Rapport de Projet : E-Tron

ROUSEE Tom
AUSSANT Jonathan
HENRY Amand
SICOT Théo

28 mars 2025

1 Jeu de Tron multi-joueur et coalitions

Table des matières

1.1 Description du projet

Notre projet vise à implémenter une version multijoueur du jeu de Tron en C++ avec une visualisation en 3D sous DirectX11. L'IA repose sur des algorithmes de recherche. L'objectif est d'analyser l'impact de la profondeur de recherche sur les performances des équipes en fonction de la taille de la grille et de la composition des groupes.

1.2 Répartition des tâches

— Amand a été chargé de s'occuper de la transcription des algorithmes pseudo-code dans le jeu de Tron

— Théo a réalisé la structure du jeu et s'est occupé de la question scientifique en analysant les données obtenues par les algorithmes et en affichant les résultats dans une page en react

— Jonathan et Tom on pu s'occuper d'une partie visuel avec une fenêtre sous Windows32 et un moteur de jeu en DirectX11

2 Objectif du projet

2.1 La Problématique

Quelle influence la profondeur de recherche a-t-elle sur le jeu en fonction de la taille des équipes et de la taille de la grille ?

2.2 Nos intérrogations

Une question centrale est de savoir s'il existe un seuil de profondeur au-delà duquel la recherche devient inefficace. À partir de quel point l'exploration supplémentaire n'apporte-t-elle plus d'amélioration significative sur les décisions de nos IA ? Et ce seuil dépend-il de la taille de la grille et/ou du nombre de joueurs ? Si un tel seuil existe, comment le déterminer ? Peut-on l'identifier en mesurant l'évolution des performances en fonction de la profondeur ? Est-il fixe, ou bien varie-t-il selon la situation en cours ? Enfin, à quel moment la recherche devient-elle trop coûteuse par rapport au gain stratégique ? Augmenter la profondeur améliore-t-il toujours la qualité du jeu, ou atteint-on un point où le coût dépasse l'intérêt tactique ? Existe-t-il un compromis optimal entre précision et rapidité pour garantir de bonnes décisions sans alourdir excessivement les calculs ?

2.3 Nos ambitions

Dès le départ, nous avons voulu nous lancer un défi en choisissant **C++** pour ce projet. Plutôt que d'opter pour une solution plus simple, nous avons voulu exploiter un langage plus exigeant afin de mieux comprendre les rouages d'un moteur de jeu et de gagner en maîtrise sur la gestion des performances.
L'idée d'utiliser **DirectX** est d'ailleurs née d'une blague entre nous. Ce qui devait être une simple plaisanterie s'est transformé en un véritable challenge que nous avons décidé de relever. Finalement, ce choix nous a permis d'explorer en profondeur les aspects techniques du rendu 3D, la gestion de la mémoire et l'optimisation des performances.
En plus du moteur de jeu, nous avons voulu aller plus loin en développant un **outil d'analyse** pour mieux comprendre les performances de nos algorithmes et leurs prises de décision. Cela nous a amenés à concevoir une interface permettant d'afficher visuellement les tendances de déplacement et d'étudier l'impact des différentes stratégies d'IA.

3 Quels algorithmes pour répondre ?

Leur utilisation, leur fonctionnement et leur lien avec le projet Dans notre projet de jeu de Tron, l'intelligence artificielle joue un rôle central. Nous avons exploré plusieurs approches pour permettre aux agents de prendre des décisions stratégiques. Deux familles d'algorithmes se sont avérées particulièrement pertinentes : **Minimax** (avec ses variantes **MAXN** et **Paranoid**) et une méthode inspirée de **SOS (Socially Oriented Search)**. Ces méthodes étaient explicitement demandées dans les attendus du projet.

3.1 Minimax

Le Minimax est un algorithme décisionnel classiquement utilisé dans les jeux à information parfaite comme les échecs ou le go. Il repose sur une évaluation récursive des coups possibles, cherchant à minimiser les pertes tout en maximisant les gains. Cependant, dans un jeu comme Tron, qui peut inclure plusieurs joueurs et des interactions complexes, nous avons exploré deux variantes plus adaptées : **MAXN** et **Paranoid**. Un seul de ces deux algorithmes était requis, mais nous avons pris la décision d'implémenter les deux dans notre projet car le temps nous le permettait.

3.1.1 MAXN

Le Minimax standard est conçu pour des jeux à deux joueurs, alternant entre un joueur qui maximise et un joueur qui minimise. Or, dans un jeu multi-joueur comme Tron, il faut un algorithme capable d'évaluer plusieurs adversaires simultanément. Le MAXN est une extension naturelle du Minimax aux jeux à plus de deux joueurs. Son fonctionnement repose sur une évaluation de chaque action non pas sous un simple critère de maximisation/minimisation binaire, mais en attribuant une valeur à chaque joueur.

Utilisation dans notre projet :

— Chaque joueur dans la simulation a une fonction d'évaluation qui estime son score potentiel à partir du nombre de positions "sécuritaires" autour de lui (nombre de cases accessibles).

— L'algorithme explore l'arborescence des coups possibles, attribuant un vecteur de scores (un par joueur) à chaque état du jeu.

— Il prend ensuite la décision qui maximise son propre score sans considérer spécialement un adversaire comme principal opposant.

Cette approche permet une prise de décision plus naturelle en situation multi-joueur, mais elle suppose que chaque joueur agit de manière strictement rationnelle, ce qui n'est pas toujours le cas dans une partie réelle.

3.1.2 Paranoid

Contrairement à MAXN, qui considère chaque joueur comme une entité indépendante maximisant son propre score, **Paranoid** adapte Minimax à un cadre multi-joueur en sup-

posant que tous les adversaires sont alignés contre le joueur actif. Dans cette approche :

— Le joueur actif maximise son gain.

— Tous les autres joueurs sont considérés comme un seul agent combiné, jouant de manière coopérative pour réduire son score.

Pourquoi utiliser Paranoid ?

— Dans Tron, un joueur peut se retrouver encerclé par plusieurs adversaires. Dans ce cas, considérer les autres comme un groupe homogène hostile permet d'anticiper les pires scénarios et d'opter pour des stratégies plus prudentes

— Cette approche peut être plus efficace dans un cadre où la survie est prioritaire sur le score brut.

3.1.3 Comparaison entre MAXN et Paranoid

— **MAXN** offre une modélisation plus fidèle d'un jeu multi-joueur libre, mais peut sous-estimer les alliances tacites.

— **Paranoid** prépare mieux à des situations hostiles, mais peut donner des résultats trop défensifs.

Dans notre projet, nous avons implémenté les deux méthodes car nous avons estimé qu'il valait mieux disposer d'un large éventail de résultats statistiques. Nous analyserons ultérieurement les performances de ces approches et verrons si nos hypothèses sur leurs avantages respectifs sont confirmées par les résultats expérimentaux.

3.2 implémenter un algorithme en s'inspirant de SOS

En plus des algorithmes de prise de décision individuelle, nous avons cherché un moyen d'organiser les joueurs tout en conservant un système de prise de décision efficace. Pour cela, nous nous sommes inspirés du concept **SOS (Socially Oriented Search)**, couramment utilisé dans l'optimisation et la théorie des jeux. Cette approche faisait également partie des attendus du projet. **Pourquoi SOS ?**

— SOS permet de formuler des contraintes de déséquilibre entre différents agents, créant des comportements agressifs ou préventifs.

— Simple d'utilisation, il permet d'appliquer un poids à chaque possibilité, en fonction de sa dangerosité, à savoir, la proximité d'un ennemi.

Application dans notre projet :

— Une matrice d'affinités peut être générée aléatoirement ou spécifiée

— Lors de chaque tour, l'algorithme MAXN, modifié pour intégrer SOS va lister normalement les chemins possibles par score puis ajoute un multiplicateur, une pondération, à chaque chemin, en fonction du risque représenté.

— Cela permet de préserver la survie d'un joueur face à un adversaire agressif, prioritairement à un chemin à possibilité élevée.

Cette approche, bien que différente des résolutions classiques des jeux de stratégie, nous a permis d'obtenir des parties plus dynamiques et plus intéressantes à observer. Nous évaluerons ultérieurement l'impact réel de ce modèle sur la compétitivité et l'équilibre des parties.

3.3 AutoMoveBFS : un algorithme supplémentaire pour l'exploration

Bien que cet algorithme n'ait pas été explicitement demandé, nous avons décidé de l'ajouter afin d'améliorer la capacité du joueur à se déplacer de manière autonome en fonction de l'espace disponible.
L'algorithme **AutoMoveBFS** utilise une approche de recherche en largeur **(BFS – Breadth-First Search)** pour évaluer la zone accessible depuis une position donnée. Son fonctionnement se déroule en deux étapes principales :

— **Calcul de la zone accessible (bfsArea)** : À partir de la position actuelle du joueur, une exploration en largeur est réalisée pour compter le nombre de cases atteignables. Une file (queue) est utilisée pour parcourir la carte, et un ensemble de cases visitées empêche de repasser sur les mêmes positions. L'objectif est d'évaluer la taille de la zone accessible depuis une position donnée.

— **Choix du meilleur déplacement (decideMoveBFS)** : Le joueur teste chaque mouvement possible et simule son déplacement. Pour chaque position atteignable, la fonction bfsArea est appelée pour estimer la taille de la zone qui serait accessible après ce déplacement. Une fois toutes les options évaluées, le mouvement qui offre la plus grande liberté de déplacement est sélectionné.

Cet algorithme permet ainsi d'optimiser les déplacements du joueur en favorisant les zones offrant le plus grand espace exploitable, ce qui peut être utile pour éviter les blocages et maximiser les options stratégiques.

3.4 Conclusion provisoire

L'IA dans notre jeu de Tron repose donc sur des algorithmes avancés de prise de décision et d'organisation des joueurs. **MAXN** et **Paranoid** permettent d'aborder différemment les défis du jeu multi-joueur, tandis que l'utilisation d'une méthode inspirée de **SOS** nous a permis de structurer les joueurs de façon efficace.

4 Comment structurer le projet ?

Le projet consiste à développer un jeu inspiré de Tron, où plusieurs joueurs contrôlent des motos lumineuses qui laissent un mur derrière elles. Le but est d'éviter de percuter un mur placé par une autre joueur ou sois même, tout en tentant de piéger les adversaires. Ce projet nécessitera la mise en place d'une logique de jeu fluide ainsi qu'un rendu graphique performant.

4.1 Le jeu de Tron

Pour concrétiser le jeu en code, nous avons adopté une approche modulaire en représentant le plateau sous forme d'une **grille**, où chaque case peut contenir différents types d'entités. Les principales entités du jeu incluent les **murs**, qui définissent les limites et les obstacles, ainsi que les **joueurs**, qui se déplacent tour par tour en fonction d'un **algorithme de décision** qui leur est assigné. Chaque joueur suit un algorithme spécifique qui détermine sa prochaine action en fonction de l'état actuel de la grille. Cet algorithme permet comme MAXN ou Paranoid d'anticiper les mouvements adverses. À chaque tour, les joueurs exécutent leur algorithme, choisissent une direction, et mettent à jour la grille en conséquence. Le jeu se poursuit jusqu'à ce qu'il ne reste plus qu'un seul joueur en vie ou qu'une condition de fin soit atteinte. Ce modèle permet une grande flexibilité : il est facile d'expérimenter avec différents algorithmes d'IA en remplaçant simplement l'algorithme de décision d'un joueur, sans modifier la structure générale du jeu.

4.2 Choix technologiques

Pour réaliser ce projet, plusieurs technologies ont été sélectionnées afin d'assurer un bon équilibre entre performance, rendu graphique et analyse des parties jouées :

- **C++** : Nous avons choisi C++ car, après avoir largement travaillé avec Java et Python durant nos trois années de licence, nous souhaitions explorer un langage plus bas niveau, réputé pour sa rapidité et son efficacité. C'était également un challenge intéressant qui nous permettait de mieux comprendre la gestion fine de la mémoire et d'optimiser les performances du jeu.
- **CMake** : CMake a été choisi pour sa capacité à gérer la compilation de manière efficace et à s'adapter à différents environnements de développement. Cela nous a permis de simplifier le processus de construction du projet et d'assurer une portabilité entre différentes plateformes.
- **DirectX 11** : Nous avons décidé d'utiliser DirectX 11 afin d'éviter les bibliothèques graphiques intermédiaires et travailler directement à un niveau plus bas pour mieux maîtriser le rendu. Cette approche nous a permis d'avoir un contrôle total sur l'affichage, d'assurer un rendu fluide et optimisé, et d'explorer en profondeur le fonctionnement d'un moteur graphique minimaliste.
- **Multithreading** : Le multithreading a été utilisé pour améliorer la fluidité et l'efficacité du jeu. Nous avons séparé le calcul des algorithmes des joueurs du moteur

de jeu lui-même afin d'éviter les blocages et d'assurer une exécution réactive. De plus, nous avons intégré le multithreading dans DirectX 11 pour afficher un écran de chargement pendant que les indices et vertices du jeu sont calculés, améliorant ainsi l'expérience utilisateur.

— **React** : Nous avons utilisé React pour créer une interface web permettant d'afficher et d'analyser les statistiques des parties. Cela inclut le suivi détaillé des actions des joueurs, la visualisation des décisions prises à chaque tour, ainsi que l'exploitation des données sous forme de graphiques et d'indicateurs pertinents. React nous a semblé être un choix pertinent pour présenter ces informations de manière claire, interactive et esthétique.

— **Double Git** : Pour pouvoir profiter pleinement de ce projet pour notre avenir, nous avons mis en place un double git afin de mettre le projet non seulement sur la forge pour le rendu de l'UE, mais aussi sur Github pour pouvoir garder ce projet et ses commits de notre côté après la fin de l'année.

Cette combinaison de technologies nous a permis d'optimiser à la fois les performances du jeu, la gestion des calculs et l'analyse des résultats, tout en relevant un défi technique stimulant.

4.3 Présentation de l'architecture du projet

Représentation du projet via les packages et fichiers principaux.

4.3.1 Exécuter le jeu

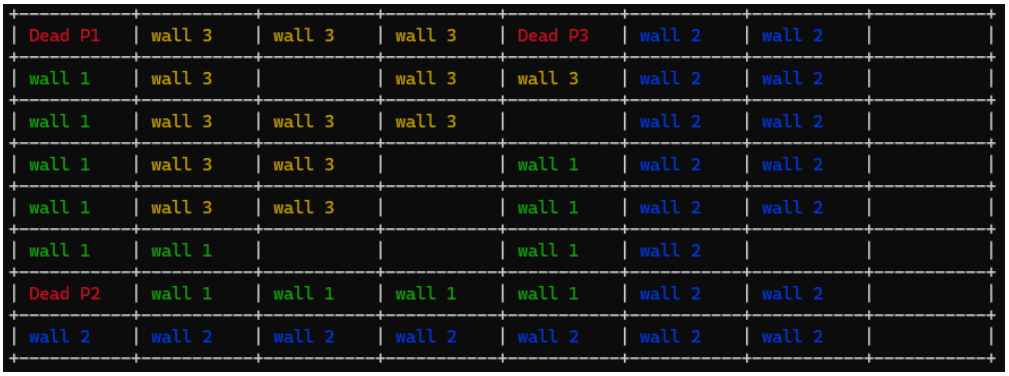


FIGURE 1 – visualisation du jeu en console

Notre projet offre deux modes d'exécution : une **visualisation en console** et un affichage graphique avec DirectX. La version console permet de suivre facilement le déroulement du jeu avec une représentation claire des murs et des positions des joueurs, offrant une compréhension rapide de l'état de la partie en temps réel.
Pour personnaliser les conditions de jeu, nous avons intégré un fichier **config.ini** à la racine du projet. Ce fichier permet d'ajuster différents paramètres comme la taille de la

grille, le nombre de joueurs, l'algorithme utilisé, ou encore des options plus spécifiques comme l'activation du spawn aléatoire des joueurs et la profondeur d'exploration des algorithmes. Grâce à cette approche, nous pouvons tester rapidement différentes configurations sans modifier directement le code source.

4.3.2 Représentation avec Gource

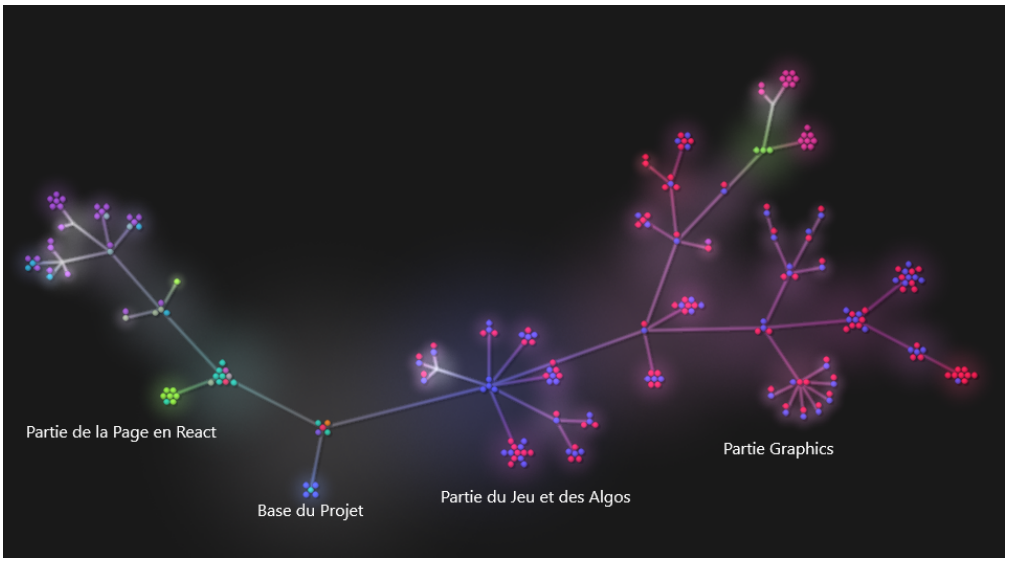
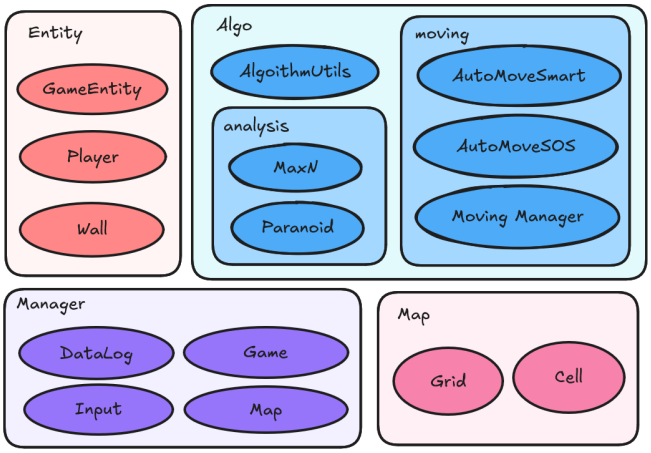


FIGURE 2 – Représentation du projet avec Gource

Juste ci-dessus on peut voir une représentation du projet avec Gource, qui nous permet de percevoir tous les fichiers sous formes de bulles. A la base du projet on peut voir les fichier de configuration tel que le CMakeLists, le README etc... sur la droite on peut voir le /src avec les dossiers principaux tels que les managers, les algos, etc.. Si on continue on peut voir toute la partie graphique avec tout ses sous dossier qui occupe une grande partie du projet. Si on revient sur la gauche on peut voir toute la partie de la page en react qui nous sert à voir les résultats de nos algorithmes.

4.3.3 Structure plus précise

Diagramme de la base



Juste ici on peut voir une structure plus détaillée de la structure de base du jeu avec les fichier associée par package principaux.
Les managers vont relier ces fichiers ensemble pour faire le jeu et envoyer les résultats à la page web.

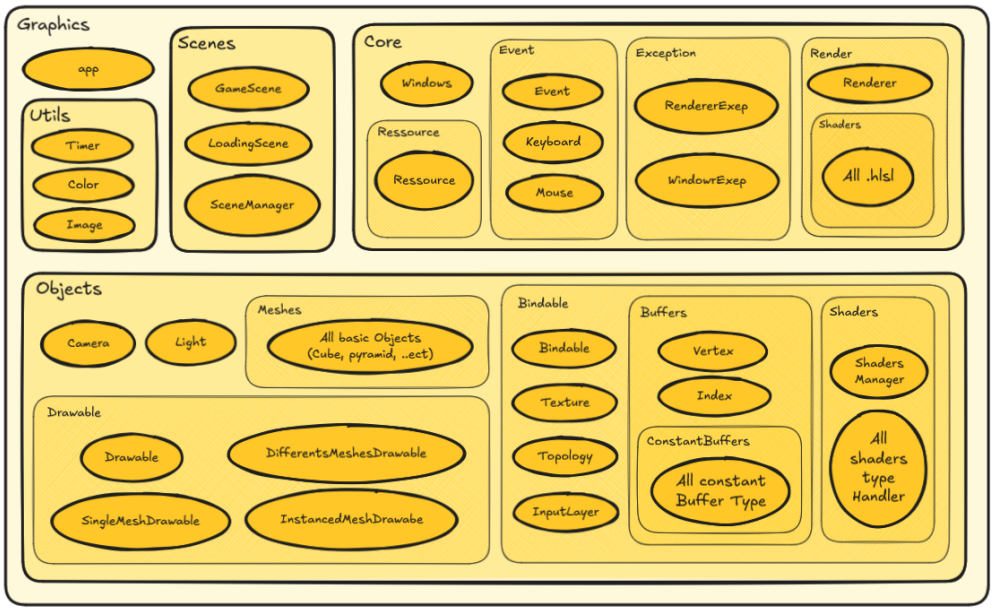


FIGURE 3 – Représentation du projet par fichier et package important

Juste au dessus de ça on peut voir une structure plus détaillée mais toujours simplifiée pour ne montrer que les fichier et package important de la partie graphique. On détaillera son fonctionnement dans la partie suivante, mais cela permet de se représenter que la cette partie nous a pris plus de ressource que prévu et représente une partie importante du projet.

5 Le visuel en DirectX11

5.1 Le rendu visuel du jeu

Dans ce projet, l'aspect visuel joue un rôle essentiel pour offrir une représentation claire et fluide du jeu. L'objectif était de créer un rendu simple mais efficace, permettant aux joueurs de suivre l'évolution de la partie sans latence ni ambiguïté. Pour cela, nous avons choisi d'utiliser DirectX 11 en C++, une combinaison qui nous permet d'avoir un contrôle total sur l'affichage tout en optimisant les performances. Dans les sections suivantes, nous détaillerons le fonctionnement du moteur graphique, la manière dont nous avons modélisé les éléments du jeu, les effets visuels implémentés, ainsi que les optimisations mises en place pour assurer un rendu fluide et réactif.

5.2 Fonctionnement du moteur graphique

Le fonctionnement du moteur graphique repose sur une architecture assez technique, mais dans cette section, nous allons volontairement éviter les détails trop complexes pour nous concentrer sur l'idée générale.
Le jeu tourne à l'intérieur d'une application principale qui gère la création et le rendu de scènes. Une **scène** représente un état visuel du jeu à un instant donné, et elle est composée d'un ensemble d'objets que nous appelons des **Drawables**. La scène principale du jeu, celle qui affiche tout ce qui se passe pendant une partie, est appelée GameScene.
Les **Drawables** sont les briques de base de l'affichage. Chaque Drawable représente un objet du jeu (joueur, mur, grille, etc.). Ils sont répartis en trois grandes familles : SingleMeshDrawable, InstanceMeshDrawable et CompositeDrawable, que nous détaillerons plus tard. De manière générale, un Drawable contient un **Mesh**, c'est-à-dire une structure composée de points **(vertices)** et de triangles **(indices)** qui permet de modéliser des formes simples comme un cube ou un cylindre.
Le rôle d'un **Drawable** est d'être préparé pour le rendu. Cela signifie qu'il doit rassembler toutes les données nécessaires (position, couleur, texture, etc.) et les transmettre efficacement à DirectX. Pour cela, nous utilisons des **Bindables**, qui agissent comme des **wrappers** autour des appels nécessaires à l'envoi des buffers. Ces buffers, déjà créés en amont, sont ensuite intégrés dans le **pipeline** de rendu de DirectX. Puisque DirectX repose sur le modèle **COM (Component Object Model)**, nous avons utilisé des pointeurs intelligents (ComPtr) pour gérer automatiquement la durée de vie des objets et éviter les fuites mémoire.
Une fois ces données correctement mises en mémoire, des **Shaders** prennent le relais pour effectuer les calculs indispensables, notamment le positionnement des objets dans la scène (via les matrices de transformation) et l'application des effets visuels comme la gestion de la lumière ou des couleurs.
Enfin, tout ce rendu est affiché via un Renderer basé sur DirectX 11, intégré dans une fenêtre native Windows32, ce qui nous permet d'avoir un affichage fluide et contrôlé à bas niveau.
Cette structure nous permet d'avoir un système modulaire et optimisé, dans lequel chaque

élément du jeu est clairement défini, facilement manipulable, et intégré de façon cohérente dans la scène graphique globale.

5.3 Optimisation du rendu et fluidité

Au cours du développement, lors de nos tests, nous avons rencontré un problème majeur : l'affichage d'une grille de 20x20 cubes prenait **6,7 secondes**, ce qui était bien trop long au vu de notre objectif d'afficher de grande grille pour tester les algorithmes. Il était donc impératif d'optimiser notre moteur graphique pour réduire ce temps de chargement et améliorer la fluidité du jeu.
Pour y parvenir, nous avons mis en place plusieurs optimisations clés, que nous allons détailler dans les sections suivantes.

5.3.1 Utilisation des Constant Buffers

Dès le début du projet, nous avions anticipé certains besoins d'optimisation en mettant en place des **Constant Buffers**, une structure native de DirectX, qui sert à transmettre des données aux Shaders de manière rapide et optimisée. Son principal avantage réside dans sa rapidité d'accès, ce qui en fait un outil clé pour le rendu graphique.
Le Constant Buffer est principalement utilisé pour stocker des données qui changent rarement, comme certaines propriétés globales de la scène (ex. l'éclairage général). Cependant, nous l'avons également utilisé pour des informations qui évoluent régulièrement mais nécessitent des mises à jour optimisées, comme le déplacement des objets, les changements de caméra, ou encore l'ajustement de la fenêtre. Grâce à l'alignement mémoire, DirectX permet d'accéder à ces données de manière efficace, évitant ainsi des recalculs lourds ou inutiles.

5.3.2 Mise en place d'un écran de chargement

Un autre problème que nous avons rencontré était la perception du temps de chargement par l'utilisateur. Même avec des optimisations, l'initialisation du jeu demandait plusieurs secondes. Pour améliorer l'expérience utilisateur, nous avons donc intégré un écran de chargement dynamique qui masque le temps de traitement en affichant une animation.
Pour éviter que cet écran bloque l'exécution principale du jeu, nous avons utilisé le **multithreading asynchrone** : pendant que le jeu charge en arrière-plan, l'écran de chargement est rendu sur un thread séparé, offrant une transition fluide et évitant les impressions de freeze.

5.3.3 Optimisation du rendu avec l'Instancing

L'un des problèmes majeurs venait de notre gestion des objets dans la grille de jeu. Avant l'optimisation, chaque cube était traité individuellement, ce qui entraînait une explosion du nombre de triangles envoyés au GPU. Par exemple, si nous avions 400 cubes

(une grille 20x20), cela signifiait que nous générions 4800 triangles supplémentaires rien que pour cette grille.

Pour résoudre cela, nous avons repensé notre gestion des Drawables, en les séparant en trois catégories :

- **SingleMeshDrawable** : utilisé pour les objets uniques comme la skybox.
- **CompositeDrawable** : regroupe plusieurs Mesh ensemble, permettant de les manipuler efficacement sans recalculer chaque élément individuellement. Par exemple les motos des joueurs qui sont unique et complexe
- **InstanceMeshDrawable** : permet de dupliquer un objet sans recréer de nouvelles données.

L'instancing a donc été une solution clé : au lieu de générer chaque objet individuellement, nous avons créé un seul Drawable en mémoire et l'avons dupliqué autant de fois que nécessaire sous forme d'instances fantômes. Cette technique est particulièrement efficace pour **les objets identiques**, car toutes les instances partagent la même géométrie, ce qui réduit considérablement la charge de calcul. Elle est donc idéale pour les éléments répétitifs comme les murs ou le sol, où chaque objet est strictement identique aux autres. Elle nous a permis d'augmenter de manière conséquente les images par secondes car le fait qu'il y ait moins d'appels au GPU, on a moins de switch CPU / GPU.

5.3.4 Fix des rechargements inutiles des shaders

Un autre problème de performance était lié aux **shaders**, qui sont des programmes exécutés directement sur le GPU pour gérer l'affichage des objets (lumière, textures, transformations...). Nous avons découvert, grâce à la mise en place d'exceptions personnalisées, que nous rechargions les shaders à chaque création de Mesh, ce qui alourdissait considérablement les performances. La solution a été simple mais efficace : plutôt que de recharger un shader à chaque fois qu'un nouvel objet était créé, nous avons **préchargé tous les shaders au démarrage du jeu** et les avons stockés en mémoire vive. Ensuite, chaque objet pouvait simplement les réutiliser, évitant ainsi des chargements redondants et coûteux en ressources.

5.3.5 Bilan des optimisations

Grâce à toutes ces optimisations, nous avons transformé les performances du jeu. Avant, une grille de 20x20 cubes prenait 6,7 secondes à générer. Aujourd'hui, nous pouvons générer une grille de **1000x1000 cubes en seulement 2 secondes**, un gain spectaculaire qui prouve l'efficacité des optimisations mises en place. Cela nous a permis d'avoir un jeu fluide, rapide et réactif, tout en conservant une grande flexibilité pour gérer des scènes complexes.

5.4 Effet visuel et expérience utilisateur

Effet visuel et expérience utilisateur L'expérience utilisateur est un élément clé dans tout jeu, et bien que certains ajouts ne soient pas indispensables au gameplay, ils améliorent considérablement l'immersion et le confort visuel. Nous avons donc intégré plusieurs **effets visuels**, qui apportent une meilleure perception de l'environnement et rendent le jeu plus agréable à parcourir. Ces améliorations passent par l'éclairage, l'utilisation de textures, l'ajout d'une skybox et un système de caméra dynamique.

5.4.1 Lumière

Pour gérer l'éclairage correctement, nous avons dû revoir la structure de nos Meshs : chaque face possède désormais ses propres sommets (vertices), évitant ainsi le partage d'indices entre plusieurs triangles. Cette modification était nécessaire pour **calculer correctement les normales des objets**, et donc permettre un rendu cohérent des ombres et des reflets. À l'aide d'un shader de calcul matriciel utilisant **l'algorithme de Phong**, nous avons pu simuler un éclairage réaliste, améliorant nettement la profondeur et la lisibilité des scènes.

5.4.2 Textures

Nous avons ajouté des textures pour rendre les éléments plus attrayants, notamment dans l'écran de chargement. Cela évite l'impression d'un rendu trop brut ou trop "fait maison", en offrant une meilleure cohérence visuelle et une sensation de fini plus professionnelle.

5.4.3 Skybox

Lors des premiers tests, nous avons remarqué que l'environnement paraissait **trop vide**, ce qui nuisait à l'immersion. Pour corriger cela, nous avons intégré une skybox : une immense sphère englobant toute la scène, avec une texture appliquée sur sa face intérieure. Cela donne l'illusion d'un ciel dynamique, renforçant l'ambiance du jeu et rendant l'espace moins artificiel.

5.4.4 Caméra

Le système de caméra repose sur la détection des événements clavier, ce qui permet de modifier dynamiquement la position et l'orientation de la vue. Concrètement, plutôt que de déplacer la scène, nous appliquons ces transformations sur tous les objets visibles, simulant ainsi un déplacement fluide de la caméra. Ce système offre une liberté totale pour explorer la scène sous tous les angles, renforçant le contrôle et l'immersion du joueur.

5.4.5 Son

Un son a été rajouter lors du splashscreen grâce à Windows32. Cela permet de rendre le jeu plus vivant et d'immerger le joueur dans l'expérience.

5.5 Mise en place du jeu dans le visuel

Pour assurer la liaison entre la logique du jeu et son affichage, nous utilisons un **DataLinker**. Plutôt que de mettre à jour l'ensemble de la grille à chaque tour, ce qui serait inefficace, nous avons choisi d'optimiser le processus en enregistrant uniquement les informations essentielles : les déplacements des joueurs et leur état. À chaque mise à jour, **GameScene** récupère ces données et applique les transformations nécessaires aux objets correspondants. Cette approche permet de réduire la charge de calcul tout en assurant une synchronisation fluide et précise entre le gameplay et le rendu visuel sous DirectX.

5.6 Quelque exemples

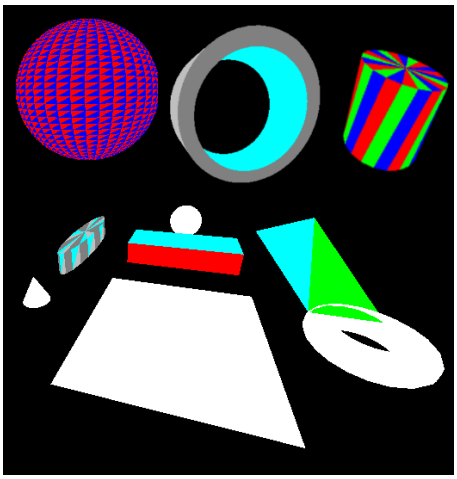


FIGURE 4 – Les meshs du jeu sous leur forme brute

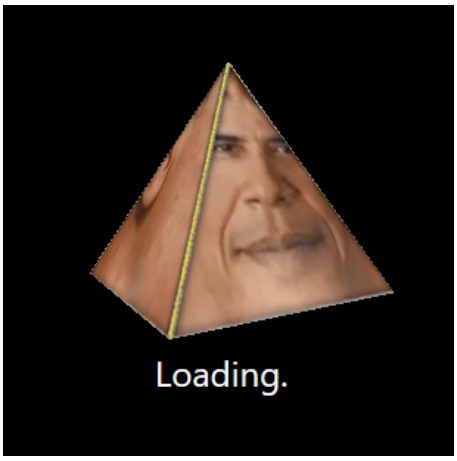


FIGURE 5 – Rendu du menu de chargement du jeu

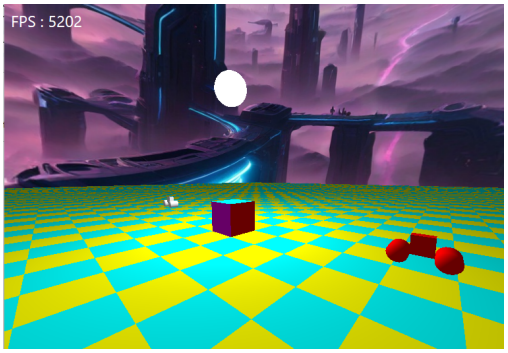


FIGURE 6 – Premiers tests du jeu avec une grille et des joueurs

FIGURE 7 – Rendu final du jeu avec une grille et des joueurs

6 Analyse des Fonctionnalités et Résultats de la Page React

6.1 Fonctionnalités de la Page React

La page React développée permet d'analyser les résultats des parties de Tron jouées par les algorithmes. Elle repose sur le chargement d'un fichier JSON généré après chaque partie, et offre plusieurs fonctionnalités pour visualiser et interpréter les performances des IA.

1. Chargement et Analyse des Données

— La page accepte un fichier JSON contenant les données de la partie.

— Elle extrait et affiche les statistiques des joueurs sous forme de graphiques et de timeline.

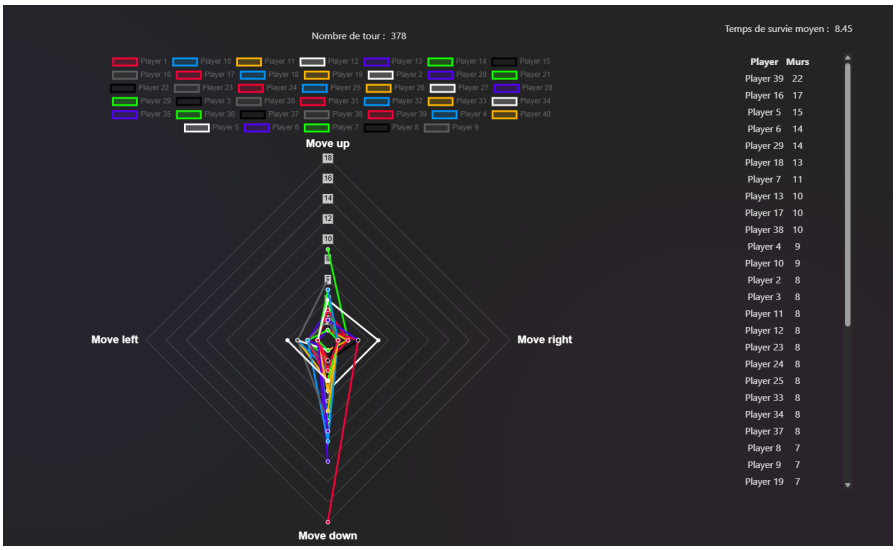


FIGURE 8 – Visualisation des Tendances de Déplacement

2. Visualisation des Tendances de Déplacement

— Un graphique radar montre les moyennes des mouvements effectués par chaque joueur.

— Les tendances de déplacement sont représentées pour comparer le comportement des IA.

— Il est possible d'afficher ou masquer les statistiques d'un joueur en activant/désactivant son affichage.

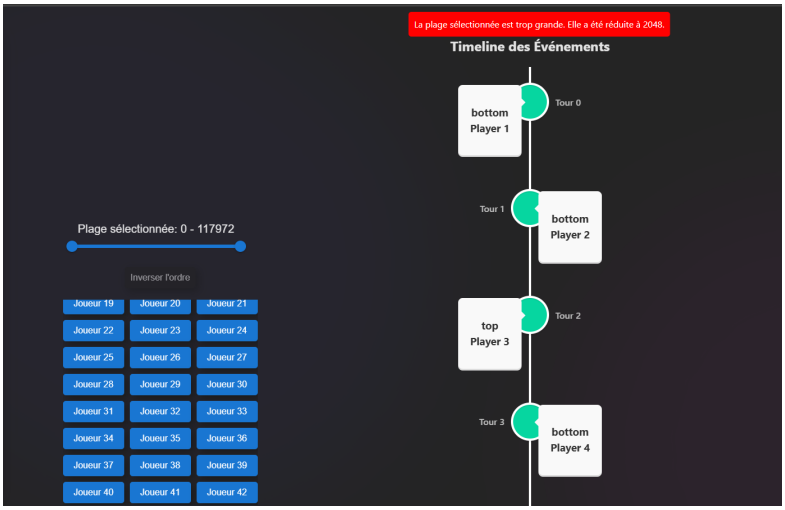


FIGURE 9 – Timeline des Actions

3. Timeline des Actions

— Une seconde page permet de voir les actions des joueurs tour par tour.

— Un curseur permet de sélectionner une plage de tours à afficher.

— Les actions peuvent être filtrées par les joueurs.

FIGURE 10 – Temps de Survie Moyen selon la Profondeur de Recherche

4. Temps de Survie Moyen selon la Profondeur de Recherche

— Un graphique linéaire affiche l'évolution du temps de survie moyen en fonction de la profondeur de recherche des algorithmes.

— Cela permet d'analyser l'impact de la profondeur de recherche sur la performance des joueurs.

6.2 Analyse des Résultats

L'analyse des parties jouées par les algorithmes met en lumière plusieurs tendances récurrentes dans leur comportement, ainsi que des observations intéressantes sur les performances en fonction de la profondeur de recherche.

6.2.1 Identification d'un Seuil d'Exploration

L'un des premiers constats issus de notre analyse est que les algorithmes ont tendance à adopter des schémas de déplacement optimisés pour éviter les conflits entre joueurs. Quel que soit l'algorithme utilisé (MAXN, Paranoid, SOS ou BFS), on observe une forte propension à évoluer en ligne droite, souvent en synchronisation avec les autres joueurs, en alternant des mouvements de gauche à droite ou d'avant en arrière.
Ce phénomène s'explique par le fait que ces déplacements permettent aux agents d'exploiter l'espace de manière rationnelle en minimisant les risques de collision et en maximisant le contrôle de leur zone de jeu. Ainsi, même sans coordination explicite entre eux, les joueurs adoptent des comportements qui rappellent des stratégies collaboratives, où chacun cherche à préserver son propre espace sans perturber les trajectoires des autres.
Cependant, nous avons remarqué que cette stratégie atteint une limite lorsque les joueurs sont trop nombreux ou que l'espace devient restreint. Une fois ce seuil d'exploration dépassé, les IA doivent adopter des mouvements plus erratiques et commencent à s'éliminer plus rapidement.

6.2.2 Mesure des Performances en Fonction de la Profondeur

L'un des éléments clés de notre analyse repose sur l'étude du temps de survie moyen des joueurs en fonction de la profondeur de recherche de l'algorithme. Le graphique généré met en évidence plusieurs tendances :

— À faible profondeur, les IA prennent des décisions rapides mais manquent d'anticipation, ce qui conduit souvent à des éliminations précoces.

— En augmentant la profondeur de recherche, on observe une nette amélioration du temps de survie moyen, car les joueurs parviennent à mieux anticiper les menaces et à optimiser leurs déplacements.

— Toutefois, passé un certain seuil, l'amélioration devient moins significative et peut même se dégrader dans certains cas.

Cette observation suggère qu'il existe un compromis à trouver entre la profondeur d'exploration et la rapidité d'exécution, afin d'optimiser la performance des joueurs sans alourdir inutilement les calculs.

6.2.3 Équilibre entre Coût et Gain Stratégique

Le principal défi des algorithmes utilisés réside dans l'équilibre entre le coût de calcul et le gain stratégique obtenu. Nos observations montrent que si une recherche trop superficielle entraîne des décisions sous-optimales, une profondeur excessive ne garantit pas forcément un avantage déterminant.
Ce constat est particulièrement visible avec les algorithmes comme Paranoid et MAXN, qui nécessitent une grande puissance de calcul pour évaluer plusieurs niveaux de jeu. À l'inverse, BFS, bien qu'explorant un grand nombre de possibilités, peut parfois être pénalisé par sa manière d'étendre l'arbre de décision.
En pratique, le choix du paramètre de profondeur doit être ajusté en fonction du contexte de la partie et des ressources disponibles. Une future amélioration pourrait être d'adapter dynamiquement cette profondeur en fonction des situations rencontrées, afin d'optimiser en temps réel l'équilibre entre réactivité et stratégie à long terme.

7 Comment aller plus loin ?

7.1 A travers les algorithmes

Bien que nous ayons expérimenté les algorithmes **MAXN** et **Paranoid**, ainsi que des idées issues de l'approche **SOS (Survival-Oriented Search)**, nous avons identifié d'autres algorithmes intéressants à explorer. Si nous avions eu plus de temps, nous aurions probablement ajouté ces approches pour améliorer la prise de décision des joueurs et comparer leur efficacité.

7.1.1 MCTS (Monte Carlo Tree Search)

L'algorithme **MCTS** est particulièrement utilisé dans les jeux stratégiques comme le Go ou les échecs. Plutôt que d'explorer exhaustivement toutes les possibilités comme MAXN, il réalise des simulations aléatoires pour estimer la qualité des coups. Dans notre jeu, MCTS aurait pu être intéressant pour :

— Trouver des stratégies adaptatives sans avoir à définir explicitement une heuristique.

— Réduire le coût computationnel par rapport à une recherche exhaustive en ne simulant que les coups les plus prometteurs.

— S'adapter dynamiquement aux comportements adverses grâce aux simulations.

Cependant, un défi avec MCTS dans Tron est qu'il repose souvent sur des heuristiques de fin de partie, ce qui aurait nécessité un bon modèle d'évaluation des positions.

7.1.2 Alpha-Beta Pruning pour un jeu à deux joueurs

Si nous voulions nous concentrer sur des scénarios en duel (1v1), nous aurions pu utiliser Minimax avec **élagage alpha-bêta**. Cet algorithme :

— Réduire l'espace de recherche en éliminant les branches inutiles, rendant la recherche plus efficace.

— Permet d'intégrer une fonction d'évaluation plus fine pour estimer les chances de victoire.

— Serait particulièrement adapté aux parties à deux joueurs, contrairement à MAXN, qui est optimisé pour les jeux multijoueurs.

L'inconvénient de Minimax avec alpha-bêta est qu'il dépend fortement de la qualité de la fonction d'évaluation et peut être limité si la profondeur de recherche est trop faible.

7.1.3 Conclusion

Si nous avions eu plus de temps, l'ajout de MCTS et de Minimax Alpha-Bêta aurait constitué notre prochaine étape. Ces algorithmes auraient permis d'explorer des stratégies alternatives et de comparer leurs performances face à MAXN et Paranoid. Tester ces différentes approches aurait enrichi notre analyse et aurait potentiellement révélé des comportements de jeu plus optimisés.

7.2 A travers le rendu graphique

Bien que notre moteur graphique permette déjà une visualisation fluide du jeu, plusieurs améliorations pourraient être envisagées pour enrichir l'expérience visuelle et sonore.

7.2.1 L'animation

Actuellement, certains éléments du jeu apparaissent de manière instantanée sans transition visuelle, ce qui peut rendre l'affichage un peu abrupt. Une amélioration notable serait d'ajouter des animations pour rendre ces changements plus fluides et dynamiques. Par exemple, l'apparition des murs pourrait être accompagnée d'une animation de montée progressive ou d'une matérialisation avec un effet visuel, plutôt qu'une simple apparition soudaine. De même, lorsqu'un joueur perd la partie, une animation de destruction, comme une explosion ou une désintégration progressive, pourrait être ajoutée pour rendre l'événement plus marquant.
Ces améliorations rendraient le jeu plus agréable visuellement et donneraient davantage d'impact aux événements clés d'une partie.

7.2.2 La gestion de mesh externe

Pour le moment, les objets affichés sont construits à partir de primitives relativement simples. Une amélioration majeure serait d'ajouter la possibilité d'importer des modèles 3D externes au format .obj ou .fbx, ce qui permettrait d'avoir des représentations plus détaillées des joueurs, obstacles et autres éléments du décor. Cela ouvrirait également la voie à une personnalisation plus poussée du jeu.

7.2.3 Une meilleure gestion de l'audio grâce à XAudio2

L'intégration d'un moteur audio plus avancé, tel que **XAudio2**, permettrait d'améliorer significativement l'ambiance sonore du jeu. Cela inclurait des effets sonores dynamiques pour les déplacements des joueurs, des collisions ou encore des bruitages spécifiques selon l'évolution de la partie. De plus, une meilleure spatialisation du son renforcerait l'immersion en adaptant l'audio en fonction de la position des éléments dans la scène.

7.2.4 Conclusion

Ces améliorations visuelles et sonores contribueraient à rendre le jeu plus attrayant et immersif. Elles permettraient également d'exploiter davantage la puissance de **DirectX** et d'améliorer notre moteur graphique en lui offrant plus de flexibilité et de réalisme.

Références

[1] Tristan Cazenave. Negamax avec alphabeta. https://www.lamsade.dauphine.fr/~cazenave/papers/berder00.pdf.

[2] ChiliTomatoNoddle. C++ 3d directx programming. https://www.youtube.com/playlist?list=PLqCJpWy5Fohd3S7ICFXwUomYW0Wv67pDD.

[3] Richard E. Korf. Maxn avec coupes alphabeta. https://faculty.cc.gatech.edu/~thad/6601-gradAI-fall2015/Korf_Multi-player-Alpha-beta-Pruning.pdf.

[4] Nathan R. Sturtevant and Richard E. Korf. Maxn et paranoid. https://cdn.aaai.org/AAAI/2000/AAAI00-031.pdf.

[5] Brandon Wilson1, Inon Zuckerman2, and Dana Nau. Algorithme sos. https://www.cs.umd.edu/~nau/papers/wilson2011modeling.pdf.