Université Polytechnique Hauts-de-France
Institut National des Sciences Appliquées

Rapport de projet

Moteur de jeu pour un jeu de plateforme 2D

Ethan MARLOT - 8 Avril 2022

Table des matières

Chapitre 1

Introduction

Dans le cadre de ce projet tuteuré de cette année 2022, j'ai choisi de développer un jeu de type "Plateformer-Shooter", similaire aux jeux d'acrade *Metal Slug*. Afin de réaliser ce projet, j'ai décidé de ne pas utiliser un moteur de jeu existant mais d'en créer une implémentation simple.

En l'état, voici l'apparence du jeu :

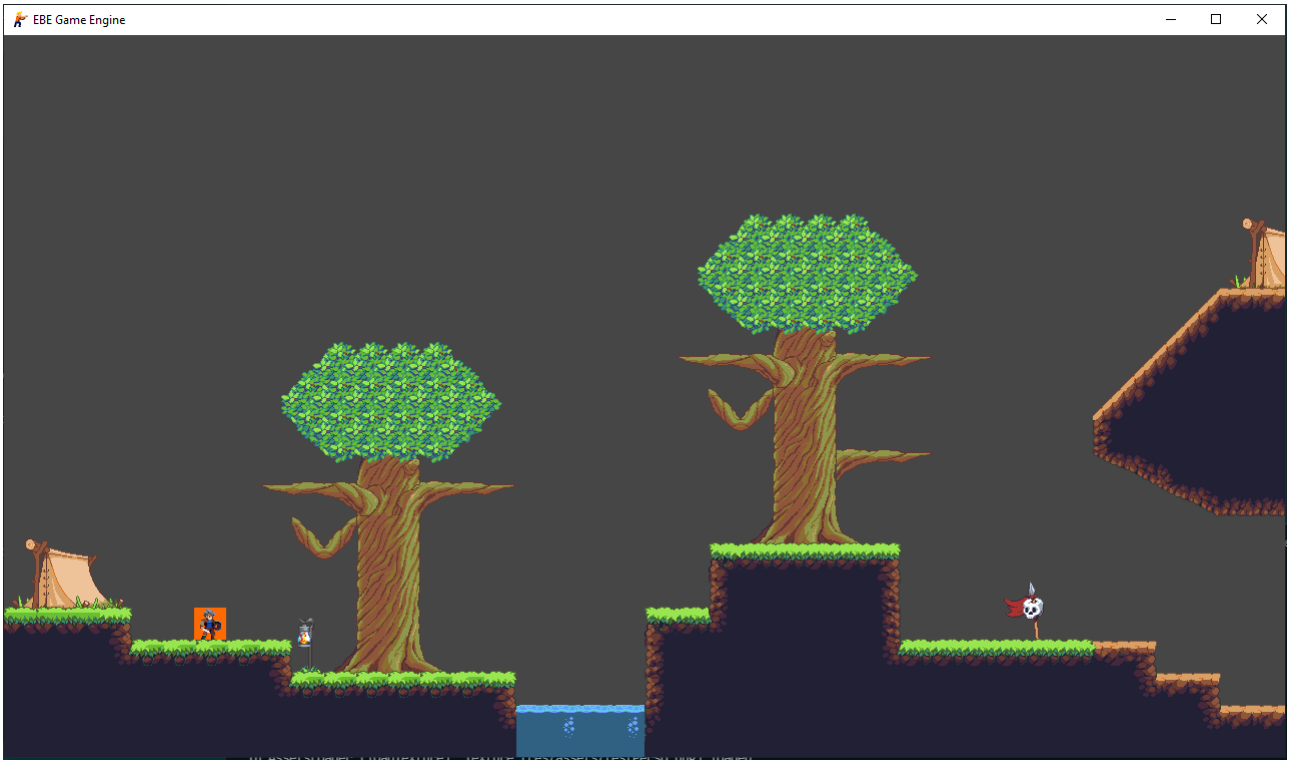


Figure 1.1 – L'etat du jeu en fin de projet

Nous avons une carte, ainsi qu'un personnage pouvant se déplacer librement sur cette dernière.

Chapitre 2

Analyse des besoins du projet

Tout d'abord, j'ai entrepris des recherches sur les moteurs de jeu. Lors de ces dernières, j'ai trouvé deux principaux modèles de moteur de jeu :

— Le modèle basé sur le paradigme de l'orienté objet ;
— Le modèle basé sur le paradigme de l'orienté donnée, ;

J'ai, dans un premier temps, pensé à choisir le paradigme orienté objet, étant donné que nous l'avons beaucoup travaillé lors de ces derniers semestres. Mais après réflexions, pour un moteur de jeu, l'orienté objet atteint ses limites. Prenons l'exemple simple suivant :

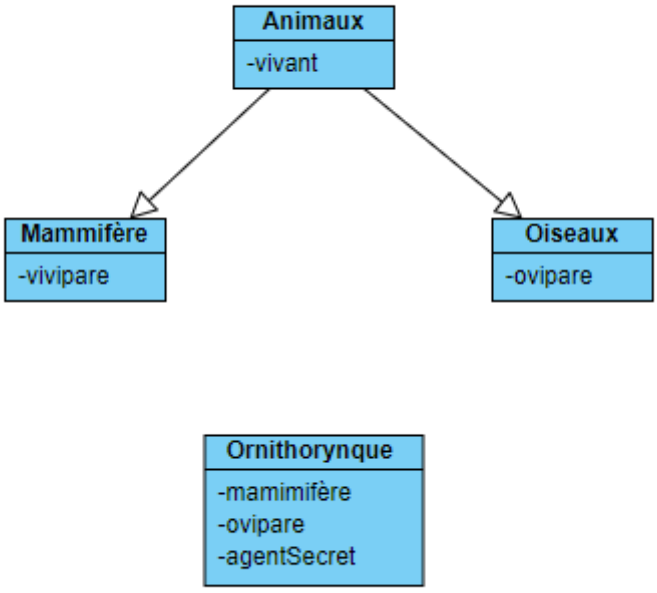


Figure 2.1 – Le patterne orienté objet

Dans cet exemple, nous avons une classe abstraite Animaux, dont héritent deux classes, Mammifères et Oiseaux. Les Mammifères sont vivipares tandis que les Oiseaux sont ovipares. Dans ce cas, quid de l'Ornithorynque, ce mammifère ovipare ? Il est vrai qu'en C++, il y a la possibilité d'héritages multiples, mais cela peut poser d'autres problèmes, comme dans l'exemple au dessus, si la classe Oiseaux a un attribut ailes, alors Ornithorynque ne devrait pas hériter d'Oiseaux.

C'est là où le paradigme de l'orienté donnée entre en jeu :

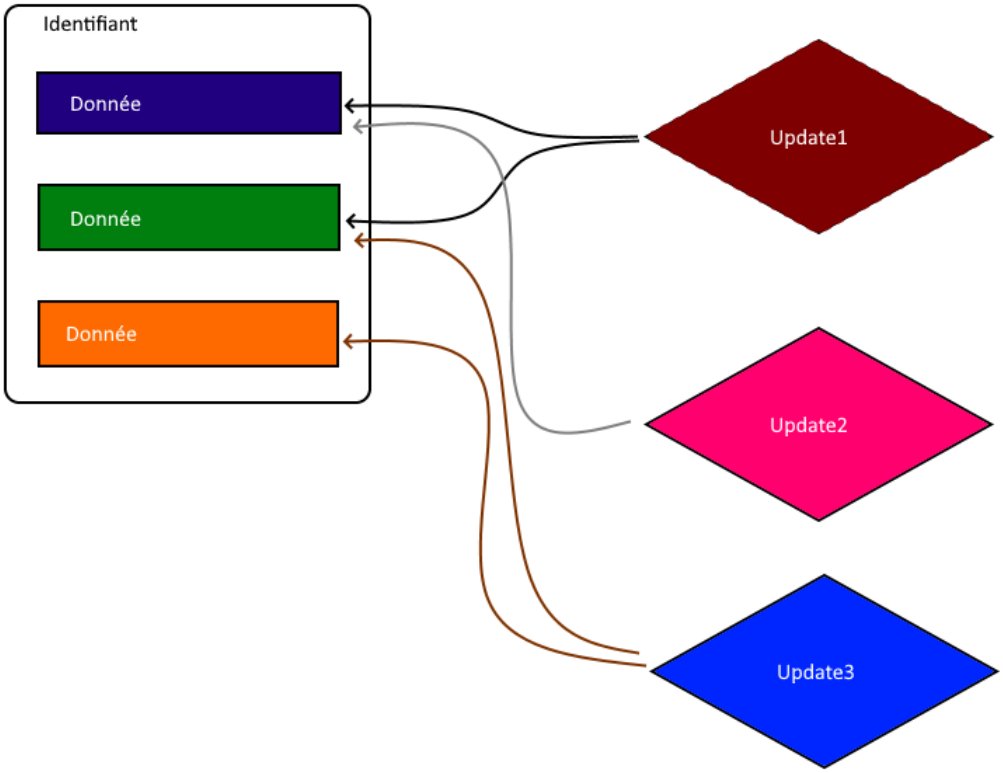


FIGURE 2.2 – Le patterne orienté donnée

Dans ce paradigme, nous associons des données à un identifiant, qui seront manipulé par des fonctions en les récupérant selon leur identifiant.

Dans le développement de jeux-vidéo, l'utilisation de ce paradigme mène à un patterne nommé ECS, pour Entity Component System. Les Entity (entités) sont les identifiants, les Components (composants) sont les données, et les Systems (systèmes) sont les fonctions.

Chapitre 3

Moyens utilisés pour la conception du jeu

3.1 Pour le développement du jeu

3.1.1 Langage C++

Pour la réalisation du projet, j'ai choisi d'utiliser le langage de programmation C++, car c'est un langage de bas niveau, permettant une gestion assez poussée de la gestion de la mémoire, ce qui est indispensable afin de rendre le jeu utilisable sur le maximum de machines, qu'importe leur puissance. J'ai utilisé la bibliothèque standard C++17, qui dispose de classes conteneurs très utiles, comme des tableaux associatifs ou des files, grâce à la STL (Standard Template Library).

3.1.2 Bibliothèque graphique SDL2

Quant à la partie graphique, j'ai choisi d'utiliser la bibliothèque SDL2 (Simple Directmedia Layer), étant donné que nous l'avons déjà manipulé au semestre 3 lors du module Développement d'Application, et que je l'utilise également dans certains de mes projets personnels. J'ai également utilisé la dépendance SDL_image, permettant l'utilisation d'images de différents formats, notamment PNG, là où la bibliothèque de base ne permet que l'utilisation du format BMP.
J'ai utilisé la version 2.0.16 de la bibliothèque, et 2.0.5 pour sa dépendance.

3.1.3 Bibliothèque d'analyse syntaxique TinyXML

Finalement, pour une raison que j'aborde dans de la partie suivante, j'ai décidé d'utiliser la bibliothèque TinyXML, qui est un parseur (analyseur syntaxique) du format XML. J'ai choisi cette dernière en particulier car elle est de loin la plus simple d'utilisation que j'ai trouvé.

3.2 Pour les ressources graphiques

3.2.1 Itch.io

N'étant pas un fin dessinateur, j'ai décidé d'utiliser des supports visuels libres de droits pré-existants. Pour cela, j'ai cherché sur itch.io, un site web utilisé par des créateurs indépendants pour distribuer leur travaux. Je me suis bien assuré que les ressources choisies étaient libres de droits avant de les utiliser, puis je les ai arrangé pour former un jeu de tuiles, notamment pour le logiciel suivant.

FIGURE 3.1 – Le jeu de tuiles formé

3.2.2 Logiciel Tiled Map Editor

Ce logiciel permet la conception de cartes à l'aide de tilesets (jeux de tuiles). Après conception de la carte, nous obtenons un fichier au format TMX (Tiled's Map Format), dérivé du langage XML, justifiant le besoin de la bibliothèque TinyXML, contenant les informations du jeu de tuiles ainsi que celles de chaque couches de la carte formée dans le logiciel. J'ai utilisé la dernière version de ce logiciel (1.8.0).

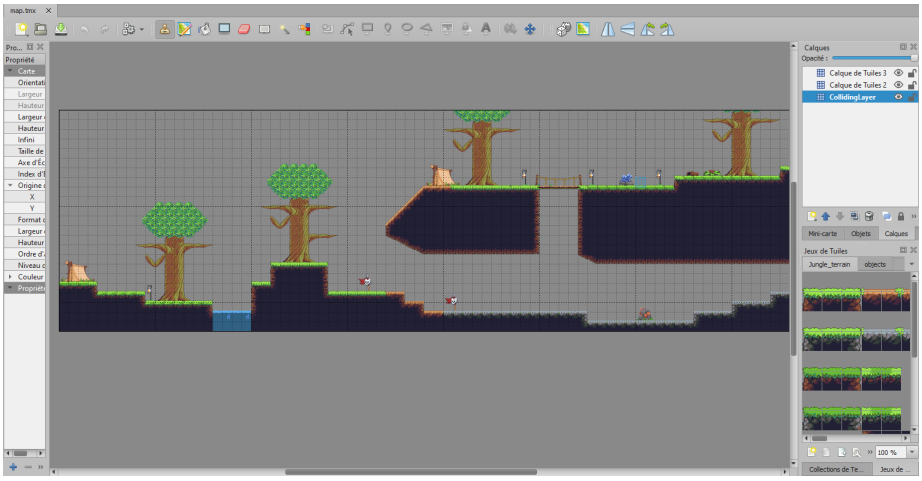


FIGURE 3.2 – La carte créée sur Tiled Map Editor

Chapitre 4

Analyse des solutions apportées

4.1 Évaluation du temps prévu et du temps réel utilisé pour concevoir le projet

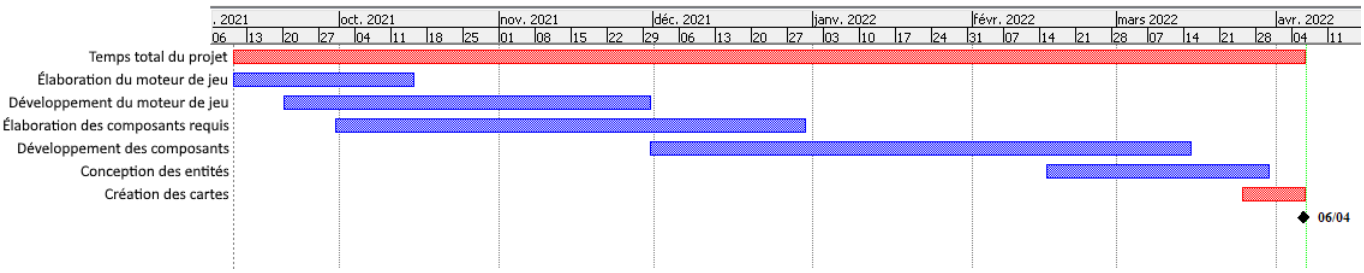


FIGURE 4.1 – Temps initialement prévu pour le projet

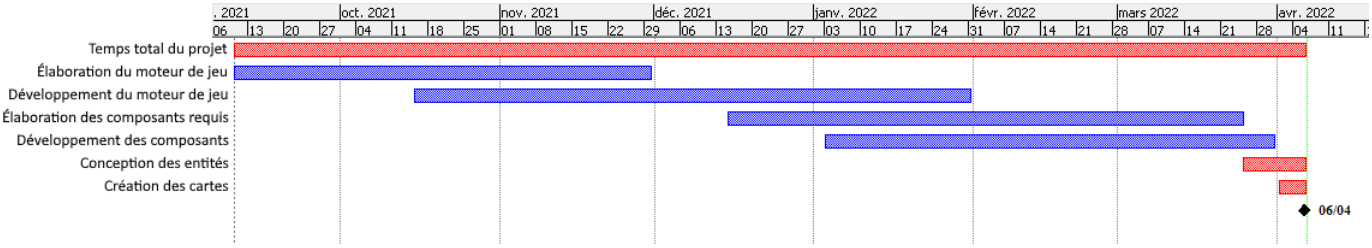


FIGURE 4.2 – Temps réellement passé pour le projet

Nous pouvons voir, grâce aux diagrammes de Gantt ci-dessus, que j'ai bien trop sous-estimé la conception du moteur de jeu, qui m'a pris cinq mois au lieu des trois que je prévoyais, ce qui m'a poussé à précipiter la conception des composants. De ce fait, je n'ai pas eu le temps de concevoir ni d'autres entités que le joueur, ni plus d'une carte.

4.2 Analyse et conception de l'ECS

Afin de concevoir l'Entity Component System, j'ai déterminé les besoins :
- Des entités, qui servent des identifiants, pouvant donc se résumer à des entiers ;
- Des composants, contenant des données, mais surtout l'entité à laquelle ils sont associés ;
- Des tableaux regroupant les composants par type ;
- Un tableau regroupant les tableaux de composants ;
- Les systèmes, mettant à jour les composants ;
- Des tableaux de booléen, pour savoir si un composant est associé à une entité ;
- Un tableau regroupant les différents systèmes ;
- Une classe englobant les différents tableaux afin de centraliser l'ECS.

Les entités n'étant, comme dit plus haut, que des identifiants, on peut limiter l'entité à un entier, d'où le choix de représenter l'entité comme dessous :

```cpp
using Entity = std::uint32_t;
```

Figure 4.3 – Définition du type Entity

Les composants doivent être séparé selon leur type, un composant A ne devant pas être dans le du conteneur du composant B. Pour cela, l'interface des composants utilise un template : le composant héritant de l'interface IComponent doit être du type implémenté, expliquant le choix de la définition de l'interface IComponent comme une classe générique. De plus, la surcharge de l'opérateur d'égalité permettra de savoir si un composant existe déjà pour une entité donnée ou non. Ce qui donne :

```cpp
/**
 * @brief Interface for the components
 *
 * @tparam T the component type
 */
template<typename T>
struct IComponent {
    IComponent() {e = -1;}
    IComponent(Entity _e) : e(_e) {}
    virtual ~IComponent() = default;

    Entity e;

    bool operator==(const IComponent<T>& c) {
        return e == c.e;
    }
};
```

Figure 4.4 – Définition du type IComponent

Les tableaux de composants sont définit dans une classe indépendante, mais héritent d'une interface qui oblige l'implémentation d'une fonction *entityDestroyed*, qui assure de supprimer les composants d'une entité détruite, pour éviter que, si l'entité est réutilisée, les composants liés à l'ancienne entité subsistent :

```
/**
 * @brief Interface for the Component Arrays
 *
 */
class IComponentArray {
    public:
        virtual void entityDestroyed(Entity e) = 0;
};
```

FIGURE 4.5 – Définition du type IComponentArray

Le système a besoin d'une méthode pour mettre à jour les composants, une méthode determinant si une entité peut être mise à jour par ce système, et un tableau de booléen qui détermine quels composants sont requis pour que le système mette une entité à jour :

```
/**
 * @brief Interface for the systems
 *
 */
struct ISystem {
    virtual void update(float dt) = 0;
    virtual bool isUpdatable(Signature entitySignature) = 0;
    Signature systemSignature;
};
```

FIGURE 4.6 – Définition du type ISystem

Le type *Signature* est un bitset, un tableau de bits, servant de booléen. J'ai également défini des fonctions attribuant des identifiants aux différents composants et systèmes, pour savoir quel index du bitset est associé à quel composant.

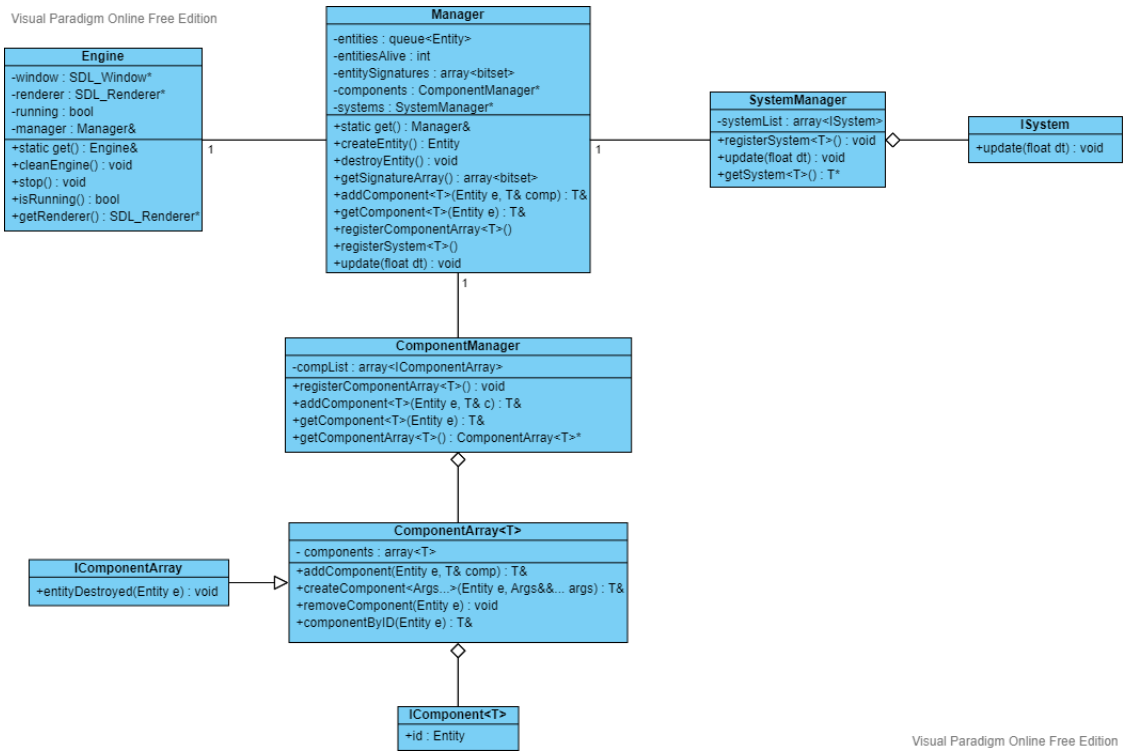


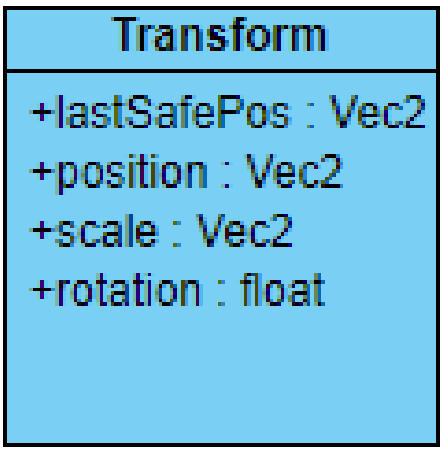
FIGURE 4.7 – Diagramme UML de l'Entity Component System

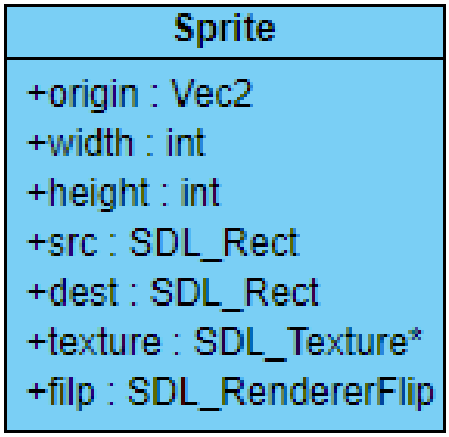
Ci-dessus le diagramme UML de mon Entity Component System. Les classes Manager et Engine sont des objets auto-instanciables, utilisant un patterne similaire au Singleton, mais sans pointeurs. Les templates de *IComponent* et *ComponentArray* sont utilisé avec les composants implémentant l'interface *IComponent*.

4.3 Analyse des composants

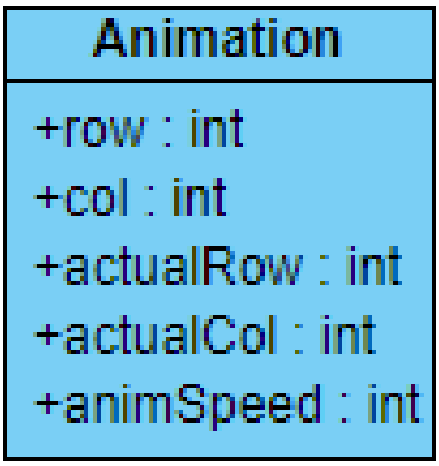
En préambule de cette section, je précise ce qu'est la structure Vec2, utilisée dans presque tous les composants. Il s'agit d'un vecteur, comportant un x et un y, définissant par exemple une position dans l'espace. Dans une fenêtre de la bibliothèque SDL2, un Vec2(0,0) correspond au coin haut-gauche.

Le composant Transform représente la position d'une entité. Si l'entité a un Sprite (dont on parlera plus loin), le composant Transform détient également l'échelle et la rotation du Sprite. Finalement, si l'entité a un Collider2 et un RigidBody (dont on parlera également plus loin), Transform contient également la dernière position sûre de l'entité, c'est-à-dire la dernière position où l'entité ne traverse pas le sol ou une autre entité.

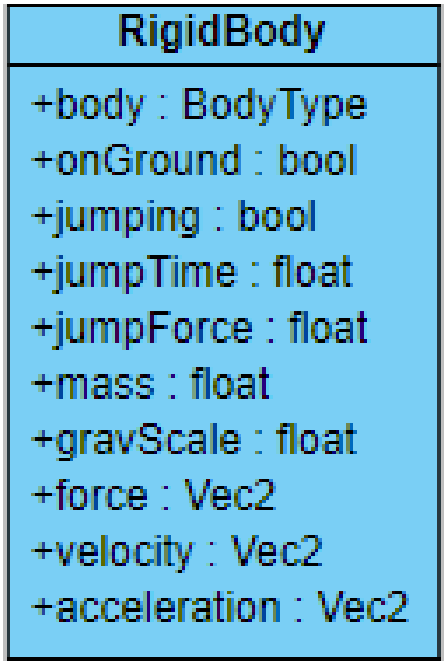




Le composant Sprite détient la texture de l'entité. Il contient également un vecteur représentant le centre du Sprite, la largeur et la hauteur de la texture, ainsi que l'éventuel retournement de la texture (si le personnage court à gauche, par exemple, il sera retourné vers la gauche). Finalement, il contient deux rectangles : le rectangle destination, qui représente la position effective à l'écran, et le rectangle source, qui représente la position du Sprite dans la texture (si le Sprite est animé, par exemple, la position x et y du Sprite changera lors des mises à jour).

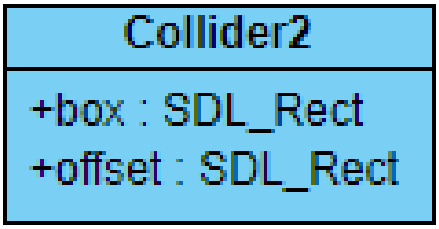


Le composant Animation détermine si une entité a un Sprite animé ou non. Ce composant détient le nombre de lignes et de colonnes du Sprite animé (la ligne 1 sera, pour les personnages, l'animation d'attente, la deux sera l'animation de course, et la trois aurait été l'animation de saut). Il contient aussi la position actuelle de la colonne et de la ligne, ainsi que la vitesse d'animation (le nombre de millisecondes entre chaque changement de colonnes).



Le composant RigidBody représente le corps physique de l'entité. Il contient plusieurs vecteurs, notamment la force appliquée à l'entité, son accélération et sa vélocité. Il y a également deux booléens, déterminant si le personnage est en train de sauter ou s'il est sur le sol, ainsi que 4 nombres, représentant la masse du corps, l'échelle de gravité appliquée à ce dernier (la gravité est ici un vecteur constant ayant un y égal à la constante de gravitation terrestre), le temps de saut et la force du saut.

Le composant Collider2 représente la hitbox de l'entité. Il contient le rectangle de la hitbox, ainsi qu'une marge.



De plus, j'ai créé deux composants spéciaux, le premier est le composant Camera, qui est un composant auto-instanciable représentant la vue de l'utilisateur, déplacant la carte au gré des déplacements de l'entité du joueur.

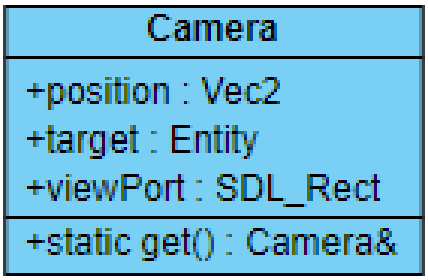


Figure 4.8 – Composant Camera

Le second est le composant Inputs, comportant deux méthodes, *keyUp* et *keyDown*, mettant à jour l'unique attribut du composant, un tableau d'entier représentant l'état du clavier, c'est-à-dire quelles touches sont pressées ou ne le sont pas.

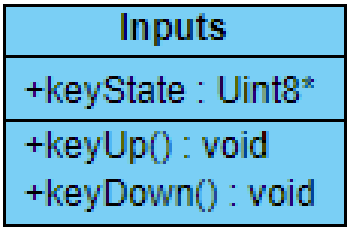


Figure 4.9 – Composant Inputs

4.4 Analyse des systèmes

Du côté des systèmes, on commence par percevoir les entrées, puis on met à jour les composants, avant de les afficher à l'écran.

Tout d'abord, le système d'entrée : la première étape, c'est de voir quelle entité a les composants Inputs et RigidBody. Ensuite, on récupère ces composants grâce au gestionnaire central des composants et des systèmes. Puis on réinitialise la force appliquée au corps de l'entité. Si l'entité a un composant Animation, on la remet sur l'animation d'attente. On perçois après cela les événements. Finalement, on résout les entrées, grâce aux fonctions d'assistance *isKeyUp* et *isKeyDown*.

```
void update(float dt) override {
    SignatureArray& signArr = Manager::get().getSignatureArray();
    for(Entity i = 0 ; i < MAX_ENTITIES ; i++) {
        if(isUpdatable(signArr[i])) {
            Inputs& in = Manager::get().getComponent<Inputs>(i);
            RigidBody& r = Manager::get().getComponent<RigidBody>(i);

            r.clearForce();
            if(signArr[i][getComponentTypeID<Animation>()]) Manager::get().getComponent<Animation>(i).actualRow = 0;

            SDL_Event ev;
            while(SDL_PollEvent(&ev)) {
                switch(ev.type) {
                    case SDL_QUIT:
                        Engine::get().stop();
                        break;
                    case SDL_KEYDOWN:
                        in.KeyDown();
                        break;
                    case SDL_KEYUP:
                        in.KeyUp();
                        break;
                }
            }

            if(isKeyPressed(in, SDL_SCANCODE_ESCAPE)) {
                Engine::get().stop();
            }
```

FIGURE 4.10 – Système d'entrée

```
bool isKeyPressed(Inputs in, SDL_Keycode key) {
    if(in.keyState[SDL_GetScancodeFromKey(key)] == 1) return true;
    return false;
}
bool isKeyPressed(Inputs in, SDL_Scancode key) {
    if(in.keyState[key] == 1) return true;
    return false;
}
```

FIGURE 4.11 – Fonctions d'assistance isKeyUp et isKeyDown

Ensuite, tous les systèmes sont basés sur la même structure : on vérifie que l'entité a bien les composants nécessaires, si oui, il le met à jour, si non on teste la prochaine entité. Par exemple, voici le système de physique :

```
void update(float dt) override {
    SignatureArray& signArr = Manager::get().getSignatureArray();
    for(Entity i = 0 ; i < MAX_ENTITIES ; i++) {
        if(isUpdatable(signArr[i])) {
            Transform& t = Manager::get().getComponent<Transform>(i);
            RigidBody& r = Manager::get().getComponent<RigidBody>(i);

            if(r.type == BodyType::DYNAMIC) {
                r.acceleration.x = (r.force.x - r.drag.x) / r.mass;
                r.acceleration.y = (GRAVITY.y * r.gravScale) + (r.force.y / r.mass);

                r.velocity = r.acceleration * dt;
            }
        }
    }
}
```

FIGURE 4.12 – La fonction update de PhysicSystem

Dans ce système, on calcule l'accélération du corps en fonction de la force, qui est mise à jour dans le système d'entrée, puis la vélocité en fonction de l'accélération et du temps système. La position, elle, est mise à jour dans le système de collision, car si elle était mise à jour avant la gestion des collision, l'entité pourrait passer au travers de la carte.

```
void update(float dt) override {
    SignatureArray& signArr = Manager::get().getSignatureArray();
    for(Entity i = 0 ; i < MAX_ENTITIES ; i++) {
        if(isUpdatable(signArr[i])) {
            Transform& t = Manager::get().getComponent<Transform>(i);
            RigidBody& r = Manager::get().getComponent<RigidBody>(i);
            Collider2& c = Manager::get().getComponent<Collider2>(i);

            t.lastSafePos.x = t.position.x;
            t.position.x += r.velocity.x * dt;
            c.set(SDL_Rect{(int) t.position.x, (int) t.position.y, 32, 32});

            //FIXME: taille de la map
            if(c.box.x < 0 || (c.box.x + c.box.w) > (2 * SCREEN_WIDTH)) t.position.x = t.lastSafePos.x;

            if(CollisionHandler::get().mapColliding(c.box)) t.position.x = t.lastSafePos.x;

            t.lastSafePos.y = t.position.y;
            t.position.y += r.velocity.y * dt;
            c.set(SDL_Rect{(int) t.position.x, (int) t.position.y, 32, 32});

            bool isGround;
            if(CollisionHandler::get().mapColliding(c.box, &isGround)) {
                t.position.y = t.lastSafePos.y;
                if(isGround) r.onGround = true;
            } else {
                r.onGround = false;
            }
        }
    }
}
```

FIGURE 4.13 – La fonction update de CollisionSystem

Chapitre 5

Conclusion

Premièrement, je dirais que je n'ai pas réussi à atteindre les objectifs que je m'étais fixé. Après près de huit mois de travail, je n'ai atteint que deux tiers de l'objectif. J'ai pu conceptualiser un moteur de jeu très simple, basé sur le patterne de l'Entity Component System, et j'ai pu produire une petite démonstration technique de ce moteur.

Cependant, mon objectif n'était pas que le moteur de jeu mais un jeu complet, avec les événements, l'usine à entité, nom donné aux fonctions (ou à la classe) facilitant la création d'entités prédéfinies (le joueur, les personnages non-joueurs, les ennemis...). Il me manque de plus les composants permettant de tirer des projectiles, de faire en sorte que les ennemis poursuivent le joueur, ainsi que les déclencheurs, et plusieurs autres cartes.

Je dirais en conclusion de ce rapport que la conception d'un moteur de jeu est bien différente que celle d'un jeu, et que l'objectif de mon projet aurait dû se limiter au moteur de jeu. Malgré cela, grâce à ce projet, j'ai appris différents patternes de développement que nous n'avons pas vu dans notre Licence, ainsi que la syntaxe du C++.

Bibliographie

Pour la partie graphique :
- Ressources du jeu de tuiles : OPP 2017 - Jungle and temple set
- Ressources du sprite du personnage : Generic Character Asset v 0.2

Pour la partie programmation :
- Recherches sur l'ECS : L'Entity Component System - Qu'est ce que c'est et comment bien s'en servir ? - Guillaume Belz
- Recherches sur l'ECS : Entity Component System - Wikipedia
- Recherches sur l'ECS : Evolve your hierarchy - cowboyprogramming
- Recherches sur l'ECS : Implementation of a component-based entity system in modern C++ - Vittorio Romeo à la CppCon 2015
- Recherches sur les composants : Documentation officielle de Unity2D