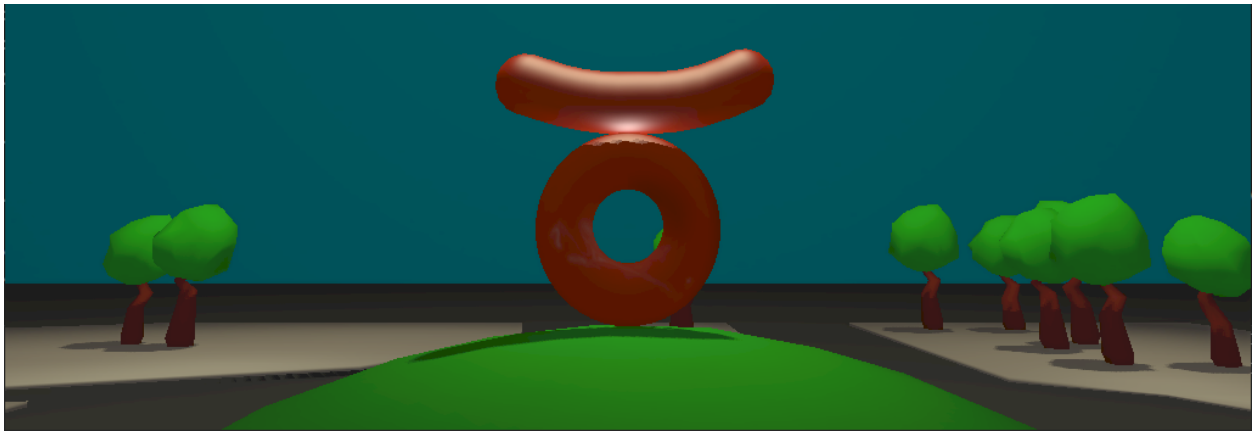
UNIVERSITÉ DE MONTPELLIER

RAPPORT DE MOTEUR DE JEU - HAI819I



Crousman

Réalisé par
Ange CLEMENT
Erwan REINDERS

Sous la direction de
Noura FARAJ

Dépot Github
GitHub - ange-clement/Crous_man: Crous_man game : the movie : the game

Année universitaire 2021-2022

Sommaire

Présentation du projet

Dans le cadre de notre formation en master IMAGINE, nous avons eu l'occasion de pouvoir développer un moteur de jeu dans le langage de notre choix, et orienté dans le but de pouvoir concevoir le jeu de notre choix. Nous avons décidé de partir sur un moteur codé en C++, reprenant des librairies que nous avons pu manipuler lors de précédents travaux pratiques, tel que la librairie GLM (structure de données vectorielles et transformations). Pour l'aspect graphique, nous implémentons nos rendus via la librairie OpenGL.

Le jeu en lui-même est un jeu inspiré des jeux de Kaiju. Ce sont des jeux dans lesquels le joueur contrôle un monstre qui doit tout détruire autour de lui, un peu à la manière de Godzilla.

Au travers de ce rapport sera donc détaillé les différentes fonctionnalités implémentées pour mener à bien ce projet. On abordera dans un premier temps l'architecture globale de la solution, puis les éléments présents au sein de ce moteur de jeu, et on indiquera enfin les possibilités d'améliorations d'un tel projet.

Architecture du projet

Pour structurer le moteur de jeu, nous avons opté pour la mise en place d'une architecture ECS (Entité-Composant-Système). Cette architecture a pour particularité d'orienter la programmation et le comportement des éléments de jeu sur les données. En effet, un comportement ne sera réalisé sur une entité que si elle possède les données spécifiques à ce dernier. C'est le cas par exemple de la gestion des collisions, où l'entité a besoin d'un collider pour que le système de détection des collisions la prenne en compte dans son traitement.

Représentation des Entités

Les entités sont stockées dans une liste d'entités. Cette liste appartient à "EntityManager". Une entité est composée d'un identifiant unique, d'une "Bitmap" et de données utiles pour construire un graphe de scène. En effet, une entité est aussi un nœud de notre graphe de scène. Elle possède donc également un objet modélisant sa transformation locale, sa transformation globale, son parent (qui est aussi une entité) et ses enfants (entités également).

La "Bitmap" sert à pouvoir renseigner les différents composants associés à une entité de la scène. Plus tard, nous allons pouvoir, pour chaque entité de la scène rendue, connaître ses composants en comparant cette "Bitmap" à l'identifiant unique du composant qui nous intéresse.

```
#ifndef SYSTEM_IDS_HPP
#define SYSTEM_IDS_HPP

enum SystemIDs {
    MeshID = 0,
    RendererID,
    PointLightID,
    CameraID,
    SpinID,
    FollowObjectID,
    DestructibleID,
    DeleteAfterTimeID,
    ColliderID,
    RigidBodyID,
    FlyingControllerID,
    SimplePlayerControllerID,
    CrousManControllerID,
    NUMBER
};
#endif
```

Identifiants des composants possibles pour une entité

On conserve les identifiants des composants dans une énumération, en faisant commencer le premier élément à 0, afin que les autres suivent une valeur croissante de un en un (spécificité C++). Pour la manipulation ensuite des composants au sein d'une entité, cela devient une simple comparaison de bits, efficace pour le traitement.

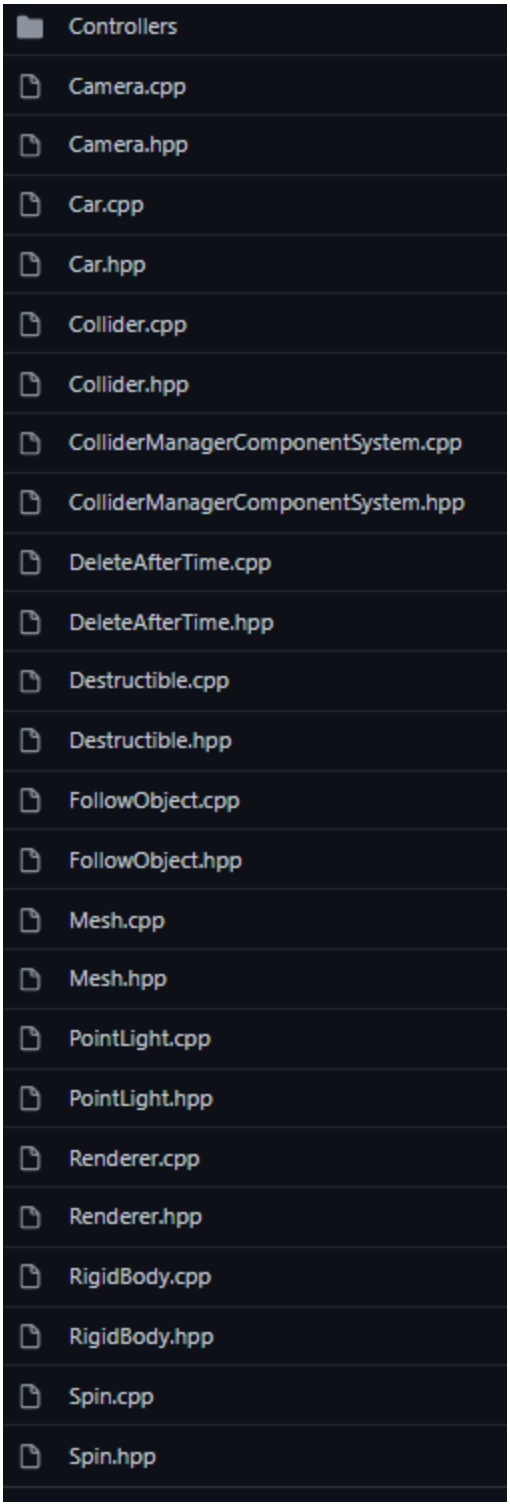
```
[...]

void Bitmap::addId(SystemIDs other) {
    this->bitmap |= 1 << other;
}
void Bitmap::removeId(SystemIDs other) {
    this->bitmap ^= 1 << other;
}
void Bitmap::addBitmap(const Bitmap* other) {
    this->bitmap |= other->bitmap;
}
void Bitmap::removeBitmap(const Bitmap* other) {
    this->bitmap ^= other->bitmap;
}

Bitmap* Bitmap::combine(const Bitmap* other) {
    return new Bitmap(this->bitmap & other->bitmap);
}


bool Bitmap::equals(const Bitmap* other) {
    return this->bitmap == other->bitmap;
}
```

Exemple de code du fichier Crous_man/ECS/Bitmap.cpp

Les transformations sont des instances d'une classe modélisant les transformations 3D triviales applicables à un élément de notre scène, composées d'une "Rotation" (sous structure de données pour les rotations, composée d'une matrice 3x3), d'un vecteur 3D modélisant sa translation et d'un vecteur 3D modélisant la possible mise à l'échelle inégale d'un objet dans l'espace.

Représentation des Composants

Les composants sont stockés dans les fichiers du système qui l'utilise. Pour le composant de maillage par exemple ("Mesh" dans "Crous_man/compoenents/Mesh.hpp"), on définit un système associé qui le manipule. Ce composant peut avoir des comportements, comme le fait de pouvoir calculer ses normales pour un maillage. Un composant sera principalement utile dans notre moteur de jeu pour garder, dans une structure de données précise, un certain nombre d'informations. Ces dernières viendront être récupérées ensuite par le bon système pour qu'il puisse opérer le bon traitement sur l'entité possédant ce composant.



Liste des différents composants du dossier Crous_man/Components/

Représentation des Systèmes

Les systèmes sont des classes de notre solution. Ils sont définis, pour chacun des composants possibles associés à une entité de jeu, en renseignant un comportement précis. En effet, dans la structure ECS, un comportement n'est appliqué à une entité que si celle-ci possède les bonnes informations, soit les bons composants. Cela va s'opérer pour chacun des systèmes, qui conservent un sous ensemble d'identifiants des entités de jeu possédant son composant associé. Cela permet ensuite au système de ne boucler que sur ses composants pour opérer son traitement. Cette liste d'identifiants est transmise au système à la création de l'entité, par la classe "EntityBuilder", dont nous en détaillerons le fonctionnement plus tard.

Un système hérite nécessairement de la super classe mère de tous les systèmes, la classe "ComponentSystem" :

```
class ComponentSystem {
public:
    std::vector<unsigned short> entityIDs;
    Bitmap* requiredComponentsBitmap;
public:
    ComponentSystem();

    ~ComponentSystem();

    virtual void updateAll();
    virtual void updateCollisionAll();
    virtual void updateOnCollideAll();
    virtual void updatePhysicsAll();
    virtual void updateAfterPhysicsAll();
    virtual void renderAll();
    virtual void updateAfterRenderAll();

    void initializeAll();

    void addEntity(unsigned short entityID);
    void removeEntity(unsigned short entityID);
    bool containsEntity(unsigned short entityID);

    unsigned short getComponentId(unsigned short entityID);

    virtual void update(unsigned short i, unsigned short entityID);
    virtual void updateCollision(unsigned short i, unsigned short entityID);
    virtual void updateOnCollide(unsigned short i, unsigned short entityID, const
std::vector<ColliderResult*> & collisionResults);
    virtual void updatePhysics(unsigned short i, unsigned short entityID);
    virtual void updateAfterPhysics(unsigned short i, unsigned short entityID);
    virtual void render(unsigned short i, unsigned short entityID);
    virtual void updateAfterRender(unsigned short i, unsigned short entityID);

    virtual void initialize(unsigned short i, unsigned short entityID) = 0;
    virtual void addEntityComponent() = 0;
};
```

Fichier header Crous_man/ECS/ComponentSystem.hpp

Une classe système peut donc redéfinir un des comportements suivants, afin d'opérer un traitement sur les bonnes entités à un moment particulier de la boucle de rendu. Un système

modélisant un comportement spécifique à une entité pour un jeu de données précis, il est admis de le définir qu'une seule fois dans l'application, puis de rappeler son unique instance au besoin, afin qu'elle mette à jour ses entités associées. Pour cela, on passe par la création d'une unique instance dans la classe centralisant le cœur de la gestion de notre moteur, la classe "EntityManager".

Un système peut nécessiter la présence de plus de données que celles qu'il a à gérer dans son composant associé, comme c'est le cas de "Rigidbody", qui oblige ses entitées associées à également posséder un composant "Collider" afin d'opérer la résolution de collision ensuite (Bitmap* requiredComponentsBitmap).

De plus, on remarque qu'un système ne possède pas directement la totalité des composants sur lesquels il doit opérer, ni les entités qui ont seulement une "Bitmap", car cela limiterait les performances de notre solution, lié à une mauvaise complexité en cache.

Au moment de générer une nouvelle entité dans notre système, on va la construire à partir d'une classe particulière, issue du patron de conception "constructeur", la classe "EntityBuildeur".

```
EntityBuilder::EntityBuilder(std::initializer_list<SystemIDs> systems) {
        this->buildEntity = new Entity(systems);
        EntityManager::instance->addEntity(this->buildEntity);

        this->mesh = NULL;

        this->rendererSystem = NULL;
        this->renderer = NULL;
        this->rendererID = (unsigned short)-1;

        this->destructible = NULL;

        this->pointLight = NULL;

        this->rigidBody = NULL;

        this->followObject = NULL;

        this->crousManController = NULL;
}
```

Constructeur d'un EntityBuilder dans le fichier Crous_man/ECS/Entitybuilder.cpp

On construit une entité à partir d'une liste de "SystemID", renseignant ses composants. On passe également par une classe "EntityManager", qui va centraliser les différentes informations liées à nos entités de jeu. En effet, pour une meilleure complexité en cache, il est plus intéressant de conserver tous les composants au même endroit, soit sous forme de listes dans "EntityManager".

```
EntityManager* EntityManager::instance = NULL;

EntityManager::EntityManager() {
    if (EntityManager::instance == NULL) {
        EntityManager::instance = this;
        Entity* root = new Entity();
        root->id = 0;
        entities.push_back(root);
        initUtil();
        initShaders();
        initSystems();
    } else {
        std::cerr << "Error : cannot instanciate two EntityManager" << std::endl;
    }
}

EntityManager::~EntityManager() {
    entities.clear();
    spinComponents.clear();
    EntityManager::instance = NULL;

    delete SoundManager::instance;
}

void EntityManager::initUtil() {
    new BasicShapeRender();
    new InputManager();
    new SoundManager();
}

void EntityManager::initShaders() {
    new BasicGShader();
    new TextureGShader();
    new DepthMeshEShader();
    new ShadowQuadEShader();
    new BlinnPhongShadowSSAOLShader();
    new BlinnPhongShadowLShader();
    new BlinnPhongLShader();
    new SingleTextureQuadShader();
    new SphereColliderShader();
    new BoxColliderShader();
}

void EntityManager::initSystems() {

    if (SystemIDs::NUMBER >= 64) {
        std::cout << "WARNING TOO MANY COMPONENTS" << std::endl;
    }
    systems.resize(SystemIDs::NUMBER);

    systems[SystemIDs::MeshID] =                    new MeshSystem();
    systems[SystemIDs::RendererID] =                new RendererSystem();
    systems[SystemIDs::PointLightID] =              new PointLightSystem();
    systems[SystemIDs::ColliderID] =                new ColliderSystem();
    systems[SystemIDs::RigidBodyID] =               new RigidBodySystem();
    systems[SystemIDs::CameraID] =                  new CameraSystem();
    systems[SystemIDs::SpinID] =                    new SpinSystem();
    systems[SystemIDs::FollowObjectID] =            new FollowObjectSystem();
    systems[SystemIDs::DestructibleID] =            new DestructibleSystem();
```

```
    systems[SystemIDs::DeleteAfterTimeID] =          new DeleteAfterTimeSystem();
    systems[SystemIDs::FlyingControllerID] =         new FlyingControllerSystem();
    systems[SystemIDs::SimplePlayerControllerID] =   new SimpleMovementPlayerSystem();
    systems[SystemIDs::CrousManControllerID] =       new CrousManControllerSystem();
}
[...]
```

Début de code de la classe EntityManager du fichier Crous_man/ECS/EntityManager.cpp

On peut remarquer dans cette classe l'instanciation des différents systèmes, stockés ensuite dans une liste précise, et renseignés par leur identifiant unique. Comme un système est par nature lié à son composant de traitement, on renseigne un seul identifiant pour manipuler les deux au sein de notre système de jeu. En effet, on l'utilise pour indiquer les composants d'une entité et construire sa Bitmap, mais également dans ce cas, pour générer à un emplacement unique sa seule instance dans la liste de "EntityManager".

On passe par un patron de conception "singleton" dans cette classe, car on veut n'avoir qu'un seul "EntityManager" dans toute notre application. Par la suite, il est possible d'appeler sa seule instance statique depuis n'importe quel endroit du code, ce qui facilite sa réutilisation et permet de lui faire sauvegarder plus d'informations, comme les programmes de shaders écrits en GLSL. Ces programmes (dans le dossier "Crous_man/Shaders") sont construits qu'une seule fois, car comme pour les systèmes, ils renseignent un comportement de rendu. Nous détaillons le traitement spécifique des ces programmes dans la partie suivante, sur les aspects mis en avant.

Ainsi, le code pour construire une entité devient le suivant :

```
   Entity* saucisse = (new EntityBuilder({ SystemIDs::MeshID, SystemIDs::RendererID }))
        ->setTranslation(glm::vec3(0.0f, 7.0f, 0.0f))
        ->setMeshAsFilePLY("../ressources/Models/saucisseCentre.ply")
        ->updateRenderer()
        ->setRendererDiffuse("../ressources/Textures/saucisseColor.ppm")
        ->build();

    Entity* doughnutSaucisse = (new EntityBuilder({ SystemIDs::MeshID, SystemIDs::RendererID,
SystemIDs::ColliderID, SystemIDs::RigidBodyID, SystemIDs::CrousManControllerID }))
        ->setTranslation(glm::vec3(0.0f, 20.0f, 0.0f))
        ->setCrousManControllerRotatingCenterForCamera(rotatingCenterForCamera)
        ->setCrousManControllerCameraTarget(wantedCameraPosition)
        ->setCrousManControllerCameraEntity(cameraEntity)
        ->setCrousManControllerSaucisseEntity(saucisse)
        ->setCrousManControllerLaserEntity(laser)
        ->setMeshAsFilePLY("../ressources/Models/dougnut.ply")
        ->updateRenderer()
        ->setRendererDiffuse("../ressources/Textures/dougnutColor.ppm")
        ->setRigidBodyMass(1.0f)
        ->fitOBBColliderToMesh()
        ->setRenderingCollider()
        ->build();

    Entity* crousLight = (new EntityBuilder({ SystemIDs::PointLightID }))
        ->setChildOf(saucisse)
        ->setTranslation(glm::vec3(0.0, -4.0, 0.0))
```

```
        ->setLightColor(glm::vec3(.958, .985, .938))
        ->setLightLinear(0.2)
        ->setLightQuadratic(0.04)
        ->build();

    Entity* croustopLight = (new EntityBuilder({ SystemIDs::PointLightID }))
        ->setChildOf(saucisse)
        ->setTranslation(glm::vec3(0.0, 100.0, 0.0))
        ->setLightLinear(0.01)
        ->setLightQuadratic(0.0001)
        ->setLightColor(glm::vec3(.984, .948, .628))
        ->build();
```

Exemple de construction du personnage principal de "Crousman", dans le fichier Crous_man/Scenes/ECS_test.cpp

Le personnage principal possède deux lumières en haut et devant lui (crousLight et croustopLight), ainsi qu'un maillage au dessus de lui qui le suit, une saucisse. L'entité doughnutSaucisse possède donc les composants suivants :

- SystemIDs::MeshID : pour renseigner sa forme (maillage 3D).
- SystemIDs::RendererID : pour indiquer qu'il est capable de se rendre.
- SystemIDs::ColliderID : pour indiquer qu'il doit être pris en compte dans le calcul des collisions par le ColliderSystem.
- SystemIDs::RigidBodyID : pour indiquer qu'il doit également être pris en compte par le système de résolution des collisions.
- SystemIDs::CrousManControllerID : pour indiquer ses déplacements dans la scène.

Enfin, la boucle de rendu dans notre moteur de jeu va donc simplement se charger de construire la bonne scène (appels aux fonctions de "Crous_man/scenes/ECS_test.cpp"), de construire une unique instance de "EntityManager", puis d'appeler les bonnes fonctionnalitées de ce dernier dans le boucle principale de rendu, en plus des fonctionnalités de gestion des entrées claviers ou sorties audios ("InputManager" et "SoundManager" du dossier "Crous_man/").

```
    [..INIT OPENGL..]
    new EntityManager();

//createSceneCollider();
createSceneGame();
//createSceneECS();

EntityManager::instance->updateTransforms();
EntityManager::instance->initializeAllSystems();
EntityManager::instance->updateTransforms();


SoundManager::instance->play("../ressources/Sounds/start.wav");

InputManager::instance->initialize(window);
```

```
    do{
        InputManager::instance->update(window);
        SoundManager::instance->update();

        EntityManager::instance->update();
        EntityManager::instance->updateTransforms();
        EntityManager::instance->updateCollision();
        EntityManager::instance->updateOnCollide();
        EntityManager::instance->updatePhysics();
        EntityManager::instance->updateAfterPhysics();
        EntityManager::instance->render();
        EntityManager::instance->updateAfterRender();

        // Swap buffers
        glfwSwapBuffers(window);
        glfwPollEvents();
        //glfwSetWindowShouldClose(window, true);
    }
    while( glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS &&
           glfwWindowShouldClose(window) == 0 );

    delete EntityManager::instance;

    // Close OpenGL window and terminate GLFW
    glfwTerminate();

    return 0;
}
```

Boucle principale de rendu dans le fichier Crous_man/Game.cpp

Les différentes fonctions appelées peuvent être résumées ainsi :

- InputManager::instance->update(window) : mise à jour des connaissances des entrées claviers utilisateur
- SoundManager::instance->update() : mise à jour et gestion des évènements audios
- EntityManager::instance->update() : appel à tous les updates de chacun des systèmes, dépend du traitement d'un système.
- EntityManager::instance->updateTransforms() : mise à jour des positions monde des entités.
- EntityManager::instance->updateCollision() : détection des collisions dans notre scène.
- EntityManager::instance->updateOnCollide() : résolution des collisions dans notre scène (dépend des résultats de la phase précédente).
- EntityManager::instance->updatePhysics() : application des forces sur un objet (forces de gravité et forces d'impulsions de la phase précédente).
- EntityManager::instance->updateAfterPhysics() : mise à jour après application de la physique.
- EntityManager::instance->render() : rendu de la scène grâce aux différents programmes shaders contenus dans EntityManager.
- EntityManager::instance->updateAfterRender() : mise à jour après le rendu de la scène.

Fonctionnalitées du projet

Dans un premier temps, on a d'abord cherché à structurer correctement le code de notre solution, en mettant en place tout un système de classes ECS, ainsi qu'un système de gestion des différents composants qui en hérite. C'est dans ces composants hérités que les différents comportements possibles pour une entité de jeu vont être définis. C'est également là que va être définie la structure de données associée au système qui la manipule.

Éléments de structuration

Les principaux éléments de structuration de notre moteur de jeu sont dans un premier temps le graphe de scène, qui est matérialisé par la classe Entity (dossier "Crous_man/ECS/"). La classe "EntityManager" possède donc une "Entity" racine de notre graphe de scène, d'indice 0.

Pour les différents systèmes, nous les modélisons par le biais de classes, dont les composants sont représentés par des structures de données (la plupart du temps), car simples à manipuler. Les informations qu'ils conservent dépendent essentiellement du traitement à opérer.

Pour les programmes de shaders, on passe par une hiérarchie particulière, expliquée dans la partie "Aspects mis en avant". Elle permet ensuite la création et la manipulation d'instances uniques de programmes de shaders. Le comportement pouvant varier d'un shader à l'autre, il est intéressant d'en renseigner un par typologie de traitement graphique à appliquer.

On utilise une unique instance de gestionnaire des entités et composants-systèmes de notre moteur. Elle centralise tous le traitement ECS et va nous permettre une meilleure manipulation de ces données, ainsi qu'une meilleure complexité spatiale (et donc d'augmenter les performances de jeu).

Exemples de Composants-Systèmes

En modélisant l'architecture ECS de notre projet, nous avons pu ensuite définir une variété de sous-systèmes, opérant chacun sur des parties du traitement final de la boucle de jeu.

Le composant le plus simple est "Spin" ("Crous_man/Components/Spin"). Une entité possédant ce composant se met à tourner.

```
struct Spin {
    float speed;
    float spinAmount;
};
```

Dans cette structure, speed est la vitesse de rotation, et spinAmount mesure la rotation appliquée.

Dans le système de “Spin”, on a :
Dans le constructeur, il faut spécifier la bitmap.

```
SpinSystem::SpinSystem() : ComponentSystem(){
    requiredComponentsBitmap = new Bitmap({SystemIDs::SpinID});
}
```

Dans initialise(), on récupère le composant “Spin” et on l’initialise.

```
void SpinSystem::initialize(unsigned short i, unsigned short entityID) {
    getSpin(i)->speed = .5f;
    getSpin(i)->spinAmount = 0.0f;
}

Spin* SpinSystem::getSpin(unsigned short i) {
    return &EntityManager::instance->spinComponents[i];
}
```

Dans update, on récupère le composant “Spin” afin de connaître la vitesse de rotation et la mesure de rotation. On met à jour cette mesure à partir de la vitesse de rotation et de la différence de temps depuis la dernière trame (deltaTime). Enfin, on modifie la rotation de l’entité.

```
void SpinSystem::update(unsigned short i, unsigned short entityID) {
    Spin* s = getSpin(i);
    s->spinAmount += s->speed * InputManager::instance->deltaTime;

EntityManager::instance->entities[entityID]->transform->rotation.setRotatio
n(s->spinAmount * 3.14159263, glm::vec3(0.0, 1.0, 0.0));
}
```

Afin de compléter le système, il faut implémenter la fonction addEntityComponent(), afin d’ajouter le nouveau composant au bon emplacement (dans “EntityManager”).

```
void SpinSystem::addEntityComponent() {
    EntityManager::instance->spinComponents.push_back(Spin());
}
```

Aspects mis en avant

Sur la base du code imposé pour le rendu de ce projet, nous avons eu la possibilité de venir pousser certains aspects composant un moteur de jeu, en rapport avec le type de jeu que nous voulions faire. Ainsi, nous avons poussé la physique de notre jeu afin d'opérer des collisions avec différents types de formes possibles (Sphère-AABB-OBB). Cela nous paraissait important, compte tenu du fait que l'on doit détruire l'environnement dans lequel on se trouve. Nous avons également implémenter un système de forces afin de pouvoir opérer des impulsions sur des objets au moment de la destruction d'une entité par exemple. Enfin, nous avons mis en place un système de lancer de rayons, afin d'opérer par exemple la simulation d'un rayon "destructeur" tiré depuis la caméra vers un élément de jeu ayant un composant pouvant détecter la collision.

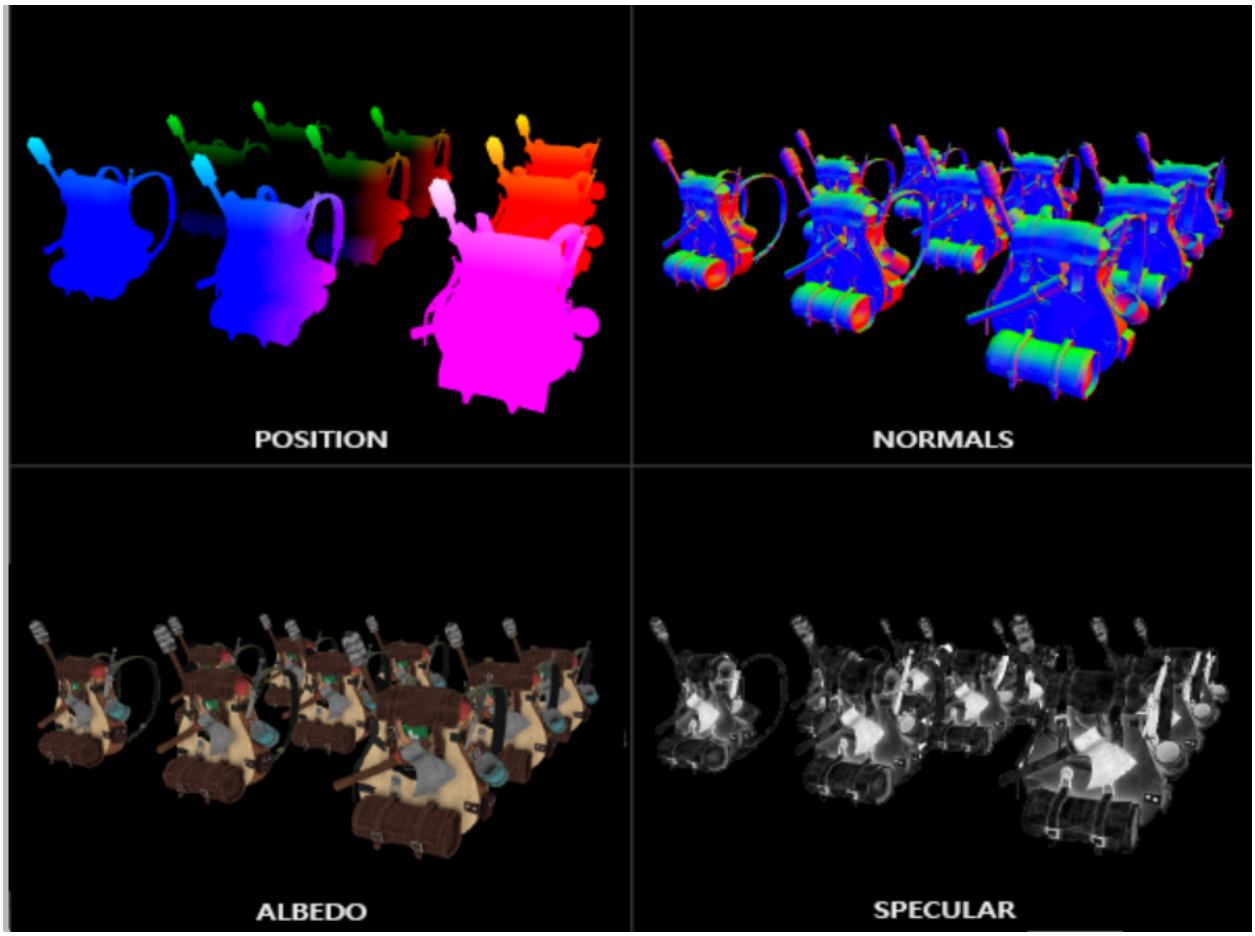
Le composant Collider et le système "ColliderSystem" vont venir définir dans notre moteur cette première partie de détection des collisions. Ils vont également, dépendant du type de collider utilisé, venir renseigner des informations utiles à la résolution de collision ensuite (point de contact et normale à la surface, ainsi que distance de pénétration). Le composant Rigidbody et le système "RigidBodySystem" vont venir appliquer le traitement de résolution de collisions. Ces deux paires de composants-systèmes sont disponibles dans le dossier "Crous_man/components".

Nous nous sommes, pour la détection et résolution de collisions, grandement inspirés de l'ouvrage "Game physics Cookbook" de Gabor Szauer. Nous avons également implémenté différentes interpolations de temps, d'après notre compréhension des cours disponibles pour cette matière.

Nous avons aussi poussé les aspects graphiques de notre moteur afin d'inclure les ombres portées par exemple. Cela ajoute du réalisme à une scène de milieu urbain ou en extérieur. Il y a également la possibilité sur notre moteur, d'opérer du rendu en différé des éléments de notre scène, ou non. C'est une fonctionnalité présente sur de nombreux moteurs de rendu modernes, et permet, par exemple, un gain de performances sur des scènes denses en éléments graphiques, car on ne va opérer le traitement du calcul des lumières qu'une seule fois, et pas pour chacun des maillages à rendre.

Nous avons également utilisé une librairie de gestion de sons pour notre moteur de jeu, la librairie "Irrklang". Cela nous permet de jouer des sons dans notre scène, en un point donné de celle-ci par exemple (son 3D).

Afin de rendre la scène, nous avons mis en place un système permettant d'organiser les shaders. Les shaders sont dans "Crous_man/Shaders". Dans un premier temps, on rend tous les objets avec un "GShader". Ce shader nous permet d'obtenir plusieurs images en espace écran : la position en coordonnée monde, la normale, la couleur et la valeur spéculaire.



Capture du résultat donné par le GShader (https://learnopengl.com/Advanced-Lighting/Deferred-Shading)

Cette phase nous permet d'optimiser la vitesse du rendu. En effet, le calcul de lumière, qui est effectué plus tard, possède un fort coût. On veut donc minimiser le nombre de calculs. Grâce à cette phase, ce calcul est fait seulement grâce aux données "visibles", c'est-à-dire dans l'espace écran. On n'effectue pas les calculs lourds en dehors de l'écran, donc on gagne du temps.

Ensuite, on entame la phase d'Effect Shader ("EShader"). Il y a plusieurs type de "EShader", les "MeshEShader", et les "QuadEShader" :

Un "MeshEShader" va faire un rendu de tous les objets qui l'intéresse et donc produit des textures.

Un "EShader" prend en entrée plusieurs textures et en produit d'autres.

On a donc une liste de "EShader" ainsi qu'une liste de textures. Lors de cette phase, on va donc mettre le résultat du "GShader" dans une liste de textures, et, pour chaque "EShader", on l'exécute soit à partir des textures de la liste, soit en rendant la scène, et on ajoute ses textures résultantes dans la liste. On a déjà implémenté plusieurs "EShader" : "BlurQuadEShader" permet de flouter une texture. "DepthMeshEShader" permet de donner la distance des fragments par rapport à un point. "ShadowQuadEShader" permet de donner une valeur d'ombrage aux fragments. "SSAOQuadEShader" permet de donner une valeur d'occlusion aux fragments.

Après, la phase de lumière est exécutée grâce à un “LShader”. Le “LShader” va calculer la couleur en fonction de l’éclairage. Il va donc utiliser la liste de texture précédemment générée et la liste des lumières. Nous avons implémenté un shader Blinn-Phong qui prend en compte l’ombrage et l’occlusion ambiante.

Enfin, on a la phase de Post Effect. Un “PEShader” va effectuer un dernier traitement à partir de la texture de lumière et appliquer le résultat sur l’écran. On fait donc une correction gamma avant de l’afficher (https://learnopengl.com/Advanced-Lighting/Gamma-Correction).

Pistes d'améliorations

Les pistes d'améliorations pour un tel projet sont très vastes. En effet, nous avons orienté le développement de notre projet autour de la réalisation d'un jeu. Si nous avions voulu faire un moteur de jeu plus générique, comme c'est le cas pour Unity par exemple, nous aurions de nombreux points de progrès possibles.

Dans un premier temps, il est possible d'implémenter une quantité plus importante d'effets de rendus, rendus possibles par l'ajout dans le code de classes "Shader". En effet, les programmes shaders étant des objets à part entière de notre solution, il est possible d'en générer des nouveaux en renseignant leurs comportements, par l'implémentation d'une série de fonctions présentes dans les classes mères de la hiérarchie de shaders. Un seule entité de ces programmes est générée pour toute la solution, ce qui évite de redéfinir plusieurs fois un même comportement de rendu pour deux entités similaires de notre scène.

Ensuite, sur le plan physique, il est possible de raffiner encore plus la détection des collisions dans un premiers temps, pour l'étendre à la détection de collisions sur le maillage entier (détection de collision sur des maillages convexes autrement que par SAT), et pas sur des formes de collision prédéfinies (Sphère, AABB, OBB). En ce qui concerne la détection de collision avec ce type de formes, il est possible également d'en raffiner le calcul, comme c'est le cas pour deux OBB. En effet, pour déterminer les points de contacts entre deux boîtes alignées sur les axes de l'objet, on regarde dans notre implémentation quand une arête de l'une des boîtes intersecte le plan de l'autre boite. Dans les faits, il existe une variété plus importante, et potentiellement plus significative pour le traitement à opérer derrière, de points de contacts possibles entre deux boîtes 3D, résultant de situations de contact différentes. L'une de ces situations est quand on a dans une même scène deux boîtes posées l'une sur l'autre, qui peut entraîner la détection de points de contacts aux extrémités des arêtes de l'une des boîtes. On aurait dans ce cas plus tendance à vouloir prendre comme point de contact le point médian de la surface en contact par exemple (car plus pertinent pour l'application d'impulsions).

Pour l'application des différentes vitesses, il est possible d'utiliser des intégrations plus stables que celle d'Euler par exemple, afin de limiter les écarts possibles au cours du temps entre la réelle position physique d'un objet soumis à une force, et la position calculée numériquement. Cependant, ce gain de précision peut avoir un coût significatif sur les performances de la solution finale.

Dans ce projet, nous n'avons pas mis en place de réel moteur d'animation 3D. Bien que ce soit une structure importante d'un moteur de jeu classique, cela ne faisait pas partie des priorités de notre solution. Nous avons voulu nous concentrer sur des aspects plus liés à la physique et au rendu dans notre moteur. En implémentant les quaternions, il est possible par exemple de réaliser des animations par points clés et interpolation entre ceux-ci tout en évitant le problème de blocage de cadran (Gimbal lock) par exemple. Cela peut s'intégrer via l'ajout de structures de données stockant les différents points clés pour un modèle donné, et d'un système gérant les différentes interpolations possibles permettant de passer d'une position clés de l'animation du personnage, à une autre.

Enfin, il est également possible d'implémenter des structures d'accélérations pour le traitement d'une scène par exemple, avec la possibilité de générer un QuadTree, voir un Octree pour la détection de collisions, et ne venir regarder les collisions entre deux éléments que s'ils se trouvent dans une même cellule de cette structure de partitionnement.