Université de Montpellier
Master 1 - IMAGINE

Moteur de Jeu

Gestion des différentes fonctionnalités constituant un jeu vidéo

Rapport de projet
Projet informatique — HAI819I

Étudiants :
M. Florentin DENIS
M. Khélian LARVET

Encadrant :
Mme. Noura FARAJ

Année : 2021 - 2022

Nous tenons à remercier notre encadrante Mme. Noura FARAJ pour son accompagnement, sa bienveillance et ses conseils précieux tout au long de ce projet qui nous ont permis de le porter à son état actuel.

Table des matières

Introduction

Dans le cadre du module "Moteur de Jeu" du second semestre de M1 IMAGINE, nous avons développé notre propre moteur de jeu 3D dont le dépôt est disponible à l'adresse suivante :

`https://github.com/Flare00/Moteur-de-jeu`

Un moteur de jeu est un ensemble de composants logiciels qui effectuent des calculs de géométrie et de physique utilisés dans les jeux vidéo. Ils permettent donc à une équipe de développement de gagner du temps et de se concentrer uniquement sur le contenu et le déroulement du jeu.
Plusieurs composants peuvent être implémentés dans un moteur tels que : le système (entrée/sortie, interface utilisateur), le graphisme, le son, le réseau, la physique et l'intelligence artificielle.

Notre moteur de jeu permet donc de gérer en temps réel la plupart des fonctionnalités nécessaires au bon fonctionnement d'un jeu vidéo, à savoir les fonctionnalités liées au graphisme et à la simulation physique.

Objectifs du projet et cahier des charges

L'ensemble du cahier des charges peut être résumé par les deux objectifs suivants :

Le premier et le plus important des objectifs : être capable d'initialiser un projet OpenGL et d'implémenter un maximum de fonctionnalités propres aux moteurs de jeux. Parmi ces fonctionnalités voici les plus importantes :

- Gérer les différents axes de rotation pour une caméra et son affichage dans un espace à trois dimensions.
- Utiliser les entrées claviers et souris pour produire des actions.
- Gérer les différentes coordonnées locales et globales des objets à l'aide d'un graphe de scène.
- Manipuler les "shaders" pour afficher des effets d'illumination (modèle de Phong).
- Appliquer un ensemble de lois physiques dans une scène.

Le deuxième objectif réside dans la capacité à utiliser l'ensemble des fonctionnalités développées auparavant pour produire un jeu.

Technologies utilisées

Le choix des technologies a été imposé dans l'objectif de se familiariser avec les librairies OpenGL (GLEW, GLFW, GLM, etc). Nous avons également choisi d'utiliser Discord pour communiquer et Github pour héberger notre projet avec un système de versionnage.

Discord :
Logiciel de messagerie instantanée gratuite. Il nous permet de communiquer au sein de l'équipe, de s'entraider, et d'avoir une trace de nos discussions.

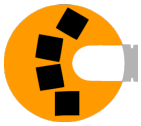
GitHub :
Service d'hébergement et de gestion de développement informatique en ligne, bâti sur le logiciel de versionnage Git. Il nous permet de garder une trace de chaque version de fichiers, en ligne (accessible par tout le monde, qu'importe l'endroit), lors du développement du projet.

C++ :
Langage de programmation compilé permettant la programmation sous de multiples paradigmes. Il nous permet d'améliorer nettement les performances de notre projet.

OpenGL :
Ensemble de fonctions de calcul d'images 2D/3D permettant à un programme de déclarer la géométrie d'objets sous forme de points, de vecteurs, de polygones, de bitmaps et de textures.

GLFW :
GLFW est une bibliothèque multiplateforme Open Source pour le développement OpenGL. Il fournit une API simple pour créer des fenêtres, des contextes et des surfaces. Il permet également de recevoir des entrées et des événements.

GLM :
GLM (OpenGL Mathematics) est une bibliothèque libre utilitaire d'OpenGL apportant au programmeur C++ tout un ensemble de classes et de fonctions permettant de manipuler les données pour OpenGL.



Bullet :
Moteur physique simulant la détection de collisions ainsi que la mécanique des corps rigides et déformables. Ce moteur nous a permis d'accéder à des fonctionnalités physiques plus poussées.

Chapitre 1

Architecture du projet

Notre projet est composé d'un ensemble de composant permettant la création d'un jeu. Ces composants sont répartis en six catégories :

— Le système de scène.
— Le système de graphe de scène avec des `GameObjects` et ses composants.
— Le système de caméra.
— Le système de contrôle entrées/sorties.
— Le système d'éclairage, avec ShadowMap.
— Le système de physique avec Bullet.

La représentation de notre architecture se trouve également dans notre annexe et nous développerons plus en détail chaque composant dans le prochain chapitre.

Chapitre 2

Implémentation des fonctionnalités

2.1 Classe Game et Scène

Les classes `Game` et `Scene` sont les plus importantes de notre projet.

La classe `Game` permet de gérer l'affichage des différentes scènes du moteur. Ainsi elle possède un ensemble de scènes dans une liste, qui, à l'aide d'un "Callback" nommé `askSceneChange(int numero)` vont pouvoir être chargées ou déchargées en mémoire.

```
1  class Game : IGlobalGameCallback {
2  private :
3      size_t activeScene = -1;
4      size_t nbScene = 0;
5      std::vector<Scene*> scenes;
6
7  public :
8      Game() {
9          // Add a Scene
10         this->scenes.push_back(new SceneLight(this));
11         this->scenes.push_back(new SceneProjectile(this));
12
13         // Scene Selector
14         this->activeScene = 1;
15
16         // Load scene
17         this->scenes[activeScene]->Load();
18
19         this->nbScene = this->scenes.size();
20     }
21
22     [...]
23
24     virtual void askSceneChange(int numero) {
25         if (numero >= 0 && numero < this->nbScene) {
26             this->scenes[this->activeScene]->UnLoad();
27             this->activeScene = numero;
28             this->scenes[this->activeScene]->Load();
29         }
30     }
31 };
```

Ainsi la fonction `Loop(float deltaTime)` présente dans `Game`, permet d'afficher à chaque "frame" la scène chargée en mémoire.

```
void Loop(float deltaTime) {
    waintingBeforeNextSceneValue -= deltaTime;
    if (this->activeScene >= 0 && this->activeScene < this->nbScene) {
        if (this->scenes[activeScene]->isActive()) {
            this->scenes[activeScene]->Draw(deltaTime);
        }
    }
}
```

La classe `Scene` quant à elle, utilise la structure de données utilisée communément par la plupart des outils de modélisation 3D et jeux, à savoir le graphe de scène. Cette structure est implémentée avec notre classe `GameObject` que nous détaillerons dans la prochaine section.

Ainsi notre classe `Scene` possède un `GameObject` correspondant à la racine du graphe de scène, une `Camera` et des "Callback" avec diverses méthodes facilitant les changements de scène et de caméra.

```
class Scene : public IGlobalSceneCallback {
protected:
    // Scene control
    IGlobalGameCallback* globalGameCallback;
    std::string id;

    // Camera control
    int activeCamera = -1;
    std::vector<Camera*> cameras;

    // Graph
    GameObject* scene;
    bool active = false;

public:
    Scene(IGlobalGameCallback* globalGameCallback = NULL, std::string id = "Scene") {
        this->globalGameCallback;
        this->id = id;
    }

    [...]
};
```

Par la suite, cette classe peut être spécialisée par des scènes particulières, permettant ainsi pour une scène donnée, de gérer ses propre éléments : moteur physique, système d'éclairage, caméras, contrôles des entrées/sorties. (cf. annexe)

2.2 Classe GameObject

La classe `GameObject` permet l'implémentation d'un graphe de scène comme énoncé précédemment en stockant son `GameObject` parent (s'il existe) et ses `GameObject` enfants.
Elle stocke par la même occasion des informations concernant ses transformations dans l'espace, sa physique mais également des informations permettant de retrouver rapidement un `GameObject` particulier dans le graphe de scène (label et type de composant).

```
class GameObject
{
protected:
    // Identification
    std::string identifier;
    std::vector<Component*> composants;

    // Graphe de Scène
    GameObject* parent;
    std::vector<GameObject*> childs;
    ITransformation* transform = NULL;

    // Physique
    BulletRigidbody* rigidBody = NULL;
    bool isBulletDependent = false;

[...]
};
```

Nous pouvons parcourir notre graphe de scène, en utilisant les célèbres parcours BFS (Breadth First Search) et DFS (Depth-first search), permettant de calculer et d'afficher rapidement tous les éléments composant le graphe de scène.

```
void computeDFS(CameraData* data,Lightning* lights){ [...] }
void computeBFS(CameraData* data, Lightning* lights){ [...] }
```

D'autres méthodes sont disponibles pour chercher des éléments particuliers dans notre graphe de scène avec par exemple la recherche d'un label ou d'un type :

```
// Informations relatives à l'identifier
GameObject* findChild(std::string identifier){ [...] }
GameObject* findDirectChild(std::string identifier) { [...] }

// Informations relatives au Component::Type
std::vector<GameObject*> getAllGameObjectByComponentType(Component::Type type) { [...] }
std::vector<Component*> getAllComponentsByTypeRecursive(Component::Type type) { [...] }
std::vector<Component*> getComponentsByType(Component::Type type) { [...] }
Component* getOneComponentByType(Component::Type type) { [...] }
```

Actuellement, seuls deux types de composant sont utilisés à savoir `Aucun` ou `Modele`. Mais nous pouvons très facilement ajouter d'autres composants dans notre moteur comme par exemple des sons.

Chaque `GameObject` possède une transformation `ITransformation* transform` qui est une interface désignant les éléments de base pour définir une transformation. Cette définition est faite dans la classe `Transformation` avec les attributs "Translate, Rotation, Scale" pour l'environnement local et global.

```
class Transformation : public ITransformation
{
protected:
    // Transformation that can be applied to descendance.
    float scale;
    glm::vec3 rotation;
    glm::vec3 translation;

    // Self Transformation
    float selfScale;
    glm::vec3 selfRotation;
    glm::vec3 selfTranslation;

    // Matrix
    glm::mat4 globalMatrix;
    glm::mat4 localMatrix;

    // Check changes
    bool globalHasBeenChanged;
    bool localHasBeenChanged;

    [...]
};
```

Le moteur physique Bullet n'ayant pas de graphe de scène et utilisant des informations de transformation globale sur chacun de ses objets, nous avons mis en place une classe de liaison entre nos données de transformation dans OpenGL et les données de transformation dans le moteur physique de Bullet.

2.3 Composant modèle

Pour créer un modèle dans notre moteur et l'afficher, nous devons créer un objet de la classe `ModeleLOD` et l'ajouter dans une scène.

```
class ModeleLOD : public GameObject
{
protected:
    // 0 : High Poly, 1 : Low Poly, 2 : Impostor
    ModeleComponent *modeles[3];
    float distanceLOD[2];

    [...]
};
```

Cette classe permet d'échanger les `ModeleComponent` en fonction des distances limites passées dans l'attribut `distanceLOD`. Cette classe est utile seulement si elle possède des `ModeleComponent` qui est la classe stockant toutes les informations d'un modèle ainsi que son "shader" :

```
class ModeleComponent : public Component
{
private:
    GlobalShader* shader;
protected:
    // Modele
    BoundingBox* boundingBox = NULL;
    std::vector<glm::vec3> vertexArray;
    std::vector<glm::vec2> texCoords;
    std::vector<glm::vec3> normals;
    std::vector<unsigned int> indices;

    // Material
    Material material;

    // Texture
    struct TextureContainer
    {
        Texture* texture = NULL;
        bool destroyAtEnd; // is texture shared?
    };
    std::vector<TextureContainer> textures;
    int heightMapId;

    // Buffers
    GLuint VAO;
    GLuint VBO[3];
    GLuint EBO;

    // Draw only if hasData
    bool hasData;

    [...]
};
```

Plusieurs constructeurs sont possibles en fonctions des informations que nous possédons sur l'objet :

```
// Constructeur par fichier externe (OFF ou OBJ)
ModeleComponent(GlobalShader* shader, FileType type, std::string file) { [...] }

// Constructeur vide
ModeleComponent(GlobalShader* shader) { [...] }

// Constructeur avec buffer
ModeleComponent(
    GlobalShader* shader,
    std::vector<glm::vec3> indexed_vertices,
    std::vector<glm::vec3> normals,
    std::vector<unsigned int> indices,
    std::vector<glm::vec2> texCoords
) { [...] }
```

Une fois que nous possédons toutes les informations nécessaires au bon fonctionnement du modèle, nous utilisons la fonction `loadBuffer()` pour charger les VAO, VBO et EBO. Puis nous pouvons afficher notre modèle avec la fonction `draw()`.

Tous les `ModeleComponent` possèdent un "shader", qui par défaut est désigné sur notre `GlobalShader` permettant d'implémenter le modèle de Phong et les "Shadow Maps".
Toutefois, nous pouvons utiliser la classe `GlobalShaderExtended` permettant d'ajouter des fonctionnalités supplémentaires en utilisant le `GlobalShader`.

Le `GlobalShaderExtended` est très intéressant pour utiliser la tessellation dans la Pipeline du rendu d'OpenGL. En utilisant la tesselation, nous pouvons indiquer au GPU comment afficher un ensemble de points par rapport à une texture de hauteur mais également comment les afficher par rapport à la caméra.

Cette méthode nous permet donc de créer des terrains rapidement à l'aide d'une carte de hauteur, et ainsi produire une subdivision assez importante pour rendre le terrain lisse. De plus, cette méthode permet de créer un LOD dynamique par l'intermédiaire de "chunks" gérer par OpenGL.

2.4 Classe Camera

Nous avons créer une classe caméra gérant les attributs "Pitch Roll Yaw" et possédant une matrice de transformation. Nous avons également mis en place des méthodes permettant de générer une caméra orbitale :

```
const float YAW = -90.0f;
const float PITCH = 0.0f;
const float ROLL = 0.0f;
const float MOVE_SPEED = 10.0f;
const float ROTATE_SPEED = 50.0f;

class Camera
{
public:
    // Nécessaire pour les ShadowMaps
    CameraData data;

    // Transformation
    glm::mat4 transformation;

    // Referentiel
    vec3 front;
    vec3 up;
    vec3 right;
    vec3 worldUp;

    // Angle d'Euler
    float yaw;
    float pitch;
    float roll;

    // Options
    float moveSpeed;
    float rotateSpeed;

    // Orbital
    vec3 cibleOrbital = vec3(0.0f);
    bool modeOrbital = false;

    // FOV
    float distanceMax = 10000.0f;
    float distanceMin = 0.01f;
    float fov = 45.0f;
    float aspect = 4.0f / 3.0f;

    // Flag
    bool dirty = true;
```

Notre `Camera` possède un "Viewing frustum" permettant selon deux méthodes, d'afficher ou non un objet dans la scène. La première méthode rapide mais peu efficace consiste à calculer si un point de l'objet se trouve dans la vue de la caméra en passant dans la quatrième dimension.
La seconde méthode consiste à récupérer la "Bounding box" de l'objet et de calculer si elle se situe dans les bornes du "Frustum".

2.5 Classes de Contrôles

Pour générer des actions à partir des entrées de l'utilisateur, nous avons créé une classe `Input` permettant de gérer a minima les changements de scène et la touche pause.
Cette classe peut par la suite être spécialisée par d'autres classes de contrôles (par exemple `InputLight` ou `InputProjectile`) permettant l'implémentation de plusieurs autres actions à l'aide de "Callback".

Ces classes de contrôles pourront ensuite être ajoutées comme paramètres lors de la création d'une scène. Voici un exemple de "Callback" utilisé dans notre `SceneProjectile` :

Nous définissons d'abord une interface "Callback" avec l'ensemble des actions virtuelles que nous voulons réaliser pour une scène donnée :

```
class IProjectileCallback
{
public:
    virtual void actionFireBall() = 0;
    [...]
};
```

Puis nous créons une classe permettant d'assigner une touche pour chaque action de l'interface "Callback" :

```
class InputProjectile : public Input
{
protected:
    IProjectileCallback *callback;
    [...]

    virtual void keyboardInput(float deltaTime)
    {
        // Récupération des input à chaque frame
        Input::keyboardInput(deltaTime);
        if (!pause)
        {
            if (glfwGetKey(global_window, GLFW_KEY_K) == GLFW_PRESS)
            {
                this->callback->actionFireBall();
            }
        }
        [...]
    }
    [...]
};
```

Enfin, nous désignons quelle est la classe contrôle à utiliser et nous implémentons les fonctions virtuelles dans la scène :

```
class SceneProjectile : public Scene, IProjectileCallback
{
private:
    InputProjectile* inputProj;
    [...]

public:
    SceneProjectile(IGlobalGameCallback* globalGameCallback) : Scene(globalGameCallback)
    {
        this->inputProj = new InputProjectile(globalGameCallback, this, this);
    }

    [...]

    virtual void actionFireBall()
    {
        // Création d'un balle
        ModeleLOD* b = this->createBall();
        this->listBall.push_back(b);
        this->scene->addChild(b);
        this->bullet->addRigidbodyToPhysique(b->getRigidBody(), canon->getGroup(), 1);

        // Ajout d'une force sur la balle
        b->getTransform()->setTranslate(this->cameras[0]->getPosition());
        b->getBulletTransform()->applyImpule(this->cameras[0]->getFront() * 2000.0f);

        [...]
    }
};
```

2.6 Classe d'Éclairage

Dans l'objectif d'augmenter le réalisme de notre moteur nous avons décidé de mettre en place un système de lumière qui est représenté par la classe `Lightning`.
Cette classe contient un tableau de lumière qui est actualisé à chaque "frame" et où chaque lumière et est défini par la classe `ILight`.

La classe `ILight` sert d'interface pour imposer le minimum requis afin de créer une lumière :

```
class ILight
{
public:
    enum Type
    {
        AUCUN,
        POINT,
        DIRECTIONAL
    };

protected:
    vec3 position;
    vec3 color;
    float intensity;
    Type type;
    bool active = true;
    bool dirty = false;
    [...]
};
```

Nous pouvons ensuite créer deux spécialisations de la classe `ILight` à savoir :
- La classe `PointLight` permettant de définir une lumière éclairant tout autour d'elle.
- La classe `DirectionnalLight` qui à l'instar `PointLight` créer une lumière mais avec un vecteur de direction est un attribut "Shadow Map".

La classe `ShadowMap` utilise un système de "FrameBuffer" afin de réaliser un calcul de profondeur sur les différents objets de la scène. Dans l'objectif de réaliser cette tâche, la classe `ShadowMap` contient son propre "shader" qui sera utile pour rendre les différents éléments de la scène du point de vue de la lumière.

```
// DirectionnalLight.hpp
virtual void compute(GameObject *scene){
    if (generateShadow){
        if (this->dirty){
            this->shadow.setMappingParameter(this->position, this->direction);
            this->dirty = false;
        }
        glCullFace(GL_FRONT);
        this->shadow.drawData();
        scene->compute(this->shadow.getData(), NULL, true);
        glCullFace(GL_BACK);
    }
}
```

2.7 Classe Physique avec Bullet

Avant l'utilisation du moteur physique Bullet, nous avions essayé d'implémenter notre propre système physique. Ce système utilisait des "Bounding box" AABB afin de détecter des collisions entre les objets et d'y appliquer des forces ou des impulsions.

Nous avons donc choisi d'utiliser le moteur physique Bullet afin d'accéder à des fonctionnalités physiques avancées. Pour ce faire, nous avons créé un ensemble de classe :

La première classe est `PhysiqueBullet` qui sert d'interface au moteur Bullet. Elle initialise le moteur, demande à chaque "frame" le calcul physique pour chaque objet dans la scène, et permet d'y rajouter des éléments physique tel que des éléments de la classe `BulletRigidbody`.
Pour améliorer les performances avec Bullet, nous l'avons configuré avec un système d'arbre dynamique AABB.

```
class PhysiqueBullet
{
public:
    // Data
    GestionContraintes *gestionContraintes = NULL;
    CollisionFilter *collisionFilter = NULL;
    btDiscreteDynamicsWorld *dynamicsWorld = NULL;
    btSequentialImpulseConstraintSolver *solver = NULL;
    btDbvtBroadphase *overlappingPairCache = NULL;
    btCollisionDispatcher *dispatcher = NULL;
    btDefaultCollisionConfiguration *collisionConfig = NULL;

    // Rigid list
    std::vector<BulletRigidbody *> rigidbodies;

    // Debug
    bool debugState = false;

    [...]
};
```

La seconde classe `BulletRigidbody`, a été créé afin de rendre plus accessible les informations des `btRigidbody` lié à Bullet, et de conserver leurs caractéristiques, telles que leur forme ou leur masse.

```
class BulletRigidbody
{
public:
    enum Type
    {
        AUCUN,
        AABB,
        SPHERE,
        CAPSULE,
        CYLINDER,
        HEIGHT_TERRAIN
    };


```

```
15
16  protected:
17      Type type = Type::AUCUN;
18      btRigidBody *rigidbody = NULL;
19      btCollisionShape *shape;
20      float masse = 0.0f;
21      glm::mat4 modelTransformation;
22
23      [...]
24  };
```

La troisième classe, `GestionContraintes` permet de relier deux éléments en utilisant des contraintes proposées par Bullet, et de les rajouter automatiquement au moteur physique.

Enfin la dernière classe `CollisionFilter` sert à gérer les groupes et masques de collision. Ainsi nous avons arbitrairement décidé les règles suivantes :

- Les éléments appartenant au groupe zéro : ne rentre jamais en collision.
- Les éléments appartenant au groupe un : entre toujours en collision.
- Si les éléments appartiennent au même groupe, alors :
  - Les éléments appartenant au masque zéro : ne rentre jamais en collision avec le reste du groupe.
  - Les éléments appartenant au même masque : entre en collision avec le reste du groupe.
  - Les éléments n'appartenant pas au même masque : ne peuvent pas rentrer en collision.

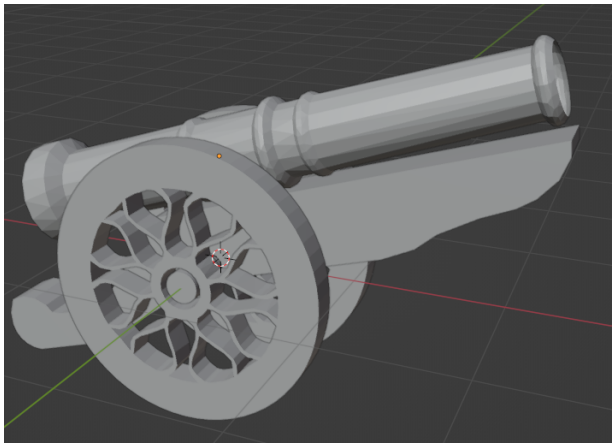
```
1   class CollisionFilter : public btOverlapFilterCallback{
2   public:
3       // Group 0 : Nothingness ; Group 1 : Everything
4       // Group n : ... obj ... (collide if mask == mask)
5       virtual bool needBroadphaseCollision(btBroadphaseProxy *proxy0, btBroadphaseProxy
        ↪  *proxy1) const
6       {
7           if (proxy0->m_collisionFilterGroup == 0 || proxy1->m_collisionFilterGroup == 0){
8               return false;
9           }
10          if (proxy0->m_collisionFilterGroup == 1 || proxy1->m_collisionFilterGroup == 1){
11              return true;
12          }
13          if (proxy0->m_collisionFilterGroup != proxy1->m_collisionFilterGroup){
14              return true;
15          }
16          if (proxy0->m_collisionFilterGroup == proxy1->m_collisionFilterGroup){
17              if (proxy0->m_collisionFilterMask == 0 || proxy1->m_collisionFilterMask ==
                ↪  0){
18                  return false;
19              }
20              else if (proxy0->m_collisionFilterMask == proxy1->m_collisionFilterMask){
21                  return true;
22              }
23              else{
24                  return false;
25              }
26          }
27          return true;
28      }
29  };
```

Chapitre 3

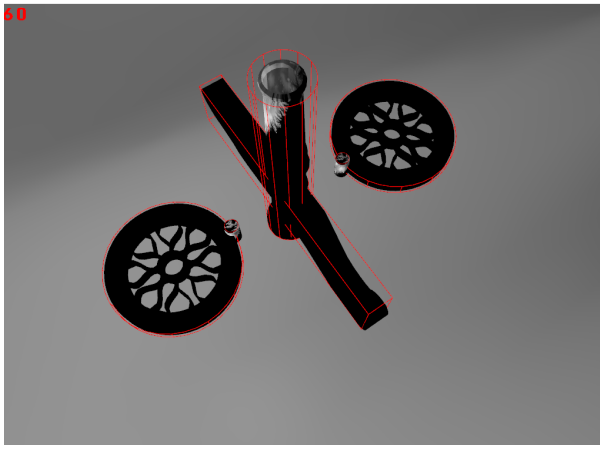
Création d'un jeu avec notre moteur

Notre moteur de jeu nous a permis de créer un petit jeu basique dont l'objectif est de tirer sur des boîtes en contrôlant un canon.

Pour ce faire, nous avons utilisé notre importateur de fichier OBJ pour modéliser un canon avec le logiciel Blender. Une fois le canon modélisé, nous avons séparé chaque pièce constituant le canon pour leur appliquer une "bounding box" et les reconstituer avec les contraintes du moteur physique Bullet dans une classe nommée `Canon`.



Modélisation du canon



Importation et application des contraintes

Nous avons ensuite mis en place les différentes actions possibles pour le canon, à savoir se déplacer et tirer des balles. Nous avons fait en sorte que le déplacement du canon s'effectue de manière physique en appliquant une force rotation sur les roues.
En ce qui concerne la projection des balles par le canon, nous avons implémenté une fonction permettant de créer une balle et de la tirer avec une certaine puissance à partir de la position du cylindre du canon :

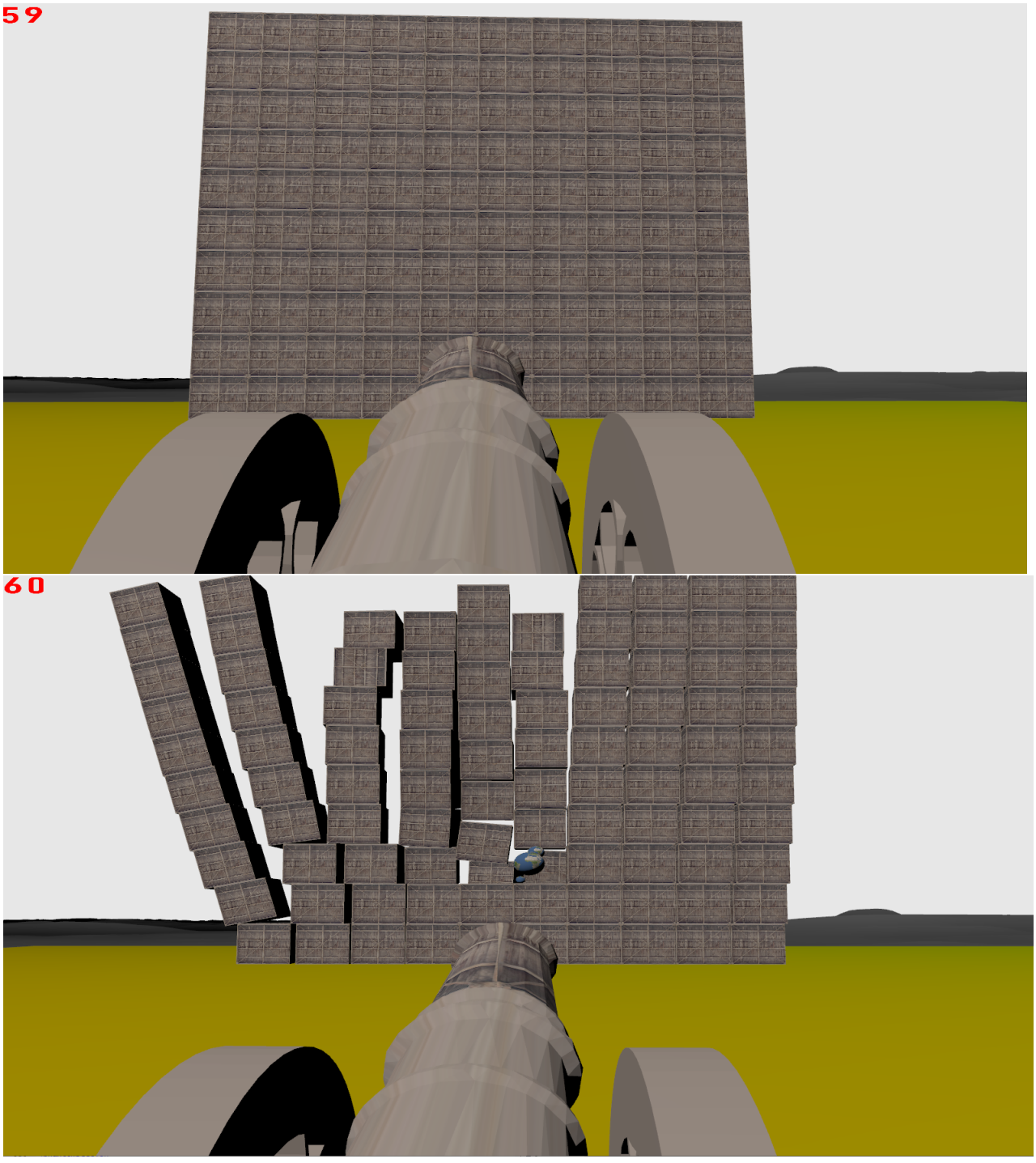
```
1  // SceneProjectile.hpp
2  ModeleLOD *createBall()
3  {
4      ModeleComponent *ballComponent = new ModeleComponent(globalShader);
5      ballComponent->addTexture(texBall, false);
6      PrimitiveMesh::generate_uv_sphere(ballComponent, 16, 16, 0.2f);
7      BulletRigidbody *ballRigidBody = new BulletRigidbody();
8      ballRigidBody->setToSphere(0.2f, 50.0f);
9      ModeleLOD *ball = new ModeleLOD("Ball", ballComponent, NULL, NULL, ballRigidBody);
10     return ball;
11 }
```

```
virtual void actionFireBall()
{
    if (this->waitTimeFireBall <= 0)
    {
        this->waitTimeFireBall = this->cooldownFireBall;
        if (this->listBall.size() > 20)
        {
            ModeleLOD *b = this->listBall[0];
            this->scene->removeChild(b);
            this->bullet->removeRigidbodyFromPhysique(b->getRigidBody());
            this->listBall.erase(this->listBall.begin());
            delete b;
        }
        if (activeCamera == 0)
        {
            ModeleLOD *b = this->createBall();
            this->listBall.push_back(b);
            this->scene->addChild(b);
            this->bullet->addRigidbodyToPhysique(b->getRigidBody(), canon->getGroup(),
            ↪  1);

            b->getTransform()->setTranslate(this->canon->getCanonPos());
            b->getBulletTransform()->applyImpule(this->canon->getFront() * 2000.0f);
        }
    }
}
```

Le canon étant opérationnel, il nous fallait un élément sur lequel tirer. C'est pourquoi nous avons mis en place une classe `Mur` permettant de créer un mur de boîte de taille x,y,z.

```
// Wall.hpp
ModeleComponent *cubeComponent = new ModeleComponent(globalShader);
cubeComponent->addTexture(texture, false);
PrimitiveMesh::generate_cube(cubeComponent);
ModeleLOD *cube = new ModeleLOD("Cube", cubeComponent, NULL, NULL, rigid, go);
glm::vec3 pos(10, -7.0f, -10.0f);
cube->getTransform()->setTranslate(pos);

for (int i = 0; i < width; i++)
{
    for (int j = 0; j < height; j++)
    {
        for (int k = 0; k < depth; k++)
        {
            if (!(i == 0 && k == 0 && j == 0))
            {
                ModeleLOD *cubeTmp = cube->duplicate();
                cubeTmp->getTransform()->setTranslate(glm::vec3((float)k * 2, (float)j *
                ↪  2, (float)i * 2) + pos);
            }
        }
    }
}
```

Nous voilà ainsi avec un jeu basique où nous pouvons contrôler un canon et tirer sur des boîtes :



Conclusion

Avancement du projet

À l'heure actuelle, notre projet est fonctionnel et possède un moteur de jeu avec un ensemble satisfaisant de fonctionnalités. Nous avons également un ensemble de classe permettant la création d'un petit jeu. Ainsi notre moteur de jeu permet :

- La gestions des entrées clavier et souris pour produire des actions dans une scène.
- La création de divers objets dans une scène à l'aide des types primitifs (Cube, Sphère, etc) ou de fichiers externes (fichiers .OBJ).
- Un rendu des objets en temps réel ("Shadow map", modèle de Phong).
- D'optimiser les performances avec diverses méthodes (LOD dynamique, tessellation, "Frustum culling").
- Une simulation physique des collisions entre les objets (Bullet).
- Une gestion de multiples scène avec chargement et déchargement de celle-ci.

Difficultés rencontrées

Notre première difficulté a été d'implémenter correctement des lois physiques dans notre scène. Nous avons réussi à simuler les 3 premières lois de Newton, mais nous avons eu beaucoup de mal à implémenter des jointures ainsi que des mouvements non linéaires. C'est pourquoi nous avons choisi d'utiliser le moteur physique bullet.

Nous avons également eu des difficultés à mettre en place le moteur physique Bullet pour accéder à des fonctionnalités physiques avancées. En effet, la documentation étant dépréciée pour le langage C++ et certains modules n'étant plus à jour, nous avons été contraints de consulter plusieurs forums pour pouvoir produire des classes fonctionnelles.
De plus le moteur physique Bullet ne possède pas de système de graphe de scène mais uniquement des contraintes. Nous avons donc dû trouver un moyen d'appliquer les transformations du monde physique de Bullet à nos `GameObjects` dans l'affichage d'OpenGL.

Enfin, les "Shadow map" ont été problématiques sur notre projet car nous n'avions pas envisagé l'utilisation des "framebuffer" dès le début avec notre classe gérant le système de caméra. Cela nous a demandé beaucoup de reformatage pour pouvoir l'implémenter correctement.

Améliorations possibles

Plusieurs perspectives d'améliorations sont possibles pour rendre ce projet plus complet, stable et efficace telles que :

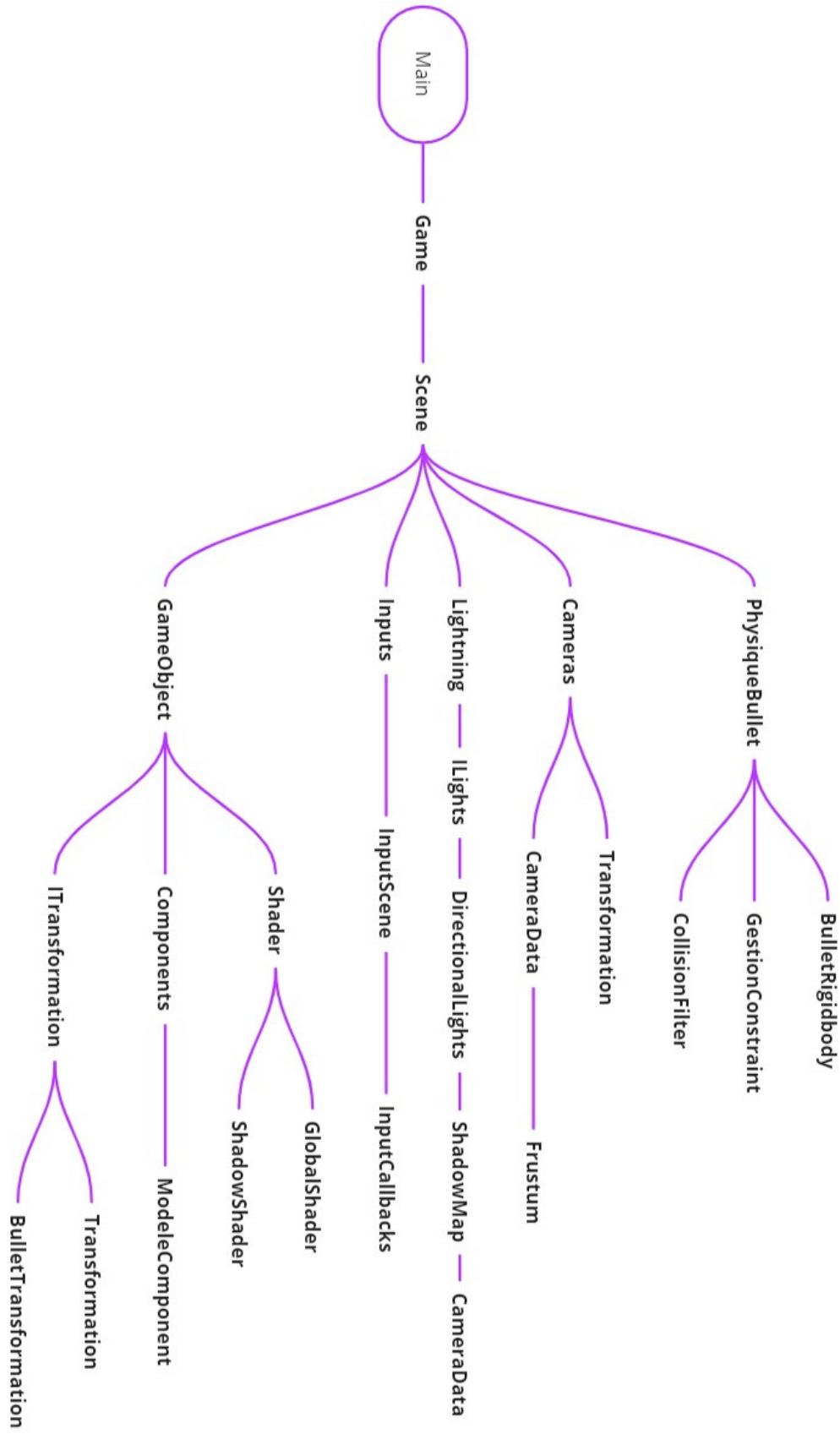
- L'ajout de plusieurs autres fonction pour gérer l'affichage des menus cliquables à la souris.
- L'implémentation d'une structure de données spatiales ("Quadtree", "Octree", "kd-tree", etc) pour simplifier les calculs d'affichage.
- L'amélioration de notre jeu basique.
- L'ajout d'un système de communication réseau.

Annexes

```
class SceneLight : public Scene, ILightCallback
{
private:
    // Init
    GlobalShader *globalShader = NULL;
    Lightning *lightScene = NULL;
    DebugShader *debugShader = NULL;
    InputLight *inputLight;
    Text2D *text2D;
    PhysiqueBullet *bullet;
    Texture *texBall;

    // Cooldowns
    int fps = 0;
    float cooldownFPS = 0.1f;
    bool wait1Frame = true;

public:
    SceneLight(IGlobalGameCallback *globalGameCallback) : Scene(globalGameCallback)
    {
        this->inputLight = new InputLight(globalGameCallback, this, this);
    }


    virtual void Load()
    {
        // Init Load()
        Scene::Load();
        glEnable(GL_CULL_FACE);
        glCullFace(GL_BACK);

        // Create LightScene
        lightScene = new Lightning();

        // Texte2D
        Text2DShader *textShader = new Text2DShader("Shaders/text2d_vertex.glsl",
        ↪  "Shaders/text2d_fragment.glsl", glm::ortho(0.0f, 1.0f * screen_width, 0.0f,
        ↪  1.0f * screen_height));
        Texture *atlasText = new Texture("Textures/Font/Atlas_Monofonto.jpg");
        text2D = new Text2D(textShader, atlasText, 128, 256);

        // Set Camera
        Camera *c = new Camera(vec3(0, 0, -15), 90, 0);
        this->cameras.push_back(c);

        // Set global Shader
        globalShader = new GlobalShader("Shaders/vertex_shader.glsl",
        ↪  "Shaders/fragment_shader.glsl");
        this->activeCamera = 0;
```

```
51          // Création de la physique
52          bullet = new PhysiqueBullet();
53
54          // Debug de la physique
55          debugShader = new DebugShader("Shaders/Debug/debug_vertex.glsl",
            ↪  "Shaders/Debug/debug_fragment.glsl");
56          DebugDrawer *debug = new DebugDrawer(debugShader);
57          debug->setDebugMode(btIDebugDraw::DBG_DrawWireframe +
            ↪  btIDebugDraw::DBG_DrawContactPoints);
58
59          // Initialisation de la physique
60          bullet->init(debug);
61
62          // Place une lumière
63          this->lightScene->addLight(new DirectionnalLight(glm::vec3(-8, 0, 0),
            ↪  glm::vec3(-1, 0, 0)));
64
65          // Création d'un composant balle avec texture
66          texBall = new Texture("Textures/SystemeSolaire/earth_daymap.jpg");
67          ModeleComponent *ballComponent = new ModeleComponent(globalShader);
68          ballComponent->addTexture(texBall, false);
69          PrimitiveMesh::generate_uv_sphere(ballComponent, 16, 16, 1.0f);
70
71          // Création d'un modèle LOD d'une balle
72          ModeleLOD *ball = new ModeleLOD("Ball", ballComponent, NULL, NULL, NULL,
            ↪  this->scene);
73          ball->getTransform()->setTranslate(glm::vec3(0, -1, 0));
74          ModeleLOD *tmpB = ball->duplicate();
75          tmpB->getTransform()->setTranslate(glm::vec3(0, 1, 0));
76          [...]
77
78          //Ajoute une balle a la Physique
79          bullet->addRigidbodyToPhysique(ball->getRigidbody());
80          // Création des plans pour l'affichage de l'ombre
81          ModeleLOD *plane = new ModeleLOD("Plane", globalShader, ModeleComponent::OBJ,
            ↪  "Model/wallplane.obj", NULL, this->scene);
82          [...]
83      }
84
85      // Fonction d'affichage
86      virtual void Draw(float deltaTime)
87      {
88          // Création des entrées utilisateur
89          inputLight->processInput(deltaTime);
90
91          // Affichage
92          Scene::Draw(deltaTime);
93
94          if (isActive())
95          {
96              this->cooldownFPS -= deltaTime;
97
98              // Calcul de la prochaine étape pour la physique
99              if (!global_pause)
```

```
100             {
101
102                 if (!wait1Frame)
103                 {
104                     if (debugShader != NULL)
105                     {
106                         debugShader->drawView(this->cameras[this->activeCamera]);
107                     }
108                     bullet->loop(deltaTime);
109                 }
110                 else
111                 {
112                     wait1Frame = false;
113                 }
114             }
115
116             // Calcul des shadowmap
117             this->lightScene->compute(this->scene);
118
119             // Bloque l'update de la camera si pause
120             if (!global_pause)
121             {
122                 if (this->activeCamera >= 0 && this->activeCamera <
                    ↪ this->cameras.size())
123                 {
124                     this->cameras[activeCamera]->checkUpdate();
125                 }
126             }
127
128             // Affichage de la scène à partir de la camera active
129             if (this->activeCamera >= 0 && this->activeCamera < this->cameras.size())
130             {
131                 globalShader->drawView(this->cameras[this->activeCamera]);
132                 this->scene->compute(this->cameras[this->activeCamera]->getData(),
                    ↪ this->lightScene, true);
133             }
134
135             // compute FPS
136             if (this->cooldownFPS <= 0)
137             {
138                 fps = (int)(1.0f / deltaTime);
139                 this->cooldownFPS = 1.0f;
140             }
141
142             // Affichage text2D
143             this->text2D->DrawText(std::to_string(fps), -1, 1, 0.9f);
144         }
145     }
```



Architecture du projet

Bibliographie

[1] Learn OpenGL, site officiel, 2022
https://learnopengl.com/

[2] Game Physics Cookbook, Gabor Szauer, 2017
https://www.onlineprogrammingbooks.com/game-physics-cookbook/

[3] Making maps with noise functions, Red Blob Games, 2022
https://www.redblobgames.com/maps/terrain-from-noise/

[4] Bullet Documentation, 2018
https://pybullet.org/Bullet/BulletFull/

[5] Bullet User Manual, 2015
https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf

[6] opengl-tutorial, 2017
http://www.opengl-tutorial.org/fr/