Incrustation d'objets 3D dans une photo à l'aide de la bibliothèque PoLAR

Sommaire

Introduction :

Ce projet met en œuvre ce qu'on appelle la réalité augmentée, il s'agit du fait d'ajouter à des scènes provenant du monde réel (photos, vidéos) des objets virtuels. Dans notre cas il s'agit plus particulièrement d'intégrer des objets modélisés en 3D dans une photo numérique. Par exemple ajouter des meubles dans une pièce ou des objets pour la rendre plus vivante. L'intérêt de ce projet est l'utilisation de la bibliothèque PoLAR (http://polar.inria.fr) qui permet de faire de la réalité augmentée à partir de photos notamment. Cette bibliothèque va s'occuper dans notre cas de l'intégration des objets virtuels dans la photo « réelle ». Pour que les objets soient correctement rendus, il faut bien entendu fournir les bons paramètres à cette bibliothèque.

Le cas des ombres notamment est intéressant car en plus de fournir l'objet, il faut spécifier une surface de projection et la position d'une source de lumière par exemple. L'objectif du projet est donc d' « augmenter » des scènes à l'aide d'objets virtuels et de les rendre les plus convaincantes possible.

Installation de PoLAR :

Sous Ubuntu, l'installation de PoLAR est relativement simple, il suffit d'installer ses dépendances puis de le compiler avec cmake.

Les dépendances peuvent être installées avec apt-get (commande apt-get install <nom du paquet>)

Paquets de Qt5 (sauf pour Ubuntu <= 14.04, voir note plus bas) :

```
qt5-default qttools5-dev-tools libqt5opengl5-dev qtmultimedia5-dev
libqt5multimedia5-plugins qtdeclarative5-dev libqt5svg5-dev
```

OpenSceneGraph :

```
openscenegraph libopenscenegraph-dev
```

Gstreamer :

```
libgstreamer0.10-0 libgstreamer0.10-dev streamer-tools
```

Doxygen (pour la documentation)

```
doxygen graphviz
```

PoLAR utilise l'utilitaire Cmake :

```
Cmake-curses-gui
```

Le téléchargement de la bibliothèque PoLAR se fait grâce à la commande suivante (il peut donc être nécessaire d'installer git s'il n'est pas déjà présent) :

```
git clone https://scm.gforge.inria.fr/anonscm/git/polar/polar.git
```

Pour compiler et lancer la bibliothèque, il faut se rendre dans le dossier polar et lancer la commande **ccmake .** un écran de configuration apparaît et on peut modifier un certain nombre d'options, comme le chemin des différentes librairies.

Il faut appuyer sur c pour configurer puis g pour générer le makefile. On peut ensuite quitter l'utilitaire.

Juste avant de compiler PoLAR avec la commande **make** il est nécessaire de configurer les variables d'environnement adéquates avec le fichier polar_config fourni. (l'exécuter avec ./polar_config)

Note pour Ubuntu <= 14.04 : Il ne faut pas télécharger Qt depuis les dépôts officiels mais depuis le site de Qt car la version 5.4 est requise.

Adresse de téléchargement de Qt : https://www.qt.io/download-open-source/

Dans CMake, il faut changer la configuration correspondant à l'emplacement de Qt et choisir de ne pas utiliser la version fournie dans les dépôts officiels.

Pour changer la configuration de cmake, il est nécessaire de supprimer le fichier CMakeCache.txt

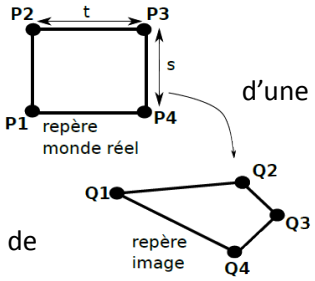
Calcul de la matrice de projection :

Une fois la photo sélectionnée et la bibliothèque PoLAR correctement installée, pour pouvoir ajouter un objet 3D dans notre photo, il faut calculer la matrice de projection pour pouvoir « traduire » les

formes présentes dans le mode réel de manière à ce que les objets apparaissent posés sur le sol ou la table par exemple.

Pour cela il est nécessaire de calculer la matrice de projection, cette matrice va permettre de positionner l'objet pour que la partie virtuelle ressemble au monde réel.

J'ai choisi de programmer ce calcul en python.



La première étape est de récupérer les positions des pixels aux coins d'une zone rectangulaire qui va permettre de traduire ces coordonnées en matrice après différents calculs.

Après avoir récupéré ces points (Q1-4), il faut les recentrer dans le repère de l'image :

$$Qi' = \begin{bmatrix} Qi_x - w/2 \\ Qi_y - h/2 \\ 0 \end{bmatrix}$$

avec w = width (la largeur de l'image)

H = height (la hauteur de l'image)

L'étape suivante est de calculer la distance focale, pour cela il faut calculer les lignes dde fuites données par nos points (l1-4) :

$$l1 = \begin{bmatrix} Q1'_x \\ Q1'_y \\ 1 \end{bmatrix} \otimes \begin{bmatrix} Q2'_x \\ Q2'_y \\ 1 \end{bmatrix}; l2 = \begin{bmatrix} Q4'_x \\ Q4'_y \\ 1 \end{bmatrix} \otimes \begin{bmatrix} Q3'_x \\ Q3'_y \\ 1 \end{bmatrix}; l3 = \begin{bmatrix} Q2'_x \\ Q2'_y \\ 1 \end{bmatrix} \otimes \begin{bmatrix} Q3'_x \\ Q3'_y \\ 1 \end{bmatrix}; l4 = \begin{bmatrix} Q1'_x \\ Q1'_y \\ 1 \end{bmatrix} \otimes \begin{bmatrix} Q4'_x \\ Q4'_y \\ 1 \end{bmatrix}$$

Avec ⊗ le produit vectoriel.

Les points d'intersections de ces lignes sont donnés par les points suivants (V et W) :

$$V^H = l1 \otimes l2; W^H = l3 \otimes l4; V = \begin{bmatrix} V_x^H / V_z^H \\ V_y^H / V_z^H \\ 1 \end{bmatrix}; W = \begin{bmatrix} W_x^H / W_z^H \\ W_y^H / W_z^H \\ 1 \end{bmatrix}$$

On peut alors directement calculer la distance focale :

$$f = \sqrt{-(Vx.Wx + Vy.Wy)}$$

Cette distance donne accès à la matrice intrinsèque indépendante du point de vue (K) :

$$K=\begin{bmatrix} f & 0 & w/2 \\ 0 & f & h/2 \\ 0 & 0 & 1 \end{bmatrix}$$

Il nous faut ensuite la matrice d'homographie h :

$$A=\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & -Q2'_x & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & -Q2'_y & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & -Q3'_x & -Q3'_x \\ 0 & 0 & 0 & 1 & 1 & 1 & -Q3'_y & -Q3'_y \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & -Q4'_x \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & -Q4'_y \end{bmatrix} ; Qi'=\begin{bmatrix} Q1'_x \\ Q1'_y \\ Q2'_x \\ Q2'_y \\ Q3'_x \\ Q3'_y \\ Q4'_x \\ Q4'_y \end{bmatrix}$$

h est donnée par la formule **$h=A^{-1}.Qi'$**

Il faut ensuite transformer h de la façon suivante pour obtenir une matrice 3X3 :

$$\begin{bmatrix} h11 \\ h12 \\ h13 \\ h21 \\ h22 \\ h23 \\ h31 \\ h32 \end{bmatrix} => \begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & 1 \end{bmatrix}$$

On doit ensuite calculer une nouvelle matrice intermédiaire B :

$$B=\begin{bmatrix} 1/f & 0 & 0 \\ 0 & 1/f & 0 \\ 0 & 0 & 1 \end{bmatrix}.h$$

On pose alors : B1 : la première colonne de B, B2 la 2eme colonne etc…

Ces valeurs nous permettent de calculer R1-3 et T :

$$
\begin{array}{lcl}
R1 & = & B1/\|B1\| \\
R3 & = & R1 \otimes B2/(\|R1 \otimes B2\|) \\
R2 & = & R3 \otimes R1 \\
\lambda & = & \|B1\|/s \\
t_{calcule} & = & \|B2\|/\lambda \\
T & = & B3/\lambda
\end{array}
$$

Avec ces vecteurs, on reforme la matrice extrinsèque E :

(R1-3 et T forment les colonnes de cette matrice)

$$
E = \begin{bmatrix} \vdots & \vdots & \vdots & \\ R1 & R2 & R3 & T \\ \vdots & \vdots & \vdots & \end{bmatrix}
$$

Grâce à E et K calculée précédemment on peut enfin récupérer la matrice de projection :

M = K.E

Création d'un projet personnalisé :

Une fois la matrice de projection calculée, il faut l'enregistrer au format texte, on peut alors l'utiliser directement dans les exemples fournis par la bibliothèque PoLAR.

Par exemple l'exécutable svisu3 (/bin/svisu3) prend en paramètres une matrice de projection et un objet 3D. Il permet de visualiser un objet dans une photo. L'objet aura alors une taille et une position cohérente par rapport à l'image.

L'affichage reste très simple et manque de flexibilité, c'est pourquoi on peut recréer un projet PoLAR qui appelle la librairie et qui contiendra la code spécifique à notre application (avec par exemple des transformations sur l'objet ou des objets transparents).

Le rôle des objets transparents est de simuler les propriétés d'un objet réel (par exemple une table). On ajoute un objet ayant une forme simple (un rectangle par exemple) que l'on rend transparent. Des ombres pourront alors se projeter sur l'objet.

L'autre utilité des objets transparents est de cacher un objet. En effet, si dans notre projet on veut ajouter un objet derrière un autre, il faut faire un objet transparent que l'on mettra sur l'objet en premier plan pour qu'il puisse cacher celui qui va se situer à l'arrière-plan. Cette méthode va nous aider à obtenir un rendu beaucoup plus réaliste.

Les étapes de la création d'un projet externe à la bibliothèque sont relativement simples, il suffit de reprendre le fichier CMakeList.txt de l'exemple fourni (runPolar) et de changer les lignes 250 à 262

par les noms de nos propres fichiers (changer également les appels à runPolar.cpp par le nom de fichier du projet).

Il faut également supprimer les liens vers openCV qui ne sont pas utiles dans notre cas (l.210-230 environ).

Il est aussi nécessaire d'ajouter au début du fichier le chemin vers le dossier ou se situe la bibliothèque PoLAR.

```
set(PoLAR_LIBRARY "/home/valentin/Documents/pt4/polar/lib/libPoLAR.so")
```

```
set(PoLAR_SOURCE_DIR "/home/valentin/Documents/pt4/polar/")
```

Une fois cette étape réalisée il faut coder notre projet dans le fichier projet.cpp, pour le compiler il faudra procéder comme pour PoLAR et lancer la commande « **ccmake .** » dans le dossier du projet.

La commande **make** permet ensuite de compiler le projet.

- ## Création d'une fenêtre polar :

La création d'une fenêtre PoLAR est relativement simple, il faut importer le fichier « Viewer.h », c'est l'objet viewer qui va ensuite gérer les fonctions liées à la fenêtre.

La spécification de la taille de la fenêtre se fait à l'aide de la méthode resize de l'objet viewer.

Création d'un Viewer simple et chargement d'une image de fond (« fond.png »). La définition de l'image de fond se fait à l'aide de setBgImage.

```
// Create the Qt application
QApplication app(argc, argv);
PoLAR::Viewer viewer(0, "Viewer", 0, true);
//MyViewer viewer (0, "Viewer", 0);
osg::ref_ptr<PoLAR::Image<unsigned char> > myImage;

myImage = new PoLAR::Image<unsigned char>("fond.png", true, 1);
viewer.setBgImage(myImage);
viewer.bgImageOn();

// Resize the widget
int width = myImage->getWidth();
int height = myImage->getHeight();
viewer.resize(width, height);
```

Il ne reste plus qu'à demander au programme de s'afficher :

```
viewer.show();
app.connect( &app, SIGNAL(lastWindowClosed()), &app, SLOT(quit()) );
return app.exec();
```

Le résultat est le suivant :

--->

- ## Ajout d'objets 3D dans l'image :

Pour ajouter un objet 3D il faut charger son fichier puis éventuellement le déplacer avec des translations pour le placer correctement dans la scène.

Mais avant cela il faut définir la matrice de projection, sinon l'objet n'aura pas un aspect réaliste dans la scène.

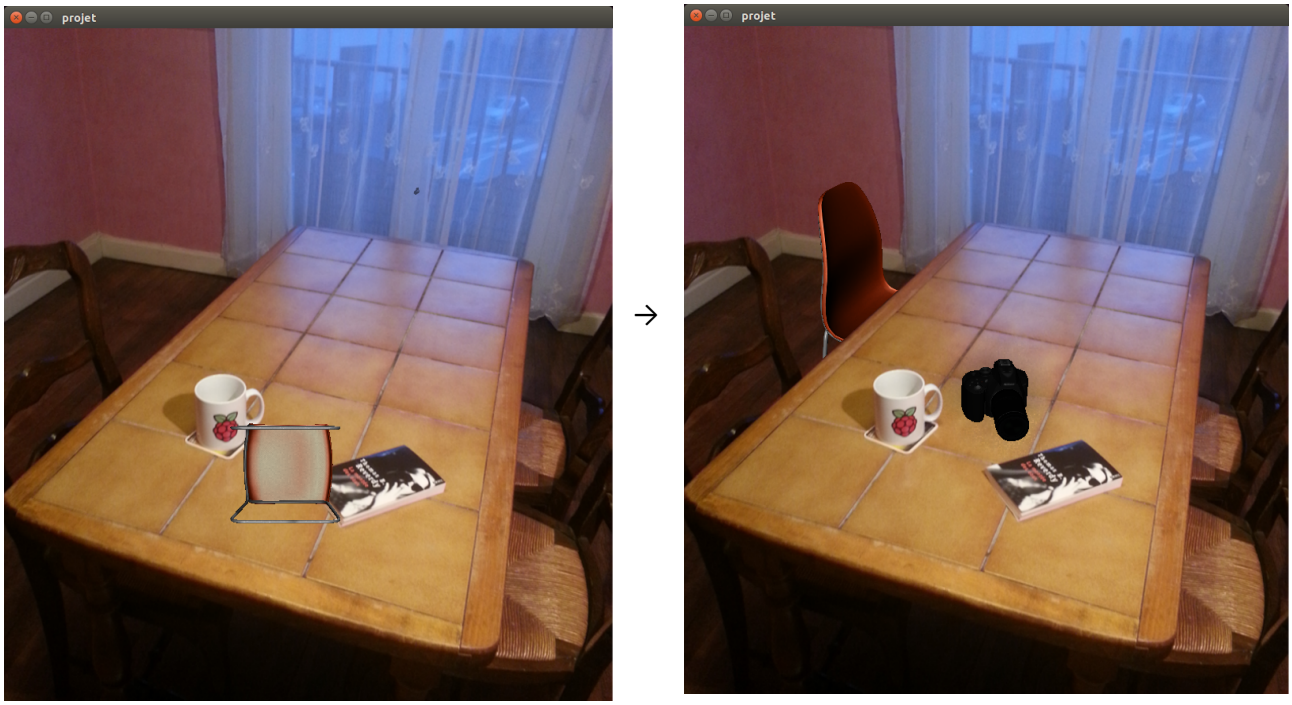
```
PoLAR::Viewer::ProjectionType pt=PoLAR::Viewer2D3D::VISION;
osg::Matrixd P = PoLAR::Util::readProjectionMatrix("mat.proj");
viewer.setProjection(P, pt);
```

Ajout d'un objet au viewer PoLAR :

```
osgUtil::SmoothingVisitor smooth;
osgUtil::Optimizer optimizer;

//chargement de la chaise
osg::ref_ptr<osg::Node> loadedModel2;
loadedModel2 = osgDB::readNodeFile("../data/plastic_chair.obj");
optimizer.optimize(loadedModel2.get());
loadedModel2->accept(smooth);
osg::ref_ptr<PoLAR::Object3D> obj2 = new PoLAR::Object3D(loadedModel2.get(), true, true);

obj2->scale(0.5f);
obj2->translate(-42,85,-100);
obj2->rotate(M_PI/2, 0, 0, 1);

viewer.addObject3D(obj2.get());
viewer.setTrackNode(obj2.get());
```

Après la définition des objets utiles au chargement dde l'objet, on peut effectuer des modifications (taille, translations, rotations, etc.)

objet affiché sans matrice de projection avec la matrice de projection(et translations)



- ## Ajout d'ombres :

Pour pouvoir avoir un affichage correct des ombres, il est nécessaire d'avoir une surface sur laquelle l'ombre doit se projeter et une source de lumière qui peut produire des ombres.

On peut gérer cela avec les objets fantômes dans polar. Par exemple pour l'image ci-dessus j'ai choisi de faire un rectangle qui représente le plateau de la table. De cette façon l'appareil photo pourra projeter son ombre.

Le placement de la lumière doit correspondre à la position de la source de lumière dans le monde réel pour que l'effet soit le plus réussi possible.

L'ajout d'une source de lumière est très simple :

```
viewer.addLightSource(5,-5, 5.5, true);
```

Les paramètres sont les coordonnées de cette source, true signifie que cette source de lumière peut provoquer la création d'ombres. Une seule source de lumière dans la scène peut générer des ombres.

Création du rectangle :

```
osg::ref_ptr<osg::Geode> tableGeode = new osg::Geode;
osg::ref_ptr<osg::ShapeDrawable> shape;
shape = new osg::ShapeDrawable(new osg::Box(osg::Vec3(1.0f, 2.5f, 0.0f), 7.0f, 12.0f, 0.00001f));
tableGeode->addDrawable(shape.get());
osg::ref_ptr<PoLAR::Object3D> object3D = new PoLAR::Object3D(tableGeode.get());
object3D->setName("Table");
object3D->setPhantomOn();
viewer.addObject3D(object3D.get());
```

setPhantomOn() permet de rendre l'objet invisible, il pourra cependant cacher les objets positionnés derrière lui (la chaise) et également faire apparaître les ombres (de l'appareil photo par exemple).

On voit sur la 2ème image l'effet dde la source de lumière ajoutée à la scène.

Après avoir rendu transparent le rectangle :

- ## Animations des objets :

Pour gérer l'animation des objets, il faut créer une nouvelle classe qui étend les fonctionnalités de la classe Viewer.

Il faut ensuite gérer les évènements Qt pour lire les entrées clavier. On peut par exemple faire disparaître les objets 3D (voir MyViewer.h) :

```
case Qt::Key_F1:
{
    osg::ref_ptr<PoLAR::Object3D> o=getObject3D(0);
    osg::ref_ptr<PoLAR::Object3D> o2=getObject3D(1);
    if (o.valid())
    { // toggle the display of the Object3D
        if (o->isDisplayed())
            {
                o->setDisplayOff();
                o2->setDisplayOff();
            }
        else
        {
          o->setDisplayOn();
          o2->setDisplayOn();
        }
    }
    break;
}
```

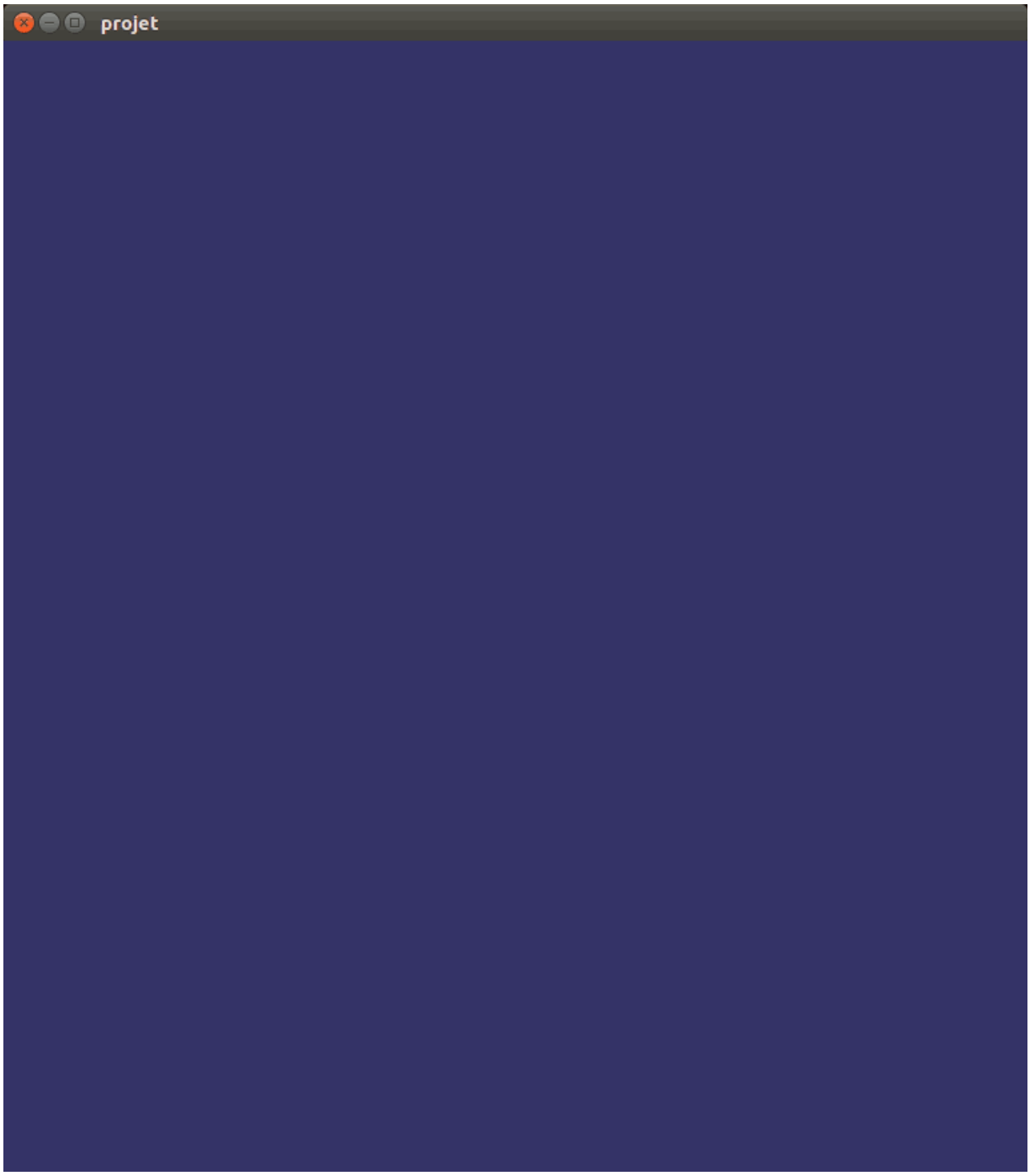
On peut accéder directement aux objets à partir d'un index représentant leur ordre d'ajout dans la scène.

SetDisplayOff() permet de faire disparaître un objet.

On peut également les faire tourner avec la fonction rotate.

```
o->rotate(0.01, 0., 0., 1.);
```

Présentation des différentes opérations réalisées :


projet