EKTOKLEPTO

Projet réalisé par Axel Dona et Mattéo Leclercq

Sommaire

Livrable

https://github.com/MatteoL-W/EKTOKLEPTO

Dossier

I - Présentation du jeu et design

a - Histoire

Dans EKTOKLEPTO, vous incarnez le personnage éponyme, un fantôme voleur d'œuvres d'art. Il utilise ses pouvoirs de possession pour entrer dans les tableaux et les déplacer hors des musées pour les dérober. Vous devez donc prendre possession des différents tableaux du musée et les mettre en lieu sûr.

b - Design

Nous avons designé les différents éléments du jeu en prenant en compte le fait que celui-ci ne laisserait pas de place à la narration pour expliquer l'histoire. Il fallait donc que le design soit assez explicite : du fantôme cartoonesque équipé pour son cambriolage, aux tableaux célèbres qui doivent être assez reconnaissables tout en étant de petite taille.

c - Utilisation

Le bloc joueur peut être déplacé à gauche et à droite avec les **flèches**, ainsi qu'en sautant avec **ESPACE**. Après un saut, le joueur peut faire un **double saut**, ou, s'il est positionné contre une paroi, effectuer un **walljump** qui l'envoie plus haut, et dans la direction opposée.

Le menu d'accueil est utilisable à la souris.

TOUCHE	**UTILITÉ**
FLECHES	Déplacer le joueur / déplacer le curseur dans le menu
TAB	Changer de joueur / changer d'options dans le menu

1-3	Changer les joueurs à partir de leur id.
ECHAP	Accéder au menu du jeu
ESPACE	Sauter
R	Réinitialiser le niveau actuel

II - Architecture

Lors de ce projet, nous avons essayé de cumuler toutes les bonnes pratiques que nous avons pu apprendre lors de nos projets personnels ou les cours. Cela se traduit par un effort pour la subdivision des classes (qui, certes, peut toujours être amélioré notamment pour Player et Map), d'une rédaction du code en anglais et commenté uniquement lorsque c'est nécessaire.

a - CMAKE

Nous avons développé le projet à l'aide d'un cmake. Une grande partie de celui-ci a été récupérée de nos anciens projets. Remerciement à **Enguerrand** pour la partie d'installation de GLM, une librairie mathématique que nous avons essentiellement utilisé pour les vecteurs.

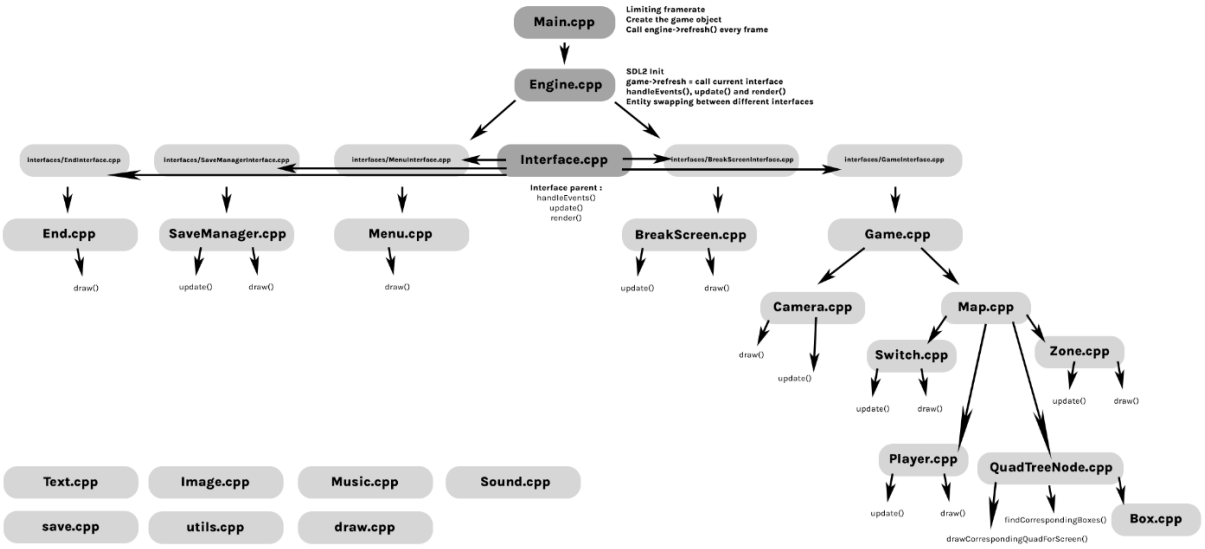
Pour l'installation de SDL ainsi que ses librairies annexes (SDL2_IMAGE, SDL2_TTF, SDL2_MIXER) sur Linux, nous avons utilisé un submodule git pointant vers le repository GitHub sdl2-cmake-modules réalisé par **aminosbh**.

Pour Windows, c'était un peu plus compliqué. Le CMAKE est fonctionnel pour une configuration sous MinGW-32. Nous avons dû créer un dossier **lib/** contenant les 4 librairies et effectuer une copie des DLL et entêtes dans le dossier **bin/** où est créée l'exécutable. Il est tout de même nécessaire pour le faire fonctionner de déplacer les dll dans les fichiers du compilateur.

b - Interfaces & POO

Nous avons décidé de mener notre projet à l'aide de la programmation orientée objet. L'intérêt de cette technologie est multiple. D'une part, elle permet la création d'architectures complètes tout en bénéficiant des avantages de l'encapsulation, de la composition et des héritages. Nous avons notamment utilisé le principe des classes abstraites (et même des interfaces) pour la classe Interface.hpp. Elle est parente de

toutes les interfaces créées : InventoryInterface.cpp, ExplorationInterface.cpp, etc… Cela s’apparente au design pattern **State**.



Une idée intéressante aurait été de faire hériter Player et Box d’une entité commune puisqu’ils ont certains points communs (la forme, les méthodes de dessins et d’update). Nous avons malheureusement pensé trop tard à ce détail qui, pour être mis en place, exigeait un refactoring trop important.

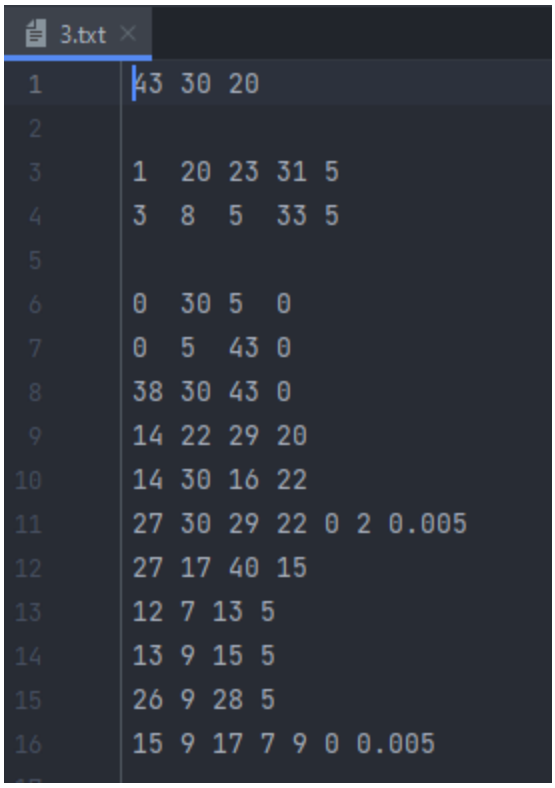
c - Fichiers maps

Fichier d’usages

usage.txt

```
// Création de la map
[mapWidth] [mapHeight] [mapZoom]

// On positionne tous nos joueurs (quel type ? position de départ ? position de fin))
// [idJoueur] [x(positionDepartCarré1)] [y(positionDepartCarré1)] [x(positionFinaleCarré1)] [y(positionFinaleCarré1)]
// [idJoueur] [x(positionDepartCarré2)] [y(positionDepartCarré2)] [x(positionFinaleCarré2)] [y(positionFinaleCarré2)]
// [idJoueur] [x(positionDepartCarréN)] [y(positionDepartCarréN)] [x(positionFinaleCarréN)] [y(positionFinaleCarréN)]

// Boxes // xMax, yMax and speed et optionnels
[xTopLeft] [yTopLeft] [xBottomRight] [yBottomRight] [xMax?] [yMax?] [speed?]

// Interrupteur
[idSwitch] [idType] [xInterrupteur] [yInterrupteur]

// Zones // Lien interrupteur (idSwitchReference) optionnel
[idChangement] [xZoneTopLeft] [yZoneTopLeft] [xZoneBottomRight] [yZoneBottomRight] [idSwitchReference?]
[idChangement] [xZone2TopLeft] [yZone2TopLeft] [xZone2BottomRight] [yZone2BottomRight] [idSwitchReference?]
[idChangement] [xZone3TopLeft] [yZone3TopLeft] [xZone3BottomRight] [yZone3BottomRight] [idSwitchReference?]

```

Exemples de cartes



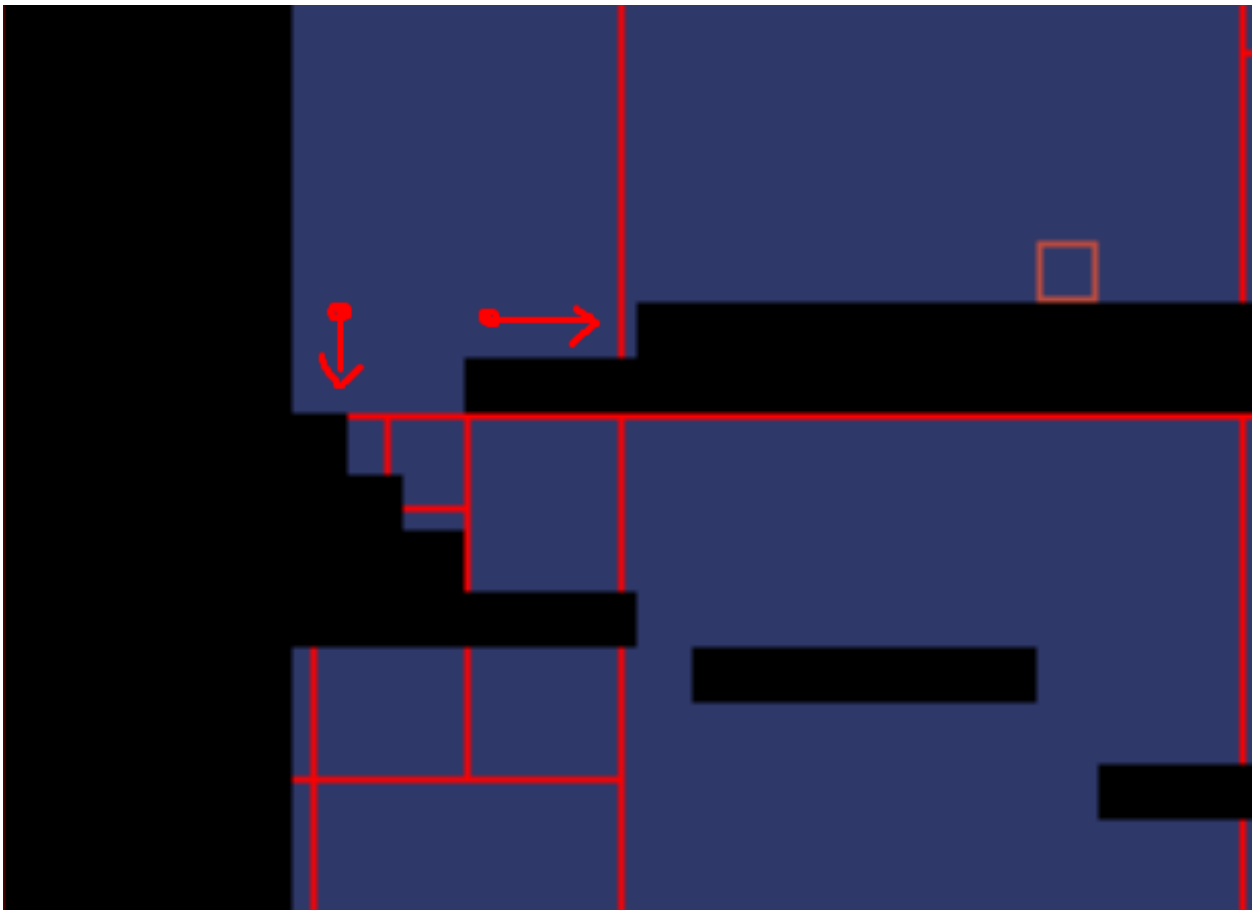
Nous avons mis en place, pour la création des cartes, un système de création textuel souple (notamment grâce à des entrées optionnelles) et capable de modifier l'intégralité des paramètres d'une carte.

III - Fonctionnalités

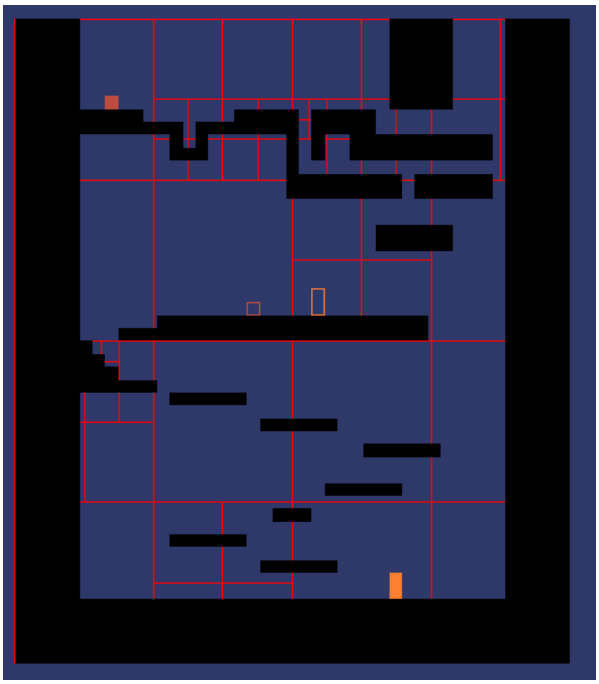
a - Quadtree et optimisation

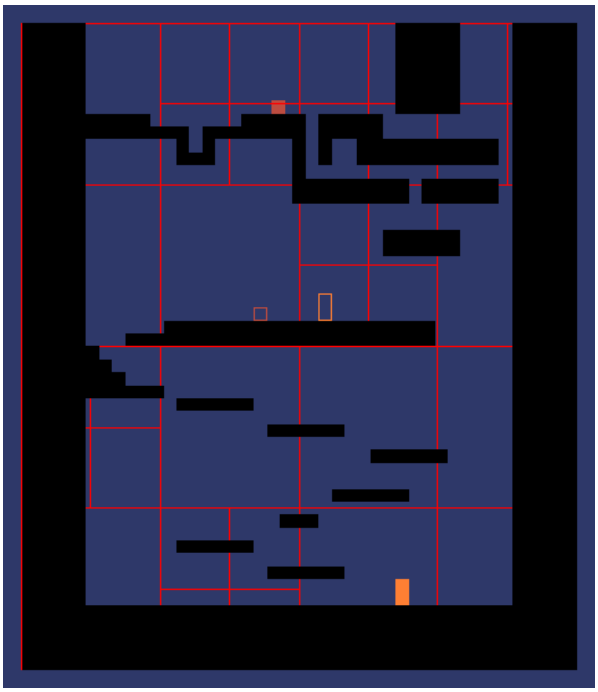
Notre quadtree nous est utile pour deux cas de figure.

D'une part, il nous est utile pour récupérer les boites autour du joueur afin de calculer les collisions. Il s'agit de la fonction **findCorrespondingBoxes()**. Pour être le plus fonctionnel possible, nous avons limité la récursion. Lors du passage des coins du joueur en paramètre, nous rajoutons quelques unités supplémentaires de hauteur et de largeur. Cela permet de limiter les cas de figure où une collision se trouve à la limite d'un noeud. Voici un exemple :



AVANT LIMITE DE RECURSION	APRES LIMITE DE RECURSION





D’autre part, nous utilisons le quadtree pour effectuer un rendu plus optimisé. En effet, nous avons développé une fonction **drawCorrespondingQuadsForScreen()** qui permet de dessiner les noeuds qui apparaissent à l’écran. Cela permet de réguler les performances et de ne pas générer un rendu inutile.

b - Interrupteur et Zones

Nous avons repris la mécanique des interrupteurs de Thomas Was alone. Nous avons implémenté des zones qui effectuent des modifications sur le joueur entrant dans la zone ou sur sa physique (changement de gravité, minimisation et maximisation de taille, super jump). Ces zones peuvent être actionnées par des interrupteurs ou par défaut dès la création de la carte.

c - Outils Music, Sound, Image et Text

Nous avons développé des outils nommés Text, Image, Sound, Music qui permettent de créer très facilement des éléments à l’aide des librairies annexes de SDL (SDL_TTF, image, mixer). Cela nous a permis de faire un code propre même avec beaucoup d’éléments créés, notamment pour le menu.

d - Blocs mouvants

Nous avons laissé la possibilité de créer des blocs mouvants entre 2 points. Cette création s'effectue dans le fichier texte des cartes. Il suffit de passer un maximum de translation en x, en y ainsi qu'une vitesse. Les pointeurs des blocs mouvants sont ajoutés dans tous les noeuds du quadtree dans lesquelles ils se déplacent.

e - Double saut et paroi verticale

Nous avons instauré une mécanique supplémentaire par rapport au jeu initial. Il s'agit de la faculté d'effectuer un double saut. Il existe aussi la possibilité de sauter à partir d'une paroi verticale.

f - Sauvegardes

Nous avons crée un manager de sauvegardes basé sur de l'écriture dans un fichier txt. 5 emplacements de sauvegardes sont disponibles et fonctionnels.

g - Caméra fluide

Lors du changement de joueur, la caméra s'adapte et se redirige vers le nouveau joueur. Ce mouvement est fluide par sa déccélération à l'approche du nouveau joueur ciblé.

h - Menu

Un menu permettant d'accéder aux sauvegardes ou de quitter est accessible en utilisant la touche **ECHAP**.

i - Zoom progressif

Au début de chaque carte et pour permettre à l'utilisateur de repérer rapidement les alentours, la camera est assez large puis réalise un zoom sur le joueur ciblé. Le niveau actuel s'affiche par la même occasion.

j - Sons et musique

Pour ajouter du dynamisme et de la vie au jeu, des sons et une musique d'ambiance sont utilisés à l'aide de la librairie SDL Mixer.

k - Mouvement et collisions

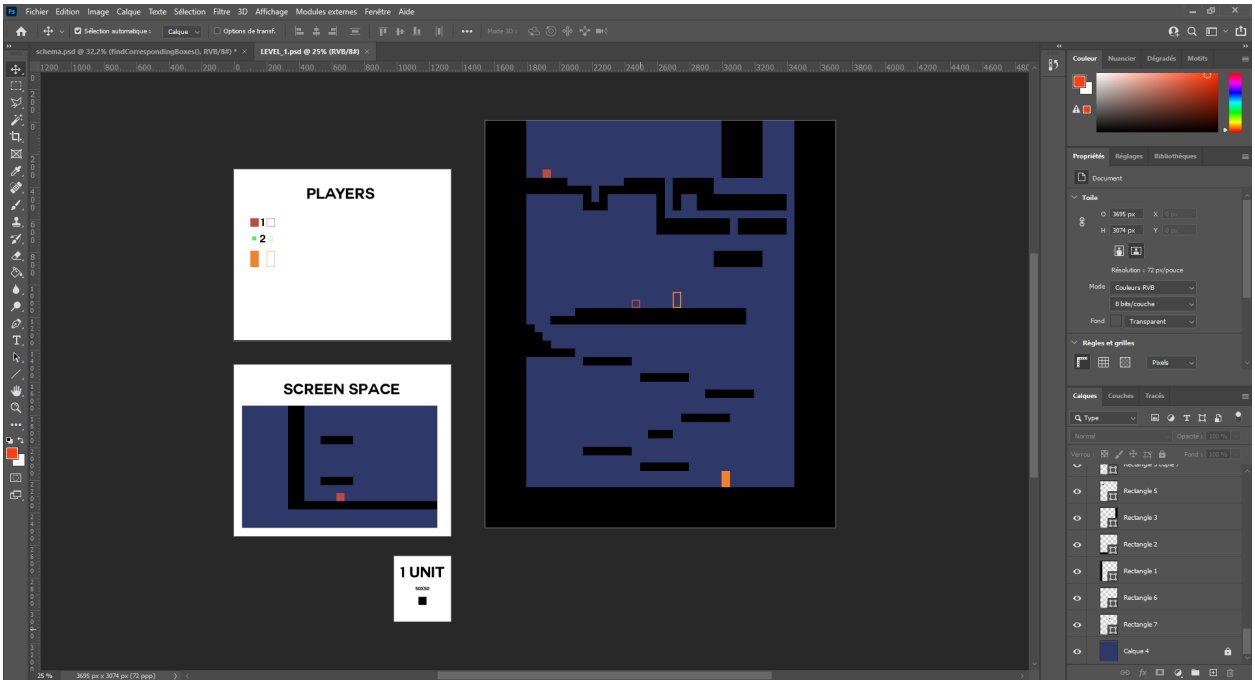
Afin d’obtenir un mouvement avec inertie, les blocs joueurs disposent de valeurs de **vitesse** et d’**accélération**. Celle-ci est située entre 0 et 1 (sauf modificateurs particuliers) et agit comme un multiplicateur de la vitesse maximale.

Les collisions entre les blocs (**joueur/environnement** ou **joueur/joueur**) sont gérées en récupérant les blocs proches du joueur à l’aide du quadtree, puis en cherchant d’éventuelles intersections. La position, et si besoin, l’accélération et la vitesse du joueur sont adaptées en conséquence. Les collisions sont donc vérifiées dans **deux sens** (du bloc joueur vers le bloc qu’il intersecte, et l’inverse, car la taille d’un bloc et son éventuel déplacement peuvent modifier comment l’intersection est détectée), et ce pour les deux types de collisions (joueur/environnement et joueur/joueur).

III - Gestion de projet

a - Helper Photoshop

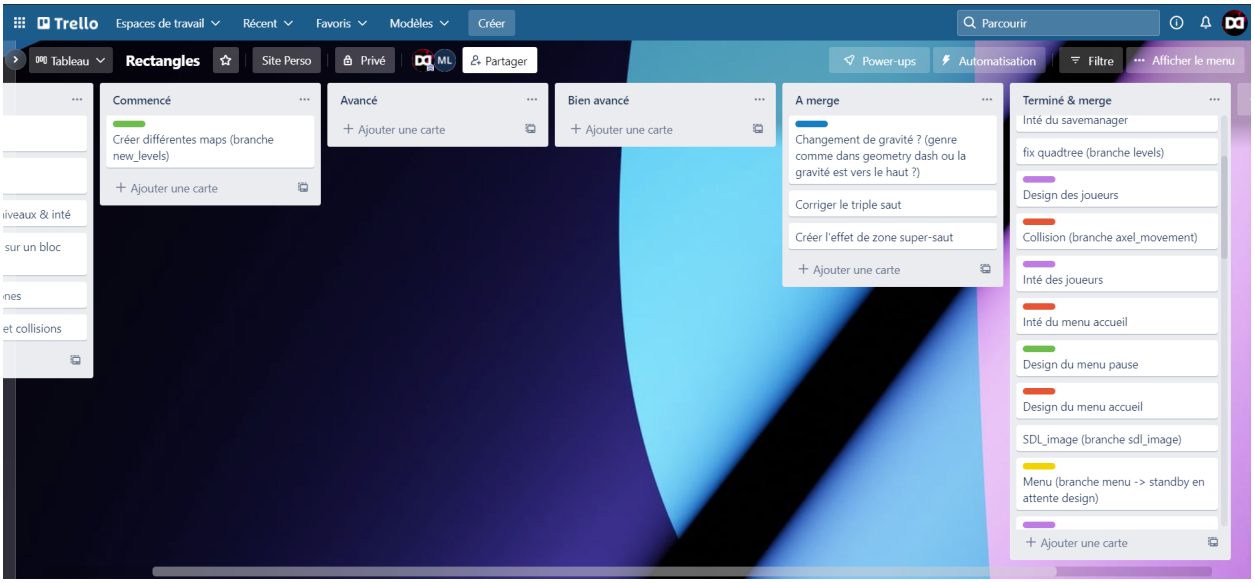
Pour s’aider lors de la création des cartes, nous avons créé un fichier Photoshop



b - Trello

Pour la répartition des tâches, nous avons utilisé l’outil en ligne Trello. Il nous a permis de compartimenter et synchroniser le travail de chacun et de permettre une vue

d’ensemble de l’avancement du projet en direct.



c - Git

Nous avons utilisé git pour comme système de versioning. Nous avons essayé de systématiquement créeer une branche pour chaque nouvelle fonctionnalité (18 branches créées à l’heure actuelle).