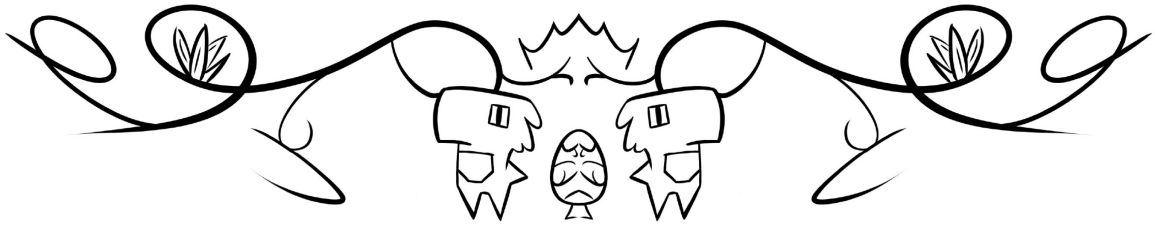
Play Together



Projet de programmation 2

PLAY TOGETHER

Auteurs
Mateo Avventuriero
Donovann Zassot

Année universitaire 2023-2024
L3 Informatique - Faculté des Sciences - Université de Montpellier

Table des matières

1 Présentation du projet

Le projet que nous avons réalisé au cours des quatres derniers mois est un jeu de plateforme [1] collaboratif en ligne. Il demande à quatre joueurs de coopérer pour surmonter des défis afin de progresser dans leur aventure. Cette aventure traverse plusieurs mondes variés avec des styles graphiques distincts et des mécaniques de jeu variées. Ces mondes sont sous-divisés en niveaux, ajoutant ainsi une progression au cours du jeu.

FIGURE 1 – Visuels du projet

Le but de ce projet est de comprendre comment créer un jeu vidéo 2D en partant de zero, sans moteur de jeu ni framework. C'est pourquoi nous avons choisi de le réaliser en C++ avec SDL.
En faisant ce choix, nous nous confrontons à la nécessité de développer des fonctionnalités généralement fournies nativement par des moteurs de jeu tels que Unity ou Unreal Engine. Parmi ces fonctionnalités, on peut citer la gestion des collisions, la mise en place d'une caméra, la gestion du réseau pour le multijoueur, l'optimisation des ressources incluant la limitation du framerate, l'intégration du delta time, ainsi que les phases larges et étroites.

Au-delà de son caractère ludique, notre projet couvre de nombreux domaines informatiques, et offre ainsi une occasion d'apprendre et de développer des compétences variées.
Le côté multijoueur en ligne pose des défis complexes en termes de gestion réseau, tandis que l'optimisation des performances est primordiale. En effet, étant une application en temps réel, un jeu nécessite un traitement rapide de chaque action pour éviter les retards perceptibles par le joueur. De même, la gestion des collisions et des mouvements dans un environnement dynamique constitue un défi algorithmique de taille. Il est important que les éléments du jeu aient des comportements réalistes et cohérents.
Cependant, un projet ne se résume pas uniquement à une somme de points techniques. Il s'agit également de développer un code de qualité, facilement réutilisable par l'équipe de développement, et de mettre en place une conduite de projet et une organisation efficaces. Cela inclut une gestion rigoureuse des versions, une documentation claire et concise, ainsi qu'une communication fluide au sein de l'équipe.

1. Un jeu de plateforme est un genre de jeu vidéo où le joueur contrôle un personnage qui évolue à travers des niveaux constitués de plates-formes et d'obstacles. L'objectif principal est souvent de naviguer à travers ces niveaux en sautant, en évitant les dangers et en résolvant des énigmes pour atteindre un objectif final, comme sauver un personnage, collecter des objets ou vaincre un ennemi. Les jeux de plateforme sont souvent caractérisés par leur gameplay basé sur la précision des sauts et des mouvements du personnage principal.

2 Technologies utilisées

Au cours du développement du projet, nous avons fait appel à plusieurs technologies clés. Dans cette section, nous examinerons les outils que nous avons utilisés, en mettant l'accent sur les choix que nous avons faits et les alternatives envisagées.

2.1 C++

Le choix du langage C++ comme pilier de notre projet découle de sa performance et son contrôle précis sur la gestion de la mémoire. En tant que langage compilé, il nous permet de tirer pleinement parti des ressources système ce qui est essentiel dans un environnement aussi exigeant que celui des jeux vidéo en temps réel.

2.2 Threads

Dans un jeu, des éléments tels que le calcul du jeu, la gestion du réseau et la lecture de la musique doivent être exécutés en parallèle pour éviter tout ralentissement perceptible par le joueur.

Pour permettre cette simultanéité, nous avons utilisé des threads. Pendant que le thread principal se concentre sur le calcul des mouvements et des collisions, des threads secondaires sont dédiés à d'autres tâches critiques. Par exemple, un thread gère les interactions des joueurs sur le réseau tandis qu'un autre prend en charge la gestion des effets sonores, synchronisant ainsi la musique et les bruitages avec l'action du jeu.

Bien que l'utilisation de threads n'ait pas nécessité un choix entre plusieurs alternatives, il est important de souligner leur importance dans le développement de notre projet. Ce sont eux qui permettent une gestion efficace des différentes composantes du jeu.

2.3 SDL

SDL, ou Simple DirectMedia Layer, est une bibliothèque logicielle très populaire dans le développement de jeux vidéo. Elle est largement utilisée pour fournir les fonctionnalités de base nécessaires à la création de jeux, telles que la création de fenêtres, le rendu graphique, la gestion des événements d'entrée et la lecture audio.

Cette popularité de SDL s'explique en partie par sa grande communauté, qui travaille activement à l'extension de ses fonctionnalités. Parmi ces extensions, on trouve notamment :

- *SDL_mixer* : un module pour la lecture et la gestion des fichiers audio, permettant notamment la lecture de plusieurs pistes audio simultanément, le contrôle du volume et l'ajout d'effets sonores.
- *SDL_ttf* : un module pour le rendu de texte TrueType sur les surfaces SDL, offrant ainsi la possibilité d'afficher du texte à l'écran.
- *SDL_image* : un module pour le chargement et la manipulation de divers formats d'images (PNG, JPEG et GIF).

Dans notre projet, nous avons opté pour SDL en raison de plusieurs facteurs clés :

- Stabilité et maturité : SDL bénéficie d'une longue réputation de stabilité et de performance dans l'industrie du jeu vidéo.
- Large adoption dans l'industrie : Avec le soutien de grandes organisations telles que Valve, SDL est largement utilisé dans l'industrie du jeu. Cette adoption généralisée peut offrir un avantage en termes de ressources, de support et de compatibilité avec les différentes plateformes.
- Communauté : SDL bénéficie d'une documentation officielle assez complète et d'une communauté active sur les forums.

Il est important de noter que **SDL n'est pas un moteur de jeu complet**. Par exemple, prenons le jeu de pong classique : SDL serait utilisé pour gérer le rendu graphique des paddles, de la balle et de l'environnement de jeu, ainsi que la gestion des entrées clavier pour déplacer les paddles. Cependant, la logique du jeu elle-même, telle que la détection de collision entre la balle et les paddles, le déplacement de la balle en fonction des collisions et le suivi des scores, devrait être implémentée par le développeur.

Pour comprendre, l'algorithme 1 montre comment le jeu de Pong serait implémenté en utilisant SDL :

```
Initialiser SDL
Créer une fenêtre SDL
Créer des surfaces SDL pour les paddles, la balle et l'environnement de jeu

tant que le jeu n'est pas terminé faire
    pour chaque événement SDL faire
        si l'événement est une touche de clavier enfoncée alors
            Mettre à jour la position des paddles en fonction de la touche pressée
        si l'événement est de quitter le jeu alors
            Quitter la boucle principale

    si la balle touche un bord vertical alors
        Inverser sa direction verticale
    si la balle touche un paddle alors
        Inverser sa direction horizontale et augmenter sa vitesse

    Effacer l'écran
    Dessiner les paddles, la balle et l'environnement de jeu sur la fenêtre
    Rafraîchir l'affichage

Libérer les ressources SDL
```

Algorithme 1 : Algorithme du jeu de Pong avec SDL

2.4 Gestion de la communication réseau

Pour la communication entre les joueurs, nous avons étudié plusieurs possibilités, allant des sockets de bas niveau (TCP/UDP) aux solutions plus haut niveau comme HTTP et les WebSockets. Nous avons pris en compte l'efficacité de la communication, la flexibilité offerte pour créer un protocole personnalisé, et le degré de contrôle sur les échanges de données entre les joueurs pour orienter notre choix.

Finalement, nous avons choisi d'utiliser des sockets TCP/UDP, considérant qu'elles répondaient le mieux à nos besoins. Contrairement aux solutions plus haut niveau comme HTTP ou les WebSockets, les sockets offrent un contrôle direct sur la connexion réseau, nous permettant ainsi de maintenir une gestion fine des opérations réseau sans surcouche supplémentaire.

Cette décision découle de notre volonté de concevoir un système de communication personnalisé, parfaitement adapté aux exigences de notre jeu de plateforme collaboratif en ligne. En optant pour des sockets simples, nous avons pu développer un protocole de communication sur mesure, garantissant un contrôle granulaire sur la transmission des données entre les clients et le serveur.

Nous avons également envisagé SDL_Net, une bibliothèque spécialement conçue pour la gestion des communications réseau dans les jeux vidéo. Bien qu'elle puisse nous faire gagner du temps en offrant une solution pré-packagée pour nos besoins, son utilisation aurait aussi limité notre possibilité d'apprentissage et de personnalisation, ce qui était l'un des objectifs fondamentaux de notre projet.

3 Outils utilisés

Git

Git a joué un rôle central dans notre processus de développement collaboratif. Initialement, nous privilégions les fusions (merges). Cependant, suite à des problèmes rencontrés avec des fusions non maîtrisées, nous avons préféré utiliser les pull requests avec une politique stricte de validation avant fusion dans la branche principale. Cette politique demande trois validations pour chaque fusion, et nous avons également sécurisé la branche principale contre toute modification non autorisée.

Lien du projet github : https://github.com/Play-Together-dev/play-together

Cmake

Nous avons utilisé CMake pour la configuration de notre projet et la génération de nos fichiers de build. Nous avons personnalisé notre script CMake pour qu'il trouve automatiquement les bibliothèques SDL2 nécessaires (SDL, SDL_image, SDL_mixer, SDL_ttf), en utilisant les fichiers écrits par Amine Ben Hassouna et Eric Wing. Cette approche nous a permis de simplifier le processus de configuration et de garantir la portabilité de notre projet sur différentes plateformes.

SonarLint

SonarLint est un outil d'analyse de code statique qui identifie les problèmes de qualité du code en temps réel. Il fournit des suggestions d'amélioration pour maintenir un code propre et conforme aux bonnes pratiques de programmation. Nous avons pris en compte la plupart des avertissements générés par SonarLint et avons pris des mesures pour les corriger, ce qui a contribué à améliorer la qualité globale de notre code.

Doxygen

Doxygen est un outil de documentation de code source qui génère automatiquement une documentation à partir des commentaires dans le code source. Nous avons intégré Doxygen dans notre processus de développement pour faciliter la documentation de notre code. Bien que notre utilisation de cet outil soit restée simple, nous avons reconnu l'importance de maintenir une documentation claire et structurée, notamment pour les fichiers d'en-tête.

Clion

CLion a été notre principal environnement de développement pour la programmation en C++. Son intégration transparente avec CMake a simplifié notre flux de travail, nous permettant de générer et de gérer le projet CMake directement depuis l'IDE. En outre, la prise en charge native de Git dans CLion a facilité la gestion des branches et des pull-requests, contribuant ainsi à notre processus de développement collaboratif.

4 Développement du jeu

4.1 Game design

Que nous utilisions un moteur de jeu déjà conçu ou que nous créions le nôtre, le Game Design, processus de création des règles et des mécaniques de jeu, reste primordial peu importe son niveau de développement. Afin de nous plonger dans Play Together et de faciliter sa compréhension, nous allons brièvement aborder ses mécanismes ainsi que son univers artistique.

Avant de choisir les mécaniques du jeu, nous avons d'abord défini l'ambiance graphique de celui-ci. N'ayant aucun membre du groupe en mesure de faire des graphismes, nous avons choisi de prendre des ressources libres sur internet. Les personnages seront des petits dinosaures colorés, et le décor sera une grotte sombre, ce qui permettra de détacher facilement les personnages du décor. Ce dernier sera également séparé en plusieurs couches afin d'utiliser un effet de parallaxe [2]. En partant de cette ambiance, nous avons décidé que le thème du jeu sera sur les dinosaures qui doivent essayer de survivre à leur extinction.

FIGURE 2 – Ambiance graphique

Pour ce qui est du gameplay, nous avons repris les codes classiques d'un jeu de plateforme 2D : le joueur peut se déplacer, sauter, s'accroupir, attaquer, et récupérer des objets qui lui permettent de gagner de la vitesse ou de grandir sa taille temporairement. L'environnement est également assez classique, avec des plateformes mouvantes, des tapis roulants et des pièges qui peuvent écraser les joueurs.

Afin de pousser les joueurs à interagir entre eux nous avons choisi plusieurs mécaniques coopératives :
- Système de vie : lorsqu'un joueur meurt, il réapparaît sous forme d'œuf, les autres joueurs doivent alors frapper cet œuf pour briser sa coquille et le faire réapparaître.
- Système de score : les joueurs peuvent collecter des pièces qui leur feront gagner des points, le joueur ayant le plus de points arbore fièrement une couronne sur sa tête.
- Leviers : les leviers permettent de changer le comportement d'un mécanisme (arrêter une plateforme, changer le sens d'un tapis roulant, etc.), un joueur doit alors activer les leviers au bon moment pendant qu'un autre doit avancer dans le niveau.

L'enjeu du projet étant avant tout d'implémenter un moteur de jeu et toutes ses spécificités, nous concentrerons principalement notre analyse sur les points techniques fondamentaux d'un jeu de plateforme multijoueur. La prochaine section abordera les différentes étapes du processus de développement, en mettant en évidence les choix d'implémentation et les résultats obtenus.

2. La parallaxe est un effet visuel où les objets proches semblent se déplacer plus rapidement que les objets éloignés lorsque l'observateur change de perspective. Utilisé dans les jeux vidéo et les animations, il crée une sensation de profondeur et de perspective, ajoutant un réalisme visuel à l'environnement du jeu.

4.2 Architecture du projet

Menu et jeu

Dans notre architecture, nous avons deux éléments distincts : le menu et le jeu. Le menu représente l'état initial du programme, où le joueur peut choisir lancer le jeu. Une fois le jeu lancé, les joueurs entrent dans l'état de jeu où ils peuvent interagir avec l'environnement et jouer.

Les deux états peuvent être représentés par deux boucles tant que comme sur l'algorithme 2. Si le menu doit être affiché, la boucle se limite à traiter les événements SDL et à rendre le menu à l'écran. En revanche, lorsque le jeu démarre, la boucle exécute la fonction game.run(), qui lance une seconde boucle tant que, bloquant ainsi la boucle principale jusqu'à ce que le jeu soit terminé.

Le jeu est ensuite lui-même sous-divisé en plusieurs parties : la caméra, les joueurs, le niveau et les différents managers comme le montre la figure 4. Cette partie, comprenant le menu et le jeu avec ses composants, monopolisent le thread principal. Les autres parties du programme sont donc traitées dans des threads secondaires.

```
tant que le programme est lancé faire
    si le jeu ne doit pas être lancée alors
        - Gestion des entrées claviers et souris avec les événements SDL
        - Gestion des interactions avec les composants
        - Rendu graphique du menu avec SDL
    sinon
        tant que le jeu est lancée faire
            - Gestion des entrées claviers avec les événements SDL
            - Calcul des mouvements des joueurs
            - Calcul des mécaniques de jeu
            - Interpolation réseau des mouvements
            - Gestion des collisions
            - Mise à jour de la caméra
            - Mise à jour des textures et des animations
            - Rendu graphique du jeu avec SDL
```

Algorithme 2 : Boucles du menu et du jeu

Communication Réseau

La gestion des communications réseau est réalisée à l'aide de deux threads distincts : un pour le protocole TCP et un autre pour le protocole UDP. Chaque thread est responsable de l'établissement et de la maintenance des connexions réseau correspondantes. Un manager nommé "Network Manager" coordonne ces threads, démarrant et arrêtant les serveurs ou les clients TCP et UDP en fonction des besoins du jeu.

Console de Développement

La console de développement fonctionne dans un thread distinct, elle offre un moyen pratique de modifier le jeu pendant son exécution. Elle permet aux développeurs d'accéder à divers outils de débogage, de visualisation et de contrôle pendant le processus de développement.

Communication avec le Médiateur

Toutes ces parties communiquent entre elles grâce à un médiateur, qui est une classe statique[3]. Ce médiateur permet une communication bidirectionnelle entre le réseau, le menu et le jeu.
Par exemple voici, sur la figure 3, les étapes de communications entre les différents composants lorsque l'utilisateur clique sur le bouton "Rejoindre une partie" sur le menu. Comme on peut le voir, le menu appelle d'abord le médiateur, puis c'est ensuite le médiateur qui fait appel au menu.

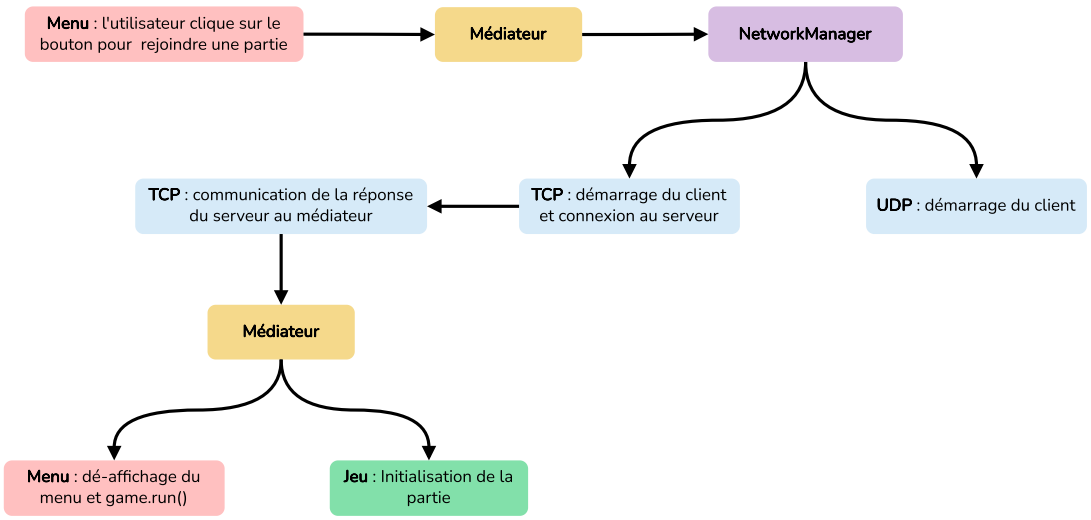


FIGURE 3 – Communication entre les éléments lorsque l'utilisateur rejoint une partie

Le fait d'utiliser un médiateur pour faciliter la communication entre les différentes parties du jeu présente deux avantages clés. Tout d'abord, cela simplifie la façon dont les différentes parties du jeu interagissent entre elles. Le médiateur agit comme un point central, permettant aux composants de communiquer entre eux sans avoir à se soucier des détails techniques des autres parties. Deuxièmement, cette approche rend le jeu plus flexible et facile à étendre. En passant par le médiateur, il est plus simple d'ajouter de nouvelles fonctionnalités ou de modifier celles déjà existantes sans perturber le fonctionnement des autres parties du jeu.

3. En programmation C++, une classe statique est une classe pour laquelle une seule instance est créée et partagée entre tous les objets de la classe. Cela permet un accès global à ses membres sans nécessiter d'instanciation spécifique

Pour mieux comprendre les différents composants du jeu ainsi que leurs interactions entre-eux vous pouvez vous référer à la figure 4. Les boites représentent une classe ou un ensemble de classe, et les flèches les inclusions simplifiées entre ces classes. Par exemple la Console est utilisé par le Main, et les Game Managers sont utilisés par la classe Game.

Pour une vue exhaustive de l'architecture du logiciel, vous pouvez vous référer à la figure 22, présente en annexe, où un diagramme de classe complet est présenté.

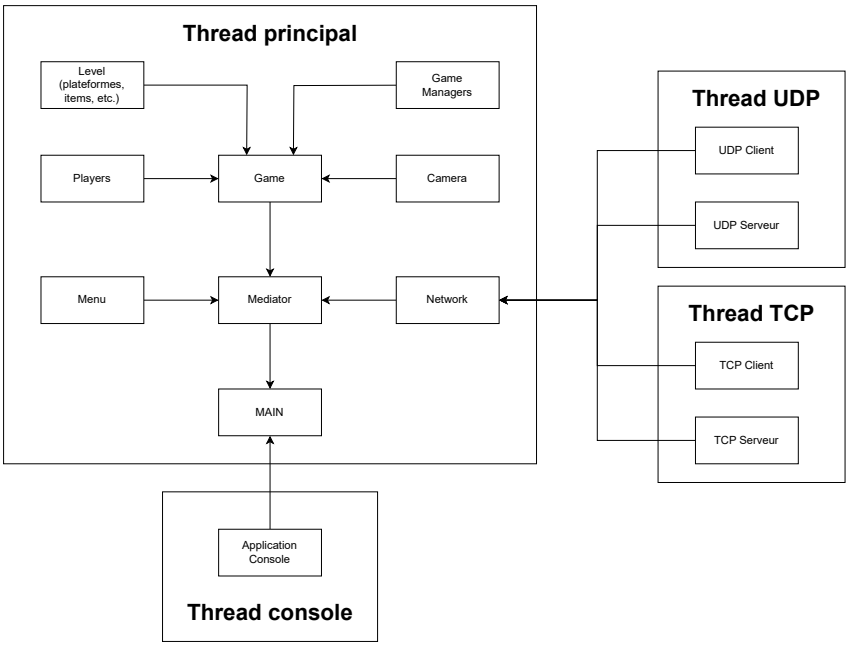


FIGURE 4 – Illustration de l'architecture simplifiée du projet

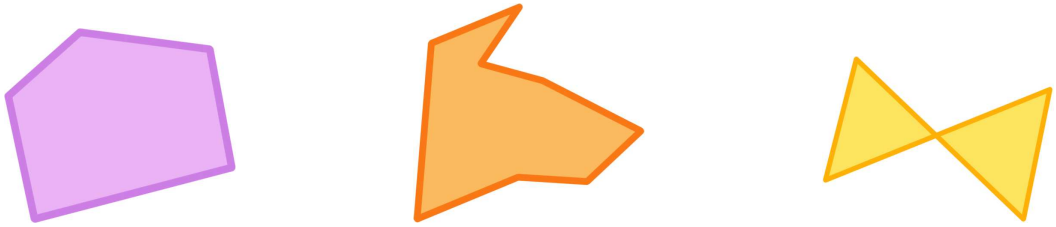
4.3 Gestion des collisions

Dans un jeu, les collisions déterminent si un personnage peut marcher sur le sol, traverser un mur ou entrer en contact avec d'autres objets. Pour que les collisions soient gérées de manière efficace, il faut choisir des formes appropriées pour représenter les objets du jeu et d'utiliser des algorithmes efficaces pour détecter et réagir aux collisions.

4.3.1 Choix des formes de collisions

Une solution pour créer le système de collisions serait d'opter pour des formes exclusivement constituées de rectangles, appelés AABB (Axis Aligned Bounding Box) ou de cercles.
Cependant, ce choix présente des limitations, notamment lors de la conception des niveaux, empêchant par exemple l'intégration de pentes. Ainsi, une méthode différente a été privilégiée pour la création des collisions.

Dans cette approche, une grande partie des formes du niveau dotées de collisions seront des polygones convexes. Un polygone convexe est défini comme un polygone non croisé, où toutes les diagonales se situent à l'intérieur du polygone. Une autre définition pratique serait que la somme de tous les angles intérieurs du polygone est inférieure à 180°.



(a) Polygone Convexe (b) Polygone Concave (c) Polygone Croisé

FIGURE 5 – Schéma des différents types de polygones

4.3.2 Algorithme de triangulation de l'oreille

Lors de la conception des niveaux, il est généralement préférable d'éviter l'utilisation de polygones concaves afin de simplifier le calcul des collisions. Cependant, si l'inclusion de polygones concaves devient nécessaire, la gestion de ces formes peut présenter des défis supplémentaires.

Dans de tels cas, avant la détection des collisions, il faudrait découper le polygone concave en plusieurs triangles, qui sont des formes convexes. L'algorithme de triangulation de l'oreille (Ear Clipping) s'avère être un choix adapté car il est efficace pour les polygones simples ou modérément complexes, notamment ceux comportant jusqu'à quelques dizaines de sommets, comme c'est le cas ici.
Cet algorithme identifie les sommets du polygone qui forment des triangles convexes avec leurs voisins, les "clippant" successivement jusqu'à ce que le polygone soit complètement triangulé.
La complexité de cet algorithme est en O(n^2).

Entrée : Un polygone P
Sortie : La liste des triangles de la triangulation

tant que P *a plus de 3 sommets* **faire**
    Trouver un sommet v de P qui forme une "oreille"
    Ajouter le triangle $v - v_{precedent} - v_{suivant}$ à la liste des triangles de la triangulation
    Supprimer v du polygone P

Ajouter le triangle formé par les trois sommets restants de à la liste des triangles
return *La liste des triangles de la triangulation*

Algorithme 3 : Algorithme d'Ear Clipping

Si on appliquait cet algorithme sur le polygone concave de la figure 5b, on obtiendrait 6 triangles. En effet, notre polygone est constitué de 8 sommets et passerait 5 fois dans le *tant que*. Voici, sur la figure 6, les étapes de l'algorithme :

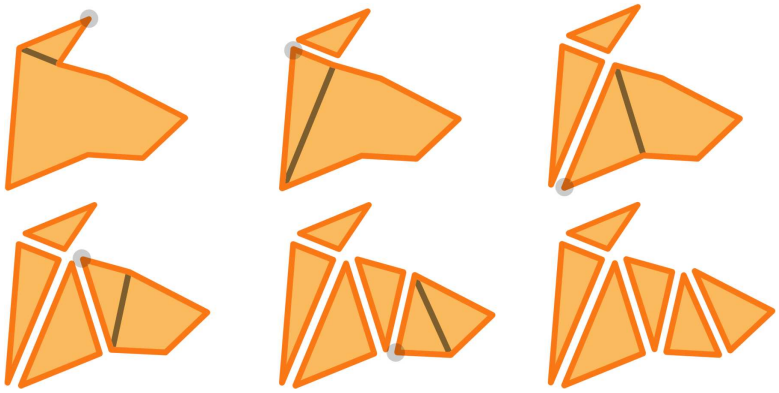


FIGURE 6 – Illustration de l'algorithme de triangulation de l'oreille

4.3.3 Gestion des collisions

On part maintenant du principe que toutes les formes de collision dans le niveau seront des polygones convexes, l'étape suivante consiste à mettre en œuvre un système de collisions basé sur ces polygones. Pour cela, on va utiliser une méthode appelée la méthode des axes de séparation (SAT - Separating Axis Theorem).

La méthode des axes de séparation est une technique de détection de collision qui part du principe suivant : deux objets ne peuvent pas être en collision si et seulement s'il existe un axe tel que leur projection ne se chevauche pas. Un axe est une ligne imaginaire perpendiculaire à un côté du polygone. Donc concrètement, pour chaque polygone, on regarde tous les axes formés par ses côtés. On projette les sommets des polygones sur ces axes, on obtient des intervalles de projection. Si les intervalles de projection des deux polygones se chevauchent sur tous les axes, alors il y a une collision entre les polygones. Sinon il n'y a pas de collision entre les polygones et une ligne de séparation peut être tracée.

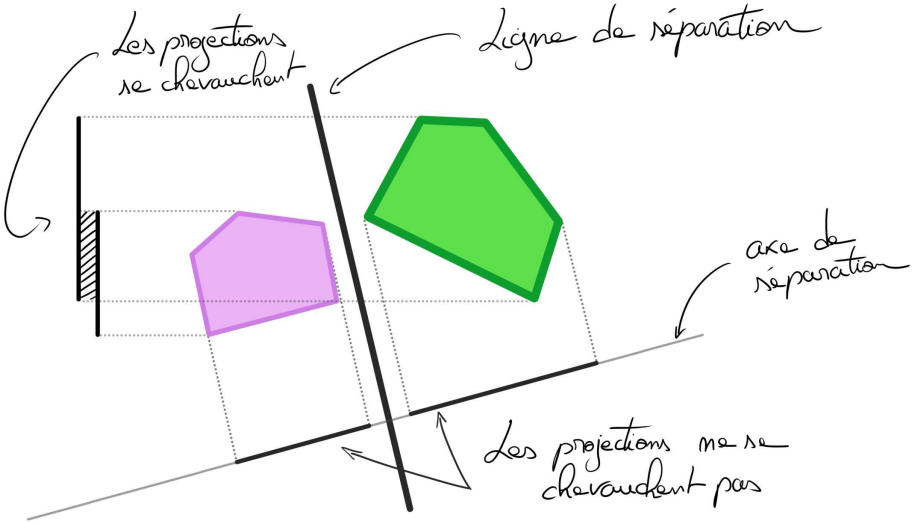


FIGURE 7 – Illustration de l'algorithme de l'algorithme SAT (Separating Axis Theorem)

Cependant, cette méthode ne peut être utilisée que sur des polygones convexes. En effet, il peut arriver qu'un polygone convexe ne soit pas en collision avec un polygone concave, même s'il n'y a pas de ligne de séparation évidente entre eux comme le montre la figure 8b.

(a) Deux polygones convexes

(b) Un polygone convexe et un polygone concave

FIGURE 8 – Séparation de deux polygones avec le SAT

Voici comment nous avons implémenté la méthode des axes de séparation :

1. **Vérification de la convexité de l'obstacle :**
   Avant d'appliquer la méthode des axes de séparation, le programme vérifie si l'obstacle est convexe. Cela est important car la méthode des axes de séparation ne s'applique correctement qu'aux polygones convexes (voir figure 8a).

   Pour déterminer si un polygone est convexe, le programme calcule la somme des angles intérieurs du polygone.
   Si la somme des angles intérieurs est égale à $(n-2) \times 180$ degrés, où n est le nombre de sommets du polygone, alors le polygone est considéré comme convexe.

2. **Projection sur les axes :**
   Pour chaque obstacle (polygone) et le joueur, le programme considère plusieurs axes. Ces axes sont des lignes imaginaires perpendiculaires aux côtés des objets. On projette ensuite les sommets du joueur et de l'obstacle sur ces axes. La projection d'un point sur un axe est la coordonnée du point le long de cet axe.

3. **Vérification des chevauchements de projection :** Si les projections du joueur et de l'obstacle sur tous les axes ne se chevauchent pas, alors il n'y a pas de collision. Sinon, une collision est détectée.

Complexité :

La complexité temporelle de cet algorithme de détection de collision dépend principalement du nombre d'obstacles (k) et du nombre total de sommets dans ces obstacles (m).

La première étape consiste à vérifier la convexité de chaque obstacle, ce qui nécessite un temps proportionnel au nombre de sommets de chaque obstacle ($O(n)$). Supposons qu'il y ait k obstacles, la complexité totale pour cette étape serait $O(k \times n)$.

Ensuite, pour chaque paire formée par le joueur et un obstacle, l'algorithme considère plusieurs axes, le nombre d'axes étant égal à la somme des nombres de côtés des deux polygones. Il projette ensuite les sommets du joueur et de l'obstacle sur ces axes. La complexité de cette étape est également proportionnelle au nombre total de sommets dans tous les obstacles ($O(m)$). Si on a k obstacles, la complexité totale pour cette étape est $O(k \times m)$.

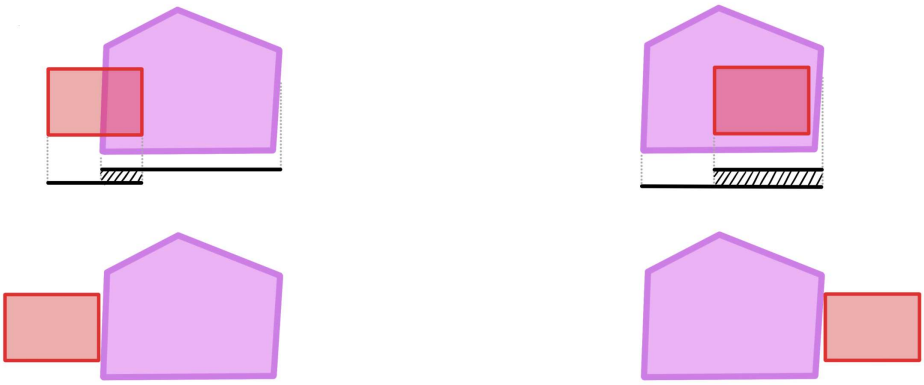
Finalement, la complexité temporelle globale de l'algorithme est approximativement $O(k \times n + k \times m)$. En pratique, la complexité dépend également du nombre d'itérations dans la boucle principale, mais dans le pire des cas, elle reste dominée par ces deux étapes. Ainsi, la performance de l'algorithme dépendra principalement du nombre d'obstacles et de la complexité de leurs formes géométriques.

4.3.4 Correction des collisions

Maintenant que nous savons comment détecter une collision entre deux polygones, il nous faut une réaction à cette collision.

Une première solution possible consiste à calculer la profondeur de pénétration entre les deux objets. Pour ce faire, nous ajoutons une variable dans l'algorithme du SAT qui enregistre l'axe de séparation le plus petit, cet axe sera la profondeur. Ensuite, cette valeur de pénétration est ajoutée à l'objet en collision, le faisant ainsi sortir du polygone et le plaçant juste au bord, comme souhaité.

Cette solution pose néanmoins un problème important : si la pénétration de l'objet est trop profonde, il risque de sortir de l'autre côté du polygone, et ce n'est absolument pas ce que l'on veut !



(a) Avant et après une pénétration correcte

(b) Avant et après une pénétration trop profonde

FIGURE 9 – Illustration de la correction avec le vecteur de pénétration

L'approche que nous avons choisie est de corriger nous même la collision en déplaçant le joueur petit à petit jusqu'à le sortir du polygone. L'algorithme va ajuster la position du joueur par dichotomie, jusqu'à être à 0.1 pixel du bord du polygone.

Entrée : Un joueur J et un polygone P

min = position de J avant son déplacement
max = position de J actuelle

tant que J + *move est à plus de 0.1 pixel du bord de* P **faire**
    move = (min + max) / 2
    si *une collision est détectée entre* J *et* P **alors**
        max = move
    sinon
        min = move

Appliquer la position *min* à J

Algorithme 4 : Algorithme de correction d'une collision

4.4 Gestion de la caméra

4.4.1 Choix du type de caméra

Une caméra dans un jeu vidéo représente le point de vue du joueur sur l'environnement du jeu, elle est responsable de ce que le joueur voit à l'écran. La caméra peut être fixe, suivre le personnage principal, être contrôlée par le joueur lui-même, ou même adopter différents angles et perspectives selon les besoins du jeu. Avant de s'aventurer sur la façon dont nous avons implémenté la caméra dans notre jeu, nous devons définir le comportement de celle-ci.

Les caméras peuvent se distinguer en deux catégories :
— Vue à la première personne : le point de vue se situe à l'intérieur du personnage, le joueur voit donc ce que son personnage voit, un peu comme si son personnage portait une GoPro sur la tête (figure 10a)
— Vue à la troisième personne : le point de vue se situe à l'extérieur du personnage, le joueur voit donc le corps de son personnage et le décor autour, un peu comme un drone qui observe l'action, ici la caméra peut se déplacer indépendamment du personnage (figure 10b)

(a) Vue à la première personne

(b) Vue à la troisième personne

FIGURE 10 – Illustration des deux points de vue [10]

Afin d'appuyer l'aspect collaboratif du jeu nous souhaitons que tous les joueurs voient la même chose à leur écran, autrement dit, cela signifie qu'il n'y a qu'une seule caméra partagée par tous les joueurs. Nous voulons également que tous les personnages soient constamment affichés à l'écran et que lorsqu'un joueur sort de celui-ci, il meurt. Nous avons donc fait le choix d'une caméra à la troisième personne, qui suivra l'ensemble des personnages à l'écran.

4.4.2 Gestion de l'affichage

Avant de commencer, il est important de clarifier la distinction entre l'écran et la caméra. L'écran représente ce que le joueur voit, c'est le rendu graphique du jeu. On peut le définir comme la zone d'affichage, qui ne pourra jamais se déplacer et se trouvera toujours en position $(0, 0)$. En revanche, la caméra est une entité mobile dans le monde du jeu.

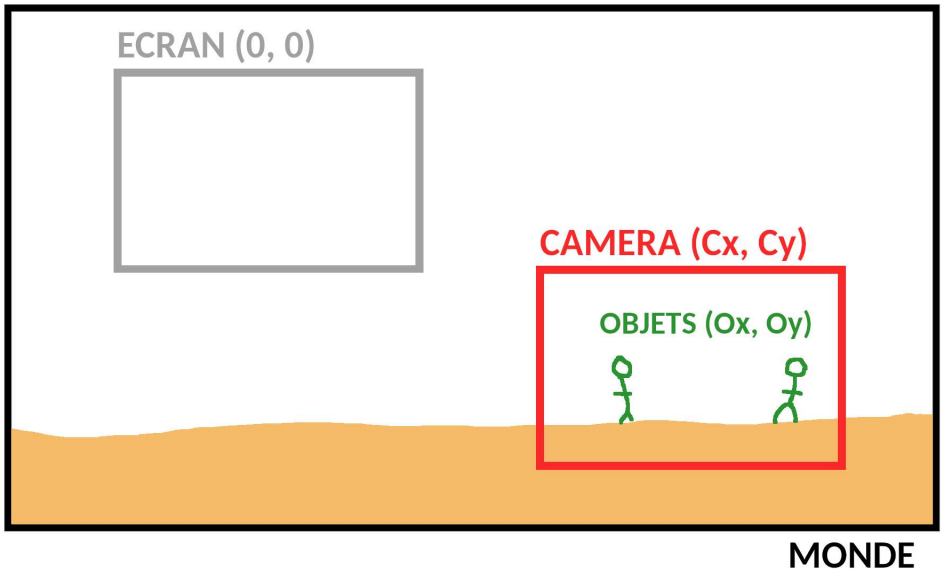


FIGURE 11 – Différence entre écran et caméra

La solution pour gérer un système de caméra est de calculer la position des objets dans la zone d'affichage selon la position de la caméra. Tous les objets étant fixe dans la scène, il est possible de calculer cette position d'affichage en soustrayant leur positions avec la position de la caméra, autrement dit, nous avons la formule suivante : $ecran = monde - camera$.

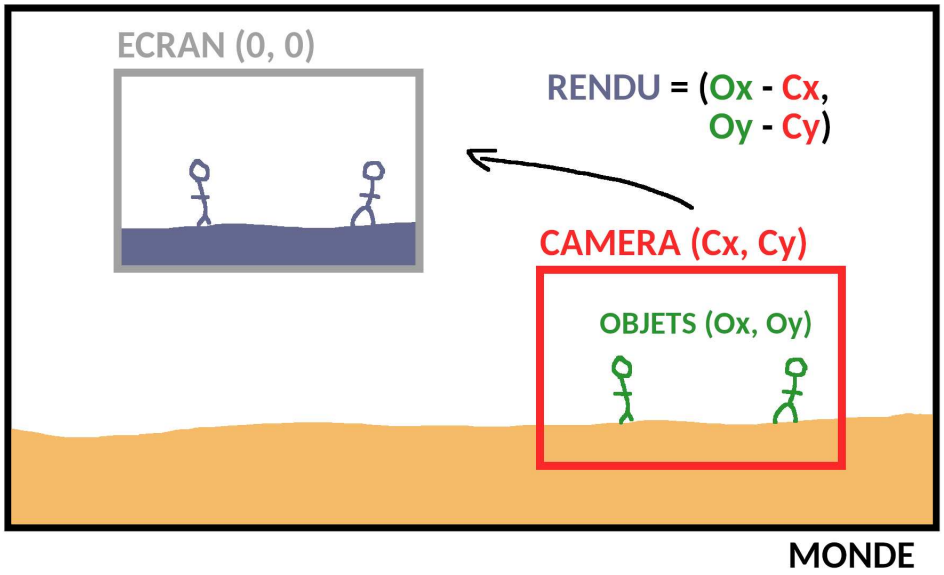


FIGURE 12 – Calcul des positions pour l'affichage

4.4.3 Mouvements de la caméra

De Mario à Celeste en passant par Metroid, la caméra a toujours été un sujet important dans les jeux de plateformes 2D. Les mouvements de celle-ci est un choix de game design important lors de la conception d'un jeu, et ces mouvements qui vont influencer la façon dont les joueurs appréhendent et interagissent avec le monde.

L'aspect coopératif étant une caractéristique importante de notre jeu, nous avons opté pour un système où la caméra se centre sur un point moyen représentant la position collective des joueurs. Cela permet de pousser les joueurs à coopérer pour déplacer la caméra où ils le souhaitent.

Afin d'éviter des mouvements excessifs de la caméra dus aux déplacements des joueurs, nous avons introduit une zone de "stabilité visuelle". Cette zone, représentée dans la figure 13, est un rectangle positionné au centre gauche de l'écran de la caméra. Lorsque la caméra suit le point moyen des joueurs, elle vérifie si ce point se situe à l'intérieur de cette zone. Si ce n'est pas le cas, la caméra ajuste sa position pour que le point moyen se trouve au bord de la zone. Cette technique permet d'assurer une transition fluide et d'éviter les mouvements brusques de la caméra à chaque petit déplacement des joueurs.

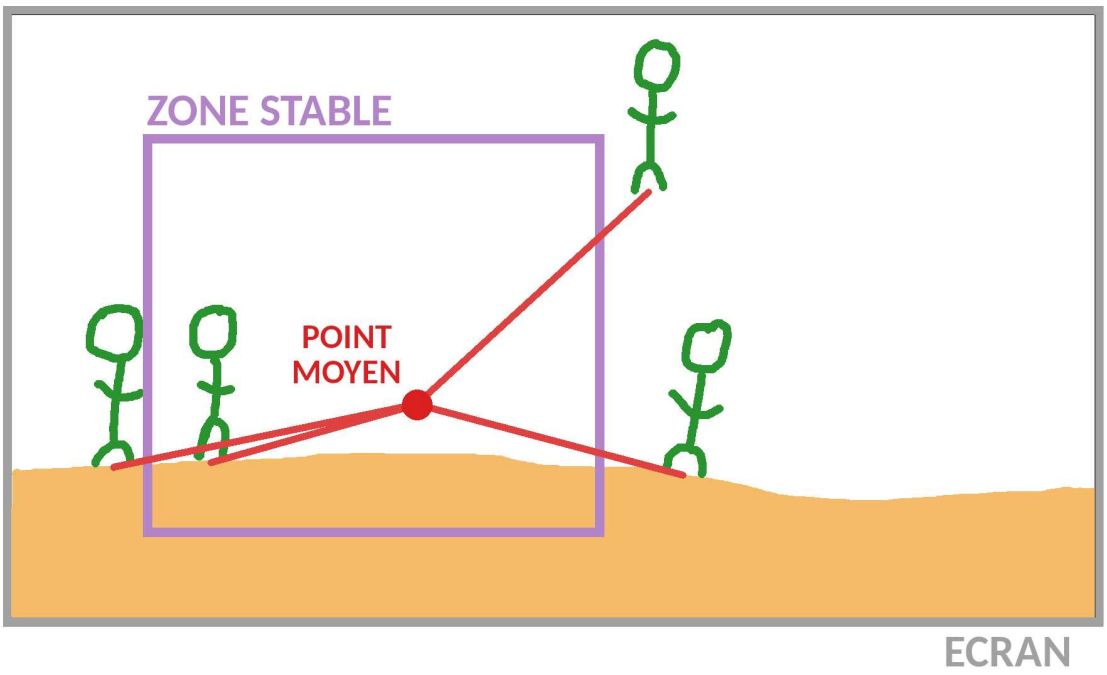


FIGURE 13 – Zone de "stabilité visuelle"

4.5 Architecture réseau hybride TCP/UDP

Lorsque le multijoueur est impliqué, le réseau est incontournable ! En C, deux protocoles de communication servent des besoins spécifiques :

— Le TCP, orienté connexion, assure la fiabilité et l'ordre des échanges entre les entités du réseau. Il est utilisé principalement pour des fonctionnalités nécessitant une transmission fiable et séquentielle, comme la synchronisation du jeu entre serveur et clients.

— L'UDP, sans connexion, offre une communication plus légère et rapide, mais sans garantie de fiabilité ou d'ordre des données.

Lors du développement de l'application, nous avons pris la décision d'utiliser une approche combinant les protocoles TCP et UDP pour la communication réseau. Cette décision découle de la nécessité de trouver un équilibre entre la fiabilité et la robustesse assurées par TCP, et les performances en temps réel offertes par UDP.

Cependant, l'utilisation conjointe de TCP et d'UDP peut potentiellement entraîner des problèmes de performance, comme le souligne l'étude menée par Sawashima, Hori, Sunahara et Oie, intitulée "Characteristics of UDP Packet Loss : Effect of TCP Traffic"[11]. Lorsque nous partageons la bande passante entre ces deux protocoles, des goulets d'étranglement peuvent se former, ce qui peut entraîner des retards de transmission, des pertes de paquets et des fluctuations de la latence.

Bien que notre approche actuelle soit fonctionnelle pour le jeu à quatre joueurs, on peut noter que les problèmes de performance liés à l'utilisation de TCP et d'UDP peuvent devenir plus apparents dans certaines situations, telles que des connexions Internet instables ou des conditions de réseau chargées.

Bien que notre choix de combiner TCP et UDP soit approprié pour les besoins immédiats du jeu, il existe des alternatives à considérer. Par exemple, une architecture entièrement basée sur UDP pourrait offrir des performances plus constantes et une latence plus faible dans un environnement de jeu à quatre joueurs. Cependant, une telle approche aurait nécessité la mise en place d'un système complet de gestion de la connexion, de la fiabilité des données, de l'ordonnancement et de l'évitement de la congestion[14][15], augmentant ainsi considérablement la charge de développement et dépassant sûrement les contraintes de temps du projet.

4.5.1 Serveur : Gestion des connexions et des échanges de données

Le serveur TCP constitue le point d'entrée fondamental de la connexion dans notre architecture multijoueur. Lorsqu'un joueur-client cherche à établir une connexion avec un joueur-serveur, une requête TCP est émise. Une fois cette connexion établie, le serveur TCP génère un nouveau thread dédié pour traiter les échanges de données avec ce client. En parallèle, il stocke le descripteur de fichier associé à l'adresse du client, permettant ainsi d'identifier les messages reçus en UDP et de les attribuer au bon destinataire.

Les données reçues d'un client via TCP sont traitées en fonction des besoins de l'application, que ce soit des informations de connexion, d'actions, pour mettre à jour l'état du jeu. De même, lorsqu'un client se déconnecte, le serveur gère proprement cette déconnexion, libérant les ressources associées.

De manière similaire, le serveur UDP contribue à la gestion des échanges de données en offrant une latence réduite, idéale pour les communications en temps réel. Écoutant les messages entrants sur un port spécifique, il les traite en fonction de leur contenu, comme les mises à jour de position des joueurs, diffusées à tous les participants dans la même session de jeu.

Le serveur est capable d'envoyer, de recevoir et de diffuser des messages dans les deux protocoles, chaque client étant identifié de manière adéquate.

4.5.2 Client : Interaction avec le serveur et échanges des données

Les clients TCP et UDP jouent un rôle essentiel dans l'interactivité des joueurs avec le serveur dans notre application multijoueur.

Le client TCP est responsable d'établir une connexion avec le serveur et de transmettre les actions du joueur. Lorsqu'un joueur effectue une action jugée importante, le client TCP envoie ces données au serveur via la connexion établie. Le serveur traite ensuite ces données et met à jour l'état du jeu en conséquence, assurant ainsi une expérience cohérente pour tous les joueurs connectés.

En parallèle, le client UDP est utilisé pour les communications en temps réel, telles que les mises à jour de position des joueurs. Tout au long de la partie, le client UDP envoie à chaque frame un message contenant l'état actuel du clavier de l'utilisateur via des datagrammes UDP. Ces messages sont ensuite diffusés par le serveur à tous les autres joueurs dans la même session de jeu, leur permettant de voir en temps réel les mouvements des autres participants.

4.6 Optimisation des ressources

4.6.1 Limitation du frame rate

Comme montré plus tôt, un jeu fonctionne avec une boucle tant que. Si cette boucle n'est pas limitée en vitesse, le processus du jeu va continuer à s'exécuter aussi vite que possible, ce qui peut entraîner une consommation excessive des ressources disponibles. Afin d'éviter cela, il est essentiel de limiter le nombre de tours de boucles par seconde, c'est-à-dire le nombre d'images par seconde (fps).

Prenons une seconde du jeu et imaginons-la comme une clôture (figure 14), où chaque lame représente une image du jeu. Dans un état normal, c'est-à-dire calculé par un ordinateur assez performant, la clôture aurait 30 lames (30 fps).

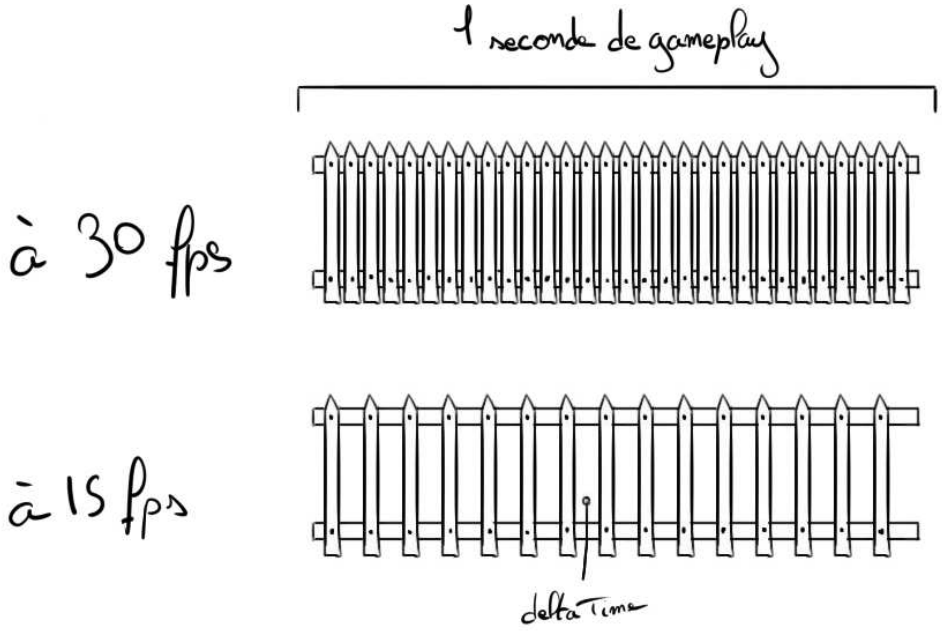


FIGURE 14 – Explication du delta time avec une clôture

Mais si l'ordinateur est plus lent, il y a moins de lames dans la clôture. L'espace entre les lames est donc plus grand. Cet espace entre les lames est le Delta Time.

Prenons l'exemple de deux joueurs : Alice et Bob. Alice joue sur un ordinateur haut de gamme qui peut facilement maintenir le jeu à 30 fps. Cependant, Bob joue sur un ordinateur plus ancien et moins performant qui peine à maintenir le jeu à 15 fps. Dans le jeu, le personnage se déplace de 1 pixel à chaque tour de boucle. Donc, pour se déplacer d'un point A à un point B, un personnage a besoin de 30 images. Sur l'ordinateur d'Alice, une seconde suffira. Par contre, sur l'ordinateur de Bob, il en faudra deux. C'est évidemment inacceptable, surtout dans un jeu multijoueur.

Pour corriger ce problème, il faut multiplier le mouvement du joueur par le Delta Time. En faisant cela, on s'assure que tous les joueurs atteignent le point B en même temps quelque soit la vitesse de l'ordinateur.
En effet, plus la vitesse d'exécution de l'ordinateur est lente, plus le Delta Time sera grand et plus le mouvement du joueur sera grand pour combler le manque d'images. Ainsi, le joueur de Bob aura une vitesse de 2px par image et le joueur d'Alice de 1px par image.

C'est exactement ce qui se passait dans Doom en 1993. Le jeu était programmé pour s'exécuter à une vitesse fixe, mais sur des machines plus puissantes, il pouvait tourner beaucoup plus rapidement. Cela donnait un avantage considérable aux joueurs sur ces machines, car leur personnage se déplaçait plus vite et était donc plus difficile à toucher.

4.6.1.1 Implémentation de la limitation du framerate avec le Delta Time

Voyons comment nous pouvons implémenter le Delta Time avec SDL.
Dans le jeu, le nombre cible de frames par seconde (fps) est défini au lancement du jeu et correspond au taux de rafraîchissement de l'écran de l'utilisateur.
À chaque itération de la boucle principale du jeu, si le temps écoulé depuis le début de la boucle est inférieur au temps nécessaire pour atteindre le prochain rafraîchissement de l'écran, le jeu attend le temps restant en utilisant la fonction `SDL_Delay()`.

De plus, le Delta Time calculé est utilisé pour calculer les variations temporelles entre deux frames. En multipliant les mouvements des objets par le Delta Time, le jeu s'adapte automatiquement aux performances de l'ordinateur de l'utilisateur. Ainsi, même si le temps entre deux frames varie en raison des performances matérielles, les mouvements des objets restent fluides et cohérents pour tous les joueurs, quel que soit leur matériel.

```
// Obtention des informations sur l'écran
SDL_DisplayMode displayMode;
SDL_GetCurrentDisplayMode(0, &displayMode);
int refreshRate = displayMode.refresh_rate;

// Boucle principale du jeu
bool running = true;
Uint32 lastFrameTime = SDL_GetTicks();

while (running) {
    // Calcul du deltaTime
    Uint32 currentFrameTime = SDL_GetTicks();
    float deltaTime = (currentFrameTime - lastFrameTime) / 1000.0f; // Conversion en secondes
    lastFrameTime = currentFrameTime;

    // Gestion des événements SDL
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        // Traitement des entrées clavier
        // Traitement des mouvements de la souris
    }

    // Mise à jour du jeu en fonction du deltaTime
    updateGame(deltaTime);

    // Affichage du jeu
    renderGame();

    // Limite de framerate au taux de rafraîchissement de l'écran
    Uint32 frameTicks = SDL_GetTicks() - currentFrameTime;
    if (frameTicks < 1000 / refreshRate) {
        SDL_Delay((1000 / refreshRate) - frameTicks);
    }
}
```

FIGURE 15 – Implémentation de la limitation du framerate avec le Delta Time

4.6.2 Réduction du temps de calcul

Maintenant que nous avons optimisé la boucle du jeu, essayons d'aller encore plus loin en s'intéressant au contenu de la boucle. Actuellement le jeu calcule l'entièreté des éléments à chaque frame, ce qui signifie que même lorsque les joueurs sont à la fin d'un niveau, les collisions, le rendu graphique et le mouvement des éléments du début du niveau sont calculés.

Pour optimiser le système de collision il faut séparer la détection en deux parties distinctes :
— La Phase Large (broad phase) : on utilise des algorithmes peu coûteux et on identifie les objets qui pourraient potentiellement collisioner entre eux.
— La Phase Étroite (narrow phase) : on part des collisions détectées lors de la phase large et, avec des algorithmes plus coûteux, on vérifie si les éléments entrent réellement en collision.

Le fait de séparer la détection en deux phases permet de déterminer rapidement et à moindre coût si deux éléments peuvent se collisionner avant d'appliquer des algorithmes plus coûteux pour déterminer les collisions réelles. Une des méthodes les plus utilisées est d'entourer les polygones avec une forme plus simple, telle que des AABB [4], pour détecter les collisions. Ainsi, lors de la phase large, des algorithmes de détection d'AABB, réputés rapides et efficaces, sont appliqués en premier à tous les objets.

Dans la phase suivante, la phase étroite, le polygone lui-même sera utilisé. Des algorithmes plus complexes, tels que le SAT, seront employés, mais uniquement sur les objets détectés lors de la phase large, réduisant ainsi le nombre d'objets concernés.

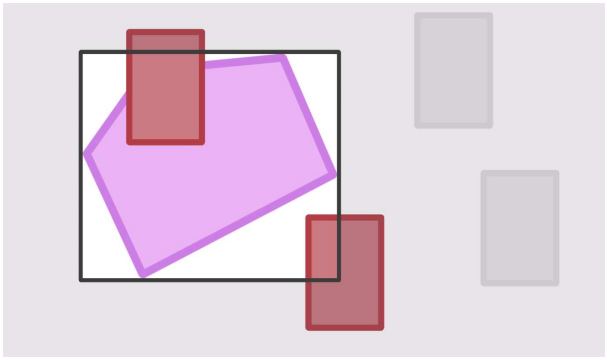


FIGURE 16 – Détection des collisions potentielles lors de la phase large

Dans notre jeu, seuls les joueurs peuvent entrer en collision avec d'autres éléments, et ceux-ci sont toujours visibles à l'écran. Par conséquent, il est inutile de vérifier les collisions en dehors de la zone d'affichage. Nous utilisons donc une version simplifiée de la phase large : au lieu de vérifier si un polygone peut potentiellement collisionner avec un joueur, on vérifie uniquement s'il est affiché à l'écran.

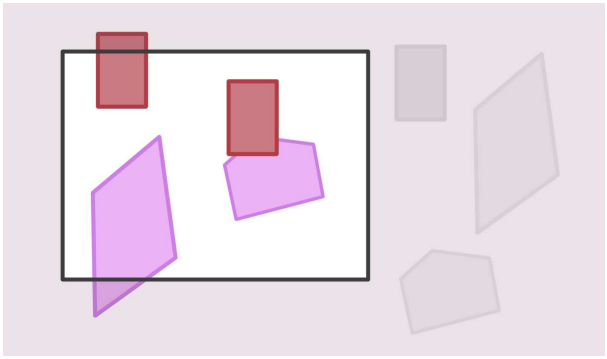


FIGURE 17 – Détection des collisions avec l'écran lors de la phase large

Pour ce qui est du rendu graphique et du calcul des mécaniques de jeu, la même méthode est appliquée : si un objet est affiché à l'écran il est pris en compte ; sinon il est ignoré.

4. Les Axis Aligned Bouding Box, traduit par Boîte de Délimitation Alignée sur l'Axe, sont des formes rectangulaires qui sont alignées avec les axes du système de coordonnées.

5 Gestion du projet

Avant de démarrer le projet, nous avons rédigé un document détaillant les normes à respecter pour Git et le code. Ensuite, nous avons découpé le projet en principales tâches, telles qu'illustrées dans la figure 18.

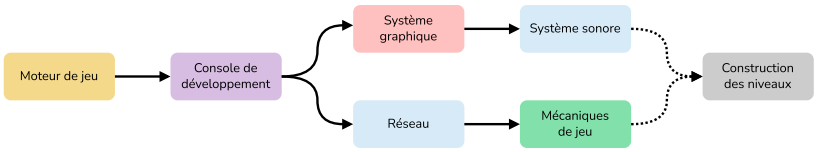


FIGURE 18 – Diagramme simplifié des tâches du projet

Nous avons ensuite subdivisé ces tâches en sous-tâches, comme montré dans la figure 19. Malheureusement, malgré ces préparatifs, toutes les étapes du projet n'ont pas été achevées dans les délais impartis.

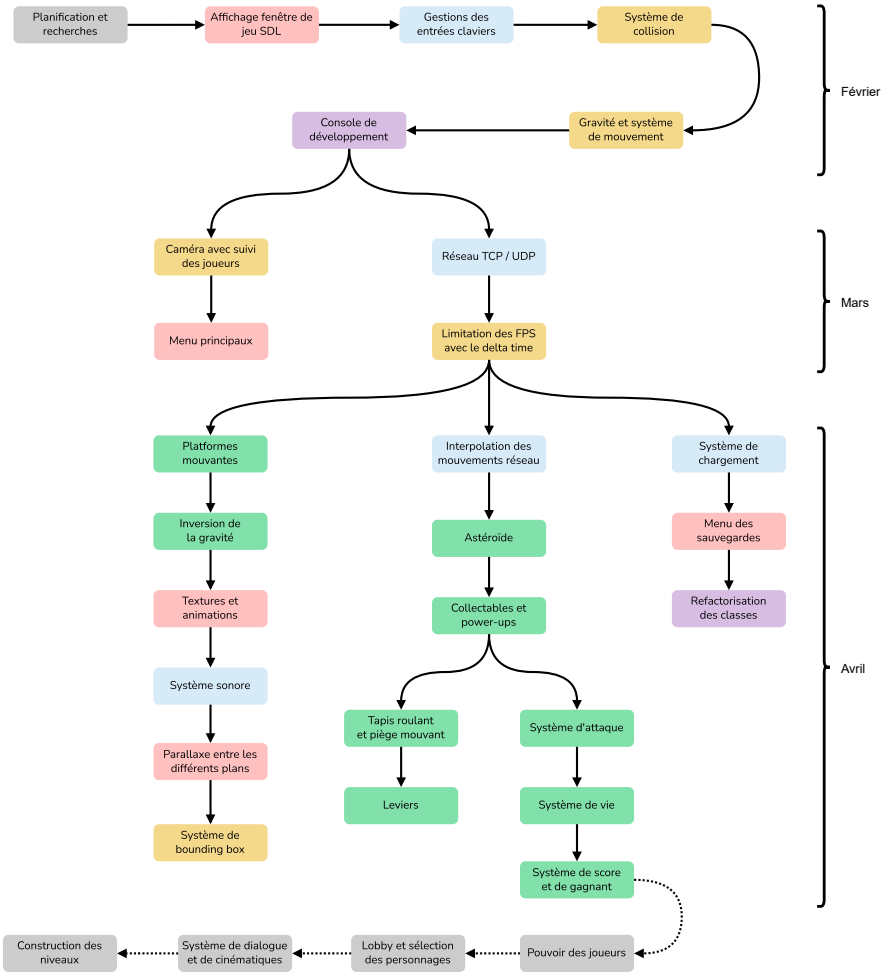


FIGURE 19 – Diagramme des tâches du projet

6 Conclusion

En résumé, ce projet de développement de jeu vidéo a été une véritable aventure pour notre équipe. Concevoir un moteur de jeu à partir de zéro a représenté un défi de taille. Nous avons plongé dans le monde complexe de la création de jeux, ce qui nous a permis de progresser individuellement et collectivement. Bien sûr, nous avons rencontré des difficultés et nous avons dû surmonter des obstacles internes qui ont testé notre capacité à travailler en équipe. Mais au final, le résultat en valait la peine.

Ce qui est remarquable, c'est que nous avons tous énormément appris. Pour la plupart d'entre nous, ce projet était notre première expérience sérieuse avec le langage C++ (et C), et nous avons été gratifiés de constater nos progrès au fil du temps. Certains d'entre nous ont également découvert les bases de la programmation orientée objet, appliquant les concepts appris lors de nos cours précédents à la faculté. Cette expérience nous a permis de consolider nos connaissances et de découvrir de nouvelles perspectives auxquelles nous n'avions pas pensé auparavant. Nous avons également découvert des aspects fascinants de la programmation de jeux vidéo, notamment des algorithmes comme le SAT. En revanche, avec le recul, nous aurions peut-être évité d'utiliser le SAT, préférant explorer des systèmes de physique avec des rigid bodies pour simuler des interactions plus réalistes entre les objets. Cependant, à ce stade, il était trop tard pour apporter des modifications majeures à notre moteur de jeu.

Cette expérience nous a ouvert de nouvelles perspectives sur le langage et nous sommes fiers de nos réalisations malgré les défis rencontrés. Nous sommes désormais impatients de voir où cette expérience nous mènera et nous sommes prêts à relever de nouveaux défis avec enthousiasme.

Références

Gestion des collisions

[1] Demers Jean-Christophe (2002). *Décomposition d'un polygone concave en polygones convexes. École de technologie supérieure*. Consulté le 2 février 2024, de https://cours.etsmtl.ca/gpa665/Cours/Laboratoires/GPA665_Lab3_DecompositionDUnPolygonConcaveEnPolygonesConvexes.pdf

[2] Fvirtman (26 nov. 2013). *Théorie des collisions*. Consulté le 27 janvier 2024, de https://jeux.developpez.com/tutoriels/theorie-des-collisions/formes-complexes/

[3] *Loi des cosinus*. Wikipédia. Consulté le 2 février 2024, de https://fr.wikipedia.org/wiki/Loi_des_cosinus

[4] *Polygone*. Wikipédia. Consulté le 2 février 2024, de https://fr.wikipedia.org/wiki/Polygone

[5] *Polygone triangulation*. Wikipédia. Consulté le 2 février 2024, de https://en.wikipedia.org/wiki/Polygon_triangulation

[6] *Séparation des convexes*. Wikipédia. Consulté le 3 février 2024, de https://fr.wikipedia.org/wiki/SÃl'paration_des_convexes

Gestion de la caméra

[7] Birch Carl (18 dec. 2017). *How To Make A Game #20 : 2D Player Follow Camera in C++ And SDL2 Tutorial* [Vidéo]. Let's Make Games, Youtube. https://www.youtube.com/watch?v=QeN1ygJD5y4

[8] (2016). *How to use the SDL viewport properly ?*. Game Development Stack Exchange. Consulté le 10 février 2024, de https://gamedev.stackexchange.com/questions/121421/how-to-use-the-sdl-viewport-properly

[9] Keren Itay (9 nov. 2015). *How Cameras In Side-Scrollers Work* [Vidéo]. Let's Make Games, Youtube. https://www.youtube.com/watch?v=pdvCO97jOQk

[10] *Vue à la première personne*. Wikipédia. Consulté le 5 mai 2024, de https://fr.wikipedia.org/wiki/Vue_%C3%A0_la_premi%C3%A8re_personne

Gestion du réseau

[11] Hori Yoshiaki, Sawashima Hidenari, Sunahara Hideki (juillet 1997). *Characteristics of UDP Packet Loss : Effect of TCP Traffic*. Proceedings of INET'97, Engineering 3-1. Consulté le 28 février 2024, de https://www.isoc.org/INET97/proceedings/F3/F3_1.HTM

[12] Glenn Fiedler (1 oct. 2008). *UDP vs. TCP*. Gaffer On Games. Consulté le 28 février 2024, de https://gafferongames.com/post/udp_vs_tcp/

[13] Glenn Fiedler (3 oct. 2008). *Sending and Receiving Packets*. Gaffer On Games. Consulté le 28 février 2024, de https://gafferongames.com/post/sending_and_receiving_packets/

[14] Glenn Fiedler (8 oct. 2008). *Virtual Connection over UDP*. Gaffer On Games. Consulté le 28 février 2024, de https://gafferongames.com/post/virtual_connection_over_udp/

[15] Glenn Fiedler (20 oct. 2008). *Reliability and Congestion Avoidance over UDP*. Gaffer On Games. Consulté le 28 février 2024, de https://gafferongames.com/post/reliability_ordering_and_congestion_avoidance_over_udp/

[16] Glenn Fiedler (24 fev. 2010). *What Every Programmer Needs To Know About Game Networking*. Gaffer On Games. Consulté le 29 février 2024, de https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/

Optimisation des ressources

[17] Tyroller Jonas (30 juin 2021). *Dear Game Developers, Stop Messing This Up !* [Vidéo]. Let's Make Games, Youtube. https://www.youtube.com/watch?v=yGhfUcPjXuE

7 Annexes

FIGURE 20 – Menu principal du jeu

FIGURE 21 – Menu sélection/création de la partie

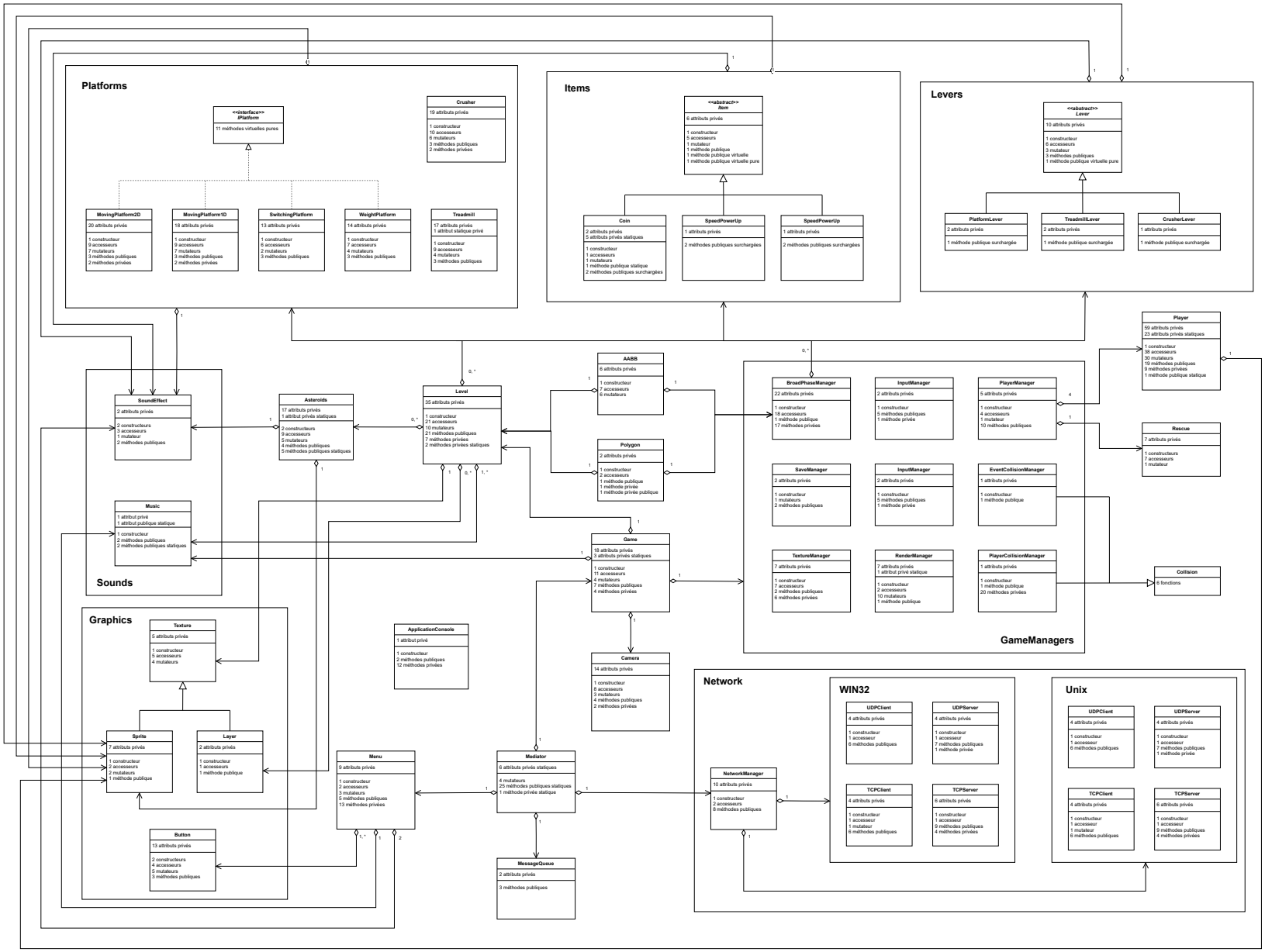


FIGURE 22 – Diagramme de classe du projet