Application informatique

Another Side-scrolling Platformer Engine

Réalisé par : Séverine Delaplace
Amaury Fauvel
Quentin Godart
Aurélien Villette

Table des matières

1 Introduction

1.1 Présentation

Another Side-scrolling Platformer Engine (ASPE) est la démonstration d'un jeu-vidéo en 2 dimensions à défilement horizontal (la caméra se déplace latéralement avec le joueur). Le joueur s'y déplace librement avec les flèches du clavier et la barre d'espace.

Ce type de jeu figure parmi les plus représenté dans l'histoire du média interactif, avec de nombreux titres marquant comme Super Mario, Sonic the Hedgehog ou Rayman. Le but d'un tel jeu est de parcourir des niveaux (souvent de gauche à droite) parsemés d'obstacle et ennemis.

Super Mario Bros - Ninendo - 1985

L'éxécutable se trouve dans `build-debug` sous le nom `./jeu2d`.

1.2 Objectifs du projet

Le but du projet est de réaliser un moteur graphique adapté aux jeux de plateforme. Il doit être doté d'une architecture robuste et extensible qui pourrait être utilisée dans le cadre d'un jeu complet. Le nombre de fonctionnalités est minimal car l'objectif est de fournir la base d'un jeu, et non un jeu complet.

2 Choix de programmation et workflow

Langage

Le langage utilisé pour l'intégralité du projet est le C++. Le C++est un langage orienté objet influencé par le C. Un de ses principaux atouts est la gestion fine des ressources et ses performances. C'est un langage de choix pour toute application en temps réel employant des calculs graphiqueset est couramment utilisé par les studios de développement de jeux.

C'est un langage que nous n'avons pas appris au cours de notre scolarité, c'était donc l'occasion pour nous de nous familiariser avec un nouveau langage.

Bibliothèque graphique

La bibliothèque graphique employée est le Simple and Fast Multimedia Library (SFML). C'est une bibliothèque assez bas niveau, elle nous permet de gérer efficacement les ressources (images affichées). Elle a l'avantage d'être conçue pour les langages orientés objet, c'est pourquoi nous l'avons préféré à la Simple DirectMedia Layer (SDL).

La SFMl est séparée en différents modules :

— Window : gère l'ouverture de la fenêtre basée sur OpenGL. C'est dans ce contexte que se déroule le jeu. Sert aussi à récupérer les inputs du clavier.
— Graphics : implémente le chargement de textures, l'affichage de sprites et la gestion de la caméra.
— System : pour les configurations diveres (taux de rafraîchissement notamment)
— Sound : gère les sons et pistes audio (malheureusement non utilisé ici)
— Network : pour les communications en réseau (non utilisé ici)

Méthodes de travail

Nous avons choisi l'environnement de développement CLion (de la suite JetBrains) pour ses fonctionnalités utiles comme l'intégration de Git, notre gestionnaire de versions.

Comme outil de communication, nous avons utilisé Slack. Slack est une application web de chat adapté au monde du travail grâce à son système de salon et à la centralisation d'autre application comme Trello.

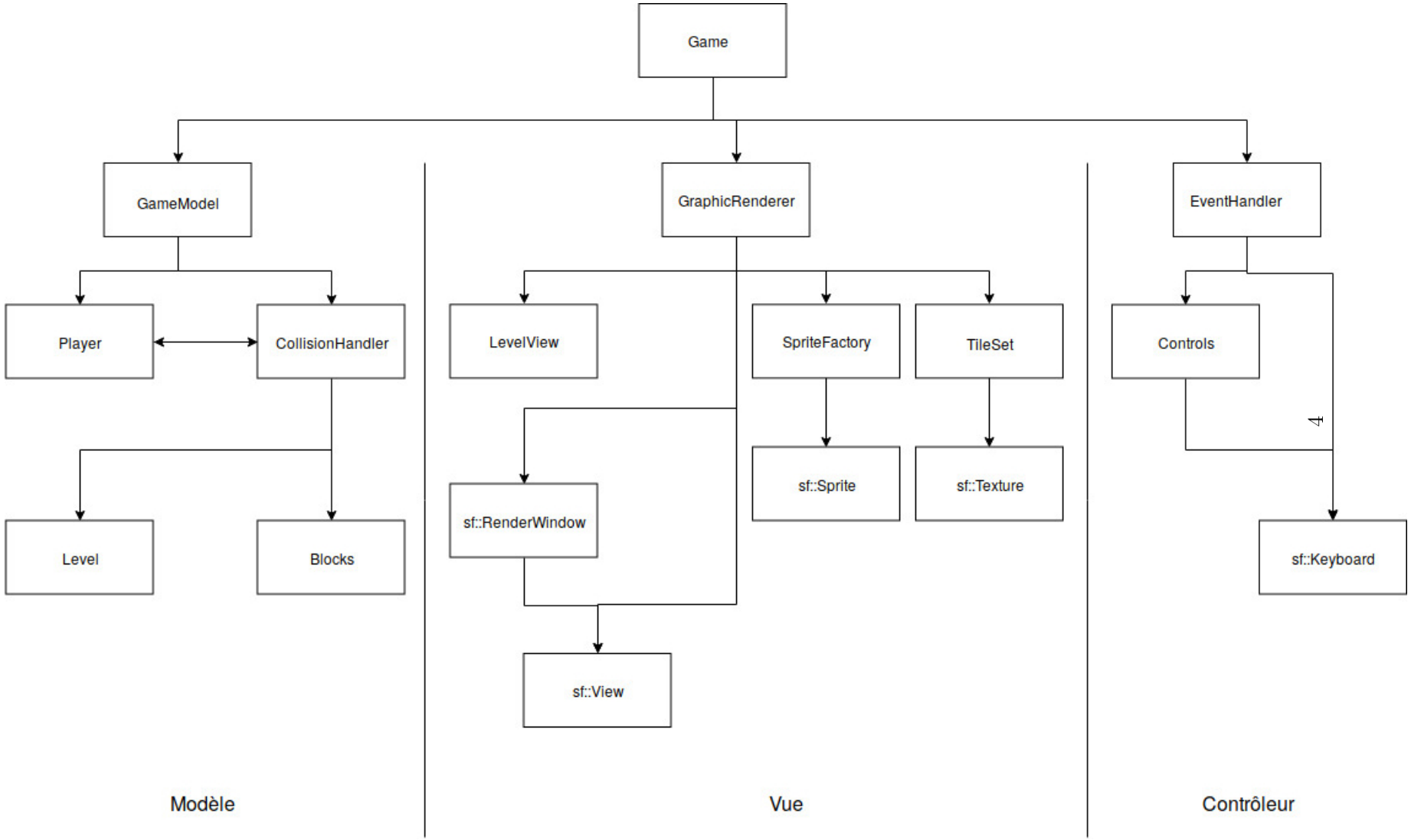
Trello se présente sous la forme d'un tableau où l'on affiche les différentes tâches à faire, en cours et terminées, assignables aux membres du groupe. Finalement, nous avons rarement utilisé Trello car nous n'étions que 4 et nous côtoyons tous les jours.

3 Implémentation

La qualité de l'architecture était notre principale préoccupation. Elle permet d'avoir une base de jeu largement extensible, car la quasi-totalité des fonctionnalités d'un jeu complet ne concernent que des ajouts au modèle du jeu.

Nous avons choisi une architecture Modèle-Vue-Contrôleur. Elle permet une grande maintenabilité et une clarté du code. De plus elle connaît son lot de bonne pratique qui assure la robustesse du programme.

Le jeu étant en temps réel, une boucle principale tourne jusqu'à la fermeture de la fenêtre. Dans cette boucle, la classe principale `Game` agit comme un chef d'orchestre et appelle tour à tour le `EventHandler` qui récupère les inputs clavier, le `GameModel` qui met à jour tout le modèle à partir des inputs récupérés, et le `GraphicRenderer` qui affiche le niveau et les entités qui y évoluent. Un tour de boucle est l'unité temporelle de base, appelé *tick*.


Game
GameModel
GraphicRenderer
EventHandler
Player
CollisionHandler
LevelView
SpriteFactory
TileSet
Controls
Level
Blocks
sf::RenderWindow
sf::Sprite
sf::Texture
sf::Keyboard
sf::View
Modèle
Vue
Contrôleur

Ces trois gestionnaires ne devraient être instancié qu'une seule fois. Nous avons pensé à les implémenter suivant le patron de conception singleton mais cela compliquait un code qui n'a pas pour but d'être distribué et utilisé largement (dans une équipe réduite aucun de nous n'aurait essayé de créer plusieurs de ces objets).

3.1 Contrôleur

La partie contrôle est la plus simple : le `EventHandler`, par sa méthode `pullEvents()`, récupère l'état de chaque touches associée à un contrôle avec la méthode SFML :

```
sf::KeyBoard::isKeyPressed()
```

L'association entre les contrôles (directions, saut) et les touches du clavier (espace, flèches) est faite par l'espace de noms `Controls`.

> Note : en C++, pour `Nom::fonction()`, `Nom` est un espace de nom pour une collection de fonctions utilitaires.
> `sf` est l'espace de nom de la SFML.

3.2 Modèle

La classe `GameModel` s'occupe de modéliser le jeu. sa méthode principale, `update()`, fait se dérouler un tour de jeu et met à jour toutes ses composantes.

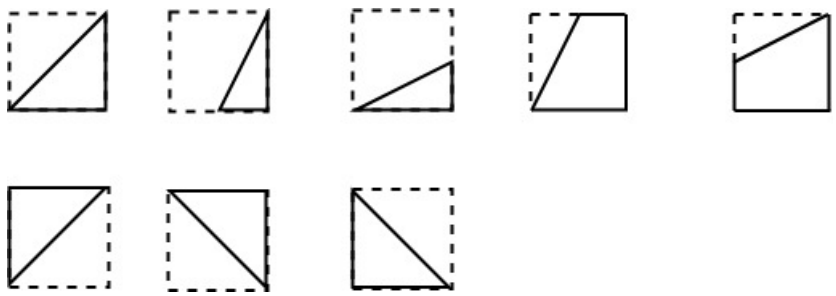
Level

Le jeu étant en 2 dimensions, tous les objets évoluent sur un repère orthonormé dont l'unité est le pixel. l'origine du repère se situe en haut à gauche du niveau, les coordonnées dans le niveau sont donc positives (x augmente en allant vers la droite et y augmente en allant vers le bas).

Le terrain du niveau est décomposé en blocs de 16×16 pixels alignés sur une grille. Les blocs sont représentés par des entiers codés sur 1 octet. Ils sont énumérés dans l'espace de nom `Blocks`.

Il existe 3 types fondamentaux de blocs : l' `Air` pour l'espace libre, le `Ground` pour le sol et le `Wall` pour les murs et plafond. On ne testera les collisions que vers le bas dans le cas d'un `Ground`, ce qui limite le nombre d'opérations. (voir Collisions) alors que le `Wall` est solide dans toutes les directions.

Ces deux derniers blocs sont déclinés en 3 types de pentes ($45°$, $22,5°$ et $67,5°$) déclinées dans toutes les directions.

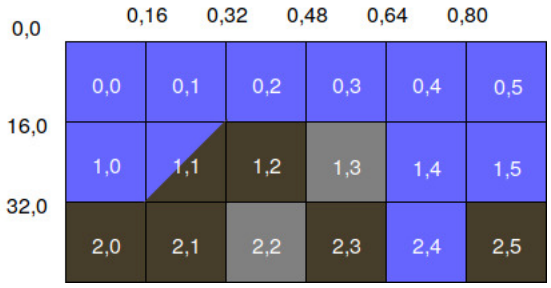


Cette variété de blocs entraîne la présence de 42 items, que la classe `Blocks` encapsule avec les fonctions suivantes :

`isGround(block b)`	Renvoie *VRAI* si b est un sol ou dérivés
`isWall(block b)`	Renvoie *VRAI* si b est un mur ou dérives
`isAir(block b)`	Renvoie *VRAI* si b est vide
`isSlope(block b)`	Renvoie *VRAI* si b est une pente
`isUp(block b)`	Renvoie *VRAI* si b est une pente montante
`isDown(block b)`	Renvoie *VRAI* si b est une pente de plafond
`slope(int x, block b)`	Renvoie la hauteur y correspondant au décalage x horizontal dans le bloc

Les terrains de niveaux sont stockés dans des fichiers à l'emplacement `asset/map/`*niveau*`/level.bin` dans une matrice d'entiers codant les blocs. Il est à noter que le terrain du niveau et son affichage sont séparés et indépendant (voir LevelView dans la partie Vue).

0	0	0	0	0	0
0	12	1	22	0	0
1	1	22	1	0	1



Actuellement il n'est pas permis à l'utilisateur de changer de niveau. Seul le niveau de test est chargé au lancement.

Player

La classe `Player` implémente l'entité que contrôle l'utilisateur. Ses principaux attributs sont sa position et sa vitesse (en *pixel*/*tick*) représentés par des couples d'entier (`Pair`). Chaque tour de jeu, le `GameModel` ajuste la vitesse du `Player` en fonction des inputs récupérés. Puis le `Player` se met à jour selon son comportement :

1. La gravité augmente la vitesse dans les positifs (vers le bas).
2. Le frottement réduit la vitesse horizontale. Ainsi la vitesse est réduite progressivement dans les sauts.
3. On empêche les vitesses de dépasser un maximum.
4. Si on est sur le sol et que la vitesse est positive (vers le bas), on ramène à 0 la vitesse.

5. On essaie de se déplacer de *speed* pixels, en appliquant les collisions (voir `CollisionHandler`).

Actuellement le `Player` est la seule entité du jeu. Pour un moteur de jeu plus complet il aurait fallu gérer un ensemble d'`Entity` et appliquer une fonction `update()` à chacune d'elle. L'affichage aurait aussi d'afficher les entités une à une.

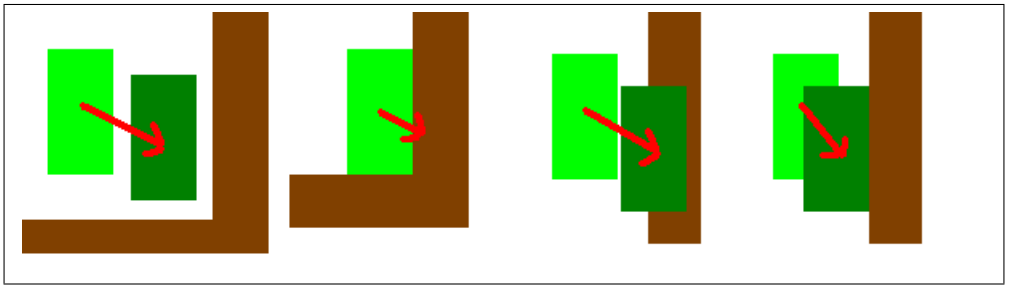
CollisionHandler

Le gestionnaire de collision est l'outil qui indique au `Player` les blocs avec lesquels il entrera en collision. Sa principale méthode est :

```
std::vector<block> getColliders(Pair pos, Pair size);
```

Cette méthode prends en paramètres deux couples d'entier : la position hypothétique et la taille de l'entité. Le `CollisionHandler` teste chaque pixel de la zone et ajoute chaque bloc trouvé dans un tableau dynamique (`vector` du C++). La précision au pixel près est un peu plus coûteuse que de tester 1 pixel sur 16, mais cc'est le seul moyen de gérer les pentes.

Si le tableau renvoyé est vide, le joueur pourra se déplacer sans souci. Sinon il essaiera dans chaque direction de se déplacer pixel par pixel pour se coller au bloc sans rentrer dedans. Aussi il teste les pixels un peu au dessus pour monter la pente (et ne pas être bloqué par 1 pixel).



3.3 Vue

C'est la partie la plus sensible car elle va déterminer les performances du moteur : les accès aux fichiers d'images, et le rendu graphique est ce qu'il y a de plus coûteux. C'est la classe `GraphicRenderer` qui fait la gestion des composantes graphiques. La méthode principale `render` prends en paramètre le `Player` du modèle, et s'occupe d'afficher son environnement.

sf::Sprite

La SFML gère les images par des sprites. C'est à dire un rectangle d'une taille définie sur lequel est plaquée une texture. La texture est l'image du sprite. Lorsqu'on plaque une texture sur un sprite, on donne la partie de la texture que l'on veut garder. Exemple :

```
sf::Texture texture;
texture.loadFromFile("image.png");
sf::Sprite sprite;
sprite.setTexture(texture);
sprite.setTextureRect(sf::IntRect(10, 10, 32, 32));
```

Cette portion de code charge la texture stockée dans `image.png` et l'affecte à un sprite. L'enjeu est de minimiser les appels à `loadFromFile()` et les déclarations de `sf::Texture`

LevelView

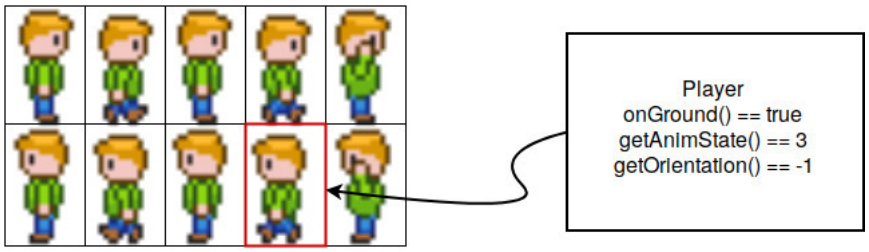
La classe `LevelView` stocke la partie visuelle d'un niveau d'une manière similaire au `Level` du modèle : les tuiles (tiles) sont représentées par des entiers codés sur 1 octet et stocké dans un fichier binaire (emplacement `assets/map/`*niveau*`/view.bin`). Cette fois, pas d'énumération, car l'entier correspond directement au décalage de la tuile sur le tileset.

Ainsi, une seule image contient toutes les tuiles correspondant à une seule texture à plaquer sur un sprite. La différence visuelle entre deux sprites est uniquement due à la portion de texture plaquée avec `sprite.setTextureRect()`.

SpriteFactory

Cette classe gère l'association d'un sprite à une entité. Pour cela le patron de conception fabrique a été employé : la méthode `sf::Sprite create(Player entity)` prends en paramètre une entité et renvoie le sprite adéquat. La feuille de sprite du joueur contient toutes les étapes de l'animation, et c'est la `SpriteFactory` qui affecte la bonne partie de l'animation en fonction de l'état du joueur.

Si le joueur est en l'air son sprite sera celui du saut/chute. Sinon un entier qui évolue de manière cyclique lorsque le joueur marche sert à choisir l'étape de l'animation de marche.



sf : :View

La SFML gère la caméra par cette classe. Il suffit d'une translation de tout l'affichage vers une zone centrée sur le joueur pour avoir un défilement horizontal.

```
camera.setCenter(player.pos.x, SCREEN_HEIGHT / 2);
```

3.4 Éditeur de niveaux

Nous avions prévu de créer un éditeur de niveau, mais étant un "accessoire" nous nous sommes concentrés sur le jeu. Pour créer un niveau, nous utilisons un éditeur de texte pour écrire les nombres en base 62 (chiffres, lettres minuscules et lettres majuscules). Un programme écrit en C fait la conversion vers un fichiers binaire avec les bons nombres.

4 Répartition des tâches

Amaury : Base de l'architecture, gestion et affichage du niveau (`Level` et `LevelView`), outil de conversion texte - binaire.
Aurélien : Physique du joueur et collisions
Séverine : Physique du joueur et collision, outils sur les blocs
Quentin : Association sprite-entité, dessin des sprites et création du niveau de démonstration.

5 Pistes d'amélioration

Pour graphique minimal mais complet, il aurait fallut gérer une liste d'entités et pas un unique `Player`.
Aussi il faudrait un gestionnaire de niveaux pour pouvoir charger et décharger des environnements (pour recommencer un niveau à la mort, et passer au niveau suivant à la fin).
Enfin, nous n'avons pas prévu l'affichage d'interface (informations sur le joueur, bulles de texte...)

6 Conclusion

En faisant le choix de développer ce projet en C++avec la bibliothèque SFML nous nous sommes imposé une première difficulté, celle d'apprendre le langage et son fonctionnement avec la bibliothèque graphique.

De plus, l'idée du sujet étant notre initiative nous avons du nous même établir une ligne directrice et nous y tenir. Le sujet étant vaste et nouveau, nous avons dû apprendre les différentes principes de fonctionnement de moteurs graphiques ceux qui nous a valu d'explorer de nombreuses pistes.

Ce projet nous a apporté de l'expérience sur un nouveau type d'application, sur un travail en groupe plus poussé pendant une durée plus longue, sur la gestion d'un projet en partant de sa création et sur l'apprentissage d'un nouveau langage avec de nouveaux outils.