ESIEE PARIS

Année 2019-2020
Semestre 1

Rapport Intermédiaire de Projet E4

Construction d'un moteur de jeu vidéo

Projet réalisé par :

CHOMICKI Clément • BARUZY Amaury • PARE Barbara
LEBLON Alexandre • GOISLARD Nicolas • MONTANI Maÿlis
ZHOU Sebastien • DELADRIERE Hugo • SALLET Aymeric

Projet encadré par :

Nabil MUSTAFA • Sylvain GLAIZE

PRJ 4000S
Numéro du projet : E16

une école de la CCI PARIS ILE-DE-FRANCE

Remerciements

Nous remercions notre professeur encadrant Nabil MOUSTAFA de nous avoir suivi lors de la réalisation de ce projet et pour avoir organisé de multiples réunions aux cours desquelles nous avons pu obtenir des réponses claires à nos interrogations.

Nous remercions également Sylvain GLAIZE pour avoir proposé ce sujet à l'ESIEE et nous avoir permis de travailler sur un projet motivant et très complet. Nous le remercions aussi pour ses conseils dans le suivi du projet et du temps qu'il a investi dans les quelques réunions que nous avons eu en sa présence.

Nous remercions enfin l'ESIEE Paris pour nous avoir fourni une salle durant ces quelques mois d'étude du projet.

Sommaire

I - Introduction

Dans le cadre de notre deuxième année du cycle ingénieur à l'ESIEE, nous avons dû choisir parmis plusieurs sujets proposés à l'école par des entreprises. Nous avons choisis de réaliser un projet de la filière informatique proposé par Sylvain GLAIZE, représentant de l'entreprise UBISOFT, intitulé : « Construction d'un moteur de jeu vidéo ». Nous avons par la suite été encadrés par Nabil MOUSTAFA, notre professeur référent à l'école.

Notre groupe de projet est composé de 9 membres : Clément CHOMICKI, Maÿlis MONTANI, Alexandre LEBLON, Amaury BARUZY, Barbara PARÉ, Sebastien ZHOU, Nicolas GOISLARD, Hugo DELADRIERE et Aymeric SALLET.

La rotation des membres au cours de ce projet est organisée comme suit :

- Clément CHOMICKI, Alexandre LEBLON, Amaury BARUZY et Nicolas GOISLARD sont des membres du projet constants qui seront présents pendant toute la durée du projet.
- Maÿlis MONTANI et Barbara PARÉ sont des membres du projets qui partiront à l'étranger au deuxième semestre de E4 et qui ne participeront que pendant le premier semestre.
- Sebastien ZHOU, Hugo DELADRIERE et Aymeric SALLET sont des membres du projet qui sont partis à l'étranger au premier semestre de E4, et qui ne participeront qu'au deuxième semestre.

II - Objectifs (Sujet Original)

L'objectif du projet est la construction d'un moteur de jeu vidéo, et d'un jeu "test" pour prouver son fonctionnement.

Ce moteur de jeu devra être capable de :

1. Rendre une scène à l'écran.
2. Pouvoir y déplacer un élément contrôlé par un joueur.
3. Y faire se déplacer des éléments contrôlés par le moteur.
4. Résoudre des collisions entre les éléments de jeu.
5. Plus largement, résoudre des interactions entre les éléments de jeu.
6. Afficher des éléments de signes et feedbacks au joueur (texte et/ou particules, son).

En fonction du temps et de l'avancée, le moteur peut aussi intégrer (par ordre de complexité) :

1. Une gestion de musique d'ambiance.
2. Gérer un écran de titre et différents niveaux.
3. Des éléments animés par déformation.
4. De la communication entre deux moteurs (jeu en réseau).

Les étudiants devront entre autre :

1. Identifier les différentes parties constituantes d'un moteur de jeu vidéo.
2. Implémenter ces parties en veillant à leur modularité / architecture.
3. En architecture, déterminer en particulier sur quel modèle les entités du monde virtuel sont gérées.
4. Optimiser le programme (CPU/GPU ainsi que mémoire).
5. Décider et implémenter un "jeu test" pour montrer les possibilités du moteur.

III - Etat de l'art

III.1 - Qu'est ce qu'un moteur de jeu ?

Un moteur de jeu vidéo est situé entre le jeu vidéo en lui même et la plateforme (OS ...), voire le matériel (console, ordinateur…). Il permet de prendre en compte les spécificités du matériel tout en offrant aux développeurs de jeux vidéos une abstraction suffisante pour ne pas se préoccuper des services généralement rendus par un jeu complet. En somme, c'est la partie bas niveau du code qui peut servir à plusieurs jeux.

En nous renseignant sur la structure des moteurs de jeux vidéos, nous avons, par le biais d'internet et des explications de Sylvain GLAIZE, pu dissocier plusieurs parties composant cette entité :

- Une partie **ECS** : Cette partie du moteur est un paradigme qui permet de maximiser la séparation entre le code et les données, elle permet également de rendre plus facile le modding, et est en théorie efficace au niveau de la gestion du cache. Elle représente entre autre la logique interne du jeu.
- Une partie **gestion des entrées** : Cette partie du moteur permet de gérer les entrées (clavier, sourie, manette, joystick…) et permet au joueur d'interagir avec le jeu.
- Une partie **graphique** (gestion des sorties) : Cette partie permet de gérer les objets du jeux et leur affichage dans l'espace. A partir de la position et de l'orientation de la caméra, elle permet de projeter une vue sur la scène 3D en 2D en respectant les perspectives des objets et leur éloignement à la caméra. Parmis les moteurs graphiques les plus connus, on peut citer OpenGl et Vulkan, ou bien Direct3D.
- Une partie **physique** : Cette partie permet de gérer la cohérence des objets, de gérer la résistance des objets, leurs collisions entre eux et avec le sol ou les murs, et enfin de simuler la gravité dans l'univers. Il existe par exemple Bullet développé en C et C++, ou bien nphysics développé en Rust, qui permettent la gestion de la partie Physique du moteur de jeu vidéo.
- Une partie **son** : Cette partie permet de gérer les différentes pistes musicales ou les effets sonores dans le jeu. La partie son permet d'ouvrir, de jouer des pistes dans le jeu, et éventuellement de modifier des pistes d'une certaine façon (echos, amplitude, oscillations, effet Doppler….) pour améliorer le rendu et adapter les sons au Gameplay.
- Une éventuelle partie **intelligence artificielle** : Cette partie permet de gérer le comportement des ennemis ou des personnages du jeu et de simuler leur intelligence.

III.2 - Historique

À la naissance des jeux vidéos, les développeurs utilisaient de l'assembleur. Ce langage permet notamment d'avoir une gestion directe de la mémoire et suffisait à l'époque puisque les performances des machines étaient très loin derrières les machines que nous avons à disposition de nos jours et ne permettaient pas de créer des jeux très complexes, ni de facilement réutiliser du code.

En avançant dans les années, les développeurs se sont dirigés vers le C. Ce langage est rapide et plus accessible que l'assembleur. Il permet de faire des choses plus complexes plus facilement qu'avec l'assembleur, bien que la gestion de la mémoire soit moins directe. Doom est le premier jeu possédant un moteur de jeu nettement séparé du reste et réutilisable, l'Id Tech 1 (ou Doom Engine).

Plus récemment, les développeurs ont commencé à utiliser le C++ pour leurs moteurs de jeux vidéos. Ce langage a permis de s'éloigner un peu plus de la gestion de la mémoire tout en gardant les performances et la rapidité du C. Entre les différentes versions de C++, les moteurs de jeux vidéos sont devenus de plus en plus évoluées avec un niveau plus grand d'abstraction. Certains moteurs de jeux vidéos possèdent encore des morceaux écrits en C car les développeurs n'ont pas nécessairement le temps ni l'argent pour réécrire ces parties en C++.

Nous pouvons aussi voir que d'autres langages ont pu être utilisés pour le développement des moteurs de jeux vidéos. Par exemple, certains ont décidé d'écrire des moteurs en C# car c'est un langage plus facile à appréhender que le C++, bien qu'il soit plus lent. Il permet notamment de faire du plus haut niveau pour des opérations plus compliqués, en liant différents codes écrits en C++ pour continuer d'être rapide. De manière générale, très peu de moteurs de jeux vidéos ont été écrits en Java ou en Python pour des question de performances. Ces deux langages sont lents et ne permettent pas d'obtenir des rendus dans des temps décents.

En avançant dans le temps, nous constatons que nous utilisons des langages plus puissants et en adéquation avec les performances de nos machines, bien que cela nous fasse perdre du contrôle direct sur la gestion de la mémoire. En effet, utiliser des langages de plus haut niveau nous permet de gagner du temps dans le développement, ce qui vaut parfois les quelques pertes de performance.

En faisant des recherches des différents moteurs de jeux actuels que nous pouvions trouver sur internet, nous avons découvert que les moteurs de jeux étaient majoritairement réalisés en C, en C++ et en C#. Par exemple, le moteur de jeu "Unreal Engine" a été développé en C++ par Epic Games ; "Unity" a été développé en C# par Unity Technologies ; et "ANVIL" a été développé en C++ par UBISOFT. "ANVIL" a notamment été utilisé pour réaliser certains Assassin's Creed, et contient (pour référence: wikipedia…) plus de 7 millions de lignes de code.

Bien qu'aujourd'hui les entreprises continuent d'utiliser des langages tels que que C et le C++ pour développer leurs moteurs de jeu, les nouvelles générations tendent à utiliser des langages plus récents tels que le Rust.

IV - Décisions préalables

IV.1 - Choix du langage

Aux vues des objectifs énoncés précédemment, nous nous sommes dans un premier temps concertés avec Nabil MOUSTAFA et Sylvain GLAIZE afin de déterminer le langage de programmation dans lequel nous voulions construire notre moteur de jeu vidéo.

Clément étant déjà un utilisateur aguerri de Rust et disposant de connaissances basiques sur des moteurs de jeux vidéos réalisés en Rust, il nous a proposé de réaliser ce projet dans ce langage. Étant donné notre faible expérience en C++ à l'époque, l'idée d'apprendre un nouveau langage que nous n'aborderons pas à l'école nous a tous motivés. En effet, il semblait plutôt intéressant de développer des compétences dans un langage forçant à prendre de bonnes habitudes dans la gestion de l'ownership, car ce sont des compétences précieuses applicables dans énormément de langage dont le C++.

Il est également important de noter que le Rust est un langage très strict sur la gestion de l'ownership des données, sur les lifetimes et le typage. Cela permet aux compilateurs Rust de déduire énormément d'information du code, ce qui les rend capable de plus d'optimisations et d'exprimer des messages plus clairs au développeur en cas d'erreurs.

Nous avons ensuite parlé avec Nabil et Sylvain pour connaître leurs avis sur le fait d'implémenter le moteur de jeux en Rust. Sylvain nous a signifié qu'il était intéressant pour lui que nous l'implémentions en Rust. Nabil nous a cependant averti du fait qu'il ne pourrait pas nous aider si nous choisissons ce langage plutôt que le C++.

Après quelques discussions, et malgrés ces quelques risques, nous avons décidé d'implémenter notre projet en Rust. Cette décision, qui peut sembler ambitieuse, permet à la fois de coller aux recommandations de notre professeur encadrant ainsi qu'aux exigences de base du projet, c'est-à-dire de faire quelque chose d'ambitieux et d'innovant. Il existe en effet peu de moteurs de jeux vidéos écrits en Rust, ce qui nous garanti un moteur plus unique que si nous l'avions implémenté en C++, langage dans lequel énormément de moteurs de jeux vidéos ont déjà été réalisés.

Nous avons également commencé à implémenter ce moteur de jeu avec une connaissance quasi-nulle en moteur de jeux vidéos. Nous ne serons donc pas biaisés par des expériences/connaissances passées, et cela nous assure que notre moteur de jeu sera le fruit d'une approche originale.

La plupart d'entre nous ne connaissant pas Rust, nous avons alors commencé à faire des cours en groupe avec Clément CHOMICKI qui nous a appris les bases du langage.

IV.2 - Choix de la plateforme de développement

Nous avons en parallèle cherché à nous accorder sur la plateforme de développement de notre moteur de jeu. Nous avons longtemps cherché des solutions pour pouvoir implémenter notre jeu en cross-plateforme sur Windows et Linux, mais aux vues du peu d'outils supportant le Rust sous Windows (nous avons notamment utilisé Visual Studio Code), nous avons finalement quasiment tous choisis de passer sous Linux. Nous avons commencé par essayer de développer via des machines virtuelles puis nous avons, après quelques semaines, décidé d'installer un double boot Windows/Linux sur les ordinateurs de Barbara, Maÿlis et Alexandre. Cette installation a été réalisée à la suite de divers problèmes liés à la gestion des cartes graphiques par les machines virtuelles.

À ce jour Clément développe sous emacs ; Alexandre, Amaury et Nicolas principalement sous Visual Studio Code ; Maÿlis et Barbara sous gedit

IV.3 - Choix des librairies

Après ces quelques réglages nous avons pu commencer à étudier plus en profondeur les moteurs de jeux vidéos et nous renseigner sur leur fonctionnement. Nabil nous a tout d'abord présenté un tutoriel nommé "Learn OpenGL" détaillant l'utilisation et les spécificités d'OpenGL. Après plusieurs jours de lecture nous avons conclu que ce tutoriel détaillait plus le principe d'OpenGL que la façon dont nous devions l'utiliser. Il fournit cependant des morceaux de code intéressants qui nous ont permis de comprendre comment et pourquoi utiliser OpenGL.

À la suite de notre deuxième réunion, Sylvain nous a fournis un tutoriel du nom de "Rust and OpenGL from scratch". Après étude de ce tutoriel, nous avons remarqué que cela ne correspondait pas à nos besoins. En effet, ce tutoriel a été développé par un professeur qui précise bien qu'il a appris en réalisant le tutoriel et qu'il n'était donc qu'un débutant à l'heure où il l'a écrit. Nous avons donc décidé de chercher à remplacer la crate GL par Glium, un safe wrapper d'OpenGL écrit en Rust facile d'utilisation, bien documenté et alimenté par une petite communauté. Glium est l'alternative la plus pratiquable que nous ayons trouvé pour faire de l'OpenGL avec Rust.

Pour l'ECS, Clément a d'abord passé un long moment à tenter une implémentation à nous, mais il s'est avéré illusoire que quelques étudiants arrivent à la cheville d'équipes de professionnels qualifiés ayant travaillé pendant des années sur SPECS. Ici, faire les choses à moitié n'aurait pas suffit. Ainsi pour l'ECS, nous avons choisis SPECS. L'alternative aurait été Legion, mais cette dernière n'est pas à un stade de développement suffisant pour vraiment faire concurrence à la première. De plus, Clément avait déjà utilisé SPECS auparavant et savait l'utiliser.

Pour la physique, nous avons d'abord cherché des bindings de Bullet pour Rust, mais il s'est avéré qu'aucun binding utilisable n'existe. En effet, les seuls binding que nous avons trouvé datent de plus de 2 ans et ne possèdent pas de documentation. Il est sûrement judicieux de penser qu'ils ont été abandonnés, et donc, d'aucune utilité pour nous car Rust est en constante évolution (les références sont disponibles dans la sitographie). Ainsi, nous nous sommes dirigés vers NPhysics, qui est utilisé par Amethyst, un autre moteur de jeu écrit en Rust. La grande quantité d'exemples, de tutoriels et la grande qualité de la documentation nous a décidé sur son adoption. Selon le créateur de NPhysics, celui-ci n'égale pas encore Bullet mais cela ne saurait tarder.

Pour la gestion d'entrées, nous avons choisis Glutin, un module utilisable avec des contextes OpenGL et contenant des détection d'entrées. Le choix s'est porté sur ce module pour sa compatibilité avec Glium (que nous utilisons pour la partie graphique).

IV.4 - Spécificités de notre jeu test

Au cours des premières réunions avec Nabil concernant notre projet, il nous a été demandé de penser aux spécificités de notre jeu. Nous devions imaginer une ambiance, le type de graphismes, l'objectif et les possibilités offertes dans notre jeu de test. Cela allait nous permettre d'orienter le développement de notre moteur de jeu vidéo pour avoir des objectifs clairs de ce qu'il nous fallait développer.

Sous le conseil de Nabil, nous avons donc choisis de ne pas nous limiter dans notre imagination, car il valait mieux, selon lui, être ambitieux et ne pas réussir à aller au jusqu'au bout plutôt que de se fixer des objectifs que nous savions atteignables, et donc apprendre moins.

Voici donc une liste non exhaustive des quelques idées que nous avons eu :

- Notre univers sera en 3D.
- Notre jeu test aura une ambiance vaguement horrifique.
- Il y aura dans notre jeu des portails (téléportation…).
- La caméra peut aller de la première à la 3ème personne et gère les collisions avec l'environnement.
- Il sera possible pour le personnage principale de se déplacer dans les airs
- Il sera possible pour le personnage de ramasser et déposer des entités, d'avoir un inventaire et un familier.
- Les PNJ auront eux même des inventaires, pourront ramasser et déposer des entités, et gérer leurs déplacements.
- Les entités autres que le joueurs pourront interagir (ex : lapin+carotte).
- Il y aura une boucle de Gameplay à parcourir pour arriver au but.
- But du jeu : Escape Game.
- Le jeu sera extensible en multijoueur.

IV.5 - Mise en commun du travail et des rapports de réunions

Pour mettre en commun notre travail, nous avons décidé d'utiliser GitHub. Cet outil nous permet de partager rapidement des fichiers et du code. GitHub permet une gestion des conflits de version facilitant le travail de groupe.

Amaury est le propriétaire du projet sur la plateforme et gère la cohérence de notre dépôt. Il a notamment rendu le dépôt public afin que notre travail soit accessible à notre groupe et aux superviseurs du projet.

Au début du projet, nous n'avions pas fixé de règles concernant les commit sur la plateforme mais nous essaierons de les rendre plus clairs et propres à l'avenir.

A l'issue de chaque réunion, nous publions les compte rendus de réunion dans le dossier *RAPPORTS/réunions_avec_prof/* afin d'avoir un suivi clair de ce qu'il s'est dit pendant celles-ci. Cela nous permet aussi de dater l'avancement du projet et de garder une trace de nos réflexions/discussions avec nos référents. Nous avons par ailleurs choisis de rédiger nos rapports en Markdown pour les rendre plus lisibles.

Nous avons également ajouté plusieurs fichiers d'aide à la prise en main des différents outils sur le repos. Il y a notamment un README.md qui indique les quelques informations principales relatives au projet et des indications concernant l'installation de cargo (pour Rust), et des renvois vers d'autres fichiers pour des explications plus approfondies sur l'utilisation de GitHub et de Rust.

Le lien de notre dépôt Github est disponible en Annexe.

V - Déroulement du projet

V.1 - Répartition des tâches

Nous nous sommes répartis les rôles comme suit :

- Clément CHOMICKI a été désigné chef de projet. Il se charge de la structure de notre moteur de jeu et de l'implémentation des différents éléments de ce moteur de jeu en Rust.
- Maÿlis MONTANI et Barbara PARE s'occupent des réglages sur la partie graphique, c'est à dire l'implémentation des matrices de rotation, l'angle de vue de la caméra, l'ajout de différents programmes de lecture des objets selon leur texture par lecture de fichiers fragments et vertex, charger et décharger des ressources ;

- Alexandre LEBLON se charge de la partie physique, c'est à dire la cohérence des objets entre eux, leur résistance, la gravité et leurs collisions avec d'autres objets ou le sol et les murs ;
- Amaury BARUZY se charge de la gestion des exceptions et de la cohérence de notre plateforme GitHub ;
- Nicolas GOISLARD se charge de de la Graphical User Interface (GUI) ;

V.2 - ECS

Un jeu vidéo est avant tout un programme traitant une grande quantité de données. Pour gérer ces données (c'est à dire la logique du jeu), nous avons opté pour le paradigme Entity Component System plutôt que l'orienté objet, car cela permet de séparer totalement le code des données. Cela rend le modding beaucoup plus aisé, et pourrait permettre de rendre le moteur encore plus générique. Les ECS permettent aussi une très forte parallélisation, ce qui est critique dans des programmes manipulant autant de données. Aussi, les ECS sont dites plus performantes en cache que l'orienté objet, mais il est clair que si gain il y a, il sera négligeable devant le coût de rendu graphique. Ce n'est donc pas sur ce point que nous nous appuyons pour notre choix. De plus, Clément connaissait déjà bien SPECS, une implémentation d'ECS plutôt populaire en Rust, déjà utilisée par Amethyst, un autre moteur de jeu open source développé en Rust.

Là où, dans l'orienté objet, nous avons divers objets responsables de leur données et du code les affectant mis en relation, dans une ECS, les objets sont représentés par des entitées et des composants. Les composants sont les différents "services" en donnée rendus. Par exemple on peut définir un composant "position" qui contiendra une position. Une entité est globalement un ensemble de composants. Ces composants sont stockés dans leurs stockages respectifs, qui sont alignés en mémoire. Ainsi, on peut représenter l'ensemble des composants comme une table de type "base de donnée", où chaque colonne est un type de composant et chaque ligne une entité.

Les systèmes sont la partie "code" de l'ECS, ce sont des procédures appelées périodiquement (typiquement, à chaque tick de jeu), qui vont appliquer des opérations sur les composants (par exemple, modifier les composants "Position" des entitées possédant aussi un composant "Vitesse").

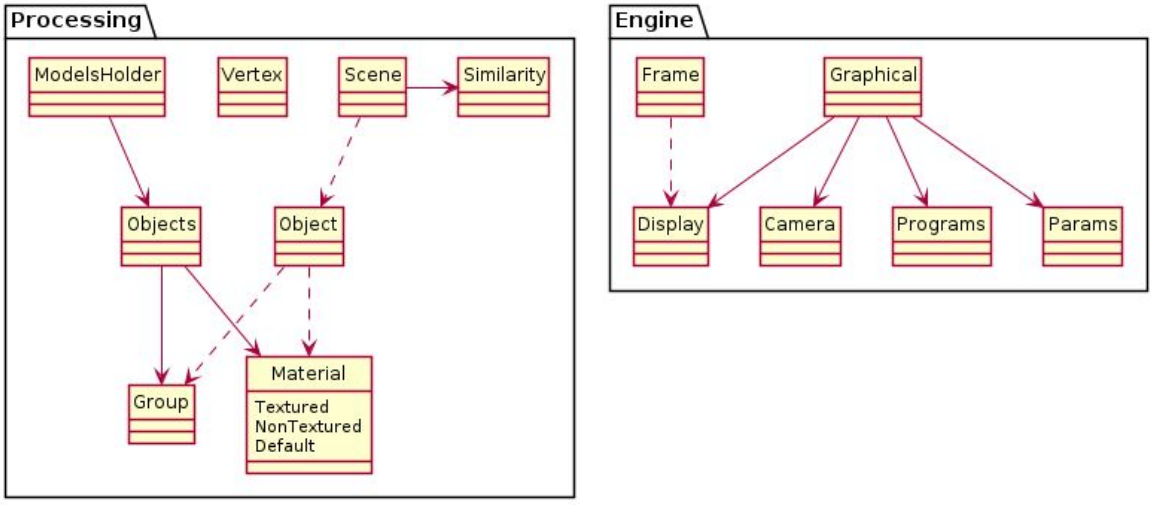
V.3 - Graphics

Cette partie graphique permet de projeter une vue sur la scène 3D en 2D à partir de la caméra, en tenant compte de son orientation et de sa position et ainsi de pouvoir l'afficher à l'écran en respectant les perspectives des objets et leur éloignement à la caméra. La partie graphique permet par ailleurs de gérer la lumière du jeu et les ombrages. Parmis les moteurs graphiques les plus connus, on peut citer OpenGL et Vulkan, ou bien Direct3D. Les objets de l'univers sont stockées sous forme de polyèdres associés à des matériaux.

Pour cette partie, nous utilisons Glium, un wrapper OpenGL en Rust simple d'utilisation (nous ne rédigeons pas directement les instructions OpenGL, Glium le fait à notre place via des fonctions nous facilitant le travail). Nous utilisons le format Wavefront pour nos fichiers 3D. Pour pouvoir les parser, nous nous reposons sur la crate obj.

Les différents objets graphiques (textures, meshes...) sont stockés dans la mémoire graphique. Ainsi, nos structures représentant ces objets ne contiennent que des pointeurs vers cette mémoire graphique (encapsulés proprement dans des objets fournis par Glium).

Notre moteur de rendu comporte plusieures classes, hiérarchisées ainsi:



Les flèches continues représentent un ownership et les discontinues un borrow non-mutable.

Figure 1 : Hiérarchie des classes du moteur graphique

Nous avons séparé le code en deux parties, qui sont deux parties indépendantes : une partie "Processing", qui sert à importer les objets depuis les Wavefront et à créer nos différentes structures de données les représentant, et une partie "Engine", contenant le moteur de rendu en lui-même.

Partie Processing:

Note : Les noms de ces différentes parties sont amenés à changer dans les prochains mois, pour une question de clarté et de cohérence avec les moteurs de jeux existants.

Material :

Représente un matériau. C'est une enum couvrant les différents types de matériaux supportés par notre moteur (directement liés aux différents programmes de shaders).

Le différentes variantes de l'enum contiennent les données représentant le matériaux (pointeur vers la texture, opacité, diffuse color...)

Pour l'instant, nous gérons les objets texturés, les non-texturés (les aplats donc) et un matériau violet flashy, donné aux objets dont les matériaux sont non-supportés ou non-existants. Ce dernier nous permettra de constater les erreurs de texture plus facilement.

Group :

Représente un groupe de voxels uniformes en matériaux. Un objet 3D sera composé de plusieurs groupes. C'est, avec *Material*, la deuxième structure pointant vers la mémoire graphique.

Objects :

Représente tous les objets 3D qu'un fichier Wavefront peut contenir, c'est à dire les *Material* et les *Group*.

ModelsHolder :

Objet central de la partie processing. Il contient tous les *Objects*, et ainsi, a l'ownership de toutes les données. Son nom devra changer à court terme, car il ne contient pas que des modèles, mais aussi des textures, et nous voulons lui donner l'ownership des programmes de shader, ainsi que de toutes les ressources du jeu.

Object :

Objets qu'utilisera le moteur de rendu. C'est un vecteur de couple de références groupes-matériaux. En effet, il n'est pas intéressant de manipuler des objets ayant l'ownership des données car nous n'aurons pas besoin de cet ownership.

Scene :

Contient tous les objets d'une scène, ainsi que toutes les matrices de position des objets.

Similitude :

Matrice de position de l'objet. Nommée d'après le nom de l'objet mathématique auquelle elle correspond.

Vertex :

Les vertex de nos meshes. Cette structure sert à générer nos vertex OpenGL.

Partie Engine:

Graphical :

Structure centrale du moteur de rendu. Possède l'ownership des structures rendant les différents services du moteur.

Display :

Gère la zone d'affichage ainsi que le contexte OpenGL, via la structure *Display* de Glutin.

Camera :

Gère la projection et la caméra. Cette structure sera à court terme scindée en plusieures structures distinctes.

Programs :

Contient les programmes de shaders. Ce rôle sera très probablement en partie délégué à la partie *Processing* à moyen terme.

Params :

Décrit les paramètres de rendu (comme par exemple le depth test). Cela permet par exemple de gérer l'affichage des objets quand ils se superposent, leur transparence ou d'appliquer un filtre de couleur aux objets.

Frame :

Là où on dessine. Elle est générée par *Graphical* et est éliminée après le swap.

Voici le pipeline de notre moteur graphique:

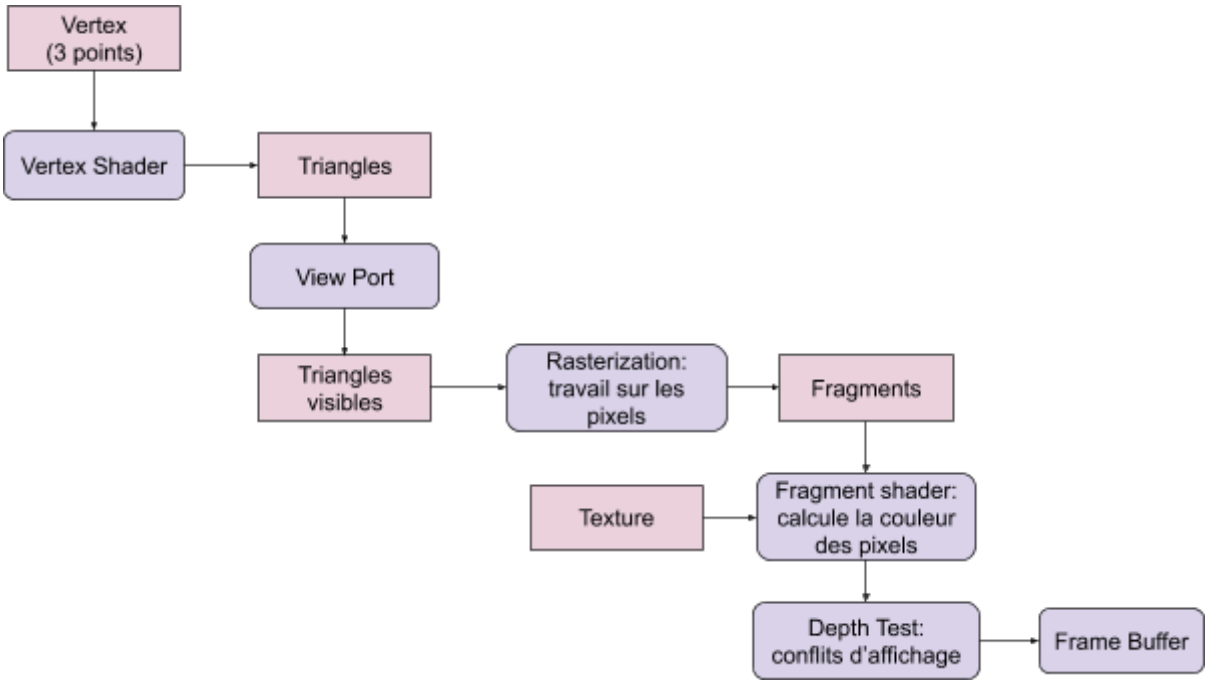


Figure 2 : Pipeline du moteur graphique

L' appellations des structures sont basées sur celles utilisées des logiciels 3D comme par exemple Blender et du standard Wavefront.

Voici quelques notions utilisées pour cette partie. La caméra est définie à partir d'une position et 2 vecteurs, up et orientation.

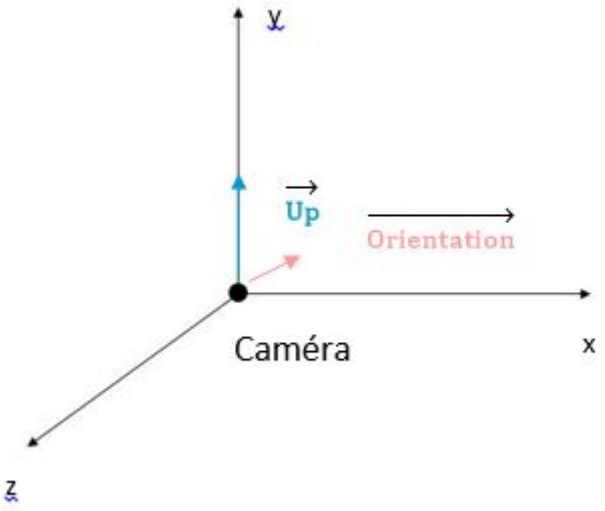


Figure 3 : Un placement de caméra et de ses axes par rapport à la scène

Ce que nous appelons groupe est un ensemble de vertex et d'un *Material*. Un *Object*, possédant plusieurs *Material*, est donc composé de groupes à la texture uniforme. Nous avons choisi, lorsque nous voulons afficher plusieurs objets similaires à des endroits différents, d'instancier tous les groupes à la fois. Cela permet d'importants gains de performance comparé au dessin de tous les groupes semblables uns par uns. Par exemple si nous souhaitons afficher plusieurs portes, composées d'une planche en bois et d'une poignée en métal, nous allons tout d'abord afficher toutes les planches et ensuite toutes les poignées.

Pour le moment, notre code possède une fonction "main" qui crée une scène de démonstration nous permettant de tester nos classes. Nous avons utilisé comme ressources une scène préfaite trouvée sur free3d.com stockées dans répertoire du même nom. Nous y avons ajouté sur la même scène des cubes texturés et rouges (unis). Le "main" gère pour le moment les entrées claviers permettant de se déplacer dans la scène, mais ce n'est normalement pas le rôle de la partie graphique.

Voici ce que nous obtenons :

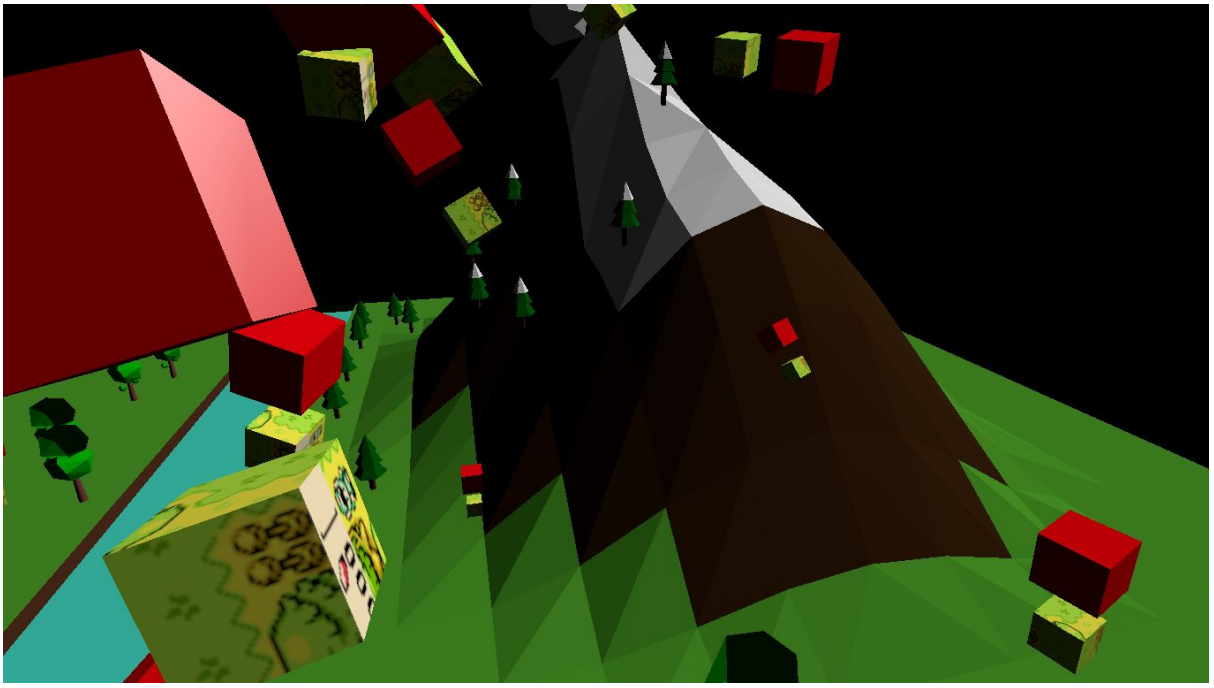


Figure 4 : Démonstration de la partie graphique

V.4 - Physics

Comme expliqué précédemment, la partie physique s'occupe de gérer les forces qui s'appliquent à un objet, les potentielles collisions entre plusieurs objets, faire en sorte qu'un objet soit fixe ou ne puisse pas être touché par le joueur…

Cette partie physique encapsule NPhysics. Nous utiliserons dans notre moteur de jeu vidéo cette partie physique pour s'abstraire de NPhysics et éventuellement pouvoir le remplacer par la suite par d'autres librairies comme Bullet par exemple (si des wrappers récents font leur apparition).

NPhysics nous permet de construire un 'RigidBody' à partir de certaines informations (tout comme Bullet). Cette structure nous permet de simuler la dynamique de solides indéformables ainsi que d'intégrer la trajectoire de solides dont la vélocité est contrôlée par l'utilisateur (ex : plateformes mouvantes). Nous pouvons aussi utiliser NPhysics pour créer des 'Collider'. Ces derniers sont des formes géométriques qui sont responsables de générer les contacts. Ils sont attachés à un 'RigidBody' ou à un corps déformable.

À ce jour la partie physique est constituée de :

Coordinates :

Une structure tout simplement composée de trois float représentant les coordonnées cartésiennes auxquelles se trouve un objet.

MeshType :

Un *MeshType* est une enum qui peut valoir 'Ball', 'Triangle', 'Compound', et bien d'autres choses représentant des formes. En effet, tous les types répertoriés dans cette enum ont non seulement été définis par les soins d'Alexandre, mais sont aussi des formes basiques que l'on peut créer à l'aide de NCollide. Le type *Object* que nous expliquerons plus tard possède un champ comportant un *MeshType* permettant de savoir quelle forme de 'RigidBody' et de 'Collider' nous devons créer avant de placer l'objet dans le monde.

Object :

Il s'agit d'une structure comportant toutes les informations nécessaires à la création d'un 'RigidBody' et un 'Collider' qui lui correspondent. Certaines informations définissent la rigidité de l'objet ainsi que sa capacité à se déplacer et être déplacé ; on peut notamment avoir certains objets qui se déplacent sans que leur inertie ne soit affectée par la gravité ou d'autres forces extérieures. Il n'est pas exclu que divers autres champs soient ajoutés à cette structure.

ObjSet :

Il s'agit d'un vecteur contenant tous les objets pour lesquels il faut construire un 'RigidBody' et un 'Collider'. C'est donc tout simplement un vecteur de *Object*.

Main :

Une fois que nous avons créé un *Object* pour chaque objet que nous voulons intégrer à notre monde (une sphère, un cube...), nous les mettons dans un *ObjSet* que l'on parcours. Parcourir ce vecteur nous permet ensuite de créer le 'RigidBody' et le 'Collider' qui correspond à chacun de ses éléments en prenant en compte si l'objet peut bouger ou non, sa restitution, sa densité et les forces qui s'y appliquent. Nous pouvons ensuite les placer dans le monde. Une fois ces structures créées, elles sont stockées dans un 'ColliderSet' et un 'BodySet'. Ces ensembles sont ensuite utilisés comme paramètres de la fonction 'step' qui permet de simuler le temps. Le monde est mis à jour 60 fois par seconde par défaut.

La fonction 'step' est appelée sur un 'MechanicalWorld' qui peut notamment comporter un vecteur gravité. Il est bon de noter que l'on peut choisir les valeurs du vecteur gravité et donc avoir une faible gravité ou encore une gravité horizontale. Ce 'MechanicalWorld' est responsable des phénomènes physiques tels que la gravité, les forces de contact…

Parmis les paramètres de la fonction 'step' nous retrouvons, non seulement un 'BodySet' ainsi qu'un 'ColliderSet' comme expliqué plus tôt, mais aussi un 'GeometricalWorld'. Ce 'GeometricalWorld' est responsable des requêtes géométrique (détection de collisions, ray-tracing,...). On y retrouve également un 'ForceGeneratorSet' qui répertorie les forces extérieures qui s'appliquent aux objets du monde, ainsi qu'un 'JoinConstraintSet' qui défini les contraintes sur les degrés de liberté de certains objets (ex : en général le sol n'a aucun degré de liberté, une plateforme mouvante ne peut en avoir qu'un...).

V.5 - GUI (Graphical User Interface)

Jouer à un jeu-vidéo, à proprement parler, signifie interagir avec un environnement et ses éléments. Le GUI permet cette interaction via 3 parties. D'une part la gestion des contrôles (clavier / souris) ; d'une autre part la mise en place des éléments de l'interface (menu / barre d'action) ; et pour finir l'affichage de ces zones (aspect / état).

La gestion de contrôle s'effectue à l'aide du module Glutin. En créant une fenêtre et un contexte, des fonctions permettent d'écouter les entrées des périphériques (clavier / souris). En fonction de la touche entrée, une action prédéfinie peut-être effectuer. La combinaison touche / action est écrit en dur dans le code mais l'objectif est de mettre la configuration de touches sous la forme d'un fichier modifiable que l'on charge (ou crée s'il n'existe pas) au lancement du jeu.

L'interface possède 2 types d'éléments, les zones d'interactions (zones actives) et les zones d'informations (zones passives).

- Les zones d'interactions correspondent à des actions avec des prérequis. L'idée est que lorsque l'on détecte une entrée (clavier / souris), une comparaison est faite entre les coordonnées de la souris dans la fenêtre et celles des zones actives, de forme, taille et coordonnées pré-établies. Si la combinaison entrée + position de souris correspond à une zone active alors l'action associée à cette zone est enclenchée.
- Certaines zones sont passives, c'est à dire qu'aucune interactions n'est possibles avec elles, elles ne font que donner des informations comme les points de vie restant, la batterie de la lampe torche, la version du jeu…

L'affichage des zones comprend deux parties : l'aspect des zones et leur état.

- La partie aspect correspond aux chargements et à l'affichage des images à l'emplacement des zones correspondantes. Cela permet de rendre visible au joueur les éléments de l'interface.
- La partie état définit si les cases sont activées ou désactivées, celles qui sont affichées ou masquées. L'interface change entre le menu, les options et la phase de jeu, cette partie décide quelle interface et éléments afficher et activer en fonction de l'état du jeu.

VI - Pistes d'approfondissement

N'étant qu'à la moitié du temps défini pour le projet, il nous reste encore beaucoup de points à perfectionner dans les prochains mois :

VI.1 - Concernant les différentes parties du moteur de jeu

Nous avons par exemple choisis de développer dans un premier temps toutes les parties énoncées précédemment de manière parallèles et complètement séparées. Cela nous a permis d'avancer dans chacune de ces parties pour construire une base solide sans pénaliser l'avancement des autres sur leurs parties. Nous devons donc, pour la suite, mettre en commun les trois parties principales de notre moteur de jeu, à savoir, la partie Graphique, Physique et la GUI.

Nous n'avons pour le moment pas encore pu nous pencher sur le moteur de son. Nous allons probablement faire de l'encapsulation d'une librairie déjà existante mais nous n'en avons pour le moment pas vraiment parlé. Cette partie devrait nous permettre de gérer les différentes ambiances musicales et, plus globalement, sonores du jeu, en s'adaptant au mieux au Gameplay.

N'ayant pas encore mis les différentes parties de notre moteur de jeu en commun, nous n'avons pas pu commencer à élaborer le TestGame pour le moment. Comme énoncé précédemment, nous avons beaucoup de choses à faire pour ce TestGame, qui devra exploiter toutes les fonctionnalités offertes par notre moteur de jeu, et éventuellement en soulever les limites.

VI.2 - Concernant le code et son organisation

A la suite d'une réunion avec Sylvain, celui-ci nous a fait des remarques concernant le nommage de nos entités. En effet, le nom de certaines variables n'était pas clair, et n'étaient pas en cohérence avec les moteurs de jeux existants. Nous avons donc modifié la plupart de ces noms mais il nous reste encore quelques éléments, notamment dans la partie Graphics, à remettre aux normes.

Pour le moment, nous n'avons pas encore structuré l'intégralité du code proprement. Il reste encore beaucoup à faire. Sylvain nous a fait remarquer que certaines classes comme *Graphical* n'étaient pas assez bien structurées. En effet, il est important dans un code de connaître la responsabilité de chaque partie de notre code, et au mieux, de pouvoir lui attribuer

une seule et unique utilité. Par exemple, la partie *Frame* est en charge de construire une image à partir d'objets graphiques. Mais pour le moment, dans notre code, la responsabilité de la partie *Graphical* est plus obscure, car elle gère beaucoup de choses qu'elle ne devrait pas gérer. Nous devons donc réorganiser et scinder le code (éventuellement en différents fichiers) afin de mieux pouvoir dissocier les rôles de chacune des parties le constituant. Cela s'étend également aux différentes parties du moteur de jeu, comme dans la partie *Physics* où, pour l'instant, l'intégralité du code se situe dans le "main".

Pour la partie graphique, nous avons utilisé les outils mathématiques que nous connaissions pour définir par exemple les matrices de rotation et la view_matrix de la caméra. Nous aurions dû utiliser des outils pré-fournis par NAlgebra par exemple et chercher des modes de compatibilité avec les autres modules existants afin de gagner en performance.

VII - Conclusion

Ce projet a été très formateur. Il nous a donné l'opportunité d'apprendre le Rust et de se former entre autre, à l'affichage 3D et à la gestion de la physique. Nous avons pu nous familiariser avec différentes librairies telles que NPhysics ou OpenGL (par le biais de Glium). Le projet nous a également permis d'appréhender les fondements des moteurs de jeux vidéos.

Les prochaines grandes étapes sont de pouvoir assembler toutes les parties du moteur (graphique, physique et GUI) et de pouvoir commencer la construction du jeu vidéo test.

VIII - Sitographie

Site de nphysics [en ligne], nphysics, 2020 [consulté fréquemment].
Disponible sur : https://nphysics.org/

Site de ncollide [en ligne], ncollide, 2020 [consulté fréquemment].
Disponible sur : https://ncollide.org/

Site de nalgebra [en ligne], nalgebra, 2020 [consulté fréquemment].
Disponible sur : https://nalgebra.org/

Documentation de Glium [en ligne], Pierre Krieger [consulté fréquemment].
Disponible sur : https://docs.rs/glium/0.26.0-alpha5/glium/

Documentation de Glutin [en ligne], Pierre Krieger & The Glutin Contributors [consulté fréquemment].
Disponible sur : https://docs.rs/glutin/0.22.1/glutin/

Moteur de jeu [en ligne], Wikipedia en Français, 11 mars 2019 [consulté le 14 Janvier 2020].
Version consultée Disponible sur :
https://fr.wikipedia.org/w/index.php?title=Moteur_de_jeu&oldid=157453228

Game Engine [en ligne], Wikipedia en Anglais, 14 Janvier 2020 [consulté le 14 Janvier 2020].
Version consultée Disponible sur :
https://en.wikipedia.org/w/index.php?title=Game_engine&oldid=935743639

Rust and OpenGL from scratch [en ligne], Nerijus Arlauskas, 8 Février 2018 [consulté le 25 Novembre 2019]. Disponible sur :
https://nercury.github.io/rust/opengl/tutorial/2018/02/08/opengl-in-rust-from-scratch-00-setup.html

Bullet Physics wrapper for the Rust langage [en ligne], not-fl3, 8 Mars 2018 [consulté le 25 Novembre 2019].
Disponible sur : https://github.com/not-fl3/bulletrs

Rust bindings for the Bullet Real-Time physics library [en ligne], elrnv, 11 Août 2017 [consulté le 25 Novembre 2019].
Disponible sur : https://github.com/elrnv/bullet-sys

Learn OpenGL [en ligne], Joey De Vries, 2014 (mis à jour en 2019) [consulté le 14 Octobre 2019]. Disponible sur : https://learnopengl.com/

IX - Retours d’expériences

IX.1- Dans le cadre de l’école

Concernant les heures:

Le temps accordé au projet nous a semblé faible et mal répartis. En effet, nous avons uniquement entre 4 et 6h de projet par semaines, uniquement le lundi après midi. Ce temps est perturbé par les cours que nous avons tout au long de la semaine, nous empêchant de nous consacrer pleinement au projet.

Il existe une semaine dédiée au projet pendant le premier semestre qui nous semble tomber très tôt dans l'année. Durant cette semaine, nous avons principalement dû mettre en place les outils nécessaire au futur bon déroulement du projet et nous n'avons pas eu le temps de beaucoup avancer. Nous avions pourtant bien besoin pour nous plonger dans la création de nos structures.

Toutefois, cela nous permet de nous rendre compte de l'implication et la gestion des heures de travail que nécessite ce type de projet étalé sur plusieurs mois.

Concernant les outils informatiques de l'école :

Les outils informatiques de l'école ne nous ont pas permis d'installer les programmes que nous voulions pour pouvoir développer notre projet.

Concernant la gestion de l’équipe

Ces mois de travail se sont passés sans tension au sein de l’équipe. Nos rôles ont été parfaitement définis dès le début du projet.

Clément, aidant les autres membres de l’équipe et en gérant la structure reliant toute les classes, a permis de superviser la bonne cohérence de toutes les parties du projet.

Du fait que nous partageons la majorité de nos cours, nous pouvions échanger à propos du projet tout au long de la semaine. Nous ne faisions de ce fait pas particulièrement de grandes réunions entre nous.

IX.2- A titre personnel

Alexandre :

Après un lancement assez difficile, le projet est sur de bons rails et me permet de m'investir dans quelque chose que j'aime faire. En effet, les compétences que je peux développer via ce projet sont nombreuses et le voir avancer est satisfaisant.

J'ai déjà appris les bases du langage Rust ainsi que quelques notions de moteur de jeux vidéos. De plus, la communication nous est indispensable pour se coordonner et avancer le plus vite et bien possible. Nous nous sommes réparti les tâches de façon ordonnée et devons tous garder un oeil sur l'avancement des autres car chaque partie est liée aux autres.

En travaillant sur la partie physique j'ai non seulement appris à me servir de NPhysics, mais j'ai aussi appris à avancer de plus en plus vite grâce à mes erreurs et à justifier des choix (comme celui de NPhysics).

C'est pour moi une occasion d'apprendre un langage que je n'aborderai pas en cours mais qui n'en est pas moins important, du fait de la communauté active qui gravite autour de celui-ci, ainsi que de réaliser un projet d'assez grande envergure avant d'entrer dans le monde du travail et, par conséquent, développer des compétences indispensables lors de la réalisation de ce type de projet.

Amaury :

Honnêtement, c'est compliqué, mais c'est ainsi qu'on apprend. Ainsi, par rapport à un TP ou un projet d'unité, il a de nombreux intérêts je trouve. Déjà, il plonge dans un bain bien plus proche de la réalité. De plus, il nous force à faire des recherches pour trouver des solutions, un vrai travail d'ingénieur.

Le choix de nous aventurer dans le langage Rust est du coup très enrichissant. Déjà, on apprend un nouveau langage, et ensuite, il nous force à chercher des solutions par nous-même, car Nabil ne peut pas nous aider sur ce langage.

Cependant, les moteurs de jeux-vidéos sont de base des gros projets très complexes. Notre aventure aurait peut-être été plus simple si nous avions choisi le C++ ? Et nous aurait garanti peut-être un résultat plus rapidement ? Mais qu'importe, nous défendons le choix de Rust, et c'est ce qui fait que même si la forme de projet n'est pas super sexy, il y a beaucoup de fond derrière. Beaucoup de recherches. Beaucoup de choses apprises. Et c'est chouette.

Du coup, c'est aussi décourageant. Clément connaît Rust, mais nous autres, nous sommes complètement en terre inconnue. Et ça peut-être décourageant. J'aurais préféré aller à petit pas, plutôt que de prendre d'un seul coup l'apprentissage de Rust, OpenGL, et pleins d'autres joyeusetés.

Mais c'est cool. Et démotivant en même temps.

Et pour finir, tout ceci me force aussi à être plus tolérant sur la non-maniaquerie de mes camarades. Sérieusement, les noms de commits, c'est archaïque je trouve. Mon rêve, c'est de tous les renommer ! Et c'est faisable d'ailleurs, avec le "git rebase".

Barbara :

Ce projet m'a permis de m'initier à de nombreuses compétences. Il m'a permis d'appréhender ce qu'est un moteur de jeu vidéo et ses principaux composants.

Avec Maÿlis nous avons exclusivement travaillé sur la partie du rendu graphique du moteur de jeu vidéo. J'ai beaucoup aimé travailler sur le deuxième tutoriel fourni. Nous y avons travaillé environ 2 semaines et demies. Bien que nous n'avons pas utilisé cette version d'OpenGL pour la création du moteur, le fait d'utiliser des fonction de plus bas niveau pour créer une fenêtre et y afficher quelque chose a été très formateur. Travailler sur ce projet m'a permis d'apprendre à utiliser la librairie OpenGL et d'étudier l'affichage 3D.

Je suis également très contente d'avoir pu apprendre un nouveau langage de programmation, Rust. C'est un langage qui m'a paru étonnant au début. Mais je n'ai pas rencontré de problème à me familiariser avec lui grâce à Clément qui a sû répondre à toutes mes questions.

Clément :

Cela faisait déjà plusieurs mois que j'étudiais d'autres moteurs de jeux en Rust, ce projet a pour moi été une opportunité d'aller plus loin. J'ai fais un peu de tout, aidé tout le monde, mais la majorité de mon travail a été l'architecture du code et le début de la partie graphique, ainsi que la totalité de la partie ECS (que nous avons finalement remplacée par SPECS par pragmatisme).

Je connaissais déjà assez bien le Rust, mais j'ai tout de même énormément appris, que ce soit en Rust ou en OpenGL. J'adore ce projet, et je pense que nous avançons très vite aux vues du peu d'heures dont nous disposons, de la non-formation à la programmation en tant que tel que nous avons (Zuul est trop guidé pour compter vraiment), et de la difficulté de la tâche.

J'aurais néanmoins aimé pouvoir compter sur le matériel de l'école pour pouvoir travailler, mais nous n'avons a priori pas les droits nécessaire pour faire de l'informatique (dans une école d'ingénieurs orienté informatique, c'est le comble). Les droits root ne sont pas en option.

Pour en revenir au projet en lui-même, j'ai espoir que notre moteur de jeu puisse être viable pour des usages (relativement) modestes, et je compte m'en servir dans le futur pour concevoir des jeux ou autres simulations.

Maÿlis :

Ce projet m'a dans un premier temps semblé trop ambitieux pour les compétence que j'avais. Nous avons fait certains choix de développement, comme le Rust et l'OpenGL, qui m'ont sortie de ma zone de confort mais qui m'ont permis de découvrir de nouvelles choses dans le domaine du développement. J'avais peur au début mais je suis certaine que cette expérience sera bénéfique pour nos futurs projets/métiers, dans le sens où nous aurons besoin de nous confronter à de nouveaux langages/outils, et d'apprendre à les utiliser seuls, et sans cours au préalable.

Clément a énormément travaillé, et je regrette de ne pas avoir plus participé au projet, par manque de temps, et parce qu'il est difficile de se replonger dans le code lorsqu'on a que quelques heures dédiées au projet, une fois par semaine. Cela nous a vallu de grosses pertes de temps, notamment au début du projet.

J'ai trouvé les réunions avec Nabil et Sylvain très enrichissantes, et très intéressantes. Nous avions notre place et nous pouvions discuter les choix que nous faisions sans que qui que ce soit ne nous impose quelque chose. Je suis vraiment très reconnaissante de la liberté qu'ils nous ont laissé prendre et de leur bienveillance, en général, et concernant nos choix. Cela nous a vraiment permis d'expérimenter des choses que nous n'aurions pas eu l'occasion de faire en cours.

Nicolas :

Le début du projet était source de stress mais aussi d'inspiration : Développer un moteur de jeu pour Ubisoft. Cela m'a paru un peu effrayant au début car je m'attend à ce que Ubisoft soit très exigeant envers le produit final, mais après avoir rencontré Sylvain j'ai compris que nous participions à une expérience pour essayer de nouvelles façons de développer des moteurs de jeux-vidéo et cette idée m'a beaucoup plu.

L'étude d'un moteur de jeu, de son architecture fut la première partie du projet et cela m'a permis de comprendre la complexité de la tâche sans pour autant me démotiver.

Le développement sous Rust fut une découverte, je n'avais jamais codé en Rust, d'ailleurs je n'en avais jamais entendu parler, mais l'objectif du projet était d'expérimenter en cela je trouve que nous avons fait un très bon choix.

Ce projet est loin d'être facile mais les connaissances et compétences que j'en tire me seront très utile pour mes futures travaux. Au jour d'aujourd'hui j'apprécie participer à ce projet.

X - Table des Figures

XI - Annexe

Lien de notre GitHub :
https://github.com/SailorSparkle/moteur_jeu_video/