Moteur

Sujet

- Le but du projet est de développer un moteur de jeu par équipe de 3 pendant 75 jours.
- Les équipes sont restreintes (3 personnes) 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 faire de l'outil pour de l'outil, le moteur sera développé pour permettre de développer un très simple FPS puzzle-game (exemple de gameplay : https://www.youtube.com/watch?v=WA7jW1yBEhw)
- L'intérêt est donc de développer uniquement les features du moteur qui seront nécessaire au FPS, et donc faciliter les choix architecturaux
- Comme c'est un long projet, il sera développé en plusieurs milestones
- Comme c'est un projet complexe, chaque grosse feature, avant d'être implémentée, devra faire l'objet d'une recherche de l'état de l'art en la matière, d'une projection de développement sur papier (organigramme, UML…), et d'une validation auprès de la pédagogie
- Le challenge du projet est multiple :
  - investigation, recherche
  - technique
  - architecture
  - organisation et communication

Voici les contraintes et spécificités du projet

- Doit être une executable Windows (donc pas de multiplateforme) qui se présente sous la forme d'un éditeur comme Unity
- La librairie utilisée pour l'interface graphique de l'éditeur est au choix (QT, Windows Form, …)
- Le code du moteur hors editeur doit être dans une DLL à part, donc il doit être indépendant du code de l'éditeur qui vient se greffer après
- Une seule API de rendu doit être supportée : DirectX 12, Vulkan, OpenGL 4.5 au choix
- Vous devez intégrer un moteur physique existant, au choix : PhysX ou Bullet
- API de son au choix
- Librairie de mathématiques entièrement codée en interne (pas de glm, respectez-vous)

- Le moteur doit supporter l'import des fichiers .obj et éventuellement .FBX
- L'éditeur doit comporter les fonctionnalités suivantes :
  - Une fenêtre Assets pour voir le dossiers des assets du projet (mesh, textures, sons, niveaux)
  - Double-cliquer sur un niveau charge ce dernier
  - Une fenêtre Scene qui permet de naviguer dans le niveau (comme Unity) avec le clavier et la souris
  - Les objets dans la scène doivent être sélectionnable, et donc doivent pouvoir être tournés, translatés, et scalés avec les gizmos correspondants.
  - Il doit y avoir une fenêtre Inspector 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 sauvegarder la scène
  - Il doit y avoir un bouton play permettant de tester le niveau ingame
  - Il doit être possible de switcher en mode plein écran
- Les scripts du jeu (équivalents aux monobehavior d'Unity) seront pour l'instant simplement des classes C++ intégrées au code du moteur

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 pushez sur votre git
- Dans votre document, vous devez expliquer comment fonctionnent les librairies externes (ex: librairie pour charger textures), fichiers sources à intégrer ? Si oui dans quel dossier ? Librairie statique ? Librairie dynamique ? Comment la charger ?
  Chaque librairie utilisée doit faire l'objet d'un Wrapper : c'est à dire que les fonctions de cette librairies 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 librairie en l'encapsulant, et de garantir une nomenclature uniforme au sein du moteur.
- Dans votre document vous devez indiquer le choix de l'API graphique (au choix : DirectX 12, Vulkan, OpenGL 4.5) et en expliquer les raisons
- Vous devez montrer votre document à l'intervenant avant de passer à la suite

Tâche 1 : Prototype de rendu

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

Tâche 2 : Librairie mathématique minimale

Vous devez coder les structures suivantes (sans utiliser aucune librairie externe):

- Vector 2d, Vector 3d avec les fonctions suivantes :
    - operateurs ==,!=,+,-,+=,-=
    - la norme
    - fonction qui normalise le vecteur
    - le produit scalaire
    - le produit vectoriel (uniquement 3d)
    - le scale fois un flottant
- 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 pas de 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

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

Une fois que vos classes fonctionnent et que vous avez montré le code et les tests unitaires à l'intervenant, vous pouvez
passer à l'étape suivante.

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 librairies, il est important d'englober ces fonctions dans un wrapper qui sera utilisé par le code de rendu.

Ce wrapper est généralement appelé Render Harware Interface (RHI) dans la littérature. Il permet en particulier de regrouper des appels de fonctions de l'API permettant de faire simplement des choses un peu complexe. Par exemple, charger un shader
en OpenGL impose de nombreuses étapes et donc appels de fonctions (crée 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 du 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-là à l'intervenant.
- Uniquement après que l'intervenant ait validé votre liste, implémenter cette dernière. Votre projet, qui doit être propre et bien organisé, doit alors contenir ces premières versions du RHI et de la librairie mathématique
- Afin de tester ces parties du projet déjà réalisées, modifier votre prototype pour remplacer tous les appels à l'API de rendu par des appels à votre RHI, et tous les calculs mathématiques par des appels à votre librairie. N'hésitez pas à modifier la RHI si vous vous rendez compte que c'est devenu nécessaire. Vous ferez attention en particulier à ce que la RHI ou la librairie de maths créé les matrices de projection
- 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 claviers et souris, vous allez devoir utiliser une librairie à cet effet.
- Vous utiliserez au choix : SDL, SFML, ou DirectX. Vous devez argumenter ce 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 librairie de fenêtrage utilisée. C'est à dire que votre code doit se comporter comme un wrapper, 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 SMFL, 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 à l'intervenant avant de passer à l'implémentation
- Comme précédemment, modifier votre prototype pour qu'il utilise votre wrapper de fenêtrage et entrées-sorties et montrer le résultat à l'intervenant

Tâche 5 : Core
- Scene Graph. Le scene graph représente tous les objets de la scène, il fait office d'interface entre la gameplay et le rendu.

Il est donc fondamental de bien l'architecturer afin qu'il puisse supporter d'importantes modifications du rendu graphique sans
être altéré.
- Gestionnaire de ressources. Chaque resource est contenu dans un fichier, mais représente des données différentes en fonction du type de resource (modèle 3D, texture, ...). Il faudra donc, pour chaque type de resource, déterminer s'il est plus judicieux de coder vous même la serialization, ou s'il est préférable d'utiliser une librairie 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 en pseudo-code.
Une fois ce travail effectué, vous devez montrer le résultat à l'intervenant. Après que celui-ci ait validé votre travail,
vous pouvez commencer à développer une première version de ces 3 features, mais pas avant.

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.