

**Le Mans Université**  
Licence Informatique - 2<sup>e</sup> année  
Module 174UP02 Rapport de Projet  
**Chapper's Fallout**

Lucien Defosse, Ekrem Ayyildiz, Mehdi Ouaïl, Loup Picault

4 avril 2025

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Conception</b>	<b>4</b>
2.1	Cahier des charges . . . . .	4
2.2	Fonctionnalités . . . . .	5
<b>3</b>	<b>Organisation du projet</b>	<b>6</b>
3.1	Planning prévisionnel . . . . .	6
3.2	Répartition des tâches . . . . .	6
<b>4</b>	<b>Développement</b>	<b>8</b>
4.1	Création du moteur de jeu 3D . . . . .	8
4.1.1	Conception du moteur de rendu (OpenGL) . . . . .	8
4.1.2	Conception du moteur de physique . . . . .	9
4.1.3	Conception de l'arbre de données . . . . .	11
4.1.4	Conception des classes et héritage . . . . .	12
4.1.5	Conception des entrées sorties . . . . .	12
4.2	Création des ressources . . . . .	13
4.2.1	Conception du modèle 3d et des textures . . . . .	13
4.2.2	Conception des musiques et effets sonores . . . . .	14
4.3	Gestion des scènes . . . . .	15
4.3.1	Gestion des objets . . . . .	15
4.3.2	Gestion des PNJ (personnages non joueurs) . . . . .	16
4.3.3	Placement des collisions . . . . .	16
<b>5</b>	<b>Résultats et conclusion</b>	<b>18</b>
<b>6</b>	<b>Bibliographie</b>	<b>18</b>
<b>7</b>	<b>Annexes</b>	<b>23</b>
7.1	Exemples de débogage . . . . .	23
7.2	Code d'une classe générique . . . . .	23
7.3	Rendu différé . . . . .	24
7.4	Pipeline graphique . . . . .	26
7.5	Musique . . . . .	28
7.6	Tableau des différents noeuds de la scène . . . . .	29
7.7	Tableau des commandes Socket du mini-jeu « FPS Chess » . . . . .	30

## Résumé

Ce projet consiste à développer un jeu vidéo d'horreur en 3D en langage C, se déroulant dans l'institut Claude Chappe. Le joueur doit y résoudre des mini-jeux tout en échappant à un monstre qui le poursuit. Le jeu repose sur un moteur 3D original, entièrement conçu par notre équipe, sans recourir à un moteur existant, et intègre modélisation 3D, gameplay et éléments sonores. Le projet s'inscrit dans le cadre de la formation de L2 Informatique de l'Université du Mans (année universitaire 2024–2025).

## 1 Introduction

Ce document présente le projet du jeu Chapper's Fallout réalisé dans le cadre de la formation de L2 informatique de l'université du Mans pendant la période de janvier à avril 2025 (mais développé en avance depuis septembre 2024). Ce projet a été développé en langage C avec la librairie SDL.

Dans ce jeu, le joueur peut se déplacer dans l'institut Claude Chappe, avec pour objectif de résoudre tous les mini-jeux 2D disponibles sur les PC disséminés dans différents endroits de la zone. Il doit réussir cela tout en survivant au monstre Chapper, qui le poursuit par moments, et en surmontant des parcours semés de trous et de plateformes.

Le joueur possède un saut pour l'aider dans ces parcours, ainsi qu'une lampe torche pour éclairer son chemin. Un tutoriel introduit les principes du jeu et les contrôles permettant les déplacements, l'utilisation de la lampe et du saut avant que la partie principale ne commence.

Nous présenterons dans une première partie notre jeu, ses scénarios d'utilisation et ses principales fonctionnalités, puis dans une deuxième partie la gestion du projet. Ensuite, dans une troisième partie, nous détaillerons les éléments principaux de conception (algorithmes, structures de données, etc.). Nous exposerons par la suite l'architecture de notre application (structuration du code en fichiers) avant de montrer les principaux résultats obtenus. Enfin, nous conclurons sur les points forts et les limites de notre travail, les écarts entre la planification prévisionnelle et le déroulement réel du projet, ainsi que les leçons tirées de cette expérience.

En annexe, nous présenterons un exemple de débogage et des tests (jeux d'essai et cas de test d'un exemple au moins — le fichier .c étant disponible dans le dépôt Git dans un répertoire dédié "test").

## 2 Conception

### 2.1 Cahier des charges

#### - Contexte et objectifs du projet :

Ce projet, réalisé dans le cadre de la L2 Informatique à l'Université du Mans, consiste à concevoir un jeu vidéo en langage C. Nous avons choisi de créer un jeu d'horreur en 3D prenant place dans l'institut Claude Chappe où le joueur explore un bâtiment, parfois poursuivi par un monstre, tout en résolvant des mini-jeux (voir détails dans l'introduction ??).

#### - Contraintes techniques :

- Langage imposé : C (exclusivement ou presque) - Aucune utilisation de moteur de jeu existant (Unity, Unreal...) - Bibliothèques autorisées : SDL / OpenGL - Durée du projet : janvier à avril 2025 (avec un travail amorcé dès septembre 2024)

#### - Fonctionnalités prévues :

- Déplacement en 3D dans une reproduction fidèle du bâtiment - Monstre poursuivant le joueur à certains moments - Mini-jeux en 2D intégrés dans des PC interactifs - Mécaniques de plateforme (saut, trous, obstacles) - Lampe torche pour éclairer l'environnement - Tutoriel expliquant les commandes

#### - Architecture générale :

Pour cela, nous développons notre propre moteur 3D, *RaptiquaX*, en C avec SDL. L'institut est modélisé en 3D sur Blender, puis intégré au moteur. Un éditeur de niveau nous permet de placer objets, collisions et éléments interactifs. Les mini-jeux sont codés séparément puis intégrés au jeu.

#### - Répartition des rôles :

- **Lucien** : modélisation 3D du bâtiment/objets, design sonore
- **Loup** : moteur 3D, éditeur de niveau, mécaniques principales
- **Ekrem** : conception et développement des mini-jeux
- **Mehdi** : placement des collisions et éléments de jeu via l'éditeur

## 2.2 Fonctionnalités

Chapper's Fallout est un jeu d'horreur en 3D où le joueur doit explorer l'institut Claude Chappe tout en évitant un monstre qui le poursuit et en utilisant des plateformes pour naviguer dans l'environnement. Pour mener à bien ce projet, nous avons développé un moteur 3D en C, nommé *RaptiquaX*, qui gère les déplacements, les collisions et les interactions avec l'environnement, en plus de permettre un affichage 3D avancé avec OpenGL.

- *Déplacement en 3D* - Le joueur peut se déplacer dans l'institut Claude Chappe en utilisant les touches ZQSD (ou flèches directionnelles) pour avancer, reculer et tourner. Il peut également sauter avec la touche espace.
- *Monstre* - Le joueur est poursuivi par un monstre qui apparaît à certains moments. Il doit éviter d'être touché par le monstre, sinon il perd une vie.
- *Mini-jeux* - Le joueur peut interagir avec des ordinateurs pour accéder à des mini-jeux en 2D. Ces mini-jeux sont variés et nécessitent différentes compétences pour être résolus.
- *Lampe torche* - Le joueur peut activer une lampe torche pour éclairer son chemin.
- *Vidéo exclusive* - Le joueur peut visionner une vidéo exclusive à la fin du jeu, qui lui explique la fin de l'histoire.
- *Affectation des touches* - Le joueur peut configurer les touches de son clavier pour contrôler le jeu.
- *Guide de jeu* - Un tutoriel interactif guide le joueur à travers les différentes étapes du jeu, séparés sous forme de chapitres.

## 3 Organisation du projet

### 3.1 Planning prévisionnel

Dès les prémisses du projet Chapper's Fallout, une planification claire et structurée a été établie, plusieurs mois avant le lancement effectif du développement. Cette anticipation visait à garantir une répartition cohérente des rôles, adaptée aux compétences et à l'expérience de chaque membre de l'équipe.

Loup, le membre le plus expérimenté en développement de jeux vidéo, a été naturellement désigné responsable de la conception et de la mise en place du moteur 3D. Son rôle central consistait à fournir les fondations techniques du jeu : moteur de rendu, système d'entrée/sortie, structure de données, ainsi qu'un éditeur de niveau pour faciliter le travail des autres membres.

Lucien, de son côté, s'est vu confier la création des ressources graphiques et sonores : modélisation 3D de la carte et des objets, textures, musiques et effets sonores, apportant une identité visuelle et sonore au jeu.

Mehdi et Ekrem, moins expérimentés en développement 3D au démarrage du projet, devaient capitaliser sur les outils fournis par Loup ( notamment l'éditeur de niveau ) pour implémenter des fonctionnalités concrètes du jeu. Leur mission initiale était donc orientée vers des tâches techniques réalisables et progressives, comme la gestion d'objets interactifs, des scènes ou encore l'intégration de mini-jeux.

Ce planning prévisionnel reposait ainsi sur une forte complémentarité des compétences : Loup assurait la direction technique et le développement du socle du jeu, pendant que les autres membres construisaient progressivement les éléments du gameplay, tout en se formant à mesure de l'avancement du projet.

Afin d'assurer le bon déroulement du projet, une stratégie spécifique a été mise en place dès le départ. Conscients que la réussite globale dépendait en grande partie de la création du moteur 3D, nous avons décidé que Loup commencerait son développement bien en amont, plusieurs mois avant le lancement officiel des séances de projet. Cette avance lui permettait de poser des bases techniques solides sur lesquelles le reste de l'équipe pourrait s'appuyer. Parallèlement, Mehdi et Ekrem avaient pour objectif de se familiariser rapidement avec ce moteur, afin de pouvoir intervenir efficacement sur le développement des différentes fonctionnalités du jeu. De son côté, Lucien devait produire une première version fonctionnelle du modèle 3D assez tôt, ce qui permettrait de tester les interactions en jeu et de débuter le codage des mécaniques. La suite du travail de modélisation visuelle pouvait ensuite se dérouler de manière plus progressive, en respectant les délais impartis.

### 3.2 Répartition des tâches

Tout au long du développement de Chapper's Fallout, notre groupe a su faire preuve d'une forte capacité d'adaptation face aux imprévus et aux défis techniques rencontrés. Si les rôles de Loup et Lucien sont restés globalement stables, Loup étant responsable du moteur 3D et Lucien de la modélisation 3D et des ressources sonores, les tâches initialement confiées à Ekrem et Mehdi ont

quant à elles évolué au fil du projet, en réponse aux réalités du terrain.

À l'origine, Mehdi et Ekrem devaient s'occuper du développement des fonctionnalités du jeu, en exploitant le moteur mis en place par Loup. Cependant, cette mission s'est avérée ambitieuse compte tenu de leur niveau initial en programmation 3D. Très tôt dans le projet, une remise en question collective a permis de recentrer leurs objectifs vers des tâches plus ciblées et adaptées : la création de mini-jeux en 2D avec SDL, destinés à enrichir l'expérience de jeu tout en s'inscrivant dans un cadre technique plus accessible.

Par ailleurs, l'équipe a su faire face à un autre ajustement majeur concernant la gestion des collisions. Initialement, cette tâche devait être assurée par Lucien. Cependant, la charge de travail en modélisation 3D s'est révélée plus importante que prévu, notamment pour finaliser les nombreux éléments visuels du jeu dans les temps. Pour permettre à Lucien de se concentrer pleinement sur cette mission essentielle, la responsabilité des collisions a été transférée à Mehdi. Ce dernier a alors assumé avec efficacité cette tâche minutieuse et critique, qui nécessitait une intervention manuelle dans les scènes 3D.

Ces ajustements successifs témoignent de la réactivité de notre groupe et de sa capacité à se réorganiser intelligemment face aux contraintes. Chacun a su faire preuve de flexibilité pour maintenir la dynamique du projet et garantir une progression constante jusqu'à son aboutissement.

## 4 Développement

### 4.1 Creation du moteur de jeu 3D

Pour composer les elments du jeu et assurer une bonne qualit graphique et technique, le choix d’assembler le *moteur de jeu 3D « RaptiquaX »* a et  convenu. *SDL2* permet un affichage dans un contexte *OpenGL*, la librairie graphique la plus utilis e dans l’industrie du jeu vid o avec la librairie *Vulkan*<sup>1</sup>.

Pour permettre une composition modulable et accessible a tous, le mod le d’architecture de *Godot*<sup>2</sup> a et  utilis  comme r f rence, en raison de son potentiel<sup>3</sup>.

#### 4.1.1 Conception du moteur de rendu (*OpenGL*)

Un moteur de rendu graphique moderne sous *GLSL* (ou ses alternatives<sup>4</sup>) permet deux pipelines diff erentes et incompatibles<sup>5</sup> [4] :

- Le rendu diff r  qui permet d’appliquer le fragment shader une seule fois par fragment en utilisant plusieurs couches de rendu.
- Le rendu direct qui affiche imm diatement sans pr calculer le G-Buffer ce qui peut s’av rer plus rapide selon le contexte.

Nous avons opt  pour l’utilisation du rendu diff r , car il permet de g rir plus facilement les effets de post-processing et d’optimiser le rendu de la lumi re. De plus, il est plus adapt  pour les sc nes complexes avec de nombreux objets et lumi res. (Pour aller plus loin sur le sujet, voir la section 7.3).

Le traitement de l’image est effectu  majoritairement par le GPU, exploitant la strat gie invent e par Silicon Graphics[9] pour le traitement de l’image :

- Le GPU est divis  en plusieurs unit s de calcul
- Chaque unit  de calcul est responsable d’un aspect du rendu
- Les unit s de calcul travaillent en parall le pour traiter l’image rapidement
- La g om trie est transform e en primitives (triangles, lignes, points) par un programme appell  *vertex shader* (figure 1a)
- Les primitives sont ensuite rasteris es pour cr er des fragments (figure 1b)

---

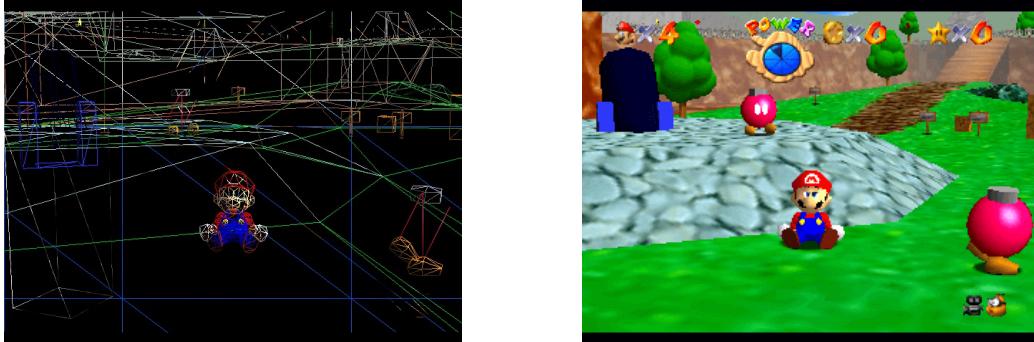
1. Vulkan (anciennement *OpenGL Next*), a et  d velopp  par le groupe Khronos comme le successeur qui remplace OpenGL.

2. Godot - 2014 - est un moteur de jeu libre multiplateforme proposant a la fois une interface en 2<sup>e</sup> et en 3<sup>e</sup> dimension.

3. Godot a et  tr s utilis  ces derni res ann es, m me dans la r alisation de jeu dit AAA comme Sonic Colors Ultimate[1]

4. p. ex. HLSL sous Direct3D

5. Depuis quelques ann es, la pipeline "Forward+" cherchant a m ler les deux par des m thodes avanc es s’est popularis .



(a) Super Mario 64 en mode fil de fer pour imager le (b) Super Mario 64, après le passage par le fragment vertex shader

FIGURE 1 – Exemple de rendu avec la stratégie de Silicon Graphics [10]

Enfin, le GPU est de nouveau sollicité pour le rendu de la scène finale, en utilisant les informations de la scène stockées dans le G-Buffer et en appliquant les effets de post-processing. Dans notre moteur, nous avons implémenté les techniques de rendu suivantes :

- Le rendu différé
- Le rendu de la lumière en suivant le modèle de Phong
- Le rendu des ombres en utilisant le shadow mapping
- Le rendu de réflexions et de reflets en utilisant le SSR (Screen Space Reflection) (à but expérimental en utilisant le shader de *Imanol Fotia* [3])
- Le rendu d'occlusion ambiante en utilisant le SSAO (Screen Space Ambient Occlusion)
- Le rendu d'éblouissement en utilisant le bloom
- L'anti-aliasing en utilisant le SMAA (Sub-pixel Morphological Anti-Aliasing)

Une grande partie de ces techniques sont inspirées des travaux de Joey de Vries dans son site web LearnOpenGL [2], auquels nous avons ajouté des améliorations et des optimisations pour notre moteur grâce aux travaux de Adrian Courrèges sur DOOM 2016 [5].

Pour un aperçu de la pipeline graphique de notre moteur, voir la section 7.4.

#### 4.1.2 Conception du moteur de physique

Le moteur de physique est responsable de la gestion des collisions et des interactions entre les objets de la scène. Il utilise un système de détection de collisions basé sur des formes géométriques simples, comme des sphères, des boîtes aux transformations non uniforme, des plans, etc.

La physique est ensuite calculée en utilisant une approche simplifiée des équations *cinématiques* et des équations du *principe fondamental de la dynamique (deuxième loi de Newton)*.

Soit deux objets rigides,  $A$  et  $B$ , se déplacent et entrent en collision. Voici comment nous calculons la réponse à la collision en utilisant les concepts de base de la physique des corps rigides.

#### 4.1.2.1 Norme de la collision

La norme de la collision entre les deux corps est donnée par un vecteur normal à la surface de collision. Si les corps ont une normale de collision définie, cette norme est utilisée dans les calculs suivants.

#### 4.1.2.5 Calcul de l'impulsion

$$I_{scalaire} = \frac{(1 + e) \cdot v_{normal}}{m_A^{-1} + m_B^{-1}} \quad (5)$$

#### 4.1.2.6 Calcul du vecteur d'impulsion

$$\mathbf{n} = \lambda_A \cdot \lambda_B \quad (1)$$

$$\mathbf{I} = I_{scalaire} \cdot \mathbf{n} \quad (6)$$

#### 4.1.2.2 Vitesse relative

La vitesse relative des deux objets rigides  $A$  et  $B$  est calculée comme la différence entre leurs vitesses respectives :

$$\mathbf{v}_{relative} = \mathbf{v}_B - \mathbf{v}_A \quad (2)$$

#### 4.1.2.7 Application de l'impulsion et du couple

$$\mathbf{r}_A = \mathbf{P}_{impact} - \mathbf{C}_A \quad (7)$$

$$\mathbf{r}_B = \mathbf{P}_{impact} - \mathbf{C}_B \quad (8)$$

$$\text{Couple sur A} = \mathbf{r}_A \times \mathbf{I} \quad (9)$$

$$\text{Couple sur B} = \mathbf{r}_B \times \mathbf{I} \quad (10)$$

#### 4.1.2.3 Vitesse relative selon la direction normale

#### 4.1.2.8 Application de l'impulsion

$$\mathbf{v}'_A = \mathbf{v}_A + \mathbf{I} \cdot m_A^{-1} \quad (11)$$

$$v_{normal} = \mathbf{v}_{relative} \cdot \mathbf{n} \quad (3)$$

$$\mathbf{v}'_B = \mathbf{v}_B - \mathbf{I} \cdot m_B^{-1} \quad (12)$$

#### 4.1.2.9 Correction de la pénétration

#### 4.1.2.4 Coefficient de restitution (élasticité de la collision)

$$\mathbf{C} = \delta \cdot \mathbf{n}$$

$$\mathbf{P}'_A = \mathbf{P}_A - \mathbf{C} \quad (13)$$

$$e = 0.02 \quad (4)$$

$$\mathbf{P}'_B = \mathbf{P}_B + \mathbf{C} \quad (14)$$

#### 4.1.2.10 Calcul du couple et de l'inertie    4.1.2.11 Accélération angulaire

$$\mathbf{I}_{\text{locale}} = \frac{m}{12} \begin{bmatrix} h^2 + d^2 & 0 & 0 \\ 0 & w^2 + d^2 & 0 \\ 0 & 0 & w^2 + h^2 \end{bmatrix} \quad \alpha = \mathbf{I}^{-1} \cdot \mathbf{T} \quad (16)$$

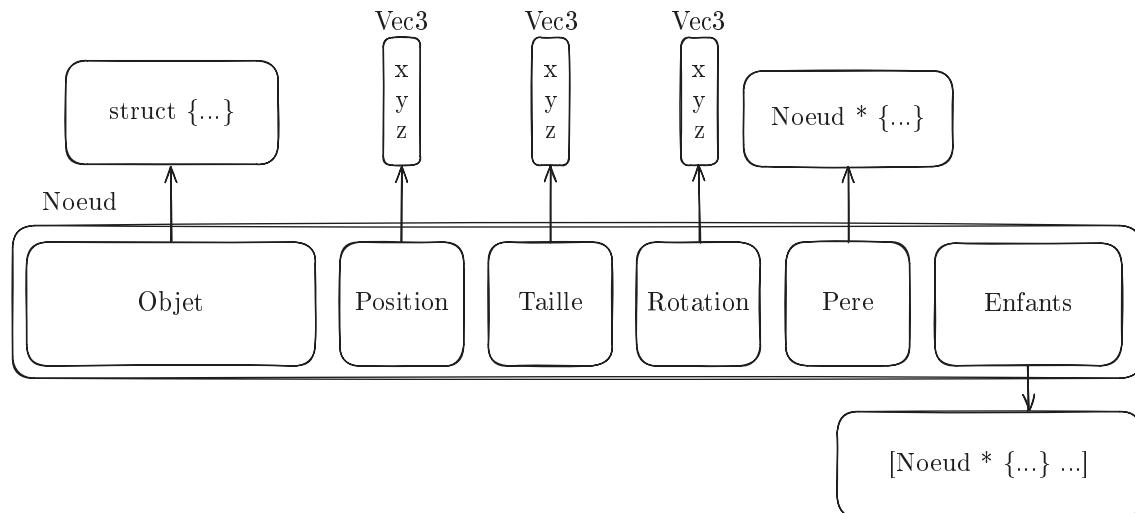
(15)

#### 4.1.3 Conception de l'arbre de données

Dans notre moteur, nous avons choisi d'utiliser un arbre de données pour représenter la scène 3D. Cet arbre est composé de nœuds, où chaque nœud représente un objet de la scène. Chaque nœud contient des informations sur la position, la rotation et l'échelle de l'objet, ainsi que des références vers ses enfants et son père. Cela permet de créer une hiérarchie d'objets, où chaque objet peut être transformé indépendamment tout en étant affecté par les transformations de ses parents.

Ce modèle d'arbre de données est inspiré du moteur de jeu Godot, qui utilise également un arbre de nœuds pour représenter une scène.

Les nœuds de l'arbre sont génériques, ce qui garantit un stockage efficace des données et une gestion facile des objets de la scène (figure 2).



*La structure présentée ici est une simplification de la structure utilisée dans notre moteur de jeu.*

FIGURE 2 – Structure d'un noeud générique

Vous pouvez vous référer à la table 1 pour découvrir les différents types de nœuds que nous avons implémentés dans notre moteur de jeu. Chaque nœud est responsable d'un aspect spécifique de la scène, ce qui permet de créer un tout cohérent et fluide.

#### 4.1.4 Conception des classes et héritage

Pour assurer une bonne organisation du code et une gestion efficace des objets de la scène, nous avons utilisé un système de classes capable de gérer l'héritage.

Le code source du moteur est précompilé par un outil développé par nos soins, qui permet de générer un code source en C à partir d'un fichier classe inspiré de la syntaxe de C++. Les classes ainsi générées sont ensuite liées entre elles par un header de liaison, lui aussi généré par l'outil. Cela permet de créer un code source propre et facilement lisible, tout en assurant une bonne gestion de la mémoire et des performances.

Un exemple des classes utilisées par l'outil de précompilation est présenté dans le listing 1.

Le code source généré utilise les arguments variadiques et prend en charge les promotions automatiques des types.

#### 4.1.5 Conception des entrées sorties

Le moteur prend en charge les entrées/sorties de sorte à permettre aux développeurs de créer des jeux en utilisant des fichiers de ressources sans avoir à se soucier de l'implémentation ou de la gestion de la mémoire et des performances.

##### 4.1.5.1 Chargement des ressources

Les ressources du moteur sont chargées à l'aide de diverses fonctions d'entrée/sortie, qui permettent de charger des fichiers de différents formats. Voici une liste non exhaustive des formats de fichiers pris en charge par le moteur :

- **OBJ** : format de fichier 3D utilisé pour stocker des modèles 3D. Il est simple et largement utilisé dans l'industrie du jeu vidéo. [12]
- **PNG** : format de fichier image utilisé pour stocker des textures.
- **JPG** : format de fichier image compressé utilisé pour stocker des textures.
- **WAV** : format de fichier audio brut utilisé pour stocker des sons.
- **OGG** : format de fichier audio compressé utilisé généralement pour stocker des sons.
- **MP3** : format de fichier audio compressé utilisé généralement pour stocker des musiques.
- **SCENE** : format de fichier propriétaire utilisé pour stocker des scènes 3D.
- **FS** : format de fichier shader utilisé pour stocker des shaders de fragment. [13]
- **VS** : format de fichier shader utilisé pour stocker des shaders de vertex. [13]

Les ressources sont stockées en cache pour éviter de les recharger plusieurs fois. Le moteur utilise un système de gestion de ressources qui permet de charger et de décharger les ressources du cache sur demande. Cela permet de réduire la consommation de mémoire et d'optimiser les performances du moteur.

#### **4.1.5.2 Communication en réseau (SocketIO)**

Le moteur permet en outre de gérer la communication en réseau entre des clients et un serveur en multithreads. Pour cela, nous avons utilisé la librairie *Socket* en C, qui permet de gérer la communication en temps réel entre le serveur et les clients. Nous avons implémenté un serveur *Socket* qui gère les connexions des clients et les messages échangés entre eux. Le serveur est capable de gérer plusieurs clients en même temps et de leur envoyer des messages en temps réel. Il est également capable de gérer les connexions et déconnexions des clients, ainsi que les erreurs de communication. Le serveur de démonstration que nous avons mis en place est capable de gérer une dizaine de commandes différentes (table 2).

Comme preuve de concept, nous avons mis en place un serveur pour « *FPS Chess* », un mini-jeu fps multijoueur qui gère plusieurs parties de deux joueurs en même temps.

## **4.2 Création des ressources**

### **4.2.1 Conception du modèle 3d et des textures**

La map du jeu, représentant le bâtiment Claude Chappe et ses environs, a été créée en plusieurs étapes via Blender :

#### **- Modélisation 3D des modèles :**

On utilise des objets préfabriqués de Blender, modifiables grâce à divers outils pour obtenir les formes désirées (ex : transformation d'un cube en mur ou en porte comme on le voit figure 3 et 4). Nous avons d'abord modélisé la structure de base du bâtiment, puis ajouté des éléments exclusifs au jeu, comme des trous dans le sol et les murs pour un parcours de poursuite, ainsi que trois salles secrètes avec des ordinateurs pour les mini-jeux.

#### **- Normals :**

Les normales définissent l'orientation des faces des objets et influencent l'effet de la lumière. Bien que généralement correctes par défaut, il est parfois nécessaire de les ajuster pour éviter des erreurs d'affichage.

#### **- Texturage :**

Une fois les objets modélisés, ils sont texturés en utilisant des textures d'image ou des couleurs unies, compatibles avec le moteur 3D. Par exemple, une porte est texturée en rouge uni, tandis que les murs utilisent une texture image de plastique gris/blanc.

#### **- Séparation des zones :**

Les objets sont regroupés en "collections" selon leur emplacement (amphithéâtre, couloirs, salles de classe, etc.), ce qui facilite leur exportation en fichiers .obj distincts et optimise le chargement des ressources dans le moteur.

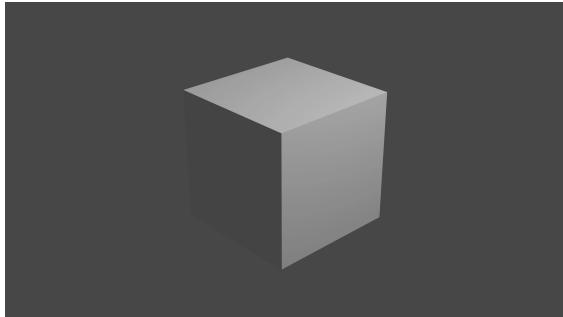


FIGURE 3 – Cube de base Blender

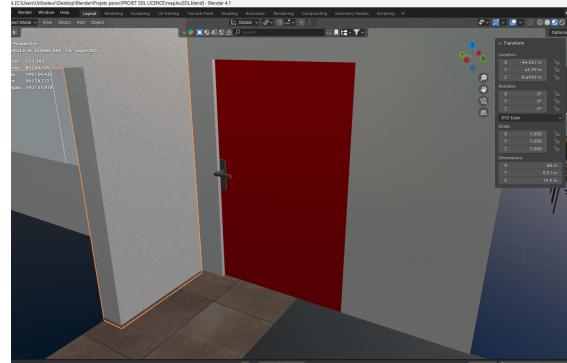


FIGURE 4 – Cubes modifiés en divers objets

#### - Optimisation des modèles et textures :

Les erreurs de modélisation sont corrigées pour garantir une bonne intégration. Les UV Maps des textures sont ajustées pour assurer un affichage correct après l'exportation.

#### - Exportation :

Les objets d'une zone sont fusionnés (JOIN) pour simplifier l'importation dans le moteur (un seul objet au lieu de plusieurs centaines). On les exporte ensuite en .obj et .mtl (textures), en veillant à inclure toutes les images de textures nécessaires.

### 4.2.2 Conception des musiques et effets sonores

Les musiques et les effets sonores sont tous les deux composés sur le logiciel FL Studio.

#### - Conception des musiques :

À partir de sons déjà existants (libres de droits), on les utilise en les modifiant (ou non), pour en faire des mélodies (à l'aide d'un clavier synthétiseur), des boucles de divers instruments, bref créer une musique.

#### - Conception des effets sonores :

Pour les effets sonores, on utilise à nouveau des sons déjà existants, évidemment sans faire des mélodies cette fois. On les modifie à travers divers plugins du logiciel (pitch grave/aigu, réverbération, etc.).

Vous trouverez en annexe un exemple de composition musicale (voir figure 13).

## 4.3 Gestion des scènes

### 4.3.1 Gestion des objets

Les objets sont des entités gérées par des scripts importés dans les scènes du moteur 3D.

Les scripts sont écrits en C et sont chargés par le moteur 3D au moment de l'importation de la scène. Ils sont ensuite exécutés par le moteur 3D lors de la boucle de jeu à des instants clés.

On peut ainsi définir des comportements spécifiques pour l'initialisation des objets, leur mise à jour et lors de la réception de signaux.

Les interactions entre les objets et l'environnement sont gérées par ces signaux qui sont émis notamment lorsque les objets entrent en collision avec d'autres objets, mais il est possible de paramétrier divers signaux personnalisés.

#### 4.3.1.1 Mini-jeux

Dans le cadre de Chapper's Fallout, les mini-jeux en 2D sont conçus pour être totalement intégrés au sein d'un environnement homogène. Chaque mini-jeu débute par l'activation d'une fenêtre qui simule le démarrage d'un ordinateur. Cette fenêtre, gérée par un système de fenêtrage complet, initialise automatiquement l'affichage du bureau virtuel et coordonne le déroulement des scènes à l'aide d'un gestionnaire d'événements centralisé.

Le premier mini-jeu, par exemple, présente un bureau sur lequel se trouvent des icônes décoratives. Le cœur du gameplay repose sur l'apparition de notifications qui indiquent la prochaine mission. Ici, le joueur reçoit un message de Souleymane lui demandant de le rejoindre pour un TP dans le bâtiment CC. Une fois le message accepté via une interaction clavier ou souris, la fenêtre signale la fin du mini-jeu et permet le retour fluide dans le monde 3D.

Le second mini-jeu suit une logique similaire : l'ordinateur se lance dans une nouvelle fenêtre affichant le bureau, puis, en cliquant sur une icône spécifique, le joueur accède à un jeu en 2D. Ce dernier consiste à esquiver des objets projetés depuis les bords de la fenêtre. Une fois l'épreuve terminée, le joueur peut ouvrir un fichier interactif, conçu pour afficher des lignes de code de l'IA, et tenter de modifier ces paramètres pour réparer le système, même si l'issue n'est pas toujours favorable.

Le troisième mini-jeu reprend le même mécanisme de lancement d'ordinateur. Après l'affichage du bureau, le joueur est amené à réaliser un jeu de décodage simple, où il doit décrypter une série de symboles. La réussite de cette épreuve permet d'accéder à un fichier de nettoyage, indispensable pour poursuivre la réparation de l'IA.

Au niveau du code, la gestion des événements est assurée par une boucle principale qui interroge SDL afin de capter en temps réel les entrées utilisateur (clics, déplacements de souris, frappes clavier). Chaque mini-jeu possède ainsi son propre « event manager » qui orchestre les transitions entre les différentes scènes. Par ailleurs, le système de gestion de fenêtres permet de créer, mettre

à jour et détruire les fenêtres de jeu en synchronisation avec ces événements, assurant ainsi une expérience utilisateur fluide et réactive.

Les structures de données, notamment celles dédiées aux éléments du bureau et aux textures animées, jouent un rôle crucial dans l'animation et l'affichage des éléments graphiques. Par exemple, la fonction de mise à jour des textures incrémente le compteur de frames pour afficher la portion adéquate d'un spritesheet, garantissant une animation cohérente et dynamique. Ce modèle a d'ailleurs été pensé pour être facilement extensible, afin d'ajouter de nouvelles interactions ou de modifier l'ordre des scènes sans perturber le fonctionnement global du système.

Dans l'ensemble, ce design modulaire basé sur le système de fenétrage intégré, couplé à une gestion des événements et une architecture orientée scène, permet de créer des mini-jeux diversifiés et immersifs, renforçant ainsi l'expérience globale du joueur dans l'univers de Chapper's Fallout.

#### **4.3.2 Gestion des PNJ (personnages non joueurs)**

Les PNJ sont des entités qui se déplacent dans la scène. Ils sont gérés par le moteur 3D et sont placés dans la scène via l'éditeur de niveau. Leurs comportements sont défini par des scripts qui sont exécutés par le moteur et utilisent le principe de la machine à état.

Pour rendre les PNJ plus réalistes, nous avons implémenté un système de chemins de navigation. Ce système permet aux PNJ de se déplacer le long d'un chemin prédéfini dans la scène. Les chemins sont définis par des points de contrôle placés dans la scène. Le moteur 3D utilise ces points de contrôle pour calculer la trajectoire des PNJ et les faire se déplacer le long de cette trajectoire.

#### **4.3.3 Placement des collisions**

Une fois que le modèle est affiché et que tous les obstacles et éléments de décors sont placés, il faut qu'on puisse interagir avec eux.

Pour éviter de passer à travers les murs il faut détecter quand le joueur est à la même position que le mur en question.

On a choisi d'abord d'afficher le modèle 3D, ensuite grâce à l'éditeur on place des "boîtes" de collisions. En jeu ces boîte seront invisible, mais infranchissable, ça permet de ne pas traverser les obstacles.

Par exemple on modelise une table avec le moteur 3D. Dans l'éditeur on rajoute manuellement une "boîte" de collisions à l'emplacement de la table, on cherche à épouser la forme de la table. En jeu on verra que la table, et on ne pourra pas la traverser, on peut même monter dessus etc.

Sur la figure 5 montre les différents types de "boîtes". On voit qu'on dessine chaque éléments avec des boîtes.

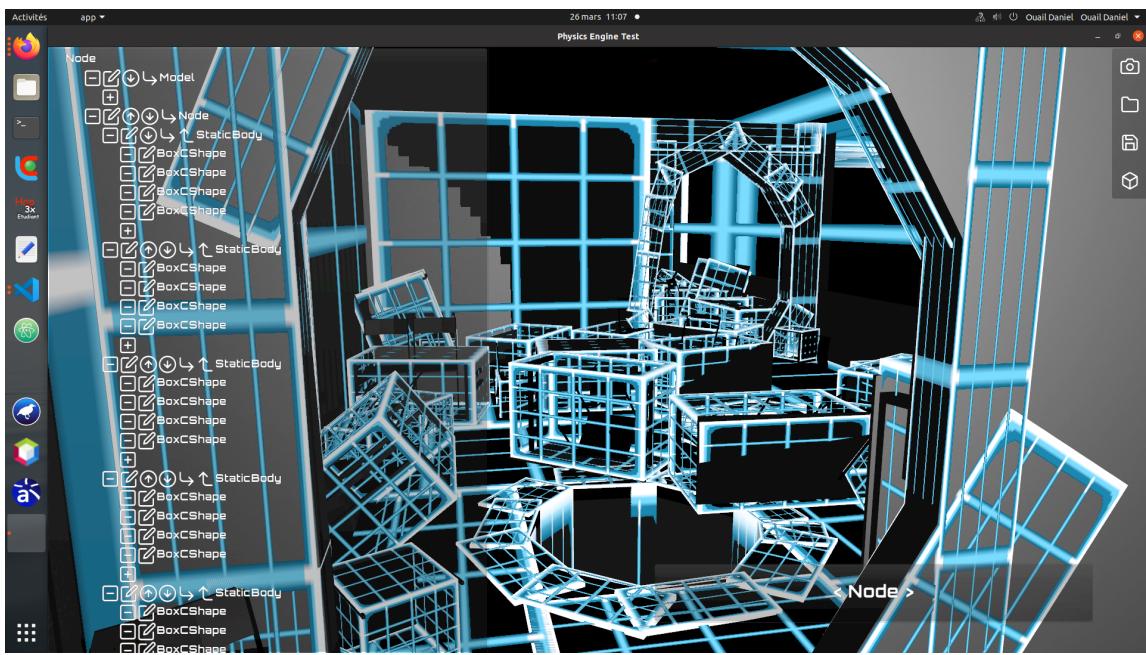


FIGURE 5 – Capture d'écran de l'éditeur de niveau

Les coordonnées de chaque boîtes de collisions sont enregistrées avec l'éditeur dans un fichier .scene, ce fichier sera lu au moment de l'execution pour charger le modèle et les collisions.

## 5 Résultats et conclusion

Finalement, nous avons réussi à créer un jeu vidéo d'horreur en 3D, avec un moteur 3D original, sans utiliser de moteur existant. Le jeu est fonctionnel et permet au joueur de se déplacer dans l'institut Claude Chappe, d'interagir avec des objets, de résoudre des mini-jeux et d'éviter un monstre qui le poursuit. Nous avons également eu l'occasion d'intégrer des éléments sonores et visuels pour améliorer l'expérience de jeu.

Ce fut un projet ambitieux qui nous a permis de mettre en pratique nos connaissances en programmation, en modélisation 3D et en conception de jeux vidéo. Nous avons appris à travailler en équipe, à gérer un projet de A à Z et à surmonter des défis techniques.

Les principales leçons tirées de ce projet sont :

- Une meilleure gestion du temps et des tâches aurait permis d'éviter le stress de dernière minute.
- La communication au sein de l'équipe est essentielle pour s'assurer que tout le monde est sur la même longueur d'onde et que les tâches sont bien réparties.
- La documentation est cruciale pour faciliter la compréhension du code et des fonctionnalités du jeu.
- La modélisation 3D et l'intégration de ressources dans un moteur de jeu sont des étapes complexes qui nécessitent de la patience et de la rigueur.
- La gestion des collisions et des interactions entre objets est un aspect clé du développement de jeux vidéo, et il est important de bien le planifier dès le début.

Quant aux apprentissages, nous avons acquis des compétences en :

- Programmation en C et utilisation de la bibliothèque SDL pour le développement de jeux.
- Modélisation 3D avec Blender et intégration de ressources dans un moteur de jeu.
- Gestion de projet et travail en équipe.
- Conception d'un moteur de rendu 3D complet et avancé avec OpenGL.

## 6 Bibliographie

### Glossaire

**bloom** Le bloom est une technique de post-traitement utilisée pour simuler l'effet de diffusion de la lumière sur les surfaces brillantes. Elle crée un halo lumineux autour des objets lumineux pour améliorer le réalisme de la scène. 9

**fragment** Un fragment est un composant graphique composé de peu de pixels qui constitue une primitive en informatique graphique. 8

**fragment shader** Le fragment shader est responsable du traitement des fragments lors de la pipeline graphique. 8, 9

**G-Buffer** Le GBuffer est un tampon de rendu utilisé dans le rendu différé pour stocker les informations de la scène, telles que la couleur, les normales et la profondeur. 8, 9, 19

**GLSL** OpenGL Shading Language (GLSL) est un langage de programmation de shaders de haut niveau dont la syntaxe est fondée sur le langage C.. 8

**GPU** processeur graphique. 8, 9, 19

**OpenGL** Branche libre de la librairie graphique GL développée par Silicon Graphics en 1992 et reprise par le groupe Khronos à partir de 2006. 8

**rasteriser** (Informatique) Convertir des données numériques en points permettant leur impression ou leur affichage. [11]. 8

**SDL2** Simple DirectMedia Layer 2 (SDL2) est la deuxième version de SDL, une librairie graphique développée par Sam Lantinga ; acclamée pour sa portabilité et son accessibilité. 8

**shader** Un shader est un programme graphique compilé en microcode graphique sur le GPU. 9

**shadow mapping** Le shadow mapping est une technique de rendu d'ombres utilisée dans le rendu différé. Elle consiste à créer une carte d'ombres (shadow map) qui stocke les informations de profondeur de la scène depuis la perspective de la source lumineuse. Lors du rendu, on compare la profondeur d'un fragment avec la profondeur stockée dans la shadow map pour déterminer s'il est dans l'ombre ou non. 9

**SMAA** Le SMAA (Subpixel Morphological Anti-Aliasing) est une technique d'anti-aliasing utilisée pour réduire les artefacts de crênelage (souvent appelé *effet escalier*) dans les images 3D. Elle combine plusieurs techniques d'anti-aliasing pour obtenir un rendu plus lisse et réaliste. 9

**SSAO** L'SSAO (Screen Space Ambient Occlusion) est une technique de rendu utilisée pour simuler l'occlusion ambiante dans les scènes 3D. Elle permet de créer des ombres douces et réalistes en tenant compte de la géométrie de la scène et de la position des lumières. 9

**SSR** Le SSR (Screen Space Reflection) est une technique de rendu utilisée pour créer des réflexions réalistes en utilisant les informations de la scène stockées dans le G-Buffer. Elle permet de simuler des réflexions sur des surfaces réfléchissantes en se basant sur les pixels visibles à l'écran. 9

**vertex** Un vertex est un point dans l'espace 3D qui définit la position d'un sommet d'une primitive (triangle, ligne, point) dans la pipeline graphique. 19

**vertex shader** Le vertex shader est responsable du traitement des primitives (triangles, lignes, points) lors de la pipeline graphique. 8, 9

## Nomenclature

$\delta$  Profondeur de pénétration

$\alpha$  Accélération angulaire

$\lambda_A$  Normale de l'objet  $A$

$\lambda_B$	Normale de l'objet $B$
<b>C</b>	Correction appliquée pour compenser la pénétration
$\mathbf{C}_A$	Centre de masse de $A$
$\mathbf{C}_B$	Centre de masse de $B$
<b>I</b>	Vecteur d'impulsion appliqué pendant la collision
$\mathbf{I}^{-1}$	Inverse de la matrice d'inertie
$\mathbf{I}_{\text{locale}}$	Matrice d'inertie locale
<b>n</b>	Vecteur normal à la surface de collision
$\mathbf{P}_A$	Position initiale de l'objet $A$
$\mathbf{P}'_A$	Nouvelle position de l'objet $A$
$\mathbf{P}_B$	Position initiale de l'objet $B$
$\mathbf{P}'_B$	Nouvelle position de l'objet $B$
$\mathbf{P}_{\text{impact}}$	Point d'impact
$\mathbf{r}_A$	Vecteur du centre de masse de $A$ au point d'impact
$\mathbf{r}_B$	Vecteur du centre de masse de $B$ au point d'impact
<b>T</b>	Couple appliqué
$\mathbf{v}_A$	Vitesse de l'objet $A$
$\mathbf{v}'_A$	Nouvelle vitesse de l'objet $A$
$\mathbf{v}_B$	Vitesse de l'objet $B$
$\mathbf{v}'_B$	Nouvelle vitesse de l'objet $B$
$\mathbf{v}_{\text{relative}}$	Vitesse relative entre deux objets
$d$	Profondeur de l'objet
$e$	Coefficient de restitution
$h$	Hauteur de l'objet
$I_{\text{scalaire}}$	Quantité scalaire représentant l'impulsion échangée
$m_A$	Masse de l'objet $A$
$m_B$	Masse de l'objet $B$
$v_{\text{normal}}$	Vitesse relative projetée sur la normale de collision
$w$	Largeur de l'objet

## Références

- [1] Sonic Team (3 septembre 2021) *Sonic Colours Ultimate*, SEGA.
- [2] Joey de Vries, *LearnOpenGL*. Disponible en ligne : <https://learnopengl.com/>. LearnOpenGL est une ressource en ligne exhaustive dédiée à l'apprentissage de la programmation graphique avec OpenGL. Le site couvre des sujets tels que la manipulation des shaders, la gestion des textures, les transformations 3D, ainsi que des concepts avancés comme le rendu différé et les ombres en temps réel. Consulté le 1 avril 2025.
- [3] Imanol Fotia, *Blog - Article 1*. Disponible en ligne : <https://imanolfotia.com/blog/1>. Ce blog aborde divers sujets liés à la programmation, à l'ingénierie logicielle et aux technologies modernes. Il propose des analyses approfondies, des tutoriels techniques et des réflexions sur les bonnes pratiques en développement. Consulté le 1 avril 2025.
- [4] Jean-Claude Iehl, *Deferred Shading vs Forward Shading*. Disponible en ligne : [https://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/M2PROIMA/2018/deferred\\_vs\\_forward.html](https://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/M2PROIMA/2018/deferred_vs_forward.html). Ce document pédagogique explique les différences entre le rendu différé (Deferred Shading) et le rendu direct (Forward Shading). Il couvre les principes fondamentaux de chaque technique, leurs avantages et inconvénients, ainsi que leurs applications dans les moteurs de rendu modernes. Consulté le 1 avril 2025.
- [5] Adrian Courrèges, *DOOM (2016) - Graphics Study*. Disponible en ligne : <https://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/>. Cet article propose une analyse approfondie du moteur graphique de \*DOOM (2016)\*, en explorant les techniques de rendu utilisées, telles que le rendu différé, l'éclairage, la gestion des ombres, le post-traitement et l'optimisation des performances. L'auteur décortique les effets visuels et les choix techniques ayant permis au jeu d'atteindre un haut niveau de fluidité et de qualité visuelle. Consulté le 1 avril 2025.
- [6] Valve Corporation, *Source Engine Features*. Disponible en ligne : [https://developer.valvesoftware.com/wiki/Source\\_Engine\\_Features](https://developer.valvesoftware.com/wiki/Source_Engine_Features). Cette page du wiki officiel de Valve décrit les fonctionnalités du moteur Source, utilisé dans de nombreux jeux comme \*Half-Life 2\*, \*Portal\* et \*Counter-Strike : Source\*. Elle couvre des aspects tels que le rendu graphique, la gestion de la physique, le réseau, l'intelligence artificielle et les outils de développement fournis avec le moteur. Consulté le 1 avril 2025.
- [7] Wikipedia, *CryEngine*. Disponible en ligne : <https://en.wikipedia.org/wiki/CryEngine>. Cet article de Wikipedia fournit une vue d'ensemble détaillée de CryEngine, un moteur de jeu développé par Crytek. Il explore ses caractéristiques techniques, ses applications dans les jeux vidéo, ses versions successives et ses améliorations au fil du temps. CryEngine est reconnu pour ses capacités de rendu photoréaliste et son utilisation dans des jeux populaires comme \*Crysis\*. Consulté le 1 avril 2025.
- [8] Chanhaeng, *Normal/Parallax Mapping with Self-Shadowing*. Disponible en ligne : <https://chanhaeng.blogspot.com/2019/01/normalparallax-mapping-with-self.html>. Cet article explique en détail l'implémentation du normal mapping et du parallax mapping avec auto-ombrage. L'auteur y décrit les concepts théoriques de ces techniques graphiques avancées et leur application dans les moteurs de rendu. Le blog fournit également un code exemple pour implémenter ces effets dans les shaders. Consulté le 1 avril 2025.

- [9] Silicon Graphics. Disponible en ligne : <https://www.sgi.com/>. Silicon Graphics, Inc. (SGI) est une entreprise américaine spécialisée dans la conception de stations de travail et de superordinateurs. Pionnier dans le domaine de l'informatique graphique, elle est à l'origine de la création d'OpenGL, une API de rendu 2D et 3D largement adoptée dans l'industrie. Leur modèle de pipeline graphique a influencé de nombreux moteurs de jeu modernes, ayant fait ses preuves dans l'industrie du jeu vidéo lors de leur collaboration avec Nintendo en 1996 pour la création de la Nintendo 64 [10].
- [10] Fabio Copetti, *Nintendo 64*. Disponible en ligne : <https://www.copetti.org/writings/consoles/nintendo-64/>. Cet article offre une analyse détaillée de la console Nintendo 64, y compris son architecture matérielle, ses composants, et les défis techniques rencontrés lors de son développement. L'auteur explore également les innovations apportées par la N64, comme le processeur graphique et le stockage sur cartouche. Consulté le 1 avril 2025.
- [11] La Langue Française. Disponible en ligne : <https://www.lalanguefrancaise.com/>. La Langue Française est un site dédié à la langue française, proposant des ressources, des articles et des outils pour améliorer la maîtrise de la langue. Il aborde divers sujets tels que la grammaire, le vocabulaire, l'orthographe et la culture francophone. Le site est une référence pour les francophones souhaitant approfondir leur connaissance de la langue française.
- [12] Wavefront Technologies, *OBJ File Format*. Disponible en ligne : [https://en.wikipedia.org/wiki/Wavefront\\_.obj\\_file](https://en.wikipedia.org/wiki/Wavefront_.obj_file). Le format de fichier OBJ est un format de fichier standard pour représenter des objets 3D. Il est largement utilisé dans l'industrie de la modélisation 3D et est pris en charge par de nombreux logiciels de modélisation. Le format OBJ est simple et facile à lire, ce qui en fait un choix populaire pour l'échange de données 3D entre différentes applications. Il permet de stocker des informations sur la géométrie et éventuellement des textures, des matériaux et d'autres attributs associés à un objet 3D s'il est accompagné d'un fichier MTL.
- [13] Khronos Group, *OpenGL Shading Language (GLSL)*. Disponible en ligne : [https://www.khronos.org/opengl/wiki/OpenGL\\_Shading\\_Language](https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language). OpenGL Shading Language (GLSL) est un langage de programmation de shaders de haut niveau dont la syntaxe est fondée sur le langage C.

## 7 Annexes

### 7.1 Exemples de débogage

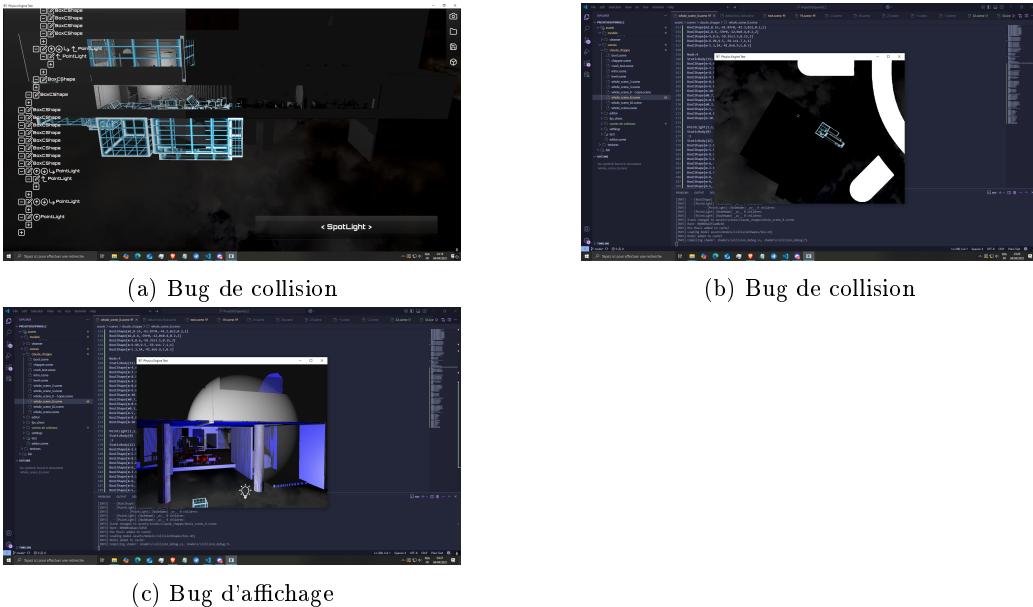


FIGURE 6 – Exemples de bug à déboguer



FIGURE 8 – Correction du bug grâce à GDB

### 7.2 Code d'une classe générique

```

1  class NomClasse : public SuperClasse {
2      __containerType__ Type_du_this *
3  private: // Les attributs privés doivent être définis en premier
4      static RessourceType ressourcePartagee;
5      Donnee locale;
6  public:
7      /**
8      * @brief Constructeur / initialiseur de la classe.
9      */
10     void constructor();
11     /**
12     * @brief Méthode générique.
13     */
14     ReturnType methode(Type param);
15     /**
16     * @brief Méthode de rendu ou de calcul.
17     */
18     void render(Context ctx);
19     /**
20     * @brief Nettoyage ou libération des ressources.
21     */
22     void destroy();
23 };

```

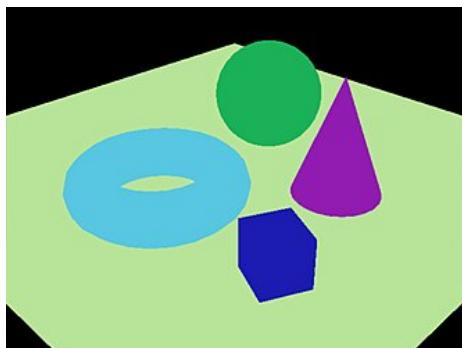
Listing 1 – Template de classe dans RaptiquaX

### 7.3 Rendu différé

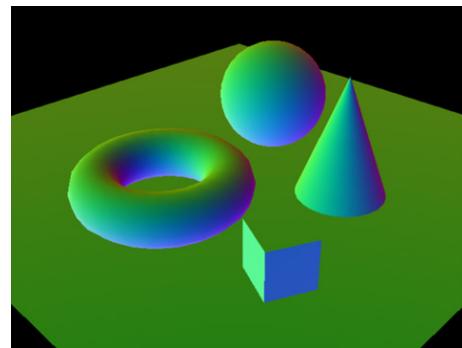
Exemple de rendu différé :

*Ici les Buffers ont été séparés pour montrer les différentes étapes du rendu différé.*

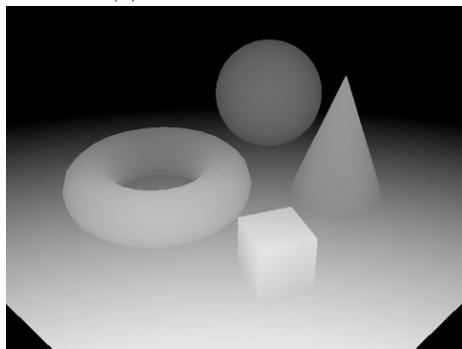
Le rendu différé s'appuie sur un G-Buffer qui contient les informations de la scène, comme la couleur, les normaux et la profondeur. Ce G-Buffer est calculé en amont, lors du rendu de la scène. C'est ensuite lors du rendu de la lumière et des effets de post-processing qu'on utilise ce G-Buffer pour éviter de recalculer les informations de la scène.



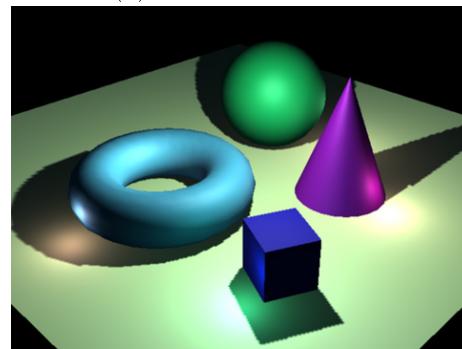
(a) G-Buffer de couleur



(b) G-Buffer de normal



(c) Z-Buffer



(d) Résultat final

FIGURE 9 – Rendu différé d'une scène de volumes simples

## 7.4 Pipeline graphique

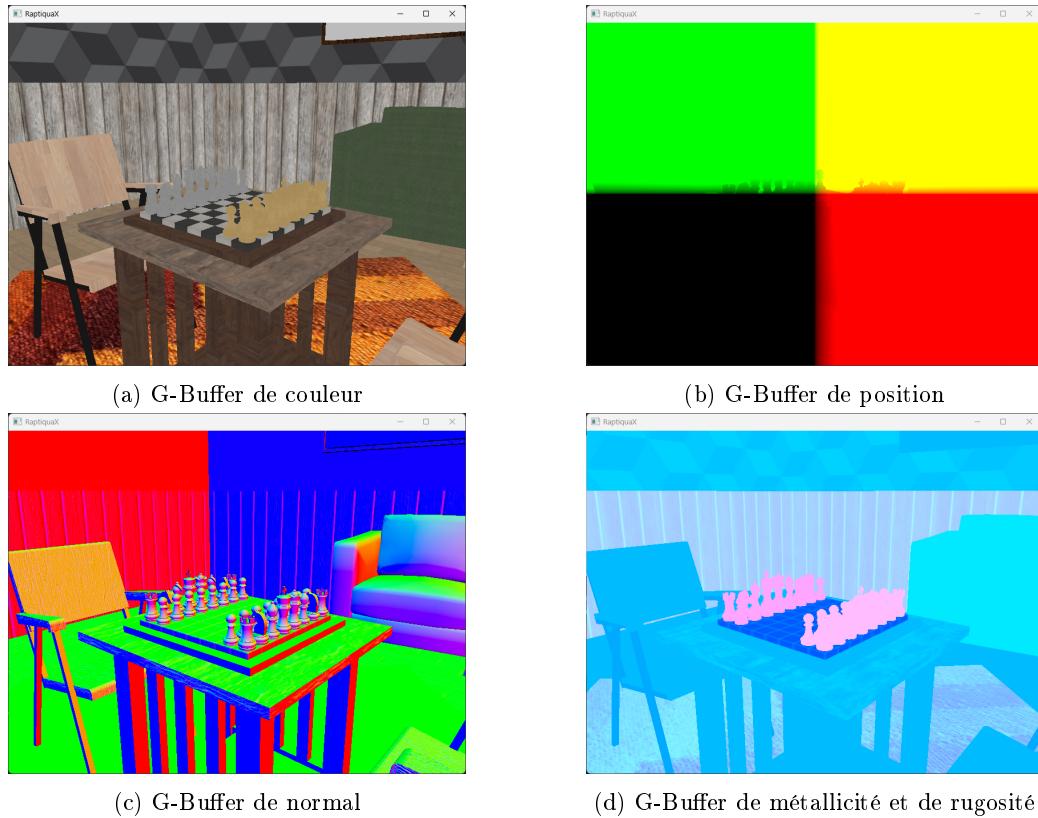
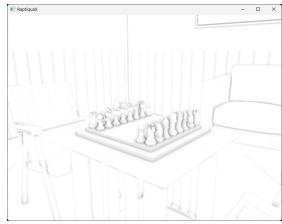
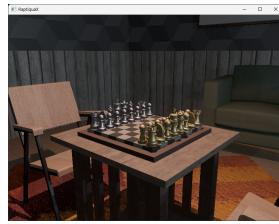


FIGURE 10 – G-Buffer de la scène



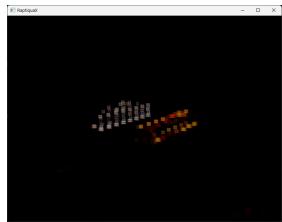
(a) Calque de l'occlusion ambiante



(b) Calque de la lumière et des ombres



(c) Calque des réflexion et reflets



(d) Calque de l'éblouissement



(e) Calque de l'interface utilisateur

FIGURE 11 – Calques de la scène

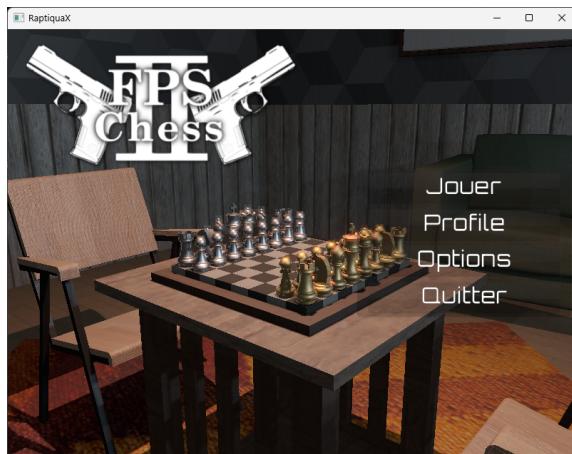


FIGURE 12 – Résultat de la pipeline graphique

## 7.5 Musique



FIGURE 13 – fl studio

## 7.6 Tableau des différents nœuds de la scène

Nœud	Description
<b>Node</b>	Classe de base pour tous les objets de la scène
<b>Camera</b>	Représente un point de vue pour le rendu
<b>BoxCShape</b>	Forme de collision en boîte
<b>CapsuleCShape</b>	Forme de collision en capsule
<b>CShape</b>	Classe de base des formes de collision
<b>MeshCShape</b>	Forme de collision basée sur un maillage
<b>PlaneCShape</b>	Plan infini utilisé pour les collisions
<b>RayCShape</b>	Détecteur de collision basé sur un rayon
<b>SphereCShape</b>	Forme de collision sphérique
<b>Button</b>	Élément d'interface cliquable
<b>CheckBox</b>	Case à cocher
<b>ControlFrame</b>	Conteneur pour organiser l'interface
<b>Frame</b>	Conteneur de base pour l'interface
<b>ImageFrame</b>	Cadre affichant une image
<b>InputArea</b>	Champ de saisie de texte
<b>Label</b>	Étiquette de texte
<b>SelectList</b>	Liste déroulante ou défilable
<b>Slider</b>	Curseur pour sélectionner une valeur
<b>DirectionalLight</b>	Source lumineuse directionnelle ( <i>ex : soleil</i> )
<b>Light</b>	Classe de base pour les sources de lumière
<b>PointLight</b>	Lumière omnidirectionnelle ponctuelle
<b>SpotLight</b>	Source lumineuse en forme de cône
<b>Mesh</b>	Maillage basique sans texture
<b>Model</b>	Modèle 3D complet et texturé
<b>Area</b>	Zone invisible déclenchant des événements
<b>Body</b>	Classe de base pour les corps physiques
<b>KinematicBody</b>	Corps physique contrôlé manuellement
<b>RigidBody</b>	Objet physique entièrement simulé
<b>StaticBody</b>	Corps physique immobile
<b>PhysicalNode</b>	Base pour les nœuds liés à la physique
<b>RenderTarget</b>	Sous scène rendu dans une texture
<b>Skybox</b>	Cube d'environnement représentant le ciel
<b>TexturedMesh</b>	Maillage avec textures appliquées

TABLE 1 – Liste des nœuds de la scène

## 7.7 Tableau des commandes Socket du mini-jeu « FPS Chess »

Commande	Arguments	Description
PONG	Aucun	Répond au ping du serveur.
LOGIN	Nom d'utilisateur	Authentifie l'utilisateur et enregistre son nom.
CREATE_PARTY	Nom de la partie	Crée une nouvelle partie avec un nom donné.
LIST_PARTY	Aucun	Liste les parties existantes.
EXIT_PARTY	Aucun	Quitte la partie actuelle.
JOIN_PARTY	ID de la partie	Rejoint une partie donnée (par ID).
RENAME_PARTY	Nouveau nom	Renomme la partie actuelle.
PARTY_SET_DATA	Clé=Valeur	Définit une valeur clé dans les données de la partie.
PARTY_GET_DATA	Clé	Récupère une valeur clé des données de la partie.
PARTY_GET_SELF_INDEX	Aucun	Récupère l'index du joueur dans la partie.
EXIT	Aucun	Déconnecte le client du serveur.
G_MSG	Message	Envoie un message global aux autres clients connectés.
MSG	Message	Envoie un message à tous les joueurs de la partie ou globalement si hors partie.

TABLE 2 – Requêtes gérées par le serveur SocketIO du moteur RaptiquaX