RAPPORT

Projet OpenGL, Programmation et Mathématiques

Par Sangaré Laurelenn & Vaillant Margaux

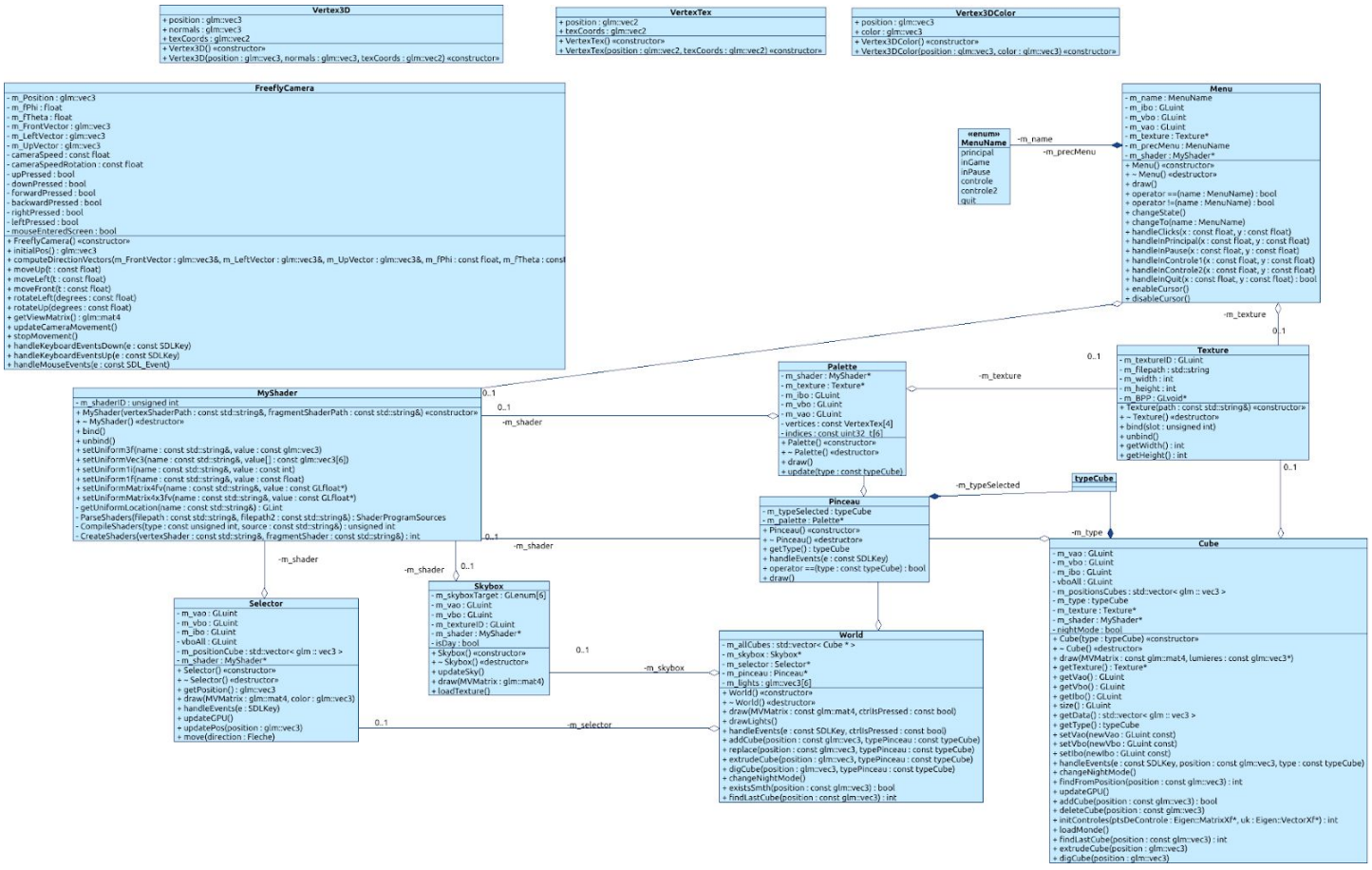
SOMMAIRE

I/ Présentation

Nous savons qu'un générateur de terrain en 3D (type Minecraft) peut avoir une architecture complexe et fouilli si nous cherchons à trop en faire. C'est pourquoi, durant les premières semaines de réflexion, nous avons pris le temps de penser notre architecture de données le plus simplement possible.

Nous avons d'abord voulu la construire en fonction du cahier des charges (héritage, polymorphisme…) mais nous nous sommes rendus compte que si nous voulions développer notre projet efficacement, il fallait que l'architecture nous soit intuitive.

1) Structure globale des données

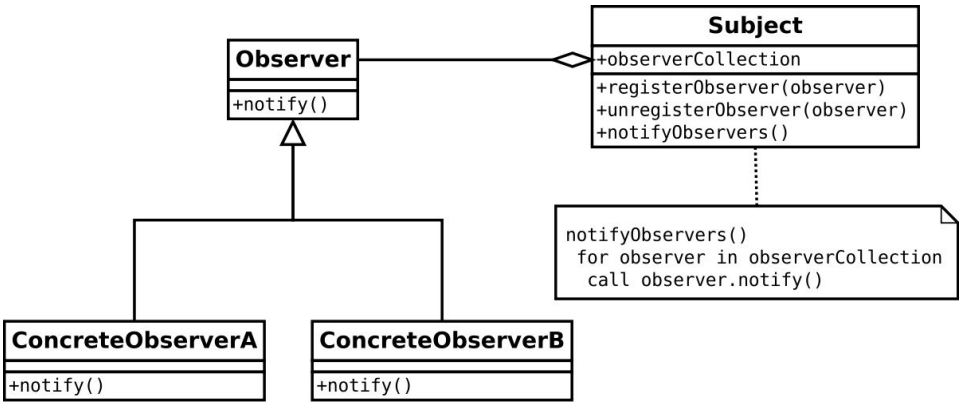


Comme nous pouvons le constater, le diagramme de classes, bien que déjà simplifié, est assez complexe. Il est facile de s'y perdre en le découvrant ainsi, mais cela nous a en réalité permis d'organiser au maximum notre code. Le détail des classes vous sera développé dans la partie suivante *(Voir II.1.),* mais nous pouvons déjà voir les éléments les plus importants :

World, cube, Myshader et texture. Ce sont sans aucun doute les éléments les plus importants de notre jeu et ce qui permet de le faire fonctionner de manière basique. Bien évidemment, les autres classes sont tout aussi importantes, notamment la camera, le menu, ou encore la palette et le pinceau qui permettent de selectionner nos textures. La palette par exemple, permet de sélectionner la texture souhaitée pour un cube. Le pinceau lui, fait le lien entre cette palette et les cubes. *(Voir II.1. pour plus de détail)*

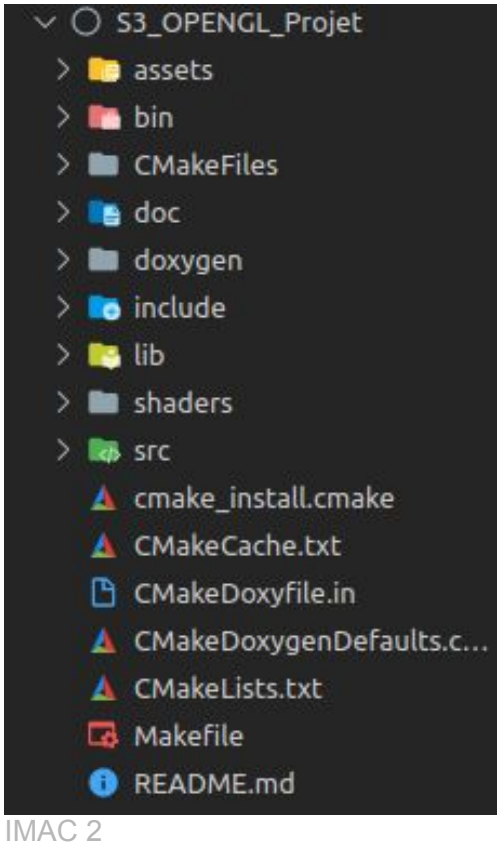
Ce qui est notable également et qui nous a bien simplifié les choses, c'est que nous avons adapté à notre jeu le modèle du **Design Pattern Observer**.

En effet, on peut voir que de nombreuses classes ont une fonction "HandleEvents()" (Camera, Cube, World, Pinceau, Selector, Menu). En effet, lorsque nous lançons l'exécutable, notre main() est continuellement en attente d'événements, et lorsqu'il en a enfin un, il appelle la fonction "HandleEvents() de ces classes afin que chacune gère individuellement l'événement reçu. Elles peuvent choisir de ne rien faire ou au contraire agir en conséquence.



En quoi cela ressemble au Design Pattern Observer ? Eh bien, dans le cas présent, le "Sujet" serait notre main, et les observer seraient chacune des classes ayant la fonction HandleEvent appelée dans ce main.

Il n'y a pas à proprement parlé d'Observer dans notre projet, mais on peut dire que chacune de nos instances sont en attente d'un événement (équivaudrait à notification) pour le traiter.



Concernant la structure de nos fichiers, nous avons choisi une organisation assez traditionnelle.

- **Assets :** Comporte les fichiers liés au développement visuel (textures, images) et sonore (musique).

- **Bin :** Dossier dans lequel se crée notre exécutable

- **Doc :** Dossier dans lequel vous trouverez le sujet du projet, le rapport que vous êtes en train de lire, ainsi que le fichier contenant nos points de contrôles; sans oublier le résumé explicatif sur les RBF que nous avons réalisé. *(Voir III.2.)*

- **Doxygen :** Dossier dans lequel sont rangés tous les éléments utiles à la création de notre documentation. Celle dernière se trouve dans le dossier Doxygen/html.

- **Include :** Nos fichiers header correspondants pour la plupart aux fichiers sources.
- **Lib :** Les librairies que nous avons utilisées (glimac et glm).
- **Shaders :** Dossier dans lequel sont stockés nos fichiers vertex shader et fragment shader.
- **Src :** Dossier dans lequel sont stockés nos fichiers sources dont le main.
- **Dossier Principal :** Dans le dossier principal se trouvent aussi notre README, notre CmakeList, ainsi que les fichiers générés par ce dernier permettant la compilation de notre projet.

2) Choix architecturaux

Nous avons d'abord voulu nous inspirer des tutoriels que certains de nos camarades (Guillaume ou Jules) avaient réalisés,ou encore regarder ce que contenait la librairie glimac mais pour les mêmes raisons que celles évoquées précédemment, nous avons choisi de tout reprendre depuis le début afin d'avoir une architecture de nos fichiers la plus compréhensible possible. En effet, certaines optimisations ne nous semblaient pas évidentes et nous avons privilégié un code simple et que nous comprenions entièrement, à du "bricolage" sur des fichiers pas totalement assimilés.

Nous avons donc décidé d'avoir des tableaux de cubes : un tableau par type de cube. Chaque cube unitaire a une position, des coordonnées de normales, et des coordonnées de texture. Chaque tableau de cube quant à lui, défini un "type" de cube, et attribue donc à chacun de ses cubes une texture. Nous avons également un sélecteur, une skybox et un pinceau. Le pinceau sert à savoir quel type de cube l'utilisateur veut dessiner. Nous nous sommes rendues compte que toutes ses classes, appelées dans le main, se dessinaient au même endroit et géraient les interactions avec l'utilisateur au même endroit.

```
class Cube{
    private:
        GLuint m_vao;
        GLuint m_vbo;
        GLuint m_ibo;
        GLuint vboAll;

        std::vector<glm::vec3> m_positionsCubes;

        typeCube m_type;
        Texture* m_texture;
        MyShader* m_shader;

        bool nightMode;
```

(Extrait de code : Structure d'un groupe de cubes)

Ce qui est important de retenir au sujet de notre structure, c'est que nous ne manipulons pas nos cubes un par un, mais plutôt les positions sur lesquels les cubes (du type sélectionné) existent. *(Voir partie II.1.)* Plutôt que d'effacer ou ajouter un cube, nous effaçons ou ajoutons l'existence de sa position sur la map. Ainsi, lorsque l'on met à jour l'ensemble du monde, la présence -ou non- d'un cube à la position p est mise à jour aussi.

```
class World{
    private:
        std::vector<Cube*> m_allCubes;
        Skybox* m_skybox;
        Selector* m_selector;
        Pinceau* m_pinceau;
        glm::vec3 m_lights[6];
```

Nous les avons donc toutes abstraites dans une classe "World" qui gère ainsi l'affichage de toutes celles-ci et les évènements. Aussi, World nous permet d'accéder facilement à chaque vecteur de cubes de la map (notamment pour les vérifications de positions).

II/ Fonctionnalités & implémentation

Comme nous avons pu l'expliquer dans la première partie *(Présentation p.2)*, nous avons d'abord voulu nous aligner avec le cahier des charges donné dans le sujet, avant de nous en éloigner petit à petit.

Notre principal objectif était de créer un jeu fonctionnel (en nous basant sur le cahier des charges), puis d'améliorer la jouabilité de celui-ci au fur et à mesure. Voici un tableau non-exhaustif des fonctionnalités en fonction de leur phase de développement, avant de détailler le développement de chacune dans les sous-parties.

Légende :	To Do	Doing	Done & Bug	Done & Work

OpenGL		Programmation *(indications)*		Mathématiques	
Todo	Statut	Todo	Statut	Todo	Statut
Gestion Caméra		Compilation		Radial basis function	
Shader Cubes		Biblio autorisées		Génération procédurale	
Affiches Cubes		Commit clairs		Utilisation de Eigen	
Génération procédurale		Exception throw		Réflexion sur fonctions radiales	
Curseur		Outils STL			
Add cube		Exception erreur système (version...)			
Delete cube		Exception erreur utilisateur			
Extrude cube		Diag classe et architecture			
Dig cube		Documentation & noms variables			
Lumières directionnelles & pt lumiere		Assert erreur prog (div / 0, val null)			
Modif Cubes		Gestion mémoire (pas de fuite)			
Placer lumières		Fonctions lambdas			
FONCTIONNALITES SUPPLEMENTAIRES		Func/var constexpr			
Bloc texturés		Templates			
Différents blocs		Héritage & polymorphisme			
Souris bloquée ds fenêtre pour la caméra		FONCTIONNALITES SUPPLEMENTAIRES			
Menus		inline functions			
Mode Nuit		const parameters			
Musique		const return			
Palette		Destructeurs			
Skybox	Done in gradient	surcharge d'opérateurs			

Fonctionnalités - Bilan rapide			
Pôle	**Développées + Bonus**	**En Cours + Bonus**	**Non-Développées + Bonus**
OpenGL	**12/12 + 8 bonus**	**-**	**-**
Programmation	**10/15 + 5 bonus**	**1/15**	**4/15**
Mathématiques	**4/4**	**-**	**-**

1) Fonctionnalités développées intégralement

- **Fonctionnalités de la caméra**

Afin d'avoir un point de vue subjectif, nous avons appliqué plusieurs contraintes à notre caméra. On la dirige grâce à la souris, qui est piégée dans la scène.

La rotation verticale s'arrête est limitée afin de ne pas faire un tour à 360° sur elle-même. Nous nous arrêtons ainsi comme si nous avions un tête, comme si nous étions un personnage en pouvant regarder au maximum au ciel et à nos pieds. La rotation horizontale par contre n'a pas de limite (même si du point de vue du code, elle s'arrête à une rotation modulo 2PI). Ces rotations sont calculées grâce aux mouvements de la souris.

Nous utilisons xrel et yrel qui calculent la position relative en x et y par rapport à la position de base et d'arrivée de la souris. Nous avons pris soin de comparer les mouvements en x et en y afin que la caméra ne se déplace pas de façon diagonale. La caméra se contrôle également grâce aux touches du clavier (ZQSD, et A et W pour la hauteur).

```
FreeflyCamera::FreeflyCamera(){
    m_Position = initialPos();
    m_fPhi = M_PI;
    m_fTheta = 0.f;
    computeDirectionVectors(m_FrontVector, m_LeftVector, m_UpVector, m_fPhi, m_fTheta);
}

//Méthodes
glm::vec3 FreeflyCamera::initialPos(){
    return glm::vec3(25.f, 1.f, 25.f);
}

void FreeflyCamera::computeDirectionVectors(glm::vec3 &m_FrontVector, glm::vec3 &m_LeftVector, glm::vec3 &m_UpVector, const float m_fPhi, const float m_fTheta){
    m_FrontVector = glm::vec3(
        std::cos(m_fTheta)*std::sin(m_fPhi),
        std::sin(m_fTheta),
        std::cos(m_fTheta)*std::cos(m_fPhi));
    m_LeftVector = glm::vec3(
        std::sin(m_fPhi+M_PI/2.f),
        0.f,
        std::cos(m_fPhi+M_PI/2.f));
    m_UpVector = glm::cross(m_FrontVector, m_LeftVector);
}
```

(Extrait de code : Initialisation de la caméra FreeFly)

- **Les différents types de cubes**

Nous avons huit types de cubes + un de lumière. Quatre sont des textures basiques (pleines), et quatre textures transparentes, semi/transparentes. Notre structure consiste donc à créer un tableau de cubes grâce à un paramètre : le type de cube. Les textures sont ainsi chargées à la création des tableaux. Il a fallu cependant pour les cubes transparents penser à dessiner en premier les éléments opaques de notre scène (cubes opaques et skybox) afin que ceux-ci soient visibles.

```
Texture* Cube::getTexture() const{
    switch(m_type){
        case GRASS:  return new Texture("assets/textures/cubes_textures/grass.png");
        break;
        case WATER :  return new Texture("assets/textures/cubes_textures/water.png");
        break;
        case SAND : return new Texture("assets/textures/cubes_textures/sand.jpg");
        break;
        case LEAF : return new Texture("assets/textures/cubes_textures/leaf.png");
        break;
        case BARBARA : return new Texture("assets/textures/cubes_textures/barbara.png");
        break;
        case JULES : return new Texture("assets/textures/cubes_textures/jules.png");
        break;
        case IMAC : return new Texture("assets/textures/cubes_textures/imac.png");
        break;
        case GROUND :
        default:   return new Texture("assets/textures/cubes_textures/cubeTerre.jpg");
        break;
    }
}
```

- **Ajouter un cube**

Pour ajouter un cube, il faut vérifier qu'aucun cube ne se trouve à la position souhaitée. Depuis notre classe "world", on regarde donc si un cube existe à cette position dans tous les tableaux de cubes, et si non, alors on ajoute un cube du type de notre pinceau. Il suffit simplement d'ajouter la position du cube au vecteur positionsCubes.

```
bool Cube::addCube(const glm::vec3 position){
    int exists = findFromPosition(position);
    bool canAdd = true;
    if(m_type == LIGHT && m_positionsCubes.size() >= 4){
        canAdd = false;
        //On ne peut avoir que 4 lumieres positionnelles
    }
    if(exists == -1 && canAdd){
        m_positionsCubes.push_back(position);
        updateGPU();
        return 1;
    }
    return 0;
}
```

- **Supprimer un cube**

Supprimer le cube et la position de celui-ci aurait été répétitif algorithmiquement, nous avons simplement décidé de faire comme si le cube n'avait plus de position (en la supprimant du vecteur positionsCubes) en l'échangeant avec le dernier élément puis en le supprimant par la fin.

```
void Cube::deleteCube(const glm::vec3 position){
    int index = findFromPosition(position);
    if(index != -1){
        int lastIndex = m_positionsCubes.size() - 1;
        std::swap(m_positionsCubes[index], m_positionsCubes[lastIndex]);
        m_positionsCubes.pop_back();

        updateGPU();
    }
}
```

- **Extrude un cube**

Nous avons compris la fonction extrude comme une fonction permettant de, une fois notre sélecteur sur un cube d'un certain type, ajouter un cube en haut de la tour de cube. Donc si nous sommes sur un carré de terre, extrude va permettre d'ajouter un carré de type "terre" au dessus de celui que nous sélectionnons.

```
/*On a le curseur à un endroit et on veut ajouter un dernier cube en haut de l'endroit où on est
S'il n'y a pas de cube, ça en ajoute un à y=0*/
void Cube::extrudeCube(glm::vec3 position){
    int yMax = findLastCube(position);
    position.y = yMax+1;
    m_positionsCubes.push_back(position);
    updateGPU();
}
```

- **Dig un cube**

De la même manière que "Extrude", dig va enlever un cube du même type que celui sélectionné dans la colonne sélectionnée. Comme expliqué plus bas dans ce rapport *(Voir III. 1)*, nous avons décidé que "Dig" retirait un cube du type sélectionné de la colonne, à condition qu'il ne soit pas le premier de son vecteur.

```
/*On a le curseur à un endroit et on veut enlever le dernier cube en haut de l'endroit où on est*/
void Cube::digCube(glm::vec3 position){
    int yMax = findLastCube(position);
    position.y = yMax;
    deleteCube(position);
    updateGPU();
}
```

- **Remplacer un cube**

Nous avons décidé que pour remplacer un cube, la commande serait la même qu'ajouter, à la simple différence qu'il faut maintenir enfoncée la touche "Ctrl". Cette fonction appelle tout simplement sur le cube existant la fonction deleteCube() suivi de la fonction addCube(). La subtilité est qu'ici, il faut prendre en compte TOUS les types de cubes et pas seulement celui du type sélectionné. C'est là que nous voyons toute l'utilité de la classe World pour la gestion des cubes.

```
void World::replace(const glm::vec3 position, const typeCube typePinceau){
    for(uint i=0; i<m_allCubes.size(); i++){
        m_allCubes[i]->deleteCube(position);
    }
    m_allCubes[typePinceau]->addCube(position);
}
```

- **Les points de contrôles**

Les points de contrôle sont chargés grâce à la lecture du fichier “controles.txt” dans le dossier “doc”. On lie les coordonnées des points qu’on met dans une matrice et leur valeur associée qu’on met dans un vecteur. Avant le début du jeu, la fonction “initControles” est donc appelée. Le monde est ensuite créé à partir de ce fichier qui indique donc la hauteur de chaque endroit de notre carte.

```
const int WORLD_WIDTH_HEIGHT = 50;
const int WORLD_DEPTH = -50;

float distance2D(Eigen::Vector2f pt1, Eigen::Vector2f pt2);
float phi(float a);

int findValue(const int i, const int j, const Eigen::MatrixXf ptsDeControle, const Eigen::VectorXf wk);
Eigen::VectorXf findOmega(const Eigen::MatrixXf ptsDeControle, const Eigen::VectorXf uk);

/*MapCoord contient tous les points de la Map et est donc de dimension N*N avec des duos de coordonnées (x, z)
ptsDeControle est une matrice (2, N) autrement dit avec des coordonnées, et autant de coordonnées qu'on veut*/
Eigen::MatrixXd getValues(const Eigen::MatrixXf ptsDeControle, const Eigen::VectorXf uk);
```

```
void Cube::loadMonde(){
    Eigen::MatrixXd map(WORLD_WIDTH_HEIGHT, WORLD_WIDTH_HEIGHT);

    Eigen::MatrixXf ptsDeControle(1, 2);
    Eigen::VectorXf uk(1);

    initControles(&ptsDeControle, &uk);

    map = getValues(ptsDeControle, uk);
    for(int x=0; x<WORLD_WIDTH_HEIGHT; x++){
        for(int z = 0; z<WORLD_WIDTH_HEIGHT; z++){
            for(int y= WORLD_DEPTH; y <= map(x, z); y++){
                m_positionsCubes.push_back(glm::vec3(x, y, z));
            }
        }
    }
}
```

- **Les menus**

Nous avons bien évidemment pensé à faire un menu au début de notre jeu. Pendant ou avant la partie, on peut consulter les commandes utilisables grâce au mode “en pause” (en appuyant sur Espace). Pour quitter le jeu (grâce à Echap ou la croix), nous avons également un menu qui nous demande si nous voulons

vraiment quitter. Ce menu nous permet également de connaître et gérer le statut du jeu (savoir s'il est in game, en pause, etc…)

Les menus sont donc définis par une position (la totalité de l'écran), et des coordonnées de texture. Nous avons pensé à ajouter une surcharge des opérateurs "==" et "!=". En effet, nous avons un enum qui nous permet de savoir dans quel menu nous sommes, afin de déterminer le menu et la texture à afficher.
Ces enum nous permettent également de définir les zones d'interaction avec les boutons, qui changent en fonction de chaque menu.

Comme nous avons choisi d'avoir la souris invisible et bloquée dans la fenêtre durant la partie, nous avons également deux fonctions nous permettant d'afficher ou masquer notre curseur.

```
class Menu{
    private:
        MenuName m_name;
        GLuint m_ibo;
        GLuint m_vbo;
        GLuint m_vao;
        Texture* m_texture;
        MenuName m_precMenu;
        MyShader* m_shader;

    public :
        Menu();
        ~Menu();

        void draw();

        bool operator==(MenuName name);
        bool operator!=(MenuName name);

        void changeState();
        void changeTo(MenuName name);

        void handleClicks(const float x, const float y);
        void handleInPrincipal(const float x, const float y);
        void handleInPause(const float x, const float y);
        void handleInControle1(const float x, const float y);
        void handleInControle2(const float x, const float y);
        bool handleInQuit(const float x, const float y);

        void enableCursor();
        void disableCursor();
};
```

```
const VertexTex vertices[] = {
    VertexTex(glm::vec2(-1.f, 1.f),
              glm::vec2(0.f, 0.f)),
    VertexTex(glm::vec2(1.f, 1.f),
              glm::vec2(1.f, 0.f)),
    VertexTex(glm::vec2(1.f, -1.f),
              glm::vec2(1.f, 1.f)),
    VertexTex(glm::vec2(-1.f, -1.f),
              glm::vec2(0.f, 1.f))
};

const uint32_t indices[] = {
    0, 1, 2,
    2, 3, 0
};
```

- **La gestion des shaders**

En TD, nous avons toujours utilisé glimac. Cependant, utiliser une librairie veut dire la maîtriser et lire son code afin de comprendre son fonctionnement. Afin de bien assimiler le fonctionnement des shaders, nous avons donc décidé de créer nos propres fichiers. En effet, ceux-ci gèrent la création des shaders ainsi que leur suppression. Nous avons donc des fonctions intermédiaires qui parcourent nos fichiers vs et fs, qui créent nos shaders, et qui mettent en place nos variables uniformes.

Nous avons quatre types de shaders : un pour nos formes 3D (nos cubes), un pour les couleurs unies (le sélecteur), un pour les textures 2D (les menus) et un pour la skybox.

```cpp
struct ShaderProgramSources{
    std::string VertexSource;
    std::string FragmentSource;
};

class MyShader{
private:
    unsigned int m_shaderID;
public:
    MyShader(const std::string& vertexShaderPath, const std::string& fragmentShaderPath);
    ~MyShader();

    void bind() const;
    void unbind() const;

    //Set uniforms
    void setUniform3f(const std::string& name, const glm::vec3 value);
    void setUniform1i(const std::string& name, const int value);
    void setUniform1f(const std::string& name, const float value);
    void setUniformMatrix4fv(const std::string& name, const GLfloat*value);
    void setUniformMatrix4x3fv(const std::string& name, const GLfloat*value);

private:
    GLint getUniformLocation(const std::string& name);
    ShaderProgramSources ParseShaders(const std::string& filepath, const std::string& filepath2);
    unsigned int CompileShaders(const unsigned int type, const std::string& source);
    int CreateShaders(const std::string& vertexShader, const std::string& fragmentShader);
};
```

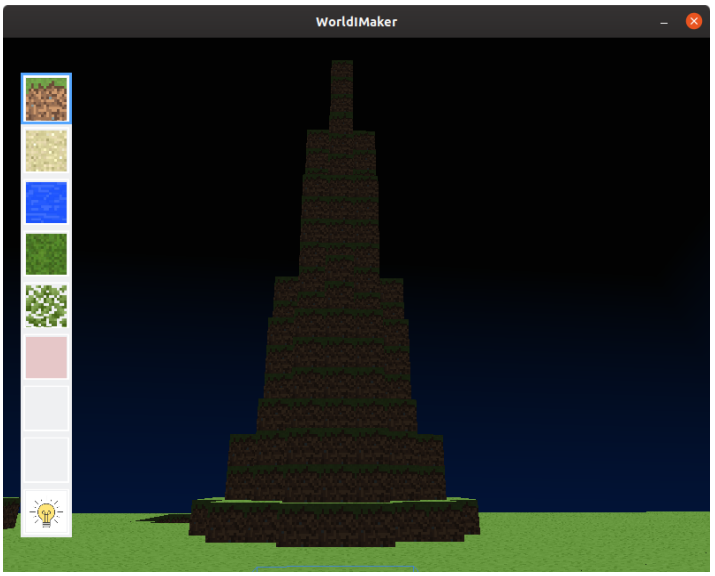
- **La lumière**

Nous avons une lumière ambiante, une lumière directionnelle, et nous pouvons poser des lumières ponctuelles. L'interaction des cubes avec la lumière directionnelle est calculée grâce à un produit entre le vecteur de la lumière et les normales. La lumière ambiante est quant à elle, juste ajoutée uniformément. Nous avons des cubes de lumières qui sont dessinés. On les stocke ensuite dans une variable m_lights qui les transforme en tableau de glm::vec3 qui sera envoyé au shader.

```cpp
void World::drawLights(){
    for(GLuint i=0; i<m_allCubes[LIGHT]->size(); i++){
        m_lights[i] = (m_allCubes[LIGHT]->getData()[i]);
    }
    for(GLuint i=m_allCubes[LIGHT]->size(); i<6; i++){
        m_lights[i] = (glm::vec3(0.f, 0.f, 0.f));
    }
}
```

- **Le mode nuit**

La luminosité ambiante est baissée lorsque nous appuyons sur la touche "N" du clavier : C'est notre mode nuit. La skybox a également à ce moment là un fond différent, qui ressemble plus à un couché de soleil. Nos structures cubes ont un attribut "nightMode" (un booléen). Celui-ci permet de savoir dans quel mode nous sommes et de changer la variable globale "AmbiantLight" après que nous ayons lié notre shader.



- **La palette**

Pour nos interactions utilisateur, nous avons décidé de ne pas utiliser ImGui et de développer notre propre interface. Nous avons donc une palette sur le côté gauche de notre écran qui nous permet de voir quel type de cube nous pouvons poser. Pour sélectionner le type de cube, on utiliser le pad numérique allant de 1 à 9.

- **Le pinceau**

Le pinceau nous sert à savoir quel type de cube l'utilisateur veut dessiner. Il est l'intermédiaire entre la palette et les cubes.

```
void Selector::move(Fleche f){
    glm::vec3 movement = glm::vec3(0., 0., 0.);
    switch(f){
        case gauche: movement = glm::vec3(-1., 0., 0.);
        break;
        case droite: movement = glm::vec3(1., 0., 0.);
        break;
        case haut: movement = glm::vec3(0., 1., 0.);
        break;
        case bas: movement = glm::vec3(0., -1., 0.);
        break;
        case arriere: movement = glm::vec3(0., 0., -1.);
        break;
        case avant: movement = glm::vec3(0., 0., 1.);
        break;
        default:
        break;
    }
    glm::vec3 newPos = m_positionCube[0]+movement;
    updatePos(newPos);
    updateGPU();
}
```

- **Le sélecteur**

Le sélecteur peut être déplacé grâce aux flèches du clavier, et aux touches "O" et "L" pour la profondeur. Il est définit grâce à une position et une couleur. Il est semblable à un cube à la différence près que nous ne voulions que les contours. L'ordre de dessin et le nombre de lignes ont donc eu être adapté ainsi que le mode de dessin. Il est visible à tout moment, même derrière les objets opaques.

Pour rendre l'expérience utilisateur plus facile, nous avons décidé de changer la couleur du curseur : **rouge** s'il est à un endroit où un cube existe déjà, **bleu** si c'est un espace vide et **jaune** si Ctrl est pressé, c'est-à-dire qu'on peut remplacer le cube sur lequel nous sommes. Ce changement de couleur se fait grâce à une variable uniforme dans le shader du sélecteur.

- **La skybox**

La skybox est définie par des vertices et des couleurs afin de créer un dégradé. Elle a deux coloris possibles : un pour le mode jour, un pour le mode nuit. Comme nous voulions un effet dégradé, nous ne sommes pas passées par une valeur uniform comme pour le sélecteur. Nous avons associé chaque vertex à une couleur. On obtient donc un dégradé de bleu clair en journée, et de bleu foncé en nuit. Pour cela nous avons créé deux vertex3DColor qui sont sélectionnés en fonction du mode (jour/nuit) choisi par l'utilisateur.

```
> const Vertex3DColor skyboxVerticesDay[] = {···
> const Vertex3DColor skyboxVerticesNight[] = {···
```

- **Les textures**

Les textures sont faciles à gérer mais prennent beaucoup de lignes de code, et en créer plusieurs d'affilé serait redondant. Nous avons donc décidé de créer une classe qui allait gérer la création, bind et unbind. Mis à part cela, il n'y a pas de différence notable avec la manière d'utiliser les textures adoptée en TD.

```
Texture::Texture(const std::string& path)
: m_textureID(0),
m_filepath(path),
m_width(0),
m_height(0),
m_BPP(nullptr)
{

if(path != ""){
    std::unique_ptr<glimac::Image> textureImg = glimac::loadImage(path);

    if(textureImg == NULL)
        std::cout << "Image located in "<< path << " not loaded !" <<std::endl;

    m_width = textureImg->getWidth();
    m_height = textureImg->getHeight();
    m_BPP = textureImg->getPixels();

    glGenTextures(1, &m_textureID);
    glBindTexture(GL_TEXTURE_2D, m_textureID);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    glTexImage2D(GL_TEXTURE_2D,
        0,
        GL_RGBA,
        m_width,
        m_height,
        0,
        GL_RGBA,
        GL_FLOAT,
        m_BPP
        );
```

- **Les différentes structures de vertex**

Nous avons créer des structures afin de clarifier notre code. Celles-ci permettent de contenir en un point plusieurs données. On a donc Vertex3DColor, qui génère une coordonnée 3D et une couleur, Vertex3D, qui génère une coordonnée 3D, une coordonnée de normale et une coordonnée de texture, et enfin VertexTex, qui génère seulement une coordonnée 2D et une coordonnée de texture. Nous avons mis dans le

même fichier nos matrices de projection et normale, ainsi qu'une fonction permettant de vérifier si une valeur est bien contenue entre deux bornes.

- **La structure World**

Afin d'abstraire toutes nos classes et de limiter au maximum la longueur de notre fichier main, nous avons créé une structure World. Elle contient donc notre skybox, notre sélecteur, notre pinceau, et nos instances de tableaux de cubes. C'est elle qui est donc créée dans le main, et qu'on appelle pour la gestion des évènements et pour l'affichage de nos scènes.

C'est grâce à celle-ci qu'on gère ce qui est dessiné en premier. En effet, on doit dessiner notre skybox en premier, afin que les textures transparentes laissent transparaître la skybox et non le fond noir de base. Ensuite on dessine le curseur dans un éventail de profondeur assez grand, puis nos cubes, et enfin la palette.

C'est aussi elle qui gère l'ajout des cubes dans la scène et le remplacement. En effet, pour ajouter un cube, on doit pouvoir vérifier qu'aucun cube, même d'une autre sorte existe. Pour cela, il faut donc pouvoir comparer les tableaux entre eux. Pour ajouter un cube, on vérifie donc qu'aucune instance de cube dans aucun tableau n'existe, et si c'est le cas on ajoute un cube du type demandé. Pour remplacer en revanche, on supprime juste tout éventuel cube à cette position, et on en ajoute un du bon type.

Cette structure gère aussi le passage au mode nuit de la skybox et des shaders des cubes, en indiquant à chaque classe qu'elle doit changer de mode.

Elle a également un attribut m_lights qui sert d'intermédiaire entre le shader des cubes et la position des lumières. On ne dessine pas les lumières comme des cubes. Elles ont une fonction draw spécifique qui va en fait envoyer les positions des lumières aux shaders qui eux vont influer sur les cubes texturés.

```
class World{
    private:
        std::vector<Cube*> m_allCubes;
        Skybox* m_skybox;
        Selector* m_selector;
        Pinceau* m_pinceau;
        glm::vec3 m_lights[6];
    public:
        World();
        ~World();

        void draw(const glm::mat4 MVMatrix, const bool ctrlIsPressed);
        void drawLights();
        void handleEvents(const SDLKey e, const bool ctrlIsPressed);

        void addCube(const glm::vec3 position, const typeCube typePinceau);
        void replace(const glm::vec3 position, const typeCube typePinceau);
        void extrudeCube(glm::vec3 position, const typeCube typePinceau);
        void digCube(glm::vec3 position, const typeCube typePinceau);
        void changeNightMode();
        bool existsSmth(const glm::vec3 position);
        int findLastCube(const glm::vec3 position);
};
```

- **La musique**

Nous avons décidé d'ajouter un peu de musique grâce à SDL_Mixer. Il n'y a pas de particularité, si ce n'est qu'elle rentre facilement dans la tête !

```
inline int playMusic(){
    if(Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, MIX_DEFAULT_CHANNELS, 1024) == -1){
        std::cout <<"ERROR initializing mix" << std::endl;
        return EXIT_FAILURE;
    }

    Mix_Music *game_music = Mix_LoadMUS( "assets/sounds/game_bg.wav");// load music for the game
    Mix_PlayMusic(game_music, -1); // play music

    return 1;
}
```

2) Fonctionnalités développées partiellement

- **La skybox texturée**

Nous avons dans un premier lieu voulu créer une skybox texturée (avec un ciel nuageux et un ciel de nuit), mais nous n'y sommes pas parvenus. Les fonctions le permettant sont donc implémentées mais ne fonctionnent pas pour une raison qui nous échappe (alors que nous avons réussi à appliquer des textures de la même manière sur nos cubes). Nous pensons qu'il s'agit d'une petite erreur de syntaxe au milieu de notre code au moment de binder la texture, mais nous n'avons pas su la trouver.

```
/*void Skybox::loadTexture(){
    glActiveTexture(GL_TEXTURE0);
    glGenTextures(1, &m_textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, m_textureID);

    for (GLuint i = 0; i < faces.size(); i++){
        std::unique_ptr<glimac::Image> textureImg = glimac::loadImage(faces[i]);

    assert(("Image located in "+faces[i]+" not loaded !", textureImg != NULL));

        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
            0,
            GL_RGB,
            textureImg->getWidth(),
            textureImg->getHeight(),
            0,
            GL_RGBA,
            GL_UNSIGNED_BYTE,
            textureImg->getPixels());

        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
        glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
    }
}*/
```

Pour contrer cela, nous avons finalement décidé de créer notre skybox grâce à un dégradé de couleur, ce qui a nécessité une recherche concernant les points de repère du dégradé (une face de la skybox étant composée de deux triangles) (Voir “skybox” et “mode nuit” dans la partie précédente).

3) Fonctionnalités non-développées

Du point de vue jouabilité, il n’y a pas de fonctionnalités évoquées dans le cahier des charges que nous n’avons pas développées. En revanche, nous avons dû faire des choix quant aux consignes concernant la partie programmation pure, notamment sur l’utilisation de fonctions et techniques qui nous paraissaient peu intuitives ou peu pertinentes pour l’utilisation que nous en faisions.

Ce dernier point est développé dans la partie suivante *(Voir IV.1.c, Bilan)*. Ainsi, nous nous sommes détachées de l’utilisation des constexpr, fonctions lambda ou même de l’héritage. Nous avons pensé, au début de la construction structurelle de notre projet, à créer une classe mère “formes 3D” dont auraient hérité les classes “cube” “light” “selector” etc… mais nous n’avons finalement plus trouvé ça pertinent une fois que nous avons décidé de ne plus créer des “cubes” mais des “vecteurs de cubes”.

Nous pouvons cependant noter que notre fichier “Vertex” comprend trois structures de vertex différents comme expliqué précédemment. *(Voir I et II.1.)*

III/ Résultats

1) Choix personnels

a) Utilisation de Glimac

Nous avons décidé de ne pas utiliser la bibliothèque glimac, mise à part pour le chargement des images de textures (loadImage()). Premièrement parce que l'utilisation de glimac provoque parfois des warning alors même que nous ne l'utilisions pas, mais aussi parce que, comme expliqué ultérieurement, nous avons préféré gérer nous mêmes les shaders et les textures. Cela nous permet de comprendre entièrement le fonctionnement de ceux-ci.

b) Interprétation des fonctionnalités

Les fonctions "extrude" et "dig" ne semblaient pas être les mêmes pour tout le monde. Certains pensaient que nous Nous avons donc choisi qu'elles permettaient d'ajouter ou supprimer en haut d'une colonne de cube, le type de cube sélectionné par l'utilisateur.

c) Jouabilité et développement des fonctionnalités

C'est là que se situent nos plus grosses prises de liberté :

Premièrement, nous avons basé notre gameplay sur le modèle de Minecraft. C'est pour cette raison que nous avons choisi de gérer la caméra comme une tête humaine. La palette n'a pas été gérée avec imGui car l'esthétique de la console n'était pas à notre goût et en personnaliser une semblait plus intéressant.

C'est aussi dans le but de développer au maximum la jouabilité du jeu et le confort de l'utilisateur que nous nous sommes penchées sur le développement de fonctionnalités optionnelles plutôt que sur le fait de "checker" les différents points du cahier des charges. En effet, après avoir développé les fonctionnalités primaires (ajouter, supprimer, dig, extrude un cube, caméra etc…) nous avons choisi de développer des fonctionnalités qui nous semblaient essentielles pour un jeu plus friendly pour l'utilisateur. Bien sûr nous sommes revenues sur le cahier des charges par la suite, mais cela nous a permis d'avoir une expérience plus agréable de notre projet.

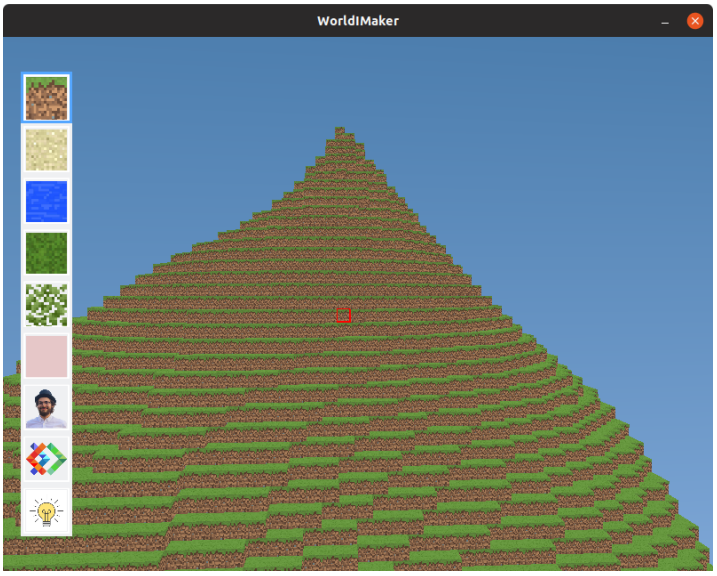
2) Fonctions radiales & points de contrôle

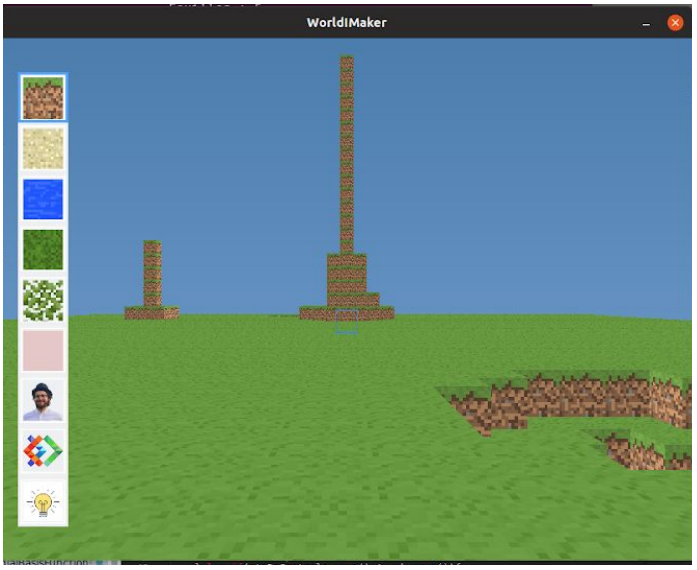
a) Explication par et pour les IMACs

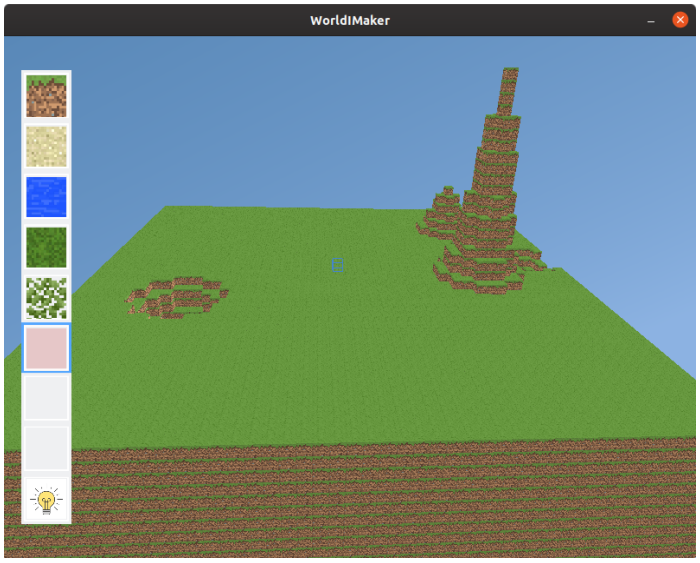
Nous avons remarqué que beaucoup d'élèves avaient eu du mal à comprendre, d'une part, ce que sont les RBF, mais surtout, comment les coder et les appliquer informatiquement. Fort heureusement, nous avons pris le temps de rédiger un petit cours "informel" afin d'éclaircir la chose. Nous vous invitons à consulter l'annexe afin de pouvoir feuilleter ce cours.

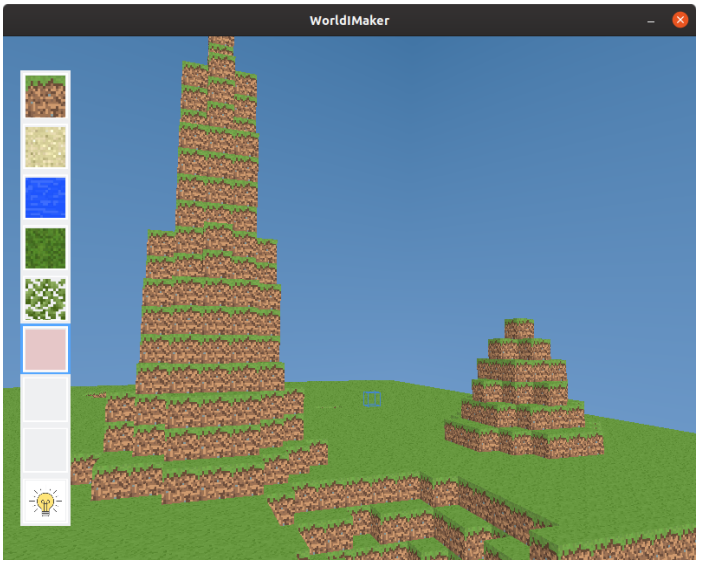
b) Nos tests personnels

Explications par Margaux









3) Améliorations

Nous avons la joie d'avoir pu satisfaire la plupart des critères de notation de ce projet, tout en ayant ajouté certaines fonctionnalités bonus. Mais bien évidemment, nous pourrions faire une tonne d'améliorations pour que notre jeu soit parfait à nos yeux.

Cependant, la chose qui nous dérange le plus au rendu du projet est la gestion de la transparence des textures. En effet, nous avons remarqué qu'en fonction de l'ordre dans lequel nous créons nos textures, la transparence ne prend pas forcément en compte les cubes autour d'elle.

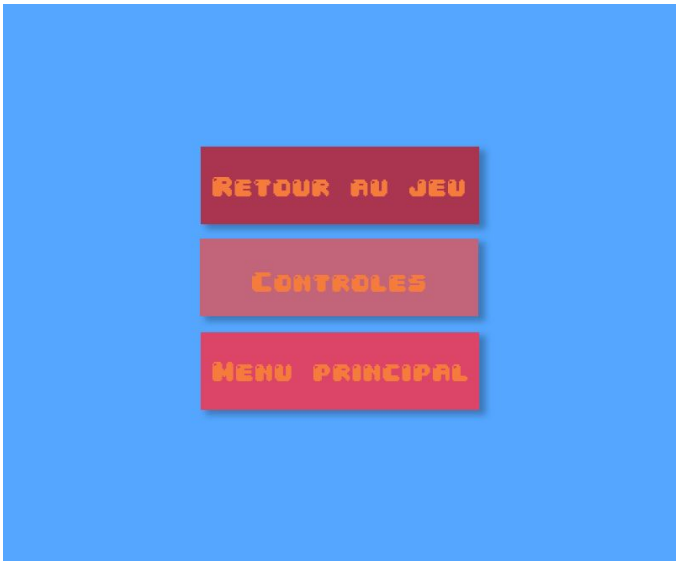
Si la texture du cube1 (transparent) est créée avant celle du cube2, en mettant le cube1 devant le cube2, on ne verra pas le cube2 derrière le cube1. En revanche nous verrons le cube1 s'il est derrière le cube2 !

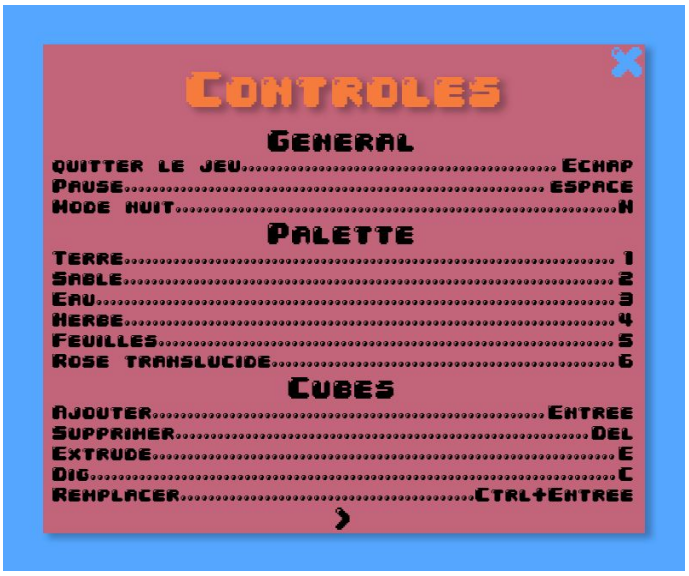
D'autres idées d'améliorations concernant surtout l'expérience utilisateur :

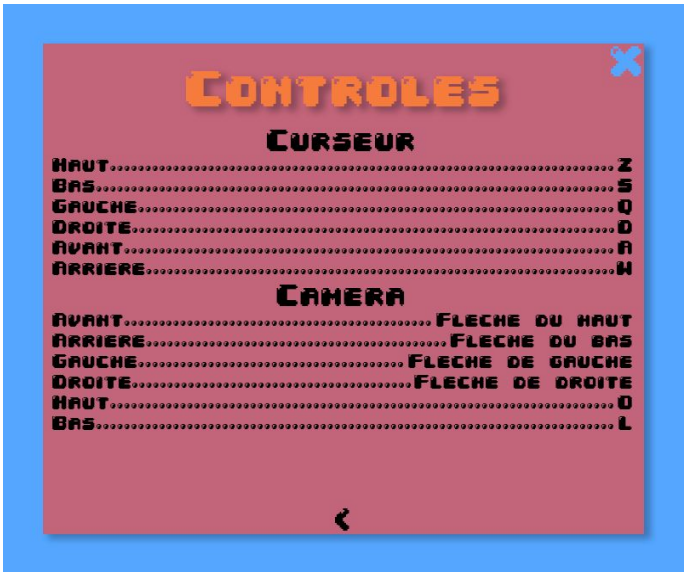
- Le choix de commencer avec un terrain plat ou en RBF
- Des sons à la création/suppression des cubes
- Des propriétés de "gravité" différentes suivant les cubes (nécessite une animation visuelle)
- Etc...

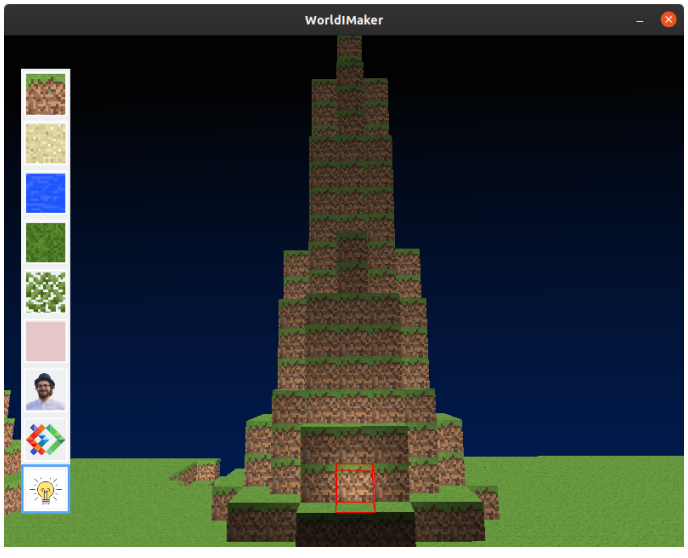
Nous sommes toutefois très satisfaites du résultat final, que nous nous pressons de vous présenter dans la partie qui suit.

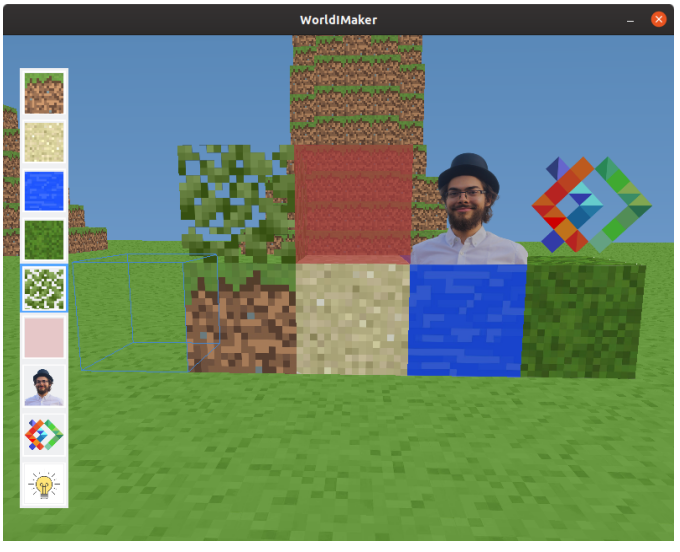
4) Captures d'écran









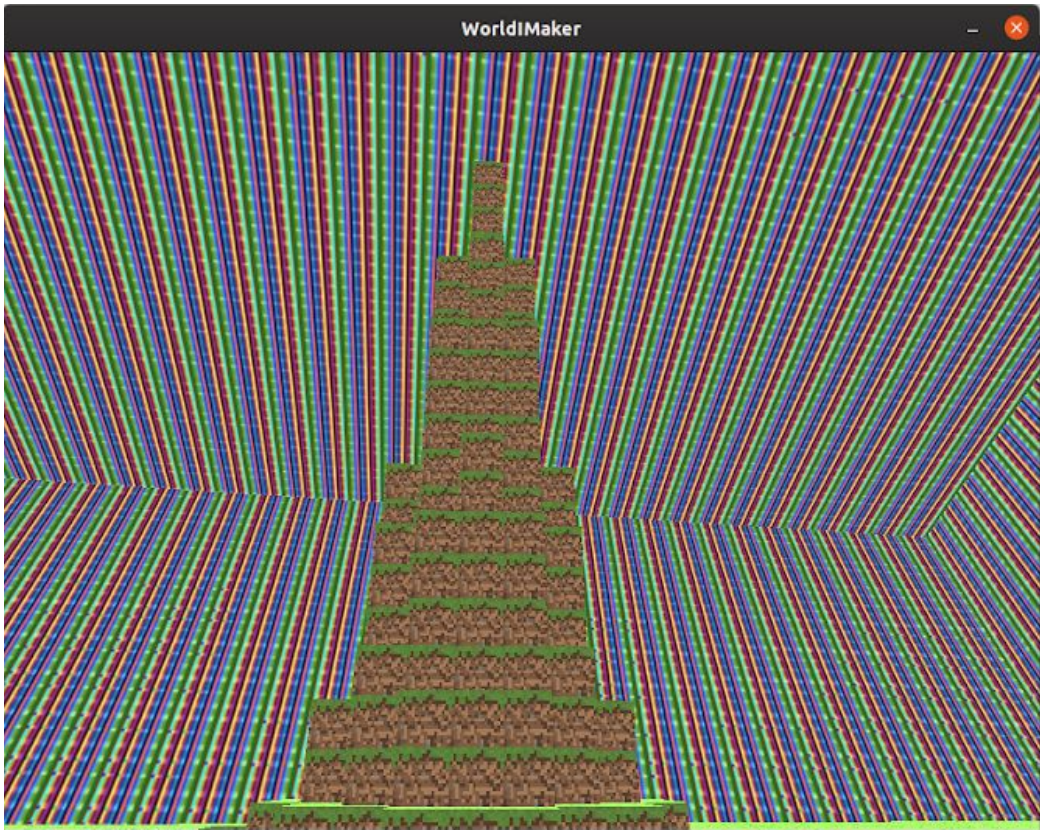


IV/ Bilan personnel & Conclusion

1) Difficultés rencontrées

a) Skybox

Nous avons essayé de faire une skybox texturée mais nous n'y sommes pas parvenue. En effet, la texture ne semblait pas de bind et nous obtenions une texture multicolore tout autour de notre scène. Nous avons finalement décidé de ne pas nous arrêter là et de créer une skybox en dégradé. *(Voir II. Fonctionnalités)*



b) Gestion des fuites de mémoires (Valgrind)

Nous avons utilisé Valgrind mais avons eu du mal à décrypter le message sortant de celui-ci. Nous avons essayé d'en tirer davantage en exécutant les paramètres --leak-check=full, -s ou même --track-origins=yes, mais cela ne nous aiguillait pas beaucoup plus.

```
==8359== HEAP SUMMARY:
==8359==     in use at exit: 392,249 bytes in 6,228 blocks
==8359==   total heap usage: 30,515 allocs, 24,287 frees, 1,940,179,949 bytes allocated
==8359==
==8359== LEAK SUMMARY:
==8359==    definitely lost: 8,077 bytes in 30 blocks
==8359==    indirectly lost: 76,608 bytes in 425 blocks
==8359==      possibly lost: 168,072 bytes in 5,271 blocks
==8359==    still reachable: 139,492 bytes in 502 blocks
==8359==         suppressed: 0 bytes in 0 blocks
==8359== Rerun with --leak-check=full to see details of leaked memory
==8359==
==8359== Use --track-origins=yes to see where uninitialised values come from
==8359== For lists of detected and suppressed errors, rerun with: -s
==8359== ERROR SUMMARY: 19 errors from 11 contexts (suppressed: 0 from 0)
```

Nous avons toutefois remarqué que le nombre de fuites de mémoire ne variait pas malgré la création/suppression de différents cubes au fil des tests. Nous en avons conclu qu'il s'agissait sûrement d'une fuite qui avait lieu à l'initialisation du jeu (dans le main). Lorsque nous avons commencé à ajouter des textures pour les menus, nous avons vu que le chiffre avait augmenté. Avec un peu de recherche, nous avons compris qu'il s'agissait des textures appelées dans le main qui étaient free() à certains endroits mais qui dont le delete manuel créait parfois des segment fault, parce que nous les manipulions via des pointeurs. Nous avons réussi à éviter les segment fault et à réduire nos fuites de mémoire, mais il en reste encore que nous n'avons pas su déceler.

c) Choix des fonctionnalités à développer

Comme expliqué dans la toute première partie de ce rapport, nous avons voulu créer un code qui nous était intuitif et compréhensible. Lorsque que nous avons relu le cahier des charges, nous étions embêtées face aux contraintes imposées dans la partie programmation. Certaines de ces contraintes ne nous paraissaient pas intuitives au premier abord, et étaient nouvelles, nous avons hésité à les intégrer pour dire que nous l'avons fait, ou à les laisser de côté pour conserver notre code "intuitif". Finalement, en nous replongeant dans nos cours et TDs, nous avons pu intégrer certaines de ces fonctionnalités qui se sont avérées très pratiques quant au développement de notre code, et nous avons pu les intégrer facilement. Notamment les fonctions relatives aux exceptions et assertions, qui nous permettent d'éviter des erreurs basiques de développement (nullptr, division par 0), de système (version Opengl, lib), ou de l'utilisateur (mauvaise touche…).

2) Bilan personnel

a) Margaux

[Avis Margaux]

b) Laurelenn

Ce projet est le plus conséquent que j'ai pu réaliser en programmation depuis le début de mes études supérieures. Il mobilise de nombreuses notions dans des matières différentes, et cela représente pour moi un véritable symbole concernant mes choix en terme de scolarité. En effet, j'ai longtemps attendu de faire un

“programme” concret, complet et utilisable, plus que simplement en TD (ce que nous avons déjà fait en première année). Mais il y a aussi le fait que je puisse enfin intégrer de manière logique et pertinente, à mes yeux, les mathématiques en informatique, sans que le sujet ne soit “les mathématiques” justement. Aussi, la découverte et l’utilisation d’OpenGL en 3D m’a permis de créer un projet esthétique à mes yeux et dont le l’interface et le retour visuel sont agréables à mes yeux et me donnent l’impression de créer quelque chose de concret. Je n’avais que très peu eu cette impression, en programmation, en première année et d’autant moins en DUT Informatique. Je suis toutefois quelques peu frustrée car je n’ai pas pu m’investir autant que je l’aurais voulu en terme de temps. Je tiens d’ailleurs à saluer l’investissement de Margaux dans ce projet sans laquelle nous n’aurions pas eu autant de fonctionnalités additionnelles notamment. J’ai toutefois pu développer ce projet avant les vacances en élaborant l’architecture de nos fichiers et données, la structure de nos classes, et les fonctions élémentaires à celles-ci. Mais également en incluant des notions que je n’avais que très peu pratiquées : Les exceptions, assertions, gestion de fuites de mémoire, textures 3D et bien d’autres choses.

Pour conclure, je dirais que je suis plus que satisfaite du résultat de notre projet et de ce que celui-ci m’a apporté, ce qui reste le plus frustrant, c’est qu’à partir du moment où nous avons compris comment développer un mini-minecraft, on ne veut plus s’arrêter ! C’est pourquoi je pense essayer de poursuivre le développement de celui-ci afin de le rendre d’autant plus complet.

3) Conclusion

Si vous regardez le dernier commit des git des IMACs, nous sommes sûres que la plupart ont été faits au dernier moment, voire un peu en retard pour certain.e.s…

Et pour cause : on peut toujours améliorer un projet (surtout s’il a été fait sur un laps de temps relativement réduit), et une fois que nous sommes lancés, il est difficile de s’arrêter ! Notre projet est bien loin du Minecraft parfait, cependant nous avons la satisfaction et la fierté d’avoir développé un projet utilisable, user friendly à nos yeux, et ayant mobilisé une grande partie de nos compétences.

Nous avons pu travailler ensemble sur notre vision du jeu, sa conception structurale et logique, son développement et sur ce que tout cela nous a apporté. Le projet était conséquent et a pu nous faire peur au début, surtout en cette période de l’année. Il a demandé beaucoup de travail autant dans la recherche, que dans le développement et que dans l’analyse de celui-ci, mais nous avons pu en ressortir plus compétentes.

De plus, d’un point de vue plus personnel, c’est la première fois que nous travaillons seulement toutes les deux sur un projet aussi important (l’an dernier dans un groupe de 4 en web), ce qui nous a permis d’en apprendre plus sur la manière de travailler de chacune et sur comment nous adapter à l’autre.

Pour finalement conclure, ce projet a représenté un véritable défi mais nous a permis de nous surpasser, et le résultat nous est satisfaisant. Nous espérons en faire une démo pour l’ajouter à notre portfolio!

V/ Annexe : RBF appliqué au World IMACker

Radial Basis Functions

Ce qu'on doit retenir du cours

En gros un fonction radial ça va servir à quoi ? On va dire par exemple, **"Tiens, au point (1, 1) je veux qu'il y ait une montagne ! Et qu'elle soit haute de 10 !"**
Et la fonction radiale va permettre à ce que les cubes autour forment une montagne, et qu'il n'y ait pas juste un pic quoi.

Mais du coup comment on fait ça ? Parce que bon...

Alors, easy easy, on va tout décrypter ! Chaque point de la map va être calculé à partir des points appelés **de contrôle** pour que tout soit harmonieux et joli ! Pour cela, on a une fonction dans le cours qui est appelée $g(x)$ et qui se note :

$$g(x) = \sum_{i=1}^{k} \omega_i * \Phi\left(|x - x_i|\right)$$

Ouhlala ça parait compliqué... Alors il faut comprendre que cette formule, c'est le cours, on va pas essayer de comprendre pourquoi la somme, pourquoi les omega tout ça. On va juste retenir qu'ici on a des ω, qu'on ne semble pas connaitre, une fonction qui s'appelle Φ et qui prend en paramètre le x qu'on cherche et les points de contrôle (les x_i).
Nous ce qui nous intéresse c'est **Comment coder tout ça ?**
En gros il faut retenir qu'on va avoir :

1. Une liste de points de contrôle
2. Une liste de valeurs associées une à une à ces points
3. Des ω
4. Des Φ

Comment on code tout ça ?

Avec Eigen ! Eigen, pour rappel, ça prend des Matrices et des vecteurs, donc on va essayer de voir ici en quoi c'est pertinent.
Déjà, si tu lis ça j'espère que tu as le cours du prof ouvert à côté sinon : Voilà
En vrai on l'utilise pas tout de suite, mais après c'est utile. Déjà on va commencer doucement.

1. Qu'est-ce qu'on va utiliser dans Eigen ? Qu'est-ce qui est sous quelle forme ?

En fait ce qu'on veut c'est la hauteur de chaque endroit de notre map. Donc on peut interpréter ça comme si on avait une grille qui correspond à tout nos points et que dans chaque case ya écrit la hauteur du point à cette coordonnée. Une grille ? Donc une matrice quoi !
On va donc avoir une fonction qui retournera notre map qui est du type `Eigen::MatrixXd` (notez qu'on prend Xd parce qu'on va lui mettre une grande taille, genre 50 cubes par 50 cubes, et qu'elle va contenir des int, d'où le d).
Ensuite on a nos points de contrôle x_i. On a donc à chaque fois 2 coordonnées (x et z comme on cherche la hauteur y), et on a N points de contrôles. On a donc une matrice qui va être sous la forme

$$\begin{pmatrix} x_1 & z_1 \\ x_2 & z_2 \\ x_3 & z_3 \end{pmatrix}$$

si on a 3 points de contrôle par exemple.
Enfin, on va avoir nos valeurs associées à ces points de contrôle que Mr Nozick a noté u_i. Là il s'agit donc d'un simple vecteur :

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}$$

On a donc notre premier point qui a comme valeur de contrôle u_1, etc...
OK ! Ca faisait beaucoup alors résumé !

Mais avant mini rappel pour Eigen, pour déclarer une matrice, on écrit Eigen::MatrixXd maMatrice(hauteur, largeur) et Eigen::VectorXd monVecteur(taille).

DONC
On a au début 3 variables :

- `Eigen::MatrixXd` **map**(taille, taille) : contient les hauteurs de tous les points de notre carte
- `Eigen::MatrixXf` **ptsDeControle**(nbPoints, 2) : nos points de contrôle
- `Eigen::VectorXf` **valeurs**(nbPoints) : nos valeurs associées aux points

2. Notre première fonction

On va l'appeler `getMap`. Elle retourne notre map, donc notre matrice avec les hauteurs, et elle prend en paramètres les contraintes, donc les **ptsDeControle** et les **valeurs**. On a donc

In [ ]:
```
Eigen::MatrixXd getMap(const Eigen::MatrixXd ptsDeControle, const Eigen
::VectorXd valeurs)
```

Maintenant qu'est-ce qu'on doit faire ? Alors on regarde le cours déjà. On regarde particulièrement bien le **4.**. Il est écrit que pour trouver la hauteur d'un point donc, on doit utiliser des ω et des Φ de bidule. On y va un par un donc :

a. Trouver les ω

Page 3, on explique un truc très simple en fait. Ces ω, c'est des valeurs qui vont être trouvées grâce à nos contraintes ! On remarque d'ailleurs qu'ils sont sous la forme d'un `Eigen::VectorXf`. Il est écrit qu'une matrice composée de $\Phi(|x_i - x_j|)$ multiplié par les ω est égal à nos valeurs u_i.
On va donc créer une fonction que j'ai personnellement appelé `findOmega` qui va donc nous renvoyer les omegas, c'est-à-dire un `Eigen::VectorXf` et qui prend en paramètre nos points de contrôle, et nos valeurs. On a donc :

In [ ]:
```
Eigen::VectorXf findOmega(const Eigen::MatrixXf ptsDeControle, const Ei
gen::VectorXf valeurs)
```

Maintenant, un tout petit peu de maths : si on note la matrice des Φ mettons, A. On sait que

$$A * \omega_i = u_i \iff \omega_i = A^{-1} * u_i$$

Or on a les u_i, ce sont nos **valeurs**, et les x_i utilisés dans les fonctions Φ sont nos **ptsDeControle**, donc on les a aussi ! Du coup écrivons cette fonction !

On va donc créer une variable A, qui sera une matrice carrée vide. Elle est aussi grande que le nombre de points de contrôle, c'est à dire le nombre de ligne de la matrice (puisque les points de contrôle sont on le rappelle sous la forme :

$$\begin{aligned} point1 &-> (x_1 \quad z_1) \\ point2 &-> (x_2 \quad z_2) \\ point3 &-> (x_3 \quad z_3) \end{aligned}$$

donc une ligne = un point.
On va donc déclarer A comme ceci :

In [ ]:
```
Eigen::MatrixXf A = Eigen::MatrixXf::Zero(ptsDeControle.rows(), ptsDeCo
ntrole.rows());
```

On veut ensuite la remplir avec les $\Phi(|x_i - x_j|)$, où i et j nous permettent en fait de parcourir les index des colonnes et lignes de la matrice ! (voir page 3 du cours toujours) On va donc simplement faire deux boucles **for** et remplir la case avec $\Phi(|x_i - x_j|)$ (pour l'instant on l'écrit juste comme ça, on verra après comment calculer). Ce qui nous donne :

In [ ]:
```
for(int i=0; i<A.rows(); i++){
        for(int j=0; j<A.cols(); j++){
            A(i, j) = phi(valeurAbsolue(xi - xj);
        }
    }
```

Mais en fait c'est quoi `xi` et `xj` ? Bah c'est dans nos points de contrôle ! Donc c'est notre points de contrôle à l'index i et celui à l'index j. Comme on l'a vu, ça veut donc dire à la **ligne** i, et la **ligne** j. On le notera donc `ptsDeControle.row(i)` et `ptsDeControle.row(j)`.

Ensuite qu'est-ce que la valeur absolue de la différence entre deux points ? Bah en fait c'est tout bêtement la norme, et ça Eigen sait le faire ! (On va appeler cette technique

)
Enfin on renvoie donc notre matrice inversée multipliée par nos valeurs.
Donc ouhla c'était compliqué mais on obtient enfin notre première fonction complète qui nous permet d'obtenir les ω !

In [ ]:
```
Eigen::VectorXf findOmega(const Eigen::MatrixXf ptsDeControle, const Ei
gen::VectorXf valeurs){
    Eigen::MatrixXf A = Eigen::MatrixXf::Zero(ptsDeControle.rows(), pts
DeControle.rows());

    for(int i=0; i<A.rows(); i++){
        for(int j=0; j<A.cols(); j++){
            A(i, j) = phi(
                (ptsDeControle.row(i) - ptsDeControle.row(j)).norm()
                );
        }
    }

    return A.inverse() * valeurs;
}
```

Mais attendez... Il y a toujours cette fonction Φ là ? O.o
Bah c'est notre fonction radiale ! Alors moi j'ai pris pour l'instant, une gaussienne, qui se note $\Phi(a) = e^{-0.2a^2}$.
Donc petite fonction facile à coder, qui prend en paramètre un float et qui renvoie un float.

In [ ]:
```
float phi(const float a){
    return exp(-0.2*a*a);
}
```

b. On revient à notre fonction générale `getMap`

Ca y est ! On a nos ω ! On peut donc écrire dans notre fonction déjà :

In [ ]:
```
Eigen::VectorXf w = findOmega(ptsDeControle, valeurs);
```

On sait que chaque hauteur dans notre map va être définie par

$$g(x) = \sum_{i=1}^{k} \omega_i * \Phi\left(|x - x_i|\right)$$

Donc on va devoir parcourir la map, et pour chaque point, appliquer cette formule. Déjà

Parcourir la map

Easy ! On va créer une matrice (celle qu'on va renvoyer), de la taille de notre map, et on la parcourt avec une double boucle **for** :

In [ ]:
```
Eigen::MatrixXd values = Eigen::MatrixXd::Zero(WORLD_TAILLE, WORLD_TAIL
LE);

    for(int i=0; i<values.rows(); i++){
        for(int j=0; j<values.cols(); j++){
           //Trouver la valeur
        }
    }
```

Maintenant comment on trouver cette valeur ? On regarde le cours ! Ca dit que ça va être $\sum_{i=1}^{k} \omega_i * \Phi\left(|x - x_i|\right)$. Donc on a notre fonction Φ de codée, il nous faut une fonction qu'on va appeler `findValue` qui va nous calculer la valeur à écrire dans la map. Elle prend en paramètre la position dans la map, donc le i et le j, et enfin les points de contrôle (dans la formule, ils sont notés x_i) et nos ω

c. La fonction `findValue`

On sait qu'elle va retourner un int, la hauteur à cet endroit de la carte. On sait aussi qu'elle a en paramètres le i et le j, et enfin les points de contrôle et nos ω. On obtient donc:

In [ ]:
```
int findValue(const int i, const int j, const Eigen::MatrixXf ptsDeCont
role, const Eigen::VectorXf w)
```

Dans la formule, chaque valeur est une somme de plein de trucs. Donc déjà on sait qu'on va avoir une boucle, qui va tourner autant de fois qu'il y a de points de contrôle, et qu'on va tout sommer.
On commence donc par écrire :

In [ ]:
```
float value = 0;

for(int index=0; index < ptsDeControle.rows(); index ++){
    //Faire quelque chose
}
```

On sait aussi que i et j désigne en fait le point qu'on va utiliser dans notre fonction Φ. On va donc convertir ces 2 int en un VectorXf qui prend 2 float en paramètres.

In [ ]:
```
Eigen::VectorXf point(2);
point << i, j;
```

Enfin, qu'est-ce qu'on va faire dans notre boucle ? Eh bien on somme nos ω avec nos Φ. Vous vous rappelez de la méthode

? Bah on la réutilise ici ! Sauf que cette fois c'est notre point par rapport à nos points de contrôle.
ATTENTION ! Eigen met les vecteurs sous la forme

$$\begin{pmatrix} x_1 & z_1 \end{pmatrix}$$

au lieu de

$$\begin{pmatrix} x_1 \\ z_2 \end{pmatrix}$$

donc on doit penser à transposer avant de soustraire ! On obtient donc notre fonction en entier :

In [ ]:
```
int findValue(const int i, const int j, const Eigen::MatrixXf ptsDeCont
role, const Eigen::VectorXf w){
    float value = 0;
    Eigen::VectorXf point(2);
    point << i, j;

    for(int index=0; index < ptsDeControle.rows(); index ++){
        value += w(index) * phi(
            (point.transpose() - ptsDeControle.row(index)).norm()
        );
    }

    return round(value);
}
```

Vous remarquerez qu'à la fin je renvoie `round(value)` et pas juste value parce que je veux que mes cubes restent à des hauteurs entières.

d. Du coup c'est quoi notre fonction générale ?

Bah du coup là vous devriez tout comprendre :

In [ ]:
```
Eigen::MatrixXd getValues(const Eigen::MatrixXf ptsDeControle, const Ei
gen::VectorXf valeurs){
    Eigen::VectorXf w = findOmega(ptsDeControle, valeurs);
    Eigen::MatrixXd map = Eigen::MatrixXd::Zero(WORLD_TAILLE, WORLD_TAI
LLE);

    for(int i=0; i<map.rows(); i++){
        for(int j=0; j<map.cols(); j++){
            map(i, j) = findValue(i, j, ptsDeControle, w);
        }
    }
```

```
    return map;
}
```

3. Mdr je mets ça où dans mon code général ?

Du coup moi j'ai appelé ce code dans une fonction `loadMonde()` dans ma classe Cube. Je déclare une map, des points de contrôle des valeurs.

In [ ]:
```
void Cube::loadMonde(){
    Eigen::MatrixXd map(WORLD_TAILLE, WORLD_TAILLE);

    const int nbPoints = 3;

    Eigen::MatrixXf ptsDeControle(nbPoints, 2);
    ptsDeControle << 10, 10,
    2, 3,
    30, 40;
    Eigen::VectorXf valeurs(nbPoints);
    valeurs << 10, -2, -20;

    map = getValues(ptsDeControle, valeurs);
```

Puis je remplis mon tableau de cube avec un glm::vec3 grâce à une triple boucle **for** où x et z sont incrémentés bêtement.
Pour y, je le fait commencer du "fond" de mon monde, pour remplir jusqu'à la hauteur souhaitée, c'est-à-dire `map(x, z)`. J'obtiens donc :

In [ ]:
```
for(int x=0; x<WORLD_TAILLE; x++){
    for(int z = 0; z<WORLD_TAILLE; z++){
        for(int y= WORLD_DEPTH; y <= map(x, z); y++){
            monTableauDeCubes.push_back(glm::vec3(x, y, z));
        }
    }
}
```