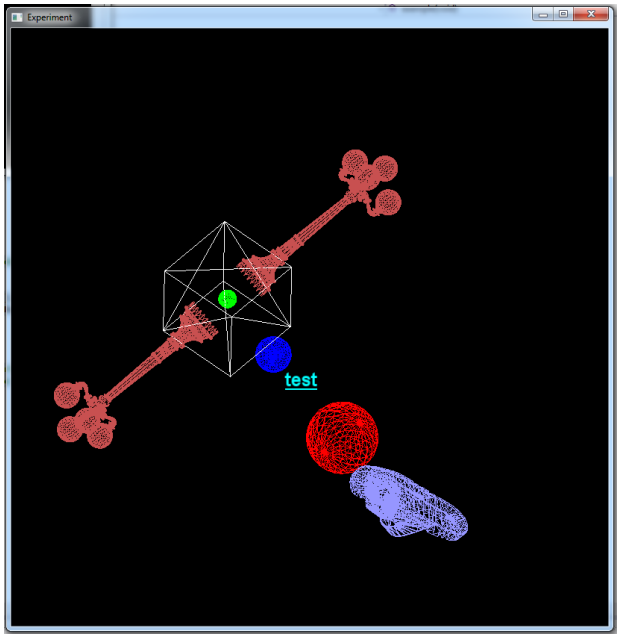
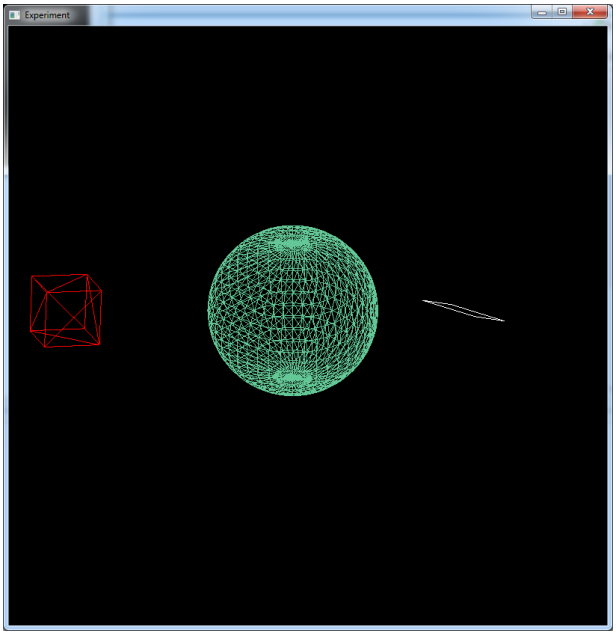
L’3 - 2013/2014
Fabien GENET
Souhaib DADI
Emmanuel KOUADIO

Rapport technique

Programmation d’un moteur graphique 3D rudimentaire





Introduction

Nous avons nommé le moteur 3D, 3D Engin et nous avons fait en sorte qu'il puisse compiler sur Linux/Windows/Mac.
De plus chaque classe appartenant au projet est contenue dans un namespace (namespace e3d). Cela permet de ne pas confondre les objets du moteur 3D avec des objets portant le même nom contenu dans d'autre librairie.

Compilation

Le projet est compilable à la fois sur Windows/Linux en mode 32 et 64Bit. Pour ce faire vous devez crée un projet et ajouter les classes de notre projet ainsi que la librairie SFML (2.1) ou OpenCV (2.4.8) ou les deux à votre projet.
Bien sûr en ajoutant la version 32 bit ou 64 de la librairie choisi.

Si vous compilez le projet sous Windows il vous faudra ajouter les fichiers .dll à l'exécutable.

Le fichier de configuration Config.h vous permet de choisir de compiler le projet avec SFML ou OpenCV ou les deux.

Attention : Il vous faudra renseigner le mode de rendu au Render avec une des constantes suivantes (cf constructeur Render & exemple) :

- e3d_SFMLRender (Pour SFML)
- e3d_OpenCvRender (Pour Opencv)

Lien vers SFML :
http://www.sfml-dev.org/download-fr.php

Lien vers OpenCV :
http://opencv.org/downloads.html

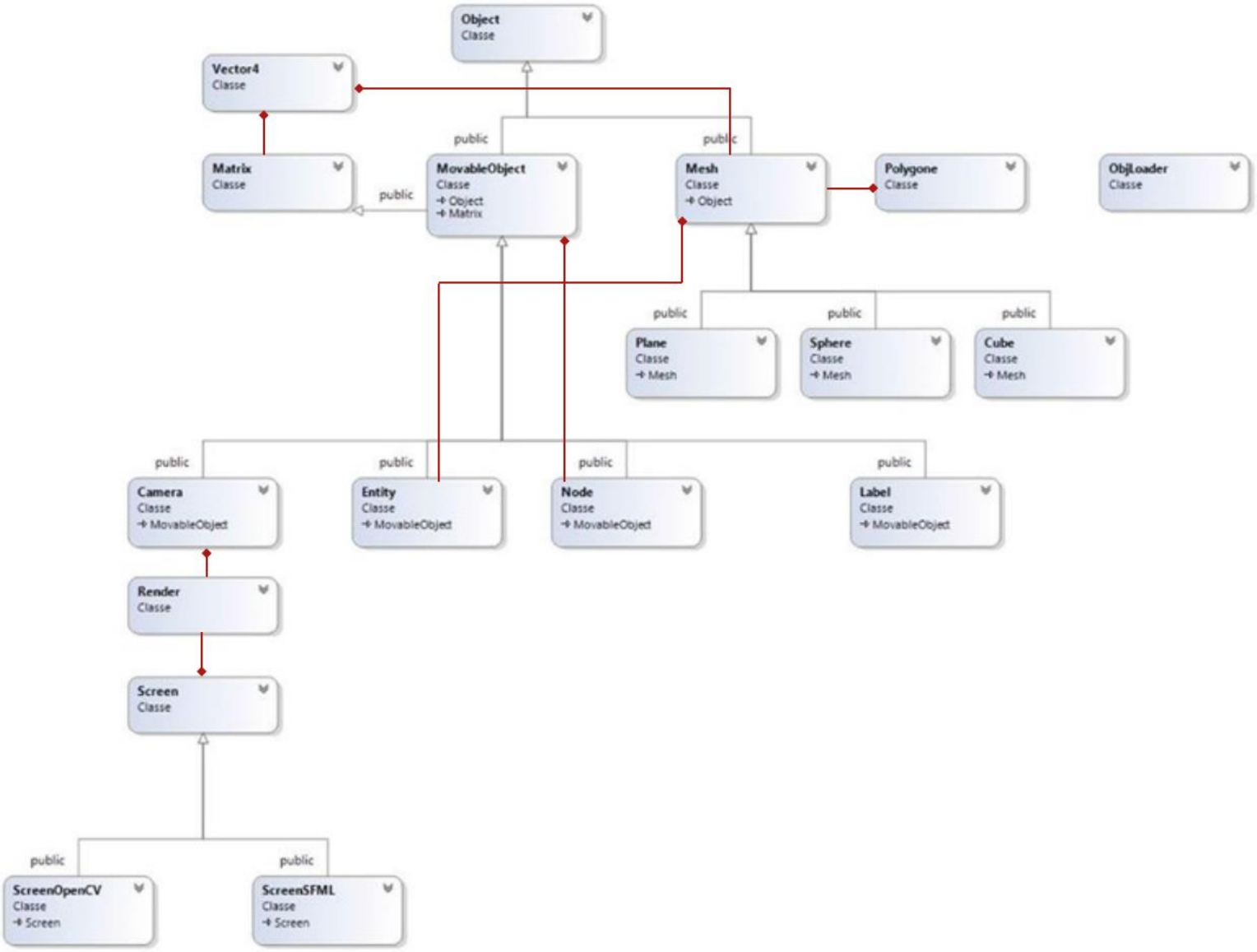
.lib pour SFML :
sfml-graphics.lib
sfml-window.lib
sfml-system.lib

.dll pour SFML :
sfml-window-2.dll
sfml-system-2.dll
sfml-graphics-2.dll

.lib pour OpenCV :
opencv_core247.lib
opencv_imgproc247.lib
opencv_highgui247.lib
opencv_ml247.lib
opencv_video247.lib
opencv_features2d247.lib
opencv_calib3d247.lib
opencv_objdetect247.lib
opencv_contrib247.lib
opencv_legacy247.lib
opencv_flann247.lib

.dll OpenCV :
opencv_highgui247.dll
opencv_core247.dll

Diagramme de classe du projet



En rouge : Agrégation
En gris : Héritage

Exemple simple d'utilisation

```
#include "Mesh.h"
#include "Cube.h"
#include "Sphere.h"
#include "Entity.h"
#include "Node.h"
#include "Camera.h"
#include "Render.h"
using namespace e3d;
void exemple(void)
{
	Sphere m_s("S1",Vector4(0,0,0,0),15,10); //Creation d'une sphere
	Entity en1("Sphere1",&m_s); //Creation d'une entiy à avec la smhere
	en1.translate(50,0,0); //translation sur X
	en1.setColor(100,200,150); // modification de la couleur

	Cube m_c("C1",Vector4(0,0,0,0),10,10,10); //Creation d'un cube 10*10*10
	Entity en2("Cube1",&m_c); //Creation d'une entity avec notre cube
	en2.setColor(255,0,0); //modification de la couleur
	en2.translate(10,0,0); //translation sur X

	Plane m_plan("Plane", Vector4(0,0,0,0),10,20); //creation d'un plan
	Entity en3("Plane1",&m_plan); //Creation d'une entit plan

	Repere m_repere("repere",Vector4(0,0,0,0),2); //creation d'un repere
	Entity rep("Plane1",&m_repere); // creation d'une entity repere
	rep.setColor(0,0,255); //changement de la couleur

	Node n("rootNode");   //creation d'un rootNode
	n.attachMovableObject(&en1); //On attachent l'entity 1
	n.attachMovableObject(&en2); //On attachent l'entity 2

	Node n2(&n,"subnode"); //creation d'un sous noeud
	n2.attachMovableObject(&en3); //on attache l'entity 3
	n2.attachMovableObject(&rep); //on attache notre repaire
	n2.translate(80,0,0); //on translate notre node sur X

	Vector4 camPos(0,-50,0,0); //position de la cam
	double fov = 10; //Reglage de la camera
	double neaR = 0.5, faR = 15;
	Camera cam("Mycam",camPos,fov,neaR,faR); //initialisation de la camera
	Render r(&cam,&n,"Experiment",e3d_SFMLRender); //initialisation du render

	while(1)
	{
		r.clean(); //On efface l'image
		r.start(); //On fais le rendu à partir du RootNode
		r.update(); //On met à jour l'image
		en1.rotateX(PI/100.0); //Rotation des objets
		en2.rotateX(PI/100.0); //rotation de l'entity 2
		en2.rotateY(PI/100.0);
		n2.rotateZ(PI/100.0); //rotation du sous noeud n2
		n2.rotateY(PI/100.0);
		n2.rotateX(PI/100.0);
	}

}
```

ICI C'EST AU DEVELOPPER DE GERER LA BOUCLE ET LES DEPLACEMENTS A APPLIQUER SUR LES OBJETS.

Classe Vector

La classe Vector représente un vecteur en mémoire. Elle implémente donc les calculs vectoriels de "base" Nous avons choisi de mettre cette classe en "friend" avec la classe Matix pour permettre une utilisation plus facile. D'un point de vue 3d, un vector peut être considéré comme un point.

```
friend class Matrix; //On met la classe Matrix en friend
```

Attribut :

`double _v[4]` : un tableau de 4 doubles permet de définir notre vecteur

Constructeurs :

La classe possède 2 constructeurs :

Le premier crée un vecteur et prend en paramètre les coefficients du vecteur

```
Vector4(double x, double y, double z, double w);
```

Le deuxième crée le vecteur et l'initialise à 0.

```
Vector4(double s=0.0);
```

Méthodes :

Ces méthodes vont permettre de faire les calculs de bases :

`Vector4 add(const Vector4& x)` : addition de 2 vecteurs

`Vector4 sous(const Vector4& x)` : soustraction de 2 vecteurs

`Vector4 mult(const double& x)` : multiplication de 2 vecteurs

`double prodScal(const Vector4& x)` : calcule le produit scalaire de 2 vecteurs

`bool isNull(const Vector4& x)` : vérifie si le vecteur est nulle

Ces méthodes vont permettre de faire les surcharges d'opérateurs :

`Vector4 operator-(const Vector4& x)` : surcharge de soustraction

`Vector4 operator+(const Vector4& x)` : surcharge addition

`Vector4& operator=(const Vector4& x)` : surcharge affection

`Vector4 operator*(const double& x)` : surcharge de multiplication

Cette méthode amie de la classe Matrix va permettre de surcharger les opérateurs de flux de sorties :

```
friend ostream& operator<<(ostream& is, const Vector4& x)
```

Cette méthode affiche un vecteur :

```
Void printf();
```

Getters et Setters :

Modifie les valeurs du vecteur

```
void setValue(double x, double y, double z, double w)
```

Modifie la valeur du vecteur à la case at.

```
void setValueAt(int at, double x)
```

Retourne la valeur contenue à la case pos

```
void setValueAt(int at, double x);
```

Ces setters permettent de modifier une composante à la fois du vecteur.

```
double setX(double x)  { return _v[0] = x; }
double setY(double y)  { return _v[1] = y; }
double setZ(double z)  { return _v[2] = z; }
```

```
            double setW(double w)  { return _v[3] = w;}
```

Ces getters permettent d'acquérir une composante à la fois du vecteur.

```
            double getX() const { return _v[0]; }
            double getY() const { return _v[1]; }
            double getZ() const { return _v[2]; }
            double getW() const { return _v[3]; }
```

Classe Matrix

La classe matrix représente une matrice en mémoire, Elle implémente donc les calculs matriciel de "base" Nous avons choisi de mettre cette classe en "friend" avec la classe Vector4 pour permettre une utilisation plus facile des valeurs et ainsi éviter de passer par des getter et setter pour modifier les valeurs de notre matrice.

Attribut :

`Vector4 _m[4]` : permet de définir la matrice grâce à un tableau de 4 vecteurs

Constructeurs :

La classe possède 3 constructeurs :

`Matrix(void)` :Le premier crée une matrice et l'initialise par la matrice identité 4*4.

`Matrix(const Vector4& v1, const Vector4& v2, const Vector4& v3, const Vector4& v4)` : Le Deuxième crée la matrice et prend en paramètre les coefficients du la matrice via 4 vecteurs.

```
Matrix(double x11, double x12, double x13, double w1,
       double x21, double x22, double x23, double w2,
       double x31, double x32, double x34, double w3,
       double x41, double x42, double x43, double w4);
```

Le troisième crée la matrice et prend en paramètre les valeurs de chaque cooeficient.

Méthodes :

Ces méthodes vont permettre de faire les calculs de bases :

`Matrix add(const Matrix& m2) const ;` : Adition de 2 matrices

`Matrix sous(const Matrix& m2) const ;`: Soustraction de 2 matrices

`Matrix mult(const Matrix& m2) const ;`: multiplication de 2 matrices

`Vector4 mult(const Vector4& v2) const ;` : multiplication d'une matrice par un vector4

`void printf()` : Cette méthode va permettre l'affichage des matrices.

`friend ostream& operator<<(ostream& os, const Matrix& m)` : Cette méthode est une méthode amie ,il s'agit de la surcharge de l'opérateur de flux de sortie :

Ces méthodes sont les surcharges d'opérateurs

`Matrix operator*(const double a) const`: multiplication d'une matrice par un reel

`Matrix operator*(const Matrix& m) const` : multiplication d'une matrice par une matrice

`Vector4 operator*(const Vector4& v) const` : multiplication d'une matrice par un vector

`Matrix operator-(const Matrix& m) const` : soustraction d'une matrice par une matrice

`Matrix operator+(const Matrix& m) const` : addition d'une matrice par une matrice

`Matrix& operator=(const Matrix& x)` : affection d'une matrice par une matrice

`bool operator==( const Matrix& m2 ) const` : égalité de deux matrices

Ces méthodes nous permettent d'inverser une matrice :

```
double determinant () const : calcul du déterminant
Matrix transpose () const : donne la transposée de la matrice
Matrix inverse () const : donne l'inverse de la matrice
Matrix comatrice() const : renvoi la co-matrice de la matrice courante.
virtual void makeTransformation(const Matrix& m);
```
Cette méthode applique la transformation passée à l'objet.

Getters et Setters :

Renvoi la valeur à la case i,j de la matrice :
```
double getValueAt(int i, int j) const;
```

Met la valeur x dans la case i,j de la matrice :
```
double setValueAt(int i, int j, double val)
```

Renvoi la matrice identité d'une matrice :
```
Matrix getMatIdentity()
```

Renvoi la matrice de translation associé au valeurs x,y,z
```
Matrix getTranslation(double x, double y, double z)
```

Surcharge de la méthode getTranslation à partir d'un Vector4.
```
Matrix getTranslation(Vector4& v);
```

Renvoi la matrice permettant un changement d'échelle.*
```
Matrix getScale(double x, double y, double z);
Matrix getScale(Vector4& v);
```

Renvoi les matrices associés aux rotations sur X, sur Y et sur Z suivant un angle donne.
```
Matrix getRotate(double angle, double x, double y, double z);
Matrix getRotateX(double angle);
Matrix getRotateY(double angle);
Matrix getRotateZ(double angle);
```

Affectation d'une matrice à l'objet courant.
```
void setMat(const Matrix& x);
```

Renvoi la matrice courante
```
const Matrix& getMat()
```

Renvoi le vecteur de ligne i de la matrice.
```
Vector4 getVector4At(int i) { return _m[i]; }
```

Classe Object

Cette classe est la classe mère de tous les types d'objet, c'est à dire les Mesh et MovableObject. Elle a comme attribut une chaine de caractère (name) qui correspond au nom des objets créer dans le programme. Cette classe contient également une variable statique pool qui permet de stocker un pointeur vers tous les objets crée dans un programme. Cette variable est statique, son contenue est donc commun a toute les instances de la classe objet et de ses classes filles.

La variable pool est de type map. Une variable map est une variable de type clé valeur (tableau associatif) ou la clé est le nom de l'objet et la valeur est un pointeur vers un objet (pointeur). On a fait le choix de ce type de variable au lieu d'un tableau car elle n'impose pas de taille limite contrairement à un tableau. De plus les méthodes de recherches sont déjà implémenté et fonctionnel.

Constructeur:

Cette classe contient un constructeur:

```
Object(const string& name) :
```

Ce constructeur créer une instance qui a comme nom la chaine de caractère passé en paramètre et stock cette instance dans la variable static pool.

Methodes:

Cette méthode permet d'afficher le nombre d'objets qu'il y a dans le pool et d'afficher chaque objet du pool.

```
static void printPool() :
```

Cette méthode permet d'ajouter l'objet passé en paramètre à la variable pool.

```
static void registerObject(Object* x, const string& name) :
```

Cette méthode permet de remplacer l'objet dont le nom est passé en paramètre.

```
static void setObject(Object *x, const string& name) :
```

Permet de vider la variable pool.

```
static void cleanPool() :
```

Retourne un pointeur de l'objet dont le nom est passé en paramètre.

```
static Object* find(const string& name) :
```

Retourne le nombre d'objet qu'il y a dans la variable pool

```
static size_t count() :.
```

Supprime l'objet dont le nom est passé en paramètre de la variable pool.

```
void unregisterObject(const string& name) :
```

Retourne le nom de l'objet.

```
string getName():
```

Modifie le nom de l'objet.

```
void setName(string name) :
```

Classe Mesh

La classe Mesh est la classe mère des classes Cuboide et sphère. Elle hérite de la classe Objet. Elle est donc caractérisé par un nom et chaque instance crée est donc référencé dans la variable pool.

La classe Mesh est le modèle de tout « volume » crée. Dans le moteur 3D un volume (un cube, une sphère..) est représenté par un ensemble de points placée de manière précise dans l'espace. Un volume, ou Mesh, est aussi caractérisé par ses faces. Une face est un ensemble de 3, 4, ou 5 points reliée pour former un triangle, un carré, ou un pentagone. Les polygones permettront la mise en place de la surface du volume.

Attributs:

Cet attribut est un tableau qui stocke l'ensemble des points du volume.

```
vector<Vector4> _vertex :
```

Cet attribut est une tableau qui stocke l'ensemble des instances polygone (les faces d'un polygones).

```
vector<Polygone> _polygone :
```

Définit si un Mesh est visible ou non.

```
int _visibility :
```

Constructeur:

```
Mesh(const string& name);
```
: ce constructeur prend en paramètre une chaine de caractère (name). Ce constructeur appel le constructeur Objet en lui passant la chaine de caractère name et remplie _vertex avec n vector de paramètre (0,0,0,0).

Methode:

```
void print()
```
Cette méthode permet d'afficher les coordonnée de tout les points stocké dans l'attribut _vertex.

```
friend ostream& operator<<(ostream& os, Mesh& m)
```
: surcharge de l'operateur << qui permet d'afficher tous les points d'un volume (mesh)

Accesseurs et mutateurs :

Retourne le nombre de points qu'il y a dans la mesh.

```
size_t getSize():
```

Retourne une référence du i éme points du mesh.

```
Vector4& getVertexAt(int i) :
```

Cette méthode prend en paramètre un indice i et une instance vector4 (un point). Elle place le vecteur x à la case d'indice i du vector _vertex.

```
void setVertexAt(int i, const Vector4& x) :
```

Cette méthode prend en paramètre référence un polygone et l'ajoute au « tableau » _vertex.

```
void addPolygone(const Polygone& p) :
```

cette méthode retourne ne nombre de polygones de la mesh.

```
size_t getNbPolygone():
```

Cette méthode prend en paramètre un entier i et retourne i éme polygone du tableau _polygone.

```
Polygone& getPolygoneAt(int i)
```

Classe Cube

La classe cube hérite de la classe mesh. Une instance cube est donc caractériser par l'attribut nom, _vertex (dans lequel il y a les points) et _polygone. Géométriquement, un cube est caractérisé par une hauteur, une largeur et une profondeur. Il est constitué de huit points.

Attributs:

double_width : la largeur du cube.
double_height : la hauteur du cube.
double_Lenght : la profondeur du cube.

Constructeur:

Le constructeur de la classe Cube prend en paramètre une chaine de caractère name qui représente le nom du volume. Un point (Vector4) qui représente la position du cube dans l'espèce, et trois variables réels. Le constructeur crée quatre points dans l'espace en fonction de la position du cube et de ses trois caractéristiques : hauteur, largeur, profondeur. Le constructeur ajoute également les polygones qui constituent le cube: chaque face du volume est divisé en deux triangles.

```
Cube(const string& name,Vector4 center, double width, double height, double lenght)
```

Methodes:

void print(): affiche les coordonnées des huit points du volume.
friend ostream& operator<<(ostream&, Cube& c) : surcharge de l'opérateur << qui permet d'afficher les coordonnées des huit points du volume.

Accesseurs et mutateurs :

double getWidth(): retourne la largeur du cube
double getHeight() : retourne la hauteur du cube
double getLenght(): retourne la profondeur du cube

Classe Plane

La classe Plane hérite de la classe mesh. Une instance Plane est donc caractériser par l'attribut nom, _vertex (dans lequel il y a les points) et _polygone (faces). Géométriquement, un plan est caractérisé par une hauteur, une largeur. Il est constitué de quatre points.

Attributs:

double_width : la largeur du plan.
double_height : la hauteur du plan.

Constructeur:

Le constructeur de la classe Plane prend en paramètre une chaine de caractère name qui représente le nom du plan. Un point (Vector4) qui représente la position du cube dans l'espèce, et deux variables réels. Le constructeur cree donc un plan à centré sur la position d'origine.

```
Cube(const string& name,Vector4 center, double width, double height, double lenght)
```

Accesseurs et mutateurs :

double getWidth(): retourne la largeur du cube
double getHeight() : retourne la hauteur du cube

Classe Repere

La classe Repere hérite de la classe mesh. Une instance Repere est donc caractériser par l'attribut nom, _vertex (dans lequel il y a les points) et _polygone (faces). Un repéré permet de representer les axes X, Y et Z dans l'espace.

Attributs:

Double _scale : la taille du repère

Constructeur:

Le constructeur de la classe Repere prend en paramètre une chaine de caractère name qui représente le nom du repère. Un point (Vector4) qui représente la position du cube dans l'espèce, et la taille du repère (réel).

```
Repere(string name, Vector4 center, double scale);
```

Accesseurs et mutateurs :

double getScale(): retourne la taille du repère

Classe Polygone

Dans le moteur3D, tout volume est constitué d'un ensemble de points, mais aussi d'un ensemble de faces. Par exemple dans un cube, on peut voir 6 polygones carrée ou 12 polygones triangle.
Ces faces sont appelé polygone. Un polygone est un ensemble de 3, 4, ou 5 points reliée pour former un triangle, un carre, ou un pentagone. Les polygones permettront la mise en place de la surface du volume.

Attribut :

`vector<unsigned int> _vertexIndice` : une instance polygone ne stocke pas des copie de vector4 ou des pointeurs. Dans le but d'augmenter les performances du moteur3D, nous avons choisie de stocker les indices des Vector4 dans le tableau _vertexe de la mesh.

Constructeur:

`Polygone(int a, int b, int c)` : Prend en paramètre trois entiers qui correspondent aux indices des trois points qui forment le polygone triangle.

`Polygone(int a, int b, int c, int d)` : Prend en paramètre quatre entiers qui correspondent aux indices des quatre points qui forment le polygone carre.

`Polygone(int a, int b, int c, int d, int e)` : Prend en paramètre cinq entiers qui correspondent aux indices des cinq points qui forment le polygone.

`Polygone(int a, int b, int c, int d, int e, int f)` :Prend en paramètre six entiers qui correspondent aux indices six points qui forment le polygone.

`size_t size()` :retourne le nombre de points qu'il y a dans le vector _vertexIndice.

`friend ostream& operator<<(ostream& os , const Polygone& p)`: surcharge de l'operateur << qui permet d'afficher les indice des points de chaque polygone.

Accesseurs et mutateurs :

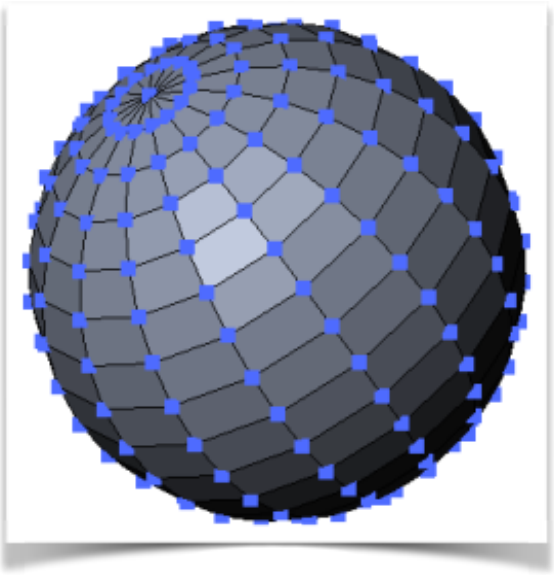
Renvoi l'indice du vertex contenu à la case i.

```
int getPointAt(int i) { return _vertexIndice[i]; }
```

Permet de modifier l'indice d'un point à la case i.

```
void setPointAt(int i, int x) { _vertexIndice[i] = x; }
```

Classe Sphère



Comme la classe cube, la classe sphère hérite de la classe Mesh. Une instance sphère est donc caractérisée par l'attribut nom, _vertex (dans lequel il y a les points) et _polygone.

Constructeur :

Constructeur qui calculs que les points de la sphère sans les relier.

```
Sphere(const string& name,Vector4 center, double radius, double nbLogitude, double nbLatitude) :
```

```
Sphere(const string& name,Vector4 center, double r, int n)) :
```

Le second constructeur de la classe sphère prend en paramètre

- un vector4 qui représente le centre de la sphère. Les autres points de la sphère seront placés en fonction de ce centre.
- un réel r qui correspond au rayon de la sphère.
- un réel n qui correspond au nombre de segment avec lequel la sphère est partagé.

Ainsi on sait que la sphère sera formée avec (4 * n + 4) points placé à équidistance du centre de la sphère.

Chaque points de la sphère de rayon r et de centre l'origine du repère peuvent être paramétrés par :

$$\begin{cases} x &= r\cos\theta\ \cos\phi \\ y &= r\cos\theta\ \sin\phi \\ z &= r\sin\theta \end{cases} \qquad \left(\frac{-\pi}{2} \leq \theta \leq \frac{\pi}{2} \text{ et } -\pi \leq \phi \leq \pi\right)$$

On peut voir θ comme la latitude et ϕ comme la longitude.

Dans le constructeur, θ = ϕ = PI / 2 / (n + 1);

Le code pour générer la sphère est un peu plus long. Distribution des points sur la sphère est la partie la plus simple. Je l'ai trouvé plus difficile de générer les triangles correctement.

Classe MovableObject

Plusieurs constantes permettent de différencier le type d'objet mobile.
Elles sont préfixé par le mot clef e3d_ .

La classe MovableObject est une classe qui permet de modéliser un objet mobile. Elle hérite de la classe Object puisque un objet mobile reste un objet. Elle hérite également de la classe Matrix. Cette matrice associée à chaque objet mobile permet de représenter le repaire local de l'objet.

Attributs :

- `bool _visible` : Permet de définir si un objet est visible ou non.
- `int _type` : Permet de définir la nature de l'objet (avec les constants e3d_)

Attribut hérité :

- `string _name` (Classe Object) : Permet de définir le nom de l'objet
  `Vector4 _m[4]` (Classe Matrix) : Correspond à une matrice, ce qui permet de définir le repaire local de l'objet.

Constructeur :

La classe possède un constructeur :

- `MovableObject(const string& name, int type)`

Attends le nom de l'objet à crée ainsi que son type (constantes e3d_).

Méthodes :

Les méthodes relatives aux transformations sont définit en virtual puisqu'elles peuvent si besoin être redéfini pour un objet dérivé de MovableObject.

Applique la rotation de l'angle donnée sur l'axe spécifié en paramètre.

```
virtual void rotate(double angle, double x, double y, double z);
```

Applique la rotation de l'angle donné suivant l'axe X.

```
virtual void rotateX(double angle);
```

Applique la rotation de l'angle donné suivant l'axe Y.

```
virtual void rotateY(double angle);
```

Applique la rotation de l'angle donné suivant l'axe Z.

```
virtual void rotateZ(double angle);
```

Applique une translation suivant les coordonnés x,y,z.

```
virtual void translate(double x, double y, double z);
```

Permet de redimensionner un objet.

```
virtual void scale(double x, double y, double z);
```

Accesseurs et mutateurs :

Retourne le type de l'objet.

```
int  getType() { return _type; };
```

Permet de definer le type de l'objet.

```
void setType(int type) { _type = type; }
```

Permet de tester si un objet est définit comme visible ou non.

```
bool isVisible()                        { return _visible; }
```

Permet de definer la visibilité d'un objet.

```
void setVisibility(bool status) { _visible = status; }
```

Constantes :

Comme expliqué précédemment, les constants permettent de définir le type d'un Object mobile et donc de tester si notre objet de type Node ou Entity par exemple.

```
const unsigned int e3d_unknownObject=0;
const unsigned int e3d_movableObject=1;
const unsigned int e3d_entity=2;
const unsigned int e3d_rootNode=3;
const unsigned int e3d_node=4;
const unsigned int e3d_camera=5;
const unsigned int e3d_label=6;
```

Classe Node

La classe Node permet de représenter un nœud. Un nœud est en fait un repaire auquel nous pouvons attacher des objets ou d'autre nœud composées eux même de plusieurs objets. Cela permet d'appliquer la même transformation à plusieurs objets (Par exemple déplacer plusieurs objets suivant la mémé trajectoire).

Un nœud principal (rootNode) doit être crée pour chaque scène. C'est à partir de ce nœud que s'effectue le rendu.

Chaque nœud contient donc un nœud parent (le rootNode dans le cas d'un nœud fils direct) avec un ou plusieurs nœud enfant et ainsi que des objets de type MovableObject.

Attributs :

- `Node* _parentNode` : Pointeur vers le nœud parent
- `vector<Node*> _childrenNode` : Tableau de nœuds enfants
- `vector<MovableObject*> _childrenObject` : Tableau d'objets attachés au node.

Attribut hérité :

- `string _name (Classe Object)` : Permet de définir le nom de l'objet
- `Vector4 _m[4] (Classe Matrix)` : Correspond à une matrice, ce qui permet de définir le repaire local du node.
- `bool _visible` : Permet de définir si un objet est visible ou non.
- `int _type` : Permet de définir la nature de l'objet (avec les constants e3d_)

Constructeur :

La classe possède deux constructeurs :

Le premier permet de crée un rootNode, il attend juste le nom du node (string).

```
Node(const string& name)
```

Le deuxième crée un nœud "enfant", il prend donc en paramètre un pointeur sur le nœud parent et son nom (string).

```
Node(Node* _ParentNode, const string& name)
```

Méthodes :

Les méthodes virtuel update et updateTrans permettent de mettre à jour la matrice associée à un node avec une transformation. (une Matrice/Matrix).

```
virtual void update(const Matrix& m); //Pour une rotation
virtual void updateTrans(const Matrix& m); //Pour une translation.
```

Permet d'attacher ou de détacher un nœud à notre objet de type node. Cette methode attends l'adresse d'un Node (Node *).

```
void attachChildNode(Node* childNode);
void detachChildNode(Node* childNode);
```

Nous avons ici redéfinit les méthodes qui permettent de faire des transformations. Elle s'utilise exactement comme pour un objet de type MovableObject.

```
void rotate(double angle, double x, double y, double z) {
        update(this->getRotate(angle,x,y,z)); }
void rotateX(double angle) { update(this->getRotateX(angle)); }
void rotateY(double angle) { update(this->getRotateY(angle)); }
void rotateZ(double angle) { update(this->getRotateZ(angle)); }
void translate(double x, double y, double z) {
      updateTrans(this->getTranslation(x,y,z)); }
void scale(double x, double y, double z)  { update(this->getScale(x,y,z)); }
```

Permet d'attacher un objet à un node.

```
void attachMovableObject(MovableObject* object);
```

Permet de compter le nombre de sous nœud et de sous objet.

```
const size_t countChildNode() const     { return _childrenNode.size(); }
const size_t countMovableObject() const { return _childrenObject.size(); }
```

Cette méthode permet de tester si nous somme en présence d'un nœud root ou non.

```
const bool IsRootNode(void) const { if(_parentNode == NULL){ return 1; }
            return 0; } //Renvoi 1 si c'est un parent 0 sinon
```

Accesseurs et mutateurs :

Renvoi l'adresse nœud parent du node instancié.

```
Node* getParentNode(void) const { return _parentNode; }
```

Permet de modifier le nœud parent d'un node

```
void setParentNode(Node* parentNode)
```

Renvoi le l'adresse d'un nœud à partir de son nom (string).

```
Node* getChildNodeByName(const string& searchName);
```

Renvoi l'adresse d'un sous noeuds à partir de son indice

```
Node* getChildNodeAt(const int i) { return _childrenNode[i] ; }
```

Renvoi l'adresse d'un objet à partir de son indice

```
MovableObject* getMovableObjectAt(const int i) { return _childrenObject[i]; }
```

Classe Camera

La classe camera permet de modéliser une prise de vue dans la scène. C'est-à-dire qu'elle permet de se placer dans la scène et elle effectue une projection des points 3 dimensions sur l'écran 2 dimension (Effet de perspective).

Il est bon de rappeler que la camera est un objet mobile, c'est-à-dire qu'il est possible de lui appliquer une rotation ou une translation.

Attributs :

- `unsigned int _renderMode` : Permet de définir le mode de prise de vue de la camera. Cette fonctionnalité n'est pas implémentée dans cette version.
- `Vector4 _position` : Permet de définir la position de la camera dans la scène.
- `double _far,_near,_fov` : Réglage de la camera.

Near = profondeur de champs,
Far = distance du plan de projection
Fov = angle d'ouverture horizontal de la camera ⇔Zoom.

Attribut hérité :

- `string _name (Classe Object)` : Permet de définir le nom de l'objet
- `Vector4 _m[4] (Classe Matrix)` : Correspond à une matrice, ce qui permet de définir le repaire local de la camera.
- `bool _visible` : Permet de définir si un objet est visible ou non. La camera est ici toujours invisible puisque c'est elle qui film la scène.
- `int _type` : Permet de définir la nature de l'objet (avec les constants e3d_)

Constructeur :

La classe possède un constructeur :

Le constructeur attends le nom de la camera, sa position dans la scène, la distance de son plan de projection, sa profondeur de champs et son angle d'ouverture.

```
Camera(string name, Vector4 position, double fov, double neaR, double faR)
```

Méthodes :

Cette méthode renvoi la matrice de projection associé à notre camera.

```
void getProject(const double  Near, const double Far, const double fov, Matrix&
r);
```

Cette méthode renvoi une matrice en vue perspective en fonction des réglages fournis à la camera.

```
void getPerspective(double angle, double imageAspectRatio, double n, double f,
Matrix &mat);
```

Cette méthode renvoi la matrice associé à une vue de type Frustum.

```
void getFrustum(double left, double right, double bottom, double top, double
Near, double Far, Matrix &mat);
```

Il faut noter que ces méthodes ne sont pas véritablement destiné à être appelé directement par le programmeur (même si elles sont utilisable) mais pas le rendu qui se base sur la camera pour avoir la projection des points.

Pour plus d'information sur le type Frustum :
http://en.wikipedia.org/wiki/Viewing_frustum

Accesseurs et mutateurs :

Les accesseur et mutateurs permette à l'utilisateur d'effectué des réglages sur la camera.

```
/* Profondeur de champs */
void setNear(double neaR) { _near = neaR; }
double getNear() { return _near; }


/* Distance du plan de projection */
void setFar(double faR){ _far = faR; }
double getFar() { return _far; }

/* Angle horizontal de la camera */
void setFov(double fov) { _fov = fov; }
double getFov() { return _fov; }

/*Modification de la position de la camera */
void setPosition(const Vector4& val) { _position = val; }
Vector4 getPosition() const { return _position; }
```

Classe Entity

La classe Entity permet de représenter une entité. Une entité est en fait un objet qui est visible à l'écran, déplaçable. Une entité comporte un nom et un model (un mesh).

Attributs :

```
Vector4 _color; // On utilise les case 0,1,2 pour stocker les valeurs RGB
Mesh *_Mesh; //On associe un Ptr Mesh
```

Constructeur :

La classe Entity possède 3 constructeur.

```
Entity(const string& name, Mesh* MeshPtr):
```

Une entité est définit par un nom et un pointeur vers un Mesh.

Les deux autres constructeurs permettent de définir une Entité avec une couleur, sous forme de Vector4 et de RGB.

```
Entity(const string& name, Mesh* MeshPtr, const Vector4& color);
Entity(const string& name, Mesh* MeshPtr, int red, int green, int blue);
```

Accesseurs et mutateurs :

Modifier la couleur d'une entité.

```
void setColor(Vector4 color) { _color = color; }
void setColor(int red, int green, int blue) {
	_color.setX(red); _color.setY(green); _color.setZ(blue);
}
```

```
Vector4 getColor() { return _color; }
```

Renvoi la couleur d'une entité.

```
Mesh* getMesh() { return _Mesh; }
```

Retourne l'adresse d'un mesh associé à une entité.

```
void attachMesh(Mesh *mesh) { _Mesh = mesh; }
```

Attache un mesh à une entité.

```
double getPointX(int at) { return _Mesh->getVertexAt(at).getX(); }
double getPointY(int at) { return _Mesh->getVertexAt(at).getY(); }
double getPointZ(int at) { return _Mesh->getVertexAt(at).getZ(); }
double getPointW(int at) { return _Mesh->getVertexAt(at).getW(); }
```

Classe Label

La classe Label permet d'afficher un texte à l'écran.

Attributs :

`string _label` : valeur du texte à afficher.
`unsigned int size` : Taille du texte.
`Vector4 _color` : Couleur du texte à afficher.

Constructeur :

Le constructeur attend le nom du label ainsi que le texte à afficher.

```
Label(string name, string value):
```

Les deux autres constructeurs attendent en plus la couleur du label à afficher.

```
Label(string name, string value, Vector4 color);
Label(string name, string value, int red, int green, int blue);
```

Accesseurs et mutateurs :

Modification de la valeur du texte.

```
string getLabel() { return _label; }
void   setLabel(string s) { _label = s; }
```

Modification de la couleur du texte.

```
void   setColor(const Vector4& color) { _color = color; }
void   setColor(int red, int green, int blue) {
       _color.setX(red); _color.setY(green); _color.setZ(blue);
}
Vector4& getColor() { return _color; }
```

Classe Render

La classe Render permet de calculer le rendu des points à l'écran. C'est-à-dire qu'elle utilise la matrice de projection fournis par la camera et les affiches sur l'écran. L'intérêt de crée cette classe est de pouvoir effectuer des calculs complexes sur les points et les polygones (Z-buffer, Ray tracing, textures, etc..) et de gagner en abstraction.

Cette classe fonctionne donc conjointement avec la classe Camera et la class Screen.

Attributs :

Camera *_camera : Un pointeur sur la camera de la scène.
Screen *_screen : Un pointeur sur l'écran.
Node* _rootNode : Le nœud principal de notre scène
Matrix _projMat : La matrice de projection fournis par la camera.
double _aspectRatio : Le ratio de l'écran
int _renderMode : Le mode de rendu.

Constructeur :

Le constructeur de la classe Render attend, un pointeur sur une caméra de la scène, la nœud principal de la scène, le nom de la fenêtre, le type de librairie 2D à utiliser (OpenCv ou SFML) définit par les constantes e3d_, ainsi que la taille largeur, hauteur de la fenêtre à ouvrir.

```
Render(Camera* cam, Node* rootNode, string screenName, int renderModeSysteme,
int          width=800, int height=800);
```

Méthodes privé :

```
void   renderProcess(Node* nod, const Matrix& transformation) :
```

Méthode du processus de rendu. Cette méthode va permettre de parcourir l'arbre des nœuds à partir du nœud principal. Elle fonctionne de manière récursive et parcours l'arbre de nœud en nœud. Enfin, si un nœud ne possède pas d'autre sous nœud, elle fait appel à la fonction processToObject pour traiter les objets qui sont attaché au nœud actuel.

```
void   processToObject(Node* nod, const Matrix& transformation) :
```

Méthode qui permet de faire le rendu d'un objet. En testant le type réel de l'objet (Ex : Entity) et le trace à l'écran. Premièrement avec les points qui lui sont associées puis ses polygones.

```
void   RenderPoint(Entity* r_Entity, vector<Vector4>& calc, const Matrix&
transformation) :
```

Méthode qui calcule la projection des points d'une Entity et stock les points projeté dans un tableau de Vector4 qui permettent de tracer les polygones.

```
void   RenderLigne(Entity* r_Entity, vector<Vector4>& calc) :
```

Cette méthode permet de tracer les lignes associées à une Entity à partir des coordonnées des points projetés à l'écran.

```
void   RenderLabel(Label* lab, const Matrix& transformation) :
```

Cette méthode permet d'afficher un texte à l'écran suivant la matrice de transformation du rootNode.

Méthodes public :

```
void   start(); //Calcule le rendu des pts
```
La méthode start est le point d'entrée principal après l'initialisation, du calcul des points.

```
void   loop(); //Exemple de boucle de rendu possible
```
Ceci c'est un exemple de boucle de rendu.

```
void   clean() { _screen->clean(); }
```
Cette Méthode permet d'effacer l'écran.

```
void   update() { _screen->update(); }
```
Cette méthode permet de mettre à jour l'écran (si les points ont changés).

Accesseurs et mutateurs :

```
Camera* getCamera() { return _camera; }
```
Retourne le pointeur sur la camera.
```
void   setCamera(Camera* cam) {_camera = cam; }
```
Permet de changer la camera.

```
Node*  getRootNode() { return _rootNode; };
```
Retourne le pointeur vers le noeud principal.
```
void   setRootNode(Node* rootNode) { _rootNode = rootNode; }
```
Permet de changer le nœud principal.

```
int    getRenderMode() { return _renderMode; }
```
Permet d'obtenir la méthode de rendu. (non implémenté de cette révision).
```
void   setRenderMode(int renderMode) { _renderMode = renderMode; }
```
Permet de modifier la methode de rnendu. (non implémenté de cette revision).

Remarque :

L'intérêt de pouvoir changer le rootNode permet dans le cas d'un changement de scène (niveau), de permettre de le charger en avance en mémoire (thread) et ainsi effectuer le changement de scène rapidement.

Le render doit être utilisé dans une boucle pour simuler le mouvement à l'écran.

Exemple de code pour le rendu:

```
Render r(&camera,&RootNode,"Experiment",e3d_SFMLRender);
while(1)
{
	r.clean(); // efface l'écran
	r.start(); // Maj les pts
	r.update(); //Maj de l'écran
}
```

Constantes disponible :

`const unsigned int e3d_OpenCvRender` (Effectuer le rendu avec OpenCV).
`const unsigned int e3d_SFMLRende` (Effectue le rendu avec SFML).

Classe Screen

La classe Screen sert d'interface pour afficher des des points/des lignes/du texte dans une fenêtre. C'est une classe abstraite. Elle est normalement utilisé par le Render pour afficher les objets à l'écran.

Attribut protected :

`string _windowName` : Le nom de la fenêtre
`int _height` : La hauteur de la fenêtre
`int _width` : La largeur de la fenêtre
`int _backgroundColor` : couleur de fond de la fenêtre

Constructeur :

```
Screen(string screenName="E3D Engine", int width=800, int height=800, int backgroundColor = 5);
```

Le constructeur par défaut de la classe Screen permet de crée une fenêtre. Il attend le nom de la fenêtre, la largeur, la hauteur et la couleur de fond.

```
virtual void init()=0;
```
Méthode virtuel pure d'initialisation de la fenêtre.

```
virtual void clean()=0;
```
Méthode virtuel pure pour effacer l'écran.

```
virtual void update()=0;
```
Méthode virtuel pure pour mettre à jour l'écran.

```
virtual void drawLine ( double p1, double p2, double p3, double p4, int red, int green, int blue)=0;
```
Méthode virtuel pure pour tracer une ligne à partir des coordonnées de deux points ainsi que la couleur RGB de la ligne.

```
virtual void drawPoint( double p1, double p2, double p3, double p4, int red, int green, int blue)=0;
```
Méthode virtuel pure pour tracer un point des coordonnées de deux points (Les mêmes coordonnées) ainsi que la couleur RGB du point.

```
virtual void drawLabel (std::string label, double x, double y, int red, int green,int blue)=0;
```
Methode virtuel pure pour afficher un label à l'écran. A partir du label, les cordonnées ou le texte doit être affiché, et la couleur RGB.

```
virtual void render()=0;
```
Méthode virtuel pure d'exemple d'affichage.

Accesseurs et mutateurs :

```
int getWidth() { return _width; }
int getHeight() { return _height; }
```
Retourne la hauteur et la largeur de la fenêtre.

```
int setWidth(int w) { _width = w; }
int setHeight(int h) { _height = h; }
```
Modifie la hauteur et la largeur de la fenêtre.

Classe ScreenSFML

La classe ScreenSFML est une implémentation de la classe Screen. Elle utilise la librairie SFML pour afficher des des points/des lignes/du texte dans une fenêtre.

Attribut :

```
sf::RenderWindow _window : Fenêtre SFML
vector<sf::Vertex> _line : Tableau de Lignes
vector<sf::Vertex> _point : Tableau de points
vector<sf::Text> _label   : Tableau de texte
sf::Font *_font : Police pour les textes
```

Constructeur :

```
ScreenSFML(string name, int width, int height, int backgroundColor).
```

Le constructeur attend le nom de la fenêtre, sa hauteur/largeur et sa couleur de fond.

Destructeur :

```
~ScreenSFML(void);
```

Le destructeur détruit la fenêtre.

Méthodes :

Les autres méthodes sont une implémentation de la classe Screen avec la librairie SFML.

```
void init();
void clean();
void update();
void drawLine( double p1, double p2, double p3, double p4, int red, int green, int blue);
void drawPoint(double p1, double p2, double p3, double p4, int red, int green, int blue);
void drawLabel(string label, double x, double y, int red, int green, int blue);

void render();

int isOpen() { return _window.isOpen(); }
```

Permet de tester si une la fenêtre est ouverte

Classe ScreenOpenCV

La classe ScreenOpenCv est une implémentation de la classe Screen. Elle utilise la librairie OpenCV pour afficher des des points/des lignes/du texte dans une fenêtre.

Attribut :

```
Mat _image : Image OpenCV.
```

Constructeur :

```
ScreenOpenCV(string screenName, int width, int height, int
backgroundColor) :
```

Le constructeur attend le nom de la fenêtre, sa hauteur/largeur et sa couleur de fond.

Destructeur :

```
~ScreenSFML(void);
```

Le destructeur détruit la fenêtre.

Méthodes :

Les autres méthodes sont une implémentation de la classe Screen avec la librairie OpenCV.

```
~ScreenOpenCV(void);

void init();
void clean();
void update();
void drawLine( double p1, double p2, double p3, double p4, int red, int green,
int blue);
void drawPoint(double p1, double p2, double p3, double p4, int red, int green,
int blue);
void drawLabel(string label, double x, double y, int red, int green, int blue);
void render();
```

Classe ObjLoader

La classe ObjLoader permet de charger un Mesh en mémoire à partir d'un objet de type .obj . Seule la définition des vertex et des polygones du format .obj est effectif dans cette implémentation.

Ceci est une classe qui n'est pas demandé pour le rendu du projet, nous avons donc pris la liberté de nous inspirer de plusieurs sources internet, notamment :
http://fr.openclassrooms.com/informatique/cours/charger-des-fichiers-obj

Pour plus d'information sur le format .obj :
http://fr.wikipedia.org/wiki/Objet_3D_(format_de_fichier)

Attribut :

`Mesh *_m` : Un pointeur vers un objet de type Mesh
`vector<Vector4> normal,texture,color` : tableau de normal/texture/color.

Constructeur :

`ObjLoader();`
Le constructeur par défaut permet de créer un loader.

Structure complémentaire :

```
struct Vertex
{
    int v, vt, vn;
    Vertex() {};
    Vertex(int v) : v(v), vt(v), vn(v) {};
    Vertex(int v, int vt, int vn) : v(v), vt(vt), vn(vn) {};
};
```

Cette structure permet de définir un vexter/un vertex texture et un vertex normal.

Méthodes :

Ici on test si l'index n'a pas une valeur incorrecte.

```
int fix_v(int index) {
return(index > 0 ? index - 1 : (index == 0 ? 0 : (int)_m->getSize() + index));
}
int fix_vt(int index) {
return(index > 0 ? index - 1 : (index == 0 ? 0 : (int)texture.size() + index));
}
int fix_vn(int index) {
return(index > 0 ? index - 1 : (index == 0 ? 0 : (int)normal.size() + index));
}
```

Permet de lire 3 entiers depuis une chaine de caractère.

```
Vertex getInt3(const char*& token);
```

Permet de charger un mesh en mémoire et de renvoyer l'adresse du Mesh.

```
Mesh* load(const string& name, const string& filename);
```