Travail pratique #05

En utilisant votre tp04 comme point de départ, veuillez effectuer les modifications décrites dans ce document.

C'est le dernier travail de la session et au terme de celui-ci votre jeu aura atteint l'objectif que nous nous étions fixé au début de la session qui était de créer un clone jouable **inspiré** de minecraft.

Objectifs du travail

1. Ajout, suppression et mise à jour de structures de données complexes
2. Génération de données en utilisant une fonction pseudo-aléatoire basée sur le bruit de perlin
3. Mettre en pratique les notions vues en classe pour déterminer de façon autonome le meilleur moyen de solutionner les problèmes demandés

Règles importantes

1. Chaque classe doit avoir son propre fichier *.h* et *.cpp*
2. N'oubliez pas les *"include guard"*
3. Prenez grand soin de respecter la casse pour les noms de classe, méthode, etc
4. Les règles de l'encapsulation doivent être respectées au maximum, seulement ce qui doit absolument être public peut l'être
5. Le code doit compiler sous visual studios 2022 (ou g++ sous linux)
6. Veuillez commenter votre code *intelligemment*
7. **Il est primordial de soumettre votre code régulièrement sur git, et cet aspect sera évalué pour ce travail ainsi que les subséquents**
8. **Votre code doit compiler sans erreurs pour être corrigé**

Destruction et création de blocs

Une des principales forces de minecraft est qu'il est possible de transformer le monde à sa guise, en enlevant et en ajoutant des blocs. Ce type de jeu est appelé un jeu 'sandbox' et apporte un très haut de rejouabilité et de flexibilité.

Le bouton gauche de la souris sert à détruire le bloc qui est devant le joueur (pointé par le *crosshair*). La destruction d'un bloc implique de pouvoir déterminer quel bloc est pointé par le joueur, et ensuite selon sa durabilité de éventuellement le détruire. Vous pouvez si vous le désirez ne pas tenir compte de la durabilité des blocs et assumer qu'ils se détruisent en un seul coup (points bonus pour ceux qui tiendront compte de la durabilité, c'est à dire qu'il faut maintenir le bouton gauche de la souris enfoncé sur un bloc pour le détruire jusqu'à ce que sa durabilité tombe égale à zéro). Une fois le bloc détruit, il disparait et le mesh du chunk est regénéré.

Le code permettant de déterminer quel bloc est pointé par le crosshair est fourni avec ce TP dans le fichier *GetBlocAtCursor.txt* sur vortex.

Deux méthodes nécessaires par ce code (EqualWithEpsilon et InRangeWithEpsilon) sont fournies dans le fichier *Epsilon.txt*. Elles permettent de faire des comparaisons de nombre en virgule flottante en tenant compte d'une marge d'erreur (l'epsilon).

Faites du bruit !

Pour générer le monde virtuel vous aurez besoin d'utiliser une fonction pseudo-aléatoire. Une bonne fonction pseudo-aléatoire doit pouvoir fournir les caractéristiques suivantes :

1. Il doit être possible d'initialiser la fonction de génération de nombre aléatoire avec une valeur initiale (seed), un peu comme si on "brassait les cartes".
2. Elle doit être déterministe, c'est à dire que pour un seed donné, la séquence de valeur retournée par la fonction aléatoire doit toujours être la même. Cela permettra qu'un chunk puisse être regénéré au besoin et être identique chaque fois que le joueur revient au même endroit (en utilisant par exemple la position du chunk dans le monde comme seed).
3. Se rapprocher le plus possible des caractéristique du *vrai* hasard. Les valeurs retournées doivent être uniformément distribuées sur leur domaine de valeur.

Il existe plusieurs fonctions pseudo-aléatoire qui respectent ces critères, la plupart sont basées sur des opérations mathémathiques relativement simples. Il existe par exemple la classe Random en C# ou la fonction rand() en C ou C++. Malheureusement une fonction de ce type n'est pas idéale pour générer un monde de jeu avec un terrain possédant des montagnes, collines, vallées, rivières, etc. Une fonction aléatoire idéale serait une fonction qui retourne des valeurs rapprochées l'une de l'autre quand on lui fourni en entrée des valeurs proches, pour éviter les trop grand écarts qui donneraient une appararence moins naturelle.

Nous utiliserons l'algorithme du bruit de perlin, idéal pour notre situation.

Premièrement un peu de lecture pour introduire le sujet :

Un article écrit par Notch, le créateur de minecraft (Terrain generation, Part 1) : `https://n0tch-blog-blog.tumblr.com/post/4231184692/terrain-generation-part-1`

Perlin Noise, de wikipedia : `http://en.wikipedia.org/wiki/Perlin_noise`

Le code est fourni dans les fichiers perlin.cpp et perlin.h. **Note :** Ce code a été fait par le créateur de l'algorithme lui-même (Ken Perlin)

Vous devez utiliser ce code pour générer le terrain, et vous devrez y ajouter votre touche personnelle pour ajouter des mineraux si désiré, ajouter des arbres, cavernes, etc. **Pour ce travail vous devez générer un terrain montagneux comportant au minimum 3 types de blocs différents**. La génération du terrain doit se faire dans le constructeur du chunk.

Important : Si le joueur modifie un chunk (en ajoutant ou enlevant un bloc) il ne faut pas oublier de sauvegarder le chunk modifié sur le disque quand il se fait détruire (dans son destructeur...) pour s'assurer que ses changements sont préservés. Une étape de plus s'ajoute à la création d'un chunk : vous devez vérifier si le chunk a déjà été sauvegardé, et si oui, le charger du disque plutôt que de le regénérer. **Pour des raisons d'espace de stockage et d'efficacité, il ne faut pas sauvegarder tout les chunks, seulement ceux ayant subit des modifications**.

Voici un exemple d'utilisation du code de perlin :

Listing 1 – Création d'un objet de type Perlin

```
// The first parameter is the number of octaves, this is how noisy or smooth the function is. This is valid between 1 and 16. A value of
      4 to 8 octaves produces fairly conventional noise results. The second parameter is the noise frequency. Values betwen 1 and 8 are
      reasonable here. You can try sampling the data and plotting it to the screen to see what numbers you like. The last parameter is
      the amplitude. Setting this to a value of 1 will return randomized samples between -1 and +1. The last parameter is the random
      number seed.
Perlin perlin(16,6,1,95);
```

Listing 2 – Création du terrain dans le constructeur de Chunk

```
for(int x = 0; x < CHUNK_SIZE_X; ++x)
{
    for(int z = 0; z < CHUNK_SIZE_Z; ++z)
    {
        // La méthode Get accepte deux paramêtre (coordonnée en X et Z) et retourne une valeur qui respecte
        // les valeurs utilisées lors de la création de l'objet Perlin
        // La valeur retournée est entre -1 et 1
        float val = perlin.Get((float)(m_posX * CHUNK_SIZE_X + x) / 2000.f, (float)(m_posZ * CHUNK_SIZE_Z + z) / 2000.f);

        // Utiliser val pour déterminer la hauteur du terrain à la position x,z
        // Vous devez vous assurer que la hauteur ne dépasse pas CHUNK_SIZE_Y
        // Remplir les blocs du bas du terrain jusqu'à la hauteur calculée.
        // N'hésitez pas à jouer avec la valeur retournée pour obtenir un résultat qui vous semble satisfaisant
    }
}
```

Au choix

Si vous aviez des rêves ou des aspirations pour votre jeu en début de session, c'est maintenant le temps de les réaliser. Rendu à ce point dans la séquence de TP, la base du jeu est maintenant en place et il est temps d'y ajouter des éléments qui le distinguera de celui de vos collègue et peut-être même du minecraft original.

Vous devez implémenter **au minimum un élément au choix** parmis la liste suivante. Il est possible de choisir un élément qui ne fait pas partie de cette liste (mais d'ampleur équivalente), **sur approbation de l'enseignant**.

C'est probablement la partie la plus complexe du cours, et il est normal que vous ayez à fouiller par vous-même pour trouver de l'information ou à discuter avec vos collègues et enseignant pour arriver à un bon résultat. N'hésitez pas à mettre en pratique ce que vous avez appris jusqu'à maintenant.

1. Monde infini, il doit être possible de se promener et le jeu doit charger et décharger les chunks visible dynamiquement
2. Inventaire et capacité de 'crafter' des objets
3. Jeu en réseau (gérer au minimum le déplacement de deux joueurs simultanés)
4. Gestion de l'éclairage
5. Système de menu hiérarchique permettant de configurer certains éléments du jeu, de quitter, etc
6. Monstres et combat avec ceux-ci (apparence des monstres et système de combat au choix)
7. Chargement et mise à jour des Chunk en arrière plan dans un thread séparé (en utilisant les threads de SFML)

Présentation de votre projet

Vous devrez présenter votre jeu à vos collègues de classe (correspond au cours de la date de remise). **Ce travail doit être terminé avant la présentation, portez une attention particulière à la date de remise sur vortex**. Cette présentation se veut amicale et sans stress, c'est le moment de partager votre réalisation de la session. Pensez à préparer une petite démo vous vous pourrez faire en démonstration en avant pour mettre en valeur les caractéristiques de votre jeu.

Tests

Aucuns tests unitaires ne sont à réaliser pour ce travail, mais vous devez vous assurer que votre jeu respecte **au minimum** ce qui est demandé dans cet énoncé.

Points bonus

Impressionnez-moi ! Des points bonus seront accordés pour les extra de votre cru que vous rajouterez à ce TP. Veuillez décrire dans le fichier texte de la remise les extras que vous avez fait et que vous voulez que je considère pour les points bonus. **Rendu à ce stade du projet, et ce jusqu'à la fin, vous êtes encouragés à personnaliser votre jeu en y ajoutant votre touche personnelle**.

Remise

A remettre sur Vortex :

1. Un fichier texte (.txt) contenant le hash `sha1` (exemple : `8fea0e32a53c59eac95c157fa060e112cf88b7a0`) du dernier commit que vous voulez que je corrige. Assurez-vous d'avoir fait un `push` sur le serveur pour que je puisse voir votre dernière version. **Ne pas remettre votre projet directement sur vortex !**