Compilation et Documentation

Source du projet : https://github.com/TheSpyGeek/VoxelEngine

Attention pour cloner utiliser l'option "recursive" de git : `git clone --recursive https://github.com/TheSpyGeek/VoxelEngine.git`

Compilation

Le projet utilise `cmake` pour le développement nous utilisons le flags type de compilation en mode `Debug` mais pour une utilisation est pour voir les performances il vaut mieux utiliser le mode `Release`. Pour ce faire , lors de la préparation avec `Cmake` au lieu de juste faire `cmake ..` vous devez faire `cmake -DCMAKE_BUILD_TYPE=Release ..`.

Le projet peut être compilé sur Linux, Windows et normalement MacOSX.

Sur Linux

Dépendances :

```
sudo apt-get install -y build-essential cmake xorg-dev libgl1-mesa-dev libfreetype6-dev
```

Pour compiler :

```
mkdir build && cd build && cmake .. && make
```

Sur Windows

Vous aurez besoin de Cmake (lien téléchargement: https://cmake.org/download/)

Avec MinGW

Vous devez avoir Mingw (lien téléchargement : https://sourceforge.net/projects/mingw-w64/)

Pour compiler vous devez faire:

- Créer un dossier `build` dans le dossier du projet
- Lancer `Cmake`
- Configurer avec `Mingw Makefile`
- Lancer le terminal `mingw64`
- Aller dans le dossier `build`
- Compiler avec la commande `mingw32-make`

Avec Microsoft Visual Studio 2017

Pour compiler vous devez faire:

- Créer un dossier `build` dans le dossier du projet
- Lancer `Cmake`
- Configurer avec `Visual Studio 15 2017`
- Ouvrir le fichier `ALL_BUILD.vcxproj`
- Dans Microsoft Visual Studio 2017 faire clic droit sur "ALL_BUILD" puis "Générer"
- Cela va compiler le projet
- Vous allez déplacer l'executable dans le dossier `build` que vous avez créer

Vous pouvez lancer l'executable

Documentation

La game loop est présente dans le fichier `main.cpp`. Elle appelle plusieurs classes qui votn s'occuper de la mise à jour des objets et du rendu.

Classes importantes (la plupart présent dans le dossier `engineClass`) :

- **Scene** : gère tous les objets dans la scène. C'est cette classe qui va appeler le méthode de mise à jour des objets et des méthodes de dessins des objets.
- **MainRenderer** : C'est lui qui s'occupe de la configuration d'`OpenGL`, qui met les bonnes options et qui décide quoi afficher. Selon si le moteur est en mode "Jeu" ou non il va afficher des choses différentes. C'est le mode "Editeur" et le mode "Jeu". On peut basculé de l'un à l'autre avec `CTRL + U`.
- **UI** : gère tout l'affichage des w idgets sur l'écran (hors rendu). Le méthode d'affichage de chaque composant d'un objet sont défini dans les objet eux mêmes. Elle affiche l'arborescence de la scène à droite de l'écran, les menus déroulant, etc.
- **InputManager** : cette classe gère les entrées clavier au niveau du moteur. Exemple: mettre en pause le jeu, passer en mode w ireframe, etc. Les contrôles du joueur sont gérés à un autre endroit.
- **Transform** : cette classe gère tout ce qui est position, rotation, scale d'un objet. Chaque objet en a un.
- **Gameobject** : Cette objet du moteur qui est géré par la scène. Il possède une liste de `Component` qui vont venir changer le comportement de ce Gameobject. Il possède un nom et un ID.
- **Component** : Ceux sont des classes qui vont modifier le comportement d'un ou plusieurs Gameobject. Ils ont plusieurs fonctions de mise àjour qui sont appelées à des moments différents (`update()`, `inputUpdate()`, `physicsUpdate()`). C'est inspiré de ce qui est fait dans **Unity**

Composants

On va faire un vite tour des **Component** et de ce qu'ils font : (dans le dossier `components` et `terrain`)

- Les **Renderer** : (`AxisRenderer`, `CameraRenderer`, `MeshRenderer`) ils font tous les trois un rendu `OpenGL`, le plus important des trois est le `MeshRenderer` qui fait le rendu d'un maillage avec les informations de normales, position, etc.
- **Collider** : il défini un box collider de type AABB (Axis-aligned bounding boxes) il fait la detection de collision **mais pour l'instant qu'avec le terrain**
- **Rigidbody** : il permet de faire les déplacements de l'objet dependant de la vitesse du `Rigidbody`.
- **CameraFollow** : defini le comportement d'une camera qui suit un objet à une certaine distance.
- **GroundFollow** : permet à l'objet de rester toujours à la surface du terrain
- **CameraControllerFirstPerson** : defini le comportement d'une caméra e vue à la première personne. Elle est utilisé pour la caméra de l'éditeur.
- **CameraProjective** : permet d'avoir les informations de projection, c'est ce composant qui donne la matrice 4x4 Projection;
- **ThirdPersonController** : permet de controller la caméra du mode "Jeu" avec la souris. La caméra va tourner autour du joueur.
- **FireProjectiles** : permet de tirer des projectiles
- **Projectile** : Défini le comportement d'un projectile. Dans notre cas, c'est un projectile qui va faire une explosion à l'impact
- **TerrainModifier** : c'est lui va modifier le terrain quand il y a des explosions
- **TerrainManager** : C'est lui qui est responsable de la génération et de l'affichage du terrain. C'est lui qui gère quels `chunk` doit être afficher et lequels modifier
- **TerraiChunk** : C'est lui qui contient les informations utiles à un `chunk`. Chaque `chunk` en ont un.