BENLAMINE Mehdi – FUMEY David – TEIXEIRA-RICCI Maxime
Master 2 - IMAGINA

Projet Moteur de Jeu

GraviCube

Documentation

I. Introduction

Ce projet est l'expression d'une envie de créer un jeu dans sa globalité et de se confronter à des problèmes de conceptions que les moteurs de jeu (comme *Unity3D* et *Unreal Engine*) nous protègent.

Il était intéressant de pouvoir concevoir une architecture par nous-même, et de réaliser un jeu sans les contraintes de ces puissants outils.

Le projet GIT est disponible à cette adresse : https://github.com/maxime-teixeirariccci/ProjectMDJ

Dans ces quelques pages, nous allons aborder les différents aspects de notre projet : de notre conception du jeu à l'implémentation en passant par nos difficultés et les solutions que nous avons mises en place.

Nous vous conseillons de jouer à la manette (Manette reconnue connue : Manette Xbox 360, Manette Xbox Pro, Manette Wii U Pro)

A. Inspiration

Illustration 1 - *Super Mario 3D World : Captain Toad Goes Forth*

Attacher à l'idée de réussir dans les temps un projet relativement complet, touchant plusieurs aspects de la création d'un jeu vidéo moderne, nous avons donc décidé de réaliser un prototype basé sur les niveaux du jeu « Super Mario 3D World » où le personnage de Captain Toad se retrouve dans des niveaux de taille réduite où le joueur doit contrôler le personnage afin de récupérer les cinq étoiles qui sont disséminées sur le terrain.

L'idée est donc de reprendre à notre compte cette mécanique de gameplay et de l'agrémenter de petite idée que nous avons.

B. Notre projet.

A travers les niveaux, le joueur devra être observateur et trouver les étoiles qui s'y cachent. Notre projet rajoute une mécanique de gameplay par rapport au jeu original : l'inversion de la gravité. Nous avons pu utiliser notre imagination pour réaliser des niveaux plus ou moins complexes afin de mettre en avant les fonctionnalités de notre jeu.

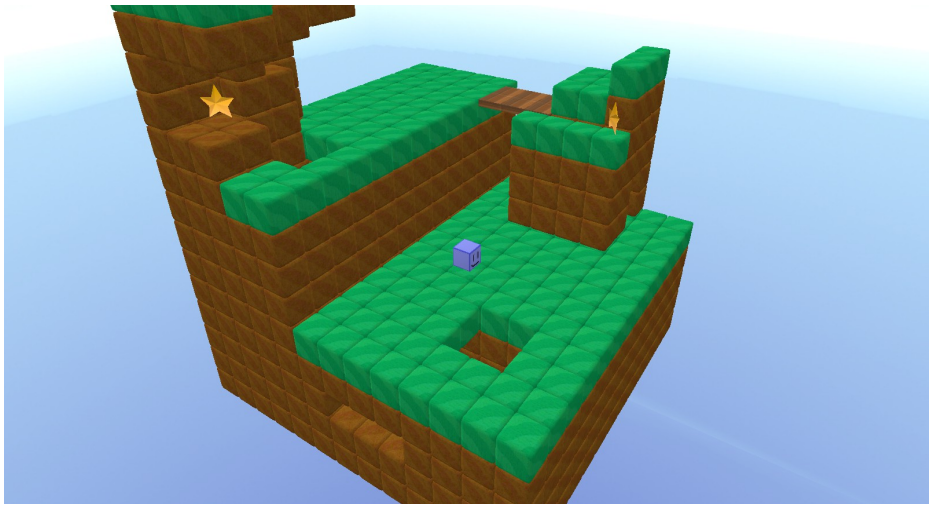
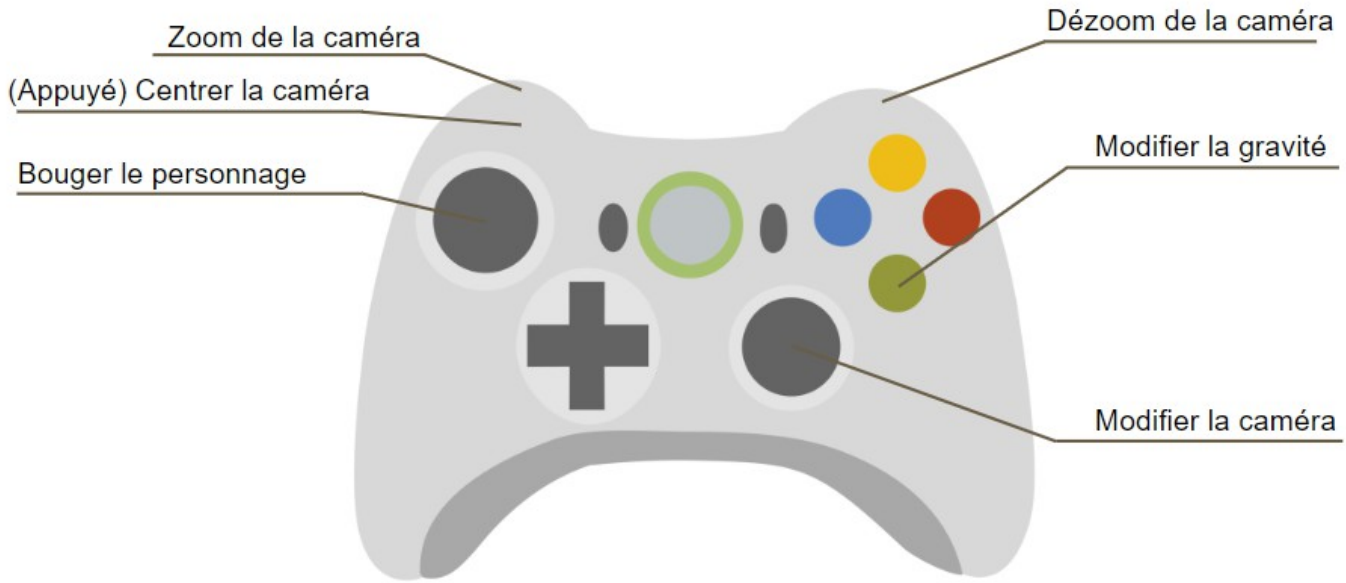


Illustration 2: GraviCube

C. Contrôles

i. Manette



ii. Clavier

ZQSD : Bouger le personnage

LEFT-RIGHT-UP-DOWN : Modifier la caméra

SPACE : Modifier la gravite

II. Programmation

A. Game Loop

Le déroulement de la game loop se passe dans le fichier *mainwidget.ccp.* Ce fichier est le fichier principal de notre projet. Dans ce fichier, deux fonctions sont importantes. En effet, la fonction void *Update*() et la fonction void *paintGL*().

La fonction *Update*() s'occupe de tous les calculs nécessaires pour le bon déroulement du jeu. Cette fonction se découpe en plusieurs phases :

- Le calcul de la position de la caméra en fonction de la position du joueur et des contrôles du joueur
- L 'exécution des composants de tous les GameObjects.
- Le test pour savoir si le joueur est en dehors des limites du jeu
- Le test pour savoir si le joueur a obtenu toutes les étoiles.

La fonction *paintGL*(), elle, s'occupe de l'affichage des GameObjects. Pour ce faire, elle calcule la matrice MVP (Model View Projection) et l'envoie au shader. Cette fonction met à jour aussi les buffers pour l'affichage.

B. Collisions

Les GameObjects peuvent posséder un collider. Ce collider permettra à l'objet d'avoir une représentation physique. Pour l'instant, le seul composant collider disponible est un Box Collider. La classe *boxcollidercomponent.cpp* contient plusieurs méthodes permettant d'utiliser ce collider :

bool Collide(BoxColliderComponent *collider) :

Cette fonction, qui renvoie un booléen, permet de déterminer si deux colliders sont en collisions. Étant donnée que ces colliders sont des boîtes, les calculs sont simples et rapide.

void *Move*(QVector3D moveVect):

En utilisant la fonction Collide, cette fonction permet de déplacer le collider en fonction d'un vecteur moveVect. Cette fonction teste donc s'il est possible à ce collider de se déplacer sans entrer en collision avec un autre collider. S'il n'y a pas d'obstacle, le collider va déplacer le gameobject selon le vecteur souhaité.

Cependant, si un autre collider rentre en collision, alors on va diviser le vecteur de déplacement moveVect par deux, pour essayer de se rapprocher du gameobject. Cette méthode par dichotomie nous permet un gain de temps et de performance non négligeable. Si après 10 divisions successives, il persiste une collisions, on ne bouge pas le collider et l'objet reste à sa place.

void *Teleport*(QVector3D pos):

Cette simple fonction permet de téléporter à la position pos s'il n'y a pas de collisions à cette position.

C. Maillages & Optimisations

i. *Maillages*

La classe *mesh.cpp* comporte plusieurs méthodes permettant la gestion d'objets 3D au format OBJ. Cette classe comporte aussi plusieurs structures :

- La structure MeshIdentity s'occupe de garder les listes des sommets et des indices, ainsi que les caractéristiques des sommets (position, texture, normales et couleurs).
- La structure VertexData récupère les caractéristiques des sommets (position, texture, normales et couleurs).

En plus de ces structures, la classe possède plusieurs méthodes qui permet de gérer les différentes données.

void Load(*const* QString fileName)

Récupère les données d'un fichier OBJ. Enregistre dans un MeshIdentity les valeurs des sommets ainsi que toutes les caractéristiques.

void LoadTexture(*const* QString fileName)

Charge une image, et l'affecte comme texture pour ce maillage.

void Compute(Transform *transform)

Cette fonction permet de charger deux listes std::vector<VertexData> outVertexData et std::vector<GLushort> outIndexData en fonction de transform.

ii. *Optimisation & Rendu*

Le rendu des GameObjects est réalisé par le singleton de la classe MeshRenderer. Cette classe s'occupe de la gestion des buffers d'affichage. Cette classe contient, entre autres, une liste de tout les maillages nécessaires et une liste comportant un ensemble de matrice de transformation.

La fonction void Draw() s'occupe, quant à elle, de l'affichage des maillages en fonction des matrices de transformations en utilisant la fonction propre à OpenGL, *glDrawArraysInstanced*.

La fonction void Init() s'occupe de créer les différents buffers pour l'affichage, et la fonction void ComputeGameObject() met à jour les matrices de transformations dans les buffers.

D. GameObjects & Composants

i. *GameObjects*

La gestion des GameObjects a été très inspiré du moteur Unity3D. Ils sont composés de plusieurs éléments :

- Un Transform, composé d'une position, d'une rotation et d'une echelle,
- Une liste de composants,
- Un collider,
- Un index de maillage,
- Un index de l'instance de maillage,
- Un gameobject et une liste d'enfants.

ii. *Composants*

Les GameObjects possèdent une liste de composants. Les composants dérivent tous d'une classe Component, possédant une méthode void *Do*(). Cette méthode est appelée pour chaque composants dans la GameLoop. Voici les composants utilisé dans ce projets :

- ColorBlockComponent
- GravityComponent
- InvisibilityComponent
- PlayerComponent
- RotationComponent
- StarCollectComponent
- SwitchComponent

E. Gestions des Inputs

Pour la gestion des inputs, nous avons mis en place un dictionnaire dans la classe *InputMapping* de façon statique *static QMap<QString,float> inputMap*.

Ainsi, dans mainwidget.cpp, la fonction *void Joypad()* enregistre des signaux et lorsqu'une touche est modifiée, on l'enregistre dans le dictionnaire :

```
connect(gamepad, &QGamepad::axisRightYChanged, this,
    [](double value)
    {
        InputMapping::inputMap["CameraVerticalAxis"] = value;
    });
```

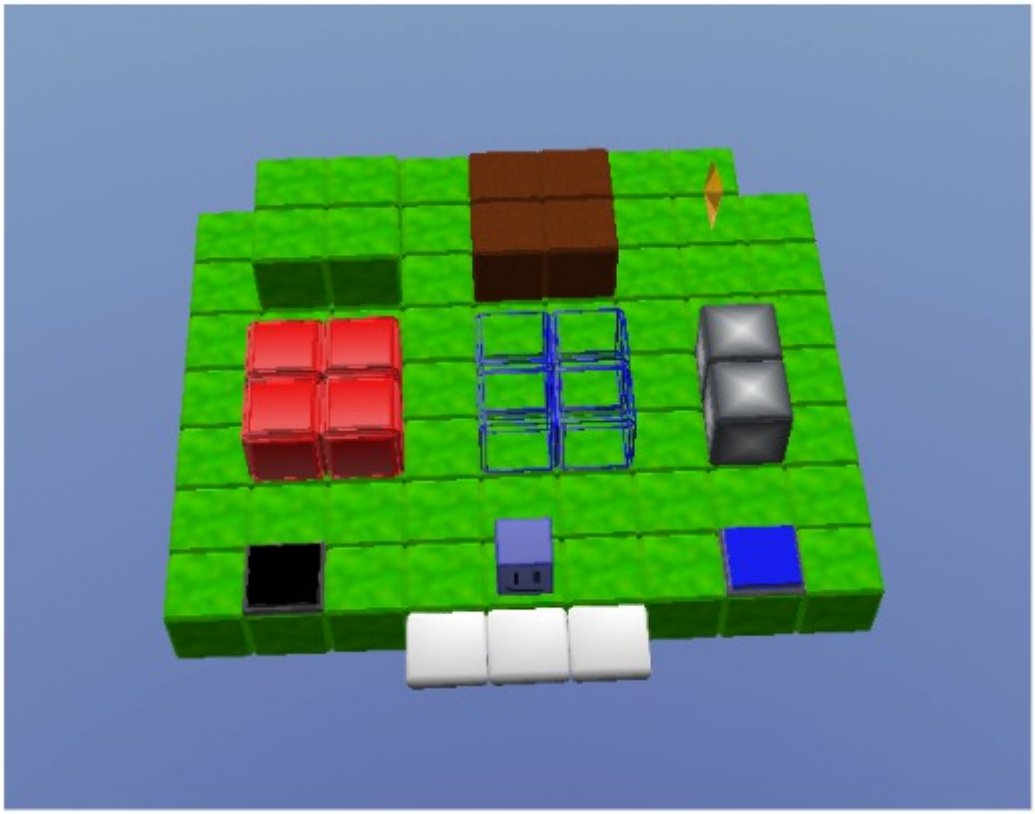
On utilise ce dictionnaire pour faire les calculs des gameobject :

```
void PlayerComponent::Move()
{
    // ...
    double dy =  InputMapping::inputMap["HorizontalAxis"];

   // ...

}
```

F. Création des niveaux

La gestion des niveaux se fait par une simple liste de GameObjects dans le mainwidget.cpp.

Afin de créer des niveaux de façon plus rapide, les niveaux sont écrit dans des fichier texte au format TXT. En utilisant un parseur dans la classe MapMaker, le jeu lit le fichier texte et positionne les blocs. Cela nous permet de créer des niveaux de façons rapides et faciles, mais aussi de pouvoir les exporter ou de les modifier en cours de jeu.



```
9 9 2
.
OOOOOOOOO
OHHOMMOCO
OHHOMMOOO
OOOOOOOOO
O11O22OXO
O11O22OXO
OOOOOOOOO
OZOOPOOQO
OOOOOOOOO
.
OHHHHHHHO
HHHHHHHHH
HHHHHHHHH
HHHHHHHHH
HHHHHHHHH
HHHHHHHHH
HHHHHHHHH
HHHHHHHHH
OIIIIIIIO
```

Illustration 3: Exemple de création de niveau

Légende :

O	- RIEN
H	- HERBE
M	- TERRE
C	- ÉTOILE
I	- NUAGE (INVISIBLE)
1	- BLOC ROUGE (ON)
2	- BLOC BLEU (OFF)
Q	- SWITCH BLEU
Z	- SWITCH ROUGE
X	- BLOC GRAVITÉ
P	- POSITION DÉPART