Rapport de Projet Java: E-Tron

ROUSEE TOM
AUSSANT Jonathan
HENRY Amand
SICOT Théo

26 mars 2025

1 Jeu de Tron multi-joueur et coalitions

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

1.3 Plan du rapport

Table des matières

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 ?

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)** pour la gestion des équipes. 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 supposant 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 Spécifier des équipes en s'inspirant de SOS

En plus des algorithmes de prise de décision individuelle, nous avons cherché un moyen d'organiser les joueurs en équipes 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, afin de générer des équipes

— 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 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 équipes de façon efficace. Ces approches faisaient partie des exigences du projet et ont été implémentées pour évaluer leurs performances respectives. Nous verrons par la suite si nos estimations initiales concernant leur efficacité et leur pertinence se confirment avec les résultats obtenus en simulation.

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.

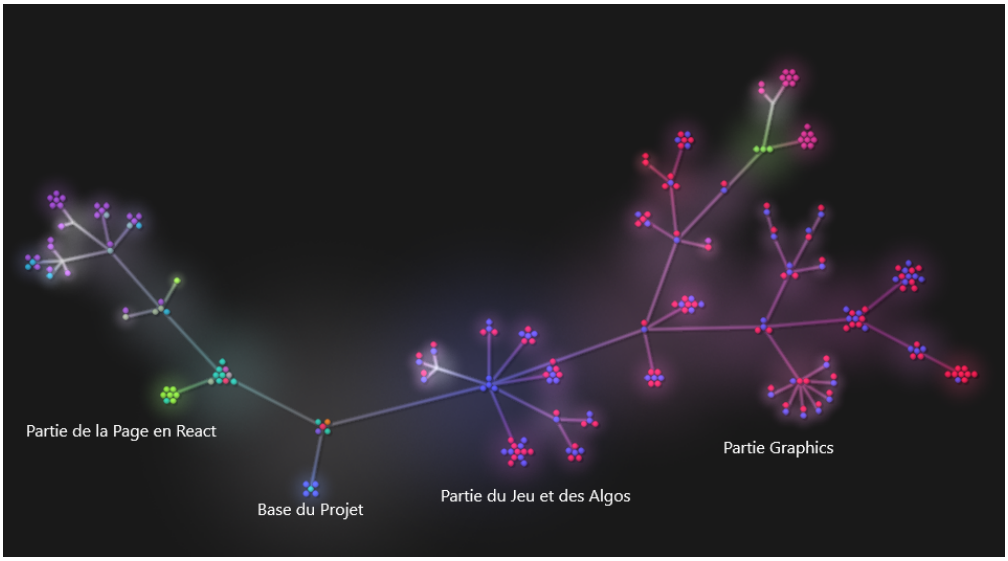
4.2 Choix technologiques

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.

— **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.

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

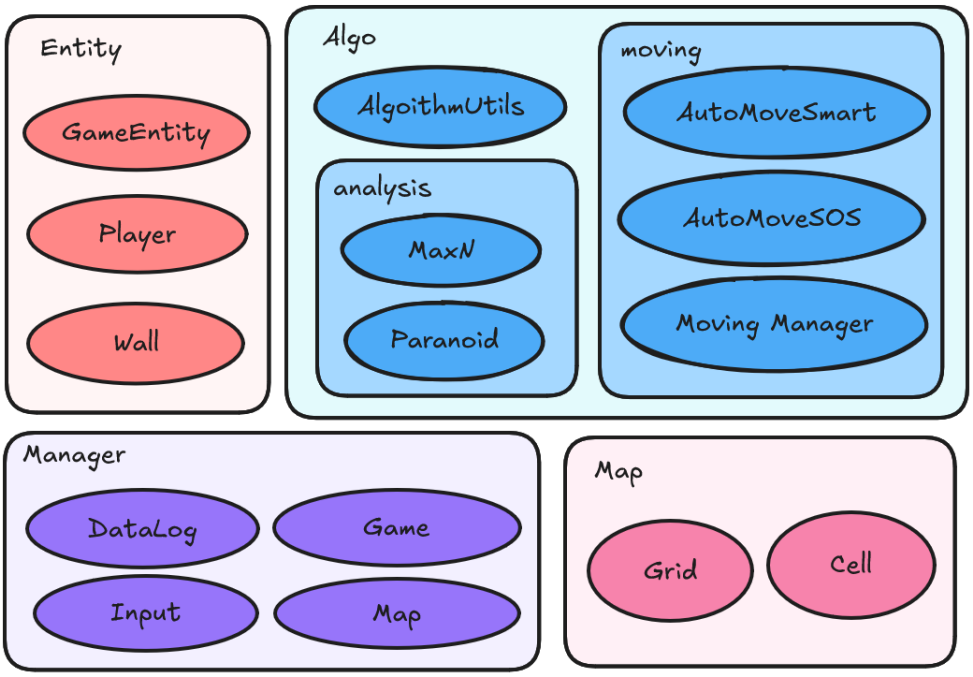
La structure du projet

4.3.1 Représentation avec Gource

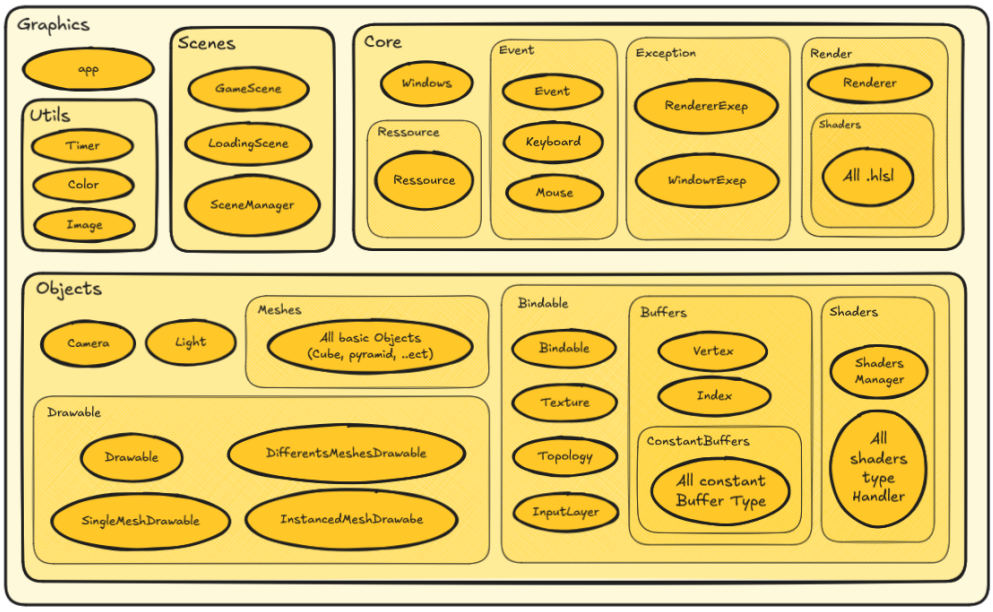
Juste ci-dessus on peut voir une représentation du projet avec Gource, qui nous permet de percevoir tous les fichiers sous formes de bulles. A la base du projet on peut voir les fichier de configuration tel que le CMakeLists, le README etc... sur la droite on peut voir le /src avec les dossiers principaux 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.

4.3.2 Structure plus précise

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. 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.


Graphics
app
Utils
Timer
Color
Image
Scenes
GameScene
LoadingScene
SceneManager
Core
Windows
Ressource
Ressource
Event
Event
Keyboard
Mouse
Exception
RendererExep
WindowsExep
Render
Renderer
Shaders
All .hlsl
Objects
Camera
Light
Meshes
All basic Objects
(Cube, pyramid, ..ect)
Drawable
Drawable
DifferentsMeshesDrawable
SingleMeshDrawable
InstancedMeshDrawabe
Bindable
Bindable
Texture
Topology
InputLayer
Buffers
Vertex
Index
ConstantBuffers
All constant
Buffer Type
Shaders
Shaders
Manager
All
shaders
type
Handler

5 Le visuel en DirectX11

5.1 Le rendu visuel du jeu

5.2 Fonctionnement du moteur graphique

5.3 Optimisation du rendu et fluidité

5.3.1 Utilisation des Constant Buffers

5.3.2 Mise en place d'un écran de chargement

5.3.3 Optimisation du rendu avec l'Instancing

5.3.4 Fix des rechargements inutiles des shaders

5.3.5 Bilan des optimisations

5.4 Effet visuel et expérience utilisateur

5.4.1 Lumière

5.4.2 Textures

5.4.3 Skybox

5.4.4 Caméra

5.4.5 Musique et son

5.5 Mise en place du jeu dans le visuel

5.6 Quelque exemples

6 Analyse des Fonctionnalités et Résultats de la Page React

6.1 Fonctionnalités de la Page React

6.2 Analyse des Résultats

6.2.1 Identification d'un Seuil d'Exploration

6.2.2 Mesure des Performances en Fonction de la Profondeur

6.2.3 Équilibre entre Coût et Gain Stratégique

7 Comment aller plus loin ?

7.1 A travers les algorithmes

7.1.1 MCTS (Monte Carlo Tree Search)

7.1.2 Alpha-Beta Pruning pour un jeu à deux joueurs

7.1.3 Conclusion

7.2 Analyse des Résultats

7.2.1 L'animation

7.2.2 La gestion de mesh externe

7.2.3 Conclusion

Références

[1] Dev Mind. Hashlife. https://www.dev-mind.blog/hashlife/.

[2] Wikipedia. Hashlife. https://en.wikipedia.org/wiki/Hashlife.