Travail pratique #03

En utilisant votre tp02 comme point de départ, veuillez effectuer les modifications décrites dans ce document.

Objectifs du travail

1. Utiliser les templates pour rendre du code générique
2. Allocation de mémoire, utilisation de pointeurs
3. Génération dynamique de mesh
4. Approfondir les notions d'OpenGL
5. Utilisation d'un outil de gestion de code source (git)

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**

Avant de commencer...

Un peu de cleanup

Enlever le cube qui tourne, nous n'en aurons plus besoin.

Type de BlockType

Par soucis de simplicité nous avions déclaré BlockType en tant qu'enum dans define.h. En C++, un enum est presque équivalent au type int, ce qui implique que chaque fois que nous stockons un type de block (il y en a 16 * 16 * 128 dans chaque chunk...) 4 octets en mémoire sont réservés. Nous allons voir que la mémoire sera de plus en plus limité à mesure que nous avançons dans le projet, une modification s'impose donc.

Si on part du principe que nous aurons jamais plus de 256 types de blocs différents, il est plus judicieux que le type de *BlockType* soit plutot un *uint8_t*. Veuillez apporter la modification suivante dans votre fichier *define.h* :

Listing 1 – BlockType

```
typedef uint8_t BlockType;
enum BLOCK_TYPE {BTYPE_AIR, BTYPE_DIRT, BTYPE_GRASS};
```

Être ou ne pas être un BlockArray3d ?

Alors qu'il pouvait sembler être une bonne idée dans le précédent TP de faire hériter la classe *Chunk* de la classe *BlockArray3d*, il s'avère que cela nous limitera en plus de compliquer inutilement l'interface du *Chunk*. Vous devez donc enlever l'héritage, un *Chunk* ne sera plus un *BlockArray3d*. Par contre, un *Chunk* devra contenir un *BlockArray3d* comme membre privé.

Plutôt que *Chunk* hérite de *BlockArray3d*, il devra contenir un membre de ce type :

Listing 2 – BlockArray3d dans Chunk

```
BlockArray3d m_blocks;
```

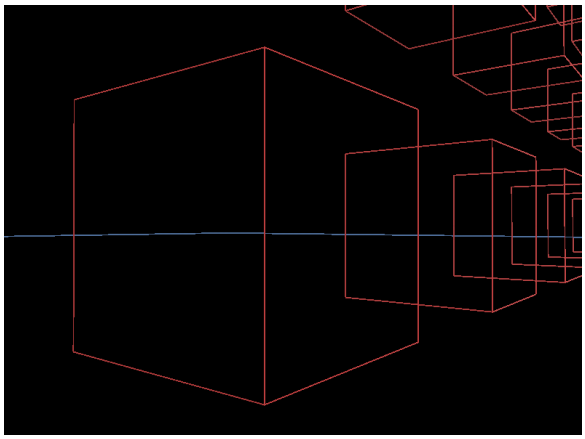
GL_CULL_FACE

Veuillez ajouter la ligne de code suivante dans la méthode *Init* de la classe *Engine*. Cela permettra d'améliorer les performance en disant à OpenGL de ne pas dessiner les faces de polygones qui sont dos à nous. Vous devez bien entendu toujours dessiner vos quads en sens anti-horaire (counter-clockwise) pour que cette fonctionnalité fasse ce que vous désirez.

Listing 3 – Activer le back face culling

```
glEnable(GL_CULL_FACE);
```

Lorsque vous faites des tests, il peut être utile de commenter cette ligne pour éviter que opengl cache des choses si vous faites une erreur dans le sens que vous utilisez. N'oubliez pas de le décommenter pour avoir les meilleures performances (et avant de la remise de ce travail).



GLEW

Selon le site officiel de la librairie GLEW :

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file. GLEW has been tested on a variety of operating systems, including Windows, Linux, Mac OS X, FreeBSD, Irix, and Solaris.

Nous devons installer cette librairie pour utiliser les fonctionnalités plus avancées et plus récentes de OpenGL dont nous aurons besoin dans ce TP (pour les VBOs... les détails sont donnés plus bas). Veuillez ajouter GLEW à votre projet :

1. Veuillez télécharger le fichier glew-1.7.0-win32.zip à partir de vortex ou du site officiel de GLEW

2. À partir du contenu du fichier ZIP :
   (a) Copier le fichier bin/glew32.dll du fichier zip dans votre répertoire Debug
   (b) Créer un répertoire pour GLEW dans votre répertoire external (external/glew170)
   (c) Copier le répertoire include du fichier zip dans le répertoire de glew
   (d) Copier le répertoire lib du fichier zip dans le répertoire de glew
3. Dans Visual Studio, allez dans les propriétés de votre projet (right-click sur le nom du projet dans l'explorateur de solution, ensuite click sur propriétés) et :
   (a) Ajouter external/glew170/include dans "Répertoires VC++ > Répertoires Include"
   (b) Ajouter external/glew170/lib dans "Répertoires VC++ > Répertoires de bibliothèques"
   (c) Ajouter glew32.lib dans "Éditeur de liens > entrée > Dépendances supplémentaires
4. Ajouter la ligne suivante au tout début du fichier define.h (très important de le mettre avant l'include de SFML...)

Listing 4 – Include de GLEW dans define.h

```
#include <GL/glew.h>
```

5. Ajouter le code suivant pour initialiser GLEW, au tout début de votre méthode Init de la classe Engine :

Listing 5 – Initialisation de GLEW

```
GLenum glewErr = glewInit();
if(glewErr != GLEW_OK)
{
    std::cerr << "ERREUR GLEW: " << glewGetErrorString(glewErr) << std::endl;
    abort();
}
```

6. Voila ! Si vous venez qu'à avoir besoin d'ajouter une autre librairie à votre projet, la procédure ci-haut restera la même. À partir de maintenant nous pouvons appeler n'importe quelle fonction de OpenGL, même celles de la version 4.* (à condition d'avoir une carte graphique qui le supporte).

Classe Array3d

Vous devez rendre la classe *BlockArray3d* générique pour qu'elle puisse contenir n'importe quel type de donnée, et pas juste des *BlockType*. Vous devez la renommer *Array3d*, et utiliser les templates.

N'oubliez pas de renommer le fichier *blockarray3d.h* vers *array3d.h* et vous pouvez effacer le fichier *blockarray3d.cpp* (en passant par git...) puisqu'il ne sera plus utile.

La classe *Array3d* doit être **complètement** générique et ne contenir aucuns types spécifiques (tel que *BlockType*). Une fois cette partie faite, modifiez le reste du code pour qu'il utilise maintenant votre nouvelle classe template (dans *chunk.h* et *chunk.cpp*)

Classe Array2d

Même chose que pour la section précédente, mais cette fois pour un array à deux dimensions.

Affichage des chunks

Le but de cette section est de faire afficher un chunk à l'écran, ou plutot tout les blocs qui sont contenus dans un chunk. Nous utiliserons une méthode différente que celle utilisée pour afficher le cube qui tourne du travail précédent.

Mode direct vs les VBOs

La technique utilisée dans le travail précédent est appelée "mode direct" et consiste à faire des appels de fonctions (les gl*) pour spécifier chacuns des vertex, couleur, coordonnées de textures, etc qui seront envoyés à la carte graphique. Bien que simple, cette technique n'est pas envisageable dans un jeu tel que le notre, surtout pour des raisons de performance. Les multiples appels de fonctions taxent le processeur et ralentissent la vitesse à laquelle notre géométrie est envoyée vers le GPU.

De plus, le mode direct a été *deprecated* depuis OpenGL version 3.0, et est complètement disparu depuis OpenGL 3.1. La nouvelle technique (nouvelle... existe depuis OpenGL 1.x quand même) consiste à utiliser des Vertex Buffer Objects (VBO) pour y mettre toutes les informations sur nos cubes (position du vertex, intensité de lumière, coordonnée de texture, etc) pour ensuite envoyer ces informations à la carte graphique toutes d'un coup. Les VBOs sont des espaces mémoires directement sur la carte vidéo (c'est possible de le spécifier autrement...) et la performance est optimale.

C'est un peu plus complexe, mais tellement plus flexible. Nous aurons recours à des shaders pour afficher le contenu de chaque VBO.

Vocabulaire :

Shader Programme qui peut être exécuté directement par le GPU d'une carte graphique. Comme dans le cas d'un programme conventionnel, un shader est composé d'un code source, qui doit être compilé avant de pouvoir être exécuter par le GPU. La compilation d'un shader se fait à partir de OpenGL, et c'est le driver de la carte vidéo qui le compile. La ressemblance avec un programme conventionnel s'arrête là. Nous utiliserons les shaders GLSL dont la syntaxe ressemble beaucoup au C++ (avec beaucoup moins de fonctionnalités). Il est possible d'utiliser les shaders pour faire toute sorte d'effets. Il est possible d'activer un shader uniquement pour une partie du rendu graphique (dans notre cas pour les chunks seulement). Nous devons toujours utiliser deux shaders à la fois, un fragment shader (parfois appelé pixel shader) et un vertex shader. Voir les liens pour plus de détails.

Mesh Un mesh est un ensemble de vertex qui représente un objet 3D. Chacun des chunks de notre jeu aura son propre mesh. Le mesh devra être regénéré chaque fois qu'on modifie un chunk (ajout d'un block, destruction d'un block, changement de la luminosité, etc). C'est ce mesh qui sera stocké dans un VBO et envoyé directement à la carte graphique.

Code fourni

Sur vortex vous trouverez ces fichiers à ajouter à votre solution : *shader.{h,cpp}*, *vertexbuffer.{h,cpp}* et *tool.{h,cpp}*.

La classe *Shader* sert à charger le code source d'un shader à partir du disque dur, à le compiler et à pouvoir l'utiliser pour le rendu de nos chunks. La classe *VertexBuffer* est un objet qui contiendra le mesh de chacun des chunk (besoin de un dans chacun des *Chunk*). Le code qui gère les VBOs est dans cette classe. **SVP prendre le temps de lire attentivement ce code et à fouiller dans la documentation de OpenGL pour apprendre comment il fonctionne.**

Vous avez aussi de fourni les deux shader nécessaires (le fragment shader et le vertex shader, *shader01.frag* et *shader01.vert* respectivement) que vous devez mettre dans le répertoire *media/shader* de votre projet.

Initialisation des shaders

L'étape du chargement, compilation et utilisation des shaders doit se faire dans la classe *Engine*, en utilisant un objet de type *Shader* défini dans *shader.h*.

Avant toute chose, n'oubliez pas d'ajouter un objet de type *Shader* dans la classe *Engine*.

Ensuite ajoutez un define dans le fichier *define.h*, un peu comme nous avions fait pour les textures :

Listing 6 – Chemin vers les shaders

```
#define SHADER_PATH        "../mcclone/media/shaders/"
```

Dans la méthode *LoadResource* de votre classe *Engine*, vous devez maintenant charger les shaders en utilisant ce code :

Listing 7 – Initialisation des shaders

```
std::cout << "Loading and compiling shaders..." << std::endl;
if(!m_shader01.Load(SHADER_PATH "shader01.vert", SHADER_PATH "shader01.frag", true))
{
    std::cout << "Failed to load shader" << std::endl;
    exit(1);
}
```

Changements dans la classe Chunk

Vous devez ajouter un objet de type *VertexBuffer* dans la classe *Chunk*.

Veuillez ajouter au minimum les méthodes suivantes dans cette classe :

Listing 8 – Méthodes à ajouter dans la classe Chunk

```
void Update();
void Render() const;
bool IsDirty() const;
```

La méthode *Update* sert à construire le mesh représentant les blocks du chunk. Cette méthode parcours tout les blocs du chunk, et pour chacun des blocs rencontrés qui ne sont pas des blocs d'air ($BTYPE\_AIR$) vous devez ajouter les informations à propos des vertex qui le compose. Dans la méthode *Update*, le mesh doit être regénéré seulement si le contenu du chunk a changé depuis la dernière fois qu'il a été généré. Pour se faire, vous aurez à maintenir un booléen qui indique si un bloc a été enlevé, ajouté, etc (en le mettant à *true* dans les méthodes *RemoveBlock* et *SetBlock*) et à *false* une fois le mesh généré. Pour le moment on ne se préoccupe pas du type de bloc, et n'importe quelle texture peut être utilisée.

La méthode *Render* demande à la carte graphique d'afficher le mesh du chunk, en appelant simplement la méthode *Render* du *VertexBuffer* contenu dans le chunk.

La méthode *IsDirty* retourne un booléen qui permet de savoir si le contenu du chunk a changé depuis la dernière fois que le mesh du chunk a été généré.

Pour avoir un chunk qui s'affiche comme dans la copie d'écran présentée dans ce document, veuillez ajouter le code suivant dans le constructeur de la classe *Chunk* :

Listing 9 – Chunk de test

```
m_blocks.Reset(BTYPE_AIR);

for(int x = 0; x < CHUNK_SIZE_X; ++x)
{
    for(int z = 0; z < CHUNK_SIZE_Z; ++z)
    {
        for(int y = 0; y < 32; ++y)
        {
            if(x % 2 == 0 && y % 2 == 0 && z % 2 == 0)
                SetBlock(x, y, z, BTYPE_DIRT);
        }
    }
}
```

Je vous fournis le code de la méthode *Update* ci-dessous, qu'il vous restera à compléter pour afficher les autres faces des blocs :

Listing 10 – Exemple de code

```
void Chunk::Update()
{
    // Update mesh
    if(m_isDirty)
    {
        int maxVertexCount = (CHUNK_SIZE_X * CHUNK_SIZE_Y * CHUNK_SIZE_Z) * (6 * 4);
        VertexBuffer::VertexData* vd = new VertexBuffer::VertexData[maxVertexCount];
        int count = 0;

        for(int x = 0; x < CHUNK_SIZE_X; ++x)
        {
            for(int z = 0; z < CHUNK_SIZE_Z; ++z)
            {
                for(int y = 0; y < CHUNK_SIZE_Y; ++y)
                {
                    if(count > USHRT_MAX)
                        break;

                    BlockType bt = GetBlock(x, y, z);

                    if(bt != BTYPE_AIR)
                    {
                        AddBlockToMesh(vd, count, bt, x, y, z);
                    }
                }
            }
        }

        if(count > USHRT_MAX)
        {
            count = USHRT_MAX;
            std::cout << "[Chunk::Update] Chunk data truncaned, too much vertices to have a 16bit index" << std::endl;
        }

        m_vertexBuffer.SetMeshData(vd, count);
        delete [] vd;
    }


    m_isDirty = false;
}


void Chunk::AddBlockToMesh(VertexBuffer::VertexData* vd, int& count, BlockType bt, int x, int y, int z)
{
    // front
    vd[count++] = VertexBuffer::VertexData(x - .5f, y - .5f, z + .5f, 1.f, 1.f, 1.f, 0.f, 0.f);
    vd[count++] = VertexBuffer::VertexData(x + .5f, y - .5f, z + .5f, 1.f, 1.f, 1.f, 1.f, 0.f);
    vd[count++] = VertexBuffer::VertexData(x + .5f, y + .5f, z + .5f, 1.f, 1.f, 1.f, 1.f, 1.f);
    vd[count++] = VertexBuffer::VertexData(x - .5f, y + .5f, z + .5f, 1.f, 1.f, 1.f, 0.f, 1.f);

    // Continuer le code pour afficher les autres faces du cube...
}
```

Affichage !

Pour les besoins de ce TP, vous devez créer un seul objet de type *Chunk* dans votre classe *Engine* et vous occuper de générer son mesh, et de le faire afficher :

Listing 11 – Code à ajouter dans la méthode Render de la classe Engine

```
if(m_testChunk.IsDirty())
    m_testChunk.Update();

m_shader01.Use();
m_testChunk.Render();
Shader::Disable();
```

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**.

Références

Vertex Buffer Object (VBO)

`http://en.wikipedia.org/wiki/Vertex_Buffer_Object`
`http://www.opengl.org/wiki/Vertex_Buffer_Object`
`http://www.opengl.org/wiki/Vertex_Specification_Best_Practices`

Shader (GLSL)

`http://en.wikipedia.org/wiki/GLSL`
`http://nehe.gamedev.net/article/glsl_an_introduction/25007/`

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 !**