Cours OpenGL - GDL-GL Avril 2014

Paul Baron paul.baron@epitech.eu

Alexandre Zieder alexandre.zieder@epitech.eu

GameDevLab gamelab@epitech.eu

1 – Library usage

1.1 - header files

Each library's class is declared in a header file. They are all in a namespace named 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>
```

To use OpenGL, you must include a header file that we provide and who takes care of including the necessary OpenGL libraries:

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

To use glm, includethe following header files:

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

1.1.2 – Compilation

Here is the compile line that will allow you to compile your program from the folder LibBomberman_linux_x64 we provide:

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

-lGL : the OpenGL library
-lGLEW : OpenGL Extension Wrangler
-lgdl_gl : GDL_GL library
-ldl : For dynamic library load
-lrt : time management library.
-lfbxsdk : 3D models management library
-lSDL2 : SDL2 library
-lpthread : pthread library

In order to run the code you will need to modify the LD_LIBRARY_PATH environment variable this way:

export LD_LIBRARY_PATH=lib_folder_path/libs/

1.2 – First step

1.2.1 – Game Loop

A game loop is an infinite loop that includes several tasks:

• inputs update

• time update

• objects behavior update

• graphics draw

To implement this loop, we provide you gdl :: Game class that contains the following pure virtual methods:

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

- In Game :: initialize method, you load the various "assets" (component of your game, that is to say, 3D models, textures and sounds)
- In Game :: update method, you update the inputs, the main clock and the behavior of your game (player movements, etc.)
- In the draw method, you draw the different elements of your game

It is up to you to create a class that inherits from gdl :: Game that will implement these methods.

1.2.2 – Start on OpenGL context

Before any graphic manipulation, it is essential to open what is called a OpenGL context, that is to say, initialize the library. This initialization is done when opening a window.

We use the graphic library SDL for that.

To open the context, we provide a SdlContext class that allows you to initialize everything that you will need to have a window in which you can draw afterwards.

This class is also responsible for updating the inputs and the main game clock.

It has the following methods :

```
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;
```

- SdlContext :: start method allows you to open a window. It takes as parameters width, height and name
- The SdlContext :: stop method allows you to close the window
- The SdlContext :: updateInputs method will update the inputs (clicks and mouse movements, keyboard events ...)
- The SdlContext :: updateClock method will update the main clock
- The SdlContext :: flush method is used to display once all objects are drawn

1.2.3 – The game clock

The gdl :: Clock class allows you to retrieve the elapsed time between each loop (frame) with the Clock:: getElpasedTime that allow you to manage the game time.

For this, you need to instantiate an object gdl :: Clock and update it each frame by calling the SdlContext :: updateClock function.

1.2.4 - Inputs

The gdl::Input class allows you to retrieve the interactions of the player (mouse movement, key on the keyboard, closing the window ...)

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

This class is updated by the SdlContext ::updateInputs.

It has the following methods to recover the 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);
```

- The Input ::getMousePosition method allows you to retrieve the position of the mouse in your window.
- Input::getMouseDelta allows you to retrieve the mouse movement since the last call to SdlContext::updateInputs
- Input::getMouseWheel allows you to recover the movement of the mouse wheel since the last call to SdlContext::updateInputs
- Input::getInput indicates whether an input has occurred or not
- Input::getKey indicates whether a key is pressed or not

The input parameter is the enums of the SDL (for instance SDLK_UP or SDK_DOWN for Input::getKey and SDL_Quit for Input::getInput).

2 – Primitives

2.1 – gdl::Geometry object

In 3D programming we use vertices. These "points" which, when connected, form a polygon. The class that will generate and store the geometry is the gdl::Geometry class.

The following member function allows you to add a vertex to your instance of the Geometry object. It takes as parameter a glm::vec3 matching a 3D point in space and that takes as parameters relative positions.

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

To display the geometries drawn, you must apply a texture. For that you need to load an image. We use the member function Texture::load of the gdl::Texture class.

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

example:

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

Warning the library only supports .tga textures

Each time you push a side of a geometry you have to give the relative coordinates of the texture you want to apply with the following method:

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

This function takes a glm::vec2 which takes two relative positions (0, 0 for the top left corner of your image and 1, 1 for the bottom right corner).

After drawing the vertices we must generate the geometry. For this the following member function is used:

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

It can then be displayed with the following member function that takes as parameter a gdl::AShader (discussed later in this tutorial) and a GLenum:

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

The enum parameter will determine how vertices are processed. Among the values of the enum, we can remember:

- GL_TRIANGLES
- GL_QUADS
- GL_LINE_STRIP

GL_TRIANGLES allows to draw a triangle from three points. OpenGL will take each triplet of vertices in the vertex group that you pushed in the Geometry object to draw triangles.

GL_QUADS requires a quadruplet of vertices to draw a quadrilateral.

GL_LINE_STRIP allow you to draw lines between each point.

To use a texture in the draw, you must use the bind method before calling Geometry :: draw. You can not draw geometry without texture.

2.2 - Exemple

2.2.1 – Cube creation

```
// Gdl :: Geometry object is instantiated
gdl::Geometry _geometry;

// We load the texture which will be displayed on each face of the cube
if (_texture.load("./assets/texture.tga") == false)
{
	std::cerr << "Cannot load the cube texture" << std::endl;
	return (false);
}

// We set the color of the first face
_geometry.setColor(glm::vec4(1, 0, 0, 1));
// all the following push vertex will be with the same color

// We push the vertices of the first 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));

// Then the UVs
_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));

            // Don't forget to build the geometry to send its informations to the
graphics card
            _geometry.build();
```

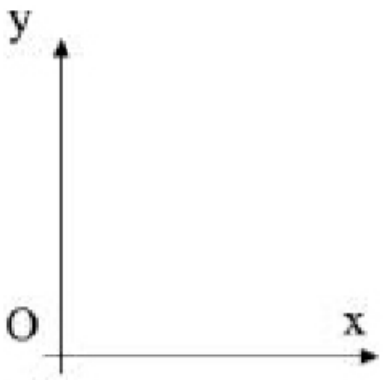
2.2.1 – Draw a cube

```
/ We bind the texture to be able to use it
_texture.bind();
// Finally we draw our cube
_geometry.draw(shader, getTransformation(), GL_QUADS);
```

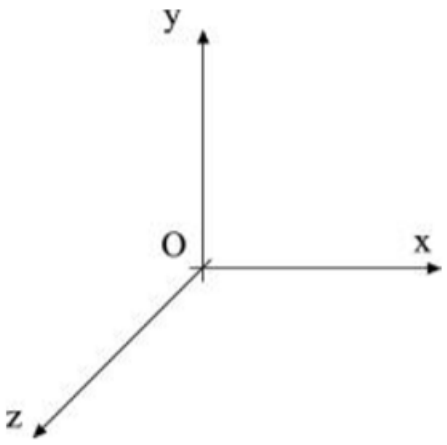
3 - 3D programming

Concepts of geometry in space are necessary in order to properly handle your items in a three-dimensional space (3D).

Space in two dimensions (2D) is represented by an xy orthonormal system. It is said because each orthonormal vector (axis) has the same unit. This space is a plane.



For a three-dimensional space, we add an extra dimension to be depth. Representation is then an orthonormal system xyz.



3.1 – Vector

A vector is used to represent a point or orientation. It can be considered as a line from the origin up to its coordinates, or simply a point lying in the coordinates. It has as much components (coordinates) as there are dimensions in the space in which it is shown.

We will use the vectors to define translations or rotation axes. As part of a 3D project , 3-dimensional vectors are going to be used.

Example: A vector (x, y) arbitrarily defined in a two-dimensional space, it has coordinates:

x =6
y =5

If we want to adapt our vector for a three-dimensional space, we add a new component that we call z (the depth). The vector is then denoted (x, y, z) whose components are set to:

x =6
y =5
z =0

To represent vectors in the code, we use glm :: vec3 and glm :: vec2 objects from the glm library.

3.2 Matrices

Matrices are used to move in a 3D world . These may seem a little off-putting at first but in the end are particularly useful . To place and animate an object in space , we will have him undergo a lot of changes , whether translations or rotations. A formula is used for each transformation applied to the object . The power of matrices is that all information and transformations can be combined into a single matrix.

The matrices are in the form of two-dimensional arrays of four by four . Why four coordinates in a space of only three dimensions? Simply because they are 3D homogeneous coordinates regularly used. But don't worry we will not have to enter the mathematical details.

To represent a matrix, we will use the glm library and object glm :: mat4 .

3.2.1 Identity Matrix

The identity matrix has the particularity of not containing any transformation. This matrix is used as the basis for any transformation performed on an item. It is characterized by its diagonal containing only one, the rest of its values is 0.

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

To create an identity matrix, simply call the constructor of the object glm :: mat4 like this:

glm::mat4(1) ;

3.2.2 – Translation matrices, rotation et homothety

There is a particular matrix to describe each of the possible transformations in 3D space. It is thanks to these matrices that transformations are made. Fortunately they are automatically calculated by the glm library.

A translation is a linear transformation along an axis. It allows you to move an object.

To perform a translation on a glm object, we use the glm :: translate function:

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

This function will return the matrix « matrix » after applying to it the « vector » translation.

A rotation is a linear transformation around an axis. It allows you to rotate an object.

To apply a rotation on an object we use the glm::rotate function:

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

This function will return the matrix « matrix » after applying the rotation « angle » according to the axis “axis”.

An homothety is a linear homogeneous transformation on all axes. It allows to scale an object.

To apply a scale we use the glm::scale function :

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

This function will return the matrix « matrix » after applying the scale “scale”. For example, if "scale" is equal to glm::vec3 (2, 2, 2), the object will be twice bigger.

4 - OpenGL : basis

The library simplifies some difficult tasks that you will need to do. However ; other more simple operation are on you.

Before we get to the various OpenGL operations , a short introduction to graphical pipeline is necessary.

4.1 – Graphical pipeline

The graphics pipeline is the process that "makes" the graphics from raw data. We will not explain all the detail of this process, only the vertex coordinate management is interesting for us. What you need to know first : you only draw vertices that is to say that you ask OpenGL to put vertices to specific coordinates in space . A set of vertices form a figure.

Several transformations are performed on object's coordinates before the rendering. Objects suffer several space change through these transformations. The coordinates of the vertices are placed in the object space by the developer. Through transformations, OpenGL determines where will be the vertices in the render window.

The pipeline consists of small programs called "shaders" executed by the graphics card. You do not need to understand how it works but it would be interesting to learn about the subject to provide a bit more advanced graphics;)

You will have to instantiate a gdl::Shader object in order to draw all your objects. To do this, you must load two files "basic.fp" and "basic.vp" that are provided in the "shader" folder of the library.

To load them you just need to call the Shader ::load and Shader ::build methods :

```
if (!_shader.load("./Shaders/basic.fp", GL_FRAGMENT_SHADER) // This one draw the
pixels
        || !_shader.load("./Shaders/basic.vp", GL_VERTEX_SHADER) // This one put the
points on the screen
        || !_shader.build()) // Then you compile the shader
        return false;
```

4.2 – The camera

To go from developer-definded 3D coordinates to 2D screen coordinates , you must have a camera.

For this we have to define two matrices :

First a matrix indicating the position and orientation of the camera.

For this we use the glm library to generate these matrices through glm :: lookAt function.

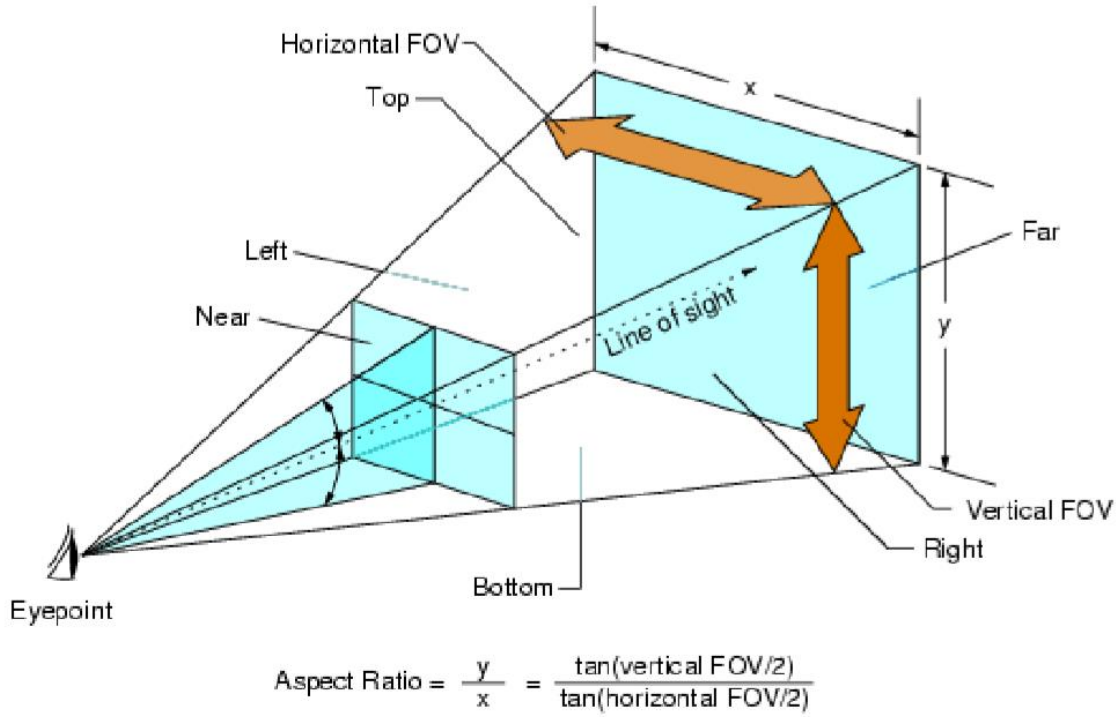
It takes 3 parameters :

1- Camera position
2- The point we want to look at
3- A vector to indicate the up direction of the camera (most of the time, it will be 0, 1, 0)

For example, to build a camera placed at coordinates 0, 10, -10 and looking at the center, we will write:

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

The second matrix corresponds to the projection of the camera pyramid(frustum), that is to say how the 3D point wil be projected on the screen.



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

Source : http://techpubs.sgi.com

To generate this matrix, we use the glm :: perspective function that takes four parameters:

1- The field of view or FOV, which is approximately 60° for human eye
2- The window's ratio (so if you open a window of 1280 pixels wide by 720 high, your aspect ratio will 1280.0f / 720.0f which is 16.0f / 9.0f)
3- The minimal distance where something can be seen. (Near plane)
4- The maximal distance where something can be seen. (Far plane)

To position the camera, you must pass these two matrices to the shader that will draw your objects.

```
projection = glm::perspective(60.0f, 1280.0f / 720.0f, 0.1f, 100.0f);
transformation = glm::lookAt(glm::vec3(0, 10, -30), glm::vec3(0, 0, 0), glm::vec3(0,
1, 0));

// we have to bind the shader before any call to the setUniform method
_shader.bind();
_shader.setUniform("view", transformation);
_shader.setUniform("projection", projection);
```

5 -3D models :

To load a model we use the gdl::Model class.
It allows to load any models in obj ; fbx and collada format(.fbx, .obj et .dae)

To do that you will have to use the Model ::load methode:

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

Then you can draw it with the following method Model ::draw :

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

It takes the shader your going to draw with, the transformation matrix of the model and the elapsed time since the last draw. (Input::getElpasedTime()).

Before drawing a model you can choose a animation to play with the Model ::setCurrentAnim methode :

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

This method allows you to select an animation to play at the next call to the draw function. You can use the name or the index of the animation. The loop parameter determines if you want the animation to loop or not.

You can also choose to cut an existing animation into several animations through createSubAnim method:

```
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);
```

The first parameter is the name of the animation you want to cut , the second is the name of the sub animation you want to create , the third is the begin frame and the last is the end frame.

6- A moving cube!!!

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 :

```
// allow to use SDL 2
#include <SdlContext.hh>
#include "BasicShader.hh"

// allow to use glm types
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

// Abstract classe for objects
class AObject
{
public:
		AObject() :
			_position(0, 0, 0),
			_rotation(0, 0, 0),
			_scale(1, 1, 1)
		{
		}

		virtual ~AObject()
		{}

		// This function load or build the object
		virtual bool initialize()
		{
			return (true);
		}

		//This function handles the object behaviour
		virtual void update(gdl::Clock const &clock, gdl::Input &input)
		{}

		// Draw the object
		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); // We create an identity matrix

		// We apply rotations on x,y and z axis
		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));
		// We perform the translation
		transform = glm::translate(transform, _position);
		// We scale the object
		transform = glm::scale(transform, _scale);
		return (transform);
	}

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

};
```

Now the cube class:

```
class	Cube : public AObject
{
private:
	// The texture used by the cube
	gdl::Texture		_texture;
	// The cube geometry
	gdl::Geometry			_geometry;
	// The movment speed
	float				_speed;

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

	virtual bool  initialize()
	{
		_speed = 10.0f;

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


```
                return (false);
}

// We set the color of the first side
_geometry.setColor(glm::vec4(1, 0, 0, 1));
// tout les pushVertex qui suivent seront de cette couleur

// We push the vertices
_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));

// Then the UVs
_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));

                // We don't forget to build our geometry to send the data to the graphics card
                _geometry.build();
                return (true);
        }


        // Move the cube using  the keyboard
        virtual void update(gdl::Clock const &clock, gdl::Input &input)
        {
                //  We multiply by the time since the last frame so that the speed does not depend on the
computer power.
                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;
                // We bind the texture to say we want to use it
                _texture.bind();
                // Then we draw our cube
                _geometry.draw(shader, getTransformation(), GL_QUADS);
        }

};
```

A small game engine :

```
#pragma once

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

#include "AObject.hpp"

/*
We create our class that inherits from gdl::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;

		// Activates the OpenGL depth test for the pixels that the eye does not see does not
appear
		glEnable(GL_DEPTH_TEST);

		// We create a shader
		if (!_shader.load("./Shaders/basic.fp", GL_FRAGMENT_SHADER)
			|| !_shader.load("./Shaders/basic.vp", GL_VERTEX_SHADER)
			|| !_shader.build())
			return false;

		// We set the camera
		glm::mat4  projection;
		glm::mat4  transformation;

		projection = glm::perspective(60.0f, 800.0f / 600.0f, 0.1f, 100.0f);
		transformation = glm::lookAt(glm::vec3(0, 10, -30), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));

		// We have the bind the shader before calling the setUniform method
		_shader.bind();
		_shader.setUniform("view", transformation);
		_shader.setUniform("projection", projection);

		// We create a cube that we had to the objects list
		AObject				*cube = new Cube();

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

		return true;
	}

	bool GameEngine::update()
	{
		// If the escape key is pressed or if the window has been closed we stop the program
		if (_input.getKey(SDLK_ESCAPE) || _input.getInput(SDL_QUIT))
			return false;
		// Update inputs an clock
		_context.updateClock(_clock);
		_context.updateInputs(_input);
		// We update the objects
		for (size_t i = 0; i < _objects.size(); ++i)
			_objects[i]->update(_clock, _input);
		return true;
	}
```

```
void GameEngine::draw()
{
	// Clear the screen
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	// We bind the shader to be able to draw the geometry
	// We can only have one shader at a time
	_shader.bind();

	// We draw the objects
	for (size_t i = 0; i < _objects.size(); ++i)
		_objects[i]->draw(_shader, _clock);

	// We update the screen
	_context.flush();
}

GameEngine::~GameEngine()
{
	// Destroy the objects before closing
	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;

};
```

Time to test.

```
#include <cstdlib>

#include "GameEngine.hpp"

int			main()
{
	// create the engine
	GameEngine engine;

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

Congrats you have now a moving colored cube.

This short tutorial is now over we hope that you will have fun making this project. Do not hesitate to post on the GameDevLab forum http://gamedevlab.epitech.eu/forum/ if you have troubles with the library.

Good Luck ☺