Deadline
Vendredi 02/04/2021 – 18h (Montréal)

Nombre d'étudiants par groupe : 3

Rendu :

- Un Tag BETA.
- Il est attendu un binaire pour le push final.
- La documentation à jour.

« Teach a man to use a game engine and he'll ship a game.
Teach a man to build a game engine and he'll never ship anything »

But

- Le but du projet est de développer un moteur de jeu par équipe de 3.
- Les équipes sont restreintes car ce projet a un but pédagogique avant tout, et doit donc permettre à tous les membres du groupe de se perfectionner dans tous les domaines du développement d'un moteur.
- Le but est aussi de limiter au maximum les problèmes de gestion d'équipe.
- Par conséquent, le nombre de features du moteur sera limité afin que le projet soit faisable à 3.
- De la même façon, afin d'éviter de développer des features inutiles et d'avoir à développer trop d'outils, le moteur sera développé pour permettre de développer un FPS puzzle-game très simple.
  (type Portal ou Attractio, exemple : https://www.youtube.com/watch?v=WA7jW1yBEhw)
- L'intérêt est donc de développer uniquement les features du moteur qui seront nécessaires à un FPS, et donc faciliter les choix architecturaux.
- Comme c'est un long projet, il sera développé en plusieurs milestones. Chaque grosse feature, avant de commencer à coder, devra suivre ce processus :
- **recherche** de l'état de l'art en la matière
- **projection** de développement sur papier *(organigramme, UML…)*
- **validation** auprès de la pédagogie avec un **estimé** du temps requis.
- Après avoir fini une **feature** ou un **bug fix**, vous **devez** faire revoir votre code par un de vos coéquipiers. La **revue de code** est une pratique standard dans l'industrie. Après chaque revue, vous **devez** préciser **qui a revu votre code** dans le message de commit.
- Le challenge du projet est multiple :
  - investigation, recherche
  - technique
  - architecture
  - organisation et communication.

Contraintes pour l'Éditeur de jeu

- Doit être une exécutable Windows *(donc pas de multiplateforme)* qui se présente sous la forme d'un éditeur monolithique, comme **Unity**.
- Le code du moteur doit se situer dans une ou des libs à part.
    - L'éditeur est donc un projet indépendant qui vient se greffer après en utilisant la ou les bibliothèques du moteur de jeu.
    - À vous de décider entre utiliser des DLL ou des bibliothèques statiques pour le moteur de jeu. Lib statique = il faut recompiler les utilisateurs à chaque changement de la lib, DLL = itération facilitée mais moins performant au runtime. Idéalement, votre pipeline de build générerait des DLL en mode Debug/Release, et des libs statiques en un mode "Shipping" ou "Retail" ultra-optimisé.
- L'éditeur doit comporter les fonctionnalités suivantes :
    - Une fenêtre **Assets** pour voir le dossier des assets du projet. Quatre types d'assets doivent être gérés :
        - *meshes (.obj et .fbx)*
        - *textures (jpg, png, tga)*
        - *sons (ogg)*
        - *niveaux.*
    - Double-cliquer sur un niveau charge ce dernier.
    - Une fenêtre **Scene** qui permet de naviguer dans le niveau en free cam *(comme Unity)* avec le clavier et la souris.
    - Les objets dans la scène doivent être **sélectionnables**, et donc doivent pouvoir être tournés, translatés, et scalés avec les gizmos correspondants.
    - Il doit y avoir une fenêtre **Inspector (ou Details)** permettant de voir les propriétés et composants de l'objet sélectionné. Il est alors possible de changer ses propriétés et de rajouter des composants.
    - Il doit être possible de créer un nouvel objet, ainsi que de sauvegarder la scène dans un fichier niveau.

    - Il doit y avoir un bouton **Play** permettant de tester le niveau *in game*. Lorsqu'on sort du mode Play, le niveau se réinitialise à la dernière version de sa vue éditeur.
    - Il doit être possible de passer la Scene en mode **plein écran**.
- Les "scripts" de jeu *(équivalent des MonoBehavior d'Unity)* seront pour l'instant simplement des classes C++ intégrées au code du moteur pour aller plus vite.

Contraintes sur les Libs autorisées

- Au niveau des libs tierces, voici les propositions (demandez si vous avez une autre idée) :
- Pour le moteur de fenêtrage (inputs, etc.), au choix :
    - **SDL :** très bas niveau
    - **SFML :** un peu plus haut niveau
    - **GLFW :** standard, compatible avec toutes les API (même DirectX)
    - **Win32 :** l'API traditionnelle de Windows, un peu plus complexe.
- Pour l'interface graphique de l'éditeur, au choix :
    - **Qt** : "monstre sacré" de la GUI, puissant, joli… Mais complexe à maîtriser.
    - **Windows MFC** (lien 1, lien 2) : l'API classique de Windows, bonne intégration avec l'OS, mais parfois complexe.
    - **ImGui** : Moins de doc que les autres mais plus rapide à maîtriser.
- Une seule API de rendu doit être supportée, au choix :
    - **OpenGL 4.6** : ce que vous utilisez depuis la GP1.
    - **Vulkan** : l'évolution d'OpenGL, beaucoup plus performant et bas niveau.
        - libs autorisées : validation layers, vulkan.hpp, shaderc.

    - **DirectX 12** : l'équivalent DirectX de Vulkan, pour un dépaysement total.
- Pour le moteur physique, au choix :
    - **Bullet** : le moteur de physique open source le plus accessible.
    - **PhysX** de Nvidia : plus complexe mais plus performant.
- Pour le moteur de son, au choix :
    - **fmod** : Suite logicielle audio de qualité professionnelle. Complexe
    - **IrrKlang** : Lib complète, mais closed source
    - **SoLoud** : Peu de docs mais open source.
    - **OpenAL**: petit mais puissant. Bas niveau (n'a pas de features très avancées)

- Bibliothèque de mathématiques entièrement codée en interne *(pas de glm sauf pour les tests)*
    - Vous êtes autorisés, si la situation le requiert, à développer une "couche de compatibilité" entre votre lib maths et la lib maths du moteur physique de votre choix
- Pour l'import des modèles 3D :
    - **Assimp** (demandez si vous avez une autre idée)
- Pour l'import de textures :
    - **stb_image** : petit mais puissant
    - **SOIL** : bibliothèque bâtie sur stb_image, plus complète mais abandonnée (pour OpenGL)
    - **DirectXTex** : la lib de manipulation de textures de DirectX.
- Choisissez judicieusement vos batailles !
    - PhysX est plus performant, mais plus "hardcore" à utiliser que Bullet...
    - Afficher un hello triangle en Vulkan puis refactorer le code en quelque chose de réutilisable peut vous prendre plusieurs semaines...

- Utilisez des technos complexes vous donnera un meilleur aperçu de comment fonctionne un “vrai” moteur de jeu mais représente plus de travail. Réfléchissez bien !
- N’oubliez pas les quatre piliers du développement logiciel : “Recherche, Développement, Tests, Intégration”. Les quatre peuvent prendre beaucoup plus de temps avec une techno que vous ne connaissez pas.

Tâches

Le projet étant conséquent, vous devez le développer suivant des tâches ordonnées.

Tâche 0

- Former un groupe de 3 et demander poliment un dépôt GIT.
- Convenir d’une norme C++ *(nomenclature, namespaces, dossier, indentation, etc)* et l’écrire en détail dans un document que vous mettez sur votre git.
- Dans votre document, vous devez expliquer comment fonctionnent les bibliothèques externes *(ex: bibliothèque pour charger les textures)*, fichiers sources à intégrer ? Si oui dans quel dossier ? Bibliothèque statique ? Bibliothèque dynamique ? Comment la charger ?
- Chaque bibliothèque utilisée doit faire l'objet d'un **Wrapper** afin de ne pas finir “mariés” avec.
  - C'est-à-dire que les fonctions de cette bibliothèque ne sont pas appelées partout à travers le code, mais uniquement par les fonctions de votre Wrapper, qui elles sont appelées par le reste du code moteur.
  - L'intérêt est d'abstraire au maximum la bibliothèque en encapsulant, et de garantir une nomenclature uniforme au sein du moteur.
  - C’est le principe du design pattern Facade.
- Vous devez indiquer le choix de l'API graphique et en expliquer les raisons.
- Vous devez montrer votre document au professeur avant de passer à la suite.

Tâche 1 : Bibliothèque mathématique minimale

- Assurez-vous avec le reste de la classe que les structures suivantes sont codées et fonctionnelles dans votre lib de maths :
    - Vector 2d, Vector 3d avec les fonctions suivantes :
        - opérateurs ==,!=,+,-,+=,-=
        - la norme (magnitude)
        - fonction qui normalise le vecteur
        - le produit scalaire
        - le produit vectoriel *(uniquement 3d)*
        - le scale par une valeure flottante

    - Matrice 4 x 4:
        - operator Matrice * Matrice
        - operator Matrice * Vector3
        - une fonction qui vérifie si la matrice est orthogonale
        - une fonction qui inverse la matrice, en considérant qu'elle est déjà orthogonale, *(donc sans calcul de déterminant)*
        - une fonction qui crée une matrice de translation *(avec en paramètre le vecteur translation)*
        - une fonction qui crée une matrice de scale *(avec en paramètre le vecteur de scale)*
        - une fonction qui crée une matrice de rotation autour de l'axe des X *(avec en paramètre l'angle en degrés)*, idem pour les axes Y et Z.

    - Quaternion
        - fonction type RotateVector(vec3)
        - combinaison de quaternion
        - inversion de quaternion
        - conversion en mat4 de Rotation et depuis des angles d'Euler.

- Ces classes doivent faire l'objet d'un test unitaire poussé et chaque fonction doit donc être testée individuellement.

- Une fois que vous avez montré le code et les tests unitaires au professeur, vous pouvez passer à l'étape suivante.

Tâche 2 : Prototype de rendu

- Développer un prototype rapide permettant d'afficher un objet en 3D *(un cube par exemple)* tournant sur lui-même, texturé et éclairé en utilisant l'API de rendu de votre choix, avec des contrôles caméra type free cam, afin de vous familiariser avec cette API *(et notamment les shaders).*
- Vous enquêterez notamment sur comment sélectionner un objet en cliquant dessus.

Tâche 3 : Render Hardware Interface

Quelque-soit l'API que vous avez choisie, celle-ci se compose de fonctions bas-niveau permettant de commander le GPU.

Tout comme pour les bibliothèques, il est important d'englober ces fonctions dans un **wrapper bas niveau** qui sera utilisé par le code de rendu de plus haut niveau.

Ce wrapper est généralement appelé une **RHI**, ou **Render Hardware Interface** dans la littérature.

Il permet en particulier de regrouper des appels de fonctions de l'API permettant de faire simplement des choses un peu complexes. Par exemple, charger un shader en OpenGL impose de nombreuses étapes et donc appels de fonctions *(créer un id, lire le code source, vérifier qu'il n'y a pas d'erreur, logger les erreurs si nécessaire...)* et il est donc préférable d'englober toutes ces étapes dans une seule fonction dont le rôle est de **charger un shader.**

- Concevoir une liste de fonctions de la RHI supportant toutes les fonctionnalités de votre prototype, en portant un soin particulier à choisir des fonctions qui regroupent plusieurs appels aux fonctions de l'API, comme dans l'exemple ci-dessus.
- Une fois cette liste de fonctions établie, montrez-la au professeur.
- Après validation, implémenter cette dernière. Votre projet, qui doit être propre et bien organisé, doit alors contenir ces premières versions de la RHI.
- Afin de tester votre RHI, modifiez votre prototype pour remplacer tous les appels à l'API de rendu par des appels à votre RHI, en s'assurant que tous les calculs mathématiques utilisent votre bibliothèque maths. N'hésitez pas à modifier la RHI si vous vous rendez compte que c'est devenu nécessaire.
- Vous devez montrer ce travail avant de passer à l'étape suivante.

Tâche 4 : Fenêtrage, entrées-sorties

Pour afficher le rendu à l'écran et récupérer les entrées claviersn et souris, vous allez devoir utiliser une bibliothèque à cet effet. Vous argumenterez votre choix dans le document prévu à cet effet.

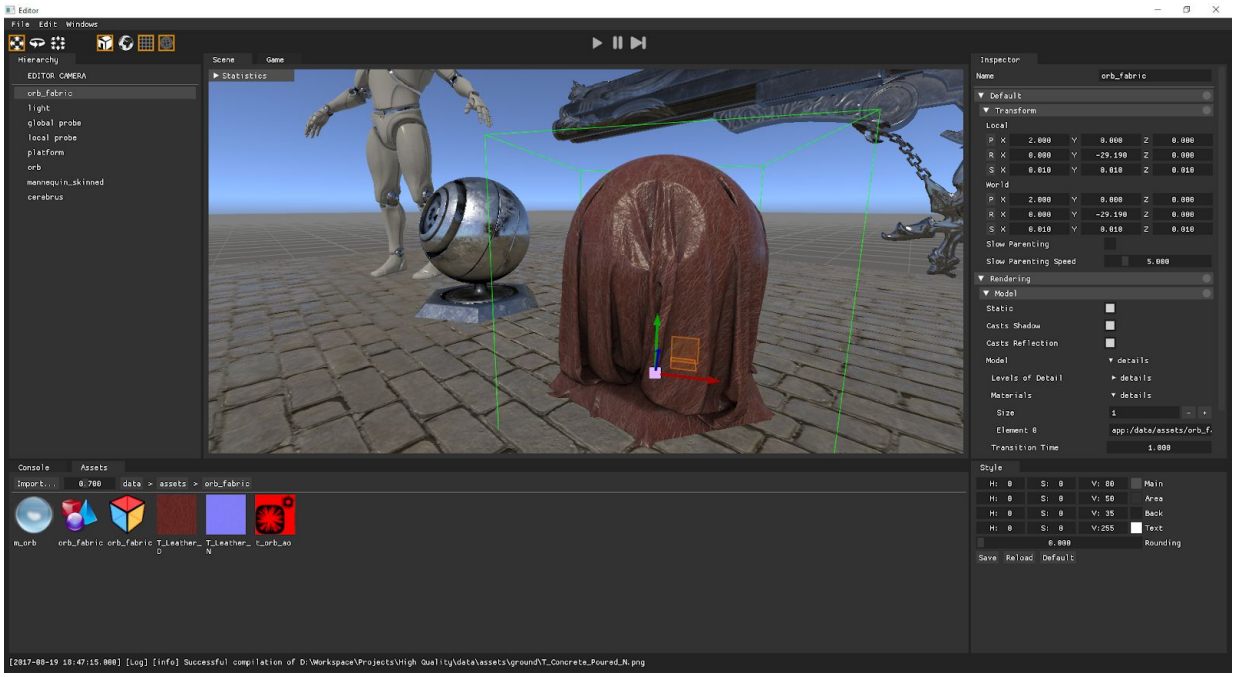
- Vous devez alors concevoir, sous forme de diagramme UML, la ou les classes permettant de gérer :
    - La création d'une **fenêtre** et du **contexte** de rendu
    - La **boucle de rendu** à l'écran
    - Lancer la **simulation**
    - La récupération des **touches clavier et de la souris** *(clics, position)*

- Cette ou ces classes doivent encapsuler entièrement la bibliothèque de fenêtrage utilisée. Votre code doit se comporter comme une **façade**, ne donnant accès qu'à des fonctions *(ex: Vec2 GetMousePos())* qui seront appelées par le code moteur qui lui ignore l'implémentation sous-jacente *(donc on pourrait par exemple remplacer SDL par SFML, ce serait transparent pour le reste du code moteur)*.

- Vous prendrez un soin particulier à ce qu'il n'y ait aucune donnée en dur, notamment la taille de la fenêtre.

- Comme précédemment, vous devez montrer vos travaux au professeur avant de passer à l'implémentation.

- Comme précédemment, modifiez votre prototype pour qu'il utilise votre wrapper de fenêtrage et entrées-sorties et montrer le résultat au professeur.



Exemple de rendu possible

Phase II

Une fois les fondamentaux de l'éditeur et de votre système de rendu implémentés et fonctionnels lors de la Phase 1, il va falloir aller plus loin.

Tâche 5 : Core

Vous allez devoir implémenter les features suivantes.

Vous devez écrire soit un diagramme UML, soit du pseudo-code **détaillé** et montrer le résultat au professeur. **Après que celui-ci ait validé votre travail**, vous pouvez commencer à développer une première version de ces 3 features, **mais pas avant**.

- **Scene Graph.**

  Un scene graph représente tous les objets de la scène ainsi que les interactions entre eux, il fait office d'interface entre le gameplay et le rendu.

  Il doit être capable de représenter les relations parent-enfant des entités présentes dans la scène. Si un parent bouge, tous ses enfants doivent être impactés par le changement de transformation.

  Il est donc fondamental de bien l'architecturer afin qu'il puisse supporter d'importantes modifications du rendu graphique sans être altéré.

  On doit pouvoir ajouter et retirer une entité du graph (un SceneNode), reparenter une entité, et une transformation d'un parent doit affecter en cascade tous ses enfants.

  Vous devez afficher votre scene graph dans l'éditeur au moyen d'une fenêtre type Hierarchy Unity ou World Outliner UE4.

- **Gestionnaire de ressources (Resource Manager).**

  Vous allez devoir reprendre et améliorer votre **Resource Manager** de début de GP2.

Pour commencer, il faut que le moteur puisse avoir une façon simple d'ouvrir n'importe quel type de ressource *(modèle 3D, texture, ...)*.

Une ressource déjà ouverte et en cours d'utilisation, si demandée de nouveau, ne doit pas reparser un fichier mais renvoie un "handle" vers la ressource existante.

Ce n'est que lorsqu'il n'y a plus de handles sur une ressource que le moteur la libère.

Le chargement d'un asset ne doit pas "freezer" l'éditeur (mais se faire sur un thread à part).

(Il sera peut-être utile d'imaginer des "*ressources permanentes*" qu'on charge une fois et ne libère jamais.)

Ensuite, vous allez devoir implémenter un **format de fichier maison** contenant des **métadonnées** concernant chaque fichier. (Pensez à l'équivalent d'un .uasset Unreal ou d'un .meta Unity).

Pour une texture par exemple, il contiendra le nombre de mipmaps utilisé, format sRGB ou pas, niveau de compression, etc… Tout ce qu'il faut pour savoir comment utiliser cette texture par la suite.

Stocker un objet sous forme de métadonnées afin de le sauvegarder et le recharger à l'identique est un procédé de **sérialisation**.

Il faudra donc déterminer s'il est plus judicieux de sérialiser chaque type de ressource vous-même ou s'il est préférable d'utiliser une bibliothèque déjà existante *(et dans ce cas préciser laquelle et surtout justifier ce choix)*.

Toute cette démarche doit être effectuée par écrit dans un document.

- **Boucle de rendu *(sur 1 seul thread)***

  Vous devez écrire la boucle de rendu (ou "main loop") du moteur en pseudo-code.

  Prenez Unity ou Unreal par exemple, et remarquez que vous n'aurez jamais à coder la main loop de la simulation vous-même, car c'est une tâche du moteur, et non du game. Votre logique moteur devrait donc être totalement indépendante de la logique game, et fonctionner pour n'importe quel type de jeu.

À quoi devrait-elle ressembler ? Pensez aux différentes phases de l'update d'un jeu :

- inputs
- update gameplay (Tick gameplay)
- update physique (Tick physique / FixedUpdate)
- update animation
- update render
- et dans quel ordre doit-on les appeler, et quel contrôle vous souhaitez donner à l'utilisateur.

Voyez-vous où pourrait-on (et où on ne pourrait pas) introduire du multithreading ici ?

Tâche 6 : Physique, UI et Comportements

- **Moteur Physique**

  Vous devez intégrer le moteur physique de votre choix de façon à pouvoir gérer facilement :
  - Les **Dynamic Rigidbodies** : a.k.a. les entités physiques les plus courantes. Seuls les **parallélépipèdes** *(box)* et les **capsules** doivent êtres supportés, afin de gérer les cubes transportables, les plateformes *(box)* et le joueur *(capsule, qui doit maintenir une orientation verticale).*

  - Les **meshs de collision**, pour gérer les décors et les collisions complexes (avec les personnages par exemple).

  - Vous devrez intégrer les capacités de debug draw de votre moteur physique avec votre moteur de rendu afin de pouvoir avoir un rendu visuel des shapes de collision.

- **UI / HUD**

- Il vous faut intégrer un gestionnaire d'UI, qui permet d'afficher des images et de gérer une fonte avec laquelle afficher du texte à l'écran en mode "overlay" par-dessus le jeu, un peu comme un Canvas2D Unity.
    - Vous devrez gérer au moins des éléments **Text** (doit pouvoir gérer n'importe quelle police) et **Image** *(dans le but de réaliser le HUD de votre jeu).*
    - Vous devrez ajouter un nouveau type d'asset "Font" à votre moteur.
    - Le HUD doit pouvoir s'adapter à un changement de résolution (passage en full screen par exemple) pour garder la même position relative dans l'image, et gérer correctement la transparence.

Wonder Boy III Remake, dont le HUD a été fait avec ImGui

- **Scripting : Behaviors**

Il faut intégrer un système de scripts de behavior équivalent aux MonoBehaviors d'Unity, de façon à pouvoir scripter facilement le comportement d'un objet dans votre jeu.

Votre Behavior sera implémenté côté moteur par une interface en C++ avec au moins des fonctions virtuelles *Start*() et *Update*(). On doit être capable d'ajouter ou retirer un behavior quelconque à n'importe quel objet dans un niveau de jeu.

Afin d'être capable de modifier les scripts facilement et rapidement, il va vous falloir introduire une compatibilité avec un **langage de script** dans votre moteur. La plupart d'entre eux sont **interprétés**, ce qui veut dire que le script est lu et traduit en langage machine **à la volée** par un **interpréteur**, contrairement aux langages comme C++ par exemple qui sont **compilés** (un **compilateur** traduit le code en langage machine et génère un exécutable prêt à être lancé). Un peu moins performants au runtime, leur grand avantage est qu'il est très facile d'itérer avec (aucun temps de compil requis).

On vous conseille le **langage Lua** pour sa facilité d'intégration avec le C et son côté "battle tested" (les addons de WoW sont programmés en Lua).

Tâche 7 : Editeur et Jeu

- C'est le moment de vérité ! Nous devrions avoir tous les ingrédients pour commencer à faire un jeu basique.

- Réaliser un puzzle game en vue à la première personne avec au moins les features suivantes :
    - Déplacement d'un personnage en vue à la première personne, qui collisionne avec les murs et peut sauter
    - Salle statique simple composée d'un mur et 4 sols, et quelques blocs, permettant au joueur de sauter de bloc en bloc.
    - Cube régi par la physique que le joueur peut transporter d'un bout à l'autre de la pièce
    - Boucle de jeu, le joueur gagne la partie lorsqu'il déplace le cube vers un point du niveau précis, un message de victoire s'affiche alors, utilisant votre système de HUD. Ensuite le jeu redémarre.

- Ce sont les features minimales mais vous êtes libres d'en ajouter.

- Dans un premier temps, il faut pouvoir jouer dans l'éditeur. Ensuite, attardez-vous sur la tâche de **packager** votre jeu, afin de pouvoir construire une build standalone qui sera capable de lancer votre jeu sans l'éditeur.

  De quoi avez-vous besoin ?

- **Vous devez coder le plus proprement possible, avoir des fichiers de configurations et de logs.**

- **Votre git doit être propre et refléter le travail de chacun.**

- **La vitesse de votre jeu ne doit pas dépendre de la puissance de la machine (utilisez le delta time).**

La soutenance évaluera :

- Si la liste des tâches est correctement effectuée.
- Sur la documentation rendue.
- Si le code est propre, bien architecturé et aussi performant que possible.

Liens utiles

- Game design Pattern - Vidéo
- Design-Patterns and Anti-Patterns
- Dependency Injection and Abstractions