Travail pratique #04

En utilisant votre tp03 comme point de départ, veuillez effectuer les modifications décrites dans ce document. Alors que dans les tp précédent les directives étaient plus strictes, à partir de maintenant vous avez plus de lattitude quant aux choix d'implémentation des fonctionnalités. Vous devez intégrer les concepts vu en classe lorsque nécessaire **et il est fortement suggéré de valider et discuter de vos choix d'implémentation avec moi au besoin**.

Objectifs du travail

1. Utiliser la surcharge d'opérateur
2. Allocation de mémoire, utilisation de pointeurs
3. Utilisation de structures de données complexes
4. Approfondir les notions d'OpenGL
5. Optimisation de l'utilisation mémoire (simplification de mesh, etc)
6. 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**

Utilisation d'un texture atlas

(Si certains ont déjà implémenté un système semblable (équivalent ou meilleur) dans un des TP précédent, vous pouvez le conserver si vous le désirez plutôt que de réimplémenter ce qui est demandé dans cette section. Dans le doute, veuillez me le demander.)

Un texture atlas est un regroupement de texture dans un seul fichier image. Les textures sont placées sous forme de grille. Le principal avantage de cette technique est de pouvoir faire un seul *Bind* de texture, mais de pouvoir utiliser chaque petite texture juste en modifiant les coordonnées u et v. Les bind de texture étant couteux en temps GPU, il vaut mieux en faire le moins possible est c'est pourquoi toutes les textures qui seront utilisées pour les blocs seront stockées dans le même atlas.

Le code est fourni, veuillez intégrer les fichiers textureatlas.h et textureatlas.cpp dans votre projet. **Ce code est à lire et comprendre.**

L'atlas est construit à l'exécution, et il utilise la librairie Devil (aussi utilisée dans la classe Texture que vous utilisez depuis le tp02) pour faire les manipulations d'images (resize, copy, etc). Une fois le fichier image de l'atlas créé en mémoire, il est associé à une texture Opengl et peut être utilisée comme n'importe quelle texture.

Pour initialiser l'atlas (dans la méthode LoadResource de la classe Engine) il faut appelé la méthode AddTexture pour chacune des textures à ajouter dans celui-ci. Une fois cette étape complétée, il faut appeler la méthode Generate, qui va changer une à une les texture fournies et construire l'atlas. A cette

étape vous devez spécifier la taille de chacune des texture de l'atlas, qui seront redimensionnées pour respecter votre taille. Une dimension de 128 pixel maximum est une valeur raisonnable. Le deuxième paramètre de la méthode Generate doit être mis à false parce qu'il fait référence à une fonctionnalité pas encore complétée.

Listing 1 – Initialisation du TextureAtlas

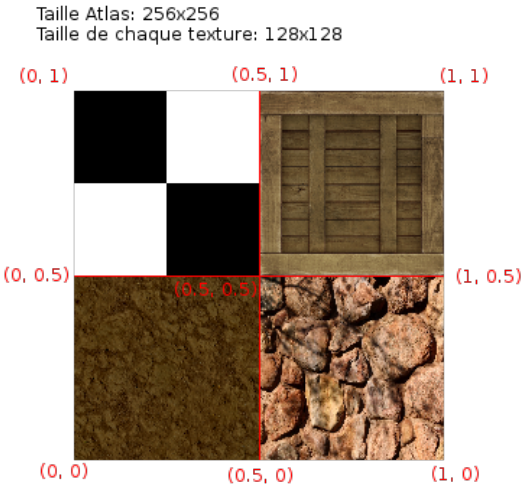
```
TextureAtlas::TextureIndex texCheckerIndex = m_textureAtlas.AddTexture(TEXTURE_PATH "checker.bmp");
TextureAtlas::TextureIndex texDirtIndex = m_textureAtlas.AddTexture(TEXTURE_PATH "dirt.png");

if(!m_textureAtlas.Generate(128, false))
{
    std::cout << "Unable to generate texture atlas..." << std::endl;
    abort();
}
```

Chaque appel à la méthode AddTexture retourne un identificateur de type TextureAtlas : :TextureIndex (c'est un typedef pour unsigned int) que vous pouvez passer à la méthode TextureIndexToCoord pour recevoir les coordonnées à utiliser pour accéder à cette texture dans l'atlas (u, v, largeur, hauteur). Ce sont ces coordonnées que vous utiliserez lorsque vous génèrerez votre chunk.

Finalement, pour utiliser l'atlas, juste avant de dessiner vos chunk, vous devez appeler la méthode Bind :

Listing 2 – Appel de Bind sur le texture atlas

```
m_textureAtlas.Bind();
// Appeler Render sur chaque chunk...
//...
```



Il existe un moyen beaucoup plus flexible et commode qui permet lui aussi de réduire le nombre de bind de texture à faire, mais cette technique (texture 2d array) implique quelques autres modifications, dont au shader. Si le coeur vous en dit vous pouvez décider de l'utiliser (ne pas oublier de le mettre dans votre liste de bonus).

HUD

En direct de Wikipedia :

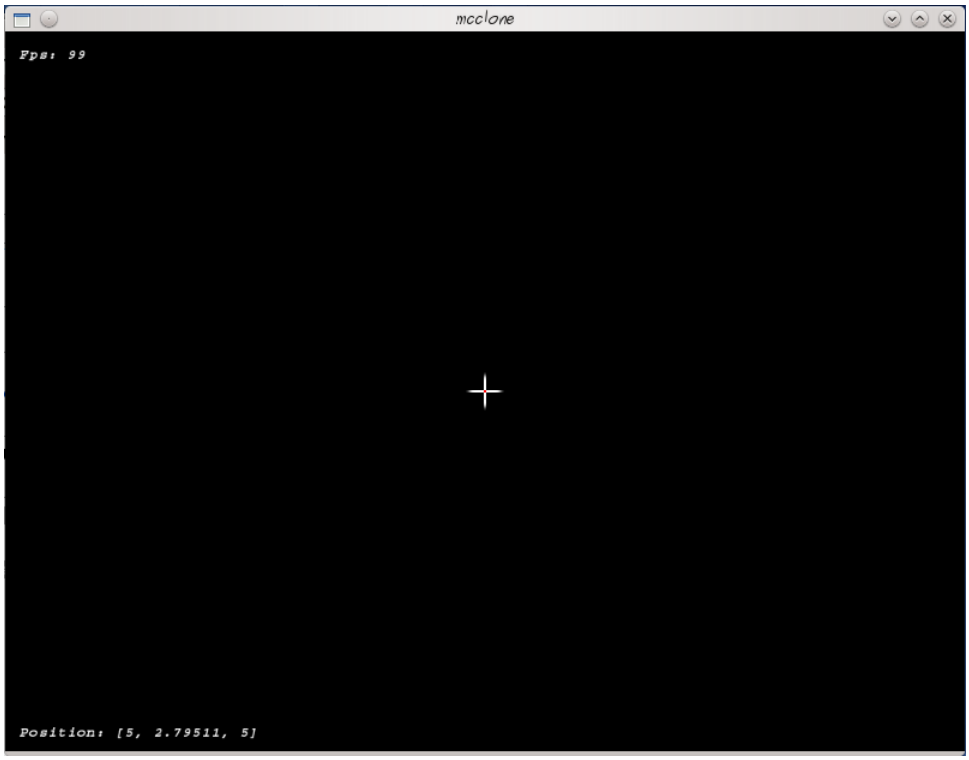
In video gaming, the HUD (heads-up display) is the method by which information is visually relayed to the player as part of a game's user interface. It takes its name from the head-up displays used in modern aircraft. The HUD is frequently used to simultaneously display several pieces of information including the main character's health, items, and an indication of game progression (such as score or level).

Du point de vue de notre jeu, nous utiliserons le HUD pour afficher des informations en 2 dimensions à l'écran. Les informations à afficher sont du texte de débogage, et la mire du joueur (crosshair).

En vous inspirant du code donné en exemple ci-dessous, vous devez être en mesure de faire afficher au minimum le nombre de FPS (frame par seconde) et position courante du joueur. Il est important que vous lisiez le code fourni pour le comprendre, n'hésitez pas à rechercher les noms de fonction opengl sur google pour lire la documentation officielle. Les deux textures nécessaires sont fournies (le font map et le crosshair).

Il est obligatoire d'utiliser l'opérateur << pour afficher votre vecteur de position (voir exemple de code ci-dessous).

Vous êtes fortement encouragés à afficher plus d'information que vous jugerez utile dans votre développement.



Listing 3 – Méthode DrawHud

```
void Engine::DrawHud()
{
    // Setter le blend function, tout ce qui sera noir sera transparent
    glDisable(GL_LIGHTING);
    glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glEnable(GL_BLEND);

    glDisable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glOrtho(0, Width(), 0, Height(), -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // Bind de la texture pour le font
    m_textureFont.Bind();

    std::ostringstream ss;

    ss << "Fps: " << GetFps();
    PrintText(10, Height() - 25, ss.str());

    ss.str("");
    ss << "Position: " << m_player.Position(); // IMPORTANT: on utilise l'operateur << pour afficher la position
    PrintText(10, 10, ss.str());

    // Affichage du crosshair
    m_textureCrosshair.Bind();
    static const int crossSize = 32;
    glLoadIdentity();
    glTranslated(Width() / 2 - crossSize / 2, Height() / 2 - crossSize / 2, 0);
    glBegin(GL_QUADS);
```

```
    glTexCoord2f(0, 0);
    glVertex2i(0, 0);
    glTexCoord2f(1, 0);
    glVertex2i(crossSize, 0);
    glTexCoord2f(1, 1);
    glVertex2i(crossSize, crossSize);
    glTexCoord2f(0, 1);
    glVertex2i(0, crossSize);
    glEnd();



    glEnable(GL_LIGHTING);
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}
```

Listing 4 – Méthode PrintText

```
void Engine::PrintText(unsigned int x, unsigned int y, const std::string& t)
{
    glLoadIdentity();
    glTranslated(x, y, 0);
    for(unsigned int i=0; i<t.length(); ++i)
    {
        float left = (float)((t[i] - 32) % 16) / 16.0f;
        float top = (float)((t[i] - 32) / 16) / 16.0f;

        top += 0.5f;

        glBegin(GL_QUADS);
        glTexCoord2f(left, 1.0f - top - 0.0625f);
        glVertex2f(0, 0);
        glTexCoord2f(left + 0.0625f, 1.0f - top - 0.0625f);
        glVertex2f(12, 0);
        glTexCoord2f(left + 0.0625f, 1.0f - top);
        glVertex2f(12, 12);
        glTexCoord2f(left, 1.0f - top);
        glVertex2f(0, 12);
        glEnd();

        glTranslated(8, 0, 0);
    }
}
```

Listing 5 – Code à ajouter à la fin de la méthode Render

```
if(m_wireframe)
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
DrawHud();
if(m_wireframe)
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

Gestion des types de blocs

Veuillez ajouter au minimum deux types de blocs de plus, au choix, dans votre enum BLOCK_TYPE et ajoutez une texture pour chacun des types de bloc dans votre projet.

Jusqu'à maintenant nous nous sommes contenté d'afficher un chunk avec une texture qui s'appliquait à tout les blocs du chunk, peu importe leur type. Vous devez dans cette section du TP instancier **un** objet de type BlockInfo **pour chacun des types de blocs** définis dans l'enum BLOCK_TYPE et y stocker les informations sur le type de bloc en question. En plus des variables déterminées dans un précédent tp (nom, durabilité) vous devez ajouter les informations supplémentaires que vous jugerez pertinente (textures, ...).

Une fois les objets BlockInfo créés, vous devez trouver une façon efficace de pouvoir retrouver facilement l'instance de BlockInfo associée à chaque type de blocs (BTYPE_*). Puisque nos chunk stockent un Array3d de BlockType, il sera primordial de pouvoir à tout moment retrouver l'objet BlockInfo associé pour aller en chercher les paramètre.

Affichage de plusieurs chunks

Vous devez faire les modification qui s'imposent dans votre jeu pour pouvoir afficher autant de chunk que nécessaire, en respectant la constante VIEW_DISTANCE définie dans le fichier define.h. La constante

VIEW_DISTANCE sert à déterminer combien de block en x et en z qui seront affichés en même temps. Plus ce chiffre est gros, plus le joueur verra loin, et plus votre jeu sera demandant en ressources (temps cpu, mémoire vive et mémoire vidéo). Une valeur aux alentours de 128 est probablement une valeur raisonnable.

Les chunks chargés en mémoire doivent être stockés dans la classe Engine, et vous devez utiliser un $Array2d < Chunk* >$. Les chunks doivent être chargés au début du jeu. Pour ce travail, il n'est pas nécessaire de gérer un monde infini (chunk qui se chargent/déchargent automatiquement quand le joueur se déplace pour simuler un monde infini).

La génération d'un terrain "réaliste" n'est pas nécessaire pour ce TP, mais vos chunks doivent quand même avoir des blocs d'affichés pour être en mesure de tester le déplacement et les collisions.

Optimisation de mesh

Dans le but de préserver la mémoire vidéo de la carte graphique et d'augmenter les performances (nombre de FPS) de votre jeu, vous devez simplifier et optimiser le mesh de chaque chunk au maximum. Vous devez :

1. Éviter d'afficher les blocks qui sont complètements enterrés par d'autre blocks
2. Éviter d'afficher les surfaces entre deux blocks adjacent

Une stratégie que vous pourriez utiliser pour faire cette partie est de parcourir les blocs qui ne sont pas de l'air (BTYPE_AIR) dans un chunk, et d'afficher seulement les faces de ce bloc qui sont en contact avec un bloc adjacent qui est de l'air. La quantité de vertex à envoyer à la carte graphique s'en trouvera grandement réduite.

Le mode d'affichage en wireframe (en appuyant sur la touche 'y') vous sera utile.

Déplacement et collision

Il est temps d'ajouter la collision entre le joueur et les blocs affichés. Pour rappel, chaque bloc mesure une unité (1x1x1) d'un mètre. Le joueur est légèrement plus petit que la hauteur de deux blocs superposés et nous allons assumer qu'il mesure 1.7 mètre. Cela veux dire que le joueur dans notre jeu pourra se promener partout, tant qu'il a au moins un espace de 1 bloc de large et deux bloc de haut pour passer. Comme dans le jeu minecraft original, le joueur peut sauter, mais seulement un bloc de haut.

Il existe plusieurs façon de gérer les collisions et vous devez en trouver une que vous jugez optimale et fiable. Dans un jeu comme minecraft, le fait que tout ce qui est affiché est sous forme de cube de même dimension (et de 1x1x1 unité en plus) nous simplifie un peu la vie. Vous connaissez déjà la position du joueur dans le monde (en appelant la méthode Position() de la classe Player), il est possible de trouver facilement la position du joueur par rapport à un chunk, et par la suite de savoir si un bloc est présent à cet endoit (le joueur peut passer uniquement dans les blocs de type BTYPE_AIR).

Pour gérer les collisions, vous aurez à utiliser la classe Vector3 déjà dans votre projet. Lorsque nécessaire, veillez implémenter les méthodes manquantes ou les opérateurs dont vous aurez besoin dans cette classe.

L'interraction entre le joueur et son environnement doit être conforme à ce que nous sommes habitué de voir dans les jeux de type FPS (First-Person-Shooter). Vous devez notament gérer **le saut**, **la gravité** (joueur qui marche et tombe au bout d'une falaise par exemple) et le **glissement sur les murs** (un déplacement en diagonale sur un mur ne bloque pas le joueur, mais il *glisse* sur le mur).

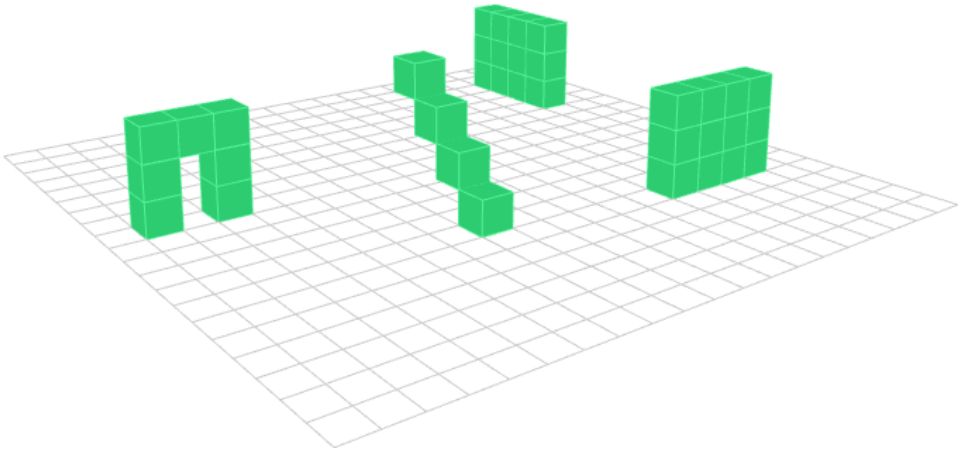
Dans l'incertitude, veuillez me consulter pour discuter de votre méthode.

Terrain de jeu

Pour tester vos collisions, vous devez créer (dans le code) au minimum les éléments suivants :

1. Un escalier à 4 marches dont chaque marche est 1 bloc plus haut que la marche précédente
2. Deux murs de 1 bloc de profond, 4 blocs de large, et 3 blocs de haut, chaque mur étant orienté dans une direction différente (un d'aligné sur l'axe des X, l'autre sur l'axe des Z)
3. Une arche de 3 blocs de large, 3 blocs de haut et un trou de 2 blocs de haut au centre

Vous devez laisser ces structures en place pour la correction



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 !**