ESIEE PARIS

Année 2019-2020
Semestres 1&2

Rapport final
de Projet E4

Construction d'un moteur de jeu vidéo

Projet réalisé par :

CHOMICKI Clément • BARUZY Amaury • PARÉ Barbara
LEBLON Alexandre • MONTANI Maÿlis
ZHOU Sebastien • 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 au 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.

Sommaire

I - Introduction **4**

II - Objectifs (Sujet Original) **5**

III - Etat de l'art **6**

III.1 - Qu'est ce qu'un moteur de jeu ? 6

III.2 - Historique 7

IV - Décisions préalables **8**

IV.1 - Choix du langage 8

IV.2 - Quelques définitions relatives à Rust 9

IV.3 - Choix de la plateforme de développement 9

IV.4 - Choix des librairies 10

IV.5 - Spécificités de notre jeu test 11

IV.6 - Mise en commun du travail et des rapports de réunions 12

V - Déroulement du projet **13**

V.1 - Répartition des tâches 13

V.2 - ECS 14

V.3 - Graphics 14

V.4 - Physics 20

V.5 - GUI (Graphical User Interface) 26

V.6 - Son 28

V.7 - Partie Centrale du Moteur de jeu 30

V.8 - Jeu test 31

VI - Pistes d'approfondissement **34**

VII - Conclusion **35**

VIII - Sitographie **36**

IX - Retours d'expériences **37**

IX.1 - Dans le cadre de l'école 37

IX.2 - A titre personnel 38

X - Table des Figures **42**

XI - Annexe **42**

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 7 membres : Clément CHOMICKI, Maÿlis MONTANI, Alexandre LEBLON, Amaury BARUZY, Barbara PARÉ, Sebastien ZHOU et Aymeric SALLET.

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

- Clément CHOMICKI, Alexandre LEBLON, Amaury BARUZY, Maÿlis MONTANI et Barbara PARÉ sont des membres du projet constants qui auront été présents pendant toute la durée du projet.
- Nicolas GOISLARD était présent lors du début du projet mais il a souhaité quitter l'ESIEE au cours de l'année.
- Sebastien ZHOU et Aymeric SALLET sont des membres du projet qui sont partis à l'étranger au premier semestre de E4, et qui ont rejoint le projet 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** (Entity Component System) : 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 **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 **GUI** (et **gestion des entrées**) permettant d'afficher des menus, des barres de vie, d'accéder aux options du jeu telles que modifier le volume du son… Cette partie permet également de gérer les entrées (clavier, souris, manette, joystick…) et permet au joueur d'interagir avec le jeu.
- 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. Elles 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 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 évolués 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 questions 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 performances.

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 (d'après 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'aborderions pas à l'école nous a tous motivé. 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 langages 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'informations du code, ce qui les rend capable de plus d'optimisations (compensant en partie le coût de l'abstraction) et d'exprimer des messages plus clairs au développeur en cas d'erreurs. La clareté du compilateur est un point non négligeable pour l'initiation au langage, l'impactant très fortement.

Nous avons ensuite parlé avec Nabil et Sylvain pour connaître leur avis sur le fait d'implémenter le moteur de jeu 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 choisissions 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 garantit 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 n'avons donc pas été biaisés par des expériences/connaissances passées, et cela nous assure que notre moteur de jeu est le fruit d'une approche originale.

Le Rust est très semblable au C++ sur énormément de points, néanmoins certains points changent du tout au tout la programmation (notamment dans le cas d'un jeu vidéo):

- Les pointeurs sont utilisés beaucoup plus parcimonieusement qu'en C++, et ce grâce à la souplesse des références de Rust. Quand on doit utiliser des pointeurs en Rust, on a à notre disposition des équivalents des smarts pointeurs de C++ tout à fait respectables.
- Parce que le compilateur sait à l'avance quelle structure pourra être échangée entre les threads, le multithreading est extrêmement aisé (sans parler de la crate rayon qui permet de paralléliser un itérateur en seulement une méthode).
- Parce que le langage permet énormément d'abstraction avec peu ou pas de coût à l'exécution, les librairies sont déjà très abstraites, ce qui rend plus ardu des modifications de bas niveau.
- Le compilateur étant tellement agréable, il est plus facile de produire du code Rust que C++, ce qui aide pour ce genre de projet pharaonesque.

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 - Quelques définitions relatives à Rust

Crate : C'est le nom que l'on donne aux librairies rust, développées et gérées à l'aide de Cargo.

Cargo : C'est le gestionnaire de projet rust. Il permet entre autres de compiler le projet (en appelant rustc, le compilateur rust), de lancer le projet, de générer la doc, et de publier la crate dans les repos officiels.

IV.3 - Choix de la plateforme de développement

Nous avons, en parallèle de notre apprentissage de Rust, 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.

Clément a développé sous emacs ; Alexandre, Amaury, Maÿlis et Sébastien, principalement sous Visual Studio Code ; Barbara sous gedit ; et Aymeric utilise Mac.

IV.4 - 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. Glium, accompagné de glutin, une librairie réimportée par glium permettant la création de fenêtre et la gestion du hardware, nous permet facilement de créer les divers vbos et autres buffers graphiques, via des structures rust amenant toutes les garanties de gestion de la mémoire du langage. Il est également possible d'afficher des choses à l'écran en seulement quelques lignes, mais beaucoup de méthodes et structures sont aussi prodiguées afin de profiter de toutes les possibilitées d'OpenGl. Une telle souplesse d'utilisation est tout ce qu'il nous fallait.

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 partie son, nous avons choisi la librairie Ears. Basée sur OpenAl et libsndfile, elle est compatible avec les fichiers son supportés par libsndfile dont notamment les formats WAV, FLAC ou RAW (voir toutes les extensions supportées ici). C'est une librairie qui permet de jouer des sons le plus simplement possible et qui est plus intuitive que rg3d_sound.

Pour la partie GUI nous avons essayé de la créer de toute pièce sans utiliser de librairies particulières. Après quelques temps, Nabil nous a fait remarquer qu'aux vues du temps qu'il nous restait, il fallait peut être utiliser une librairie existante, quitte à la modifier plus tard. Nous avons alors cherché quelles librairies seraient compatibles mais il n'y en avait que très peu de disponibles en rust utilisant Glium. Nous avons finalement opté pour l'une des librairies les plus documentées utilisant glium et winit : Imgui-rs de Gekkio.

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

IV.5 - 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 test. Cela allait nous permettre d'orienter le développement de notre moteur de jeux vidéos pour avoir des objectifs clairs de ce qu'il nous fallait développer.

Sous le conseil de Nabil, nous avons donc choisi 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 principal 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.

Nous n'avons finalement pu réaliser que quelques unes de ces idées :

- Notre univers est en 3D.
- Notre jeu test a une ambiance western.
- Nous avons une musique d'ambiance et d'autres musiques disponibles.
- Nous pouvons nous déplacer dans les airs.

En effet, construire un jeu complet aurait pu faire l'objet d'un autre projet, mais nous avons avant tout souhaité réaliser le moteur de jeu. Nous n'avons pas pu y intégrer toutes les fonctionnalités de notre moteur de jeu par manque de temps.

IV.6 - 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 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 avons tenté de les rendre plus clairs et précis par la suite.

A l'issue de chaque réunion, nous avons publié 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 mais aussi la ligne de commande permettant de lancer notre jeu test.

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) ;

La répartition des rôles à changé lorsque Sébastien ZHOU et Aymeric SALLET ont rejoint le projet et que Nicolas GOISLARD nous a quitté :

- Clément, toujours chef de projet, s'est occupé de la partie ECS ainsi que de la partie graphique avec l'aide de Maÿlis et Barbara. Il a aussi fait la mise en commun des différentes parties ainsi que la résolution des bugs les plus coriaces.
- Maÿlis a aidé Clément sur la partie graphique ainsi qu'Alexandre sur la partie physique. Elle a également réalisé le jeu test à l'aide de Clément.
- Alexandre a travaillé sur la partie physique ainsi que sur la mise en commun avec Clément. Il a également documenté la plupart de la partie physique, aidé par Maÿlis.
- Barbara a aidé Clément sur la partie graphique et a travaillé sur la partie son qu'elle a développé et implémenté dans l'exemple du projet.
- Sébastien ainsi qu'Aymeric ont travaillé sur la partie GUI du projet.
- Amaury s'est occupé de la gestion du repos GitHub.

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").

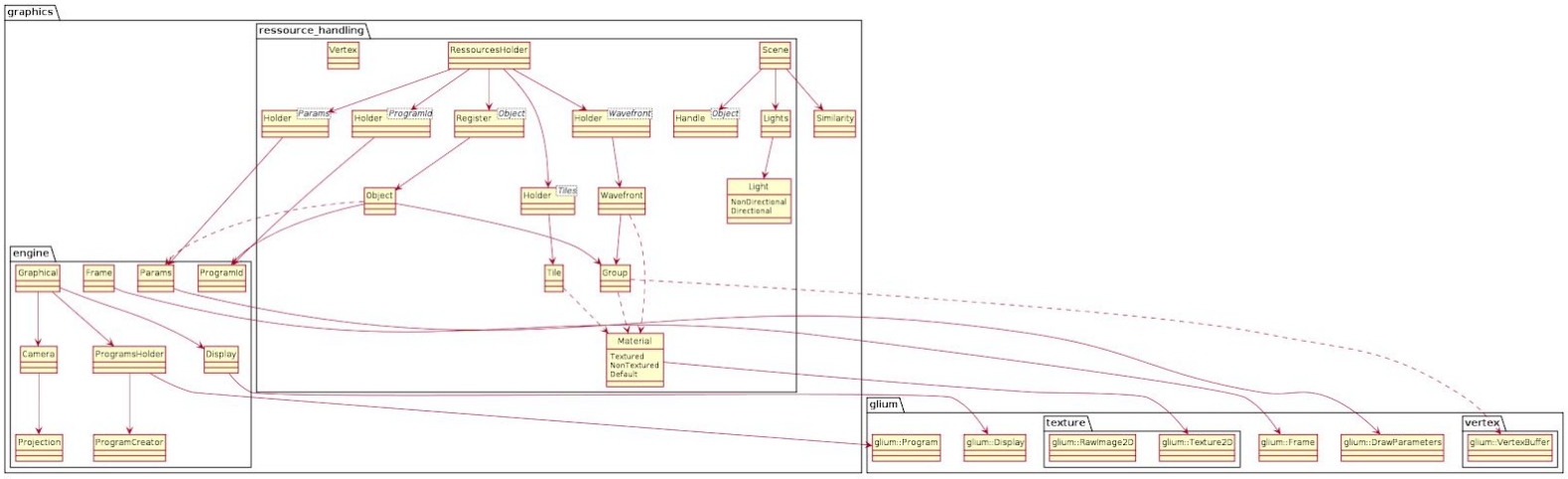
V.3 - Graphics

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. Parmis les moteurs graphiques les plus connus, on peut citer OpenGL et Vulkan, ou bien Direct3D. Les objets de l'univers sont stockés 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 plusieurs 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 indépendantes : une partie “ressources handling”, 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 ressource handling:

Vertex :

Structure désignant les Vertex. Ils possèdent notamment une position, une normale et une texture. Leur format est directement compatible avec le format attendu par le GPU.

RessourcesHolder :

Structure contenant toutes les références vers les différentes ressources, qu’il s’agisse de Wavefront, Programs, Tiles, Params, Object_register, ou Sounds_datas. Cette structure permet notamment de faire le lien entre les Handle et les Registres à l’aide de ses différentes fonctions.

Holder :

Un Holder est une structure qui map des ressources à leur nom. Il s’agit basiquement d’une HashMap sur laquelle nous pouvons implémenter des méthodes spécifiques en fonction du type de ressources.

Holder<ProgramId> :

Il s'agit simplement d'un Holder contenant une version de chaque ProgramId connus, permettant de les utiliser pour créer des objets affichables. Ils amènent l'information des shaders à utiliser.

Holder<Params> :

Simplement un Holder contenant des Params, amenant les différents paramètres de rendus (comme le depth test, le blending, si oui ou non on demande du backface culling…).

Holder<Wavefront>:

Il s'agit d'un Holder de Wavefront. On peut appeler sur ce type d'objets les méthodes load_wavefront() permettant d'importer un fichier wavefront notamment à partir de son nom ; get_object() récupère les vertex et les material d'un objet 3D correspondant au nom de l'objet donné ; get_whole_content() faisant la même chose que get_object() mais en renvoyant l'union de tous les objets du fichier ; et unload() permettant de retirer un objet de la mémoire graphique.

Holder<Tiles> :

Comme le holder de wavefronts, permet de créer des objets affichables, mais contient des Tiles plutôt que des objets 3D.

Handle :

Structure désignant les données stockées dans un Register. Elle contient un u64 désignant la génération de l'objet associé, son index dans le registre et un phantom pour ajouter un type paramètre. Elle est notamment utilisée à l'insertion d'un objet dans un registre pour avoir un genre de référence vers l'objet au lieu de l'objet lui-même. En somme, on s'en sert pour représenter un objet du Register hors de celui-ci, et notamment hors de son thread.

Register :

Stocke n'importe quel type d'objets de même type. Elle contient un nombre d'objets, un vecteur d'objets, un vecteur de booléens pour déterminer si l'objet est est toujours utile ou s'il a été "supprimé" et un vecteur des indices libres de registre. Cela permet l'insertion, la suppression et l'accès aléatoire (via les Handles) très faciles et en temps constant. Cela permet surtout de minimiser les élargissements inutiles du vecteur, en réutilisant l'espace dont nous avons plus besoin.

Scene :

Structure représentant une scène à afficher. Contient des objets, des lumières et une caméra. Elle ne contient évidemment que des pointeurs vers les objets et pas les données elles-mêmes. On peut notamment ajouter des objets ou bien des lumières à la scène, demander une mise à jour de l'aspect ratio de la caméra et demander à la partie graphique de faire un rendu.

Object :

Structure représentant un objet 3D. Il est composé de Group de vertex qui possèdent les mêmes Material. Cette structure contient notamment un champ data et un champ params correspondant aux options de rendering. Elle ne contient pas de données mais uniquement des shared pointer vers les données.

Lights :

Structure contenant un certain nombre de lumières et leur propriétés telles que leur type, leur intensité, leur couleur, leur position et leur direction, le tout rangé dans des uniform buffers.

Light :

Enum contenant deux types de lumières : Directionnelles et non directionnelles. La variante “directionnelle” peut signifier un spot (si l’entité de l’ECS correspondant a une position), ou bien juste une lumière directionnelle constante et globale. La variante “non directionnelle” peut signifier que l’on a un point lumineux ou une ambiance.

Group :

Représente un groupe de voxels uniformes en matériaux. Un objet 3D sera composé de plusieurs groupes. Cette structure contient des vertex ainsi que le matériau associé. La structure ne contient pas directement les données mais contient un shared pointer vers les données.

Tile :

Structure désignant un rectangle 2D contenant une image. Elle contient la texture, c’est à dire le Material et les dimensions de l’image.

Wavefront :

Structure représentant un set d’objets 3D et leur matériaux associés. Les informations contenues dans cette structure représentent le contenu des fichiers .obj et .mtl des Wavefronts.

Material :

Enum représentant le matériel dans le sens Wavefront. Il peut s’agir d’un matériel texturé, d’un matériel non texturé, ou d’un matériel par défaut. Chacune des possibilités possède des champs spécifiques.

Similarity :

Une similarité (une matrice 4x4 représentant un redimensionnement, une rotation et une translation). Utile quand les matrices de nalgebra ne conviennent pas lors des communications avec la carte graphique. Elles servent à représenter les position des différents objets dans la scène.

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 :

Structure gérant la caméra. Elle possède diverses fonctions permettant de déterminer le vecteur forward, les angles, les rotations, l'aspect ratio, les positions, les translations, mais aussi d'obtenir la view matrix et la perspective matrix.

ProgramCreator :

Gère la création des ProgramIds.

ProgramId :

Id d'un programme enregistré dans ProgramsHolder.

ProgramsHolder :

Contient les programmes de shaders. Possède une méthode pour charger tous les shaders d'un répertoire et une pour en ajouter manuellement.

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. Elle repose sur la structure Frame de Glium. On peut dessiner dessus des Group (possédant le même Material) et donc des objets.

Projection :

Définie par un znear et zfar. La projection est généralement utilisée pour les rendus graphiques et permet de délimiter la fenêtre visible par la caméra située entre les deux "écrans" znear et zfar.

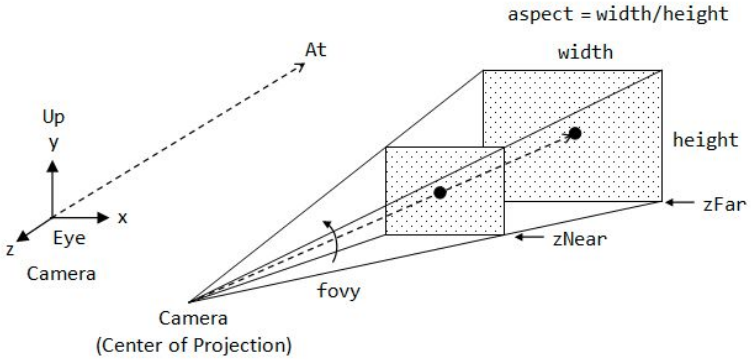


Figure 2 : Projection rendu graphique

Voici le pipeline de notre moteur graphique:

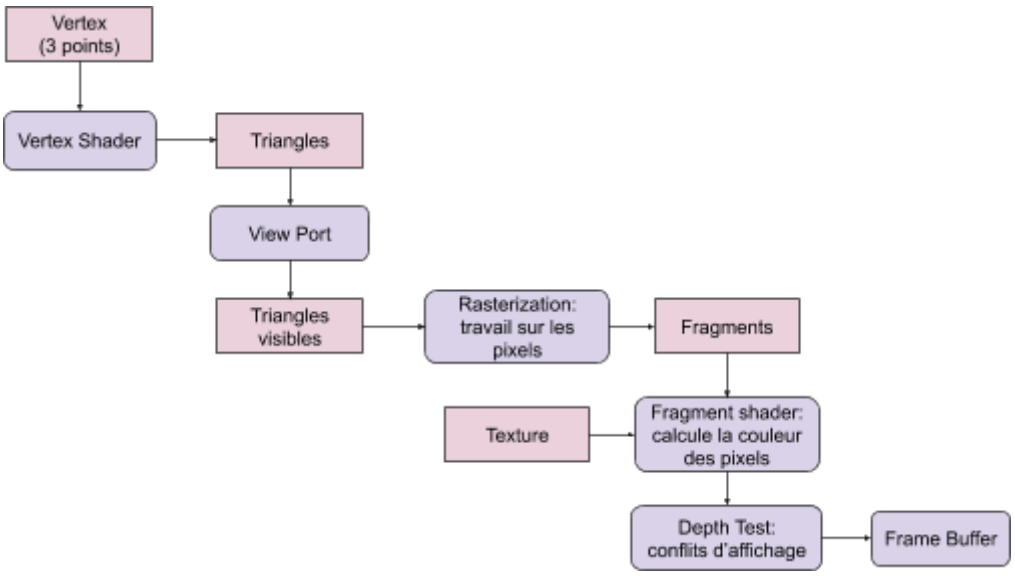


Figure 3 : Pipeline du moteur graphique

L'appellation des structures est basée sur celles utilisées dans des logiciels 3D tel que Blender, ou encore sur le standard Wavefront.

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

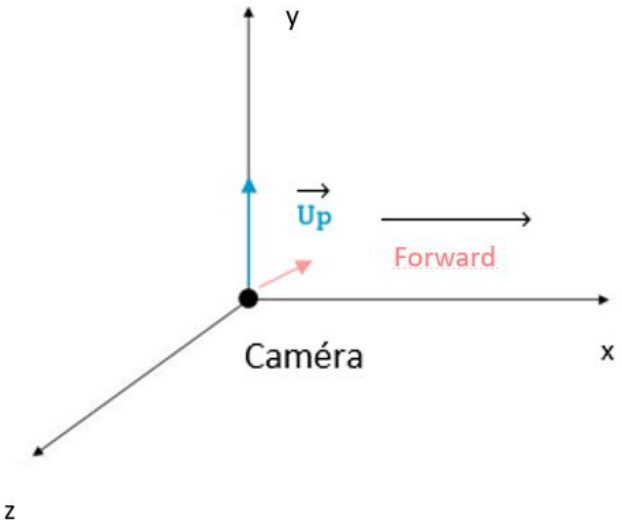


Figure 4 : Un placement de caméra et de ses axes par rapport à la scène

Ce que nous appelons Group 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.

Dans un premier temps, notre code possédait une fonction "main" qui créait une scène de démonstration nous permettant de tester nos classes. Nous avons utilisé comme ressources une scène préfaite trouvée sur free3d.com stockée dans répertoire du même nom. Nous y avons ajouté sur la même scène des cubes texturés et rouges (unis). Le "main" gérait 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 obtenions :

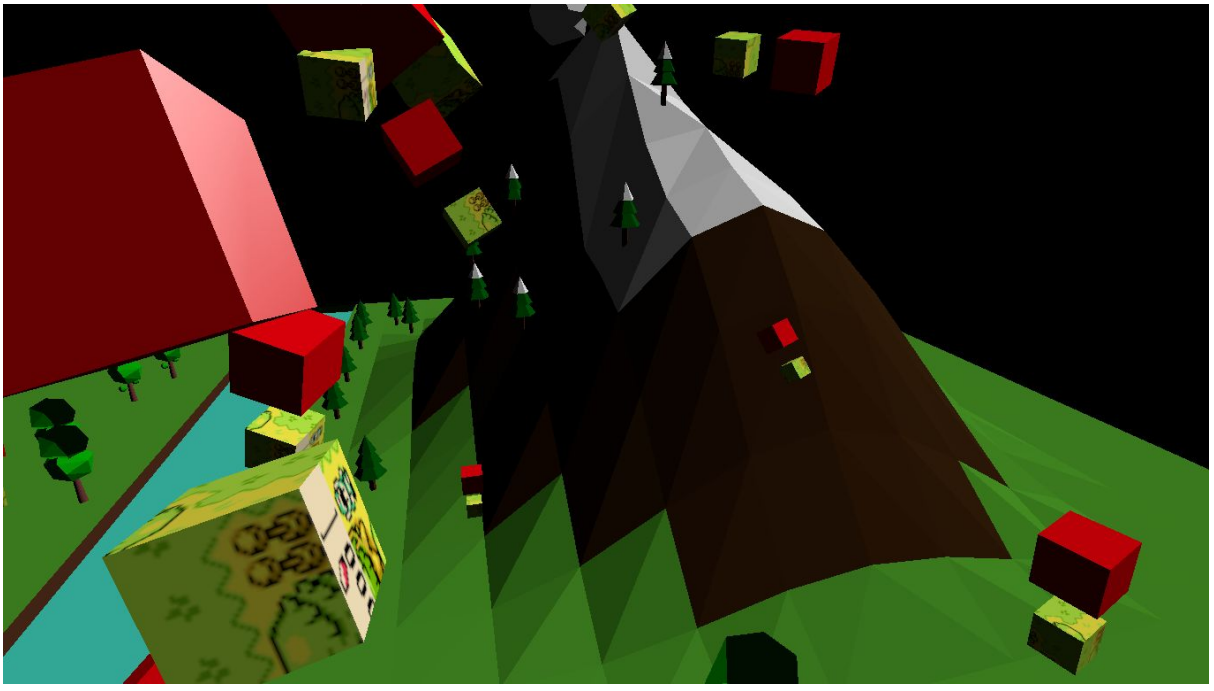


Figure 5 : Démonstration de la partie graphique

Notre partie graphique s'est maintenant déléguée de cette tâche et cela est maintenant de la responsabilité de notre exemple (cf Jeu Test).

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 jeux vidéos 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.

Les structures que contenait précédemment la partie physique :

Coordinates :

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

MeshType :

C'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 qui est une crate fonctionnant avec nphysics. Le type 'Object' que nous expliquerons plus tard possède un champ comportant un 'MeshType' qui permet 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' de ncollide 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.

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 (fonction) :

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...).

Les structures que contient la partie physique à ce jour :

ShapeType (anciennement MeshType) dont seul le nom a changé.

RbData qui contient autant de champs que de paramètres nécessaires à la création d'un 'RigidBody' aussi customisé que possible.

ColData qui contient autant de champs que de paramètres nécessaires à la création d'un 'Collider' aussi customisé que possible.

PhysicObject (anciennement Object) dont la définition n'a pas changée par rapport à celle de 'Object' mais le contenu si. En effet, au lieu de ne contenir que le minimum de données nécessaires à la création d'un 'RigidBody' et 'Collider' cette structure comporte maintenant 3 champs qui permettent de créer un 'RigidBody' et un 'Collider' exactement comme on le désire :

- shape (de type ShapeType) qui contient la forme que prendront le 'RigidBody' et le 'Collider'
- rbdata (de type RbData) qui contient les données nécessaires à la création d'un 'RigidBody'
- coldata (de type ColData) qui contient les données nécessaires à la création d'un 'Collider'

Physics qui contient tous les éléments nécessaires pour faire "tourner" le monde physique. Cela inclut mais n'est pas limité à :

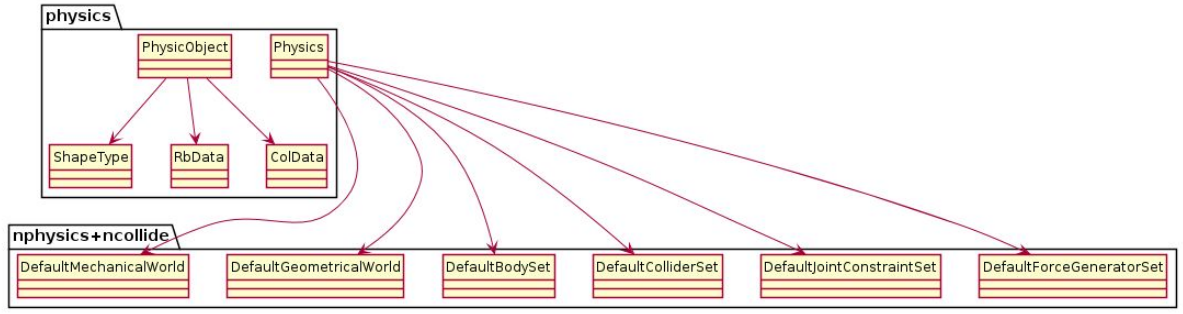
- les forces qui s'appliquent aux objets
- les 'RigidBody' et 'Collider' présents dans le monde physique
- les contraintes qui s'appliquent à certains objets ou à tous les objets

La partie physique ne possède bien évidemment plus de fonction main car c'est maintenant une crate qui a pour but d'être utilisée dans l'exemple, puis par l'utilisateur.

La structure ObjSet a disparu car la fonction qui permet de créer les 'RigidBody' et 'Collider' ne prend plus un vecteur de 'PhysicObject' mais un seul 'PhysicObject'. Cela est bien plus pratique car cela permet de choisir les objets qui auront de la physique et ceux qui n'en auront pas.

La structure 'Coordinates' n'est plus d'aucune utilité car la structure RbData possède un champ contenant les coordonnées de l'objet.

Voici le diagramme des structures de la partie physique :



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

Figure 6 : Hiérarchie des classes de la partie Physique

Processus de création du monde physique et des objets qu'il contient :

Le monde physique est créé à l'instanciation d'un objet 'Physics' à l'aide de la fonction 'Physics::default' dans l'exemple.

L'utilisateur a ensuite seulement besoin de créer les 'RigidBody' et 'Collider' de chaque objet auquel nous voulons appliquer de la physique. Le processus de création est le suivant :

- L'utilisateur fournit un 'Object' de la crate graphics du moteur de jeu à la fonction make_trimesh qui va créer et retourner une trimesh correspondant à la forme de l'objet.

- L'utilisateur doit ensuite appeler sur la trimesh obtenue une des fonctions 'make_static', 'make_kinematic', 'make_dynamic' en fonction de s'il désire ou non que l'objet soit soumis aux forces qui régissent le monde physique. La fonction 'make_dynamic_constrained' permet quant à elle de créer un objet au comportement dynamique mais qui sera limité dans ses déplacements (l'utilisateur peut choisir de faire en sorte que l'objet ne puisse que pivoter selon l'axe vertical par exemple).
- La fonction que l'utilisateur aura choisi d'appeler aura retourné un 'PhysicObject' sur lequel la fonction 'build_rigdb_col' devra être appelée pour créer le 'RigidBody' et le 'Collider' lui correspondant.
- Le handle retourné par la fonction précédemment appelée permettra ensuite à l'utilisateur de créer un 'PhysicComponent' qui servira lui-même à créer une 'Entity' de specs, mais cela n'est plus du ressort de la partie physique.

L'utilisateur peut cependant ajouter des contraintes à certains objets ou des forces autres que la gravité en modifiant les champs 'joint_constraints' et 'force_generators' de l'objet 'Physics' précédemment créé.

Fonctions clés :

make_trimesh :

C'est la fonction qui permet à l'utilisateur de créer une 'TriMesh' qui correspond parfaitement à la forme de l'objet qu'il passe en paramètre. Il est indispensable de créer une 'TriMesh' pour certains objets à la forme complexe pour éviter certains problèmes (une bouteille qui tient facilement à l'envers car elle a un 'RigidBody' et' Collider' cylindrique par exemple).

Son code est le suivant :

```rust
/// Creates the ShapeType::TriMesh associated to the object and return it.
pub fn make_trimesh(object: &Object) -> ShapeType
{
    let all_vertex = object.data.iter()
    .map(|(group, _)|
    {
        (*group.vertexes)
        .read()
        .iter()
        .flatten()
        .map(|vertex: &Vertex|
        {
            Point3::new(vertex.position[0], vertex.position[1], vertex.position[2])
        })
        .collect::<Vec<_>>()
    }
    ).flatten()
    .collect::<Vec<_>>() ;

    let indices = (0..(all_vertex.len()/3)).map(|i| {Point3::new(3*i, 3*i+1, 3*i+2)}).collect::<Vec<_>>() ;

    ShapeType::TriMesh(TriMesh::new(all_vertex, indices, None))  // We might need the scale
}
```

Figure 7 : Code de make_trimesh()

Dans un premier temps, cette fonction récupère dans le VertexBuffer les vertex de l'objet passé en paramètre par groupes de vertex possédant la même texture. Elle concatène ensuite tous ces vertex pour obtenir l'intégralité des vertex de l'objet, qu'elle utilise ensuite pour créer des TriMesh en rassemblant par 3 les vertex du vecteur.

make_object :

C'est la fonction qui est appelée par les fonctions 'make_static', 'make_kinematic', 'make_dynamic', et 'make_dynamic_constrained'. Elle permet de créer un 'PhysicObject' à partir de certaines données telles que : la position de l'objet, sa rotation, sa taille (scale) ainsi que sa capacité à subir la gravité (true ou false). On l'appelle sur le PhysicObject directement et elle permet d'associer à un ShapeType (de PhysicObject) les données servant à créer le RigidBody et le Collider. Nous ne l'avons implémenté que pour les TriMesh mais il est évidemment possible de le faire pour d'autres formes (disponibles dans le dossier shape de physic).

Son code est le suivant :

```rust
/// Constructs a PhysicObject (implemented only for TriMesh but other shapes are disponible).
pub fn make_object(
&self,
translation: Vector3<f32>,
rotation: Vector3<f32>,
_scale: f32, // Unused.
gravity: bool,
stat: BodyStatus,
kinematic_rot : Option<Vector3<bool>>,
kinematic_trans : Option<Vector3<bool>>) -> PhysicObject
{
    match self
    {
        ShapeType::TriMesh(trimesh) => {

            let shape = ShapeType::TriMesh(trimesh.clone());

            let center: Point3<f32> = trimesh
            .points.iter()
            .fold(Point3::new(0., 0., 0.), |sum, p| sum+p.coords) / (trimesh.points.len() as f32);

            let rb_data = RbData::new(
                translation,                            // translation.
                rotation,                               // rotation.
                gravity,                                // gravity_enabled.
                stat,                                   // bodystatus.
                Vector3::new(0.0, 0.0, 0.0),            // linear_velocity.
                Vector3::new(0.0, 0.0, 0.0),            // angular_velocity.
                0.8,                                    // linear_damping.
                1.8,                                    // angular_damping.
                INFINITY,                               // max_linear_velocity.
                INFINITY,                               // max_angular_velocity.
                0.0,                                    // angular_inertia.
                2000.0,                                 // mass.
                center                    ,             // local_center_of_mass.
                ActivationStatus::default_threshold(),  // sleep_threshold.
                kinematic_trans.unwrap_or(Vector3::new(false, false, false)),                // kinematic_translations.
                kinematic_rot.unwrap_or(Vector3::new(false, false, false)),                  // kinematic_rotations.
                0,                                      // user_data.
                true                                    // enable_linear_motion_interpolation.
            );


            let col_data = ColData::new(
                Vector3::new(0.0, 0.0, 0.0),            // translation relative to the RigidBody it's attached to.
                Vector3::new(0.0, 0.0, 0.0),            // rotation relative to the RigidBody it's attached to.
                0.0,                                    // density ! Since we use TriMesh objects it needs to be 0.0 or game will crash !
                0.5,                                    // restitution.
                0.2,                                    // friction.
                0.01,                                   // margin.
                0.002,                                  // linear_prediction.
                PI / 180.0 * 5.0,                       // angular_prediction.
                false,                                  // sensor.
                0                                       // user_data.
            );

            PhysicObject::new(shape, rb_data, col_data)
        },
        _ => unimplemented!()
    }
}
```

Figure 8 : Code de make_object()

build_rigbd_col :

Cette fonction construit un 'RigidBody' et un 'Collider' grâce au 'PhysicObject' qui lui est passé en paramètre. Ceux-ci sont ensuite stockés dans un 'RigidBodySet' et un 'ColliderSet' indispensables pour que l'objet soit pris en compte par le monde physique, puis la fonction retourne un handle pouvant servir à retrouver l'objet pour obtenir certaines données telles que sa position.

Son code est le suivant :

```rust
/// Creates the RigidBody and Collider from the data contained in the PhysicObject given in parameter. Store them in a RigidBodySet, ColliderSet and a Vector<Collider> and returns a handle to the collider.
    pub fn build_rigbd_col(&mut self, physic_object: &PhysicObject) -> generational_arena::Index
    {
        let shape = process_shape(&physic_object.shape); // ShapeHandle object from ncollide.

        // We create the RigidBody relative to the field rbdata of 'object'.
        let mut rb = RigidBodyDesc::new()
        .translation(physic_object.rbdata.translation)
        .rotation(physic_object.rbdata.rotation)
        .gravity_enabled(physic_object.rbdata.gravity_enabled)
        .status(physic_object.rbdata.bodystatus)
        .velocity(Velocity3::new(physic_object.rbdata.linear_velocity, physic_object.rbdata.angular_velocity))
        .linear_damping(physic_object.rbdata.linear_damping)
        .angular_damping(physic_object.rbdata.angular_damping)
        .max_linear_velocity(physic_object.rbdata.max_linear_velocity)
        .max_angular_velocity(physic_object.rbdata.max_angular_velocity)
        .angular_inertia(Matrix3::from_diagonal_element(physic_object.rbdata.angular_inertia))
        .mass(physic_object.rbdata.mass)
//          .local_center_of_mass(physic_object.rbdata.local_center_of_mass)
        .sleep_threshold(Some(physic_object.rbdata.sleep_threshold))
        .kinematic_translations(physic_object.rbdata.kinematic_translations)
        .kinematic_rotations(physic_object.rbdata.kinematic_rotations)
        .user_data(physic_object.rbdata.user_data)
        .build(); // Build the rigid-body.

        rb.enable_linear_motion_interpolation(physic_object.rbdata.enable_linear_motion_interpolation);

        // We add the RigidBody to the RigidBodySet.
        let rb_handle = self.bodies.insert(rb);


        // We create the Collider relative to the field coldata of 'object'.
        let collider = ColliderDesc::new(shape)
        .translation(physic_object.coldata.translation)
        .rotation(physic_object.coldata.rotation)
        .density(physic_object.coldata.density)
        .material(MaterialHandle::new(BasicMaterial::new(physic_object.coldata.restitution, physic_object.coldata.friction)))
        .margin(physic_object.coldata.margin)
        .linear_prediction(physic_object.coldata.linear_prediction)
        .angular_prediction(physic_object.coldata.angular_prediction)
        .sensor(physic_object.coldata.sensor)
        .user_data(physic_object.coldata.user_data)
        .build(BodyPartHandle(rb_handle, 0)); // Build the collider into the world.

        // We add the Collider to the set of colliders.
        let coll_handle = self.colliders.insert(collider);

        coll_handle
    }
}
```

Figure 9 : Code de build_rigidbd_col()

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 effectuée. La combinaison touche / action est écrite en dur dans l'exemple mais il serait intéressant de mettre la configuration de touches sous la forme d'un fichier modifiable que l'on charge (ou créé s'il n'existe pas) au lancement du jeu.

L'interface doit posséder 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.

Pour notre moteur de jeu, nous avons décidé d'utiliser la librairie imgui-rs de Gekkio. Nous avons choisi cette librairie car celle-ci utilise aussi le module Glutin. De plus pour utiliser les fonctions de imgui-rs une structure ressemblant fortement à notre structure Game est requise.

Imgui-rs utilise le moteur de rendu glium et la plateforme backend Winit. L'affichage d'éléments se fait en deux étapes :

- La création d'une nouvelle fenêtre avec Window::new, provenant de Winit.
- L'ajout d'éléments fournis par la librairie imgui-rs. Ces éléments pouvant être des boutons, des barres de progression ou encore des sous-fenêtres.

Pour chaque élément indépendant on crée donc une nouvelle fenêtre et on ajoute les éléments voulu.

Nous avons ajouté imgui-rs à notre projet mais uniquement dans l'exemple, par manque de temps. Il serait judicieux de construire une structure encapsulant imgui-rs afin de faciliter la réutilisation celle-ci et pour mieux structurer le code. Nous n'avons pas eu le temps de créer de zones actives ou passives en plus sur l'écran du jeu comme les barres de vie, la batterie de la lampe torche, la carte ou le numéro de version du jeu, mais il est parfaitement possible de les ajouter.

V.6 - Son

Cette partie permet de jouer des sons dans le jeu à partir de fichiers sonores. Elle possède 2 structures qui sont utilisées ensuite dans la partie Game :SoundRessource et OneSound.

SoundRessource:

Créée à partir du chemin vers un fichier, elle représente l'échantillon d'un son. C'est une ressource qui peut être partagée entre les sons créés.

OneSound:

C'est l'objet son qui peut être joué et manipulé. Il peut être créé directement à partir du path d'un fichier ou en lui passant un SoundResource. Elle possède 2 paramètres start et end qui servent d'indices temporels. start est un objet Instant initialisé à l'Instant correspondant à sa création. Il sera ensuite réaffecté lorsqu'il sera joué. end est un flottant qui est traité comme suit dans partie game : il est initialisé à -1 qui équivaut à une lecture entière de la piste, -2 signifie une lecture à l'infini et un flottant positif à la durée de lecture souhaitée. La structure possède une fonction pour jouer le son et des fonctions pour positionner spatialement le son ou pour modifier son volume.

Le gestion des sons dans le jeu passe par les classes Game, GameEvent et RessourcesHolder.

Une des premières implémentation créait un thread spécifique à chaque OneSound pour être joué et l'effet de lecture infinie passait par un while à true. La gestion de la durée de lecture était gérée par OneSound directement. Pour permettre la gestion des sons à partir du thread de Game (pour changer le volume de tous les sons par exemple), nous avons dû abandonner cette idée et créer tous les sons sur le thread de Game.

Pour éviter de recharger les fichiers provenant du disque à chaque création de son, on stocke dans le RessourceHolder de Graphics un Holder de SoundsRessource de tous les fichiers sons présents dans le dossier sounds/ressources. Lors de la création d'un OneSound, on utilisera un des SoundRessource préchargés.

Pour lancer un nouveau son, on passe par l'envoi d'un des deux GameEvent à l'entité World de specs. C'est une eventloop dans Game qui les traite et appelle des fonctions pour créer et jouer le son.

L'event PlaySound prend en paramètre le nom du son à jouer et en option des positions spatiales si l'on veut "placer" le son. Le son est créé en recherchant dans le RessourcesHolder avec le nom pour obtenir le SoundRessource et on lance play. Cela correspond à la lecture en entier de la piste.

PlaySoundTimeLimit prend en option en plus une durée en sec. Si le champs vaut None, cela correspond à une lecture à l'infini et le champ end du OneSound est affecté à -2. Si c'est une flottant positif, end prend cette valeur. Et le son est ensuite joué.

Tous les objets son en cours de lecture sont gardés dans une HashMap de Game. A chaque frame, on regarde pour chaque son dans la HashMap si il doit être arrêté (end>0) ou relancé (end==-1) ou retiré de la map des sons en cours.

```
self.event_loop.consume()
      .run(move |event, _, control_flow|
        {
        //some

         // sound management
        {
                let name_to_pop : Vec<_> = self.sounds_played.iter()
                    .filter
                    ( |(_name,sound)|
                    (sound.end == (-1. as f32) && !sound.is_playing()) ||
                    (sound.end == (-2. as f32) && !sound.is_playing()) ||
                    (sound.end >0.0 && sound.end == sound.start.elapsed().as_secs() as f32)
                    ).map(|(name,_sound)| name.clone())
                    .collect();

                for name in name_to_pop
                {
                    let sound = self.sounds_played.get_mut(&name);
                    match sound
                    {
                        None => {},
                        Some(sound) =>
                        {
                            if sound.end ==(-2. as f32)
                            {sound.play_all();}
                            else{
                                    if sound.end >0.0
                                        {sound.stop();}
                                    self.sounds_played.remove(&name);
                                }
                        }
                    }
                }
            }
        }
```

Figure 10 : Gestion des sons en cours de lecture à chaque frame

Cependant, on a fixé le taux de frame à 30fps donc un son en "infini" pourrait avoir des coupures de 1/30 sec, ce qui reste imperceptible.

On a également ajouté un champ vol à Game pour pouvoir modifier le volume global du jeu dans une partie settings du menu avec une barre à curseur par exemple (non ajouté à notre jeu test, mais cela reste possible). Le volume normal du jeu vaut 0. Nous avons donc ajouté 2 GameEvents pour augmenter ou diminuer vol.

V.7 - Partie Centrale du Moteur de jeu

Cette partie amènes les différentes structures utilisant les autres parties afin de fournir les services attendus d'un moteur de jeu.

Les fichiers relatifs à cette partie centrale du moteur de jeu sont situées dans le dossier "src" du repos. Cette partie permet de relier tous les éléments développés précédemment et d'amener les structures que l'utilisateur va utiliser comme dans Game et GameState.

Game :

Gère les ressources du jeu, ses états, toute la partie graphique et la gestion des sons. Elle contient un EventLoop sur tous les évènements de GameEvent.

GameEvent :

Enum de tous les événements émis par les états de jeux à destination de Game: Demande d'empilement/dépilement d'états, manipulation des sons...

GameState :

Désignent les états du jeu. Ils sont ensuite stockés dans une GameStateStack qui contient tous les états de jeu tels que le menu, le jeu en lui même… Chaque état contient un objet World de specs sur lequel on travaillera, ainsi que d'autres éléments permettant le choix du comportement de l'état de jeu dans la pile, autant pour le rendu que pour la logique (les ticks de jeu).

Les composants essentiels:

Un composant est une structure ou une enum implémentant le trait Component.

Création des composants Spatial (contenant position, rotation et scale d'un objet), Model (contenant un Handle d'un objet), Lightning (stocke une Light) et PhysicComponent (stocke les propriétés d'un objet physique).

Ces composants pourront être donnés aux entitées du jeu, leur permettant par exemple d'émettre de la lumière ou de pouvoir être rendus.

V.8 - Jeu test

le gameplay :

Évoluer dans une maison style western: rotation de la caméra avec le curseur et déplacements avec les touches Z, Q, D, S. Une musique de fond démarre au lancement du jeu. La maison est éclairée par des lumières ambiantes et des bougies. Avec la touche échap, on a accès à un menu qui permet de quitter le jeu.

Le jeu test a été développé dans le dossier example. Il est basé sur les structures Game, GameEvent, GameState et Components qui créent la structure d'un jeu. Dans exemple, le jeu possède 2 états : menu et game qui sont initialisés comme suit:

init_game() :

Création et envoi à l'objet world de specs de chaque entité à afficher. Pour chaque objet que nous souhaitons ajouter à notre jeu, nous le chargeons et nous lui donnons des positions. Il est ensuite possible de l'ajouter au monde en tant que source de lumière (la lumière sera ajoutée à la même position que lui), ou bien de lui donner des propriétés physiques (dynamique, statique, ou kinetic, avec un rigidbody et un collider).

On crée également un objet dispatcher qui contient la physique du jeu, les EventSystems et la caméra.

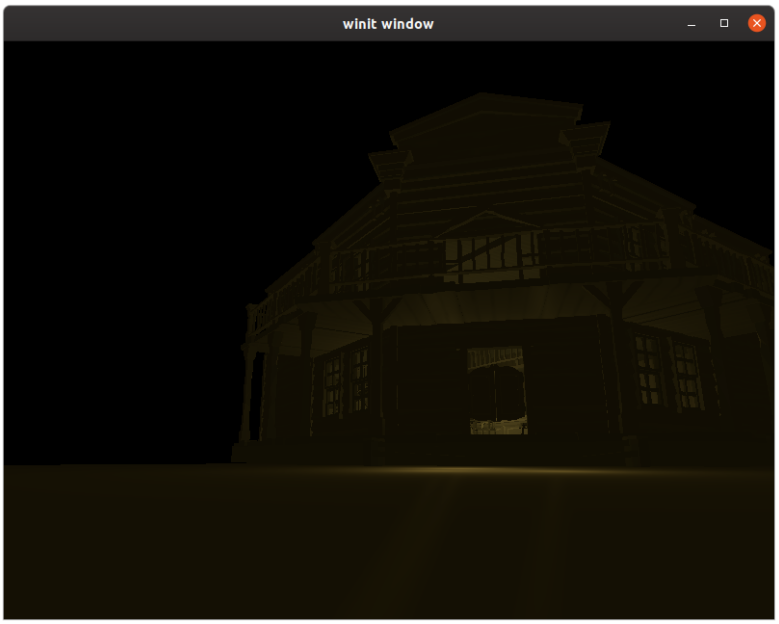
Tous les modèles Wavefront de notre jeu test ont été trouvés à l'adresse suivante https://files.nonimad.fr/public/unlisted/PolygonBundle/ dans le fichier polygon western. D'autres modèles ont été ajoutés notamment les bougies et le chandelier trouvés sur https://free3d.com/. Les ressources de notre jeu sont disponibles dans le dossier ressources de notre repos.

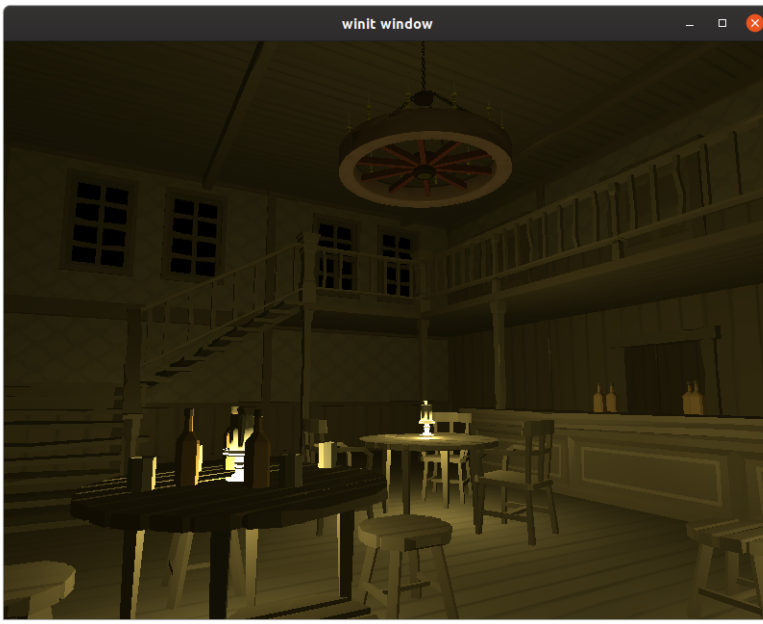
render_gui() :

Création d'une nouvelle fenêtre possédant des options (boutons) référant à différentes actions. La création de tout ce qui concerne la GUI est réalisée directement dans l'exemple (jeu test).

Le lancement du jeu se fait à l'aide de la commande *cargo run --release --example example* lancée sur un terminal.

Voici un screen des rendus obtenus:





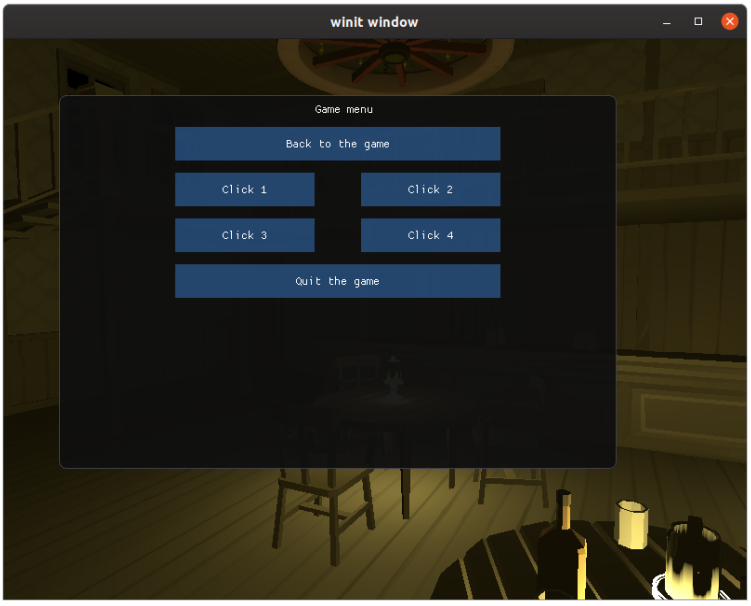


Figure 11 : Rendus scène et menu du jeu

Nous avons également un autre test pouvant être lancé à l'aide de la commande *cargo run --release --example test_phy* dans lequel nous avons mis de la physique. Cet exemple est là pour montrer que notre partie physique existe puisqu'elle n'est pas réellement visible dans l'exemple complet. Ainsi dans cet exemple, on peut voir des bouteilles tomber sur des tables ou dans le vide. Cependant, on remarque les lorsqu'elles tombent, les bouteilles ne tournent pas, et restent à leur inclinaison alors qu'elles devraient se retrouver à plat sur les tables.

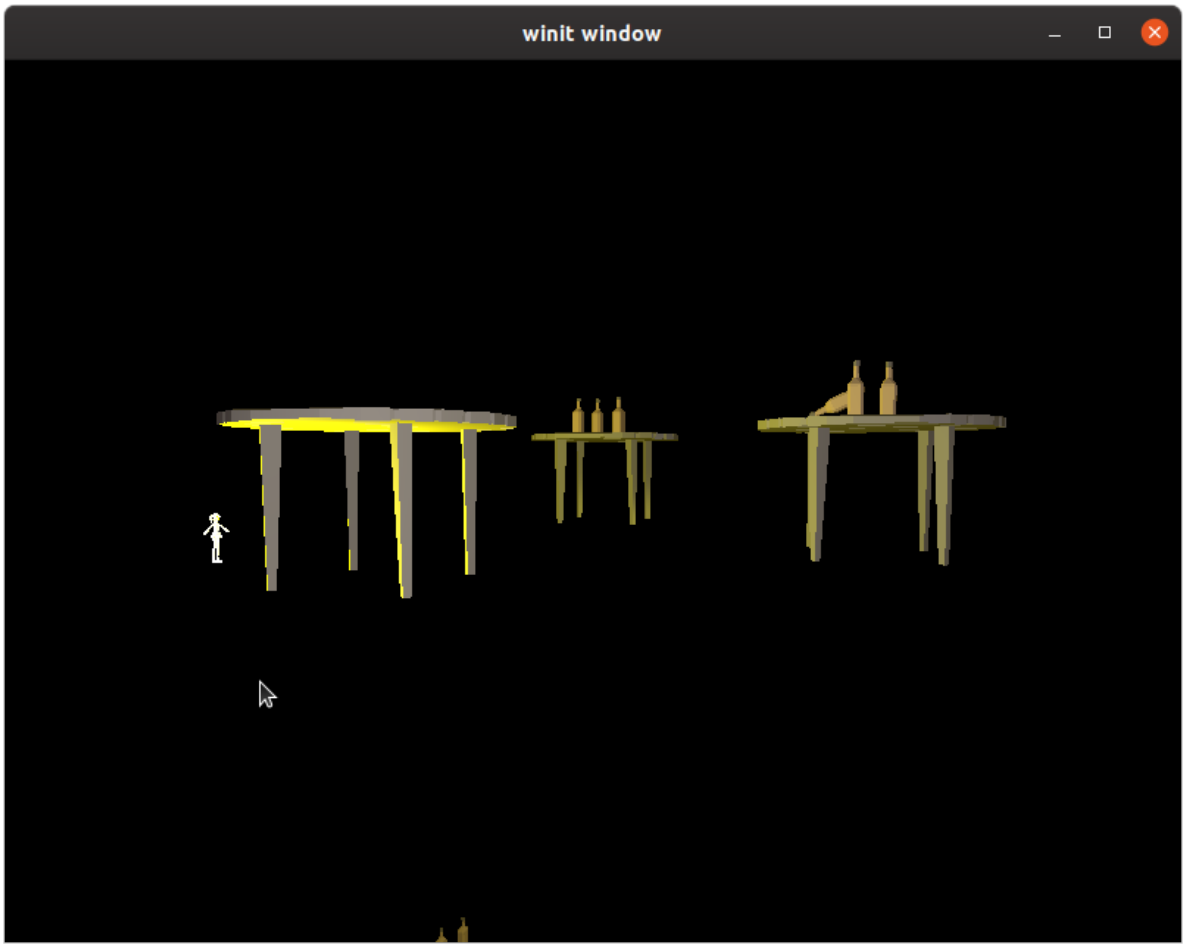


Figure 12 : Rendu test_phy avec la gravité

VI - Pistes d’approfondissement

La physique du moteur de jeu possède encore quelques inconvénients comme le fait que les objets ne tournent pas en tombant et qu’ils ne roulent pas (pour ceux qui le devrait). Néanmoins, la partie physique du moteur pourra permettre dans le futur de déplacer des objets par simple appui d’une touche (prendre et lancer/lâcher). En effet, nous avons déjà effectué du travail pour utiliser des projections de rayons, afin d'interagir avec le monde 3D via la souris.

La partie sonore pour les interactions avec les objets étant opérationnelle, il suffit de d’ajouter une détection de collision pour qu’un son se joue à chaque fois qu’un objet tombe par exemple.

Nous n’avons pas ajouté de collisions à la caméra mais il est bien sûr possible de le faire et donc d’empêcher le joueur de traverser les murs et autre objets. Il est aussi largement possible d’ajouter la fonctionnalité de saut au joueur et de l’empêcher de se déplacer dans les airs comme il le souhaite.

Il est possible d’ajouter un menu principal qui s’afficherait à l’ouverture du jeu ainsi qu’un menu de pause plus esthétique. Il serait également possible d’ajouter des zones dans le jeu comme la barre de vie, et diverses autres menus.

Concernant les choses que nous n’avons pas pu faire par manque de temps, on peut citer : Ajouter une gestion des IA pour les personnages tierces du jeu ; Rendre le jeu éventuellement jouable en ligne et en multijoueur ; Gérer différents niveaux de jeu...

Des optimisations sont aussi à faire, notamment au niveau graphique, afin de ne pas rendre les polygones qui ne sont pas à l’écran (en utilisant par exemple un système de chunks).
Il serait aussi bon de se pencher vers la possibilité de scinder le travail de Game sur plusieurs threads, notamment lancer les rendus sur un autre thread que celui qui sert à récupérer les évènements, afin de ne pas ralentir le traitement des inputs en cas de baisse de framerate.

Enfin, il serait nécessaire de rendre beaucoup plus de choses customisables, comme par exemple le nombre de layers de rendus par GameState. Il pourrait aussi être intéressant de voir pour rendre chaque scène layer par layer, et non plus de rendre tous les layers d’un coup, afin de vraiment pouvoir superposer des scènes dans des scènes et ne plus avoir à se contenter de tiles pour jouer sur la profondeur.

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.

Voici un schéma de la structure de base d'un moteur de jeu vidéo et d'un jeu test. Vous pouvez voir en vert les parties fonctionnelles implémentées lors de ce projet et en jaune celles qui nécessiteraient encore des améliorations.

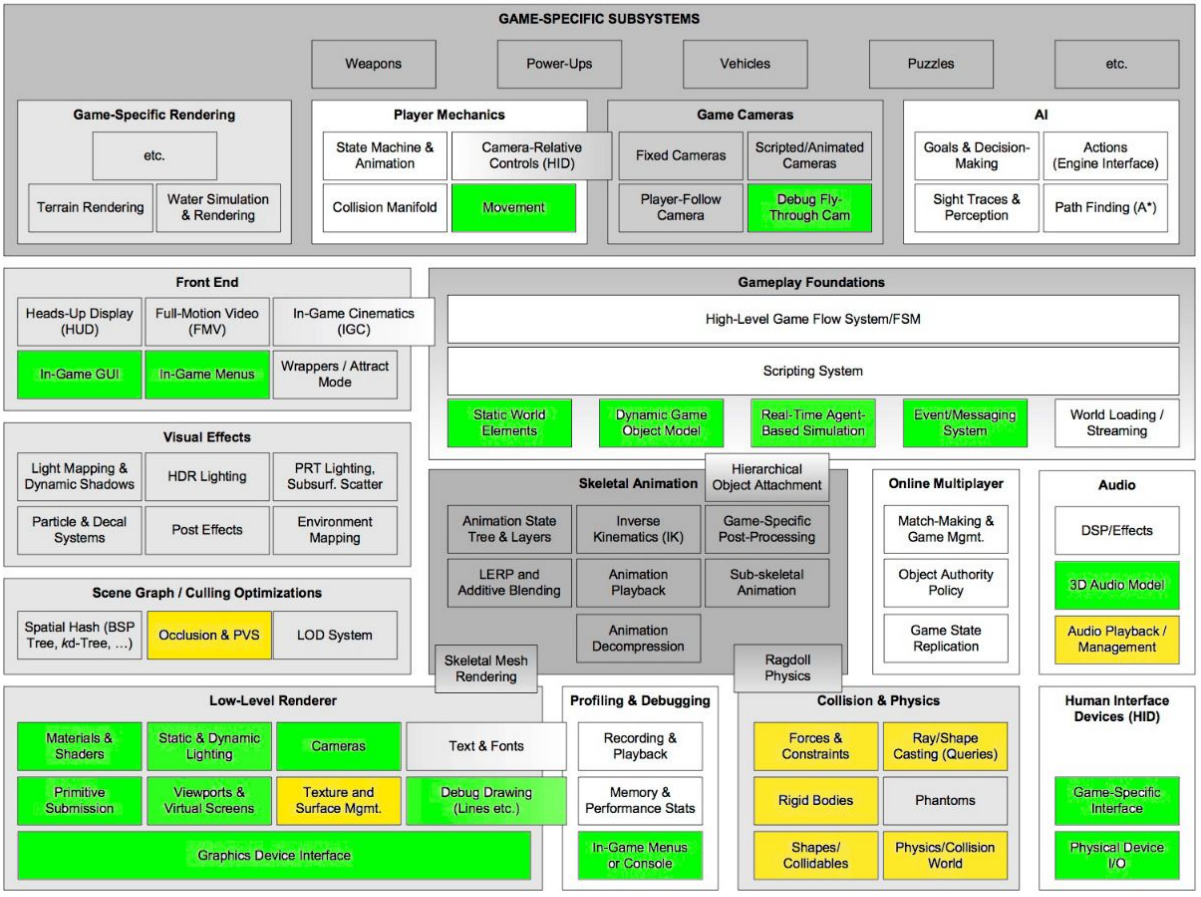


Figure 13 : Schéma global moteur de jeu vidéo (comparaison avec une structure classique)

La version complète de notre diagramme avec toutes les parties de notre moteur de jeu vidéo est disponible sur notre Git dans le dossier *RAPPORTS/diagrammes* car l'image ici serait illisible.

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 réparti. En effet, nous avions seulement entre 4 et 6h de projet par semaines, uniquement le lundi après midi. De plus, ce temps était perturbé par les cours que nous avions 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 a semblée 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 en 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.

Le confinement lié au Covid-19, nous a permis de complètement nous plonger dans le projet (après les examens) et nous avons grandement avancé en quelques semaines.

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, a aidé les autres membres de l’équipe et a géré la structure du moteur en reliant les parties entre elles. Cela 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 :

Ce projet a été pour moi l'occasion d'en apprendre plus sur l'univers des moteurs de jeux vidéos ainsi que sur le degré de communication que requiert un projet de cette envergure. J'ai également pu découvrir le langage Rust que je n'aurais sûrement jamais pratiqué lors de mon cursus d'étudiant en école d'ingénieur.

Ce projet a été l'opportunité d'apprendre de nouvelles choses sur un sujet qui me plaît et de développer ma capacité à communiquer, structurer mon code, justifier des choix, etc.

C'est une avancée satisfaisante vers mon objectif de devenir développeur de jeux vidéos.

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. C'est pour ça que j'ai fait un gros "git rebase" pour rattraper tout ça.

Barbara :

Le premier semestre m'a permis de me familiariser avec le rendu 3D et l'utilisation d'OpenGl. J'ai été contente de pouvoir apprendre le Rust, une plus-value par rapport aux autres élèves de l'école.

Durant le deuxième semestre, travailler seule sur la partie son m'a beaucoup fait progresser en Rust.

En terme d'organisation, cela m'a fait prendre conscience de la difficulté du travail en équipe sur un projet avec des parties indépendantes. Par exemple devoir réécrire des parties car elles ne sont pas optimisées ou compatibles pour être utilisée dans d'autres.

Je suis un peu déçue du résultat du jeu test que nous avons réalisé par rapport à ce que l'on pensait proposer au début du projet. Nous nous sommes pris trop tard pour assembler toutes nos parties et certaines fonctionnalités auraient pu être ajoutées avec une dizaine d'heure de plus.

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, une bonne partie du moteur de rendu, le coeur du moteur ainsi que la totalité de feu 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, OpenGL ou plus généralement en architecture logicielle. J'adore ce projet, et je pense que nous avons avancés 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, 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.

J'aurais aussi aimé que les exigences soient en cohérences avec les moyens accordés au sens large. Les créneaux de travails sont trop courts, trop peu nombreux, et les semaines de projet sont grignotées par des unitées des fois plus qu'accessoires. Or programmer ne s'improvise pas, et celà demande de très longues plages horaires dédiées uniquement à cela, et relativement peu espacées dans le temps. Je considère que 40% du temps qui nous a été attribué ne sert à rien d'autre que de se replonger dans le code et se remettre dans le bain.

Pour en revenir au projet en lui-même, je considère que notre moteur de jeu est presque utilisable, et je compte m'en servir dans le futur pour concevoir des jeux ou autres simulations.
Avoir autant avancé dans ce projet si ambitieux avec tant de bâtons dans les roues me satisfait tout à fait.

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.

Durant les dernières semaines de projet, nous avions enfin le temps de nous y consacrer pleinement et l'avancement qu'on a pu faire prouve qu'il vaut mieux travailler par grosses sessions d'une semaine que quelques heures parsemées une fois par semaine. Pendant cette dernière partie du projet, j'ai notamment travaillé sur la partie Physique avec Alexandre, et j'ai en partie réalisé notre jeu test. Tout comme Barbara, je regrette un peu de ne pas avoir eu le temps de finaliser l'exemple qui, aux vues des fonctionnalités de notre moteur de jeu, aurait pu être bien meilleur. Mais malheureusement, construire un jeu n'était pas l'objectif du projet. Il aurait pu faire l'objet d'un autre projet, et nous avons fait au mieux avec le temps que nous avions.

Sebastien ZHOU:

J'ai intégré ce projet durant le deuxième semestre, je devais donc rapidement comprendre le fonctionnement de ce projet. Grâce à l'aide des autres membres de l'équipe cela s'est fait rapidement.

La partie la plus compliqué pour moi était de coder en Rust, langage que je connaissais absolument pas. Mais ce fut enrichissant et j'ai donc pu découvrir un nouveau langage. Durant le projet mon objectif était d'intégrer le gui à notre moteur de jeu. Cette tâche était à la fois pas trop compliqué et très enrichissant dans mon apprentissage du Rust.

Personnellement j'aurais préféré coder le projet en C++. Bien que l'apprentissage du Rust était fort intéressant et enrichissant, je ne me perçois pas beaucoup utiliser ce langage dans le futur. De plus faire ce projet en C++ m'aurait permis d'apprendre/approfondir ce langage puisque je n'étais pas là au premier semestre et ai donc raté le cours sur le C++.

Aymeric SALLET:

Intégrer un projet en cours de route n'est ce qui a de plus facile, mais c'est une situation qu'on aura surement l'occasion de vivre en entreprise.
A mon arrivée, Clément et le reste de l'équipe ont pris le temps afin de nous expliquer l'ensemble du projet, ce qu'ils avaient fait, et ce qui restait à faire. Les premières réunions avec Nabil ont permis de mieux comprendre la structure du projet.
Pour ma part c'est la premiere fois que j'entendais parler du langage rust, il a donc fallu dans un premier temps explorer le langage, apprendre les bases de ce nouveau langage, et comprendre comment il fonctionne. Malheureusement nous avons eu uniquement 4 semaines a l'école avant qu'elle ne ferme.
Rentrer dans le projet a été compliqué, en effet je ne m'y connaît pas plus que ça dans les jeux vidéos, et le projet n'avait pas de lien avec la filière que je réalise a l'école (Data Science et Intelligence Artificielle). Le contexte actuel n'a pas facilité les choses, a l'école nous avions la chance de pouvoir travailler ensemble et donc de pouvoir poser des questions rapidement lorsque nous ne bloquons, ce qui a été un plus difficile a distance.
Clément nous a chargé de réaliser un menu pour le jeu, dans un premier temps il a fallu prendre en main la bibliothèque et créer des petits exemples pour voir toutes les possibilités, puis nous l'avons intégré au projet.
Le projet m'a permis de comprendre ce qui se cachait derrière les jeux videos, de découvrir un nouveau langage qui semble avoir énormément de potentiel et dont je suis sur qu'on en entendra parler dans le futur.

X - Table des Figures

Figure 1 : Hiérarchie des classes du moteur graphique 15

Figure 2 : Projection rendu graphique 18

Figure 3 : Pipeline du moteur graphique 18

Figure 4 : Un placement de caméra et de ses axes par rapport à la scène 19

Figure 5 : Démonstration de la partie graphique 20

Figure 6 : Hiérarchie des classes de la partie Physique 23

Figure 7 : Code de make_trimesh() 24

Figure 8 : Code de make_object() 25

Figure 9 : Code de build_rigidbd_col() 26

Figure 10 : Gestion des sons en cours de lecture à chaque frame 29

Figure 11 : Rendus scène et menu du jeu 32

Figure 12 : Rendu test_phy avec la gravité 33

Figure 13 : Schéma global moteur de jeu vidéo 35

XI - Annexe

Lien de notre repos GitHub :
https://github.com/734F96/moteur_jeu_video