Cours OpenGL - GDL-GL Avril 2014

Paul Baron paul.baron@epitech.eu

Alexandre Zieder alexandre.zieder@epitech.eu

GameDevLab gamelab@epitech.eu

Petite Introduction

OpenGL OpenGL, pour Open Graphics Library, est une API (Application Programming Interface) graphique bas niveau vous permettant d'afficher des éléments visuels. Il est supporté par de multiples systèmes et hardware (si la machine est compatible) ce qui lui confère un grand avantage.

Nous avons surcouché cet API afin de vous simplifier certaines manipulations qui sont beaucoup trop longues et pas toujours simples à appréhender. Cependant, une grande partie des opérations basiques avec OpenGL sont à votre charge.

Nous vous fournissons aussi la librairie de mathématiques *glm* pour vous permettre de faire toutes les opérations nécessaires à la création d'un jeu facilement.

1 - Utilisation de la bibliothèque

1.1 - Les fichiers en-tête

Chacune des classes de la bibliothèque est déclarée dans un fichier en-tête. Elles sont toutes dans un namespace nommé **gdl.**

```
#include <Game.hh>
#include <Clock.hh>
#include <Input.hh>
#include <SdlContext.hh>
#include <Geometry.hh>
#include <Texture.hh>
#include <BasicShader.hh>
#include <Model.hh>
```

Pour utiliser OpenGL, il faut inclure un fichier en-tête que vous nous fournissons et qui se charge de d'inclure les bibliothèques necessaires à l'utilisation d'OpenGL :

```
#include <OpenGL.hh>
```

Pour utiliser *glm*, il faut inclure deux fichiers en-tête :

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

1.1.2 – Compilation

Voici la ligne de compilation qui vous permettra de compiler votre programme à partir du dossier LibBomberman_linux_x64 que vous nous fournissons :

g++ *.cpp –Ichemin_de_la_lib /includes/ -Lchemin_de_la_lib/libs/ -lgdl_gl –lGL –lGLEW –ldl –lrt –lfbxsdk –lSDL2 –lpthread

-lGL : la bibliothèque OpenGL
-lGLEW : OpenGL Extension Wrangler
-lgdl_gl : la bibliothèque graphique GDL_GL
-ldl : Permet de charger des bibliothèques dynamiques
-lrt : la bibliothèque utilisée par la SDL pour gérer le temps.
-lfbxsdk : la bibliothèque qui permet de charger des modèles 3D au format FBX, OBJ et COLLADA
-lSDL2 : la bibliothèque SDL
-lpthread : la bibliothèque de thread

Afin de pouvoir exécuter le code il vous sera nécessaire de modifier la variable environnement LD_LIBRARY_PATH de cette facon :

export LD_LIBRARY_PATH=chemin_de_la_lib/libs/

1.2 - Premiers pas

1.2.1 - La boucle de jeu

Une boucle de jeu est une boucle infinie qui englobe plusieurs tâches :

• mise à jour des inputs

• mise à jour de l'horloge principale

• mise à jour du comportement des objets

• affichage des éléments graphiques

Pour implémenter cette boucle, nous vous fournissons la classe gdl::Game qui contient les méthodes virtuelles pures suivantes :

```
virtual bool initialize() = 0;
virtual bool update() = 0;
virtual void draw() = 0;
```

- Dans la méthode Game::initialize, vous chargerez les différents « assets » (composant de votre jeu vidéo, c'est-à-dire les modèles 3D, textures et sons)
- Dans la méthode Game::update, vous mettrez à jour les inputs, l'horloge principale ainsi que le comportement de votre jeu (déplacement des joueurs, etc)
- Dans la méthode draw, vous dessinerez les différents éléments de votre jeu

C'est donc à vous de créer une classe qui héritera de gdl::Game qui implémentera ces différentes méthodes.

1.2.2 - Démarrer un contexte OpenGL

Avant toute manipulation graphique, il est essentiel d'ouvrir ce que l'on appelle un contexte OpenGL, c'est-à-dire initialiser la bibliothèque. Cette initialisation est faite lors de l'ouverture d'une fenêtre.

Nous utilisons la librairie graphique SDL 2 pour cela.

Pour vous faciliter l'ouverture du contexte, nous vous fournissons une classe SdlContext qui vous permet d'initialiser tout ce qui vous sera nécessaire pour avoir une fenêtre dans laquelle vous pourrez dessiner par la suite.

Cette classe se charge aussi de mettre à jour les inputs et l'horloge principale du jeu qui dépendent de la SDL.

Elle possède les méthodes suivantes :

```
bool    start(unsigned int swidth, unsigned int sheight, const std::string &name);
void    updateInputs(Input &input) const;
void    updateClock(Clock &clock) const;
void    flush() const;
void    stop() const;
```

- La méthode SdlContext::start vous permet d'ouvrir une fenêtre. Elle prend en paramètres sa largeur, sa hauteur et son nom
- La méthode SdlContext::stop vous permet de fermer la fenêtre
- La méthode SdlContext::updateInputs va mettre à jour les inputs (clics et mouvements de la souris, évènements claviers…)
- La méthode SdlContext::updateClock va elle mettre à jour l'horloge principale
- La méthode SdlContext::flush est utilisée une fois tous les objets dessinés pour les afficher

1.2.3 - L'horloge de jeu

La classe gdl::Clock vous permet de récupérer le temps écoulé entre chaque tour de boucle (frame) avec la méthode Clock::getElpasedTime pour vous permettre de gérer le temps dans votre jeu.

Pour cela, il vous faut instancier un objet gdl::Clock et le mettre à jour à chaque frame grâce à l'appel de la fonction SdlContext::updateClock.

1.2.4 - Les inputs

La classe gdl::Input vous permet de récupérer les différentes interactions du joueur (mouvement de souris, touche du clavier appuyée, fermeture de la fenêtre…)

Celle-ci est mise à jour avec la méthode SdlContext::updateInputs.

Elle possède les méthodes suivantes qui vous permettrons de récupérer les inputs :

```
glm::i8vec2 const           &getMousePosition();
glm::i8vec2 const           &getMouseDelta();
glm::i8vec2 const           &getMouseWheel();
bool                        getInput(int input, bool handled = false);
bool                        getKey(int input, bool handled = false);
```

- La méthode Input::getMousePosition vous permet de récupérer la position de la souris dans votre fenêtre
- La méthode Input::getMouseDelta vous permet de récupérer le déplacement de la souris depuis le dernier appel de SdlContext::updateInputs
- La méthode Input::getMouseWheel vous permet de récupérer le déplacement de la molette de souris depuis le dernier appel de SdlContext::updateInputs
- La méthode Input::getInput permet de savoir si un input a eu lieu ou non
- La méthode Input::getKey permet de savoir si une touche du clavier est appuyée ou non

Le paramètre *input* correspond aux enums de la SDL (par exemple SDLK_UP ou SDK_DOWN pour Input::getKey et SDL_QUIT pour le Input::getInput).

2 – Primitives

2.1 – L'objet gdl::Geometry

En programmation 3D nous utilisons les vertices. Il s'agit de «points » qui, une fois reliés, formeront un polygone. La classe qui va générer et stocker la géométrie est la classe gdl::Geometry.

La fonction membre suivante vous permettra d'ajouter un vertex à votre instance de l'objet Geometry. Elle prend en paramètre un glm::vec3 qui correspond à un point 3D dans l'espace et qui prend en paramètres des positions relatives.

```
void    Geometry::pushVertex(glm::vec3);
```

Pour pouvoir afficher les surfaces dessinées, il faut leur appliquer une texture. Pour cela il faut pouvoir charger une image. On utilisera la fonction membre Texture::load de la classe gdl::Texture qui vous encapsule une texture OpenGL.

```
bool    Texture::load(string & path)
```

En voilà un exemple d'utilisation:

```
if (_texture.load("./assets/texture.tga") == false)
{
        std::cerr << "Cannot load the texture" << std::endl;
        return (false);
}
```

Attention la lib ne supporte que les textures au format .tga

Et après chaque face que l'on aura push il faut donner les coordonnées relatives de texture sur la surface. Pour cela on utilise :

```
void    gdl::Geometry::pushUv(glm::vec2);
```

Cette fonction prend en paramètre un glm::vec2 qui prend deux positions relatives (0, 0 pour le coin en haut à gauche de votre image et 1, 1 pour le coin en bas à droite).

Après avoir dessiné les vertices il faut générer la géométrie. Pour cela on utilise la fonction membre suivante :

```
void    gdl::Geometry::build();
```

On peut ensuite l'afficher avec la fonction membre suivante qui prend en paramètre un gdl::AShader (que nous aborderons plus tard dans ce tutorial) et un GLenum :

```
void    gdl::Geometry::draw(gdl::AShader &, GLenum);
```

L'enum que prend la fonction en paramètre va déterminer la manière dont sont traité les vertices. Parmi les valeurs de l'enum, nous pouvons retenir :

- GL_TRIANGLES
- GL_QUADS
- GL_LINE_STRIP

GL_TRIANGLES permet de dessiner un triangle à partir de trois points. OpenGL prendra chaque triplet de vertices dans le groupe de vertices que vous avez push dans l'objet Geometry pour y dessiner des triangles.

GL_QUADS nécessite un quadruplet de vertices pour dessiner un quadrilatère.

GL_LINE_STRIP peut vous permettre de dessiner des lignes entre chaque points.

Pour utiliser une texture lors du draw, vous devez utiliser sa méthode bind avant d'appeler Geometry::draw. Vous ne pouvez pas dessiner de géométrie n'ayant pas de texture.

2.2 - Exemple

2.2.1 – Création d'un Cube

```
// On instancie un objet gdl::Geometry
gdl::Geometry _geometry;
```

```
// On charge la texture qui sera affichee sur chaque face du cube
if (_texture.load("./assets/texture.tga") == false)
{
	std::cerr << "Cannot load the cube texture" << std::endl;
	return (false);
}
```


```
            // on set la color d'une premiere face
            _geometry.setColor(glm::vec4(1, 0, 0, 1));
            // tout les pushVertex qui suivent seront de cette couleur

            // On y push les vertices d une premiere face
            _geometry.pushVertex(glm::vec3(0.5, -0.5, 0.5));
            _geometry.pushVertex(glm::vec3(0.5, 0.5, 0.5));
            _geometry.pushVertex(glm::vec3(-0.5, 0.5, 0.5));
            _geometry.pushVertex(glm::vec3(-0.5, -0.5, 0.5));

            // Les UVs d'une premiere face
            _geometry.pushUv(glm::vec2(0.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 1.0f));
            _geometry.pushUv(glm::vec2(0.0f, 1.0f));

            // ETC ETC
            _geometry.setColor(glm::vec4(1, 1, 0, 1));

            _geometry.pushVertex(glm::vec3(0.5, -0.5, -0.5));
            _geometry.pushVertex(glm::vec3(0.5, 0.5, -0.5));
            _geometry.pushVertex(glm::vec3(-0.5, 0.5, -0.5));
            _geometry.pushVertex(glm::vec3(-0.5, -0.5, -0.5));

            _geometry.pushUv(glm::vec2(0.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 1.0f));
            _geometry.pushUv(glm::vec2(0.0f, 1.0f));

            _geometry.setColor(glm::vec4(0, 1, 1, 1));

            _geometry.pushVertex(glm::vec3(0.5, -0.5, -0.5));
            _geometry.pushVertex(glm::vec3(0.5, 0.5, -0.5));
            _geometry.pushVertex(glm::vec3(0.5, 0.5, 0.5));
            _geometry.pushVertex(glm::vec3(0.5, -0.5, 0.5));

            _geometry.pushUv(glm::vec2(0.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 1.0f));
            _geometry.pushUv(glm::vec2(0.0f, 1.0f));

            _geometry.setColor(glm::vec4(1, 0, 1, 1));

            _geometry.pushVertex(glm::vec3(-0.5, -0.5, 0.5));
            _geometry.pushVertex(glm::vec3(-0.5, 0.5, 0.5));
            _geometry.pushVertex(glm::vec3(-0.5, 0.5, -0.5));
            _geometry.pushVertex(glm::vec3(-0.5, -0.5, -0.5));

            _geometry.pushUv(glm::vec2(0.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 1.0f));
            _geometry.pushUv(glm::vec2(0.0f, 1.0f));

            _geometry.setColor(glm::vec4(0, 1, 0, 1));

            _geometry.pushVertex(glm::vec3(0.5, 0.5, 0.5));
            _geometry.pushVertex(glm::vec3(0.5, 0.5, -0.5));
            _geometry.pushVertex(glm::vec3(-0.5, 0.5, -0.5));
            _geometry.pushVertex(glm::vec3(-0.5, 0.5, 0.5));

            _geometry.pushUv(glm::vec2(0.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 0.0f));
```

```
            _geometry.pushUv(glm::vec2(1.0f, 1.0f));
            _geometry.pushUv(glm::vec2(0.0f, 1.0f));

            _geometry.setColor(glm::vec4(0, 0, 1, 1));

            _geometry.pushVertex(glm::vec3(0.5, -0.5, -0.5));
            _geometry.pushVertex(glm::vec3(0.5, -0.5, 0.5));
            _geometry.pushVertex(glm::vec3(-0.5, -0.5, 0.5));
            _geometry.pushVertex(glm::vec3(-0.5, -0.5, -0.5));

            _geometry.pushUv(glm::vec2(0.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 0.0f));
            _geometry.pushUv(glm::vec2(1.0f, 1.0f));
            _geometry.pushUv(glm::vec2(0.0f, 1.0f));

            // Tres important, on n'oublie pas de build la geometrie pour envoyer ses
informations a la carte graphique
            _geometry.build();
```

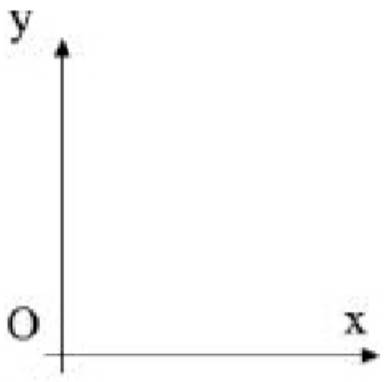
2.2.1 – Dessin du cube

```
/ On bind la texture pour dire que l'on veux l'utiliser
_texture.bind();
// Et on dessine notre cube
_geometry.draw(shader, getTransformation(), GL_QUADS);
```

3 - Programmation 3D

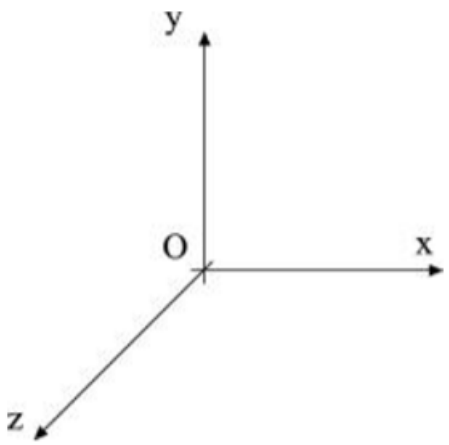
Des notions de géométrie dans l'espace sont nécessaires afin de pouvoir manipuler correctement vos éléments dans un espace en trois dimensions (3D).

Un espace en deux dimensions (2D) est représenté par un repère orthonormé xy. Il est dit orthonormal car chaque vecteur (axe) à la même unité. Cet espace est un plan.



Repère orthonormé xy

Quant à un espace en trois dimensions, nous lui ajoutons une dimension supplémentaire qui sera la profondeur. Sa représentation est alors un repère orthonormé xyz.



Repère orthonormé xyz

3.1 – Vecteur

Un vecteur sert à représenter une orientation ou un point. On peut le considérer comme une ligne partant de l'origine et allant jusqu'à ses coordonnées, ou simplement un point se trouvant en ses coordonnées. Il possède d'autant de composantes (coordonnées) qu'il y a de dimensions dans l'espace dans lequel il est représenté.

Les vecteurs nous serviront, entre autre, à définir des translations ou des axes de rotation. Dans le cadre d'un projet 3D, des vecteurs à 3 dimensions doivent être utilisés.

Exemple : Soit un vecteur a(x,y) défini arbitrairement dans un espace en deux dimensions, il a pour coordonnées :

x =6
y =5

Si nous voulons adapter notre vecteur pour un espace en trois dimensions, nous y ajoutons une nouvelle composante que nous appelons z. Le vecteur se note alors a(x,y,z) dont les composantes ont pour valeur :

x =6
y =5
z =0

Pour représenter des vecteurs dans le code, nous utiliserons les objets glm::vec3 et glm::vec2 de la bibliothèque de mathématiques *glm.*

3.2 Matrice

Les matrices servent à se déplacer dans un monde en 3D. Celles-ci peuvent paraître un peu rebutantes au début mais s'avèrent au final particulièrement pratiques. Pour placer et animer un objet dans l'espace, il va falloir lui faire subir tout un tas de transformations, que ce soit des translations ou des rotations. Une formule est donc nécessaire pour chaque transformation appliquée à l'objet. La puissance des matrices vient du fait que toutes informations et transformations peuvent être réunies en une seule et unique matrice!

Les matrices se présentent sous la forme de tableaux à deux dimensions de quatre par quatre. Pourquoi quatre coordonnées dans un espace à seulement trois dimensions? Tout simplement parce qu'en 3D on utilise régulièrement les coordonnées homogènes. Mais rassurez-vous nous n'aurons pas à rentrer dans le détail mathématique de ces dernières.

Pour représenter une matrice, nous utiliserons aussi la bibliothèque *glm* et l'objet glm::mat4.

3.2.1 Matrice identité

La matrice identité a comme particularité de ne contenir aucune transformation. C'est cette matrice que l'on utilise comme base pour toute transformation effectuée sur un point. Elle se caractérise par sa diagonale contenant uniquement des 1, le reste de ses valeurs étant 0.

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Pour créer une matrice identité, il suffit d'appeler le constructeur de l'objet glm::mat4 comme ceci :

```
glm::mat4(1) ;
```

3.2.2 - Matrices de translation, rotation et homothétie

Il existe une matrice particulière pour décrire chacune des transformations possibles dans l'espace en 3D. C'est grâce à ces matrices que les transformations sont faites. Heureusement celles-ci sont calculées automatiquement par la bibliothèque de mathématiques *glm*.

Une translation est une transformation linéaire le long d'un axe. Elle permet de déplacer un objet.

Pour effectuer une translation sur un objet avec *glm*, nous utiliserons la fonction glm::translate :

glm::mat4 glm ::translate(glm ::mat4 const &matrix, glm::vec3 const &vector);

Cette fonction nous renverra la matrice « matrix » après lui avoir fait subir une translation de « vector ».

Une rotation est une transformation linéaire autour d'un axe. Elle permet de pivoter un objet.

Pour effectuer une rotation, nous utiliserons la fonction glm::rotate :

glm::mat4 glm ::rotate(glm ::mat4 const &matrix, float angle, glm ::vec3 const &axis) ;

Cette nous renverra la matrice « matrix » après lui avoir fait subir une rotation de l'angle « angle » selon l'axe « axis ».

Une homothétie est une transformation linéaire sur tous les axes de manière homogène. Elle permet de mettre à l'échelle un objet.

Pour effectuer une mise à l'echelle, nous utiliserons la fonction glm::scale :

glm ::mat4 glm ::scale(glm ::mat4 const &matrix, glm ::vec3 const &scale) ;

Cette nous renverra la matrice « matrix » après lui avoir fait subir une mise à l'échelle de « scale ». Par exemple, si « scale » est égale à glm::vec3(2, 2, 2), l'objet sera deux fois plus grand.

4 - OpenGL : les bases

La bibliothèque simplifie certaines opérations assez difficiles que vous aurez besoin de faire. Cependant, d'autres opérations, plus simples, sont à votre charge.

Avant de nous lancer dans les différentes instructions d'OpenGL, une courte introduction au pipeline graphique de celui-ci est nécessaire.

4.1 - Le pipeline graphique

Le pipeline graphique désigne le processus permettant un rendu graphique à partir de données brutes. Nous ne détaillerons pas complètement le fonctionnement du pipeline d'OpenGL, seule la manière dont sont gérées les coordonnées des vertices nous intéresse. Ce qu'il faut d'abord savoir : vous ne dessinez que des vertices c'est-à-dire que vous demandez à OpenGL de poser des vertices à des coordonnées précises dans l'espace. Un ensemble de vertices formera une figure.

Plusieurs transformations sont effectuées sur les coordonnées de nos objets avant qu'ils ne soient affichés à l'écran. Les objets subissent un certain nombre de changement d'espace à travers ces transformations. En effet, les coordonnées des vertices sont placées dans l'espace de l'objet par le développeur. A travers les transformations, OpenGL détermine où se trouveront les vertices dans la fenêtre de rendu.

Le pipeline est constituée de petits programmes exécutés par la carte graphique appelés « shaders ». Vous n'avez pas vraiment besoin d'en comprendre le fonctionnement, pour autant il serait intéressant d'aller se renseigner sur le sujet pour proposer des graphismes un peu plus poussés ;)

Vous allez devoir instancier un objet gdl::Shader pour pouvoir dessiner tous vos objets. Pour cela, vous devez charger les deux fichiers « basic.fp » et « basic.vp » qui vous sont fournis dans le dossier « shader » de la librairie.

Pour les charger, rien de plus simple ! Il vous suffit d'appeler les méthodes Shader::load et Shader ::build comme ceci :

```
// On cree un shader, petit programme permettant de dessiner nos objets a l'ecran
if (!_shader.load("./Shaders/basic.fp", GL_FRAGMENT_SHADER) // le fragment shader se
charge de dessiner les pixels
        || !_shader.load("./Shaders/basic.vp", GL_VERTEX_SHADER) // le vertex shader
s'occupe de projeter les points sur l'ecran
       || !_shader.build()) // il faut ensuite compiler son shader
       return false;
```

4.2 – La caméra

Pour passer des coordonnées 3D définies par le développeur aux coordonnées 2D du point sur l'écran, une caméra doit être placée.

Pour cela, on doit définir deux matrices :

Premièrement une matrice indiquant la position et l'orientation de la caméra.
Pour cela, nous utiliserons la librairie *glm* qui permet de générer ces matrices grâce à la fonction glm::lookAt.

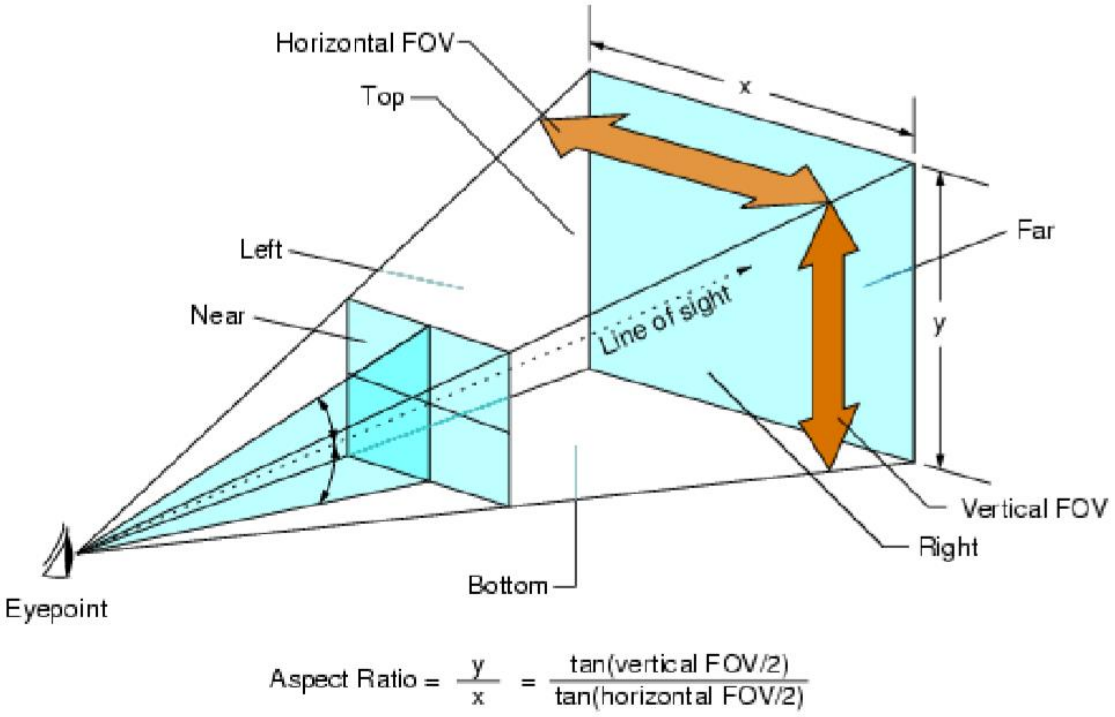
Cette dernière prend 3 paramètres qui sont :

1- La position de la caméra
2- Le point que l'on veut regarder
3- Un vecteur définissant dans quelle direction est le haut de la camera (la plupart du temps, cela sera 0, 1, 0)

Par exemple, pour construire une caméra placée aux coordonnées 0, 10, -10 et regardant au centre, nous ferons :

```
glm::lookAt(glm::vec3(0, 10, -10), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));
```

La seconde matrice correspond à la pyramide projection de la caméra (frustum en anglais), c'est-à-dire à la manière dont les points en 3D seront projetés sur l'écran.



$$\text{Aspect Ratio} = \frac{y}{x} = \frac{\tan(\text{vertical FOV}/2)}{\tan(\text{horizontal FOV}/2)}$$

Représentation schématique du frustum.
Source : http://techpubs.sgi.com

Pour générer cette matrice, nous utiliserons la fonction glm::perspective qui prend quatre paramètres :

1- L'angle d'ouverture de la caméra (field of view ou FOV en anglais, qui est d'environ 60° pour l'œil humain par exemple)
2- Le ratio de la fenêtre (donc si vous ouvrez une fenêtre de 1280 pixels de largeur par 720 de largeur, votre aspect ratio sera de 1280.0f / 720.0f ce qui correspond à 16.0f / 9.0f)
3- La distance minimum à partir de laquelle la caméra est capable de voir (Near plane)
4- La distance maximale à laquelle la caméra peut voir (Far plane)

Pour positionner la caméra, vous devez passer ces deux matrices au *shader* qui va dessiner vos objets.

```
// La projection de la camera correspond a la maniere dont les objets vont etre
dessine a l'ecran
projection = glm::perspective(60.0f, 1280.0f / 720.0f, 0.1f, 100.0f);
// La transformation de la camera correspond a son orientation et sa position
// La camera sera ici situee a la position 0, 20, -100 et regardera vers la position
0, 0, 0
transformation = glm::lookAt(glm::vec3(0, 10, -30), glm::vec3(0, 0, 0), glm::vec3(0,
1, 0));

// On doit toujours binder le shader avant d'appeler les methodes setUniform
_shader.bind();
_shader.setUniform("view", transformation);
_shader.setUniform("projection", projection);
```

5 - Les modèles :

Pour charger un modèle, nous utiliserons l'objet gdl::Model.
Il permet de charger n'importe quel objet dans les formats obj, fbx et collada (.fbx, .obj et .dae).

Pour ceci, vous utiliserez la méthode Model::load qui permet de charger un modèle :

```
bool            load(std::string const &path);
```

Ensuite, vous pouvez le dessiner grâce à la méthode Model::draw :

```
void            draw(AShader &shader, glm::mat4 const &transform, double deltaTime);
```

Celle-ci prend en paramètres le shader avec lequel le modèle vas être dessiné, la matrice de transformation de ce modèle ainsi que le temps écoulé depuis la dernière image dessinée (Input::getElpasedTime()).

Avant de dessiner un modèle, vous pouvez choisir une animation à jouer avec la méthode Model::setCurrentAnim :

```
bool            setCurrentAnim(int stack, bool loop = true);
bool            setCurrentAnim(std::string const &name, bool loop = true);
```

Cette méthode permet de sélectionner une animation à jouer lors du prochain draw soit par son nom, soit par son index. Le paramètre « loop » permet de préciser si on veut jouer l'animation en boucle ou non.

Vous pouvez aussi choisir de découper une animation déjà existante en plusieur sous animations grace à la méthode createSubAnim :

```
bool            createSubAnim(int stack, std::string const &subAnimName,
                              int frameStart, int frameEnd);
bool            createSubAnim(std::string const &name, std::string const &subAnimName,
                              int frameStart, int frameEnd);
```

Le premier paramètre permet de sélectionner l'animation à découper, le second permet de choisir un nom pour la sous animation créée puis les deux derniers paramètres correspondent respectivement à la frame de début et à la frame de fin de la sous animation.

6- Un Cube certes mais qui bouge !!!

Nous avons maintenant tous les éléments nécessaires pour commencer notre bomberman.
Vous devriez maintenant être capable de réaliser un cube qui bouge grâce aux flèches directionnelles.

Dans le cas contraire, le code qui suit devrait vous aider ;)

Notre abstraction pour les objets :

```
// Permet d'inclure la SDL 2
#include <SdlContext.hh>
```


```
// Inclus la bibliotheque de mathematiques
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

// La classe abstraite representant un objet avec sa position, sa rotation et son echelle
class AObject
{
public:
	AObject() :
		_position(0, 0, 0), // On initialise la position a 0
		_rotation(0, 0, 0), // pareil pour la rotation
		_scale(1, 1, 1) // l'echelle est mise a 1
	{
	}

	virtual ~AObject()
	{}

	// La fonction initialize charge l'objet ou le construit
	virtual bool initialize()
	{
		return (true);
	}

	// La fonction update sert a gerer le comportement de l'objet
	virtual void update(gdl::Clock const &clock, gdl::Input &input)
	{}

	// La fonction draw sert a dessiner l'objet
	virtual void draw(gdl::AShader &shader, gdl::Clock const &clock) = 0;



	void translate(glm::vec3 const &v)
	{
		_position += v;
	}

	void rotate(glm::vec3 const& axis, float angle)
	{
		_rotation += axis * angle;
	}

	void scale(glm::vec3 const& scale)
	{
		_scale *= scale;
	}

	glm::mat4  getTransformation()
	{
		glm::mat4  transform(1); // On cree une matrice identite

		// On applique ensuite les rotations selon les axes x, y et z
		transform = glm::rotate(transform, _rotation.x, glm::vec3(1, 0, 0));
```


```
                transform = glm::rotate(transform, _rotation.y, glm::vec3(0, 1, 0));
                transform = glm::rotate(transform, _rotation.z, glm::vec3(0, 0, 1));
                // On effectue ensuite la translation
                transform = glm::translate(transform, _position);
                // Et pour finir, on fait la mise a l'echelle
                transform = glm::scale(transform, _scale);
                return (transform);
        }

protected:
        glm::vec3               _position;
        glm::vec3               _rotation;
        glm::vec3               _scale;

};
```

Nous allons donc faire à présent notre classe Cube héritant de AObject :

```
class       Cube : public AObject
{
private:
        // La texture utilisee pour le cube
        gdl::Texture            _texture;
        // La geometrie du cube
        gdl::Geometry                   _geometry;
        // La vitesse de deplacement du cube
        float                           _speed;

public:
        Cube() { }
        virtual ~Cube() { }

        virtual bool  initialize()
        {
                _speed = 10.0f;

                // On charge la texture qui sera affichee sur chaque face du cube
                if (_texture.load("./assets/texture.tga") == false)
                {
                        std::cerr << "Cannot load the cube texture" << std::endl;
                        return (false);
                }

                // on set la color d'une premiere face
                _geometry.setColor(glm::vec4(1, 0, 0, 1));
                // tout les pushVertex qui suivent seront de cette couleur

                // On y push les vertices d une premiere face
                _geometry.pushVertex(glm::vec3(0.5, -0.5, 0.5));
                _geometry.pushVertex(glm::vec3(0.5, 0.5, 0.5));
                _geometry.pushVertex(glm::vec3(-0.5, 0.5, 0.5));
                _geometry.pushVertex(glm::vec3(-0.5, -0.5, 0.5));

                // Les UVs d'une premiere face
                _geometry.pushUv(glm::vec2(0.0f, 0.0f));
                _geometry.pushUv(glm::vec2(1.0f, 0.0f));
                _geometry.pushUv(glm::vec2(1.0f, 1.0f));
                _geometry.pushUv(glm::vec2(0.0f, 1.0f));

                // ETC ETC
                _geometry.setColor(glm::vec4(1, 1, 0, 1));
```


```
        _geometry.pushVertex(glm::vec3(0.5, -0.5, -0.5));
        _geometry.pushVertex(glm::vec3(0.5, 0.5, -0.5));
        _geometry.pushVertex(glm::vec3(-0.5, 0.5, -0.5));
        _geometry.pushVertex(glm::vec3(-0.5, -0.5, -0.5));

        _geometry.pushUv(glm::vec2(0.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 1.0f));
        _geometry.pushUv(glm::vec2(0.0f, 1.0f));

        _geometry.setColor(glm::vec4(0, 1, 1, 1));

        _geometry.pushVertex(glm::vec3(0.5, -0.5, -0.5));
        _geometry.pushVertex(glm::vec3(0.5, 0.5, -0.5));
        _geometry.pushVertex(glm::vec3(0.5, 0.5, 0.5));
        _geometry.pushVertex(glm::vec3(0.5, -0.5, 0.5));

        _geometry.pushUv(glm::vec2(0.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 1.0f));
        _geometry.pushUv(glm::vec2(0.0f, 1.0f));

        _geometry.setColor(glm::vec4(1, 0, 1, 1));

        _geometry.pushVertex(glm::vec3(-0.5, -0.5, 0.5));
        _geometry.pushVertex(glm::vec3(-0.5, 0.5, 0.5));
        _geometry.pushVertex(glm::vec3(-0.5, 0.5, -0.5));
        _geometry.pushVertex(glm::vec3(-0.5, -0.5, -0.5));

        _geometry.pushUv(glm::vec2(0.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 1.0f));
        _geometry.pushUv(glm::vec2(0.0f, 1.0f));

        _geometry.setColor(glm::vec4(0, 1, 0, 1));

        _geometry.pushVertex(glm::vec3(0.5, 0.5, 0.5));
        _geometry.pushVertex(glm::vec3(0.5, 0.5, -0.5));
        _geometry.pushVertex(glm::vec3(-0.5, 0.5, -0.5));
        _geometry.pushVertex(glm::vec3(-0.5, 0.5, 0.5));

        _geometry.pushUv(glm::vec2(0.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 1.0f));
        _geometry.pushUv(glm::vec2(0.0f, 1.0f));

        _geometry.setColor(glm::vec4(0, 0, 1, 1));

        _geometry.pushVertex(glm::vec3(0.5, -0.5, -0.5));
        _geometry.pushVertex(glm::vec3(0.5, -0.5, 0.5));
        _geometry.pushVertex(glm::vec3(-0.5, -0.5, 0.5));
        _geometry.pushVertex(glm::vec3(-0.5, -0.5, -0.5));

        _geometry.pushUv(glm::vec2(0.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 0.0f));
        _geometry.pushUv(glm::vec2(1.0f, 1.0f));
        _geometry.pushUv(glm::vec2(0.0f, 1.0f));

        // Tres important, on n'oublie pas de build la geometrie pour envoyer ses informations a la carte
graphique
        _geometry.build();
        return (true);
```


```
	}


	// Ici le cube bougera avec les fleches du clavier
	virtual void update(gdl::Clock const &clock, gdl::Input &input)
	{
		// On multiplie par le temps ecoule depuis la derniere image pour que la vitesse ne depende pas
de la puissance de l'ordinateur
		if (input.getKey(SDLK_UP))
			translate(glm::vec3(0, 0, -1) * static_cast<float>(clock.getElapsed()) * _speed);
		if (input.getKey(SDLK_DOWN))
			translate(glm::vec3(0, 0, 1) * static_cast<float>(clock.getElapsed()) * _speed);
		if (input.getKey(SDLK_LEFT))
			translate(glm::vec3(-1, 0, 0) * static_cast<float>(clock.getElapsed()) * _speed);
		if (input.getKey(SDLK_RIGHT))
			translate(glm::vec3(1, 0, 0) * static_cast<float>(clock.getElapsed()) * _speed);
	}

	virtual void draw(gdl::AShader &shader, gdl::Clock const &clock)
	{
		(void)clock;
		// On bind la texture pour dire que l'on veux l'utiliser
		_texture.bind();
		// Et on dessine notre cube
		_geometry.draw(shader, getTransformation(), GL_QUADS);
	}

};
```

Mettons nous maintenant sur la création de notre GameEngine :

```
#pragma once

#include <Game.hh>
#include <SdlContext.hh>

#include "AObject.hpp"

/*
On cree sa class GameEngine qui herite de game
*/
class GameEngine : public gdl::Game
{
public:
	GameEngine::GameEngine()
	{
	}

	bool GameEngine::initialize()
	{
		if (!_context.start(800, 600, "My bomberman!")) // on cree une fenetre
			return false;

		// On active le test de profondeur d'OpenGL pour que les pixels que l'oeil ne voit pas ne
s'affichent pas
		glEnable(GL_DEPTH_TEST);

		// On cree un shader, petit programme permettant de dessiner nos objets a l'ecran
```

```
		if (!_shader.load("./Shaders/basic.fp", GL_FRAGMENT_SHADER) // le fragment shader se charge de dessiner les pixels
			|| !_shader.load("./Shaders/basic.vp", GL_VERTEX_SHADER) // le vertex shader s'occupe de projeter les points sur l'ecran
			|| !_shader.build()) // il faut ensuite compiler son shader
			return false;

		// On place ensuite la camera (sa projection ainsi que sa transformation)
		glm::mat4  projection;
		glm::mat4  transformation;

		// La projection de la camera correspond a la maniere dont les objets vont etre dessine a l'ecran
		projection = glm::perspective(60.0f, 800.0f / 600.0f, 0.1f, 100.0f);
		// La transformation de la camera correspond a son orientation et sa position
		// La camera sera ici situee a la position 0, 20, -100 et regardera vers la position 0, 0, 0
		transformation = glm::lookAt(glm::vec3(0, 10, -30), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));

		// On doit toujours binder le shader avant d'appeler les methodes setUniform
		_shader.bind();
		_shader.setUniform("view", transformation);
		_shader.setUniform("projection", projection);

		// On va ensuite creer un cube que l'on va ajouter a la liste d'objets
		AObject				*cube = new Cube();

		if (cube->initialize() == false)
			return (false);
		_objects.push_back(cube);

		return true;
	}

	bool GameEngine::update()
	{
		// Si la touche ECHAP est appuyee ou si l'utilisateur ferme la fenetre, on quitte le programme
		if (_input.getKey(SDLK_ESCAPE) || _input.getInput(SDL_QUIT))
			return false;
		// Mise a jour des inputs et de l'horloge de jeu
		_context.updateClock(_clock);
		_context.updateInputs(_input);
		// Mise a jour des differents objets
		for (size_t i = 0; i < _objects.size(); ++i)
			_objects[i]->update(_clock, _input);
		return true;
	}

	void GameEngine::draw()
	{
		// On clear l'ecran
		glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
		// pour utiliser un shader (pour que ce soit ce dernier qui dessine la geometrie) il faut le binder.
		// Un seul shader peut etre actif en meme temps
		_shader.bind();
```


```
                // On dessine tous les objets composant la scene
                for (size_t i = 0; i < _objects.size(); ++i)
                        _objects[i]->draw(_shader, _clock);

                // On met ensuite a jour l'ecran pour que ce que l'on a dessine s'affiche
                _context.flush();
        }

        GameEngine::~GameEngine()
        {
                // N'oublions pas de supprimer les objets une fois le programme termine!
                for (size_t i = 0; i < _objects.size(); ++i)
                        delete _objects[i];
        }

private:
        gdl::SdlContext                 _context;
        gdl::Clock                      _clock;
        gdl::Input                      _input;
        gdl::BasicShader                _shader;
        std::vector<AObject*>           _objects;

};
```

Il est maintenant l'heure de tester tout ca.

```
#include <cstdlib>

#include "GameEngine.hpp"

int                     main()
{
        // On cree son engine
        GameEngine engine;

        if (engine.initialize() == false)
                return (EXIT_FAILURE);
        while (engine.update() == true)
                engine.draw();
        return EXIT_SUCCESS;
}
```

Félicitation vous voilà avec un joli cube coloré qui bouge.

Ce petit guide d'introduction est maintenant terminé, nous espérons que vous apprécierez ce premier projet de jeu 3D, n'hésitez pas à vous faire plaisir et à nous remonter des problèmes que vous pourriez rencontrer sur la lib.

Bonne chance à tous ☺