Travail pratique #01

Il existe plusieurs jeux basés sur les voxels, minecraft est probablement le plus populaires d'entre-eux. Au cours de la session, vous serez amenés à en créer un clone (en plus simple on s'entend !) par une série de travaux pratiques de difficulté incrémentale et permettant de mettre en pratique les notions de POO que nous verrons en classe. Voici le premier d'entre eux !

Le but de ce travail est de créer des classes qui vous seront utiles par la suite pour la réalisation du jeu. Elles serviront de base pour le reste et seront dans certains cas amenées à être améliorées par la suite.

Objectifs du travail

1. Pratiquer les notions de C++ vues jusqu'à maintenant
2. Mettre en oeuvre des classes qui respectent les bases de l'encapsulation
3. Favoriser la réutilisation de code par l'héritage
4. Écrire des jeux de tests

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. Vous devez tester chacunes de vos classes en écrivant un jeu de test qui les utilisent
6. Le code doit compiler sous visual studios 2012
7. Le programme doit fonctionner sur les ordinateurs de notre local de cours
8. Veuillez commenter votre code intelligemment

Pour le moment vous devez créer et tester les classes de ce TP dans une application console.

Voici ce qui est a à implémenter :

Constantes et types de blocs

Le monde du jeu est entièrement constitué de cubes (blocs). Il va donc de soit de garder une liste des types de blocs qui existeront. Pour l'instant nous n'en définierons que trois :

BTYPE_AIR : Un bloc d'air est vu comme une absence de bloc. Tout ce qui n'est pas un bloc physique est de l'air. Ce type de bloc est particulier parce qu'il laisse passer parfaitement la lumière et ne cause pas de collisions.

BTYPE_DIRT : Il s'agit d'un bloc de terre

BTYPE_GRASS : Bloc de gazon, semblable à un simple bloc de terre, mais puisqu'il est exposé à la lumière du gazon a poussé sur le dessus.

Veuillez définir un enum nommé *BlocType* qui contient ces valeurs (dans cet ordre..) dans un fichier nommé *define.h*. N'oubliez pas les *include guard*.

Vous devez ajouter dans ce même fichier trois constantes en utilisant la directive *#define* du préprocesseur nommées CHUNK_SIZE_X, CHUNK_SIZE_Y et CHUNK_SIZE_Z qui possèdent respectivement les valeurs 16, 128, 16.

Description des classes

Classe BlocInfo

Chaque type de bloc possède des caractéristiques différentes et la classe BlocInfo servira pour instancier chaque objet représentant les différents blocs du jeu. Pour l'instant cette classe doit contenir les informations de durabilité (combien de coup sont nécessaire pour briser le bloc), son nom, et son type (un des 3 types décrit plus haut et provenant de define.h).

Votre classe BlocInfo doit posséder au minimum les méthodes publiques ayant les signatures suivantes (attention au respect de la casse) :

Listing 1– Méthodes de BlocInfo

```
BlocInfo(BlocType type, const std::string& nom);
~BlocInfo();

BlocType GetType() const;

void SetDurabilite(int durabilite);
int GetDurabilite() const;

void Afficher() const;
```

La méthode Afficher est responsable d'afficher dans la console le type, le nom et la durabilité du bloc. Rien ne vous empêche de vous créer une ou des méthodes privées qui vous aideront, surtout si ça vous évite de dupliquer du code.

Classe BlocArray3d

Puisque le jeu se déroule en 3 dimensions, nous aurons régulièrement à manipuler des regroupement de bloc en 3d. La classe BlocArray3d représente un tableau en 3 dimensions que vous devez créer sur mesure. À la construction d'un objet de type BlocArray3d, le programmeur passe la taille en x (largeur), en y (hauteur) et en z (profondeur) du tableau 3d. Dans le constructeur, l'objet doit allouer dynamiquement l'espace mémoire nécessaire (en utilisant new pour allouer x * y * z éléments de type BlocType). Il faut libérer la mémoire à la destruction de l'objet.

Vous devez créer un constructeur qui accepte en paramêtre trois entiers de type int nommés dans l'ordre x, y et z pour initialiser la taille du tableau 3d. Le constructeur est responsable d'allouer la mémoire nécessaire qui devra être conservée à l'aide d'un pointeur de type BlocType. Une fois la mémoire allouée, le constructeur initialise le tableau 3d avec des BTYPE_AIR.

Cette classe devra comporter les méthodes publiques suivantes :

Listing 2– Méthodes qui doivent être présentes dans BlocArray3d

```
void Set(int x, int y, int z, BlocType type);
BlocType Get(int x, int y, int z) const;

void Reset(BlocType type);
```

La méthode Set permet d'assigner un type (BlocType) à n'importe quelle case du tableau 3d en spécifiant la position à modifier. La méthode Get va chercher le BlocType associé à la position passée en paramètre et la méthode Reset initialise le tableau 3d en entier avec la valeur passée en paramètre.

Puisque vous avez alloué un tableau interne à une dimension avec new, il est nécessaire de calculer vous-même la position linéaire dans ce tableau selon la position 3d qui est recues. Vous pouvez utiliser la formule suivante :

Listing 3– Accès à une position dans le tableau interne

```
// En supposant que m_x, m_y et m_z soient des variables de classe qui contiennent la taille du tableau:
// m_blocs est le tableau de taille m_x * m_y * m_z alloué dynamiquement dans le constructeur
return m_blocs[x + (z * m_x) + (y * m_z * m_x)];
```

N'oubliez pas de faire un constructeur de copie pour cet objet pour garantir que s'il se fait copier, le contenu interne (surtout le tableau dynamique) se fera copier aussi.

Classe Chunk

Le monde dans lequel le joueur évolue dans un jeu comme minecraft est infini. Rien n'est réellement infini, mais du moins il donne l'impression que c'est le cas. Le truc utilisé est de charger seulement une partie du monde en mémoire à la fois, et de s'assurer que chaque fois que le joueur se déplace seulement la partie visible du monde qui l'entoure est chargé. Pour y arriver, il faut diviser le monde en morceau, qu'il est possible de charger en mémoire et de les décharger sur demande. Nous appelerons ce "morceau de monde" un Chunk. Un chunk est un regroupement de blocs, dont les dimensions ont été définies dans le fichier define.h au début de ce TP. On peut dire que un Chunk est un BlocArray3d, mais avec plus de fonctionnalités.

Vous devez créer votre classe Chunk en la faisait dériver (hériter) de la classe BlocArray3d. Nous n'aurons pas beaucoup de nouvelles fonctionnalités à lui ajouter en ce moment, mais ce sera fait au cours des prochains TPs.

Listing 4– Méthodes à implémenter dans la classe Chunk

```
Chunk();
~Chunk();

void RemoveBloc(int x, int y, int z);
void SetBloc(int x, int y, int z, BlocType type);
BlocType GetBloc(int x, int y, int z);
```

Le constructeur de Chunk est responsable d'appeler le constructeur de la classe parent et de lui fournir les bonnes dimensions. Les dimensions à fournir sont celles de define.h (les 3 CHUNK_SIZE_*).

Les méthodes RemoveBloc, SetBloc et GetBloc acceptent une position en 3 dimensions. La méthode RemoveBloc remplace le bloc à la position donnée par un bloc de type BTYPE_AIR. Ces trois méthodes doivent appeler les méthodes de la classe parent (BlocArray3d) que vous avez créées précédemment.

Tests

Vous devez créer un ensemble de tests (dans main.cpp) qui vérifieront le bon fonctionnement de vos classes. Veuillez laisser les test pour la soumission de votre travail.

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 !**