction.
2. **Gestion des collisions**
   Pour éviter que les vaisseaux ne se superposent ou n'interagissent de manière non réaliste avec l'environnement, nous avons implémenté une série de fonctions de collision (explicité précédemment) et de rebond. Dès qu'une collision est détectée avec une planète, ou un obstacle (murs et bordures), le vaisseau concerné est immédiatement repositionné pour refléter un rebond réaliste, évitant ainsi les blocages ou les superpositions non désirées.
3. **Coordination du mouvement**
   Le mouvement des troupes est géré par la fonction *moveOrTurnTriangles*, qui orchestre le mouvement global des vaisseaux en prenant en compte leur destination actuelle, les collisions potentielles et les ajustements nécessaires en termes de direction et de vitesse.

— **Boids**
[Définition des boids] : Les boids sont utilisés pour créer des représentations synthétiques réalistes de nuées d'oiseaux, bancs de poissons, essaims d'insectes ou autres regroupements d'animaux.

L'application des boids aux déplacements de nos vaisseaux spatiaux semble pertinente selon cette définition. En effet, plutôt que d'être trop proches les uns des autres, un certain espacement entre les vaisseaux permet de mieux rendre compte de leur nombre et d'ajouter du réalisme.

Ainsi, nous avons mis en place une implémentation des boids dans le fichier model.ts afin d'améliorer la cohérence des déplacements de nos vaisseaux et d'éviter les superpositions. Bien que fonctionnelle, cette implémentation présente un problème : l'interaction et la force d'éloignement entre les vaisseaux, peu propulser dans certains cas des vaisseaux à l'intérieur des planètes ou des murs, les rendant alors coincés dans l'objet qui est considéré comme non praticable. Étant donné que nous n'avons pas réussi à résoudre ce problème pour le moment, leurs force et espacement (disponible dans le fichier config.ts) sont donc réglés à 0, ce qui fait que cette fonctionnalité est temporairement désactivée.

les vaisseaux peuvent se retrouver coincés dans les murs suite à des interactions imprévues (des problèmes de timing à cause de l'affichage des sprites en serait la cause)

FIGURE 4 – Représentation de notre boids actif

3 Conclusion et retour d'expérience

Nos objectifs pour ce projet sont atteints en grande partie. En effet nous avons bien une vue permettant de sélectionner et déplacer les vaisseaux qui sont générés autour de nos planètes conquises le tout avec des sprites, un algorithme permet de gérer le pathfinding dudit déplacement en évitant des obstacles comme les astéroïdes ou les planètes non ciblées. Il est possible de conquérir de nouvelles planètes qui sont soit inoccupées soit conquises par un ennemi. Il y a un total de 3 IA ennemis avec des comportements modulables en fonction de l'algorithme choisi, qui ont la capacité de conquérir des planètes. Et pour finir, nous avons la possibilité en fin de partie de rejouer.

Les améliorations pour ce projet sont de plusieurs ordres, dans un premier temps l'ajout d'une gestion du nombre de vaisseaux par planètes ou par joueur pour empêcher des lags sur les ordinateurs plus anciens et ainsi améliorer les performances. L'ajout de nouveaux niveaux avec un menu de sélection pour varier la difficulté des niveaux. Améliorer les boids déjà présents pour le mouvement des troupes en réglant le souci de collision et en rendant leurs mouvements plus naturels.

Nous avons également diverses idées de gameplay que nous avons dû mettre de côté faute de temps :

- **Des trous noirs** permettant la téléportation des troupes d'un côté à l'autre de la carte (complexifie le calcul pour le pathfinding).
- **Un soleil ou étoile** avec une gravité attirant et détruisant les vaisseaux se rapprochant de trop près de celui-ci (la complexité d'implémentation viendrait de la gravité (force d'attraction) et de la modification de l'algorithme de pathfinding. En effet, si celui-ci est trop efficace, les ia ne seront jamais affectés par la gravité car elles éviteront de passer trop près du soleil ou à l'inverse elles pourraient ne pas être capable de passer assez loin du soleil).
- **Des IA plus complexes** que celles implémentées avec difficulté réglable (ciblage de plus de 3 cibles possible, capacité à réagir à plus de situations telles que le nombre de planètes du joueur, la capacité de revenir défendre une planète en danger, cibler des planètes clés en priorité, etc...).
- La possibilité de mettre pause avec une touche (car actuellement si votre mère vous appelle c'est perdu).
- Permettre la sélection des vaisseaux du joueur à proximité de la planète sélectionnée (pour offrir au joueur plus de facilité à attaquer plusieurs planètes en même temps).
- Touche pour automatiser le ciblage des troupes générées par une planète vers une autre planète cible (besoin potentiellement de l'implémentation précédente).

— Rajouter des types de planètes différentes ou avec des effets bonus (augmentation de la vitesse des troupes, augmentation de la vitesse de productions des troupes, augmentation de la vitesse de régénération des planètes, etc...).
— Facilité la défense des planètes pour le joueur et les IA. Actuellement défendre une planète est compliqué, il faut placer ces troupes sur le chemin des attaquants. Donc possiblement ajouter une fonctionnalité pour que les troupes prennent des positions défensives (en arc de cercle, en ligne, etc...).

4 Utilisation du jeu et Contrôles

4.1 Lancement du jeu

Pour lancer le jeu il faut simplement se situer dans le répertoire "Version_fonctionnelle", taper les commandes :

```
yarn install
yarn start
```

Le jeu se lance ensuite dans le navigateur à l'adresse :

```
http://localhost:3000/
```

4.2 Contrôles du jeu

— Pour sélectionner un nombre limité de vaisseaux spatiaux de votre camp, vous pouvez faire une sélection en maintenant appuyé clic droit sur la zone de sélection puis relâcher le clic.
— Pour sélectionner tous vos vaisseaux vous pouvez double-cliquer n'importe où sur l'écran.
— Pour diriger les vaisseaux vous pouvez choisir un endroit sur la carte et faire clic gauche pour les y envoyer (les vaisseaux restes par défaut sélectionnés jusqu'à une nouvelle sélection).

5 Sources

— Canvas React/TypeScript : Github Arthur Escriou
— Sprites des planètes : pixabay.com
— Sprites des astéroïdes : nicepng.com
— Image en arrière plan : stock.adobe.com
— Logo et sprites vaisseaux spatiaux : DALL-E