Compte rendu : The IMAC Light Corridor

📖 Projet de fin de deuxième semestre IMAC (Synthèse d'image)
Conception : Baptiste JOUIN *(ex-binôme Léa TOUCHARD)*

Sommaire

Introduction

Etant sans réponse de mon binôme durant le projet, celui-ci à été réalisé seul. Malgré quelques difficultés je suis plutôt satisfait du résultat visuel et techniquement (organisation du code..), à noter que j'ai découvert le C++17 et OpenGL avec la formation.

The IMAC Light Corridor est développé en C++, le repository public est disponible sur GitHub à cette adresse : https://github.com/baptistejouin/the-imac-light-corridor

En cas de problèmes de compilation, une vidéo démo est disponible à cette adresse : https://youtu.be/cS5yhfG6DRs

Installation

CMakeLists.txt

J'ai conçu ce projet principalement sous MacOS, un dérivé de Linux. J'ai donc moi-même conçu le fichier CMakeLists.txt pour pouvoir compiler le projet. Il m'a tout d'abord fallu comprendre la syntaxe de CMake et l'appliquer au projet.

Il est maintenant assez simple d'installer et de lancer le code source via `cmake .`

J'ai également implémenté une commande pour compiler puis exécuter le programme via un `make run`.

Il est nécessaire d'avoir certaines bibliothèques pour ce projet :

- GLFW3
- OpenGL

Le jeu

Commandes utilisateurs

Clique DROIT permet d'avancer.

Clique GAUCHE permet de lancer la balle si elle est sticky.

La touche K permet de lancer la pause lorsque l'on est en jeu.

La touche ESCAPE permet de fermer le jeu.

La touche Q permet de fermer le jeu dans les menus.

La touche R permet de relancer le jeu dans le menu de GameOver.

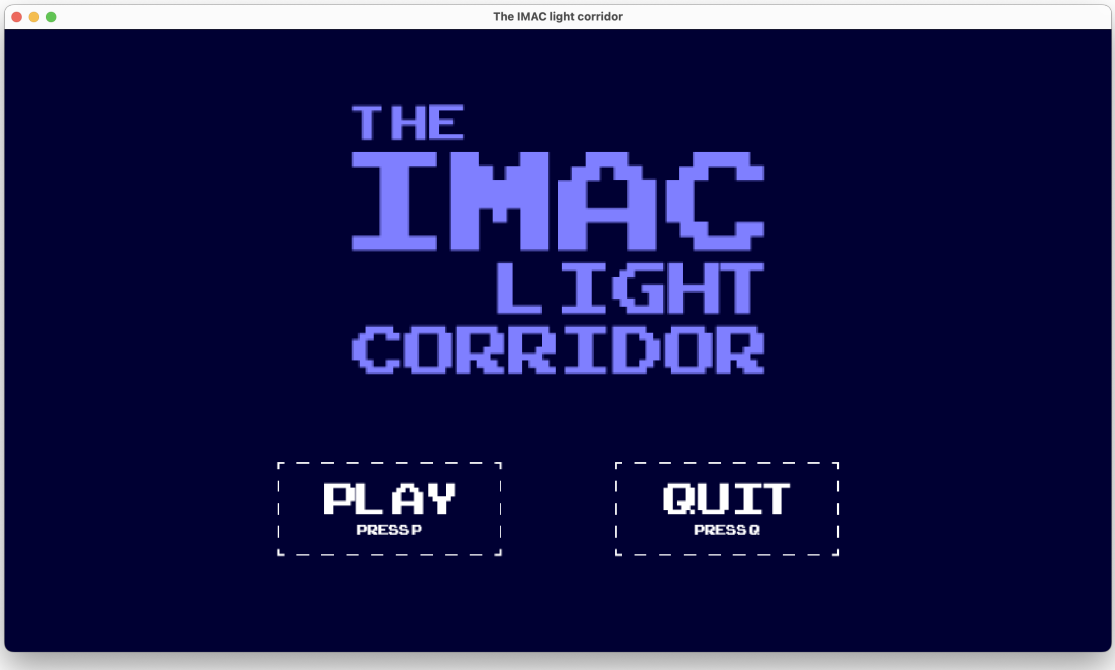
La touche P permet de lancer le jeu dans le menu.

La touche L, quand elle est maintenue, permet de voir les arrêtes.

Enfin, la touche ARROW_UP, ARROW_DOWN, ARROW_LEFT, ARROW_RIGHT, SHIFT+ARROW_LEFT, SHIFT+ARROW_RIGHT permet de débugger le jeu en se déplaçant.

Les résultats obtenus

Le jeu en image



Menu d’entré dans le jeu

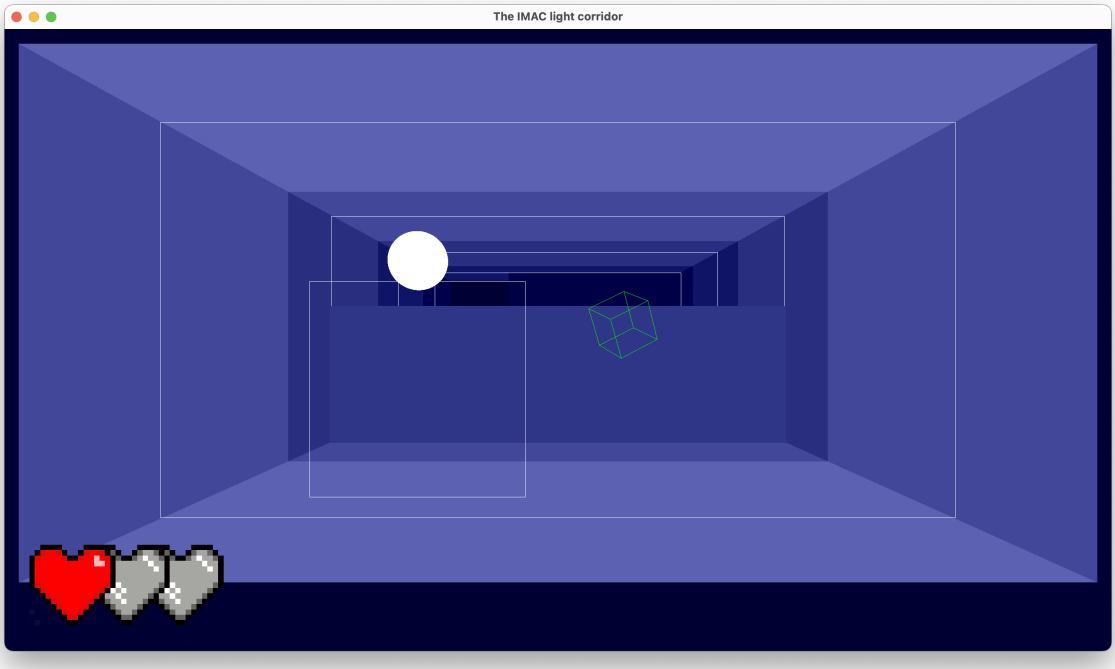


Image en jeu, on peut noter la présence de la vie du joueur, ainsi qu'un bonus de vie

Menu de pause, il peut être lancé à n'importe quel moment avec la touche K

Menu de Game Over, si le joueur n’a plus de vie

Fonctionnalité en bref

- Un menu d'entrée, pour lancer ou quitter le jeu.
- Un menu de pause.
- Un écran de fin, avec la possibilité de rejouer ou quitter.
- Le jeu comprend un système de vie, celle-ci s’affiche en jeu. Le joueur commence avec 3 vies et perd une vie chaque fois que la balle dépasse la raquette.
- En plus des vies, le jeu propose deux types de bonus. Le premier bonus, appelé "sticky", colle la balle à la raquette pour faciliter les tirs. Le deuxième bonus, appelé "vie", ajoute une vie supplémentaire au joueur (avec un maximum de 3 vies qui ne peut être dépassé). Ces bonus apparaissent sous la forme de cube 3D flottant, le joueur doit les attraper avec la raquette. Les bonus “vie” apparaissent aléatoirement sur l’axe horizontal et vertical, toujours au fond du couloir, tous les 2000 points. Les bonus sticky, eux apparaissent tous les 5000 points. (Score disponible dans le terminal, car je n’ai pas eu le temps de l’implémenter)
- J'ai également ajouté des obstacles mobiles, de taille aléatoire, la raquette ne peux pas passé au travers (ils sont donc générer pour toujours pouvoir laisser

passer la raquette), ils disparaissent en fade-out.

Architecture du projet

J'ai développé le jeu pour qu'il soit facilement et à tout moment évolutif. Cela signifie que l'ajout de fonctionnalités est normalement plus facile (comme l'ajout de bonus, par exemple). Cela a été réalisé grâce au code splitting, à l'architecture du code, mais aussi à l'architecture globale du projet.

main.cpp

Pour éviter tout problème, j'ai essayé de garder ce fichier simple et petit. La logique est immédiatement reportée sur les autres fichiers lorsque cela est possible.

game.cpp

C'est ici le vrai "coeur" du jeu, j'y ai écrit les fonctions primaires (appelées par main) :

`void initGame()`, `void gameLoop()`, `void closeGame()`, `resetGame()`

Les rôles de chaque fonction parlent d'eux-mêmes, c'est ensuite, au sein de ces fonctions que je lance mes fonctions “move” et “draw” propre à chaque éléments du jeu.

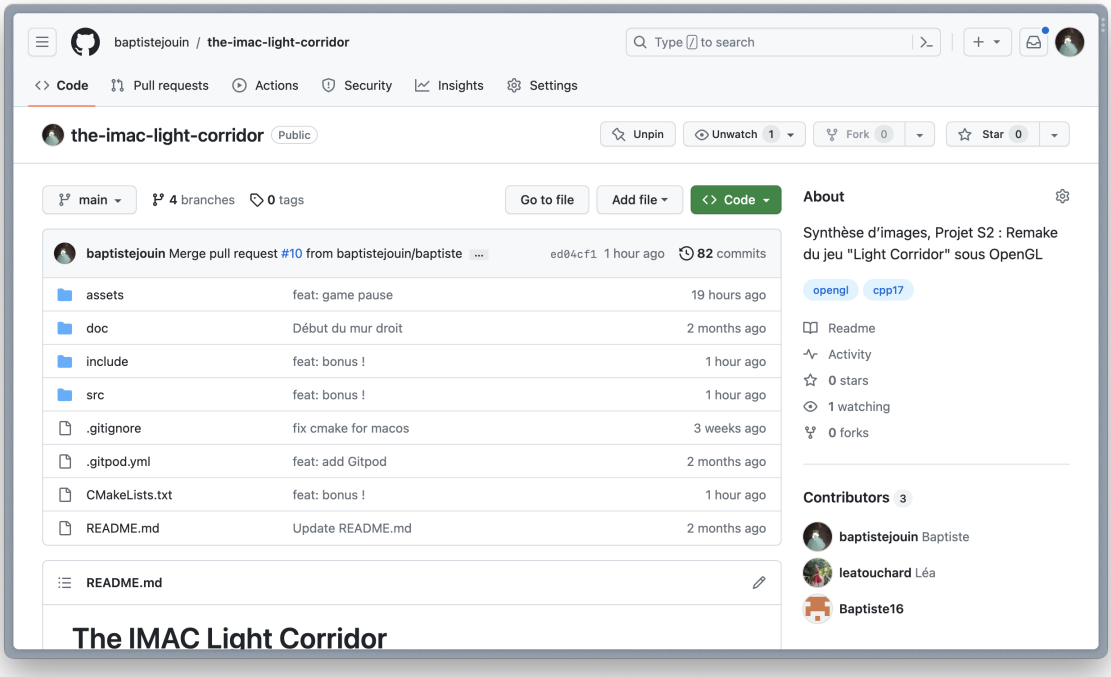
Chaque “logique”, ou entité, a son propre fichier de code et de header (nommé de manière identique pour rester organisé). J'ai écrit mes types relatifs aux éléments dans les headers et j'ai utilisé les includes quand cela était nécessaire.

Méthode de travail

Utilisation de Git avec GitHub

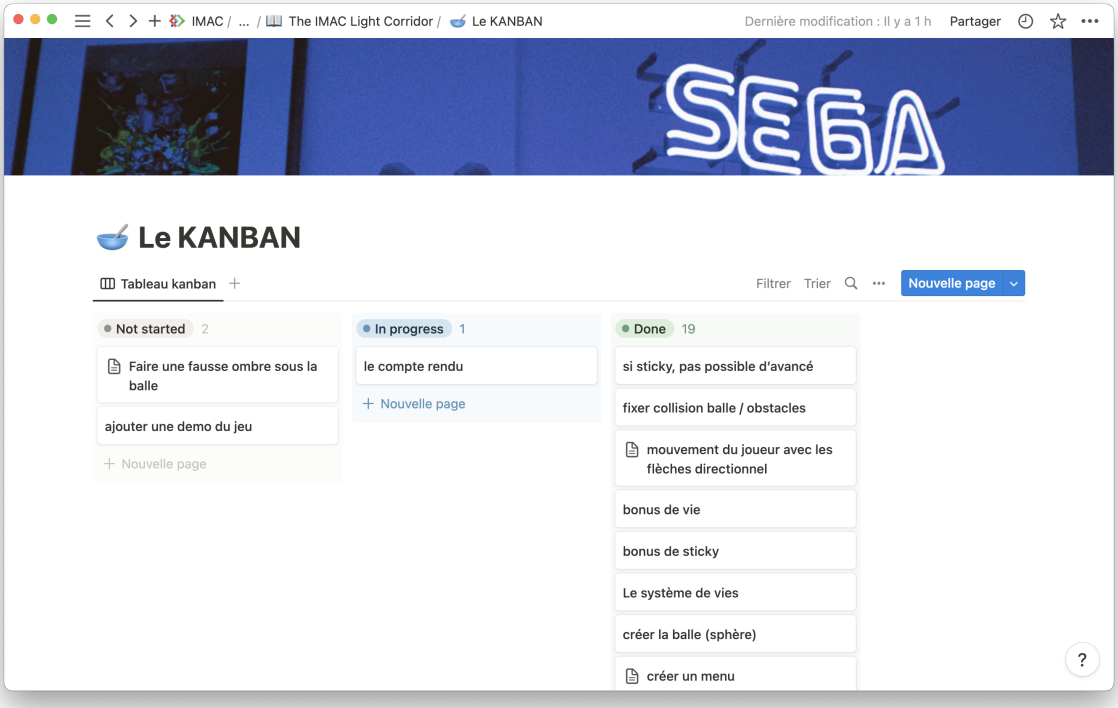
J'ai choisi de travailler avec Git/GitHub via des branches pour diviser les modifications du projet en plusieurs versions indépendantes. J'ai verrouillé la branche principale dès le début du projet, me forçant donc à utiliser des "Pull Requests" afin de fusionner (merge) mes modifications. Cela m'a permis de garder une branche principale propre et de limiter au maximum les conflits (cela a un réel intérêt en groupe, ici cela ne ma pas vraiment servit, j’ai seulement appliqué les “bonne pratique” que j’ai pu apprendre. J'ai essayé de me tenir à des conventions de

nommage pour mes commits, comme par exemple "feat" pour feature, "fix" pour un bug…



Utilisation de Notion

Dès le début du projet, j'ai créé une page Notion dédiée au projet. Il s'agit d'un tableau de bord comprenant la vidéo originale du jeu, le sujet en PDF et un Kanban pour visualiser les tâches restantes et ajouter des annotations si nécessaire. Cela s'est avéré très pratique, dès que j'avais une idée pour le projet (problème non résolut, idée d'ajout supplémentaire…) je pouvais me la noter pour plus tard.



Détails techniques

DrawSquare

J'ai implémenté une fonction `drawSquare` qui est utilisée dans de nombreuses autres fonctions. Elle permet notamment de construire le corridor, les obstacles, les vies et les menus (via des textures appliquées dessus), `drawCube`, et la raquette...

Pour pouvoir remplir toutes ces tâches, il a fallu rendre cette fonction intelligente grâce aux paramètres obligatoires/optionnels.

Par exemple, si cette fonction contient une texture (ce qui est optionnel), elle dessine un quadrilatère en gardant le bon ratio de la texture qui y est attachée.

Texture

J'ai implémenté la fonction `loadTexture` dans le but de retourner un objet bien précis :

```
typedef struct TextureLoaded
{
  GLuint textureID;
  unsigned char *stbImage;
  int width, height, nbChannels;
```

```
} TextureLoaded;
```

Les textures sont ensuite sauvegardées sous forme de pointeurs dans la structure `Game` dans un conteneur `map`, tel que `std::map<const char*, TextureLoaded> *textures;`, pour relier chaque texture à une clé.

La fonction `loadTexture` supporte tous les formats de STBI (RGB pour jpg et RGBA pour png) et donc supporte la transparence.

Pour ce qui est du design, je me suis orienté vers le côté rétro, donc via des pixel art. Cependant, les textures sont automatiquement lissées, j'ai donc ajouté les lignes nécessaires pour contrer ce phénomène.

Le menu est composé de plusieurs texture (et non pas d'une seule) pour pouvoir être changée facilement en cas de besoin.

J'ai également trouvé pertinent de faire toutes les textures de cœur dans une seule et même image (donc 4 cœurs disponibles dans 1 seule image) et ensuite de les diviser au sein du programme en 4 textures différentes. Un fichier pour toutes les dérivées du coeur. J'ai vu plusieurs que plusieurs jeux implémentaient des textures de cette façon, ce qui ma motivé dans ce choix.

Difficultés rencontrées

La principale difficulté que j'ai pu rencontrer a probablement été l'ajout des headers dans chacun des fichiers, il m'a fallu éviter les redondances pour éviter les boucles infinies et les erreurs de compilations. Le `#pragma once` ne suffisait pas toujours.

Comme dit précédemment j'ai développé le jeu seul, ce qui rendait l'avancement plus long comparé aux binômes ; cependant, si je devais trouver un avantage a cela, je dirais que j'ai n'ai pas eu besoin de scindé le travail ou résoudre des conflits de commit, et j'ai pu construire le projet comme je l'imaginais à l'origine.

Amélioration possible

Il est possible d'ajouter des bonus. Je pense que l'ajout serait peu demandeur en temps avec la structure existante (je pense à l'agrandissement de la raquette, ou encore le coeur gelé pendant un certain nombre de secondes (j'avais implémenté la texture en prévision)) .

Pour améliorer la gestion du code, il aurait été préférable de diviser davantage le code en utilisant des dossiers et des sous-dossiers. Cependant, en raison des contraintes de temps, la structure actuelle a été maintenue, mais elle reste tout de même bien structurée selon moi.

Actuellement, la simulation de la lumière dans le jeu est semi-réaliste, plus ou moins semblable au jeu original selon mes observations. Cela présente l'avantage d'optimiser les performances du jeu, mais la lumière ne paraît pas réaliste.

Enfin, si j'avais disposé de plus de temps, j'aurais aimé ajouter l'affichage du score ainsi que des effets sonores pour enrichir l'expérience de jeu. J’aurais également aimé ajouter une ombre sous la balle (je souhaitais créer un disque qui suivrait la balle selon l’axe de profondeur et qui changerait d’opacité et de taille suivant son positionnement vertical).