Procédure des rendus

Début	Alpha
Lundi 20/02/2023 – 9h	Mardi 14/03/2023 – 19h
Beta	**GOLD**
Mercredi 03/05/2023 – 19h	Mercredi 31/05/2023 – 19h
Jury	**Fin**
Jeudi 08/06/2023 – 19h	Vendredi 16/06/2023 – 16h

Nombre d'étudiants par groupe : 3 à 4

Rendu Alpha (présentation) :

- Sur Git :
  - tag ALPHA sur Git (sur la branche master)
  - libs pour qu'on puisse compiler votre projet
- Sur l'exercice : Moteur_NomDuGroupe_ALPHA.zip
  - Exe test moteur
  - Support de présentation au format PDF
  - TDD au format PDF

Rendu Beta (review) :

- Sur Git :
  - tag BETA sur Git (sur la branche master)
  - libs pour qu'on puisse compiler votre projet
- Sur l'exercice : Moteur_NomDuGroupe_BETA.zip
  - Exe moteur avec éditeur
  - TDD au format PDF

Rendu GOLD (présentation en anglais) :

- Sur Git :
    - tag GOLD sur Git (sur la branche master)
    - libs pour qu'on puisse compiler votre projet
- Sur l'exercice : Moteur_NomDuGroupe_GOLD.zip
    - Exe moteur avec éditeur
    - Support de présentation au format PDF
    - TDD au format PDF
    - Screenshots : 5 de votre éditeur dans le dossier Screenshots
    - Vidéos : 2 dans le dossier Videos
        - 1 vidéo de démonstration de l'utilisation de votre moteur (15 minutes max)
        - 1 vidéo de promotion de votre moteur qui met en valeur ses spécificités (2 minutes max)

Rendu Jury (présentation en anglais au MEET-UP) :

- Sur Git :
    - tag JURY sur Git (sur la branche master)
    - libs pour qu'on puisse compiler votre projet
- Sur l'exercice : Moteur_NomDuGroupe_JURY.zip
    - Exe moteur avec éditeur
    - Exe jeu
    - Support de présentation au format PDF
    - TDD au format PDF
    - Screenshots : 5 de votre jeu dans le dossier Screenshots
    - Vidéos : 1 vidéo de démonstration/trailer de votre jeu (2 minutes max)
      La vidéo et les screenshots du jeu peuvent être rendus pendant la semaine après la présentation JURY
      Les exe pourront également être mis à jour après la présentation mais doivent avoir une première version disponible pour la JURY

4 points seront retirés si les consignes ne sont pas appliquées

Pénalités de retard:

- 5 minutes de retard = -1 points
- 15 minutes de retard = -2 points
- 30 minutes de retard = -4 points
- 1 heure de retard = -10 points
- Plus de retard = 0

Sujet

- Ce projet à pour but de développer un **moteur de jeu**.
- Il est pédagogique avant tout et doit donc permettre à tous les membres du groupe de **comprendre et se perfectionner dans le maximum de domaines du développement d'un moteur de jeu**.
- Il devra être réalisé en équipe de **3 à 4 personnes**.
- La taille des équipes permettra de limiter les problèmes d'organisation.
  Le nombre de fonctionnalités sera donc limité pour être faisable à 3.
- De la même façon, **afin d'éviter de voir trop grand et faire de l'outil pour de l'outil, le moteur sera pensé exclusivement pour faciliter le développement de FPS-puzzle-game** (*lien exemple*)
- Puisque c'est un long projet, il sera suivi sur plusieurs **soutenances**.
  La dernière aura lieu devant des professionnels de l'industrie, **il est donc primordial de réaliser un à plusieurs niveaux de jeu aboutis montrant les spécificités de votre moteur.**
- Étant donné sa complexité, **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
  - technique
  - architecture
  - organisation
  - communication
  - présentation

Contraintes et spécificités du projet

- **Le moteur doit être un exécutable Windows** *(donc pas de multiplateforme)* qui se présente sous la forme d'un **éditeur** comme Unity3D ou Unreal Engine.
- La bibliothèque utilisée pour l'interface graphique de l'éditeur est au choix *(ImGUI, 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 ou OpenGL 4.5**
- Vous devez intégrer un moteur physique existant, au choix : PhysX ou Bullet
- **Bibliothèque 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 sélectionnables**, et donc doivent pouvoir être tournés, translatés, et mis à l'échelle 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 monobehaviour 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 : Préparation du projet

- Former un groupe de 3(ou 4) 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 envoyez sur votre git
- Dans votre document, vous devez expliquer comment fonctionnent les bibliothèques externes *(ex: bibliothèque pour charger 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 : 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 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 4.6)* 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 : Bibliothèque mathématique minimale

- Vous devez coder les structures suivantes *(sans utiliser aucune bibliothèque externe)*:
  - **Vector 2D**, **Vector 3D** avec les fonctions suivantes :
    - opérateurs ==,!=,+,-,+=,-=
    - 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 bibliothèques, il est important d'englober ces fonctions dans un wrapper qui sera utilisé par le code de rendu.

Ce wrapper est généralement appelé Render Hardware 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 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 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 bibliothèque mathématique
- Afin de tester ces parties du projet déjà réalisées, modifiez 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 bibliothèque. 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 bibliothèque 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 bibliothèque à cet effet.

- Vous utiliserez au choix : **GLFW 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 bibliothèque 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 GLFW par une autre librairie comme 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 à l'intervenant 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 à l'intervenant

Tâche 5 : Core

- **Scene Graph**.
  Le scene graph représente tous les objets de la scène, il fait office d'interface entre le 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 ressource est contenue dans un fichier, mais représente des données différentes en fonction du type de ressource *(modèle 3D, texture, ...)*. Il faudra donc, pour chaque type de ressource, déterminer s'il est plus judicieux de coder vous-même la serialization, 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 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.

Tâche 6 : Physique, UI et Comportements

- **Physique** : intégrer le moteur physique de votre choix *(Bullet ou PhysX)* de façon à gérer facilement :
  - **Rigidbody dynamiques** : uniquements parallélépipèdes *(box)* et les capsules doivent êtres supportés, pour gérer les cubes transportables, plateformes *(box)* et le joueur *(capsule, qui doit maintenir une orientation verticale)*
  - **Les meshs statiques en tant que collision**, pour gérer les décors

- **UI** : intégrer un gestionnaire de font pour afficher du texte à l'écran *(dans le but de réaliser le HUD de votre jeu)*

- **Comportements** : intégrer un système de monobehaviour équivalent à Unity, de façon à scripter facilement le comportement d'un objet, en implémentant simplement une interface en C++ *(avec des fonctions virtuelles Start() et Update())*

Tâche 7 : Editeur et Jeu

- Ajouter les fonctionnalités citées plus haut concernant l'éditeur
- Réaliser un Puzzle en vue à la première personne avec au moins les features suivantes :
    - Menu simple de jeu *(Ex : sélection de niveau, options, écran de fin de partie, etc.)*
    - Déplacement d'un personnage en vue à la première personne gérant les collisions avec les sols, murs et autres plateformes mouvantes
    - Le joueur doit pouvoir sauter de bloc en bloc afin de progresser dans le niveau
    - Déplacement d'objets régi par la physique *(ex: Cube que le joueur peut transporter d'un bout à l'autre du niveau pièce)*
    - Boucle de jeu : le joueur gagne la partie lorsqu'il arrive à atteindre un emplacement précis du niveau. Un événement pour féliciter le joueur doit alors être déclenché *(Ex: message de victoire, fx, sons, animations, etc.)*
    - Ce sont les features minimales à avoir, vous êtes libres d'en ajouter
    - Voici quelques références de jeux pouvant grandement vous inspirer dans votre développement : Red Trigger, AntiChamber, Portal 2, The Talos Principle, The Turing Test, Quantum Conundrum, Metamorphic, QUBE 2, Glitchspace, Anamorphosis, Laser Grid, Lightmatter, NaissanceE, DeadCore, Boxed, The Spectrum Retreat, Paradox Wrench, Pulse Shift, ChromaGun, Cloudbase Prime, Colortone
- **Une fois que vous aurez bien en tête votre jeu et ses mécaniques** il faudra penser aux outils permettant de créer le plus efficacement des niveaux

Tâche 8 : Son

Intégrer l'API audio de votre choix *(OpenAl, PortAudio, etc.)* à votre moteur de façon à gérer les points suivants :

- Prendre en charge le format ogg
- Lire un fichier audio *(play/pause/stop, en lecture simple ou en boucle)* à l'exécution du jeu *(via un script)* et également en mode éditeur
- Gérer la spatialisation du son 2D et 3D
- Concevoir une interface de mixage des volumes sonores *(master et sfx)* dans l'éditeur
- Ré-échantillonner le son *(192 kHz, 64 kHz, etc.)*
- Choisir son mode d'import *(decompress on load, decompress on the fly)*

Les fichiers audio sont des ressources comme le graph. Ils doivent être pris en charge par votre gestionnaire de ressources.

Tâche 9 : Animation & Editeur

Intégrer le SDK FBX à votre moteur afin de gérer les points suivants :

- Charger/décharger un .fbx
- Extraire les informations du fichier *(bones, animations, materials)* et les stocker dans des structures de données adaptées
- Utiliser les différentes animations d'un modèle au sein du jeu
- Visualiser dans l'éditeur un fbx et la liste de ses animations existantes.
- Jouer une animation sélectionnée *(play/pause/stop)* à l'exécution du jeu *(via un script)* et également en mode éditeur
- Permettre la navigation dans les keyFrames de l'animation
- Choisir une vitesse d'animation *(15 fps, 30 fps, 60 fps, etc.)*

REMARQUES

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.

Compétences évaluées

ALPHA

GESTION DE PROJET & MANAGEMENT	GESTION DE PROJET & MANAGEMENT	INFORMATIQUE GENERALE
Management : 40%	Présentation : 30%	Moteur & Editeur de jeux : 30%

BETA

GESTION DE PROJET & MANAGEMENT	INFORMATIQUE GENERALE
Management : 30%	Moteur & Editeur de jeux : 30%

GOLD

GESTION DE PROJET & MANAGEMENT	INFORMATIQUE GENERALE
Management : 30%	Moteur & Editeur de jeux : 40%

JURY

GESTION DE PROJET & MANAGEMENT	ANGLAIS PROFESSIONNEL
Présentation : 70%	Anglais professionnel : 25%