Cours OpenGL - GDL-GL

Avril 2012

Jérémie Thomas jeremie.thomas@epitech.eu
Jordan Galby jordan.galby@epitech.eu
Thomas Tu thomas.tu@epitech.eu
GameDevLab gamelab@epitech.eu

Table des matières

OpenGL

OpenGL, pour Open Graphics Library, est un API (Application Programming Interface) graphique bas niveau vous permettant d'afficher des éléments visuels. Il est supporté par de multiples systèmes et hardwares (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.

> *The OpenGL Architecture Review Board (ARB), was an independent consortium formed in 1992, that governed the future of OpenGL, proposing and approving changes to the specification, new releases, and conformance testing. In Sept 2006, the ARB became the OpenGL Working Group under the Khronos Group consortium for open standard APIs.*

`La version d'OpenGL utilisée est la 1.3. OpenGL a subi un bon nombres de changements à partir de la 3.0, ce cours n'est donc pas valide à partir de la version 3.0.`

I La bibliothèque GDL-GL

Les classes contenues dans cette bibliothèque permettent de gérer les différentes parties nécessaires à un jeu vidéo : l'affichage graphique, la gestion du temps, la gestion des entrées utilisateur et la boucle de jeu.

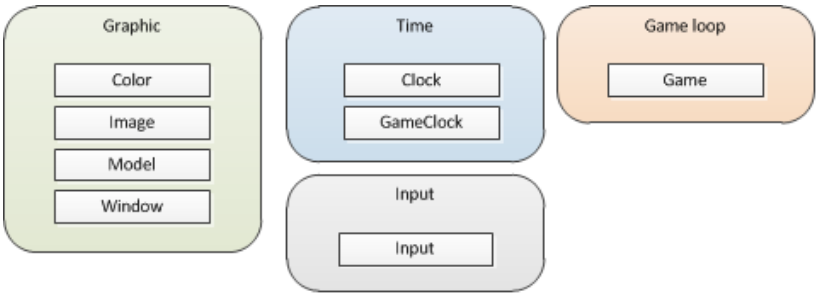


FIGURE 1 – Organisation des classes de la bibliothèque.

Indices Une **documentation** doxygen est disponible dans le dossier "public" du compte "**gamelab**".

I.1 Utilisation de la bibliothèque

I.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**.

```
1 #include <Clock.hpp>
2 #include <Color.hpp>
3 #include <Game.hpp>
4 #include <GameClock.hpp>
5 #include <Image.hpp>
6 #include <Input.hpp>
7 #include <Model.hpp>
8 #include <Window.hpp>
```

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

```
1 #include <GL/gl.h>
2 #include <GL/glu.h>
```

I.1.2 Compilation

Lors de la compilation, il vous faut inclure trois bibliothèques :

```
1 > g++ -o ... -I/afs/epitech.net/users/labos/gamelab/public/include -L/afs/epitech.net/users/labos/gamelab/
    public/lib -Wl,--rpath=/afs/epitech.net/users/labos/gamelab/public/lib -lgdl_gl -lGL -lGLU
```

-lGL : la bibliothèque OpenGL.
-lGLU : OpenGL Utility library.
-lgdl_gl : la Lbibliothèque graphique GDL-GL.

Le flag **-Wl** permet de transmettre des options au linker utilisé par **g++** (en l'occurrence **ld**) sous la forme "-Wl,option1,option2...". Dans notre cas, nous utilisons l'option –**rpath=filename** de **ld** pour lui indiquer que des symboles se trouvent dans le dossier indiqué étant donné que nous utilisons une bibliothèque partagée et non statique. Cependant, ces symboles ne seront toujours pas définis dans le binaire (ça serait fait au runtime).

Pensez à remplacer le path des flags de compilations par les paths adéquats suite au désarchivage de l'archive fournie.

I.2 Premiers pas

I.2.1 La boucle de jeu

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

- mise à jour des inputs
- mise a jour de l'horloge principale
- mise a jour du comportement des objets
- affichage des élements graphiques

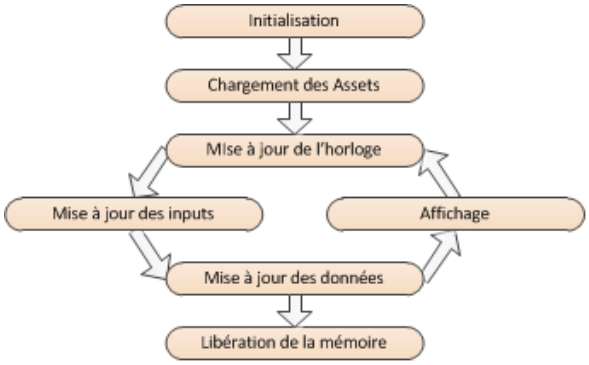


FIGURE 2 – Les différentes étapes d'un jeu vidéo.

La boucle de jeu, fournie par la classe **Game** vous permet de construire un jeu vidéo simplement puisqu'elle automatise deux tâches principales :

- mise à jour des inputs
- mise a jour de l'horloge principale

Vous avez le contrôle des tâches suivantes :

- mise a jour du comportement
- affichage des élements graphiques

L'application doit hériter de la classe **Game** puis certaines méthodes pures telles que celle-ci, qui permettent de faire des initialisations et des chargements de données, doivent être implémentées :

```
1 void Game::initialize(void);
```

La mise à jour des comportements correspond aux mises à jour des objets selon le temps et les inputs. Elle doit se faire à travers la méthode pure suivante :

```
1 void Game::update(void);
```

L'affichage des objets doit se faire avec la méthode pure suivante :

```
1 void Game::draw(void);
```

La méthode suivante permet de libérer la mémoire lorsque la fenêtre est fermée :

```
1 void Game::unload(void);
```

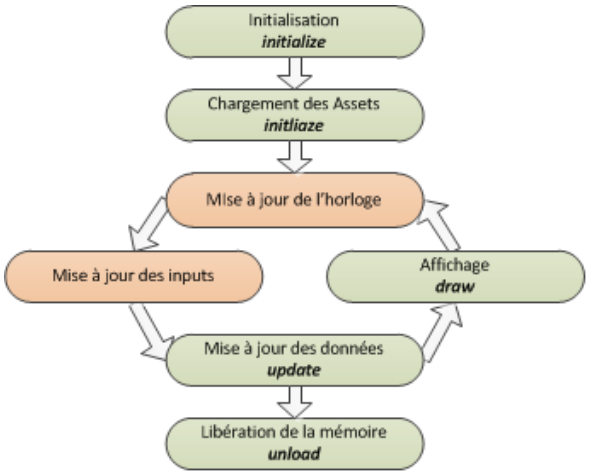


FIGURE 3 – En orange, les étapes gérées par la classe Game, en vert les étapes gérées par l'utilisateur.

I.2.2 Démarrer un contexte OpenGL

Un contexte OpenGL est NÉCESSAIRE pour pouvoir faire toute manipulation graphique (des initialisations y sont faites en interne à OpenGL). La bibliothèque s'occupe de créer ce contexte mais seulement lorsque la fenêtre doit être créée. La classe **Game** possède plusieurs **attributs protégés** qui seront utiles :

```
protected:
    ////////////////////////////////////////////////////////////////////////////
    /// La fenetre unique
    ////////////////////////////////////////////////////////////////////////////
    Window window_;

    ////////////////////////////////////////////////////////////////////////////
    /// L'horloge principale du jeu
    ////////////////////////////////////////////////////////////////////////////
    GameClock gameClock_;

    ////////////////////////////////////////////////////////////////////////////
    /// Le gestionnaire d'inputs
    ////////////////////////////////////////////////////////////////////////////
    Input  input_;
```

La méthode suivante de la classe **Window**, qui est un singleton, permet de créer le contexte OpenGL puis de créer la fenêtre.

```
void  Window::create(void);
```

Après la création du contexte, des initialisations OpenGL peuvent être faites.

Indices La convention utilisée dans la bibliothèque veut que les attributs protégés ou privés soient suffixés par un underscore.

I.2.3 Exemple

Pour appuyer le cours, un mini projet sera construit petit à petit. Ce projet consistera à créer une scène où des objets pourront se mouvoir. Les objets seront, entre autres un cube, une pyramide, un quadrilatère, un triangle et un modèle 3D. Ces objets doivent hériter de la classe abstraite **AObject** et seront ajoutés dans une liste :

```
1  ////////////////////////////////////////////////////////////////////////////////
2  /// Declaration de la classe abstraite AObject
3  ////////////////////////////////////////////////////////////////////////////////
4
5  class   AObject
6  {
7  public:
8      AObject(void)
9          : position_(0.0f, 0.0f, 0.0f), rotation_(0.0f, 0.0f, 0.0f)
10     {
11     }
12     virtual void  initialize(void) = 0;
13     virtual void  update(gdl::GameClock const &, gdl::Input &) = 0;
14     virtual void  draw(void) = 0;
15 protected:
16     Vector3f  position_;
17     Vector3f  rotation_;
18 };
```

```
1  ////////////////////////////////////////////////////////////////////////////////
2  /// Declaration de la classe MyGame
3  ////////////////////////////////////////////////////////////////////////////////
4
5  #include <cstdlib>
6  #include <list>
7
8  #include "Game.hpp"
9
10 class MyGame : public gdl::Game
11 {
12 public:
13     void   initialize(void)
14     void   update(void)
15     void   draw(void)
16     void   unload(void);
17
18 private:
19   Camera    camera_;
20   std::list<AObject*>  objects_;
21 };
```

Indices

Vous aurez besoin d'assets dans la suite de ce mini projet. Un asset désigne une ressource d'un jeu vidéo (texture, son, modèle 3D etc.). Vous les trouverez sur l'AFS, dans le dossier **public** de l'utilisateur **gamelab**.

```
1  ////////////////////////////////////////////////////////////////////////////////
2  /// Definition de la classe MyGame
3  ////////////////////////////////////////////////////////////////////////////////
4
5  void  MyGame::initialize(void)
6  {
7      ////////////////////////////////////////////////////////////////////////////////
8      /// Creation du contexte OpenGL et de la fenetre
9      ////////////////////////////////////////////////////////////////////////////////
10     window_.create();
11
12     camera_.initialize();
13
14     ////////////////////////////////////////////////////////////////////////////////
15     /// Ajout des objets heritant de AObject dans la liste
16     ////////////////////////////////////////////////////////////////////////////////
17
18     ////////////////////////////////////////////////////////////////////////////////
19     /// Appel de la methode initialize des objets
20     ////////////////////////////////////////////////////////////////////////////////
21     std::list<AObject*>::iterator itb = this->objects_.begin();
22     for (; itb != this->objects_.end(); ++itb)
23         (*itb)->initialize();
24 }
25
26 void  MyGame::update(void)
27 {
28     std::list<AObject*>::iterator itb = this->objects_.begin();
29
30     ////////////////////////////////////////////////////////////////////////////////
31     /// Appel de la methode update des objets
32     ////////////////////////////////////////////////////////////////////////////////
33     for (; itb != this->objects_.end(); ++itb)
34         (*itb)->update(gameClock_, input_);
35
36     camera_.update(gameClock_, input_);
37 }
38
39 void  MyGame::draw(void)
40 {
41     ////////////////////////////////////////////////////////////////////////////////
42     /// Vidage des buffers
43     ////////////////////////////////////////////////////////////////////////////////
44     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
45     glClearColor(0.74f, 0.84f, 95.0f, 1.0f);
46     glClearDepth(1.0f);
47
48     std::list<AObject*>::iterator itb = this->objects_.begin();
49
50     ////////////////////////////////////////////////////////////////////////////////
51     /// Appel de la methode draw des objets
52     ////////////////////////////////////////////////////////////////////////////////
53     for (; itb != this->objects_.end(); ++itb)
54         (*itb)->draw();
55 }
56
57 void  MyGame::unload(void)
58 {
59     ////////////////////////////////////////////////////////////////////////////////
60     /// Liberation memoire des ressources allouees.
61     ////////////////////////////////////////////////////////////////////////////////
62 }
```

Chacun des objets de la liste d'objets subira en boucle un appel de la méthode **update** puis de la méthode **draw**. Lorsque la fenêtre est fermée, la méthode **unload** est appelée.

```
1  ////////////////////////////////////////////////////////////////////////////////
2  /// Declaration de la fonction main
3  ////////////////////////////////////////////////////////////////////////////////
4
5  #include <cstdlib>
6
7  int main(void)
8  {
9      MyGame game;
10
11     game.run();
12     return EXIT_SUCCESS;
13 }
```

I.2.4 Classes utilitaires

```
////////////////////////////////////////////////////////////////////////////////
/// Declaration de la structure Vector3f
////////////////////////////////////////////////////////////////////////////////

struct  Vector3f
{
  float  x;
  float  y;
  float  z;

  Vector3f(void);
  Vector3f(float x, float y, float z);
};
```

```
////////////////////////////////////////////////////////////////////////////////
/// Definition de la structure Vector3f
////////////////////////////////////////////////////////////////////////////////

Vector3f::Vector3f(void)
    : x(0.0f), y(0.0f), z(0.0f)
{
}

Vector3f::Vector3f(float x, float y, float z)
    : x(x), y(y), z(z)
{
}
```

II 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.

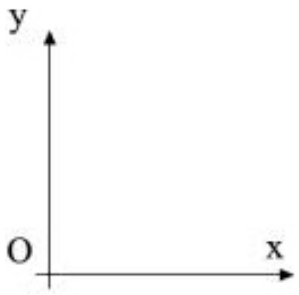


FIGURE 4 – Repère orthonormé xy

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.

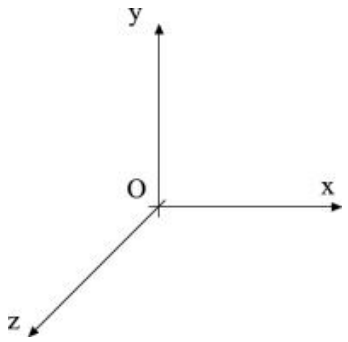


FIGURE 5 – Repère orthonormé xyz

II.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 $\vec{a}(x, y)$ 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 $\vec{a}(x, y, z)$ dont les composantes ont pour valeur :

$$x = 6$$
$$y = 5$$
$$z = 0$$

II.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 êtres 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.

II.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.

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

II.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 OpenGL lorsque l'on en a besoin.

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

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

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

III 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.

III.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 certains 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.

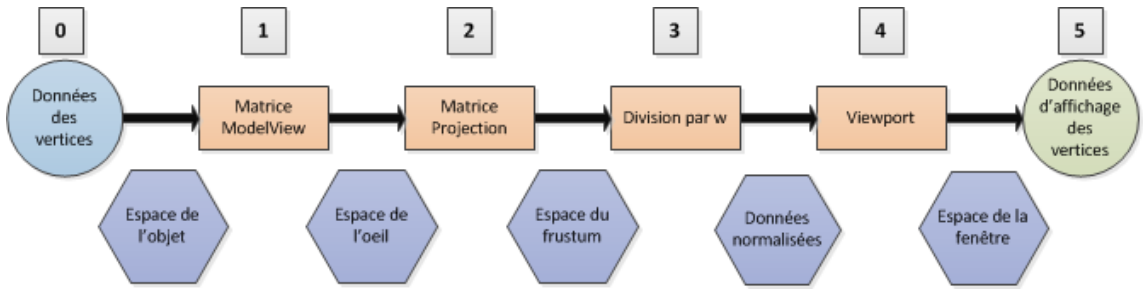


FIGURE 6 – Les transformations d'OpenGL

ModelView et **Projection** représentent des piles de matrices qui doivent être utilisées afin de pouvoir positionner la caméra et nos objets. La matrice qui se trouve au-dessus de la pile est considérée comme étant la **"matrice courante"**.

0. En entrée, les coordonnées de votre vertex.

1. Une transformation avec la matrice courante de ModelView et les coordonnées d'un vertex positionnera ce dernier dans le même espace que la caméra. ModelView est intéressante puisqu'elle permet de déplacer la caméra.

2. La pyramide de projection (frustum en anglais) partant de la caméra est ensuite définie avec la pile de matrices Projection. C'est à travers une transformation avec la matrice courante de cette pile qu'est déterminé si un vertex est dans notre champ de vision. Un changement d'espace vers l'espace du frustum est donc effectué.

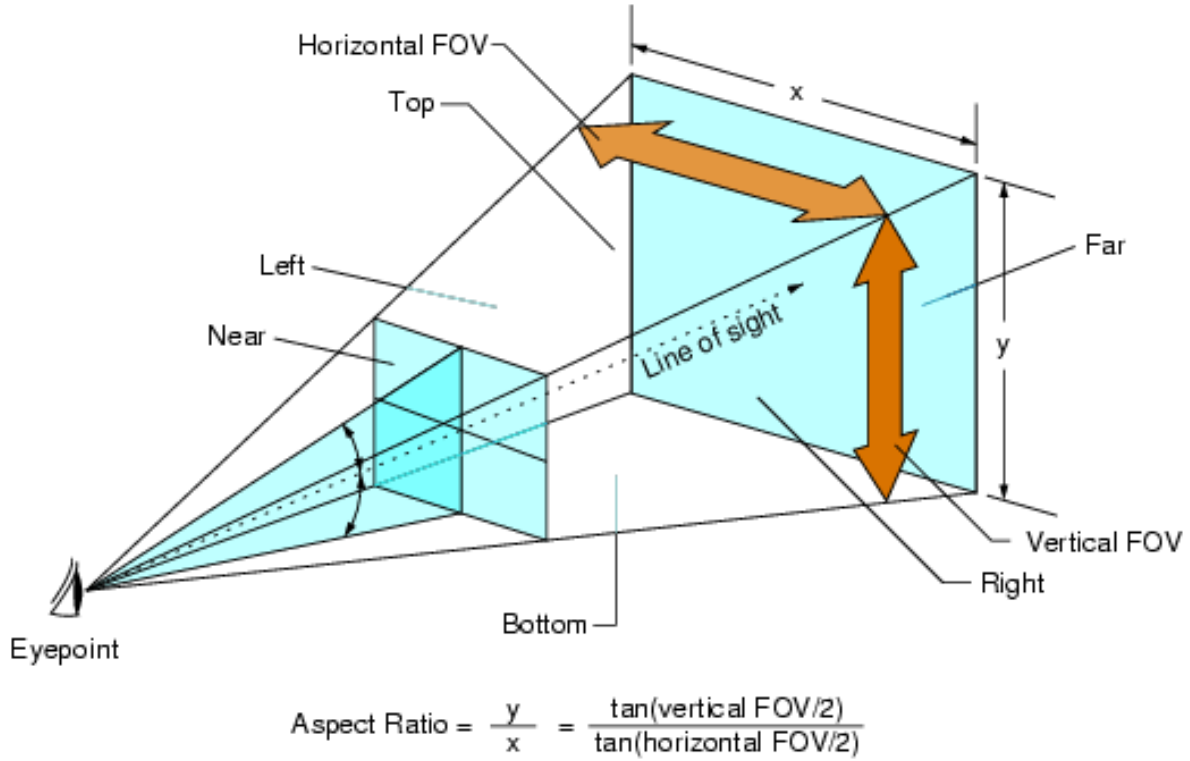


FIGURE 7 – Représentation schématique du frustum. Source : http ://techpubs.sgi.com.

Les vertices qui seront dans le frustum entre les plans Near et Far seront visibles. La distance de ces plans est configurable, ainsi que l'angle de vue (thêta). **Top** indique le plafond de la caméra.

3. Normalise les coordonnées sur chaque axe du repère. Cela est utile avec une projection perspective seulement.

Voir **IV.5** pour la définition de projection perspective et projection orthogonale.

4. La transformation avec le Viewport permet d'effectuer un nouveau changement d'espace afin que les objets soient dans l'espace du cadre.

5. En sortie, les coordonnées de votre vertex sur la fenêtre de rendu.

Indices `La matrice Projection permet de configurer la caméra. La matrice ModelView permet le positionnement de la caméra et l'affichage des modèles.`

III.2 Les options

OpenGL possède de nombreux paramètres configurables une fois que le contexte a été créé. Ces paramètres permettent de manipuler l'environnement 3D et la manière dont est calculé l'affichage final.

```
1 void  glEnable(GLenum cap);
2 void  glDisable(GLenum cap);
```

Certaines options sont activées automatiquement dans la bibliothèque.

III.3 Les tests de profondeur

Les tests de profondeur permettent de savoir quel pixel doit être affiché lorsque plusieurs objets sont les uns devant les autres. OpenGL gère cela en interne mais il doit être activé. Son activation passe par l'utilisation de la fonction **glEnable** avec pour paramètre **GL_DEPTH_TEST**. Son fonctionnement peut être configuré par la fonction suivante :

```
1 void  glDepthFunc(GLenum func);
```

La documentation d'OpenGL vous indique plusieurs paramètres possibles, parmi eux : **GL_LEQUAL** et **GL_GEQUAL**.

Par exemple si GL_LEQUAL est utilisé, le test est positif si la profondeur testée est inférieure ou égale à celle qui se trouve dans le depth buffer.

III.4 Vertex

Un vertex (vertices au pluriel) représente un point (un sommet) d'un polygone. Pour dessiner des figures, il faudra dessiner des vertices.

III.5 Les matrices

Une fonction permet de choisir la pile de matrices (ModelView, Projection) que nous voulons manipuler :

```
1 void   glMatrixMode(GLenum mode);
```

Deux fonctions permettent d'ajouter ou de retirer un élément d'une pile, respectivement :

```
1 void   glPushMatrix(void);
2 void   glPopMatrix(void);
```

Comme cela avait été dit dans la partie précédente, la matrice qui sera en haut de la pile est considérée comme étant la **"matrice courante"**. Cette matrice est utilisée pour déterminer les transformations que doit subir l'objet qui sera dessiné.

Il vous sera utile de convertir la matrice courante en **matrice d'identité** afin d'avoir une matrice de base n'ayant subit aucune transformation. Elle doit être faite avec la fonction suivante :

```
1 void   glLoadIdentity(void);
```

III.5.1 Les transformations

Les fonctions suivantes appliquent une transformation sur la matrice courante.

```
1 void   glTranslatef(GLfloat x, GLfloat y, GLfloat z);
2 void   glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
```

glRotatef prend en paramètre l'angle de rotation et un vecteur qui servira d'axe de rotation.

Voir **IV.6.1** pour l'exemple.

Indices Il existe également des fonctions équivalentes pouvant manipuler des coordonnées de type **double**.

III.6 La caméra

gluLookAt permet de définir la position absolue de la caméra et son orientation. La définition doit être faite sur la pile de matrice ModelView.

```
1 void   gluLookAt(GLdouble eyeX , GLdouble eyeY, GLdouble eyeZ,
2                  GLdouble centerX, GLdouble centerY, GLdouble centerZ,
3                  GLdouble upX, GLdouble upY, GLdouble upZ);
```

Les trois premières coordonnées correspondent à la position de la caméra. Les trois suivantes correspondent à la direction de la caméra. Elle regardera vers le point indiqué par les coordonnées. Les trois dernières coordonnées permettent de donner la direction du haut de la caméra.

La position du repère en trois dimensions change donc selon l'orientation de la caméra.

```
1 void   gluLookAt(0.0f, 0.0f, 0.0f,
2                  0.0f, 0.0f, -1.0f,
3                  0.0f, 1.0f, 0.0f);
```

Ces cordonnées correspondent à la position par défaut de la caméra (la caméra regarde vers le point -1.0 sur z, avec le haut de la caméra vers le point 1.0 sur y) :

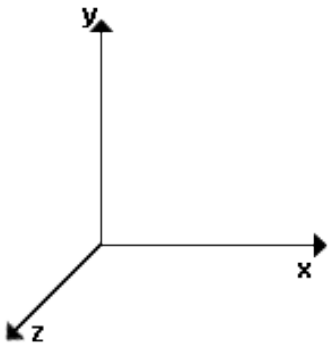


FIGURE 8 – Position par défaut de la caméra.

```
1 void   gluLookAt(0.0f, 0.0f, 0.0f,
2                  5.0f, 0.0f, 0.0f,
3                  0.0f, 1.0f, 0.0f);
```

Le point vers lequel la caméra regardait a été modifiée. Le point se trouve désormais en 5.0 sur l'axe x.

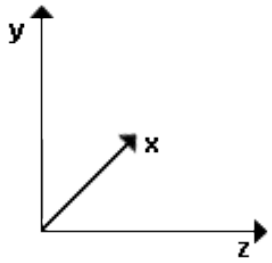


FIGURE 9 – Nouvelle position de la caméra.

III.6.1 La perspective

gluPerspective permet de définir une pyramide de projection. La définition doit être faite sur la pile de matrice Projection. Préférez cette projection pour un projet en trois dimensions.

```
1 void   gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
```

fovy
L'angle de vision.
aspect
Les proportions de l'image, sous la forme de (largeur/longueur).
zNear
La distance entre la position de la vue et le plan Near du frustum.
zFar
La distance entre la position de la vue et le plan Far du frustum.

III.6.2 La projection orthogonale

glOrtho permet de définir une projection parallèle. La définition doit être faite sur la pile de matrice Projection. Préférez cette projection pour un projet en deux dimensions.

```
1 void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble nearVal, GLdouble
      farVal);
```

left
La distance entre la camera est le plan de clip gauche.
right
La distance entre la camera est le plan de clip droit.
bottom
La distance entre la camera est le plan de clip bas.
top
La distance entre la camera est le plan de clip haut.
nearVal
La distance entre la camera est est le plan Near.
farVal
La distance entre la camera est est le plan Far.

III.6.3 Exemple

Nous ajoutons une caméra à notre scène.

```
1  ////////////////////////////////////////////////////////////////////////////
2  /// Declaration de la classe Camera
3  ////////////////////////////////////////////////////////////////////////////
4
5  # include "GameClock.hpp"
6  # include "Input.hpp"
7  # include "Vector3.hpp"
8
9  class Camera
10 {
11 public:
12     Camera(void);
13
14     void  initialize(void);
15     void  update(gdl::GameClock const &, gdl::Input &);
16 private:
17     Vector3f position_;
18     Vector3f rotation_;
19 };
```

```
1  ////////////////////////////////////////////////////////////////////////////
2  /// Definition de la classe Camera
3  ////////////////////////////////////////////////////////////////////////////
4
5  Camera::Camera(void)
6      : position_(0.0f, 0.0f, 900.0f), rotation_(0.0f, 0.0f, 0.0f)
7  {
8  }
9
10 void  Camera::initialize(void)
11 {
12     ////////////////////////////////////////////////////////////////////////////
13     /// Configuration du frustum de la camera
14     ////////////////////////////////////////////////////////////////////////////
15     glMatrixMode(GL_PROJECTION);
16     glLoadIdentity();
17     gluPerspective(70.0f, 800.0f/600.0f, 1.0f, 10000.0f);
18     gluLookAt(position_.x, position_.y, position_.z,
19             0.0f, 0.0f, -1.0f,
20             0.0f, 1.0f, 0.0f);
21
22     ////////////////////////////////////////////////////////////////////////////
23     /// Positionnement de la camera
24     ////////////////////////////////////////////////////////////////////////////
25     glMatrixMode(GL_MODELVIEW);
26     glLoadIdentity();
27
28     ////////////////////////////////////////////////////////////////////////////
29     /// Activation des tests de profondeur
30     ////////////////////////////////////////////////////////////////////////////
31     glEnable(GL_DEPTH_TEST);
32     glDepthFunc(GL_LEQUAL);
33 }
34
35 void  Camera::update(gdl::GameClock const & gameClock, gdl::Input & input)
36 {
37 }
```

III.7 Primitives

En programmation 3D, nous utilisons les vertices pour dessiner des primitives qui formeront un polygone. Pour délimiter un groupe de vertices, les deux fonctions suivantes doivent être appelées :

```
1 void   glBegin(GLenum mode);
2 void   glEnd(void);
```

La première fonction prend une valeur d'enum en argument qui va déterminer la manière dont seront traités les vertices. En effet, OpenGL peut utiliser le groupe de vertices de plusieurs manières différentes. Parmi elles, nous pouvons retenir :

- GL_TRIANGLES
- GL_QUADS
- GL_POLYGON

GL_TRIANGLES permet de dessiner un triangle à partir de trois vertices. OpenGL prendra chaque triplet de vertices dans le groupe de vertices pour y dessiner des triangles.

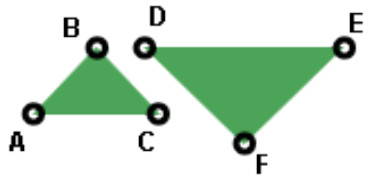


FIGURE 10 – Deux triplets de vertices

GL_QUADS nécessite un quadruplet de vertices pour dessiner un quadrilatère.

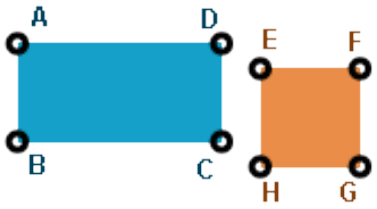


FIGURE 11 – Deux quadruplets de vertices

Quant à **GL_POLYGON**, une figure convexe est dessinée à partir de tous les vertices présents.

L'ordre des vertices est important. Il faut qu'ils soient créés de manière à ce qu'on puisse rejoindre chaque vertex au crayon sans le relever. Regardez bien sur les figures 10 et 11.

La fonction suivante vous permettra de dessiner une vertex, elle prend en paramètre des positions relatives. En effet, les transformations nous permettront de les déplacer dans l'espace.

```
1 void glVertex3f(GLfloat x, GLfloat y, GLfloat z);
```

III.7.1 Exemple

Pour le projet exemple, nous allons créer un namespace **Primitive** contenant deux classes pour l'objet **Triangle** et **Rectangle**.

```
1  ////////////////////////////////////////////////////////////////////////////////
2  /// Declaration des classes Triangle et Rectangle
3  ////////////////////////////////////////////////////////////////////////////////
4
5  namespace  Primitive
6  {
7      class Triangle : public AObject
8      {
9          void initialize(void);
10         void update(gdl::GameClock const &, gdl::Input &);
11         void draw(void);
12     };
13
14     class Rectangle : public AObject
15     {
16         void initialize(void);
17         void update(gdl::GameClock const &, gdl::Input &);
18         void draw(void);
19     };
20 }
```

```
1  ////////////////////////////////////////////////////////////////////////////////
2  /// Definition de la classe Triangle
3  ////////////////////////////////////////////////////////////////////////////////
4
5  namespace  Primitive
6  {
7      void Triangle::initialize(void)
8      {
9      }
10
11     void Triangle::update(gdl::GameClock const & gameClock, gdl::Input & input)
12     {
13     }
14
15     void Triangle::draw(void)
16     {
17         ////////////////////////////////////////////////////////////////////////////////
18         /// Ouverture d'un contexte de rendu
19         ////////////////////////////////////////////////////////////////////////////////
20         glBegin(GL_TRIANGLES);
21
22         ////////////////////////////////////////////////////////////////////////////////
23         /// Configuration de la couleur des vertices
24         ////////////////////////////////////////////////////////////////////////////////
25         glColor3f(1.0f, 0.50f, 0.75f);
26
27         ////////////////////////////////////////////////////////////////////////////////
28         /// Dessin des vertices
29         ////////////////////////////////////////////////////////////////////////////////
30         /// Vertex haut
31         glVertex3f(0.0f, 150.0f, 0.0f);
32         /// Vertex bas gauche
33         glVertex3f(-150.0f, 0.0f, 0.0f);
34         /// Vertex bas droit
35         glVertex3f(150.0f, 00.0f, 0.0f);
36
37         ////////////////////////////////////////////////////////////////////////////////
38         /// Fermeture du contexte de rendu
39         ////////////////////////////////////////////////////////////////////////////////
40         glEnd();
41     }
42 }
```

```cpp
////////////////////////////////////////////////////////////////////////////////
/// Definition de la classe Rectangle
////////////////////////////////////////////////////////////////////////////////

namespace Primitive
{
    void Rectangle::initialize(void)
    {
    }

    void Rectangle::update(gdl::GameClock const & gameClock, gdl::Input & input)
    {
    }

    void Rectangle::draw(void)
    {
        ////////////////////////////////////////////////////////////////////////////////
        /// Ouverture d'un contexte de rendu
        ////////////////////////////////////////////////////////////////////////////////
        glBegin(GL_QUADS);

        ////////////////////////////////////////////////////////////////////////////////
        /// Configuration de la couleur des vertices
        ////////////////////////////////////////////////////////////////////////////////
        glColor3f(1.0f, 0.50f, 0.75f);

        ////////////////////////////////////////////////////////////////////////////////
        /// Dessin des vertices
        ////////////////////////////////////////////////////////////////////////////////
        /// Vertex superieur gauche
        glVertex3f(-150.0f, 100.0f, 0.0f);
        /// Vertex inferieur gauche
        glVertex3f(-150.0f, -100.0f, 0.0f);
        /// Vertex inferieur droit
        glVertex3f(150.0f, -100.0f, 0.0f);
        /// Vertex superieur droit
        glVertex3f(150.0f, 100.0f, 0.0f);

        ////////////////////////////////////////////////////////////////////////////////
        /// Fermeture du contexte de rendu
        ////////////////////////////////////////////////////////////////////////////////
        glEnd();
    }
}
```

III.8 Des objets plus complexes

Des objets plus complexes peuvent être construits à partir de primitives. Nous allons donc ajouter un objet Pyramide à trois faces qui sera constitué de quatre triangles et un objet Cube à six faces constitué de six rectangles.

```
1  ////////////////////////////////////////////////////////////////////////////////
2  /// Declaration des objets Pyramide et Cube
3  ////////////////////////////////////////////////////////////////////////////////
4
5  namespace  Object
6  {
7      class Cube : public AObject
8      {
9          void initialize(void);
10         void update(gdl::GameClock const &, gdl::Input &);
11         void draw(void);
12     };
13
14     class Pyramide : public AObject
15     {
16         void initialize(void);
17         void update(gdl::GameClock const &, gdl::Input &);
18         void draw(void);
19     };
20 }
```

```
1  ////////////////////////////////////////////////////////////////////////////////
2  /// Definition de la classe Cube
3  ////////////////////////////////////////////////////////////////////////////////
4
5  namespace Object
6  {
7      void Cube::initialize(void)
8      {
9      }
10
11     void Cube::update(gdl::GameClock const & gameClock, gdl::Input & input)
12     {
13     }
14
15     void Cube::draw(void)
16     {
17         ////////////////////////////////////////////////////////////////////////////////
18         /// Ouverture d'un contexte de rendu
19         ////////////////////////////////////////////////////////////////////////////////
20         glBegin(GL_QUADS);
21
22         ////////////////////////////////////////////////////////////////////////////////
23         /// Configuration de la couleur des vertices
24         ////////////////////////////////////////////////////////////////////////////////
25         glColor3f(1.0f, 0.50f, 0.75f);
26
27         ////////////////////////////////////////////////////////////////////////////////
28         /// Dessin des vertices
29         ////////////////////////////////////////////////////////////////////////////////
30
31         /// Vertex superieur gauche
32         glVertex3f(-150.0f, 150.0f, 150.0f);
33         /// Vertex inferieur gauche
34         glVertex3f(-150.0f, -150.0f, 150.0f);
35         /// Vertex inferieur droit
36         glVertex3f(150.0f, -150.0f, 150.0f);
37         /// Vertex superieur droit
38          glVertex3f(150.0f, 150.0f, 150.0f);
39
40         /// Vertex superieur gauche
41         glVertex3f(-150.0f, 150.0f, -150.0f);
42         /// Vertex inferieur gauche
43         glVertex3f(-150.0f, -150.0f, -150.0f);
44         /// Vertex inferieur droit
45         glVertex3f(-150.0f, -150.0f, 150.0f);
46         /// Vertex superieur droit
47         glVertex3f(-150.0f, 150.0f, 150.0f);
48
49         /// Vertex superieur gauche
50         glVertex3f(150.0f, 150.0f, -150.0f);
51         /// Vertex inferieur gauche
52         glVertex3f(150.0f, -150.0f, -150.0f);
53         /// Vertex inferieur droit
54         glVertex3f(-150.0f, -150.0f, -150.0f);
55         /// Vertex superieur droit
56         glVertex3f(-150.0f, 150.0f, -150.0f);
57
58         /// Vertex superieur gauche
59         glVertex3f(150.0f, 150.0f, 150.0f);
60         /// Vertex inferieur gauche
61         glVertex3f(150.0f, -150.0f, 150.0f);
62         /// Vertex inferieur droit
63         glVertex3f(150.0f, -150.0f, -150.0f);
64         /// Vertex superieur droit
65         glVertex3f(150.0f, 150.0f, -150.0f);
66
67         ////////////////////////////////////////////////////////////////////////////////
68         /// Fermeture du contexte de rendu
69         ////////////////////////////////////////////////////////////////////////////////
70         glEnd();
71     }
72 }
```

```cpp
////////////////////////////////////////////////////////////////////////////////
/// Definition de la classe Pyramide
////////////////////////////////////////////////////////////////////////////////

namespace Object
{
    void Pyramide::initialize(void)
    {
    }

    void Pyramide::update(gdl::GameClock const & gameClock, gdl::Input & input)
    {
        this->rotation_.y = ((int)rotation_.y + 1) % 360;
    }

    void Pyramide::draw(void)
    {
        glPushMatrix();
        glLoadIdentity();

        glTranslatef(0.0f, 0.0f, -900.0f);
        ////////////////////////////////////////////////////////////////////////////////
        /// Rotation autour de l'axe Y
        ////////////////////////////////////////////////////////////////////////////////
        glRotatef(this->rotation_.y, 0.0f, 1.0f, 0.0f);
        ////////////////////////////////////////////////////////////////////////////////
        /// Ouverture d'un contexte de rendu
        ////////////////////////////////////////////////////////////////////////////////
        glBegin(GL_TRIANGLES);

        ////////////////////////////////////////////////////////////////////////////////
        /// Dessin des vertices
        ////////////////////////////////////////////////////////////////////////////////
        glColor3f(1.0f, 1.0f, 1.0f);
        /// Vertex haut : B
        glVertex3f(0.0f, -150.0f, -150.0f);
        /// Vertex inferieur gauche : C
        glVertex3f(-150.0f, -150.0f, 150.0f);
        /// Vertex inferieur droit : D
        glVertex3f(150.0f, -150.0f, 150.0f);

        glColor3f(1.0f, 0.0f, 0.0f);
        /// Vertex haut : A
        glVertex3f(0.0f, 150.0f, 0.0f);
        /// Vertex inferieur gauche : C
        glVertex3f(-150.0f, -150.0f, 150.0f);
        /// Vertex inferieur droit : D
        glVertex3f(150.0f, -150.0f, 150.0f);

        glColor3f(0.0f, 1.0f, 0.0f);
        /// Vertex haut : A
        glVertex3f(0.0f, 150.0f, 0.0f);
        /// Vertex inferieur gauche : B
        glVertex3f(0.0f, -150.0f, -150.0f);
        /// Vertex inferieur droit : C
        glVertex3f(-150.0f, -150.0f, 150.0f);

        glColor3f(0.0f, 0.0f, 1.0f);
        /// Vertex haut : A
        glVertex3f(0.0f, 150.0f, 0.0f);
        /// Vertex inferieur gauche : D
        glVertex3f(150.0f, -150.0f, 150.0f);
        /// Vertex inferieur droit : B
        glVertex3f(0.0f, -150.0f, -150.0f);
        ////////////////////////////////////////////////////////////////////////////////
        /// Fermeture du contexte de rendu
        ////////////////////////////////////////////////////////////////////////////////
        glEnd();

        glPopMatrix();
    }
}
```

Indices

Les GPU manipulent bien mieux les triangles que les autres primitives. En effet, il est possible de constituer n'importe quelle forme en trois dimensions uniquement à partir de triangles. Cela est plus difficile avec les autres primitives.

D'ailleurs, comment dessine-t-on une sphère ?

III.9 Les textures

L'utilisation des textures diffère de ce que vous avez pu voir auparavant et suit les étapes suivante :

- chargement de la texture
- liaison de la texture
- affichage de la texture

La seconde étape est importante. Elle va lier la texture au système de rendu. Il n'est possible de lier qu'une texture à la fois. L'application d'une texture se fait toujours pendant le rendu d'une primitive. La fonction suivante doit être utilisée :

```
1 void   glTexCoord2f(GLfloat s, GLfloat t);
```

Les paramètres correspondent aux coordonnées 2D de la texture qui sont normalisées. Elles varient entre 0.0 et 1.0. Chaque appel à cette fonction est suivi d'un appel au rendu du vertex correspondant.

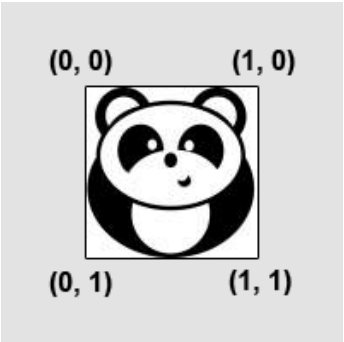
Le coin supérieur gauche a pour coordonnées :

$$x = 0$$
$$y = 0$$

Par conséquent, le coin inférieur droit a pour coordonnées :

$$x = 1$$
$$y = 1$$

Par exemple :



Voir **VI.3** pour l'exemple.

IV La bibliothèque GDL-GL en détails

IV.1 Le temps

La classe **Game** possède en privé plusieurs instances de classes dont une qui permet de manipuler l'horloge principale de type **GameClock**.
Cette horloge principale est lancée dès que la boucle de jeu démarre. La méthode suivante permet de connaitre le temps passé depuis le dernier tour dans la boucle de jeu :

```
1 float  getElapsedTime(void) const;
```

Il est également possible de connaitre le temps total écoulé depuis le lancement du jeu à travers la méthode suivante :

```
1 float  getTotalGameTime(void) const;
```

Une classe **Clock** est disponible afin que vous ayez vos propres horloges. Cette classe possède les deux méthodes présentées ci-dessus et d'autres qui vous permettent de la lancer, de l'arrêter ou de la remettre à zéro.

```
1  #include <iostream>
2  #include <cstdlib>
3
4  #include "Clock.hpp"
5
6  int main(void)
7  {
8      gdl::Clock myClock;
9
10     myClock.play();
11
12     myClock.update();
13     std::cout << "Total game time elapsed : " << myClock.getElapsedTime() << " seconde(s)" << std::endl;
14     std::cout << "Total time elapsed : " << myClock.getTotalElapsedTime() << " seconde(s)" << std::endl;
15
16     myClock.update();
17     std::cout << "Total game time elapsed : " << myClock.getElapsedTime() << " seconde(s)" << std::endl;
18     std::cout << "Total time elapsed : " << myClock.getTotalElapsedTime() << " seconde(s)" << std::endl;
19
20     myClock.pause();
21
22     myClock.update();
23     std::cout << "Total game time elapsed : " << myClock.getElapsedTime() << " seconde(s)" << std::endl;
24     std::cout << "Total time elapsed : " << myClock.getTotalElapsedTime() << " seconde(s)" << std::endl;
25
26     return EXIT_SUCCESS;
27 }
```

```
1 > ./a.out
2 Total game time elapsed : 1.00001e-06 seconde(s)
3 Total time elapsed : 2.00002e-06 seconde(s)
4 Total game time elapsed : 1.00001e-06 seconde(s)
5 Total time elapsed : 0.000187 seconde(s)
6 Total game time elapsed : 8.7e-05 seconde(s)
7 Total time elapsed : 0.000187 seconde(s)
```

IV.2 Les événements

Les événements claviers sont gérés par la classe **Input**. Tout comme l'horloge principale de l'application, cette classe possède une instance unique dans la classe **Game**.

Une méthode de cette classe permet de savoir si la touche envoyée en paramètre est appuyée.

```
1 bool  Input::isKeyDown(gdl::Key key);
```

IV.2.1 Exemple

Nous allons modifier la classe **Pyramide** afin que les rotations ne se fassent que lorsque nous appuyons sur les flèches du clavier. Nous n'aurons qu'à modifier sa méthode **update**.

```
1  namespace Object
2  {
3      void Pyramide::update(gdl::GameClock const & gameClock, gdl::Input & input)
4      {
5          ////////////////////////////////////////////////////////////////////////////
6          /// Si la fleche droite est appuyee, la pyramide tourne a droite
7          ////////////////////////////////////////////////////////////////////////////
8          if (input.isKeyDown(gdl::Keys::Right) == true)
9              this->rotation_.y = ((int)rotation_.y + 1) % 360;
10         ////////////////////////////////////////////////////////////////////////////
11         /// Si la fleche gauche est appuyee, la pyramide tourne a gauche
12         ////////////////////////////////////////////////////////////////////////////
13         if (input.isKeyDown(gdl::Keys::Left) == true)
14             this->rotation_.y = ((int)rotation_.y - 1) % 360;
15     }
16 }
```

Indices Référez-vous au fichier en-tête de la classe Input afin de connaitre les membres de l'enum **gdl::Key**

.

IV.3 Les textures

Une classe vous permet de charger une texture puis de la lier. Le chargement d'une image se fait via la méthode statique suivante :

```
1 static Image  Image::load(std::string const & filename);
```

Vous pouvez connaitre la taille de la texture via les méthodes suivantes :

```
1 unsigned int  Image::getWidth(void) const;
2 unsigned int  Image::getHeight(void) const;
```

La liaison se fait à travers la méthode suivante :

```
1 void   Image::bind(void) const;
```

L'application de la texture se fait via OpenGL avec la méthode suivante :

```
1 void   glTexCoord2f(GLfloat s, GLfloat t);
```

IV.3.1 Exemple

Nous allons modifier la méthode **draw** de la classe Rectangle afin qu'il affiche une texture.

```
1  namespace Primitive
2  {
3      void Rectangle::initialize(void)
4      {
5          this->texture_ = gdl::Image::load("assets/textures/panda.png");
6      }
7      void Rectangle::draw(void)
8      {
9          ////////////////////////////////////////////////////////////////////////////
10         /// Dessin des vertices
11         ////////////////////////////////////////////////////////////////////////////
12         /// Vertex superieur gauche
13         glTexCoord2f(0.0f, 0.0f);
14         glVertex3f(-150.0f, 100.0f, 0.0f);
15         /// Vertex inferieur gauche
16         glTexCoord2f(0.0f, 1.0f);
17         glVertex3f(-150.0f, -100.0f, 0.0f);
18         /// Vertex inferieur droit
19         glTexCoord2f(1.0f, 1.0f);
20         glVertex3f(150.0f, -100.0f, 0.0f);
21         /// Vertex superieur droit
22         glTexCoord2f(1.0f, 0.0f);
23         glVertex3f(150.0f, 100.0f, 0.0f);
24     }
25 }
```

IV.4 Les modèles 3D

Les modèles 3D sont gérés par la classe **Model**. Tout comme les images, les modèles sont chargés avec une méthode statique :

```
1 static Model  load(std::string const & filename);
```

La classe **Model** ne gère que le format **FBX** d'Autodesk. Si vous comptez utiliser vos propres modèles, vérifiez que les textures sont bien incluses dans le fichier .fbx.

L'affichage d'un modèle est plus simple que l'affichage d'une image. Il suffit de faire un appel à la méthode suivante :

```
1 void   draw(void);
```

IV.4.1 Les animations

Une méthode permet de jouer une animation donnée :

```
1 bool   play(std::string const & animationName, char state=Anim::RUN);
```

La mise à jour de l'animation, c'est-à-dire la sélection de la frame de l'animation en fonction du temps se fait avec la méthode suivante. L'animation est automatiquement mise à jour à chaque appel de **update**.

```
1 void   update(GameClock const & gameTime);
```

IV.4.2 Exemple

```
////////////////////////////////////////////////////////////////////////////////
/// Declaration de la classe Bomberman
////////////////////////////////////////////////////////////////////////////////

namespace Model
{
    class Bomberman : public AObject
    {
    public:
        Bomberman(void);
        ~Bomberman(void);

        void initialize(void);
        void update(gdl::GameClock const &, gdl::Input &)
        void draw(void)
    private:
        gdl::Model model_;
    };
}
```

```
////////////////////////////////////////////////////////////////////////////////
/// Definition de la classe Bomberman
////////////////////////////////////////////////////////////////////////////////

namespace Model
{
    void Bomberman::initialize(void)
    {
        ////////////////////////////////////////////////////////////////////////////////
        /// Charge le modele
        ////////////////////////////////////////////////////////////////////////////////
        this->model_ = gdl::Model::load("assets/models/marvin.fbx");
    }

    void Bomberman::~Bomberman(void)
    {
        ////////////////////////////////////////////////////////////////////////////////
        /// Destruction des ressources.
        ////////////////////////////////////////////////////////////////////////////////
        delete this->model_;
    }

    void Bomberman::update(gdl::GameClock const & gameClock, gdl::Input & input)
    {
        ////////////////////////////////////////////////////////////////////////////////
        /// Si une animation est en cours, nous la mettons a jour
        ////////////////////////////////////////////////////////////////////////////////
        this->model_.update(gameClock);

        ////////////////////////////////////////////////////////////////////////////////
        /// Joue l'animation portant le nom "Take 001" quand on appuie sur la touche "p"
        ////////////////////////////////////////////////////////////////////////////////
        if (input.isKeyDown(gdl::Keys::P) == true)
            this->model_.play("Take 001");
    }

    void Bomberman::draw(void)
    {
        ////////////////////////////////////////////////////////////////////////////////
        /// Affichage du modele
        ////////////////////////////////////////////////////////////////////////////////
        this->model_.draw();
    }
}
```

A savoir

- La bibliothèque graphique n'est pas thread-safe.
- La documentation doxygen de la bibliothèque GDL-GL est disponible dans le dossier **public** du compte unix **gamelab**.
- La documentation OpenGL peut aussi vous aider :
  http://www.opengl.org/sdk/docs/man/
- Toutes vos questions doivent être posées sur le forum du GameDevLab :
  http://gamedevlab.epitech.eu/forum/
- Les modèles 3D ont été fournis par Xavier Baures, un grand merci à lui !