

Tutoriel : Créez des programmes en 3D avec OpenGL

Table des matières

Créez des programmes en 3D avec OpenGL (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Crezdesprogrammesen3DavecOpenGL>)

Introduction à OpenGL (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#IntroductionOpenGL>)

Qu'est-ce qu'OpenGL ? (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Qu039est-cequ039OpenGL>)

Que dois-je installer ? (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Quedois-jeinstaller>)

Première application OpenGL avec SDL (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#PremireapplicationOpenGLavecSDL>)

Notions de base (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Notionsdebase>)

Couleur et interpolation (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Couleuretinterpolation>)

Vertices et polygones (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Verticesetpolygones>)

Fonctions et types (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Fonctionsettypes>)

Les transformations (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Lestransformations>)

Un brin de folie mathématique (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Unbrindefoliemathmatique>)

Les matrices OpenGL (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#LesmatricesOpenGL>)

Les transformations (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Lestransformations>)

Exercice : une grue (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Exerciceunegrue>)

Enfin de la 3D (Partie 1/2) (http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Enfindela3DPartie1_2)

Du réel à l'écran (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Durell039cran>)

La perspective (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Laperspective>)

Placer la caméra (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Placerlacamra>)

Enfin de la 3D (Partie 2/2) (http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Enfindela3DPartie2_2)

Un cube (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Uncube>)

Le Z-Buffer (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#LeZ-Buffer>)

Animation (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Animation>)

Les textures (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Lestextures>)

Charger une texture (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Chargerunetexture>)

Plaquage de texture (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Plaquagedetexture>)

Texture répétitive (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Texturerptitive>)

Les couleurs (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Lescouleurs>)

Les quadriques (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Lesquadriques>)

Principe d'utilisation (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Principed039utilisation>)

Les quadriques (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Lesquadriques>)

Exercice : une roquette (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Exerciceuneroquette>)

Contrôle avancé de la caméra (Partie 1/2) (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Controleavancecamera>)

opengl.html#ContrleavancadelacamraPartie1_2)
Principe d'une caméra TrackBall (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Principed039unecamraTrackBall>)
Quelques bases de C++ (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#QuelquesbasesdeC>)
Implémentation de la caméra (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Implmentationdelacamra>)
Scène de test (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Scnedetest>)
Contrôle avancé de la caméra (Partie 2/2) (http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#ContrleavancadelacamraPartie2_2)
Principe d'une caméra FreeFly (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Principed039unecamraFreeFly>)
Gestion fluide du mouvement (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Gestionfluidedumouvement>)
Implémentation de la caméra (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Implmentationdelacamra>)
Scène de test (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Scnedetest>)
La trigonométrie (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Latrigonometrie>)
Trigo dans un triangle rectangle (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Trigodansuntrianglerectangle>)
Le cercle trigonométrique (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Lecercletrigonometrique>)
Systèmes de coordonnées (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Systmesdecoordonnes>)
Les matrices (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Lesmatrices>)
L'outil matrice (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#L039outilmatrice>)
Transformations (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Transformations>)
Combinaison de transformations (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Combinaisondetransformations>)
Créer une vidéo de votre programme (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Crerunevidodevotreprogramme>)
Enregistrer la vidéo (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Enregistrerlavido>)
Encoder la vidéo (<http://sdz.tdct.org/sdz/creez-des-programmes-en-3d-avec-opengl.html#Encoderlavido>)

Créez des programmes en 3D avec OpenGL

L'envie vous prend soudainement d'ajouter une nouvelle dimension à votre vie mais vous avez peur de vous lancer car vous pensez que faire des programmes en **3D** est réservé à l'élite ?

Rassurez-vous l'apprentissage de la programmation 3D *via OpenGL* est relativement simple et par le biais de ce tutoriel vous connaîtrez les bases d'OpenGL ainsi que de multiples techniques utilisées dans les **jeux vidéo**.

Bonne lecture et surtout bonne création ! :soleil:

Ce cours nécessite la lecture préalable du cours de C / C++ (<http://www.siteduzero.com/tuto-3-8-0-apprenez-a-programmer-en-c.html>) rédigé par M@teo21. Il faut avoir lu au moins les trois premières parties (partie sur la SDL incluse).

Introduction à OpenGL

Dans ce chapitre nous découvrirons brièvement les possibilités d'OpenGL et réaliserons notre première application OpenGL grâce à SDL.

Qu'est-ce qu'OpenGL ?



OpenGL (Open Graphics Library) est une bibliothèque graphique très complète qui permet aux programmeurs de développer des applications 2D, 3D assez facilement.

Vous avez déjà dû l'utiliser ou en entendre parler, car de nombreux jeux, comme Quake III, proposent l'OpenGL comme mode d'affichage.



Quake III Arena

Bon d'accord, je dois vous l'avouer après avoir lu ce tuto vous ne saurez pas encore faire des jeux comme ça (notamment à cause du moteur physique pour gérer les collisions, déplacements et projectiles) mais il est beau de rêver. :p

OpenGL s'utilise principalement en C++, c'est pourquoi il est conseillé de connaître ce langage. Mais rassurez-vous ! Même sans base vous arriverez parfaitement à comprendre tous les exemples du cours, cependant les applications que vous serez amenés à développer par vous-mêmes seront peut-être moins poussées. La lecture du tuto C/C++ (<http://www.siteduzero.com/tuto-3-8-0-apprenez-a-programmer-en-c.html>) est **fortement recommandée**.

Il est possible d'utiliser OpenGL en Python, Delphi, Java, et les commandes y sont similaires. Ici nous utiliserons le C/C++ pour profiter du cours déjà présent sur le site.

Une multitude de fonctionnalités

OpenGL dispose de nombreuses fonctions que vous pourrez « facilement » utiliser, notamment la gestion de :



- la caméra ;
- la rotation 3D des objets ;
- le remplissage des faces ;
- les textures ;
- la lumière ;
- et bien plus encore...

Dans ce tuto je compte vous montrer un maximum de techniques telles que la génération d'un **terrain 3D**, un moteur de **particules**, le **bump-mapping**, le **cell-shading**, etc. Comment alors ne pas saliver devant toutes ces possibilités qui s'ouvrent à vous ? D'autant que la plupart sont très accessibles d'un point de vue difficulté de programmation.

D'un point de vue pédagogique, nous ne verrons la 3D qu'à partir du 4e chapitre, car des connaissances de base sont nécessaires afin de se lancer correctement.

Fenêtrage et événements

OpenGL ne fournit que des fonctions 3D qui doivent être exécutées dans un contexte graphique déjà créé. Il nous faut donc choisir un système pour créer les fenêtres et gérer les événements pour donner une interactivité aux applications. OpenGL étant implémenté sur de nombreuses plates-formes, j'ai choisi de vous faire utiliser la **SDL** (d'autant que son installation/utilisation vous est enseignée dans le cours de M@teo (<http://www.siteduzero.com/tuto-3-8-0-apprenez-a-programmer-en-c.html>)).

La SDL nous permettra ainsi :

- de créer une fenêtre ;
- de charger très facilement des textures grâce à *SDL_image* ;
- d'utiliser le clavier et la souris ;
- d'animer nos scènes.

Ainsi nous combinerons la facilité d'utilisation de la SDL avec la puissance d'OpenGL tout en gardant le côté multi plate-formes.

Et les moteurs 3D ?

Je reviendrai en détail plus tard dans ce tutoriel sur ce qui diffère OpenGL des moteurs 3D et sur comment créer son moteur 3D basé sur OpenGL. Quoi qu'il en soit ce que vous apprendrez ici est souvent applicable ailleurs et vous fournit des explications générales sur la programmation 3D. Se plonger dans un moteur tel **Irrlicht** ou **Ogre** sera d'autant plus facile que vous saurez ce qu'ils utilisent derrière.

Que dois-je installer ?

Vous êtes enfin décidés à vous lancer ? ^^ Parfait !

Les fichiers nécessaires pour développer

Si vous avez téléchargé **Code::Block** ou **DevC++** pour suivre le tuto de C/C++ ne changez rien, vous avez déjà les headers et les .a nécessaires. Vous devez donc avoir :

OS	include	lib
Windows	GL/gl.h GL/glu.h	libopengl32.a* libglu32.a
Unix**	libopengl.a libglu.a	

* Si vous utilisez Visual Studio, vous devez déjà avoir l'équivalent en .lib.

** Nous verrons lors de la création de notre premier programme quoi rajouter exactement sous Linux.

Les fichiers nécessaires pour exécuter

- Sous **Windows** sauf cas rare très exceptionnel vous avez déjà les .dll nécessaires soit : opengl32.dll et glu32.dll. Vous n'aurez pas à les fournir avec votre exécutable, car ils sont présents par défaut sous Windows.

Vous devrez continuer à fournir les .dll de la SDL.

- Sous **Linux** il vous faut les .so associés au .a.

Vérifier que l'accélération 3D est activée

Avoir les bons fichiers ne signifie pas forcément que les applications 3D tourneront parfaitement. Si l'accélération matérielle n'est pas activée, les performances en seront très fortement diminuées.

Vérifiez donc que vous avez bien installé les derniers drivers de votre carte graphique. Sous Windows si vous avez l'habitude de jouer à des jeux vidéos sans problèmes c'est que c'est bon ! :D

Sous Linux utilisez le fameux programme glxgears pour tester. Les instructions étant trop dépendantes de votre distribution et de votre carte, reportez-vous aux nombreuses ressources disponibles sur Internet.

Première application OpenGL avec SDL

Je vous l'ai dit plus haut nous allons utiliser OpenGL dans un contexte SDL. La première chose à faire est donc de créer un projet SDL de base (comme expliqué dans le tuto de M@teo). Nous viendrons y remplacer tout le code et compléter les options de compilation pour rajouter OpenGL lors de la phase d'édition des liens.

Si vous utilisez un IDE qui fournit des templates de projet (comme Code::Blocks), ne choisissez pas un projet OpenGL mais bien SDL. Nous le compléterons pour avoir un projet SDL/OpenGL.

Je fournirai d'ailleurs *en fin de chapitre* le projet final utilisé et vous pourrez vous en servir comme point de départ pour vos applications OpenGL créées en suivant ce tutoriel.

Code de départ

Souvenez-vous du code minimal pour ouvrir une fenêtre SDL :

```
#include <SDL/SDL.h>

int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_VIDEO);

    SDL_Surface* ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);

    SDL_Flip(ecran);

    bool continuer = true;
    SDL_Event event;

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = false;
        }
    }

    SDL_Quit();

    return 0;
}
```

Je ne respecte volontairement pas les règles du C car je compile en C++ et vous inviterai à en faire autant dans le futur pour suivre ce tutoriel. En effet les concepts que nous verrons plus tard seront beaucoup plus facilement implémentables en C++. Si vous ne connaissez que le C pour l'instant, rassurez-vous, nous introduirons le concept « objet » de manière simple et intuitive.

Initialiser SDL en mode OpenGL

La première chose à changer est au niveau de l'initialisation de mode vidéo. Nous allons utiliser `SDL_OPENGL` au lieu de `SDL_HWSURFACE` :

```
SDL_SetVideoMode(640, 480, 32, SDL_OPENGL);
```

Que devient le `SDL_DOUBLEBUF` utilisé avec la `SDL` normalement pour activer le double-buffering (cf. Travailler avec le double buffer (http://www.siteduzero.com/tuto-3-5790-1-la-gestion-des-evenements-clavier-et-souris.html#ss_part_3) ?

L'équivalent avec OpenGL reviendrait à appeler `SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1)`; mais il s'avère qu'il est **déjà activé par défaut** pour OpenGL. Rien de spécial à faire donc.

Le premier dessin

Ensuite nous allons faire nos premiers appels OpenGL pour dessiner un simple triangle. Nous rajoutons donc les headers nécessaires :

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

Avec **Visual Studio** (uniquement), il faut rajouter `#include <windows.h>` avant d'inclure les headers OpenGL.

Voici le code du triangle qui vous sera expliqué dans le prochain chapitre :

```
glClear(GL_COLOR_BUFFER_BIT);

glBegin(GL_TRIANGLES);
    glColor3ub(255,0,0);    glVertex2d(-0.75,-0.75);
    glColor3ub(0,255,0);    glVertex2d(0,0.75);
    glColor3ub(0,0,255);    glVertex2d(0.75,-0.75);
glEnd();
```

`SDL_Flip(ecran);` disparaît au profit de l'appel suivant :

```
glFlush();
SDL_GL_SwapBuffers();
```

La première commande, `glFlush`, vient s'assurer que toutes les commandes OpenGL ont été exécutées, et `SDL_GL_SwapBuffers` est l'équivalent de l'ancien `SDL_Flip`.

Le code complet

```

#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>

int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_VIDEO);
    SDL_WM_SetCaption("Mon premier programme OpenGL !",NULL);
    SDL_SetVideoMode(640, 480, 32, SDL_OPENGL);

    bool continuer = true;
    SDL_Event event;

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = false;
        }

        glClear(GL_COLOR_BUFFER_BIT);

        glBegin(GL_TRIANGLES);
            glColor3ub(255,0,0);    glVertex2d(-0.75,-0.75);
            glColor3ub(0,255,0);    glVertex2d(0,0.75);
            glColor3ub(0,0,255);    glVertex2d(0.75,-0.75);
        glEnd();

        glFlush();
        SDL_GL_SwapBuffers();
    }

    SDL_Quit();

    return 0;
}

```

Compilation

Pour compiler il nous faut rajouter les fichiers cités plus haut. Dans nos options de projet nous rajoutons donc :

IDE / Compilateur	lib
Code::Blocks Windows	opengl32 glu32
DevC++ Windows	-lopengl32 -lglu32

IDE / Compilateur

lib

Visual Studio Windows

opengl32.lib
glu32.lib

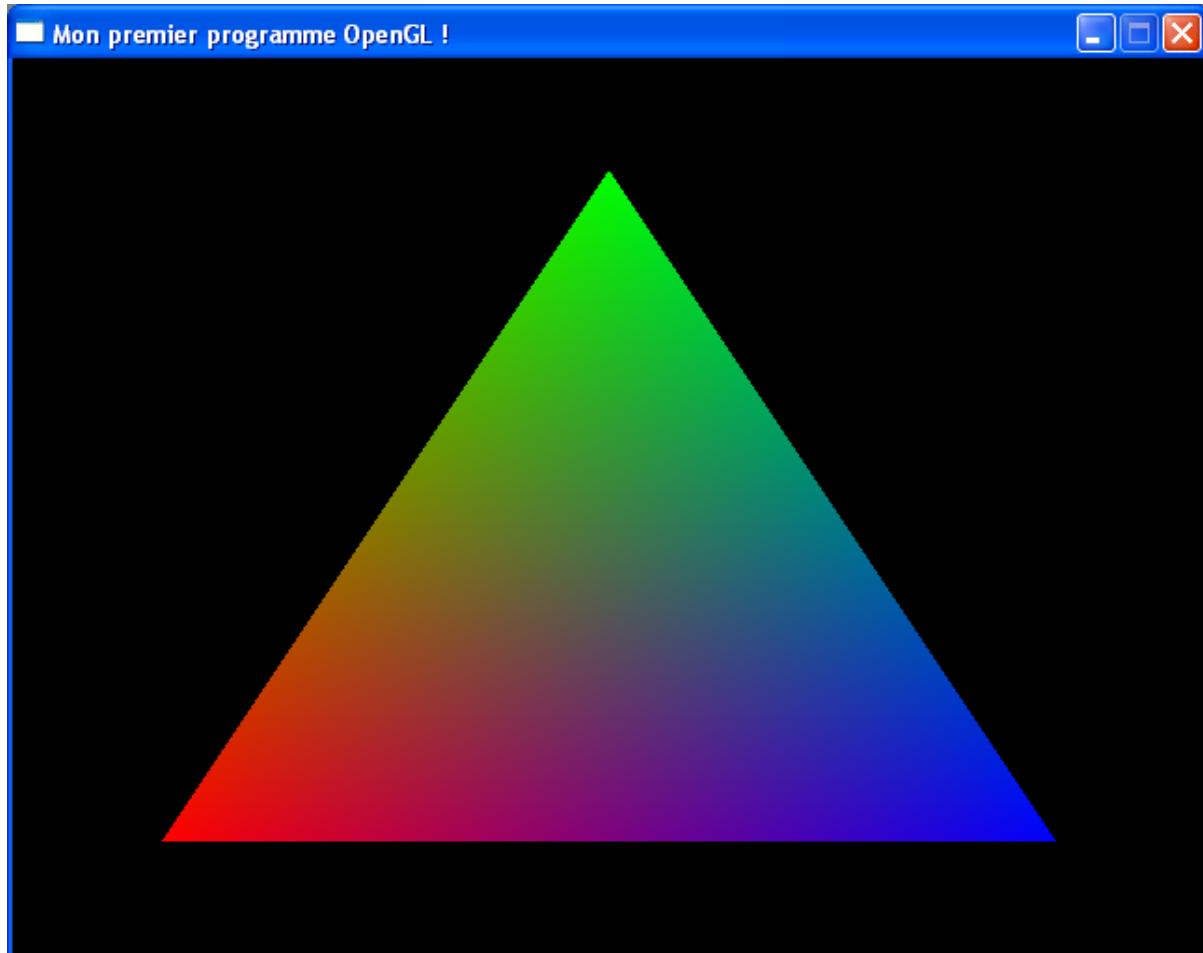
Code::Blocks Linux

GL
GLU

gcc / g++ Linux

-lGL -lGLU

Résultat



Téléchargez le projet Code::Blocks final, l'exécutable Windows et le Makefile Unix (116 Ko)
(http://62.4.17.167/uploads/fr/ftp/kayl/sdz_01_intro.zip)

Pour les utilisateurs de Code::Blocks, vous pouvez ouvrir le projet fourni et faire Project > Save project as user-template pour créer un projet de départ pour toutes vos applications SDL/OpenGL. Ainsi plus tard vous n'aurez qu'à faire File > New Project... > (Onglet) User templates > Votre template pour créer un projet opérationnel immédiatement.

Et voilà ce n'était pas sorcier ! :p Direction le prochain chapitre pour comprendre le code OpenGL que nous avons utilisé et pour dessiner plein de jolies choses.

Notions de base

Nous allons voir dans ce chapitre quelques concepts de base en OpenGL.

Bien que les exemples donnés soient en 2D vous verrez qu'ils seront toujours applicables quand nous passerons à la vraie 3D.

Couleur et interpolation

Dans le screenshot final du chapitre précédent (le triangle coloré) vous avez pu admirer la pureté et la beauté des couleurs en OpenGL.

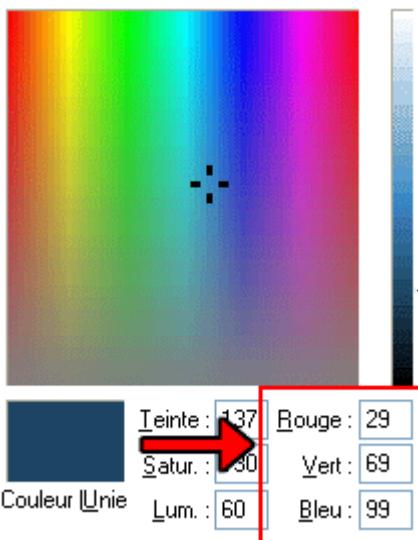
Pour définir la couleur d'un vertex on utilise :

```
glColor3ub(rouge, vert, bleu);
```

Les valeurs de rouge, vert et bleu sont des entiers entre 0 et 255.

Nous aurions pu aussi bien utiliser glColor3f (avec donc des composantes de couleurs entre 0 et 1) mais la plage de valeur [0..255] est plus facile à comprendre car fortement utilisée dans les logiciels de dessins.

À partir d'une couleur précise, pour connaître ses composantes RVB (ou RGB en anglais) il suffit d'utiliser la palette de couleurs de Windows (par exemple en utilisant Paint).



Un appel à glColor affecte la couleur courante, celle qui sera utilisée pour les vertices qui seront définis après. On peut ainsi changer de couleur à chaque vertex ou juste de temps en temps.

Et pour les points qui ne sont pas des sommets, comment la couleur est-elle définie ?

Les couleurs des points qui constituent une face sont calculées par **interpolation** des couleurs des sommets.

Citation : Dictionnaire

Interpolation : évaluation de la valeur d'une fonction entre deux points de valeurs connues.

Cela donne donc lieu à de jolis dégradés entre les sommets.

Exercice

Modifier le code du chapitre précédent pour dessiner un rectangle avec un dégradé bleu/rouge.

Corrigé

```
#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>

int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_VIDEO);
    SDL_WM_SetCaption("Un joli carr\u00e9",NULL);
    SDL_SetVideoMode(640, 480, 32, SDL_OPENGL);

    bool continuer = true;
    SDL_Event event;

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = false;
        }

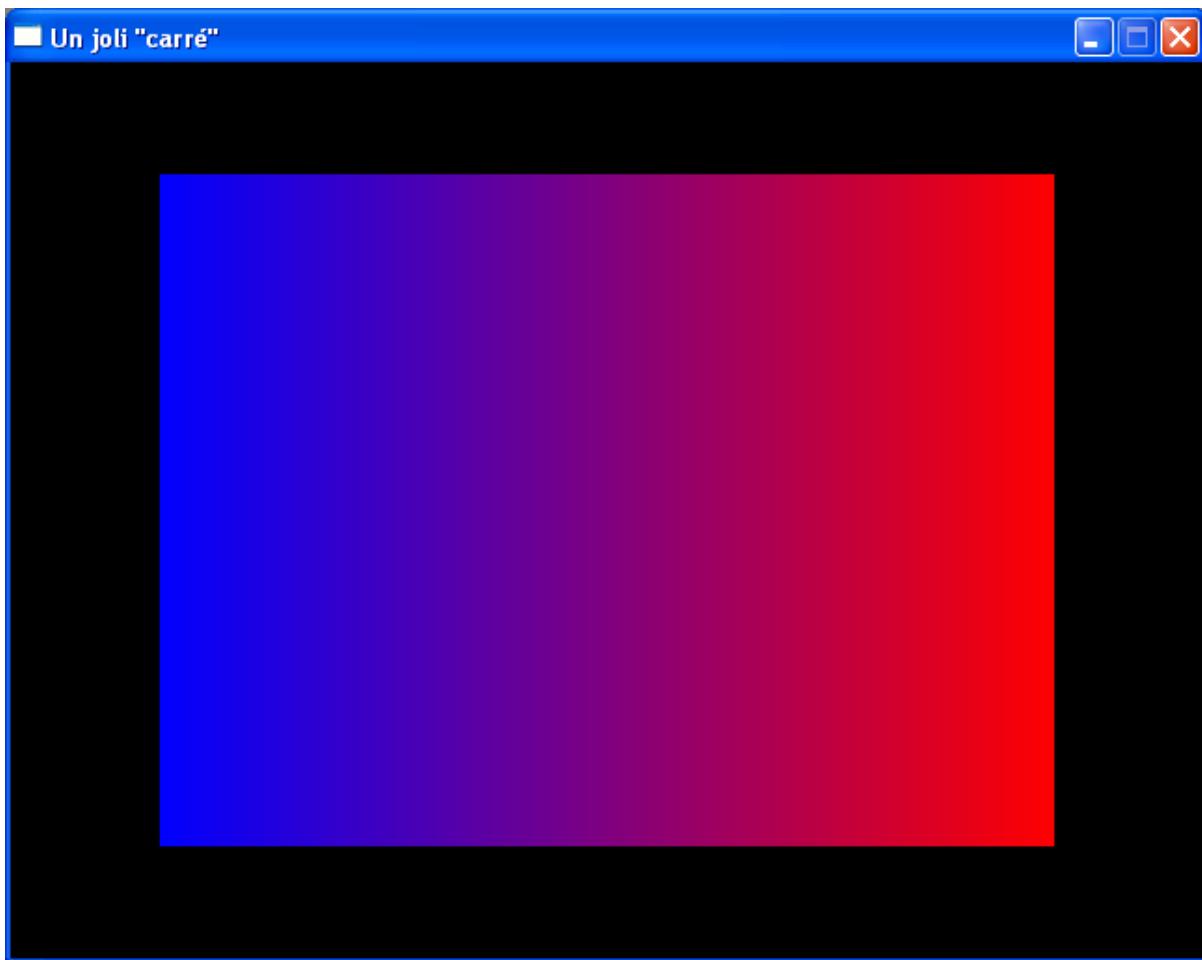
        glClear(GL_COLOR_BUFFER_BIT);

        glBegin(GL_QUADS);
        glColor3ub(0,0,255);
        glVertex2d(-0.75,-0.75);
        glVertex2d(-0.75,0.75);
        glColor3ub(255,0,0);
        glVertex2d(0.75,0.75);
        glVertex2d(0.75,-0.75);
        glEnd();

        glFlush();
        SDL_GL_SwapBuffers();
    }

    SDL_Quit();

    return 0;
}
```



Un dégradé très facilement avec OpenGL

Téléchargez le projet Code::Blocks, l'exécutable Windows et le Makefile Unix (116 Ko)
(http://sdz.tdct.org/sdz/medias/www.siteduzero.com_uploads_fr_ftp_kayl_sdlgl_02_carre.zip)

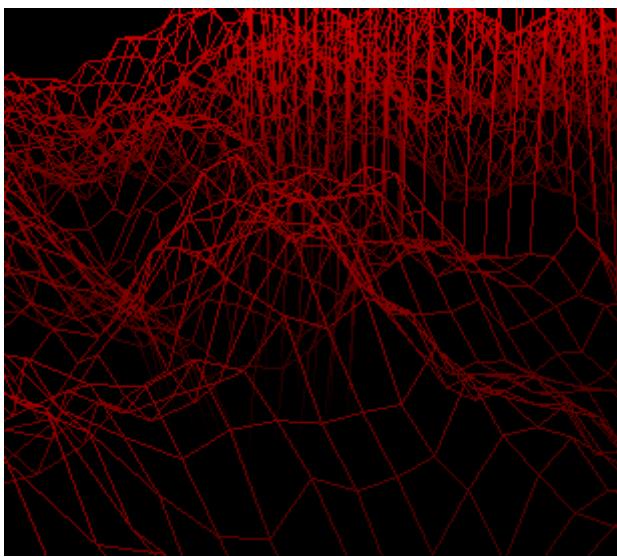
Euh oui mais ton carré n'est pas vraiment carré là !?

En effet par défaut les coordonnées vont de -1 à 1 à la fois sur X et sur Y quel que soit le ratio (rapport largeur/hauteur) de la fenêtre. Nous verrons plus tard comment tenir compte du ratio de la fenêtre et changer les coordonnées min/max.

Vous pouvez dès à présent vous amuser à dessiner des formes en 2D dans cet intervalle de coordonnées. Jouez avec les couleurs, les formes, les dégradés.

Vertices et polygones

En OpenGL tout est à base de polygones, le polygone de base le plus utilisé étant le triangle.
Même un model 3D se représente avec des polygones qui en composent les faces.



Terrain en mode filaire*

* Nous verrons dans la partie II de ce tuto comment générer un terrain en OpenGL.

Pour dessiner un polygone il faut :

- définir tous les sommets qui le composent ;
- indiquer à OpenGL comment il doit utiliser ses sommets.

Sommet se dit **vertex** en anglais, vertices au pluriel. Pour rester proches du langage OpenGL et du monde de la 3D, nous garderons l'appellation anglaise. Ça fait plus pro, isn't it ? :soleil:

Regardons la 1re ligne du code barbare du chapitre précédent :

```
glBegin(GL_TRIANGLES);
```

Avec GL_TRIANGLES on indique à OpenGL qu'il doit faire des triangles avec les vertices qui seront déclarés.

Voici la liste des modes que nous pouvons utiliser :

mode	détail
GL_POINTS	chaque vertex sera représenté comme un point
GL_LINES	les vertices seront reliés 2 à 2 pour faire des lignes. Si on définit 4 vertices, le 1er sera relié avec le 2nd, le 3e avec le 4e, mais en aucun cas le 2e ne sera relié au 3e
GL_LINE_STRIP	les vertices sont connectés par des lignes du 1er au dernier. Si on définit 3 vertices, le 1er sera relié avec le 2nd et le 2nd avec le 3e
GL_LINE_LOOP	identique à GL_LINE_STRIP, mais le dernier vertex est aussi relié au 1er (on réalise une boucle : loop)
GL_TRIANGLES	les triplets de vertices sont utilisés pour former des triangles pleins. Ce mode sera le plus utilisé lorsque nous représenterons des objets complexes à base de faces triangulaires

mode	détail
GL_TRIANGLE_STRIP	Les triangles se touchent, c'est-à-dire qu'il faut 3 vertices pour faire un premier triangle, et un 4e vertex suffit pour en définir un autre car les 2 derniers vertices seront aussi utilisés
GL_TRIANGLE_FAN	les triangles se touchent et ont en commun le 1er vertex défini. Si on définit 4 vertices, le 1er le 2e et le 3e forment un triangle, le 1er le 3e et le 4e forment un deuxième triangle.
GL_QUADS	les quadruplets de vertices forment des quadrilatères pleins
GL_QUAD_STRIP	tout comme précédemment, les quadrilatères sont connectés, il suffit de 2 nouveaux vertices pour en définir un nouveau, les 2 derniers vertices du quadrilatère précédents étant utilisés
GL_POLYGON	tous les vertices forment un polygone convexe (comme un pentagone ou tout autre polygone ayant plus de 4 sommets)

Une fois le mode de dessin défini, il faut déclarer les vertices avec glVertex :

```
glVertex2d(-0.75,-0.75);
glVertex2d(0,0.75);
glVertex2d(0.75,-0.75);
```

Nous l'avons plus haut, une même fonction OpenGL, ici glVertex, peut être appelée avec un nombre variable d'arguments. Ici nous en utilisons 2 (d'où le 2d) donc nous définissons le X et le Y du sommet en question et le Z est automatiquement mis à 0.

Il va sans dire que l'ordre de définition des vertices est important. Il faut suivre le contour du polygone si on ne veut pas se retrouver avec des aberrations.

Pour finir il faut fermer le bloc ouvert par glBegin avec glEnd :

```
glEnd();
```

Oublier de fermer un bloc ouvert avec glBegin par glEnd rendra le bloc invalide et rien ne sera dessiné.

Quelques options

Si vous êtes amenés à travailler avec des points ou des lignes, vous aurez sûrement envie d'en changer la taille... OpenGL a pensé à vous :

```
glPointSize( taille );
glLineWidth( largeur );
```

taille et largeur sont des réels qui valent 1.0 par défaut.

Comparatif

Pour comprendre l'influence du mode sur le rendu, voici une animation du rendu de 6 vertices avec des modes différents :



Certains modes n'ont pas été utilisés car ils n'avaient que peu d'intérêt dans cet exemple.

Dans le mode GL_QUADS, comme nous avons 6 vertices, les 4 premiers forment un rectangle, mais les 2 autres sont inutilisés.

Fonctions et types

Dans le chapitre précédent nous avons rencontré nos premières fonctions OpenGL.

Par exemple nous avions :

```
glBegin(GL_TRIANGLES);
	glColor3ub(255,0,0);    glVertex2d(-0.75,-0.75);
```

Pour pouvoir les différencier des autres fonctions, les fonctions OpenGL ont une syntaxe particulière dans leur nom :

glNom[NbType] ([Paramètres]);

J'imagine qu'une petite explication de la formule barbare du dessus n'est pas de refus. ;)

- **gl** ou **glu** : préfixes communs à toutes les fonctions OpenGL ;
- **Nom** : il s'agit du nom de la fonction comme Begin ou encore Vertex ;
- **Nb** : pour les fonctions à nombre de paramètres variables définit le nombre de paramètres qui vont suivre (ex : pour Begin aucun, pour Vertex 2) ;
- **Type** : pour les fonctions à type variable définit le type des paramètres utilisés. Nous utiliserons notamment :
 - i pour integer (entier) ;
 - f pour float (réel) ;
 - d pour double (réel plus précis) ;
 - ub pour unsigned byte (entier entre 0 et 255).
- **Paramètres** : pour finir si la fonction a besoin de paramètres il faut les rentrer.

Avec ce formalisme vous comprenez maintenant que glVertex2i est une fonction qui a besoin de deux paramètres de type entier.

Dans nos codes nous serons amenés à rencontrer des constantes OpenGL. On les reconnaît facilement car leur nom est intégralement en majuscules.

Ex : GL_TRUE, GL_FALSE qui sont les équivalents de true, et false (vrai et faux).

Si vous vous rappelez bien, nous sommes déjà passés devant une constante sans nous en rendre compte :

```
glClear(GL_COLOR_BUFFER_BIT);
```

Nous disions ici à OpenGL d'effacer le tampon d'affichage en passant en paramètre de glClear une constante qui lui permet de savoir quel tampon effacer.

À tout moment, n'hésitez pas à consulter la documentation pour avoir plus de détails sur l'utilisation d'une fonction.
La documentation est disponible sur le site d'OpenGL :

Documentation OpenGL (http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/)*

* par exemple pour avoir le détail sur `glVertex` nous cliquons sur `gl` (http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/) puis cherchons `Vertex` (http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/vertex.html).

Vous voyez l'OpenGL ce n'est pas si compliqué !

Dans le prochain chapitre nous restons en 2D pour nous attaquer aux transformations.

Les transformations

J'imagine votre impatience de débuter réellement la 3D. Mon but est de vous y emmener au plus vite mais avec un petit bagage technique nécessaire pour que vous puissiez tout de suite vous amuser et surtout comprendre ce que vous faites.

Ce chapitre vient introduire la notion de **transformation**, notion que nous pourrons dès à présent appliquer en 2D.

Si vous avez la tête qui tourne lors de la lecture de ce chapitre, c'est normal ! Ce sont les effets de la rotation que nous allons découvrir. ;)

Un brin de folie mathématique

Les transformations... un bien grand mot. En prenant notre bon vieux bouquin de Maths de 4e ou 3e nous trouvons comme transformations élémentaires :

- la translation ;
- la rotation ;
- la symétrie* ;
- le changement d'échelle.

* la symétrie centrale équivaut à une rotation de 180° et certaines symétries axiales sont définissables via des changements d'échelle (nous y reviendrons lorsque nous verrons comment réaliser un miroir en OpenGL)

Je ne vais pas m'étaler sur l'explication des actions de ces transformations sur les angles, les longueurs... c'était le boulot de votre prof de collège ! :diable:

Pour ma part je vais vous parler des matrices, car c'est sous cette forme que sont utilisées les transformations en OpenGL (et dans la géométrie spatiale en général).

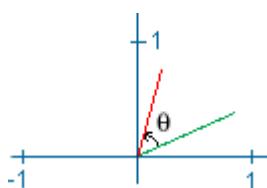
Les matrices vous connaissez ? Oui euh... non, pas *la la matrice* ok... c'est pas gagné... Image utilisateur

Une matrice se représente sous la forme d'un tableau de nombres... mais attention son sens et son utilisation vont au-delà de sa représentation. **Ce n'est pas qu'un « tableau ».**

D'une manière générale une matrice représente une transformation. La matrice définit la manière d'évoluer d'un système.

Ici ce qui nous intéresse c'est l'*utilisation géométrique des matrices* pour prédire **numériquement** le résultat d'une transformation.

Je prendrai l'exemple de la rotation 2D car il est facilement compréhensible et représentable. Le principe est le même pour la 3D, avec une dimension en plus bien évidemment.



La rotation d'angle θ sur le plan (voir dessin ci-dessus), s'écrit matriciellement :

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

En prenant le cas particulier de la rotation d'angle $\theta = 90^\circ$, la matrice devient :

$$M = \begin{vmatrix} 0 & -1 \\ 1 & 0 \end{vmatrix}$$

Maintenant prenons un vecteur de base, disons le vecteur V de coordonnées V=(1,1).

$$V = \begin{vmatrix} 1 \\ 1 \end{vmatrix}$$

Appliquons-lui la « transformation » décrite par la matrice. Pour ce faire calculons simplement le produit* matrice x vecteur : $V' = M \times V$

* Un produit matrice x vecteur est assez simple, le dessin ci-dessous l'explique en couleur.

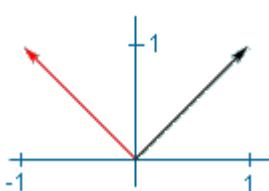
$$\left| \begin{array}{cc|c} & & \\ & & \\ x & & \end{array} \right| \quad \left| \begin{array}{c} 1 \\ 1 \end{array} \right|$$

$$\left| \begin{array}{cc|c} 0 & -1 & 0 + -1 \\ 1 & 0 & \end{array} \right|$$

On obtient comme vous le voyez le vecteur $V' = (-1, 1)$.

$$V' = \begin{vmatrix} -1 \\ 1 \end{vmatrix}$$

Et en effet en vérifiant graphiquement, le vecteur rouge est bien obtenu par rotation de 90° du vecteur noir comme nous l'avions prévu par le calcul.



Ici nous avons vu comment une matrice pouvait être utilisée pour réaliser une rotation.

En réalité les matrices utilisées sont des matrices 4x4 qui peuvent représenter à la fois une rotation, une translation et une « mise à l'échelle ». Je ne détaillerai pas ici les mathématiques des matrices 3D, mon but étant juste pour l'instant de vous faire comprendre qu'une matrice peut être (et sera) utilisée pour faire des transformations géométriques.

Les matrices OpenGL

La plupart du temps, avec OpenGL, nous ne manipulerons pas directement ces matrices mais vous devez savoir qu'il existe trois matrices que nous serons amenés à utiliser *via* des appels de fonctions simples :

GL_PROJECTION	dans laquelle nous définissons le mode de projection (orthogonale, perspective)
GL_MODELVIEW	pour positionner les objets dans la scène (caméra, vertices, lumières et autres effets). C'est celle que nous manipulerons le plus.
GL_TEXTURE	pour les textures. Nous verrons lors du chapitre sur les textures comment le fait de définir une translation grâce à cette matrice nous permettra de faire des textures animées.

Transformations cumulatives

Comme je vous l'ai dit tout à l'heure en OpenGL on modifie rarement la matrice directement (en affectant des valeurs). On vient plutôt modifier la matrice existante grâce à un appel de fonction. Par exemple si la transformation actuellement stockée dans la matrice est rotation de 90° et qu'on vient demander à faire une rotation de -10°, la matrice contiendra en fait la transformation cumulée c'est-à-dire rotation de 80°.

Pour éviter de cumuler des transformations et repartir à zéro en quelque sorte il faut réinitialiser la matrice (toute analogie avec un film est purement fortuite...).

Réinitialiser une matrice

Avant de faire des modifications sur la matrice de transformation, il faut être sûr de son état de départ. Pour cela on la réinitialise avec la **matrice d'identité***.

* La transformée d'un point par la matrice d'identité est lui-même, en gros elle ne transforme rien.

Pour ce faire on utilise la fonction :

`glLoadIdentity();`

Ainsi nous rajouterons ce morceau de code avant de faire un quelconque dessin :

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
```

Pile de matrices

Vous l'avez compris maintenant, appliquer une transformation en OpenGL revient à multiplier la matrice actuelle par la matrice de notre nouvelle transformation.

Avant d'effectuer une transformation, il faut savoir si nous allons l'appliquer uniquement à un objet, ou à tous ceux qui seront définis par la suite.

En effet si on applique une rotation à un triangle, et qu'on dessine un carré juste après il subira aussi cette transformation.

Deux fonctions permettent d'éviter ça :

- `glPushMatrix()` : sauvegarde la matrice actuelle ;
- `glPopMatrix()` : restitue la matrice sauvegardée.

Après un `glPushMatrix`, on continue à travailler avec l'état actuel de la matrice, on a juste rajouté la possibilité de revenir en arrière.

On peut toujours, à tout moment, recommencer à zéro et réinitialiser la matrice de transformation à la matrice identité : `glLoadIdentity();`

Quand une matrice est sauvegardée, elle est mise en tête de la pile (de sauvegarde) des matrices.

Un appel à `glPopMatrix` prend la matrice en tête, l'enlève de la pile et l'utilise comme matrice de transformation actuelle.

La profondeur de cette pile est de 32 matrices pour `GL_MODELVIEW`, on peut donc faire 32 appels consécutifs à `glPushMatrix`.

Maintenant que nous savons comment préserver notre matrice, voyons tout de même ce qui nous intéresse ici... la modifier pour appliquer des transformations.

Les transformations

Transformation = changement de repère.

Transformation = changement de repère.

Transformation = changement de repère.

Ah oui au fait, saviez-vous que : transformation = changement de repère ? :p

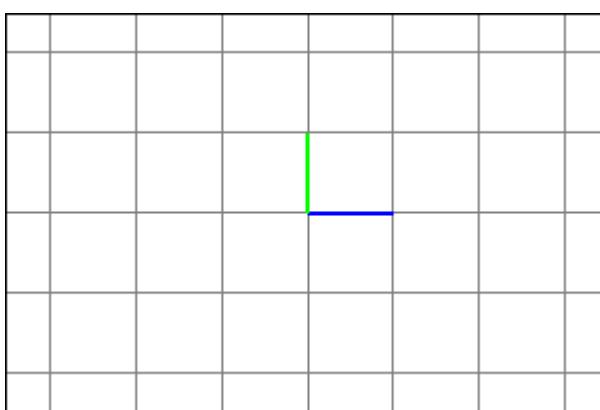
Je me permets d'insister (lourdement), car c'est ce qu'il faut assimiler pour devenir le roi des transformations.

Dans la vraie vie pour appliquer une transformation à un objet nous le placerions d'abord dans le monde puis nous le ferions tourner par exemple. Ici il faut réfléchir en terme de repère : par des modifications successives de la matrice GL_MODELVIEW nous plaçons, tournons, dimensionnons le repère dans lequel sera ensuite dessiné notre objet.

Ainsi nous appliquons d'abord les transformations que nous voulons, puis au dernier moment nous dessinons notre objet.

J'illustrerai donc les transformations sur la modification apportée au repère. Et pour marquer les esprits sur le fait que les transformations sont accumulées tant que la matrice n'est pas réinitialisée, j'effectuerai les transformations les unes à la suite des autres.

Partons donc du repère de base, ici représenté uniquement dans le plan (X,Y) par souci de simplicité.



Repère de base

Translation

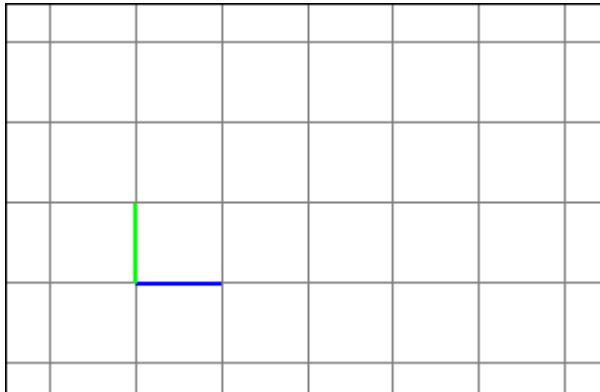
La translation permet de déplacer le repère actuel selon un vecteur $V = (x, y, z)$ où x, y , et z sont réels. Même si on ne veut déplacer que selon une composante, il faut définir les autres (en les mettant à 0).

`glTranslatef(x , y , z);`

Exemple ici :

```
glTranslated(-2,-1,0);
```

Donne :



Repère après translation

Rotation

La rotation fait tourner le repère actuel d'un angle θ (exprimé en degré) selon l'axe défini par le vecteur $V = (x, y, z)$ ou x, y , et z sont des réels :

`glRotated (theta, x , y, z);`

Généralement on ne fait tourner qu'autour **d'un des axes principaux (X Y ou Z) à la fois.**

Les rotations sur le plan (X, Y) se font autour de l'axe Z , donc pour faire tourner notre repère de 45° il faut faire :

```
glRotated(45,0,0,1);
```



Repère après rotation

Je vous l'ai dit tout à l'heure, chaque transformation modifie la matrice. Si on ne revient pas en arrière (remise à zéro, ou restitution d'une matrice sauvegardée), les transformations se combinent. C'est pour ça qu'ici le repère a tourné à partir de la position qui lui avait été donnée après la translation.

Changement d'échelle

Ce n'est pas vraiment une homothétie car on peut changer d'échelle différemment selon les axes.

Elle permet de transformer les axes du repère afin de grossir, diminuer, étirer les objets qui y seront dessinés (« scale » en anglais).

Ainsi si l'on appelle

`glScalef (i, j, k);`

le nouveau repère sera tel que $x' = i * x, y' = j * y, z' = k * z$.

Si l'on souhaite ne pas modifier un axe en particulier (par exemple Z quand on fait de la 2D) il faut mettre 1 et non 0.

Généralement on applique le même facteur à tous les axes pour ne pas déformer, mais rien ne nous empêche de transformer différemment les axes :

```
glScalef(2,0.5,1);
```



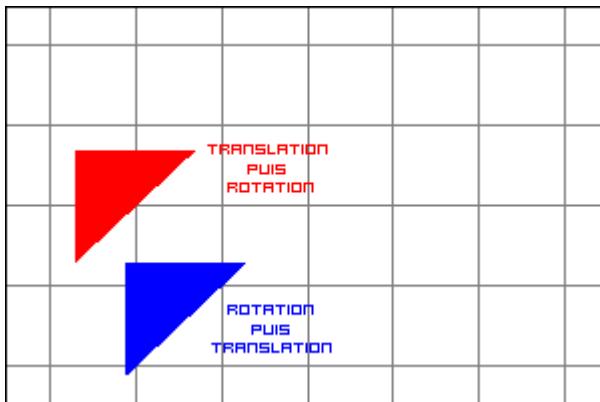
Repère après changement d'échelle

Importance de l'ordre des transformations

Il est important que vous réfléchissiez à l'ordre dans lequel vous appliquez vos transformations. Par exemple faire une translation suivie d'une rotation n'a pas forcément le même résultat que de faire la rotation puis la translation.

```
glPushMatrix();
glTranslated(-2,0,0);
glRotated(45,0,0,1);
dessin1(); //triangle rouge
glPopMatrix();

glRotated(45,0,0,1);
glTranslated(-2,0,0);
dessin2(); //triangle bleu
```



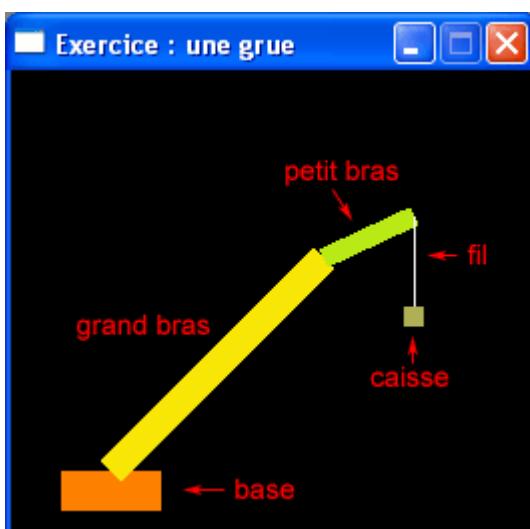
Ordre des transformations

Remarquez au passage l'utilisation qui a été faite de `glPushMatrix()` et `glPopMatrix()`, qui nous ont permis, après avoir dessiné le triangle rouge, de revenir au repère de base afin d'entamer les transformations pour le triangle bleu.

Pour être sûr que vous ayez bien saisi les finesse des transformations, rien de tel qu'un exercice pratique et rigolo !

Exercice : une grue

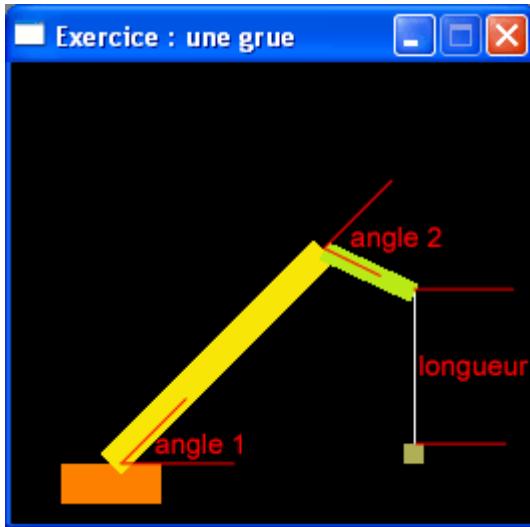
Je vous propose de vous familiariser avec les transformations par l'intermédiaire d'un petit exercice simple : la construction d'une grue 2D contrôlée par le clavier.



La grue est assez simple et est composée :

- d'une base ;

- d'un grand bras ;
- d'un petit bras ;
- d'un fil ;
- d'une caisse.



Je vous conseille de contrôler la grue au clavier (http://www.siteduzero.com/tuto-3-5790-1-la-gestion-des-evenements-clavier-et-souris.html#ss_part_2) et ainsi de pouvoir modifier :

- l'angle entre le bras et la base ;
- l'angle entre le petit bras et le grand bras ;
- la longueur du fil (pour faire monter et descendre la caisse).

Gestion du clavier

Je vous laisse libres du choix des touches. Pour ma part j'ai utilisé les flèches directionnelles : haut/bas pour le fil, gauche/droite pour les bras (shift enfoncé pour le grand bras).

Une réception des événements avec `SDL_WaitEvent` suffit car on veut ne faire bouger la grue que lors de l'appui sur une touche.

Pensez toutefois à activer la répétition des touches au préalable* :

```
SDL_EnableKeyRepeat(10,10);
```

* les valeurs proposées par `SDL` (`SDL_DEFAULT_REPEAT_DELAY` et `SDL_DEFAULT_REPEAT_INTERVAL`) sont trop lentes. Avec 10, 10 vous aurez un mouvement plus fluide.

Pour avoir quelque chose d'un chouilla réaliste, je vous conseille de limiter la plage de valeurs que peuvent prendre vos variables (angles et longueur). J'utilise :

- angle grand bras/base entre 10° et 90° ;
- angle petit bras/grand bras entre -90° et 90° ;
- longueur entre 10 et 100.

Dessin de la grue

Nous restons encore en 2D mais pour faciliter les choses il serait bien de pouvoir avoir des coordonnées de l'ordre des pixels. Pour ce faire nous allons modifier la matrice de projection pour faire de la projection 2D dont nous spécifierons cette fois les dimensions (alors qu'elles étaient par défaut entre -1 et 1 au préalable).

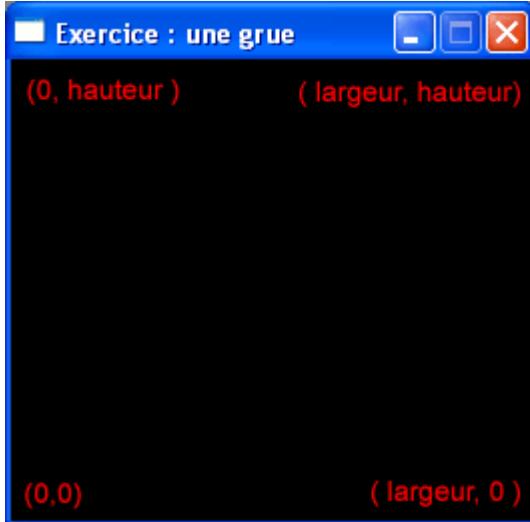
```

SDL_WM_SetCaption("Exercice : une grue", NULL);
SDL_SetVideoMode(LARGEUR_ECRAN, HAUTEUR_ECRAN, 32, SDL_OPENGL);

glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluOrtho2D(0,LARGEUR_ECRAN,0,HAUTEUR_ECRAN);

```

De cette manière nous aurons l'espace de coordonnées suivant :



Enfin dernier conseil avant de vous lâcher dans la nature, il peut être utile à tout moment de savoir où est le repère actuel et comment il est orienté. Pour ce faire voici une petite fonction que vous pouvez appeler n'importe quand dans votre dessin pour « déboguer » et mieux visualiser vos transformations.

```

/*
 Dessine le repère actuel pour faciliter
 la compréhension des transformations.
 Utiliser « echelle » pour avoir un repère bien orienté et positionné
 mais avec une échelle différente.

*/
void dessinerRepere(unsigned int echelle = 1)
{
    glPushMatrix();
    glScalef(echelle,echelle,echelle);
    glBegin(GL_LINES);
    glColor3ub(0,0,255);
    glVertex2i(0,0);
    glVertex2i(1,0);
    glColor3ub(0,255,0);
    glVertex2i(0,0);
    glVertex2i(0,1);
    glEnd();
    glPopMatrix();
}

```

Par exemple, après avoir dessiné ma base si j'appelle

```
dessinerRepere(50);
```

j'obtiens :



Je vois donc ici que je suis prêt à faire la première rotation et dessiner le grand bras.

En ce qui concerne le fil, il doit toujours être à la verticale. Il faut donc que vous trouviez, une fois au bout du petit bras, un moyen d'annuler les rotations pour remettre le repère « à l'endroit ».

Bon courage !

Correction

Je vous donne ma version de la grue. Ce n'est qu'un guide rien ne vous oblige à faire « exactement » pareil.

```
#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <cstdlib>

#define LARGEUR_BASE 50
#define HAUTEUR_BASE 20

#define LARGEUR_BRAS_1 150
#define HAUTEUR_BRAS_1 15

#define LARGEUR_BRAS_2 50
#define HAUTEUR_BRAS_2 10

#define TAILLE_CAISS 10

#define LARGEUR_ECRAN (LARGEUR_BASE + LARGEUR_BRAS_1 + HAUTEUR_BRAS_2 + 50)
#define HAUTEUR_ECRAN (HAUTEUR_BASE + LARGEUR_BRAS_1 + HAUTEUR_BRAS_2 + 50)

int angle1 = 45;
int angle2 = -20;
int longueur = 50;

void Dessiner();

int main(int argc, char *argv[])
{
    SDL_Event event;

    SDL_Init(SDL_INIT_VIDEO);
    atexit(SDL_Quit);

    SDL_WM_SetCaption("Exercice : une grue", NULL);
    SDL_SetVideoMode(LARGEUR_ECRAN, HAUTEUR_ECRAN, 32, SDL_OPENGL);

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity( );
    gluOrtho2D(0,LARGEUR_ECRAN,0,HAUTEUR_ECRAN);

    SDL_EnableKeyRepeat(10,10);

    Dessiner();

    while(SDL_WaitEvent(&event))
    {
        switch(event.type)
        {
            case SDL_QUIT:
                exit(0);
                break;
            case SDL_KEYDOWN:
                switch (event.key.keysym.sym)
                {
```

```

        case SDLK_UP:
            longueur --;
            if (longueur < 10)
                longueur = 10;
            break;
        case SDLK_DOWN:
            longueur++;
            if (longueur > 100)
                longueur = 100;
            break;
        case SDLK_LEFT:
            if ((event.key.keysym.mod & KMOD_LSHIFT) == KMOD_LSHIFT)
            {
                angle1++;
                if (angle1 > 90)
                    angle1 = 90;
            }
            else
            {
                angle2++;
                if (angle2 > 90)
                    angle2 = 90;
            }
            break;
        case SDLK_RIGHT:
            if ((event.key.keysym.mod & KMOD_LSHIFT) == KMOD_LSHIFT)
            {
                angle1--;
                if (angle1 < 10)
                    angle1 = 10;
            }
            else
            {
                angle2--;
                if (angle2 < -90)
                    angle2 = -90;
            }
            break;
        }
        Dessiner();
    }

    return 0;
}

/*
Dessine un rectangle avec comme point de référence
le milieu du côté gauche
*/
void dessineRectangle(double largeur,double hauteur)

```

```

{
    glBegin(GL_QUADS);
    glVertex2d(0,-hauteur/2);
    glVertex2d(0,hauteur/2);
    glVertex2d(largeur,hauteur/2);
    glVertex2d(largeur,-hauteur/2);
    glEnd();
}

/*
Dessine le repère actuel pour faciliter
la compréhension des transformations.
Utiliser "echelle" pour avoir un repère bien orienté et positionné
mais avec une échelle différente.
*/
void dessinerRepere(unsigned int echelle = 1)
{
    glPushMatrix();
    glScalef(echelle,echelle,echelle);
    glBegin(GL_LINES);
    glColor3ub(0,0,255);
    glVertex2i(0,0);
    glVertex2i(1,0);
    glColor3ub(0,255,0);
    glVertex2i(0,0);
    glVertex2i(0,1);
    glEnd();
    glPopMatrix();
}

void Dessiner()
{
    glClear( GL_COLOR_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );

    /* Je déplace mon répère initial (actuellement
    en bas à gauche de l'écran) */
    glTranslated(LARGEUR_BASE/2,HAUTEUR_BASE,0);

    // La base
    glColor3ub(254,128,1);
    dessineRectangle(LARGEUR_BASE,HAUTEUR_BASE);

    //Je me place en haut au milieu de la base
    glTranslated(LARGEUR_BASE/2,HAUTEUR_BASE/2,0);

    // Le grand bras
    glRotated(angle1,0,0,1);
    glColor3ub(248,230,7);
    dessineRectangle(LARGEUR_BRAS_1,HAUTEUR_BRAS_1);

    // Je me place au bout du grand bras
}

```

```

glTranslated(LARGEUR_BRAS_1,0,0);

// Puis m'occupe du petit bras
glRotated(angle2,0,0,1);
glColor3ub(186,234,21);
dessineRectangle(LARGEUR_BRAS_2,HAUTEUR_BRAS_2);

// Je me place au bout du petit bras
glTranslated(LARGEUR_BRAS_2,0,0);
/* J'annule les rotations pour avoir mon repère aligné
avec le repère d'origine */
glRotated(-angle1-angle2,0,0,1);

// Je dessine le fil
glColor3ub(255,255,255);
glBegin(GL_LINES);
glVertex2i(0,0);
glVertex2i(0,-longueur);
glEnd();

/* Je descends en bas du fil (avec un petit décalage
sur X pour anticiper le dessin de la caisse */
glTranslated(-TAILLE_CAISSSE/2,-longueur,0);

// Et je dessine enfin la caisse
glColor3ub(175,175,85);
dessineRectangle(TAILLE_CAISSSE,TAILLE_CAISSSE);

glFlush();
SDL_GL_SwapBuffers();
}

```

Téléchargez le projet Code::Blocks, l'exécutable Windows et le Makefile Unix (118 Ko) (http://sdz.tdct.org/sdz/médias/www.siteduzero.com_uploads_fr_ftp_kayl_sdlgl_03_grue.zip)

Voilà, si vous avez fait l'exercice proposé vous êtes maintenant les rois des transformations OpenGL. Nous allons enfin pouvoir nous attaquer à ce que nous attendons tant : **la 3D**. Direction le prochain chapitre pour voir comment paramétrier et utiliser la caméra.

Enfin de la 3D (Partie 1/2)

Enfin on y est ! On va pouvoir passer à la 3D !

Pour faire ce passage en douceur je scinde ce chapitre en deux parties.

Nous commencerons donc par *comprendre comment il est possible de faire de la 3D* sur un écran 2D, ce qu'il nous faut préparer pour « dessiner » en 3D, et dans la seconde partie *nous dessinerons un cube* et verrons les problèmes que cela pose.

Du réel à l'écran

Quand vous décrivez votre scène 3D en OpenGL (à coup de glVertex), vous décrivez le monde tel qu'il est dans l'absolu, dans son propre repère.

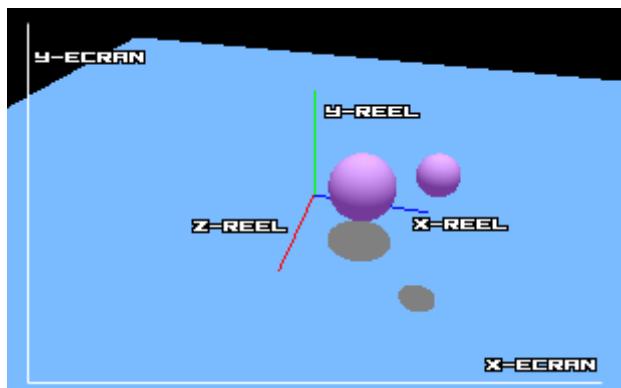
Pour passer du monde réel à l'écran, il faut donner quelques indications à OpenGL :

- définir quelle zone de la fenêtre servira pour le rendu* ;

- définir le mode de projection (perspective par exemple) ;
- placer la caméra dans le monde réel.

* Cette phase est automatiquement réalisée à la création de la fenêtre. Nous verrons dans la partie II comment la changer pour faire du split-screen (plusieurs écrans de rendu sur la même fenêtre).

Grâce à ces informations, OpenGL pourra déterminer les transformations à faire subir aux objets du monde réel à 3 dimensions pour les dessiner sur l'écran à 2 dimensions.



Dans le monde réel on travaille au niveau des vertices, et sur l'écran au niveau des pixels. Sur le dessin du haut vous avez peut-être remarqué que je n'ai pas fait apparaître de Z écran. C'était juste pour ne pas vous embrouiller car l'écran est bien à 2 dimensions. Mais le Z écran sert tout de même à donner une information de profondeur des pixels.



Le repère X,Y,Z de l'écran n'est pas direct : si on applique la règle de la main droite. L'axe Z devrait aller de l'écran vers nous. Cependant la convention a été prise en synthèse d'image que l'axe Z s'enfonce dans l'écran.

Quelle est l'unité de distance en OpenGL ?

Réponse : celle que vous voulez ! En effet quand nous y pensons qu'est-ce qui fait que quelque chose est grand ou petit en réalité ? Sans référentiel il nous est difficile de déterminer visuellement la taille d'un objet. Nous savons qu'il est petit car il y a un autre objet à côté dont nous connaissons la taille. Nous savons comparer.

Comment savoir que quelque chose est loin ? Parce que quand nous bougeons, sa taille ne varie pas beaucoup et parce que nous mettons du temps à nous en approcher.

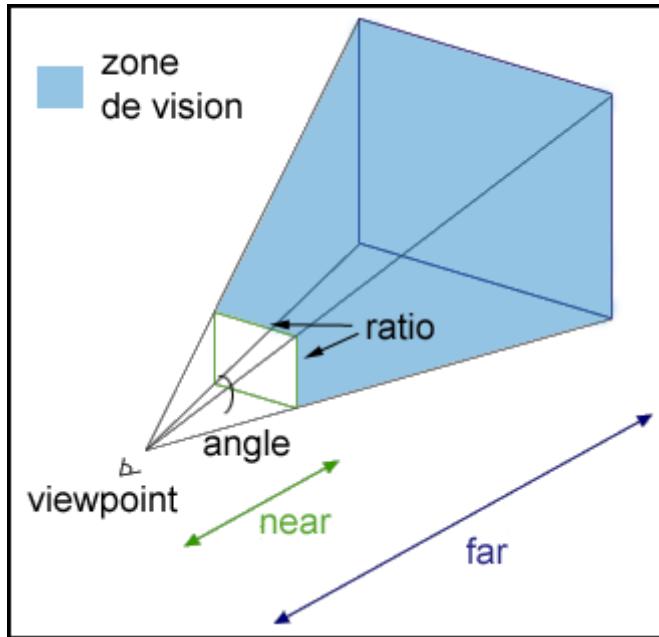
Ainsi en OpenGL se dire que « 1 » est 1 mètre comme 1 millimètre ou comme 1 kilomètre revient au même du moment que les proportions sont gardées et que les vitesses des mouvements sont adaptées. En effet si on fait face à un cube de « 1 » de largeur et qu'appuyer une fois sur la touche « avancer » nous fait bouger de 10000 on se dira : soit je vais très vite, soit l'objet était (car on vient de le perdre) très très petit en fait.

Cependant il est parfois utile de se rapporter à des unités connues. Si 1 unité OpenGL est 1 mètre, alors il sera facile d'utiliser un moteur physique qui utilise des vraies unités.

La perspective

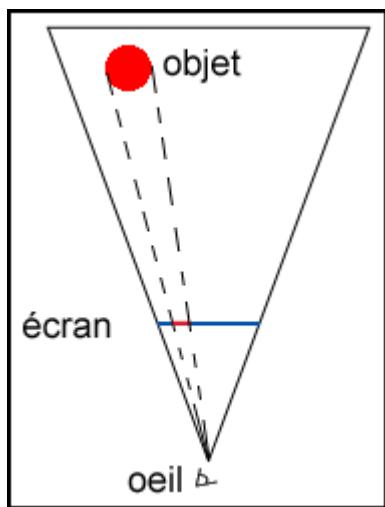
Le passage le plus important dans la 3D est la **projection**. Pour passer d'un monde décrit en 3D à une fenêtre avec des pixels 2D il nous faut perdre une dimension et donc projeter.

La méthode de projection que nous utiliserons en 3D est la perspective. La perspective est définie par la pyramide ci-dessous :



pyramide de clipping

Cette pyramide s'appelle la **pyramide de clipping**. C'est-à-dire que tout objet ne se trouvant pas à l'intérieur de la zone bleue ne sera pas dessiné. De plus sa forme permet de définir comment projeter les objets sur l'écran en faisant un parallèle avec la pyramide réelle qui a pour sommet l'**œil** de l'utilisateur et coupe son **écran** :



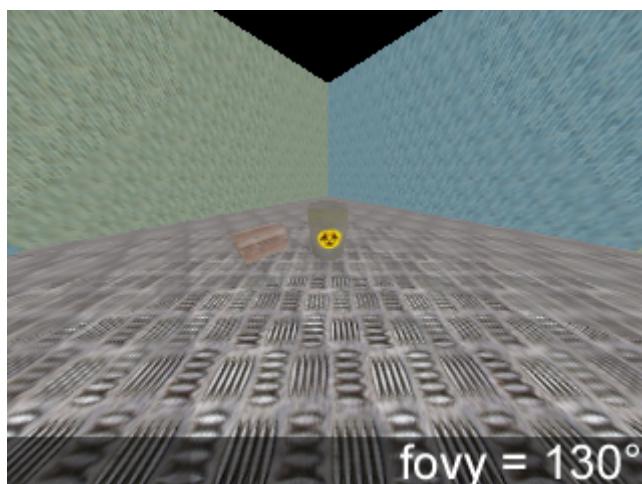
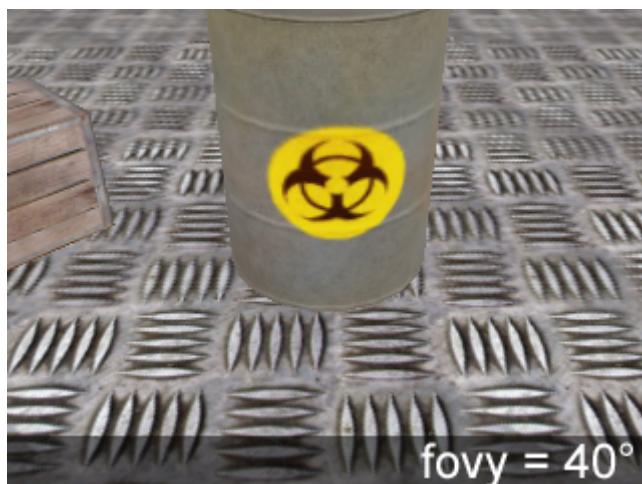
pyramide réelle (ici vue du dessus)

Le **ratio** (vu dans le schéma plus haut) est un rapport entre la largeur et la hauteur. Pour une télévision ce ratio vaut 4/3, des fois 16/9. Pour les tailles informatiques standards (1024x768, 800x600, 640x480) il est aussi de 4/3. Mais votre fenêtre n'est pas obligée d'utiliser tout l'écran et peut donc avoir ses propres proportions et il existe aussi de plus en plus d'écrans avec des ratios spéciaux. Il est donc important de ne pas le prendre comme acquis dans vos applications et toujours le définir en fonction de la taille de la fenêtre que vous créez (je ferai une annexe SDL sur comment détecter les modes disponibles pour le plein écran).

Les paramètres **near** et **far** déterminent les distances minimales et maximales des objets. En dehors de cet intervalle les objets ne seront pas affichés. Cela posera parfois problème si la valeur de **far** est trop petite, l'utilisateur risque de voir disparaître des objets quand il s'éloigne, ou pire voir la scène apparaître subitement. Nous verrons dans la partie II comment utiliser le **brouillard** pour parer à ce problème. Pour l'instant une valeur grande (vis-à-vis de la taille de votre scène) suffira amplement.

Le seul paramètre qui peut vous échapper ici est l'**angle**. Il s'agit de l'angle de vision entre les plans haut et bas de la pyramide (souvent dénoté **fovy** pour *field of view* sur l'axe y). Généralement on utilise une valeur aux alentours de 70°. Lors de la sortie d' Half-Life 2 il y a eu toute une polémique sur l'angle de vue mal choisi (90°) qui rendait certaines personnes mal à l'aise.

Pour voir l'influence de cet angle, je vous ai fait 3 rendus d'une même scène avec des angles différents :



On constate que si l'on donne un *angle petit*, ça donne un effet de *zoom*. Et je ne vous cacherai pas que c'est **exactement** ce qui est utilisé dans les jeux pour faire un zoom (jumelles ou sniper).

Application en OpenGL

Pour définir la perspective comme mode de projection il suffit d'appeler la fonction

```
gluPerspective( fovy, ratio, near, far );
```

Par exemple pour une fenêtre de 640x480 :

```
gluPerspective(70,(double)640/480,1,1000);
```

Cet appel modifie la matrice courante, même s'il ne s'agit pas de la matrice de projection. Il faut donc bien veiller à sélectionner la matrice GL_PROJECTION, l'initialiser et ensuite appeler gluPerspective.

Voici donc notre nouveau code d'initialisation de notre application OpenGL :

```
SDL_Init(SDL_INIT_VIDEO);
SDL_WM_SetCaption("SDL GL Application", NULL);
SDL_SetVideoMode(640, 480, 32, SDL_OPENGL);
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective(70,(double)640/480,1,1000);
```

Comme la définition de la perspective ne concerne que la matrice GL_PROJECTION nous pouvons nous contenter de l'initialiser une seule fois et non pas à chaque image. Bien sûr si le ratio de la fenêtre venait à changer dynamiquement il faudrait redéfinir la perspective au risque de voir tous les objets déformés.

Placer la caméra

Maintenant que nous savons comment projeter, il serait bien de pouvoir placer le point de vue n'importe où dans la scène. Pour cela nous utilisons une sorte de **caméra** virtuelle avec l'appel de :

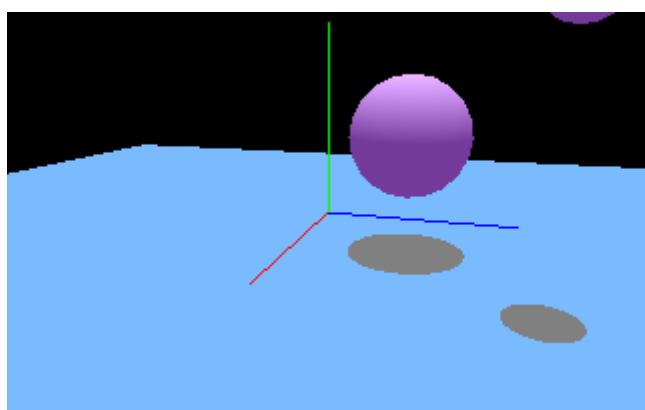
```
gluLookAt( camX, camY, camZ, cibleX, cibleY, cibleZ, vertX, vertY, vertZ );
```

camX, **camY** et **camZ** définissent la position de la caméra ;

cibleX, **cibleY** et **cibleZ** définissent la position du point que fixe la caméra (le point correspondant se trouvera au centre de la fenêtre d'affichage) ;

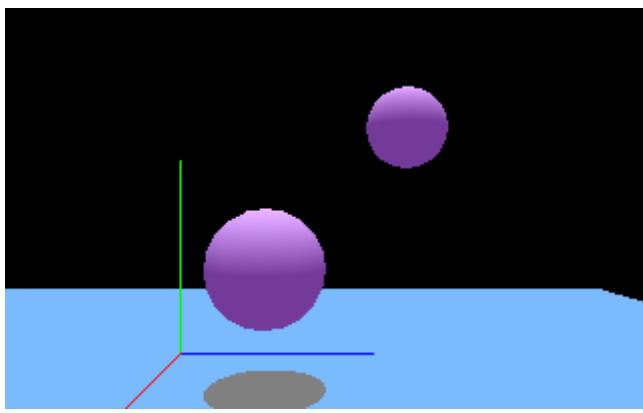
vertX, **vertY** et **vertZ** définissent le vecteur vertical.

Voici 3 images prises avec des paramètres différents de la caméra pour que vous compreniez l'utilité de chaque paramètre :



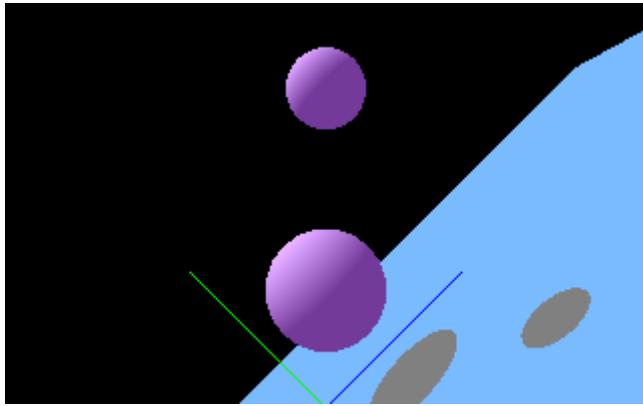
```
gluLookAt(1.5, 1.5, 5, 0, 0, 0, 0, 1, 0);
```

La caméra est en (1.5,1.5,5) et regarde en (0,0,0).



```
gluLookAt(1.5, 1.5, 5, 1.5, 1.5, 0, 0, 1, 0);
```

La caméra est en (1.5,1.5,5) et regarde en (1.5,1.5,0) c'est-à-dire droit devant elle (contrairement à l'image précédente).



```
gluLookAt(1.5, 1.5, 5, 0, 0, 0, 1, 1, 0);
```

Même position et regard que précédemment mais son vecteur vertical est (1,1,0), or la scène a été pensée avec une verticale de (0,1,0), l'image est donc « penchée » sur le côté.

L'importance de la verticale

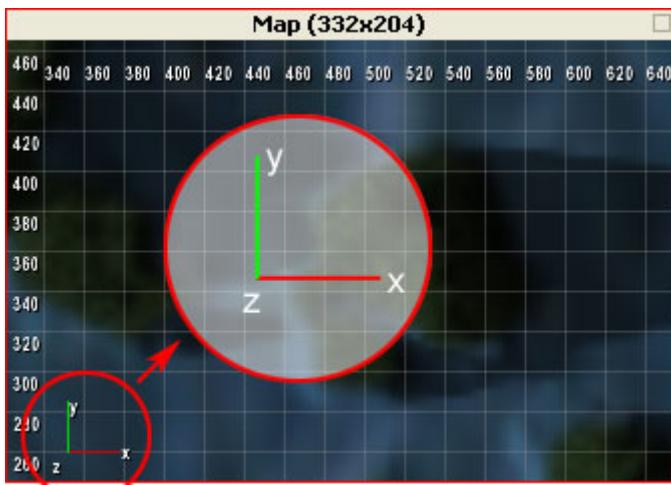
Vous l'avez vu, l'appel à `gluLookAt` vous permet de définir la verticale que vous voulez pour votre scène. Le tout est d'être cohérent et de concevoir votre scène en conséquence.

Il est néanmoins **beaucoup plus pratique** de choisir comme verticale un vecteur du repère de base X (1,0,0), Y (0,1,0) ou Z (0,0,1).

Entre Y et Z lequel est le meilleur choix ?

Bonne question. L'un ou l'autre sont des choix valables : on peut imaginer le passage à la 3e dimension comme l'ajout de la **profondeur** tout comme l'ajout de la **hauteur**. Dans le monde de la synthèse d'image il n'est pas rare de voir l'un ou l'autre.

À l'avenir, j'essayerai de respecter le choix **verticale = Z** car il a été fait dans de nombreux jeux et OpenGL semble l'avoir favorisé (nous le verrons lors du chapitre sur les **quadriques**...).



Éditeur de FarCry, Z est la verticale

Quand l'appeler ?

Placer la caméra revient à déplacer tout le monde pour qu'il soit centré sur la caméra et orienté selon l'axe du regard. Cela influe donc logiquement sur la matrice GL_MODELVIEW et vous ne serez pas étonnés donc que l'appel de la caméra soit **juste après** la réinitialisation de GL_MODELVIEW.

```
void Dessiner()
{
    glClear( GL_COLOR_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    gluLookAt(3,3,3,0,0,0,0,0,1); //exemple

    /* Dessin 3D ici */

    glFlush();
    SDL_GL_SwapBuffers();
}
```

Voilà vous savez maintenant comment il est possible de passer d'un monde 3D à un écran 2D et savez placer la caméra.

Pour récapituler, notre nouveau squelette de programme pour dessiner de la 3D :

```

#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <cstdlib>

void Dessiner();

int main(int argc, char *argv[])
{
    SDL_Event event;

    SDL_Init(SDL_INIT_VIDEO);
    atexit(SDL_Quit);
    SDL_WM_SetCaption("SDL GL Application", NULL);
    SDL_SetVideoMode(640, 480, 32, SDL_OPENGL);

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective(70,(double)640/480,1,1000);

    Dessiner();

    for (;;)
    {
        SDL_WaitEvent(&event);

        switch(event.type)
        {
            case SDL_QUIT:
                exit(0);
                break;
        }
        Dessiner();
    }

    return 0;
}

void Dessiner()
{
    glClear( GL_COLOR_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );

    gluLookAt(3,3,3,0,0,0,0,0,1);

    /* Dessin 3D */

    glFlush();
    SDL_GL_SwapBuffers();
}

```

Maintenant il ne reste plus qu'à remplir la partie dessin. Et pour ça direction le chapitre suivant avec notre premier dessin 3D : **un cube** !

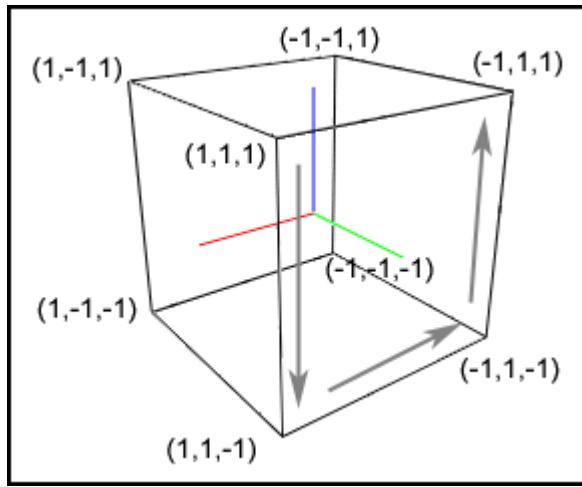
Enfin de la 3D (Partie 2/2)

Dans le dernier chapitre nous nous sommes préparés à passer à la 3D, il est donc temps de s'y mettre et d'entamer notre premier dessin : un cube ! En dessinant notre cube et en l'animant, nous rencontrerons des problèmes, prévus d'avance rassurez-vous ^^ , et nous verrons comment les résoudre.

Un cube

L'exemple du cube est assez simple et nous continuerons avec lors de la création de notre première scène texturée.

Voyons tout d'abord comment est constitué un cube :



Coordonnées des sommets du cube

Un cube est composé de 8 sommets et 6 faces, chaque face faisant intervenir 4 sommets.

Nous n'allons pas nous contenter de dessiner chacun des 8 sommets, nous n'aurions pas de faces pleines. Il nous faut donc décrire les faces une par une, en indiquant les sommets qu'elles font intervenir.

Décrire des sommets en 3D

Ici nous devons définir 3 coordonnées pour chaque sommet : X, Y et Z. Nous ne pouvons donc plus utiliser le basique `glVertex2d` que nous utilisions auparavant. Il va falloir donc utiliser la version avec 3 arguments soit `glVertex3d`.

Exemple :

```
glVertex3d(1,1,1);
```

Le 3d de `glVertex3d` ne veut pas dire « *ouais je fais de la 3D !* ». Rappelez-vous le chapitre Notions de base (http://www.siteduzero.com/tuto-3-6094-1-notions-de-base.html#ss_part_1), 3 spécifie le nombre d'arguments et d le fait que les arguments donnés soient des double (réels).

Décrire le cube

Les faces étant des carrés, nous allons utiliser le mode `GL_QUADS` pour décrire les vertices.

Pour différencier les faces nous leur attribuerons une couleur différente ; nous ferons la première (celle avec les flèches) en rouge. En ce qui concerne la caméra, j'ai choisi de la placer en (3,4,2) pour regarder le cube centré en (0,0,0), car cela donnera un bon angle de vue (c'est la position utilisée pour le schéma plus haut).

Ce qui donne donc :

```

void Dessiner()
{
    glClear( GL_COLOR_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    gluLookAt(3,4,2,0,0,0,0,0,1);

    glBegin(GL_QUADS);

    glColor3ub(255,0,0); //face rouge
    glVertex3d(1,1,1);
    glVertex3d(1,1,-1);
    glVertex3d(-1,1,-1);
    glVertex3d(-1,1,1);
    glEnd();

    glFlush();
    SDL_GL_SwapBuffers();
}

```

Facile ! Il suffit de choisir un point de départ et de suivre le contour de la face pour décrire les sommets un par un.

Je vous conseille, pour les décrire, de vous imaginer faisant face à la face (hum... Image utilisateur) et de les énumérer un à un dans le sens inverse des aiguilles d'une montre. En effet nous verrons plus tard que cela nous sera utile quand nous voudrons éviter de dessiner les faces cachées. Encore une fois quelque soit le sens que vous décidez d'utiliser, le tout est d'être **cohérents** et de garder le même sens.

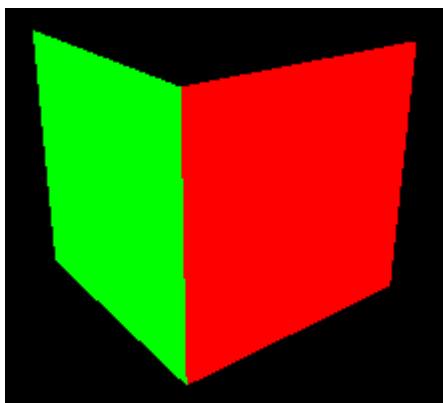
Continuons donc avec la 2e face, disons celle à gauche de la première (quand on regarde le schéma) et faisons-la en vert.

```

glColor3ub(0,255,0); //face verte
glVertex3d(1,-1,1);
glVertex3d(1,-1,-1);
glVertex3d(1,1,-1);
glVertex3d(1,1,1);

```

Ce qui me donne le résultat suivant :

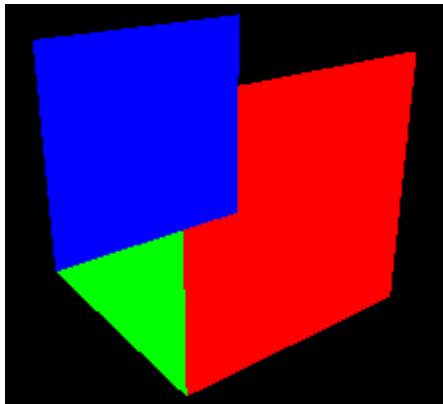


Bon maintenant parce que je veux vous montrer un problème important, attaquons-nous à la face de derrière que nous ferons... devinez... en bleu ! ;)

Rien de compliqué, il suffit de suivre le schéma de tout à l'heure pour avoir rapidement les coordonnées et écrire le code approprié.

```
glColor3ub(0,0,255); //face bleue  
glVertex3d(-1,-1,1);  
glVertex3d(-1,-1,-1);  
glVertex3d(1,-1,-1);  
glVertex3d(1,-1,1);
```

Et voilà le résultat :



o_O o_O o_O

En effet vous ne rêvez pas, la face bleue qui était censée être derrière, donc en majeure partie cachée par la rouge et la verte vient se dessiner par-dessus ces dernières.

Et c'est tout à fait logique, OpenGL dessine les carrés *dans l'ordre dans lequel on les définit*. Il ne se soucie pour l'instant pas de savoir s'il y a déjà quelque chose là où il dessine et vient donc écraser les faces précédentes.

La solution ? **Le Z-Buffer** !

Le Z-Buffer

Le Z-Buffer ou **Depth-Buffer** (pour tampon de profondeur) sert à éviter le problème que nous venons de rencontrer.

Principe du Z-Buffer

Le Z-Buffer est un tampon (buffer) qui stocke la profondeur (d'où le Z, X et Y sur l'écran étant la position en pixel) de chaque pixel affiché à l'écran.

Ensuite quand OpenGL demande à dessiner un pixel à un endroit, il compare la profondeur du point à afficher et celle présente dans le buffer. Si le nouveau pixel est situé devant l'ancien, alors il est dessiné et la valeur de la profondeur dans le buffer est mise à jour. Sinon, le pixel était alors **situé derrière** et n'a donc **pas** lieu d'être **affiché**.

Pour bien comprendre, suivons le cheminement qui est fait.

- | | | |
|---|---|---|
| ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ |

Au départ le buffer (ici de taille 3x3 pour l'exemple) est initialisé à des distances infinies (le plus loin possible vers le fond de votre écran).

- | | | |
|---|---|---|
| ∞ | ∞ | ∞ |
| ∞ | 5 | ∞ |
| ∞ | ∞ | ∞ |

OpenGL demande à dessiner un pixel en (2,2,5).

Valeur demandée : 5.

Valeur présente : infini.

5 < infini => OK pour dessin

Dessin du pixel + Mise à jour du Z-buffer avec la valeur 5.

- Demande de dessin d'un pixel en (2,2,10).

Valeur demandée : 10.

Valeur présente : 5.

10 > 5 => Dessin refusé

Application dans OpenGL

Heureusement pour nous *OpenGL gère très bien cette technique*, il nous faut juste modifier notre programme pour l'activer et bien l'utiliser !

Pour cela il nous faut :

- activer son utilisation : après la création de la fenêtre OpenGL il faut simplement appeler :

```
glEnable(GL_DEPTH_TEST);
```

- le réinitialiser à chaque nouvelle image, en même temps que le buffer des pixels :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) ;
```

Ce qui nous donne un code complet (avec le début de notre cube) :

```

#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <cstdlib>

void Dessiner();

int main(int argc, char *argv[])
{
    SDL_Event event;

    SDL_Init(SDL_INIT_VIDEO);
    atexit(SDL_Quit);
    SDL_WM_SetCaption("SDL GL Application", NULL);
    SDL_SetVideoMode(640, 480, 32, SDL_OPENGL);

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective(70,(double)640/480,1,1000);
    glEnable(GL_DEPTH_TEST);

    Dessiner();

    for (;;)
    {
        SDL_WaitEvent(&event);

        switch(event.type)
        {
            case SDL_QUIT:
                exit(0);
                break;
        }
        Dessiner();
    }

    return 0;
}

void Dessiner()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );

    gluLookAt(3,4,2,0,0,0,0,0,1);

    glBegin(GL_QUADS);

    glColor3ub(255,0,0); //face rouge
    glVertex3d(1,1,1);

```

```

glVertex3d(1,1,-1);
glVertex3d(-1,1,-1);
glVertex3d(-1,1,1);

glColor3ub(0,255,0); //face verte
glVertex3d(1,-1,1);
glVertex3d(1,-1,-1);
glVertex3d(1,1,-1);
glVertex3d(1,1,1);

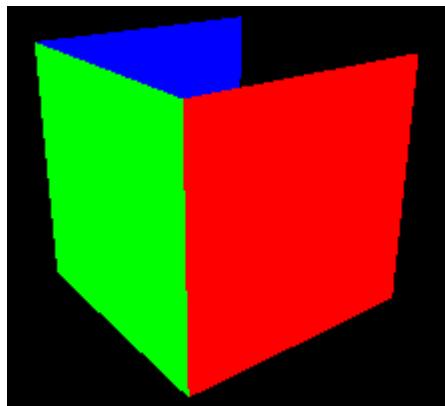
glColor3ub(0,0,255); //face bleue
glVertex3d(-1,-1,1);
glVertex3d(-1,-1,-1);
glVertex3d(1,-1,-1);
glVertex3d(1,-1,1);

glEnd();

glFlush();
SDL_GL_SwapBuffers();
}

```

Et en effet en exécutant notre nouveau code nous obtenons ceci :



Ouf ! Nous avons enfin ce que nous désirions. Bien pratique ce z-buffer !

Finir le cube

À ce stade il ne vous reste plus qu'à compléter le code avec les 3 faces restantes et leur choisir de belles couleurs. N'oubliez pas que vous pouvez utiliser le schéma du début pour facilement trouver les coordonnées des sommets de la face en cours.

Voici mon code pour ceux qui ne veulent pas essayer eux-mêmes, ou simplement pour comparer :

```

glBegin(GL_QUADS);

glColor3ub(255,0,0); //face rouge
glVertex3d(1,1,1);
glVertex3d(1,1,-1);
glVertex3d(-1,1,-1);
glVertex3d(-1,1,1);

glColor3ub(0,255,0); //face verte
glVertex3d(1,-1,1);
glVertex3d(1,-1,-1);
glVertex3d(1,1,-1);
glVertex3d(1,1,1);

glColor3ub(0,0,255); //face bleue
glVertex3d(-1,-1,1);
glVertex3d(-1,-1,-1);
glVertex3d(1,-1,-1);
glVertex3d(1,-1,1);

glColor3ub(255,255,0); //face jaune
glVertex3d(-1,1,1);
glVertex3d(-1,1,-1);
glVertex3d(-1,-1,-1);
glVertex3d(-1,-1,1);

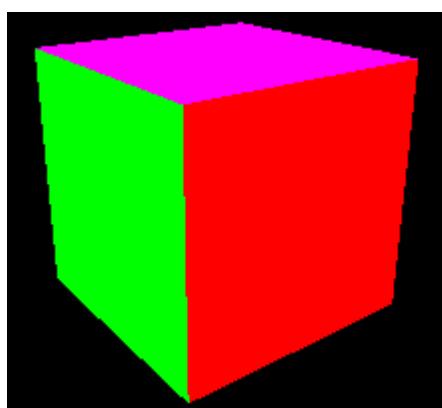
glColor3ub(0,255,255); //face cyan
glVertex3d(1,1,-1);
glVertex3d(1,-1,-1);
glVertex3d(-1,-1,-1);
glVertex3d(-1,1,-1);

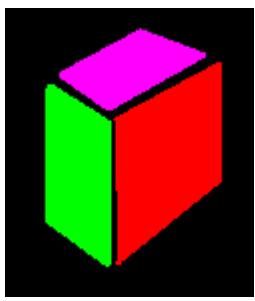
glColor3ub(255,0,255); //face magenta
glVertex3d(1,-1,1);
glVertex3d(1,1,1);
glVertex3d(-1,1,1);
glVertex3d(-1,-1,1);

glEnd();

```

Et le résultat graphique correspondant :





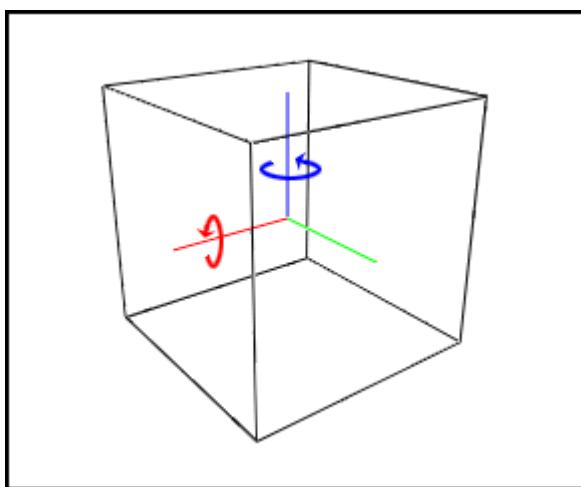
Comme vous avez le code sous les yeux vous savez que vous n'avez pas triché et que le code fait bien un cube 3D. Mais personnellement je vois 3 quadrillatères de couleur, je peux faire pareil sous Paint en 30 secondes, la preuve : Ok c'est moche et mal fait mais avec un peu de soin j'aurais pu avoir pareil ! :honte:

Il est donc temps de profiter de la puissance de la 3D temps réel et d'animer notre cube pour le voir sous toutes les coutures !

Animation

Pour animer notre cube nous allons simplement le faire tourner en utilisant ce que vous connaissez déjà par coeur : **la rotation** !

Au niveau du dessin nous n'avons vraiment pas grand chose à changer, juste à faire tourner le repère avant de dessiner le cube. Nous allons le faire tourner à la fois sur Z (la verticale) et X donc nous avons besoin de 2 variables globales* pour chacun des angles à contrôler.



* En général en programmation on essaye d'éviter les variables globales mais ici nous faisons en quelque sorte du prototypage pour apprendre et tester les concepts OpenGL, ce n'est donc vraiment pas bien grave.

Le code, simplifié, du programme devient alors :

```

#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <cstdlib>

void Dessiner();

double angleZ = 0;
double angleX = 0;

int main(int argc, char *argv[])
{
    //le code du main
}

void Dessiner()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );

    gluLookAt(3,4,2,0,0,0,0,0,1);

    glRotated(angleZ,0,0,1);
    glRotated(angleX,1,0,0);

    //dessin du cube

    glFlush();
    SDL_GL_SwapBuffers();
}

```

En terme de code SDL nous voulons que ces angles soient modifiés automatiquement avec le temps. On ne peut donc plus se permettre d'*attendre* les événements avec `SDL_WaitEvent` et nous allons en conséquence utiliser `SDL_PollEvent` pour récupérer les événements s'il y en a puis animer notre cube.

Nous modifions donc le code de notre boucle d'affichage pour incrémenter nos angles à chaque image :

```

for (;;)
{
    while (SDL_PollEvent(&event))
    {

        switch(event.type)
        {
            case SDL_QUIT:
                exit(0);
                break;
        }
    }

    angleZ += 1;
    angleX += 1;

    Dessiner();

}

```

Gérer la vitesse d'animation

En testant le code actuel vous voyez que le cube tourne beaucoup trop vite, nous n'avons franchement rien le temps de voir.

Il faut donc introduire des vitesses de rotation. Ces vitesses ne doivent pas dépendre de l'ordinateur sur lequel le programme est lancé et donc doivent prendre en compte le temps réel.

Pour ce faire, à chaque image (chaque passage dans la boucle donne lieu à une image), il faut déterminer combien de temps il s'est passé depuis la dernière image et faire bouger le cube en conséquence.

Nous avons donc besoin de 3 variables :

- une pour garder en mémoire le temps qu'il était lors de la dernière image : `last_time` ;
- une pour avoir le temps de l'image actuelle : `current_time` ;
- une (par commodité) pour le temps écoulé : `ellapsed_time`.

Pour connaître le temps écoulé, en millisecondes, depuis le lancement de l'application nous utiliserons l'instruction `SDL_GetTicks()`;

Le principe est alors le suivant.

1. On initialise *une première fois*, avant de rentrer dans notre boucle d'affichage `last_time` avec le temps actuel.
2. À chaque image on récupère le temps actuel dans `current_time`.
3. On utilise la différence entre le temps actuel et le temps qu'il était lors de l'ancien passage pour savoir combien de temps s'est écoulé. On stocke le résultat dans `ellapsed_time`.
4. On réalise nos mouvements en fonction du temps écoulé.
5. On finit par affecter à `last_time` la valeur de `current_time` car nous passons à une nouvelle image et donc le présent devient du passé (:'(c'est beau !)
- .

En ce qui concerne l'unité de mesure des **vitesses**, comme le temps écoulé est donné en *millisecondes* et que les angles utilisés dans `glRotate` sont en *degrés*, il s'agit tout simplement de **degrés par milliseconde**.

Dans notre cas j'utiliserai 0.05 °/ms.

La traduction en code du principe tout juste évoqué est la suivante :

```
Uint32 last_time = SDL_GetTicks();
Uint32 current_time, elapsed_time;

for (;;)
{
    while (SDL_PollEvent(&event))
    {

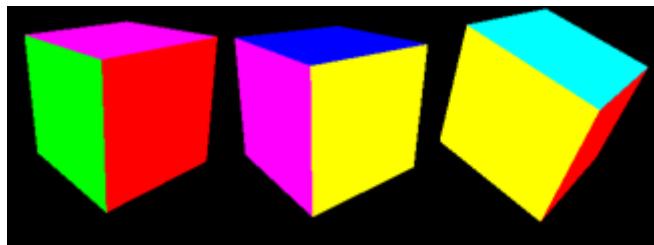
        switch(event.type)
        {
            case SDL_QUIT:
                exit(0);
                break;
        }
    }

    current_time = SDL_GetTicks();
    elapsed_time = current_time - last_time;
    last_time = current_time;

    angleZ += 0.05 * elapsed_time;
    angleX += 0.05 * elapsed_time;

    Dessiner();
}

}
```



Ne pas monopoliser le CPU

Ça rame ! Le programme prend 100% du CPU rien que pour faire tourner un simple cube, je ne vais jamais pouvoir faire un jeu !

En effet si on regarde la charge du processeur imposée par notre application on voit qu'il est totalement occupé à gérer notre programme :

Nom de l'image	Processeur
sdlglapp.exe	100

Cela ne veut pas dire que votre programme est lent c'est juste que nous bouclons en permanence et que nous ne prenons jamais de pause. En réalité nous n'avons pas vraiment besoin de boucler tout le temps. Nous allons utiliser une technique possible (celle que je préfère et donc souhaite vous expliquer) : **limiter les FPS** (frames per second - images par seconde).

En effet pour avoir une animation très fluide il nous suffit de 50 images par seconde.

En fixant le nombre d'images par secondes désirées par votre application, il est alors possible de la faire s'endormir un certain temps si elle va plus vite que nécessaire, ce qui soulagera (même s'il ne s'en plaint pas) le processeur. Pour ce faire nous allons calculer à chaque image combien de temps nous avons mis pour la dessiner, si nous avons été plus rapides que le temps moyen nécessaire, nous stopperons l'exécution pour un certain temps.

Par exemple, autoriser 50 images par seconde donne à chaque image 20 millisecondes pour s'afficher. Imaginons qu'une image mette 5 ms à s'afficher réellement, il reste alors 15 ms à tuer. Plutôt que de passer directement à l'image suivante, nous allons **endormir l'application** pendant ces 15 ms.

En terme de code nous allons utiliser `SDL_GetTicks` comme auparavant pour déterminer le temps écoulé entre le début et la fin de la création de l'image. Nous utiliserons `SDL_Delay` pour suspendre temporairement l'application.

```
Uint32 start_time; //nouvelle variable

for (;;)
{
    start_time = SDL_GetTicks();
    while (SDL_PollEvent(&event))
    {

        switch(event.type)
        {
            case SDL_QUIT:
                exit(0);
                break;
            case SDL_KEYDOWN:
                animation = !animation;
                break;
        }
    }

    current_time = SDL_GetTicks();
    elapsed_time = current_time - last_time;
    last_time = current_time;

    angleZ += 0.05 * elapsed_time;
    angleX += 0.05 * elapsed_time;

    Dessiner();

    elapsed_time = SDL_GetTicks() - start_time;
    if (elapsed_time < 10)
    {
        SDL_Delay(10 - elapsed_time);
    }

}

return 0;
}
```

Notez qu'ici nous ne voulons pas savoir combien de temps il s'est passé depuis la dernière fois mais combien de temps notre image a pris à se dessiner (j'y ai inclus la gestion des événements). Il y a donc un appel à `SDL_GetTicks` au début de notre boucle et un appel à la toute fin. Je réutilise `elapsed_time` par commodité mais pas les autres variables pour ne pas mélanger les 2 concepts : limitation des FPS et gestion du temps dans les animations.

Code final :

```

#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <cstdlib>

void Dessiner();

double angleZ = 0;
double angleX = 0;

int main(int argc, char *argv[])
{
    SDL_Event event;

    SDL_Init(SDL_INIT_VIDEO);
    atexit(SDL_Quit);
    SDL_WM_SetCaption("SDL GL Application", NULL);
    SDL_SetVideoMode(640, 480, 32, SDL_OPENGL);

    glMatrixMode(GL_PROJECTION );
    glLoadIdentity();
    gluPerspective(70,(double)640/480,1,1000);

    glEnable(GL_DEPTH_TEST);

    Dessiner();

    Uint32 last_time = SDL_GetTicks();
    Uint32 current_time,elapsed_time;
    Uint32 start_time;

    for (;;)
    {
        start_time = SDL_GetTicks();
        while (SDL_PollEvent(&event))
        {

            switch(event.type)
            {
                case SDL_QUIT:
                    exit(0);
                    break;
            }
        }

        current_time = SDL_GetTicks();
        elapsed_time = current_time - last_time;
        last_time = current_time;

        angleZ += 0.05 * elapsed_time;
        angleX += 0.05 * elapsed_time;

        Dessiner();
    }
}

```

```

    elapsed_time = SDL_GetTicks() - start_time;
    if (elapsed_time < 10)
    {
        SDL_Delay(10 - elapsed_time);
    }
}

return 0;
}

void Dessiner()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );

    gluLookAt(3,4,2,0,0,0,0,0,1);

    glRotated(angleZ,0,0,1);
    glRotated(angleX,1,0,0);

    glBegin(GL_QUADS);

    glColor3ub(255,0,0); //face rouge
    glVertex3d(1,1,1);
    glVertex3d(1,1,-1);
    glVertex3d(-1,1,-1);
    glVertex3d(-1,1,1);

    glColor3ub(0,255,0); //face verte
    glVertex3d(1,-1,1);
    glVertex3d(1,-1,-1);
    glVertex3d(1,1,-1);
    glVertex3d(1,1,1);

    glColor3ub(0,0,255); //face bleue
    glVertex3d(-1,-1,1);
    glVertex3d(-1,-1,-1);
    glVertex3d(1,-1,-1);
    glVertex3d(1,-1,1);

    glColor3ub(255,255,0); //face jaune
    glVertex3d(-1,1,1);
    glVertex3d(-1,1,-1);
    glVertex3d(-1,-1,-1);
    glVertex3d(-1,-1,1);

    glColor3ub(0,255,255); //face cyan
    glVertex3d(1,1,-1);
    glVertex3d(1,-1,-1);
    glVertex3d(-1,-1,-1);
}

```

```

glVertex3d(-1,1,-1);

glColor3ub(255,0,255); //face magenta
glVertex3d(1,-1,1);
glVertex3d(1,1,1);
glVertex3d(-1,1,1);
glVertex3d(-1,-1,1);

glEnd();

glFlush();
SDL_GL_SwapBuffers();
}

```

En comparaison, pour une animation de la même fluidité, la charge moyenne du processeur est **négligeable** :

Nom de l'image	Processeur
sdlglapp.exe	00

Notes diverses :

- il est *aussi* possible d'utiliser des timers pour limiter les FPS (voir la doc de `SDL_AddTimer` (http://www.libsdl.org/cgi/docwiki.cgi/SDL_5fAddTimer) ainsi que son exemple). Ce n'est pas la solution retenue ici ;
- les adeptes de la doc ne manqueront pas de signaler le problème de la granularité du temps de pause (cf. `SDL_Delay` (http://www.libsdl.org/cgi/docwiki.cgi/SDL_5fDelay)). Dans les tests effectués pour rédiger ce tutoriel, le temps de pause **réel** n'excédait jamais le temps **demandé** de plus de 1 milliseconde. Le nombre réel d'images par seconde est donc égal (ou très proche) au nombre fixé ;
- en terme d'ergonomie cela peut faire peur à certains de faire s'endormir le programme un certain temps. Qu'ils soient rassurés, même dans une application 3D bien plus complexe, la gestion des événements n'est en rien altérée.

Téléchargez le projet Code::Blocks, l'exécutable Windows et le Makefile Unix (117 Ko) (http://sdz.tdct.org/sdz/medias/www.siteduzero.com_uploads_fr_ftp_kayl_sdlgl_05_cube.zip)

Téléchargez la vidéo au format avi/Xvid (414 Ko) (http://62.4.17.167/uploads/fr/ftp/kayl/rotation_cube.avi)

Voilà le mystère de la 3D en OpenGL est enfin tombé !

Vous savez maintenant créer de toutes pièces un objet 3D et réaliser une animation.

En attendant le prochain chapitre sur les textures vous pouvez, si vous le souhaitez, améliorer votre programme pour créer de multiples objets, en animer certains grâce au clavier, et par exemple faire tourner la caméra autour de votre scène.

Bonne création !

Les textures

Maintenant que nous savons faire des objets en 3D, en couleur certes mais un peu moches il faut l'avouer :lol:, il est temps de les habiller grâce aux textures. Nous verrons donc dans ce chapitre les rudiments du *texturing*, comment charger une texture et l'appliquer sur un objet. De quoi nous ouvrir la voie vers des scènes 3D de plus en plus réalistes.

Charger une texture

Grâce à notre choix d'utiliser OpenGL avec la SDL, vous allez voir que la phase de chargement des textures nous sera hautement simplifiée.

Mais avant même de pouvoir charger une quelconque texture, encore faut-il en avoir... Pour cela je vous propose de travailler avec ce pack de textures hautes résolutions (<http://berneyboy.planetquake.gamespy.com/textures.htm>). Il est assez volumineux (124 Mo) mais assez complet et contient des textures photoréalistes classées dans diverses catégories : sols, caisses, végétaux, métaux, pierres, etc.



Exemples de textures incluses dans le pack

Que vous le téléchargez ou non je fournirai quand même les textures utilisées dans l'exemple en fin de chapitre. Vous pouvez donc vous en contenter.

Format et taille

Nous utiliserons principalement deux formats pour les textures : **.jpg** et **.png**.

Le format **.jpg** est parfait pour les textures, car il donne les plus petites tailles de fichier sur des images complexes (comme les textures photoréalistes).

Le format **.png (24 bits)** quant à lui est utilisé, car il gère très bien la transparence.

La largeur et la hauteur des textures doivent **impérativement** être des puissances de 2 (64,128,512,1024). En effet si elles ne le sont pas, les textures seront de toute façon redimensionnées en interne pour respecter cette contrainte et vous risquez donc de perdre inutilement en **qualité**.

En terme de **qualité visuelle**, plus la résolution de la texture est grande, meilleur est le résultat. Le pack que je vous fournis contient principalement des textures haute-résolution (512x512) avec lesquelles vous n'aurez pas de mal à avoir des rendus de meilleure qualité que Half-Life premier du nom :soleil: (effets de lumière mis à part pour l'instant).

Utiliser **SDL_Image** pour charger une texture

Le code nécessaire pour créer une texture OpenGL à partir d'un tableau de pixels n'est pas extrêmement compliqué. Cependant la phase la plus pénible est le chargement d'un fichier image (.jpg, .bmp, .tga ou autre).

Heureusement pour nous **SDL_Image** est là ; nous aurons simplement à l'utiliser pour qu'elle nous retourne une **SDL_Surface** à partir d'un nom de fichier.

Une fois cette surface créée, elle doit être **retournée verticalement** car SDL et OpenGL n'ont pas les mêmes conventions. Vous avez dû le remarquer lors du chapitre sur les transformations car le (0,0) en OpenGL était en bas à gauche alors que celui en SDL (comme indiqué dans le cours de M@teo) est en haut à gauche.

La dernière chose à faire est de convertir le tableau de pixels contenus dans la surface en texture OpenGL par des appels OpenGL appropriés.

Oh là là ! Ça a l'air compliqué tout ça... Image utilisateur :(Je vais réussir à faire ça moi ?

Savoir coder vous-mêmes cette phase n'est pas nécessaire pour la compréhension de la suite du chapitre. Le plus important est l'**utilisation des textures** créées. Nous reviendrons plus en détail sur les appels en question lorsque nous verrons les textures procédurales mais pour l'instant je vous donne tout ça sur un plateau d'argent, voilà :

Téléchargez le code pour charger une texture (3 Ko)

(http://sdz.tdct.org/sdz/medias/www.siteduzero.com_uploads_fr_ftp_kayl_sdz_sdlglutils.zip)

Dans l'archive, je vous fournis deux fichiers : sdlglutils.h et sdlglutils.cpp. J'ai choisi ce nom car j'y rajouterais petit à petit des fonctions utiles pour ce tuto qui ne sont là que pour nous simplifier la vie mais que vous auriez pu faire vous-mêmes avec un peu plus de connaissances.

La fonction qui nous intéresse pour l'instant est loadTexture qui s'utilise très simplement :

```
#include "sdlglutils.h"
GLuint identifiant_texture = loadTexture("ma_texture.jpg");
```

Si vous êtes curieux vous pouvez regarder le code derrière mais il n'a rien de compliqué, juste ce que je vous ai expliqué plus haut (chargement de l'image, retournement, et création texture OpenGL).

Le type renvoyé est GLuint soit l'équivalent d'un unsigned int. Cependant je n'utiliserais jamais le type unsigned int pour les textures pour ne pas être tenté dans mon code de faire des calculs dessus, en effet ce nombre retourné a une signification bien précise : c'est l'**identifiant de la texture** OpenGL créée. À chaque fois que nous voudrons utiliser cette texture, il suffira d'utiliser cet identifiant.

Activer le texturing

Comme nous avons déjà créé des scènes en 3D sans texture nous savons que le texturing n'est pas activé par défaut. Pour l'activer il suffit donc d'appeler :

```
 glEnable(GL_TEXTURE_2D);
```

Vous pouvez avoir envie dans vos scènes de faire cohabiter des objets texturés avec des objets non texturés (pourquoi pas après tout...). Pour ce faire il suffit de désactiver le texturing temporairement avec :

```
 glDisable(GL_TEXTURE_2D);
```

Voilà nous sommes prêts, nous savons charger une texture, nous savons activer le texturing. Maintenant voyons comment, en reprenant notre cube 3D du chapitre précédent, appliquer une texture sur un objet.

Plaillage de texture

Partons du code que nous avions pour définir la première face de notre cube :

```
 glBegin(GL_QUADS);
 glVertex3d(1,1,1);
 glVertex3d(1,1,-1);
 glVertex3d(-1,1,-1);
 glVertex3d(-1,1,1);
 glEnd();
```

Nous allons utiliser la texture « stainedglass05.jpg » que vous trouverez dans le pack dans la catégorie window (ou dans l'archive en fin de chapitre).

Commençons donc par la charger au lancement de notre application comme expliqué précédemment :

```

GLuint texture1; //en variable globale

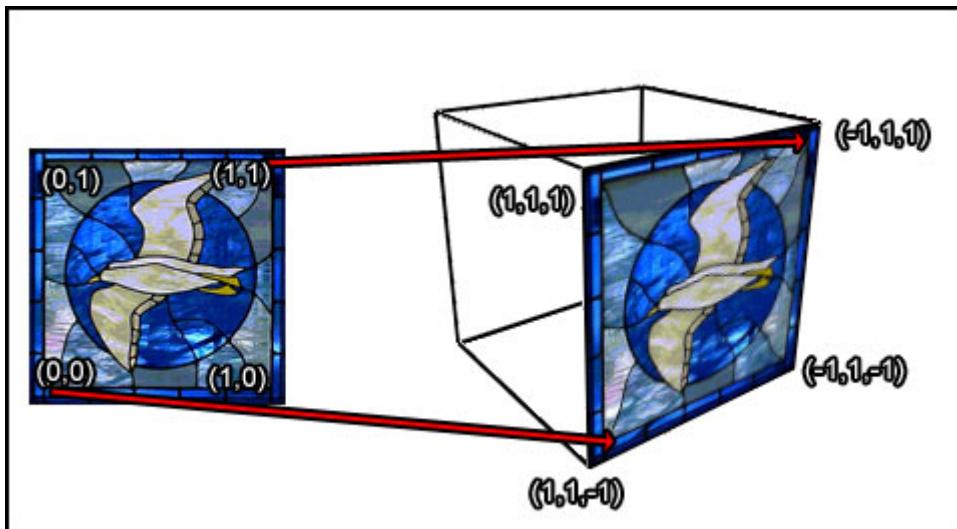
int main(int argc, char *argv[])
{
    // ... lancement de l'application
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_TEXTURE_2D);
    texture1 = loadTexture("stainedglass05.jpg"); // pendant l'initialisation d'OpenGL, avant la boucle d'affichage
    //... boucle d'affichage et de gestion des événements
}

```

Dans le code du dessin, avant de définir les vertices de notre première face texturée, il nous faut dire à OpenGL que l'on veut utiliser cette texture en appelant glBindTexture avec l'identifiant de notre texture :

```
glBindTexture(GL_TEXTURE_2D, texture1);
```

Et maintenant nous allons réaliser le *plaquage* proprement dit de la texture sur la face en faisant correspondre à chaque vertex composant notre face **une coordonnée sur la texture** comme représenté sur le schéma ci-dessous :



Correspondance coordonnées texture / coordonnées réelles

Pour définir les coordonnées du vertex nous savons utiliser glVertex, eh bien pour définir les coordonnées de texture nous utiliserons :

```
glTexCoord2d (double x_texture, double y_texture);
```

L'espace de coordonnées sur la texture est en 2D, le coin en bas à gauche a les coordonnées (0,0) et le coin en haut à droite est en (1,1), quelque soit la taille en pixels de l'image.

Ici je vais commencer par définir le vertex (1,1,1) auquel je veux plaquer le coin haut gauche de la texture soit (0,1). Je fais donc :

```

glBegin(GL_QUADS);
    glTexCoord2d(0,1);  glVertex3d(1,1,1);

```

glTexCoord2d, comme glColor3ub, s'applique à tous les vertices définis par la suite. Il ne faut donc pas oublier d'y faire à nouveau appel avant chaque vertex.

En continuant avec les autres sommets de la face cela donne donc le code complet suivant :

```

glBindTexture(GL_TEXTURE_2D, texture1);
glBegin(GL_QUADS);
glTexCoord2d(0,1); glVertex3d(1,1,1);
glTexCoord2d(0,0); glVertex3d(1,1,-1);
glTexCoord2d(1,0); glVertex3d(-1,1,-1);
glTexCoord2d(1,1); glVertex3d(-1,1,1);
glEnd();

```

Il n'est pas possible de changer de texture en cours de définition d'une face. Pour appliquer une texture différente à la face suivante il faut donc terminer le bloc de vertices par un appel à `glEnd()`, changer la texture puis définir la prochaine face :

```

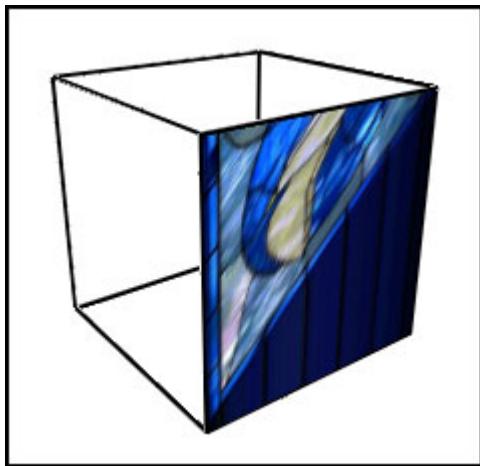
glBindTexture(GL_TEXTURE_2D, texture1);
glBegin(GL_QUADS); //première face
//définition des vertices
glEnd();
glBindTexture(GL_TEXTURE_2D, texture2); //changement de texture
glBegin(GL_QUADS); //deuxième face
//définition des vertices
glEnd();

```

Ce n'est bien sûr pas nécessaire si vous souhaitez garder la même texture.

Erreurs fréquentes

Une mauvaise utilisation de `glTexCoord2d` (mauvaises coordonnées, oublié de redéfinir `glTexCoord2d` pour le vertex suivant) donne lieu à de drôles de résultats :



Erreur en oubliant de redéfinir glTexCoord2d pour le dernier vertex

:waw: :p Ne rigolez pas ces erreurs sont fréquentes au début et je suis prêt à parier que ça vous arrivera au moins une fois :lol:. Vous saurez au moins d'où vient le problème...

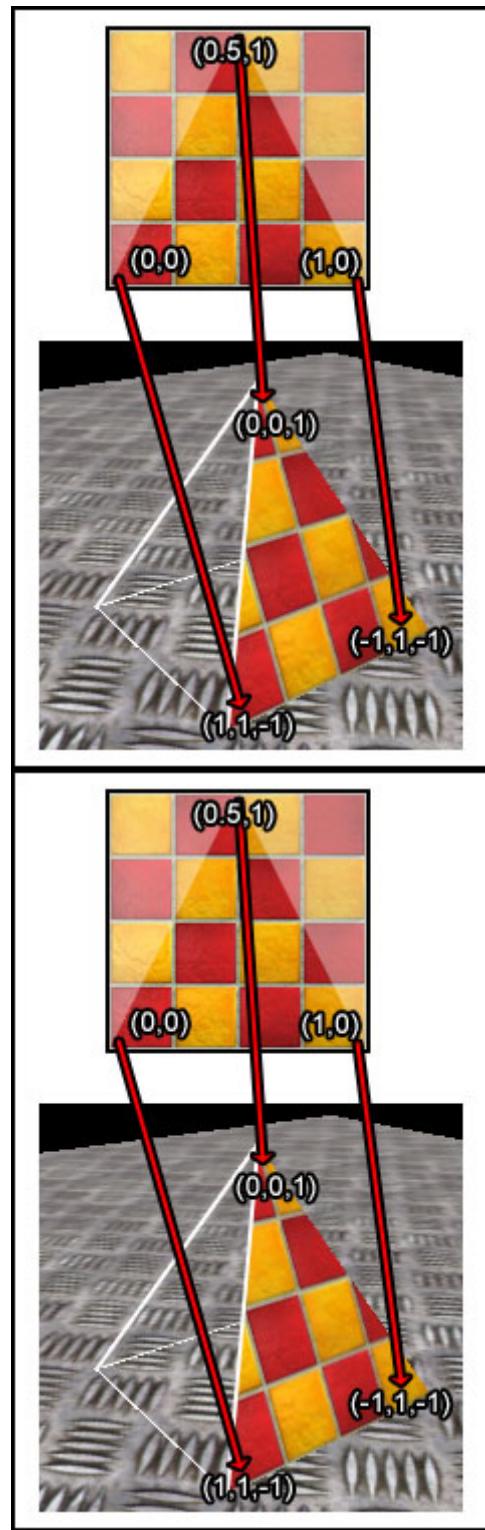
Cas d'un triangle

Dans les jeux vidéos, la primitive de base la plus utilisée est le triangle (nous verrons pourquoi quand nous importerons des modèles 3D). Le plaquage de texture fonctionne de la même manière en ne choisissant que **3 points sur la texture**, points qui ne sont donc pas forcément des coins de l'image.

```

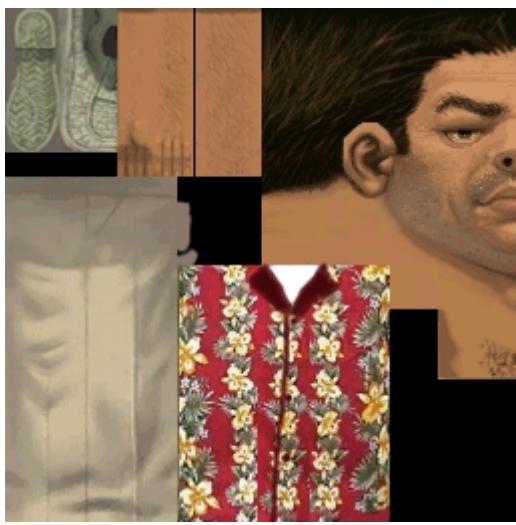
glBindTexture(GL_TEXTURE_2D, texture
2);
    glBegin(GL_TRIANGLES);
        glTexCoord2d(0,0);      glVertex3d
(1,1,-1);
        glTexCoord2d(1,0);      glVertex3d
(-1,1,-1);
        glTexCoord2d(0.5,1);    glVertex3d
(0,0,1);
    glEnd();

```



Utilisation d'une partie de la texture

Nous venons de le voir dans le cas du triangle, il n'est pas obligatoire d'utiliser toute la texture. C'est une pratique souvent utilisée pour regrouper toutes les textures pour un seul objet dans un seul fichier, comme par exemple ici la skin de Tommy Vercetti dans GTA:Vice City :



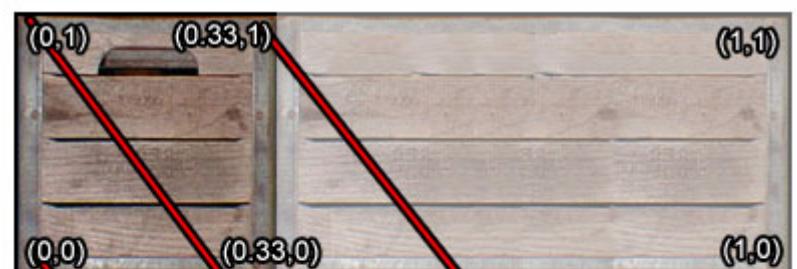
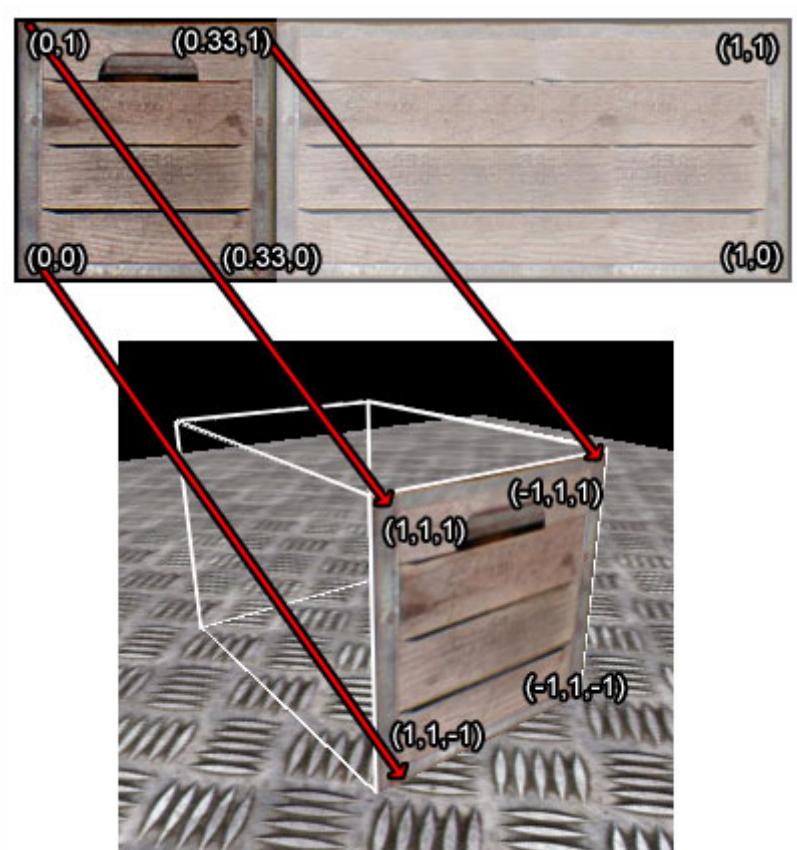
*Texture du héros (une des skins possibles) de **GTA:Vice City***

Il suffit alors, à coup de glTexCoord2d bien pensés, de délimiter la zone de la texture que l'on souhaite utiliser pour la face courante :

```

glBindTexture(GL_TEXTURE_2
D, texture3);
glBegin(GL_QUADS);
glTexCoord2d(0,1); gl
Vertex3d(1,1,1);
glTexCoord2d(0,0); gl
Vertex3d(1,1,-1);
glTexCoord2d(0.33,0);
glVertex3d(-1,1,-1);
glTexCoord2d(0.33,1);
glVertex3d(-1,1,1);
glEnd();

```

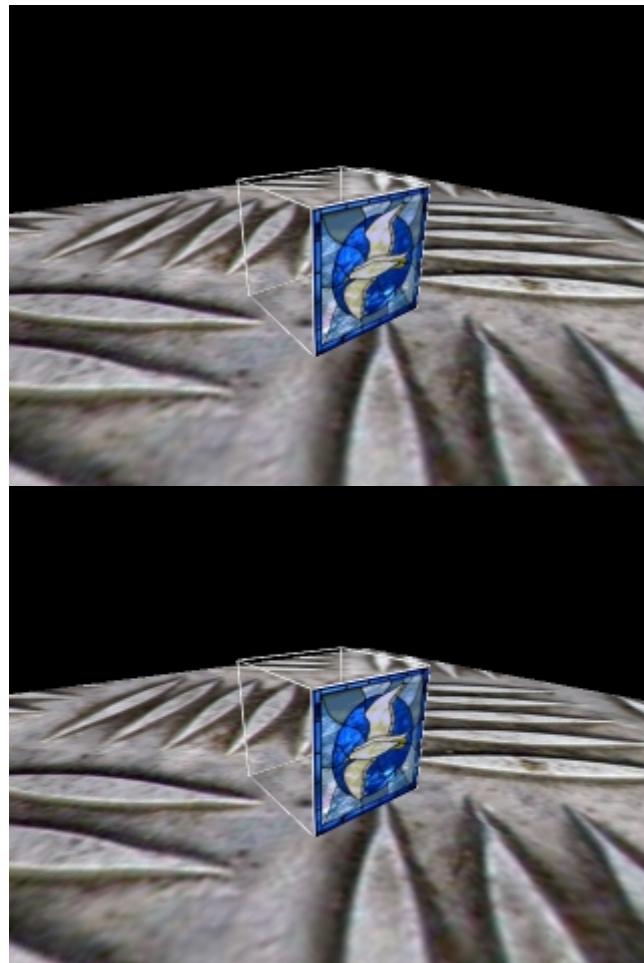


Comme vous le voyez, la texture utilisée ici n'est pas un carré (384x128) et pourtant le coin en haut à droite a toujours les coordonnées (1,1). Les coordonnées d'un point sur la texture donnent donc une information relative à la taille totale de l'image. Ici on a voulu découper une partie de 128 pixels de large sur les 384 totaux soit un rapport de 1/3 (0.33).

Texture répétitive

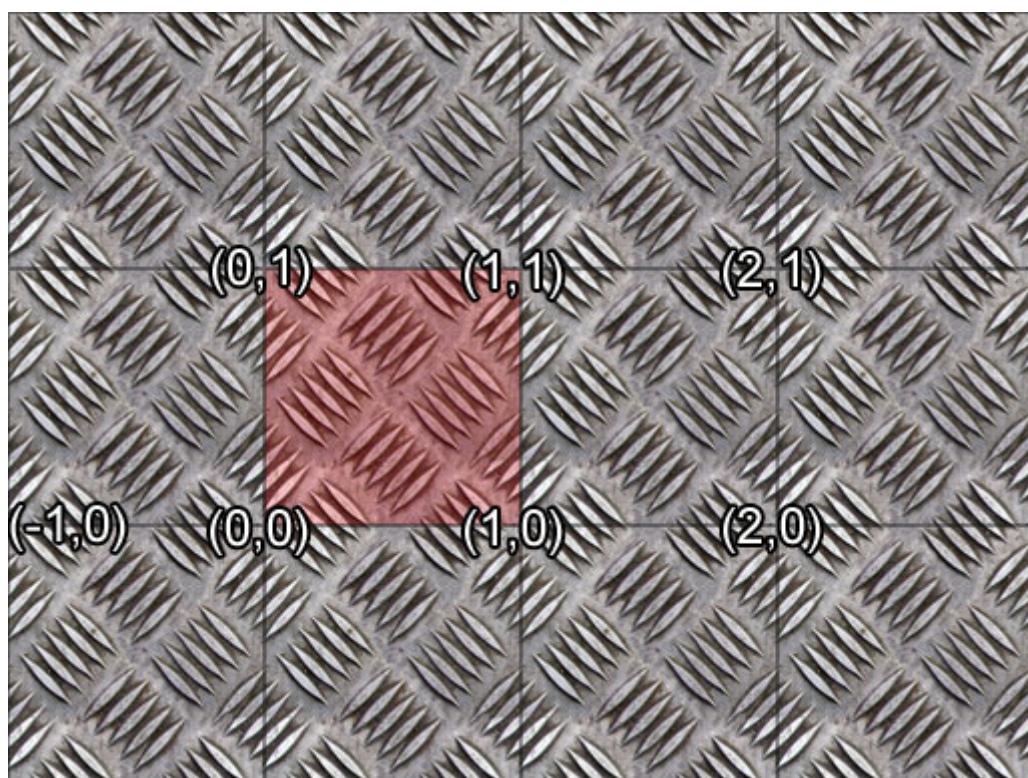
Jusqu'à présent nous appliquons tout ou une partie de la texture sur nos objets. Mais si on essaye de créer un sol (un simple carré de 20x20) dans la scène avec la même technique on obtient le résultat suivant :

```
glBindTexture(GL_TEXTURE_2D, texture4);
glBegin(GL_QUADS);
    glTexCoord2i(0,0);           glVertex
tex3i(-10,-10,-1);
    glTexCoord2i(1,0);           glVertex
ex3i(10,-10,-1);
    glTexCoord2i(1,1);           glVertex
x3i(10,10,-1);
    glTexCoord2i(0,1);           glVertex
ex3i(-10,10,-1);
    glEnd();
```



Comme vous le voyez la texture n'est pas faite pour être étalée sur une si grande surface. Nous devons donc faire en sorte qu'elle se répète !

Dans les exemples que nous avons vus nous n'utilisions que des coordonnées entre 0 et 1. Mais en réalité l'espace de coordonnées de la texture n'a pas de limite :



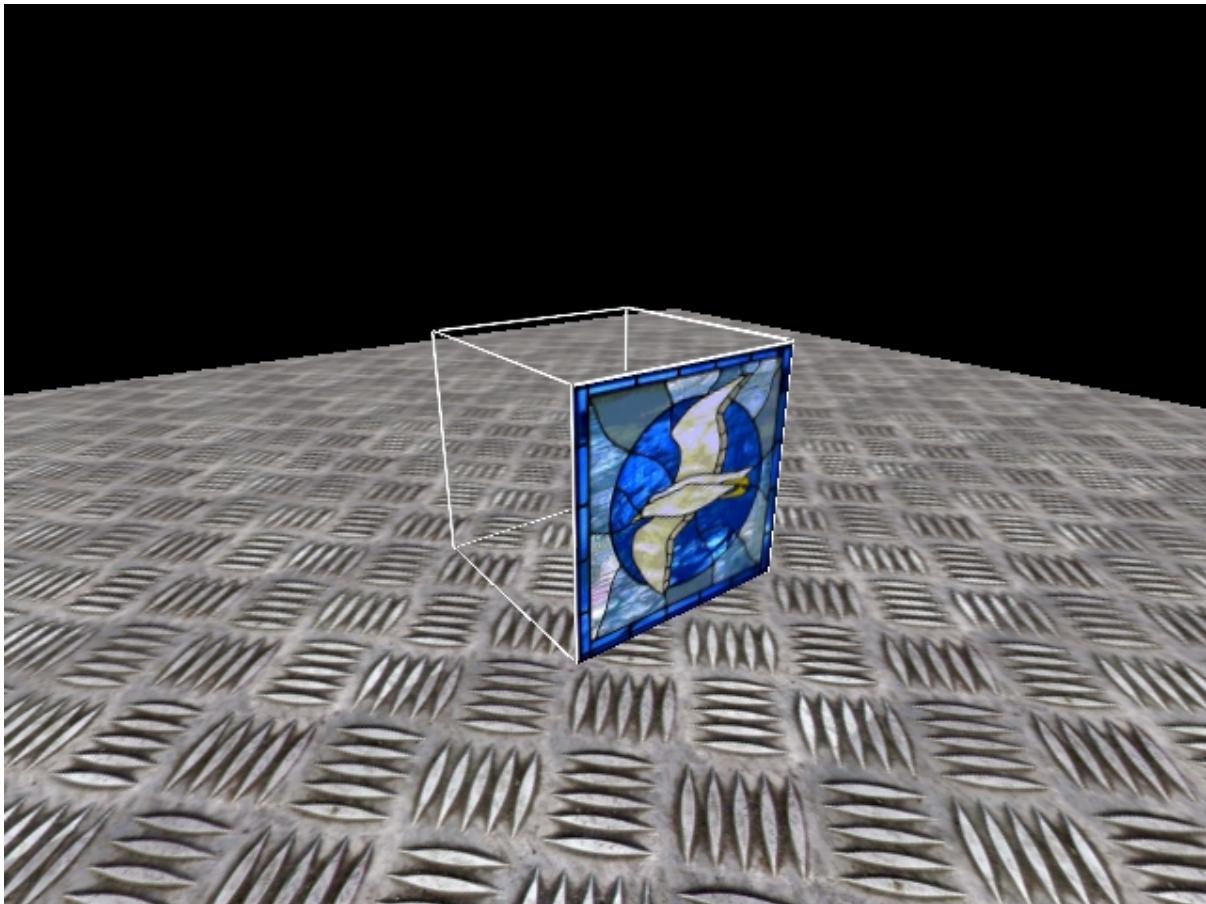
Espace des coordonnées de texture

Quand nous considérons l'image d'origine, nous nous restreignons à une partie de cet espace. Mais ici nous voulons faire répéter la texture 10 fois par exemple donc nous allons prendre des coordonnées entre 0 et 10 tout simplement !

Et en effet en modifiant le code en conséquence :

```
glBindTexture(GL_TEXTURE_2D, texture4);
glBegin(GL_QUADS);
glTexCoord2i(0,0);    glVertex3i(-10,-10,-1);
glTexCoord2i(10,0);   glVertex3i(10,-10,-1);
glTexCoord2i(10,10);  glVertex3i(10,10,-1);
glTexCoord2i(0,10);   glVertex3i(-10,10,-1);
glEnd();
```

Nous obtenons un bien meilleur résultat visuel :



Texture du sol répétée 10 fois

Toutes les textures ne sont pas faites pour être répétées. Il faut en effet qu'elles soient conçues spécialement pour les bords correspondent quand plusieurs répétitions de l'image sont mises bout à bout. Dans le pack de textures, vous pouvez être quasiment sûrs que toutes les textures de sol, de mur, de plafond, d'herbe, de rocher sont susceptibles d'être répétées.

Les couleurs

Je n'ai volontairement pas mentionné les couleurs depuis le début de ce chapitre pour rester concentré sur la nouveauté du moment : les textures. Mais nos bonnes vieilles couleurs ne sont pas mortes pour autant.

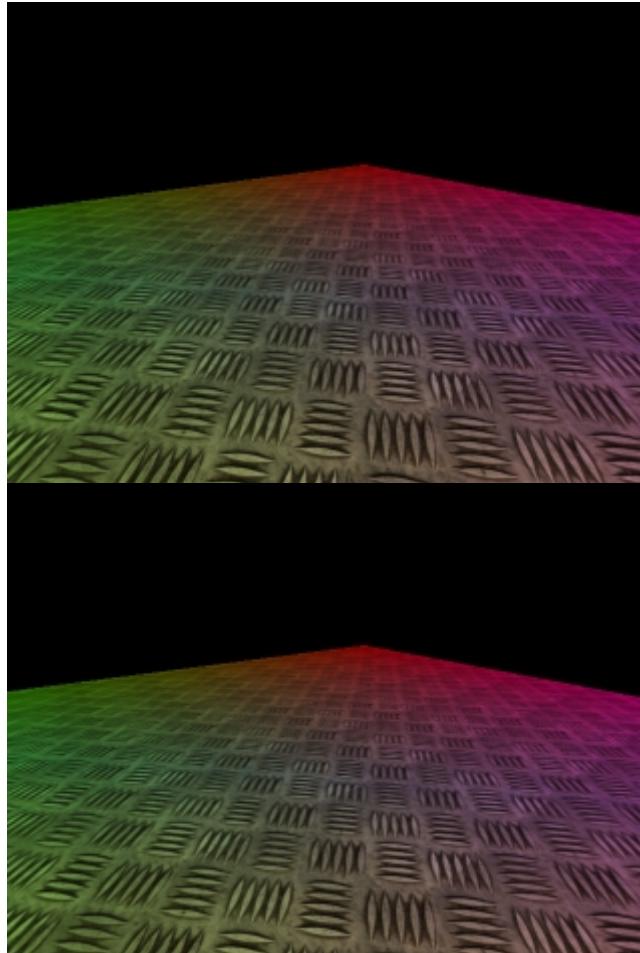
Combiner texture et couleur

Nous savons depuis le début de ce tuto définir la couleur des vertexes à venir avec `glColor3ub`. Rien ne nous interdit de continuer à l'utiliser en plus de ce que nous venons d'apprendre sur les textures. La définition complète d'un vertex peut donc maintenant contenir jusqu'à 3 lignes :

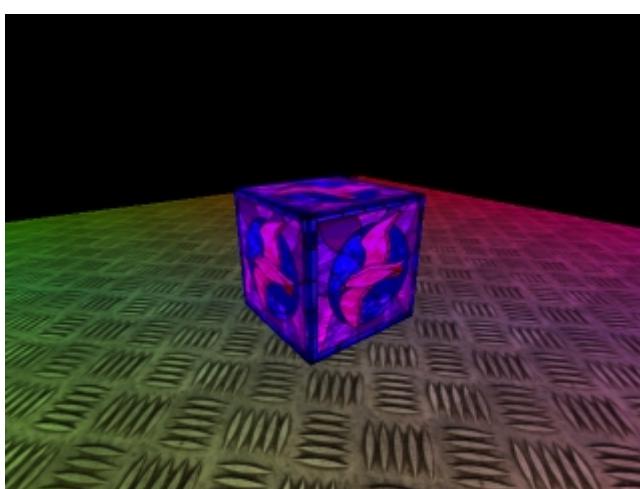
- définition de la couleur avec `glColor3ub(facultatif)`;
- définition des coordonnées sur la texture avec `glTexCoord2d(obligeatoire sur utilisation d'une texture)`;
- définition des coordonnées spatiales du vertex avec `glVertex3d`.

Il n'est bien sûr pas obligatoire de redéfinir `glColor3ub` à chaque fois si l'on ne souhaite pas changer de couleur. Appliquée au sol vu précédemment, en affectant des couleurs à chaque sommet on obtient donc :

```
glBindTexture(GL_TEXTURE_2
D, texture4);
glBegin(GL_QUADS);
glColor3ub(255,0,0); //Nouveau
glTexCoord2i(0,0);
glVertex3i(-10,-10,-1);
glColor3ub(0,255,0); //Nouveau
glTexCoord2i(10,0);
glVertex3i(10,-10,-1);
glColor3ub(255,255,0);
//Nouveau
glTexCoord2i(10,10);
glVertex3i(10,10,-1);
glColor3ub(255,0,255);
//Nouveau
glTexCoord2i(0,10);
glVertex3i(-10,10,-1);
glEnd();
```



Comme vous le voyez, la couleur agit comme un filtre et vient donner une teinte locale à la texture. Tout semble aller très bien jusqu'à ce que l'on décide d'afficher autre chose *juste après* avoir défini notre sol, un cube texturé par exemple :



Comme vous le voyez le cube est lui aussi affecté par la dernière couleur utilisée. C'est tout à fait logique car `glColor3ub` par définition est utilisé pour **tous** les vertex définis à la suite (jusqu'au prochain appel à `glColor3ub`).

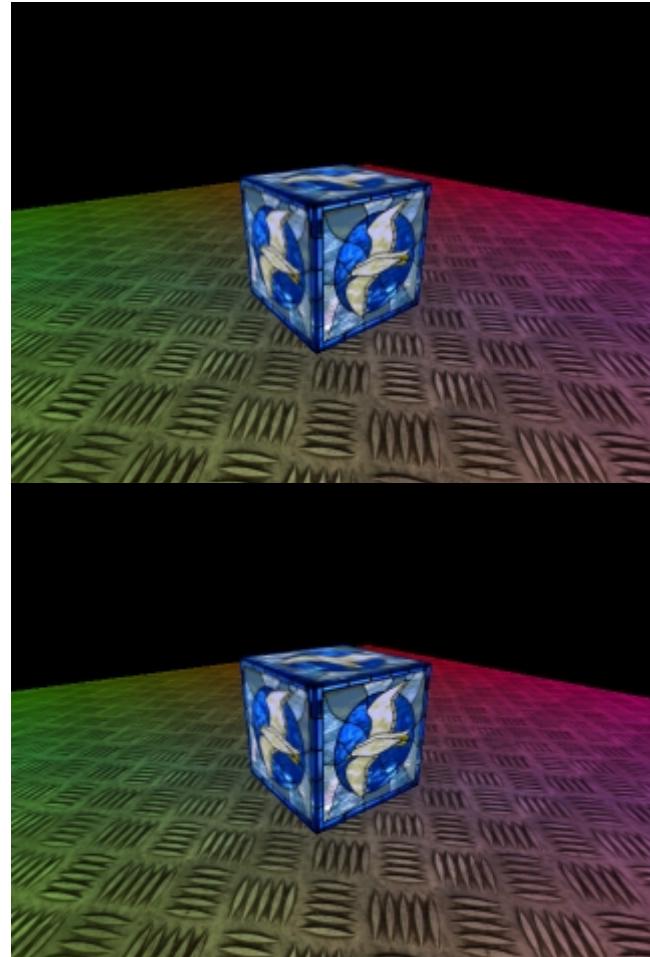
Il faut donc utiliser une **couleur neutre** qui, appliquée comme filtre, ne viendra pas modifier la couleur de la texture. Et cette couleur n'est autre que... le **blanc** !

Ainsi, pour en quelque sorte « *annuler l'effet des couleurs* », il suffit d'utiliser le blanc comme prochaine couleur avec un appel à :

```
glColor3ub(255,255,255);
```

Et en effet en interposant un appel à glColor3ub(255,255,255); entre la définition du sol et celle du cube, on obtient bien un cube vierge de tout effet de couleur :

```
//... début du sol  
glColor3ub(255,0,255);  
glTexCoord2i(0,10);  
glVertex3i(-10,10,-1);  
glEnd(); //fin du sol  
  
glColor3ub(255,255,255); //on  
enlève la couleur  
  
glBegin(GL_QUADS); //début du  
cube  
glTexCoord2d(0,1);  
glVertex3d(1,1,1);  
//... fin du cube
```



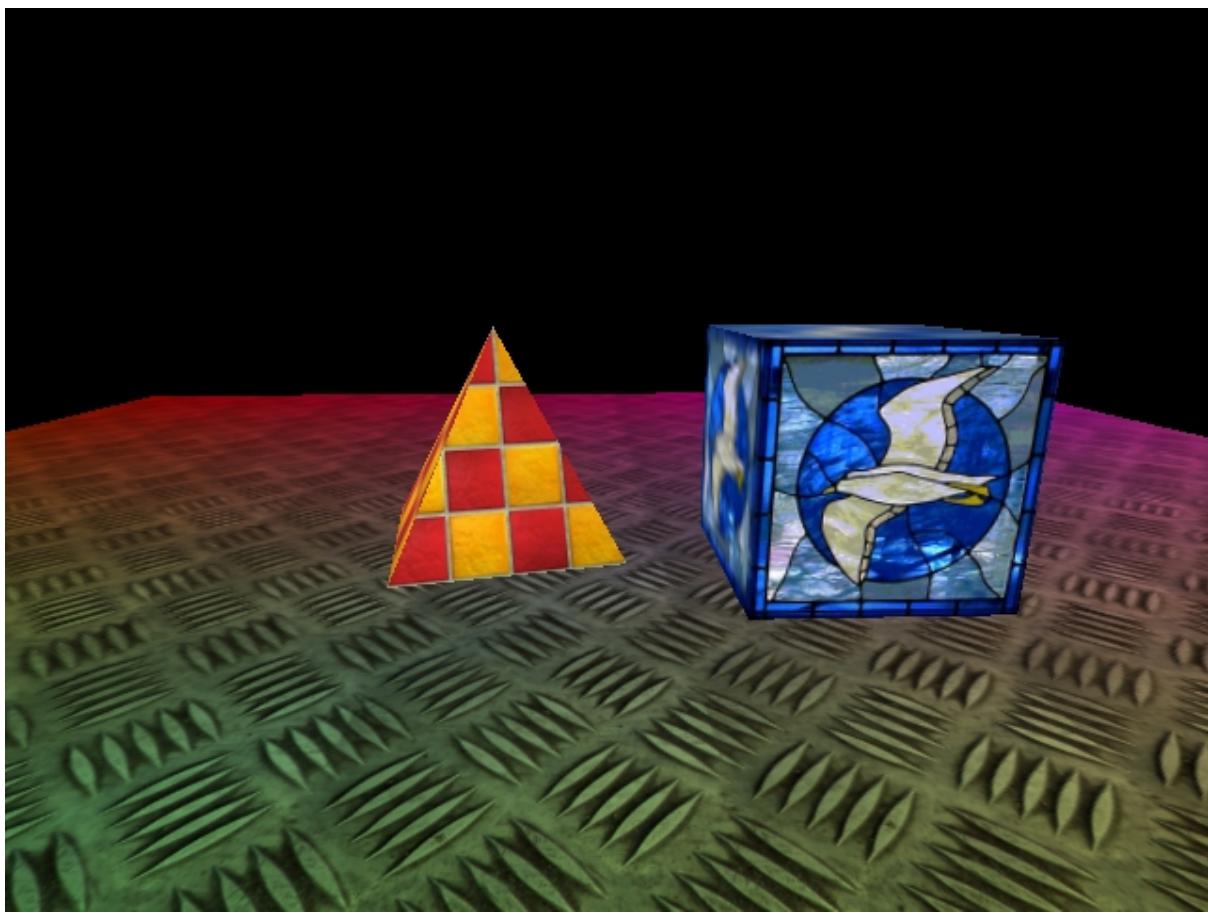
Voilà vous savez maintenant les précautions qu'il faut prendre lorsque l'on souhaite combiner dans un même code les couleurs et les textures. Rassurez-vous, il ne vous sera pas rare d'oublier de repasser en blanc de temps en temps et vous créerez souvent de jolis effets de couleur involontairement. ;)

Qui aurait cru que nous passerions aussi rapidement d'objets moches mais joliment colorés à de somptueux objets texturés ?!

Comme vous avez pu le voir il n'y a vraiment rien de sorcier dans l'application des textures, il suffit de bien savoir affecter à chaque vertex les bonnes coordonnées sur la texture et le tour est joué.

Le meilleur moyen de vous assurer que vous avez bien compris est de vous entraîner à réaliser une petite scène en 3D avec des textures, notamment une caisse de 4x2x2 avec la texture de caisse (voir caisse.jpg dans zip plus bas) utilisée dans ce chapitre.

Je vous en ai fait une rapidement dont vous pouvez télécharger le code plus bas :



Laissez libre cours à votre imagination et n'hésitez pas à poster vos créations dans les commentaires du chapitre ou sur le forum.

Téléchargez les textures utilisées et un exemple de scène texturée (482 Ko)
(http://sdz.tdct.org/sdz/medias/www.siteduzero.com_uploads_fr_ftp_kayl_sdlgl_06_textures.zip)

Téléchargez la vidéo au format avi/Xvid (1.10 Mo) (http://62.4.17.167/uploads/fr/ftp/kayl/scene_texture.avi)

Dans le prochain chapitre nous verrons comment créer des formes un peu plus complexes qu'un simple cube ou une pyramide, toujours dans le but d'enrichir le contenu de nos scènes.

Les quadriques

Le mot quadrique, équivalent de « surface quadratique », a une connotation mathématique qui peut faire peur au premier abord. :euh:

Mais ce n'est rien d'autre qu'une surface non-linéaire (pas plane), et derrière ce nouveau mot encore plus barbare, OpenGL regroupe en fait la **sphère**, le **cône**, le **cylindre**, et le **disque**.

Nous verrons donc dans ce court chapitre comment enrichir un peu plus vos scènes avec ces quelques formes prédéfinies qui nous éviteront bien des efforts.

Principe d'utilisation

Nous le verrons juste après, il existe dans OpenGL des fonctions toutes faites pour dessiner une sphère, un cylindre, etc.

Cependant ces fonctions ont besoin d'informations sur les intentions du codeur : faut-il texturer l'objet à créer, faut-il l'afficher uniquement avec des traits ?

Pour cela nous utiliserons un champ (struct (http://www.siteduzero.com/tuto-3-4350-1-creez-vos-propres-types-de-variables.html#ss_part_1)) de **paramètres**. Ces paramètres sont stockés dans une variable de type GLUquadric que l'on se doit d'utiliser d'une manière un peu particulière.

Création d'une variable de type GLUquadric

Ce n'est pas à nous directement de créer une variable de ce type. On doit utiliser un appel OpenGL qui nous renverra un pointeur (<http://www.siteduzero.com/tuto-3-3828-1-a-l-assaut-des-pointeurs.html>) sur le GLUquadric créé par OpenGL :

```
GLUquadric* params;  
params = gluNewQuadric();
```

Nous pouvons donc maintenant utiliser la variable params créée pour paramétrer nos objets et les dessiner.

Paramétrage du GLUquadric

Ce champ n'est pas manipulable directement mais seulement par l'intermédiaire de fonctions. Une fois paramétré il sera utilisé pour tous les appels de dessin de quadriques définis après.

Style d'affichage

Les fonctions pour définir des quadriques utilisent tout comme nous des appels à glVertex pour définir les vertices des objets. Cependant comme nous n'avons pas accès directement au code, et donc au glBegin, nous ne pouvons pas spécifier par exemple par un glBegin(GL_LINES); que nous voulons que l'affichage soit fait avec des lignes.

Pour ce faire nous devons utiliser la fonction :

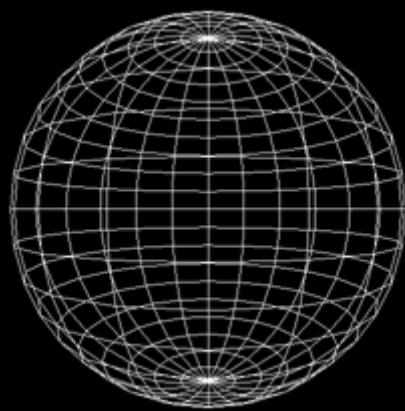
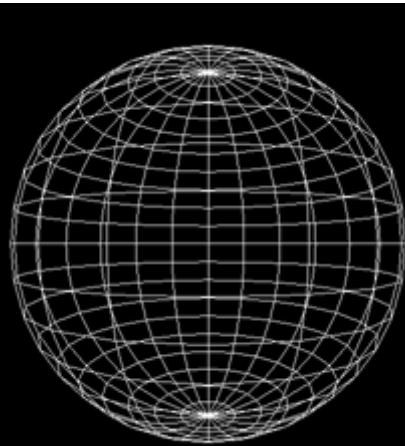
```
gluQuadricDrawStyle(params,style);  
qui permet de définir quel sera le style d'affichage. style peut valoir :
```

Style	Explication	Exemple
GLU_POINT	L'objet sera dessiné avec des points.	

Style**Explication****Exemple**

GLU_LINE

L'objet sera dessiné avec des lignes.



GLU_FILL

L'objet sera dessiné avec des faces pleines.



La valeur par défaut est GLU_FILL. Pour utiliser des faces pleines il n'est donc pas nécessaire d'appeler gluQuadricDrawStyle si aucun style n'a été précédemment défini.

Coordonnées de texture

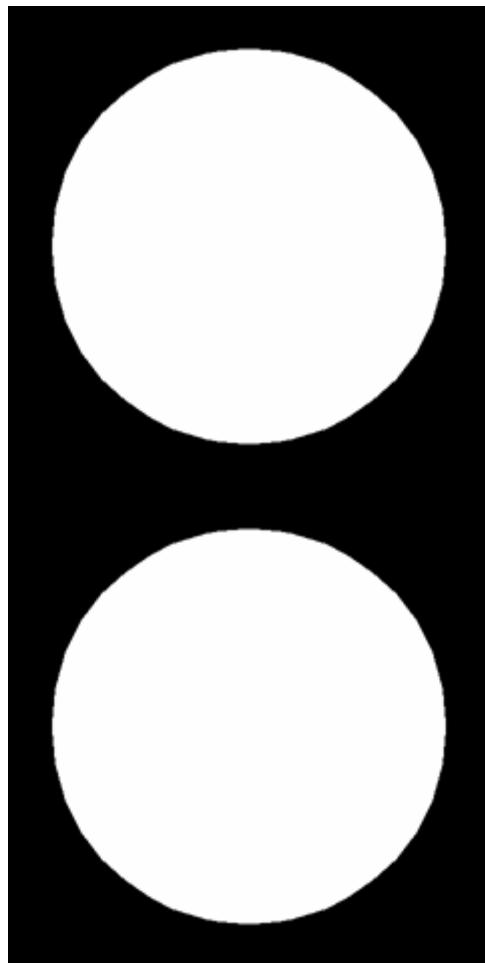
Nous l'avons vu dans le précédent chapitre, pour utiliser des textures il faut définir des coordonnées de texture avec glTexCoord2d. Si nous souhaitons utiliser des textures avec nos quadriques, il faut indiquer à OpenGL qu'il doit lui aussi incorporer les appels à glTexCoord2d lorsqu'on demandera de dessiner un quadrique. Nous utilisons donc la fonction :

```
gluQuadricTexture(params,texture);
```

où texture peut valoir GL_TRUE (vrai : pour activer les coordonnées de texture) ou GL_FALSE (faux : pour ne pas utiliser les coordonnées de texture).

La valeur par défaut étant GL_FALSE, il n'est nécessaire d'appeler initialement gluQuadricTexture que si nous souhaitons utiliser les textures sur nos quadriques.

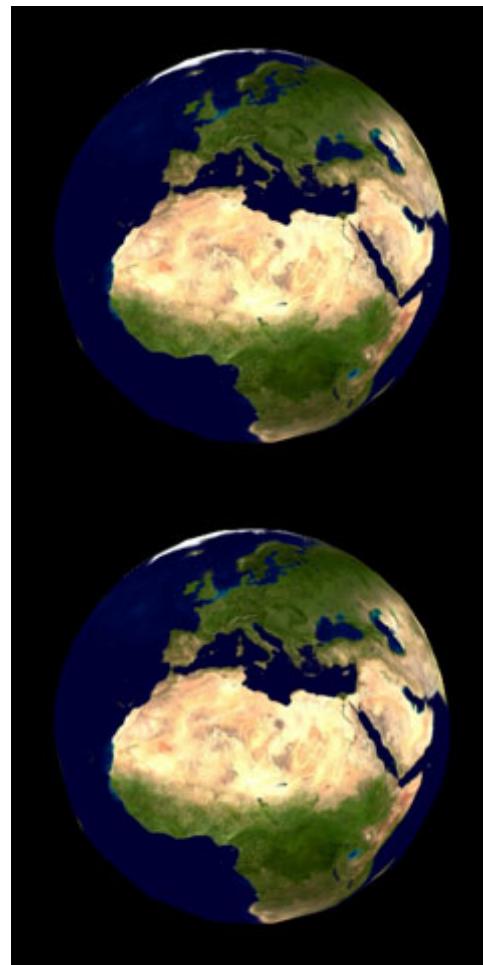
```
glBindTexture(GL_TEXTURE_2D,texture1);
GLUquadric* params = gluNewQuadric();
//dessin de la sphere... (à venir)
gluDeleteQuadric(params);
```



```

glBindTexture(GL_TEXTURE_2D, texture1);
GLUquadric* params = gluNewQuadric();
gluQuadricTexture(params, GL_TEXTURE);
//dessin de la sphère... (à venir)
gluDeleteQuadric(params);

```



Sur la première image c'était pas censé être une sphère ? On dirait un simple disque...

Sans texture et **sans lumière**, on ne peut pas comprendre que c'est une sphère. C'est un principe d'optique : la compréhension de la forme d'un objet sur une image 2D utilise pour beaucoup les différences de couleurs dues à l'éclairage (Shape from Shading (<http://www.google.com/search?hl=fr&q=shape%2Bfrom%2Bshading>)). Nous réglerons donc ce problème dans quelques chapitres lorsque nous verrons la *lumière*. En attendant, avec les **textures** et le **mouvement** nous n'aurons vraiment pas de mal à discerner la forme de nos objets rassurez-vous.

Suppression du GLUquadric

Même si nous n'avons pas nous-mêmes utilisé de malloc (ou new en C++) pour créer le GLUquadric, il faut libérer la mémoire quand on ne souhaite plus l'utiliser par le biais de la fonction (entrapérue dans les exemples de code plus haut) :

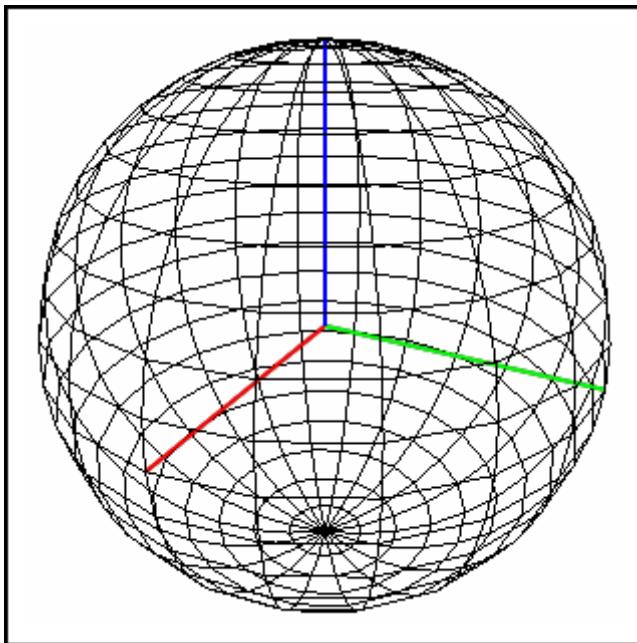
```
gluDeleteQuadric(params);
```

Maintenant nous savons créer un champ de paramètres pour nos quadriques, le paramétrier et le supprimer. Il est temps de voir ce qui nous intéresse réellement, les quadriques et leurs fonctions de dessin !

Les quadriques

Pour bien comprendre quelles sont les faces et vertices générés par les appels suivants, tous les quadriques seront représentés en filaire. J'inclus aussi une version 3D du repère pour montrer que l'axe Z (bleu) est l'axe principal utilisé par les quadriques.

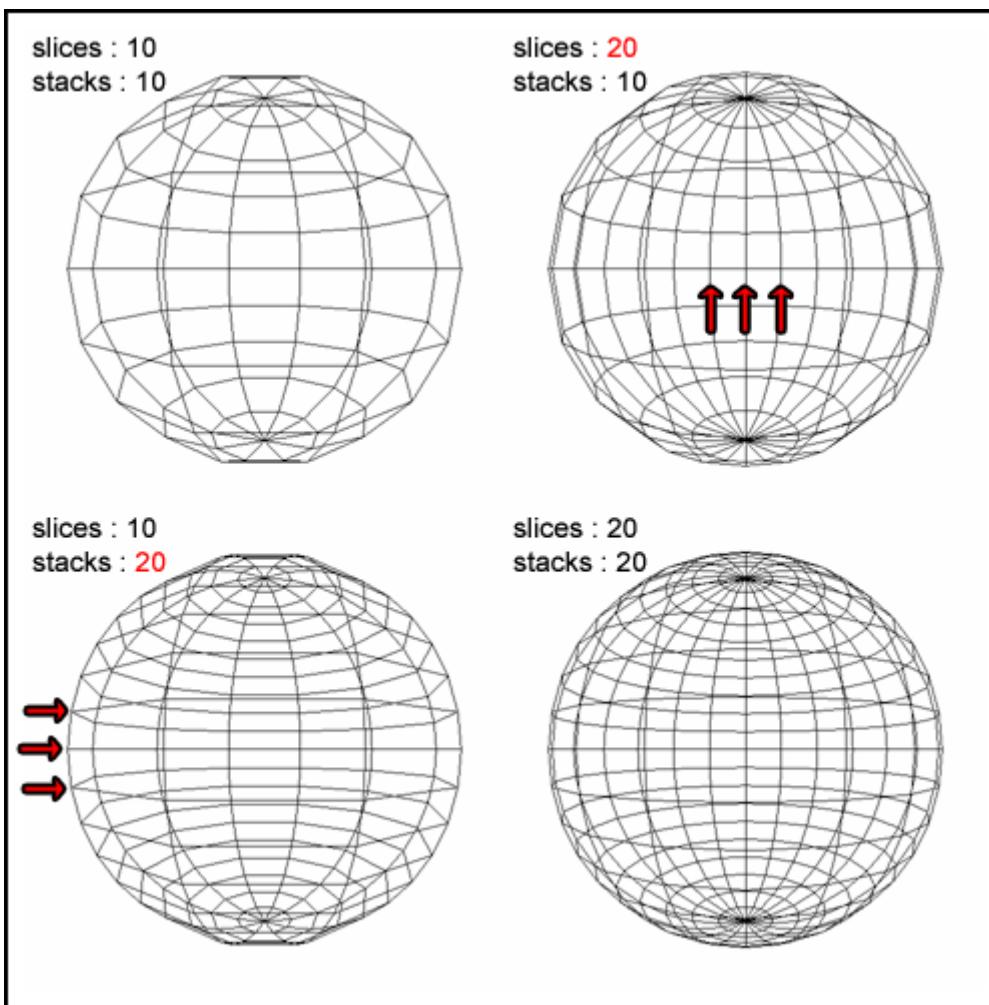
La sphère



```
gluSphere(GLUquadric* params, radius, slices, stacks);
```

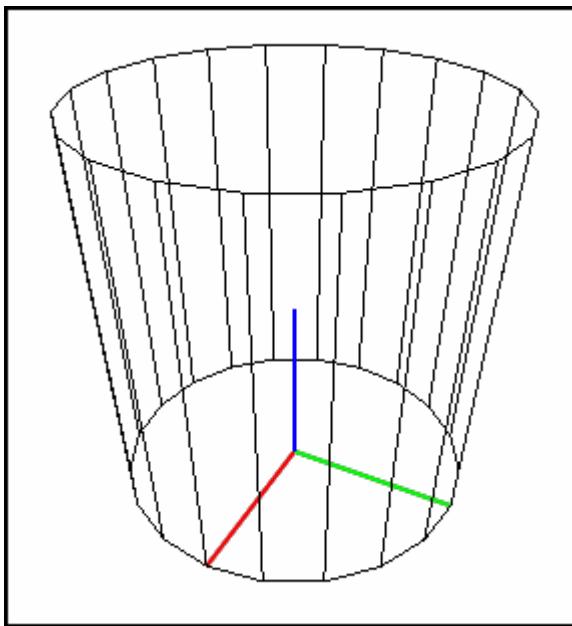
- Le premier paramètre est le champ de type `GLUquadric` que nous avons paramétré précédemment.
- Le deuxième, **radius**, est le plus simple : c'est le **rayon** de la sphère.
- **slices** est le nombre de tranches verticales qui composeront la sphère.
- **stacks** est aussi un nombre de tranches mais pour les tranches horizontales.

L'influence des ces deux paramètres est résumée par l'image ci-dessous :



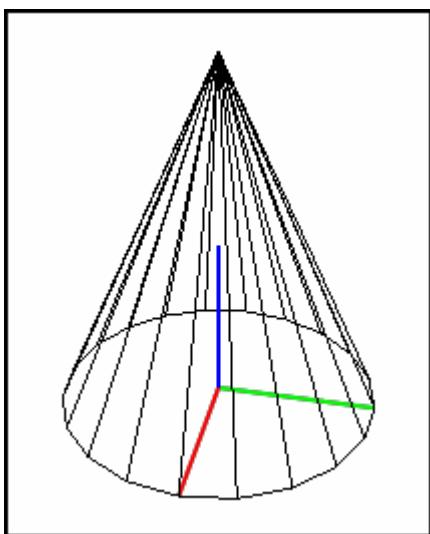
Plus ces deux nombres sont grands, plus la sphère est précise et ressemble en effet à une sphère. La valeur choisie (20x20) donne un résultat satisfaisant.

Le cylindre et le cône



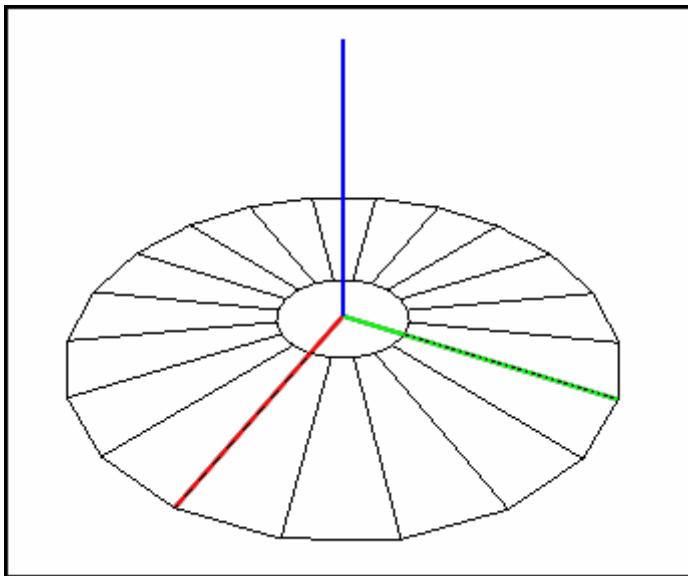
```
gluCylinder(GLUquadric* params,base,top,height,slices,stacks);
```

- Le premier paramètre est toujours le même champ de type GLUquadric.
- **Base** est le rayon du cylindre en bas, **top** est le rayon du cylindre en haut. Pour avoir un vrai cylindre il faut donc utiliser la même valeur pour les 2, mais en utilisant des valeurs différentes pour base et top, nous aurons un cône ! (voir dessin ci-dessous)
- **slices** est, comme pour la sphère, le nombre de divisions autour de l'axe Z et nous choisirons une valeur de l'ordre de 20 pour la même raison.
- **stacks** ici ne sert pas à grand chose. Mettre une valeur différente de 1 ne changerait rien au niveau de la précision du cylindre/cône (à part quand on le regarde en filaire).



```
gluCylinder(params,1,0,2,20,1);
```

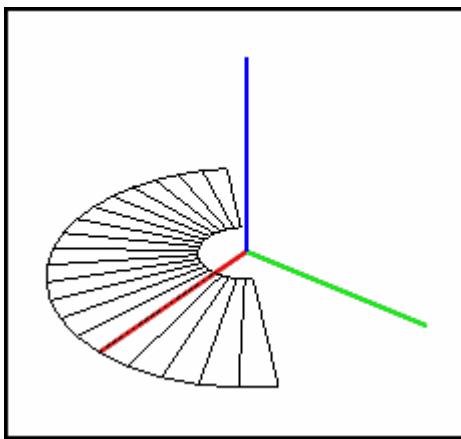
Le disque



```
gluDisk(GLUquadric* params,inner,outer,slices,loops);
```

- Le premier paramètre est toujours le même champ de type GLUquadric.
- **inner** est le rayon interne du disque, souvent à 0 mais peut (comme sur l'image) avoir une valeur différente.
- **outer** est le rayon externe du disque.
- **slices** est, comme précédemment, le nombre de divisions autour de l'axe Z.
- **loops** ici ne sert pas à grand chose. Mettre une valeur différente de 1 rajoutera des faces à l'intérieur du disque mais ne rajoute pas de précision visible (à part en filaire).

Le disque partiel



```
gluPartialDisk(GLUquadric* params,inner,outer,slices,loops,start,sweep);
```

Le disque partiel est comme le disque normal sauf qu'il ne fait pas nécessairement 360°.

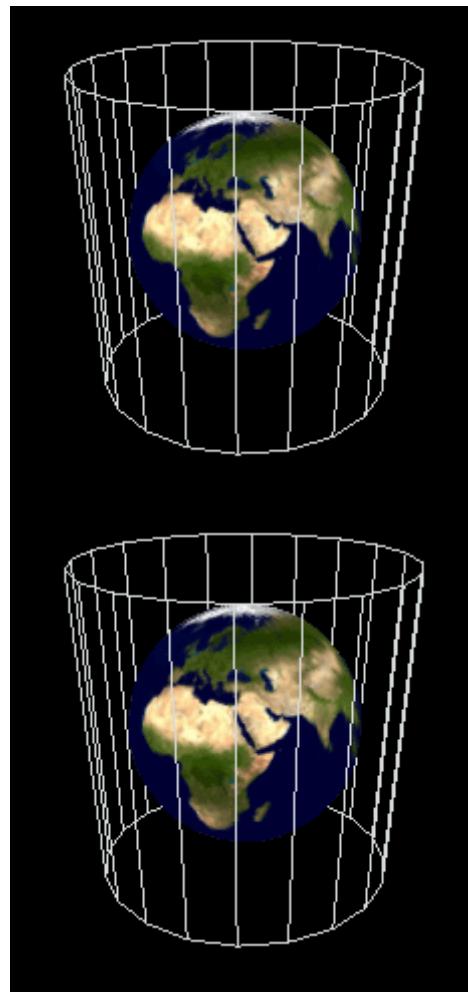
- **start** est l'angle de départ du disque partiel. Malheureusement pour nous, les concepteurs d'OpenGL n'ont pas suivi la logique mathématique qui voudrait que les angles soient exprimés dans le sens trigonométrique (sens inverse des aiguilles d'une montre) avec 0° sur l'axe X. Ici 0° pour start place le début du disque sur l'axe Y (vert), 90° sur l'axe X (rouge).
- **sweep** est la distance angulaire entre le début et la fin du disque partiel (180° sur le dessin plus haut).

La précision donnée par slices s'applique ici au disque partiel uniquement. Cela ne sert donc à rien de mettre une grande valeur (20) si on utilise un disque partiel de 90° seulement. Une précision de 5 peut suffire à avoir le même résultat de qualité sur un angle de 90° (par rapport à une précision de 20 pour un angle de 360°).

Un même GLUquadric pour dessiner plusieurs quadriques

Vous l'avez compris maintenant, l'objet *GLUquadric* n'est pas un quadrique mais simplement un champ de paramètres utilisé lors de l'appel d'une fonction de dessin de quadrique pour spécifier le mode d'affichage. On peut donc tout à fait utiliser *le même GLUquadric* pour faire dessiner des quadriques tout en changeant, si on le souhaite, les paramètres en cours de route :

```
glBindTexture(GL_TEXTURE_2D, texture  
1);  
  
    GLUquadric* params = gluNewQuad  
ric();  
  
    gluQuadricDrawStyle(params,GLU_  
LINE);  
    gluCylinder(params,1,1,2,20,1);  
  
    gluQuadricDrawStyle(params,GLU_  
FILL);  
    gluQuadricTexture(params,GL_TRU  
E);  
    glTranslated(0,0,1);  
    gluSphere(params,0.75,20,20);  
  
    gluDeleteQuadric(params);
```



Exercice : une roquette

Ce chapitre n'a rien de compliqué mais jusqu'à présent j'ai fait tout le boulot en vous détaillant les fonctions pour utiliser les quadriques. Maintenant à vous ! :diable:

Pour vous faire la main sur ces nouvelles fonctions je vous propose de créer **une roquette**, basée sur celles que l'on trouve dans le jeu Half-Life premier du nom. Mon but n'est pas de faire de vous des apprentis terroristes mais juste des pros des quadriques ! :ange:



Une roquette inspirée d'Half-Life

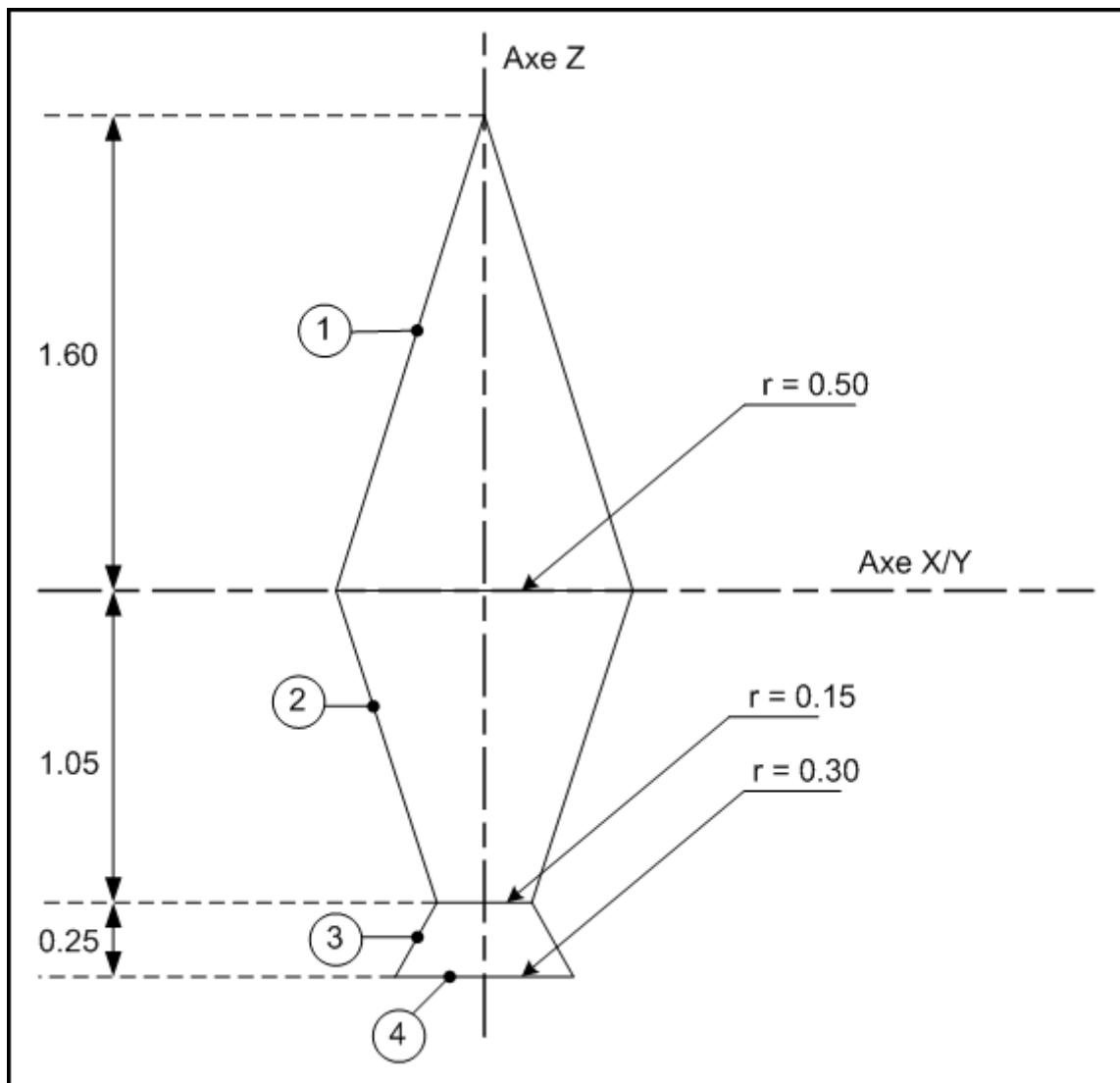
Les textures

Les textures sont elles aussi tirées d'Half-Life et légèrement modifiées par mes soins. Vous les trouverez dans le pack ci-dessous (et dans le zip final).

Téléchargez les textures pour la roquette (4.53 Ko)

(http://sdz.tdct.org/sdz/medias/www.siteduzero.com_uploads_fr_ftp_kayl_rocket_textures.zip)

Schéma de la roquette



Comme vous pouvez le voir, la roquette est constituée de quatre éléments :

1. un cône supérieur (texture rocket_top.jpg) ;
2. un cône intermédiaire (texture rocket_middle.jpg) ;
3. un cône inférieur (texture rocket_bottom.jpg) ;
4. et enfin, ça ne se voit pas trop sur le schéma, un disque (texture rocket_motor.jpg).

Méthode pour la coder

Pour coder la roquette il nous faut donc procéder en deux étapes :

- au lancement du programme il faut : charger toutes les textures utilisées ;
- au moment de dessiner il faut :
 1. créer un GLUquadric ;
 2. paramétriser le quadrique pour qu'il génère les coordonnées de texture automatiquement ;
 3. dessiner le premier objet ;
 4. se translater à la base du deuxième objet ;
 5. dessiner le deuxième objet ;
 6. etc.
 7. détruire le GLUquadric.

La phase *se translate* est importante. Comme nous l'avons vu sur les schémas des différents types de quadriques, ils sont toujours dessinés à partir de (0,0,0) dans le **repère local**. En utilisant intelligemment les transformations, il est donc possible de placer chaque quadrique où on le souhaite.

N'oubliez pas non plus de changer de texture entre chaque quadrique pour ne pas vous retrouver avec une roquette uniformément... moche. ;)

Voilà vous avez tous les outils pour dessiner cette roquette. Référez-vous bien au schéma que je vous donne pour respecter les proportions. Toute roquette déformée ne sera pas acceptée pour une utilisation sur le terrain ! (Hum...)

À vous donc !

Correction

Je ne vous mets ici que le code intéressant. Je pars du principe que vous savez parfaitement charger les textures et initialiser l'application. Je fournis bien entendu le code complet dans l'archive finale.

```

/* J'ai choisi de faire une fonction Dessiner Rocket.
Je pourrai ainsi l'appeler plusieurs fois, et dans n'importe quelle
position initiale du repère initial */
void DrawRocket()
{
    glPushMatrix(); //pour que les transformations soient réversibles

    GLUquadric* params = gluNewQuadric(); //création du quadrique
    gluQuadricTexture(params,GL_TRUE); //activation des coordonnées de texture

    glBindTexture(GL_TEXTURE_2D,top); //texture du haut
    gluCylinder(params,0.5,0,1.6,20,1); //cône 1

    glBindTexture(GL_TEXTURE_2D,middle);
    glTranslated(0,0,-1.05); //je descends pour faire le 2ème cône
    gluCylinder(params,0.15,0.5,1.05,20,1); //cône 2

    glBindTexture(GL_TEXTURE_2D,bottom);
    glTranslated(0,0,-0.25); //je descends enfin tout en bas (sur le schéma)
    gluCylinder(params,0.3,0.15,0.25,20,1); //cône 3

    //et à la même position je dessine le disque de sortie des flammes
    glBindTexture(GL_TEXTURE_2D,motor);
    gluDisk(params,0,0.3,20,1); //disque 4

    gluDeleteQuadric(params); //je supprime le quadrique

    glPopMatrix(); //hop je remets tout comme je l'ai trouvé
}

void DrawGL()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt(3,4,2,0,0,0,0,0,1); //je place la caméra à un endroit idéal

    DrawRocket(); //je dessine la 1ère roquette

    glTranslated(2,0,0); //je me déplace pour la 2ème roquette
    glRotated(90,1,0,0); /*je vais tourner celle-là pour que son axe principal
soit horizontal */
    DrawRocket(); //et je la dessine

    glFlush();
    SDL_GL_SwapBuffers();
}

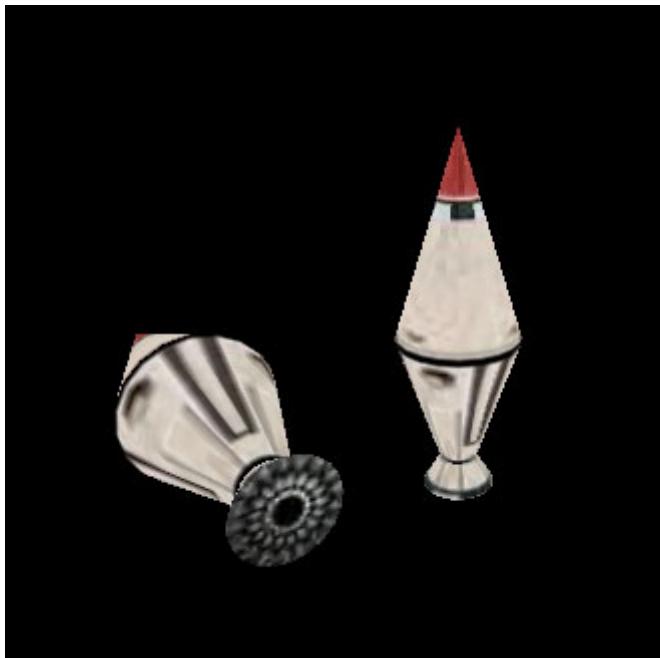
```

Comme vous le voyez j'ai décidé personnellement de faire une fonction (<http://www.siteduzero.com/tuto-3-2842-1-les-fonctions.html>) pour dessiner la roquette. Je peux ainsi dessiner autant de roquettes que je veux sans alourdir le code.

Améliorations

Vous pouvez, si vous le souhaitez, créer une sphère représentant la Terre (avec la texture EarthLow.jpg du pack final) autour de laquelle la roquette tournerait.

Conseil : pour faire tourner la roquette autour de la Terre il suffit de bien réfléchir à l'ordre des transformations à faire. Comme ce n'est pas le but de cet exercice, qui se veut simple et rapide, je vous laisse imaginer la solution adéquate.



Téléchargez le projet Code::Blocks, l'exécutable Windows et le Makefile Unix (391 Ko)
(http://sdz.tdct.org/sdz/médias/www.siteduzero.com_uploads_fr_ftp_kayl_sdlgl_07_roquette.zip)

Les quadriques, c'est fantastique et c'est magique ! Loin d'être mystiques elles sont quand même vachement pratiques ! :soleil:

C'est une solution très simple pour ajouter des objets un peu complexes dans vos scènes en attendant de savoir charger de vrais modèles 3D créés dans des logiciels 3D externes (Blender, 3dSmax...).

Dans le prochain chapitre nous verrons comment contrôler la caméra de manière plus poussée, et réaliserons entre autres un dérivé de Google Earth sans prétention qui utilisera justement la sphère. Que de bonheur en perspective !

Contrôle avancé de la caméra (Partie 1/2)

Ce chapitre en deux parties vient présenter comment créer des caméras contrôlables dans vos applications OpenGL. Fini donc le calvaire de prévoir précisément dans le code la position/orientation de la caméra.

Nous commencerons dans ce chapitre par une caméra **Trackball**, assez simple à implémenter et qui nous permettra d'introduire le concept de classes en C++.

Principe d'une caméra TrackBall

Le nom **TrackBall** vient de ce périphérique bizarre qui remplace la souris où l'on tourne directement une boule. Ici j'ai librement tiré le terme du logiciel multiplate-forme Google Earth :



Google Earth

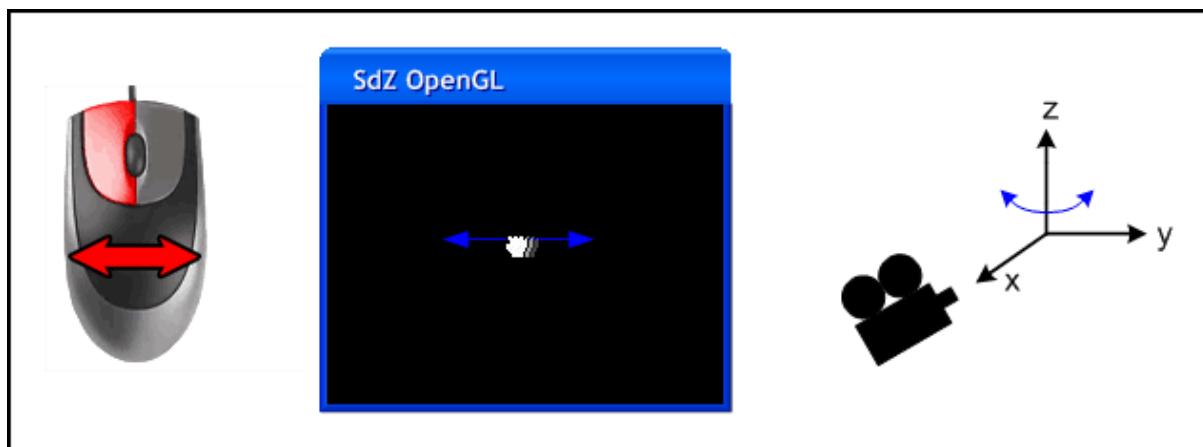
Dans Google Earth (<http://www.commentcamarche.net/download/telecharger-34055136-google-earth>) en effet on utilise la souris pour tourner autour de la Terre. Nous allons donc reproduire ce principe qui nous permettra d'avoir une caméra permettant de regarder un objet / une partie d'une scène sous tous les angles.

Rotation à la souris

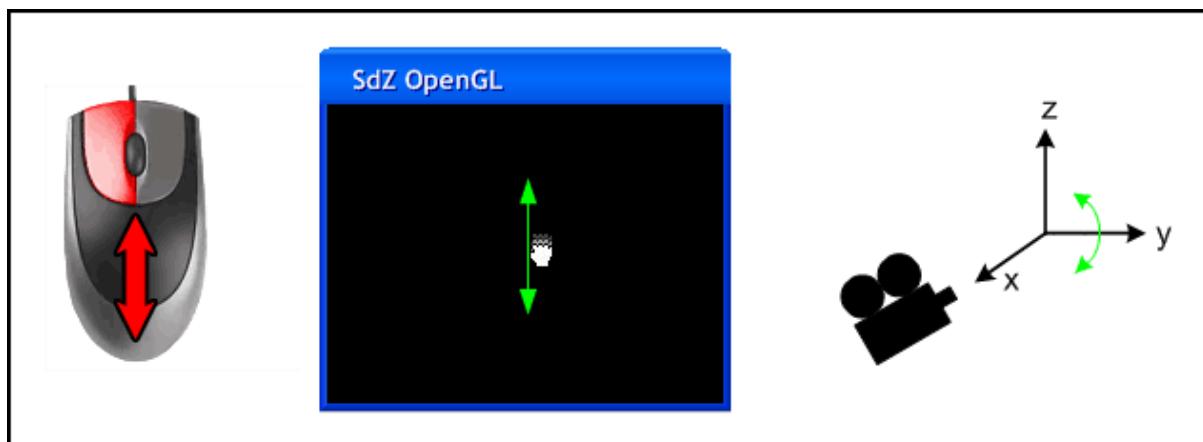
En maintenant le bouton gauche de la souris enfoncé, les mouvements de la souris feront tourner la scène :

- un mouvement horizontal de la souris donne une rotation horizontale de la scène (donc autour de sa verticale).
- un mouvement vertical de la souris donne une rotation verticale de la scène.

Ces mouvements sont illustrés par les schémas ci-dessous :



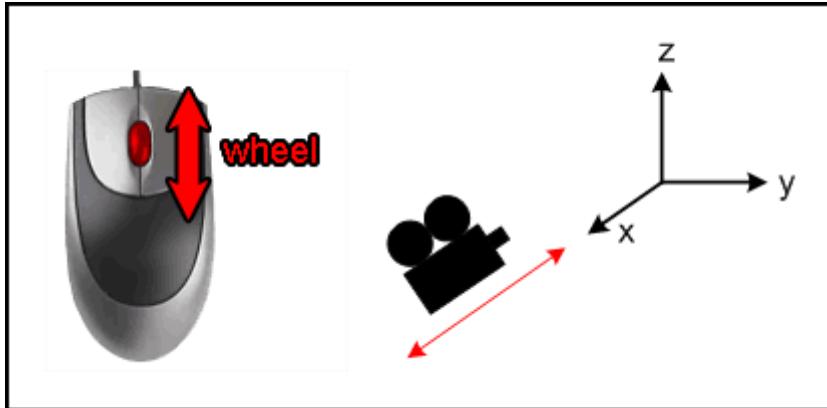
Mouvement horizontal de la souris



Notez au passage, et nous verrons l'équivalent dans le code, que ce n'est pas vraiment la caméra qui tourne mais la **scène**, même si cela revient plus ou moins au même. Ce n'est le cas que pour ce type de caméra. Pour la caméra FreeFly que nous verrons après c'est bel et bien la caméra et non la scène qui bougera.

Zoom à la molette

Pour prendre du recul ou au contraire nous rapprocher de l'objet / scène que nous souhaitons visualiser, nous allons tout simplement utiliser la molette. Un coup de molette en avant pour zoomer, un coup de molette en arrière pour dézoomer, rien de plus intuitif :



Rotation de la roulette de la souris

Ici c'est bien la caméra qui bouge, nous verrons une fois encore comment cela se répercute sur le code.

Et le clavier ?

Il est possible d'arguer que toutes les souris ne possèdent pas de molette. Dans ce cas-là rien ne vous empêche d'utiliser le clavier pour dézoomer.

Ici nous n'utiliserons le clavier que pour une chose : réinitialiser la rotation de la scène avec la touche



(SDLK_HOME avec SDL).

Quelques bases de C++

Maintenant que nous savons ce que nous voulons faire avec notre caméra, il faut faire un petit intermède apprentissage du C++.

Nous allons en effet utiliser et regrouper toutes les fonctionnalités de notre caméra dans une **classe** : TrackBallCamera.

Le cours de M@teo expliquera le concept des classes en détail. Voyons pour l'instant ça comme une extension d'une **structure** (<http://www.siteduzero.com/tuto-3-4350-1-creez-vos-propres-types-de-variables.html>).

Rappelez-vous en C un struct permettait de stocker plusieurs champs dans un même type :

```

struct NomDeVotreStructure
{
    long variable1;
    long variable2;
    int autreVariable;
    double nombreDecimal;
};

```

Une classe possède, en plus des **attributs**, des **méthodes**. Ces méthodes sont comme des **fonctions** qui s'appliquent aux **instances** de cette classe.

Ouh là là beaucoup de mots nouveaux ! Instances par exemple c'est quoi ?

Imaginons une classe nommée Chaise. Une chaise a certains attributs : hauteur, nombre de pieds, matière. On écrira donc :

```

class Chaise
{
protected:
    int hauteur;
    int nombre_de_pieds;
    string matiere;
}

```

Une fois la classe déclarée, dans le code du programme on veut pouvoir l'utiliser (des chaises). On crée donc des « instances » de la classe « Chaise » en déclarant simplement une variable de type Chaise.

Exemple :

```
Chaise maChaise;
```

Notez qu'en C++ les structures et les classes sont automatiquement des types. Nul besoin donc d'écrire un `typedef Class Chaise Chaise;` par exemple.

Vous avez pu voir un mot bizarre dans mon exemple : **protected**. Sans rentrer dans le détail, cela veut dire que les attributs déclarés protected ne sont pas accessibles de l'extérieur de la classe mais uniquement par ses méthodes.

Les méthodes justement c'est quoi ?

Une méthode est comme une fonction mais elle s'applique à une instance précise de la classe.

Reprenons notre exemple de la chaise. Imaginons que nous voulions enlever un pied à notre chaise.

En C nous aurions dû utiliser une fonction enleverPied en passant en paramètre quelle chaise modifier.

En C++ on appelle directement une méthode sur une instance de la classe.

Exemple :

La déclaration de la classe Chaise dans Chaise.h

```

class Chaise
{
public:
    void enleverPied();
protected:
    int hauteur;
    int nombre_de_pieds;
    string matiere;
};

```

L'implémentation des méthodes de la classe Chaise dans Chaise.cpp

```

#include "chaise.h"

void Chaise::enleverPied()
{
    nombre_de_pieds--;
}

```

Appel dans le corps du programme

Et maintenant ce qui nous intéresse, l'appel de la méthode enleverPied sur une instance :

```

Chaise machaise;
machaise.enleverPied();

```

Comme vous le voyez on appelle une méthode comme on utiliserait un attribut : instance.laméthode();
Quand le programme entre dans le code de la méthode il l'applique donc à l'instance souhaitée, et utilise donc les attributs propres à l'instance en question.

Notez qu'ici je mets la méthode en public et non protected pour pouvoir l'appeler de l'extérieur de la classe (c'est-à-dire à partir du corps du programme).

Deux méthodes particulières

Il existe deux méthodes particulières qui ne sont pas appelées directement par l'utilisateur : le constructeur et le destructeur.

Le **constructeur** est appelé lorsque l'objet est initialisé, généralement à sa déclaration.
C'est une méthode sans type de retour, qui porte le nom de la classe, et qui permet d'initialiser les attributs à des valeurs initiales :

Exemple :

Déclaration du constructeur dans Chaise.h

```

Class Chaise
{
public:
    Chaise(); //un constructeur ne renvoie rien mais peut éventuellement avoir des paramètres
    void enleverPied();
protected:
    int hauteur;
    int nombre_de_pieds;
    string matiere;
};

```

Implémentation du constructeur dans Chaise.cpp

```
Chaise::Chaise()
{
    hauteur = 1;
    nombre_de_pieds = 4;
    matière = "bois";
}
```

Et donc ce constructeur sera appelé dès qu'on instanciera un objet dans le corps principal du programme :

```
Chaise machaise; //déclenche l'appel du constructeur
machaise.enleverPied(); //je sais donc que maintenant elle en a 3 car une chaise a 4 pieds au départ
grâce au constructeur
```

Le **destructeur** quant à lui est appelé automatiquement quand on détruit l'objet. Dans le cas présent je n'ai rien de spécial à faire dans le destructeur, mais si nous avions alloué de la mémoire dynamiquement (attributs dynamiques de la classe), c'est dans le destructeur qu'il faut les détruire pour ne pas faire de fuite de mémoire. Comme le constructeur, le destructeur porte le nom de la classe précédé du symbole « ~ ». Ici je vais me contenter d'afficher un message lors de la destruction :

Déclaration du destructeur dans Chaise.h

```
Class Chaise
{
public:
    Chaise(); //un constructeur ne renvoie rien mais peut éventuellement avoir des arguments
    void enleverPied();
    ~Chaise(); //un destructeur ne renvoie rien, n'a pas d'arguments, et se précède du symbole ~
protected:
    int hauteur;
    int nombre_de_pieds;
    string matière;
};
```

Implémentation du destructeur dans Chaise.cpp

```
#include <iostream>
...
Chaise::~Chaise()
{
    std::cout << "Au revoir petite chaise." << std::endl;
}
```

Dans le corps du programme l'appel au destructeur est automatique à la fin du bloc où l'instance est déclarée.
Exemple :

```
int main()
{
    Chaise machaise; //appel du constructeur
    machaise.enleverPied(); //la pauvre ça doit faire mal

    return 0; //on quitte le bloc du main, donc on détruit toutes les variables -> appel automatique du destructeur de Chaise sur l'instance machaise.
}
```

Allocation dynamique

Si vous en êtes à la lecture du tuto OpenGL c'est que vous connaissez sûrement l'allocation dynamique (<http://www.siteduzero.com/tuto-3-4830-1-l-allocation-dynamique.html>) en C :

```
struct Chaise * machaise;
machaise = malloc(sizeof(struct Chaise));
```

En C++ on utilise généralement l'opérateur **new** comme ceci :

```
Chaise * machaise;
machaise = new Chaise();
```

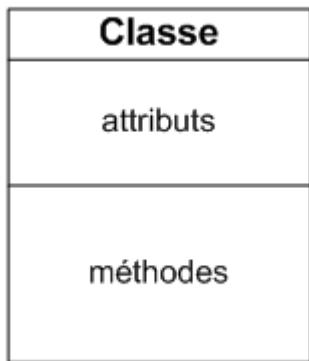
On note ici l'utilisation des () après Chaise qui montre clairement qu'on cherche à construire un objet. Une fois la mémoire allouée, le constructeur de la classe est donc automatiquement appelé.

Pour la destruction, **delete** remplace le free que vous connaissez :

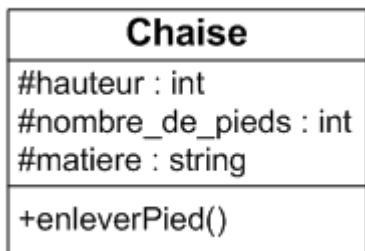
```
Chaise * machaise;
machaise = new Chaise();
machaise->enleverPied();
delete machaise;
```

Représentation UML

Je ne vais pas vous faire un cours de modélisation UML mais juste vous présenter une manière graphique de représenter une classe. J'utiliserai ce symbolisme tout au long du tuto pour résumer brièvement les fonctionnalités d'une classe :



Ce qui donne par exemple pour reprendre notre chère chaise :



Implémentation de la caméra

Nous allons implémenter la caméra TrackBall avec le concept de classe que nous venons de voir.

Nous l'avons vu plus haut il nous faut gérer trois types d'événements :

- l'appui sur le bouton gauche de la souris : nous n'activerons le mouvement à la souris que si ce bouton est enfoncé ;
- le mouvement de la souris : pour changer l'orientation de la scène ;
- l'appui sur la touche HOME pour remettre l'orientation de la scène à sa valeur initiale.

Dans le corps principal de notre programme SDL (partie suivante) nous devrons donc envoyer les événements nécessaires au fonctionnement de la caméra.

Vous savez comment placer une caméra manuellement avec gluLookAt. Ici c'est la méthode look de notre classe TrackBallCamera qui s'occupera d'appeler le gluLookAt pour nous. Dans le code d'affichage de la scène nous n'aurons donc qu'à appeler cette méthode.

Nous allons aussi rajouter deux autres méthodes pour configurer la sensibilité de notre caméra :

- setMotionSensitivity : pour déterminer la vitesse de rotation de la scène en fonction du mouvement en pixel du curseur de la souris ;
- setScrollSensitivity : pour déterminer de combien zoomer/dézoomer lorsque l'on utilise la molette de la souris.

Tout cela se traduit donc de la façon suivante en UML et C++ :

UML simplifié

TrackBallCamera
#_motionSensitivity : double #_scrollSensitivity : double #_hold : bool #_distance : double #_angleY : double #_angleZ : double #_hand1 : SDL_Cursor * #_hand2 : SDL_Cursor *
+OnMouseMove(SDL_MouseMotionEvent) +OnMouseButtonDown(SDL_MouseButtonEvent) +OnKeyboard(SDL_KeyboardEvent) +look() +setMotionSensitivity(double) +setScrollSensitivity(double)

TrackBallCamera
#_motionSensitivity : double #_scrollSensitivity : double #_hold : bool #_distance : double #_angleY : double #_angleZ : double #_hand1 : SDL_Cursor * #_hand2 : SDL_Cursor *
+OnMouseMove(SDL_MouseMotionEvent) +OnMouseButtonDown(SDL_MouseButtonEvent) +OnKeyboard(SDL_KeyboardEvent) +look() +setMotionSensitivity(double) +setScrollSensitivity(double)

Déclaration C++

```
class TrackBallCamera
{
public:
    TrackBallCamera();

    virtual void OnMouseMove(const
SDL_MouseMotionEvent & event);
    virtual void OnMouseButtonDown(const
SDL_MouseButtonEvent & event);
    virtual void OnKeyboard(const SD
L_KeyboardEvent & event);

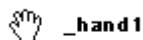
    virtual void look();
    virtual void setMotionSensitivity
(double sensitivity);
    virtual void setScrollSensitivity
(double sensitivity);

    virtual ~TrackBallCamera();
protected:
    double _motionSensitivity;
    double _scrollSensitivity;
    bool _hold;
    double _distance;
    double _angleY;
    double _angleZ;
    SDL_Cursor * _hand1;
    SDL_Cursor * _hand2;
};
```

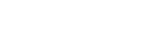
J'en ai profité pour rajouter tous les attributs que nous allons utiliser. Une petite explication s'impose donc :

- **double _motionSensitivity** : utilisé pour stocker la sensibilité de la caméra aux mouvements de la souris ;
- **double _scrollSensitivity** : sensibilité de la caméra au scroll de la souris (« pas » d'un déplacement) ;
- **bool _hold** : est-on actuellement en train de maintenir le bouton gauche de la souris enfoncé ?
- **double _distance** : distance entre la caméra et le centre de la scène ;
- **double _angleY** : angle de rotation verticale de la scène (en vert sur le schéma plus haut) ;
- **double _angleZ** : angle de rotation horizontale de la scène (donc autour de la verticale, en bleu sur le schéma).

Les deux derniers attributs sont les deux curseurs de la souris que nous utiliserons :



_hand1



_hand2

_hand1 en temps normal, _hand2 quand le bouton gauche de la souris est enfoncé.

Vous remarquerez au passage que je précède les attributs de la classe du symbole underscore « _ ». Cela permet dans l'implémentation des méthodes de distinguer plus rapidement variables temporaires (ou paramètres) et attributs. Vous n'êtes bien sûr pas forcés de suivre cette règle.

Constructeur

Dans le constructeur nous allons simplement initialiser tous les attributs à des valeurs initiales connues. Il ne faut rien laisser qui puisse être utilisé sans avoir été initialisé.

La partie la moins évidente est peut-être la création des deux curseurs. Pour faciliter les choses j'ai relégué tout le travail dans une fonction rajoutée à *sdlglutils* : cursorFromXPM (fournie dans l'archive finale).


```

        "      X.....X  ",
        "      X.....X  ",
        "0,0"
    };

    _hand1 = cursorFromXPM(hand1); //création du curseur normal
    _hand2 = cursorFromXPM(hand2); //création du curseur utilisé quand le bouton est enfoncé
    SDL_SetCursor(_hand1); //activation du curseur normal
    _hold = false; //au départ on part du principe que le bouton n'est pas maintenu
    _angleY = 0;
    _angleZ = 0;
    _distance = 2; //distance initiale de la caméra avec le centre de la scène
    _motionSensitivity = 0.3;
    _scrollSensitivity = 1;
}

}

```

Comme vous pouvez le voir, le constructeur fait appel à une fonction SDL, `SDL_SetCursor`, qui ne doit pas être exécutée avant la création de votre fenêtre SDL. De ce fait la caméra ne devra être créée qu'après l'initialisation de l'application.

On Mouse Motion

Cette méthode est la plus importante de la classe et pourtant l'une des plus courtes. Rappelez-vous le principe de la caméra : lorsque le curseur de la souris est bougé, si le bouton gauche de la souris est maintenu appuyé, alors la scène tourne. Voyons donc comment cela se traduit en code :

```

void TrackBallCamera::OnMouseMove(const SDL_MouseMotionEvent & event)
{
    if (_hold) //si nous maintenons le bouton gauche enfoncé
    {
        _angleZ += event.xrel*_motionSensitivity; //mouvement sur X de la souris -> changement de la rotation horizontale
        _angleY += event.yrel*_motionSensitivity; //mouvement sur Y de la souris -> changement de la rotation verticale
        //pour éviter certains problèmes, on limite la rotation verticale à des angles entre -90° et 90°
        if (_angleY > 90)
            _angleY = 90;
        else if (_angleY < -90)
            _angleY = -90;
    }
}

```

On Mouse Button

Cette méthode nous permet de gérer deux choses :

- l'appui et le relâchement du bouton gauche de la souris ;
- le mouvement de la molette de la souris.

Lorsque l'on bouge la molette, deux événements successifs sont générés : `SDL_MOUSEBUTTONDOWN` et `SDL_MOUSEBUTTONUP`. Nous n'utiliserons donc que le premier.

```

void TrackBallCamera::OnMouseButton(const SDL_MouseButtonEvent & event)
{
    if (event.button == SDL_BUTTON_LEFT) //l'événement concerne le bouton gauche
    {
        if (( _hold)&&(event.type == SDL_MOUSEBUTTONUP)) //relâchement alors qu'on était enfoncé
        {
            _hold = false; //le mouvement de la souris ne fera plus bouger la scène
            SDL_SetCursor(_hand1); //on met le curseur normal
        }
        else if (( !_hold)&&(event.type == SDL_MOUSEBUTTONDOWN)) //appui alors qu'on était relâché
        {
            _hold = true; //le mouvement de la souris fera bouger la scène
            SDL_SetCursor(_hand2); //on met le curseur spécial
        }
    }
    else if ((event.button == SDL_BUTTON_WHEELUP)&&(event.type == SDL_MOUSEBUTTONDOWN)) //coup de molette vers le haut
    {
        _distance -= _scrollSensitivity; //on zoome, donc rapproche la caméra du centre
        if (_distance < 0.1) //distance minimale, à changer si besoin (avec un attribut par exemple)
            _distance = 0.1;
    }
    else if ((event.button == SDL_BUTTON_WHEELDOWN)&&(event.type == SDL_MOUSEBUTTONDOWN)) //coup de molette vers le bas
    {
        _distance += _scrollSensitivity; //on dézoome donc éloigne la caméra
    }
}

```

OnKeyboard

La dernière méthode qui vient utiliser les événements est la gestion du clavier, pour l'appui sur la touche HOME. On se contente d'y remettre la rotation de la scène à zéro :

```

void TrackBallCamera::OnKeyboard(const SDL_KeyboardEvent & event)
{
    if ((event.type == SDL_KEYDOWN)&&(event.keysym.sym == SDLK_HOME)) //appui sur la touche HOME
    {
        _angleY = 0; //remise à zéro des angles
        _angleZ = 0;
    }
}

```

Look

Tout cela est bien beau, nous savons comment changer des variables avec la souris et le clavier mais ça ne fait en rien bouger la caméra dans notre scène. En effet nous n'avons pour l'instant pas vu la moindre commande OpenGL ! Il est donc temps de s'y mettre avec la méthode Look qui viendra remplacer, dans votre fonction d'affichage, l'appel à gluLookAt.

Remplacer gluLookAt ? Parce qu'il y a un truc mieux que tu nous as caché !! ?

Non non. La méthode Look appelle elle-même gluLookAt mais avec des paramètres qui dépendent de la position de la caméra, c'est pour ça que vous n'avez plus à l'appeler vous-mêmes.

Si on se réfère aux schémas en début de chapitre qui expliquent le principe de la caméra TrackBall, on remarque plusieurs choses :

- elle regarde le centre de la scène ;
- ce n'est pas la caméra mais la scène qui est tournée autour de Y et Z.

Il suffit alors de traduire tout ça en code :

```
void TrackBallCamera::look()
{
    gluLookAt(_distance,0,0,
              0,0,0,
              0,0,1); // la caméra regarde le centre (0,0,0) et est sur l'axe X à une certaine distance du centre donc (_distance,0,0)
    glRotated(_angleY,0,1,0); //la scène est tournée autour de l'axe Y
    glRotated(_angleZ,0,0,1); //la scène est tournée autour de l'axe Z
}
```

Et voilà ce n'était vraiment pas sorcier.

La dernière chose qu'il nous reste à faire dans le code même de la caméra est une destruction propre de ce qui a été alloué dynamiquement.

Destructeur

Les seules choses allouées dynamiquement sont les curseurs qu'il nous faut détruire quand la caméra est détruite :

```
TrackBallCamera::~TrackBallCamera()
{
    SDL_FreeCursor(_hand1); //destruction du curseur normal
    SDL_FreeCursor(_hand2); //destruction du curseur spécial
    SDL_SetCursor(NULL); //on remet le curseur par défaut.
}
```

Scène de test

Le code de la classe TrackBallCamera est complet et n'a besoin de rien de plus. Cependant un objet camera ne va pas recevoir tout seul les événements, il faut les lui donner. Nous allons donc voir avec une petite scène de test simple comment utiliser la caméra que nous venons de créer. Pour faire original et pas du tout inspiré de Google Earth, nous allons créer une sphère avec la texture de la Terre. Hum hum ! :-°

En variables globales nous allons donc utiliser :

```
GLuint earth; //l'identifiant de la texture de la Terre
TrackBallCamera * camera; //un pointeur vers notre caméra
```

Pourquoi pas directement une caméra ?

Rappelez-vous, le constructeur appelle des fonctions SDL qui nécessitent qu'une fenêtre SDL existe déjà. Il ne faut donc pas que la caméra soit construite dès le lancement du programme (ce qui serait le cas ici si nous n'utilisions pas de pointeur). Nous la créons donc dynamiquement après la fenêtre :

```

atexit(stop); //stop() sera appelé quand on fera exit(0);
//...
    SDL_SetVideoMode(width, height, 32, SDL_OPENGL);
//...
    earth = loadTexture("EarthMap.jpg");
    camera = new TrackBallCamera();
    camera->setScrollSensitivity(0.1);

```

Comme la caméra a été créée dynamiquement c'est à nous de la détruire proprement à la fin de l'exécution du programme. C'est ce qu'on fait dans la fonction stop, appelée quand on fera exit(0); dans le corps du programme :

```

void stop()
{
    delete camera; //destruction de la caméra allouée dynamiquement
    SDL_Quit();
}

```

Comme je vous l'ai dit plus haut, la caméra ne recevra pas les événements clavier/souris si on ne les lui donne pas. C'est pourquoi dans notre partie de gestion des événements, il faut donner à la caméra les événements dont on ne se sert pas :

```

while(SDL_PollEvent(&event))
{
    switch(event.type)
    {
        case SDL_QUIT:
            exit(0);
            break;
        case SDL_KEYDOWN:
            switch (event.key.keysym.sym)
            {
                case SDLK_p:
                    takeScreenshot("test.bmp");
                    break;
                case SDLK_ESCAPE:
                    exit(0);
                    break;
                default : //on a utilisé la touche P et la touche ECHAP, le reste est donné à la
caméra
                    camera->OnKeyboard(event.key);
            }
            break;
        case SDL_MOUSEMOTION: //la souris est bougée, ça n'intéresse que la caméra
            camera->OnMouseMove(event.motion);
            break;
        case SDL_MOUSEBUTTONUP:
        case SDL_MOUSEBUTTONDOWN:
            camera->OnMouseButton(event.button); //tous les événements boutons (up ou down) sont
donnés à la caméra
            break;
    }
}

```

La dernière chose qu'il nous reste à faire est de dessiner la scène, ici très basique.

On n'appelle plus gluLookAt nous-mêmes mais bien la méthode Look de notre objet camera.

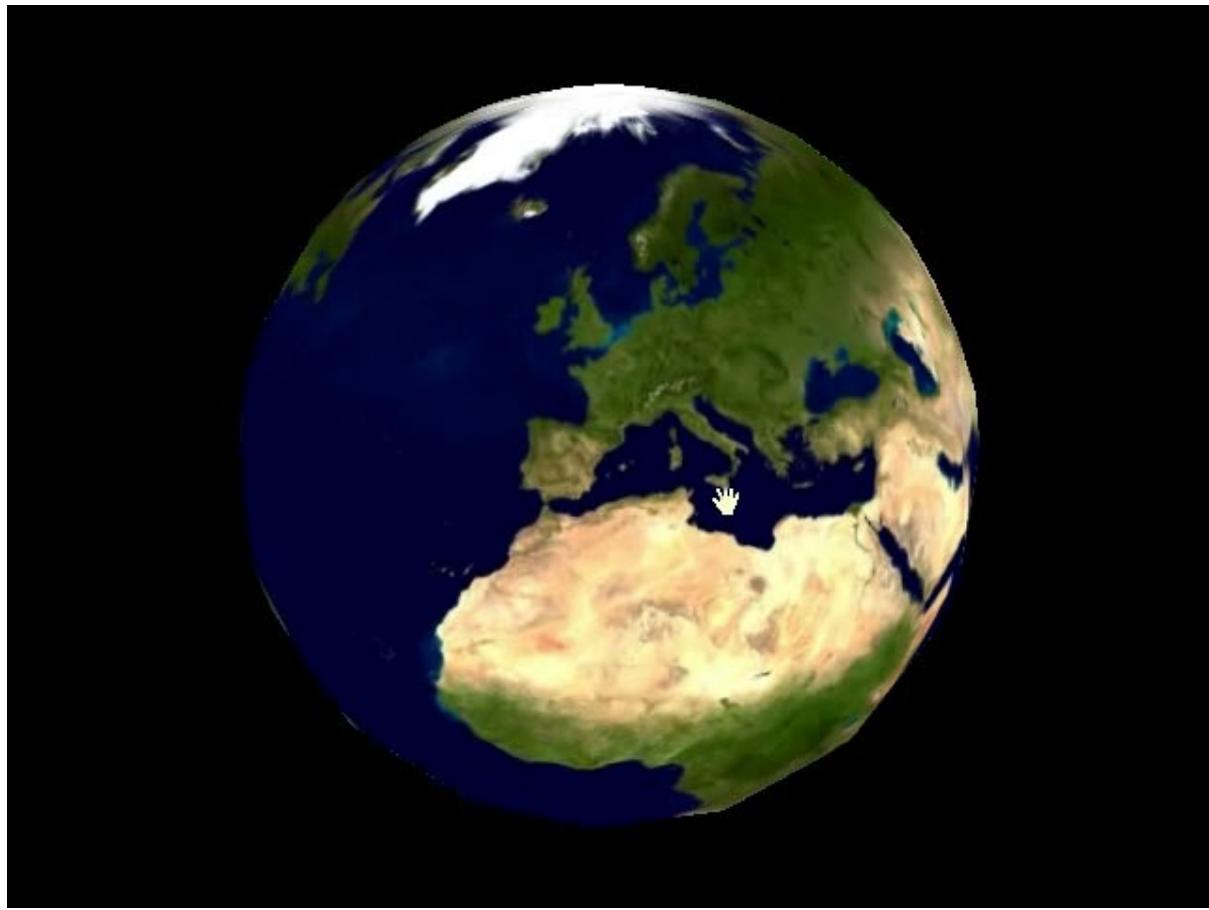
```
void DrawGL()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    camera->look();

    GLUquadric* params = gluNewQuadric();
    gluQuadricTexture(params,GL_TRUE);
    glBindTexture(GL_TEXTURE_2D,earth);
    gluSphere(params,1,20,20);
    gluDeleteQuadric(params);

    glFlush();
    SDL_GL_SwapBuffers();
}
```



Téléchargez la vidéo au format avi/Xvid (1.17 Mo) (<http://62.4.17.167/uploads/fr/ftp/kayl/trackball.avi>)

Téléchargez le projet Code::Blocks, l'exécutable Windows et le Makefile Unix (1.30 Mo)
(http://sdz.tdct.org/sdz/medias/www.siteduzero.com_uploads_fr_ftp_kayl_sdlgl_08_trackball.zip)

Et voilà finie la prise de tête de prévoir à l'avance comment placer la caméra pour bien voir votre scène !

Dans la chapitre suivant nous verrons une caméra encore plus intéressante mais un peu plus dure à implémenter : la caméra **FreeFly** qui nous permettra de voler librement dans notre scène.

Contrôle avancé de la caméra (Partie 2/2)

Dans cette seconde partie sur le contrôle avancé de la caméra, nous allons voir un autre type de caméra : la caméra FreeFly. Nous ne verrons pas de nouveau concept C++ mais cette caméra requiert un peu plus de maths que la précédente. Heureusement pour vous j'ai créé une annexe mathématique rien que pour ce chapitre mais qui peut vous être utile tout le temps : la trigonométrie (<http://www.siteduzero.com/tuto-3-8528-1-la-trigonometrie.html>).

Encore une fois deux lectures sont possibles pour ce chapitre donc ne paniquez pas si les mathématiques utilisées vous dépassent. Il est tout à fait possible de voir la caméra uniquement comme une nouvelle fonctionnalité. Comprendre son principe et son utilisation dans une scène de test est alors amplement suffisant.

Principe d'une caméra FreeFly

Une caméra FreeFly, comme son nom l'indique, permet de se déplacer/voler librement dans une scène. On retrouve parfois ce type de caméra sous des appellations incomplètes comme FreeLook, voire erronée comme FPS (First Person Shooter). FreeLook ne donne pas la même idée de déplacement, caractéristique importante de la caméra FreeFly. Une caméra FPS quant à elle est plus complexe car assujettie à la gravité, à des déplacements contraints (celui du personnage que l'on incarne), etc.

Quoiqu'il en soit, le principe est exactement le même que dans **Counter Strike**, quand on est mort (certains plus souvent que d'autres ;)) et qu'on se balade librement dans la map pour regarder la fin de la partie.



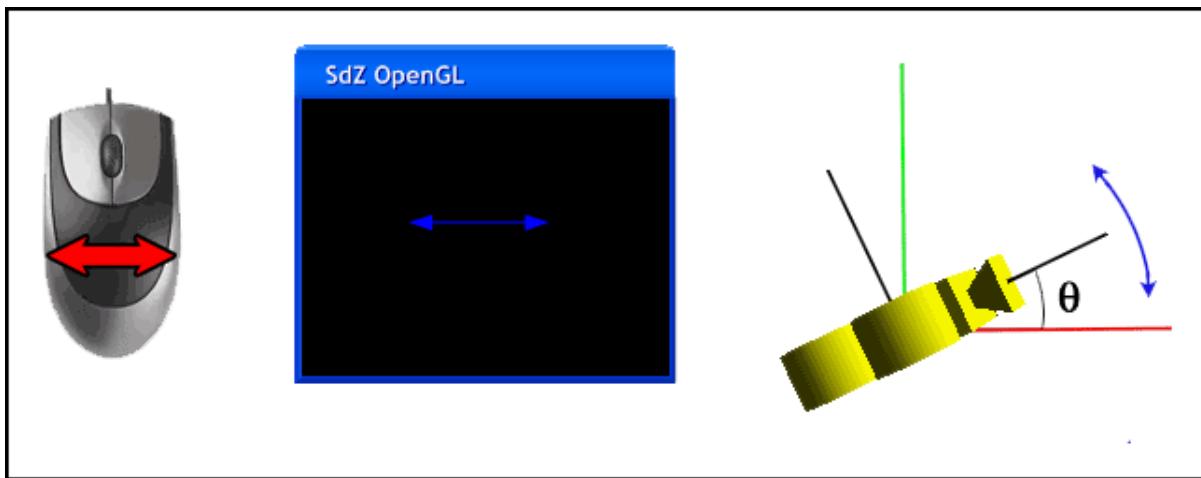
Caméra FreeFly (FreeLook) dans Counter Strike

Gestion du regard à la souris

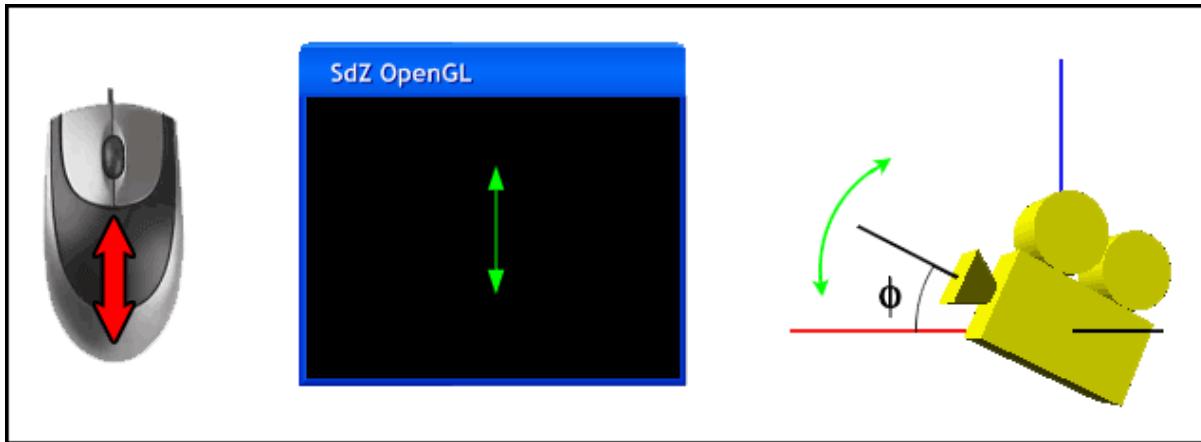
Pour pouvoir regarder tout autour de nous, nous allons utiliser les mouvements de la souris pour orienter la caméra :

- un mouvement horizontal de la souris fait tourner la caméra horizontalement autour de la verticale du monde (regard à gauche et à droite) ;
- un mouvement vertical de la souris fait tourner la caméra verticalement (le regard se lève ou se baisse).

Ces mouvements sont illustrés par les schémas ci-dessous (réutilisation des noms des angles propres aux coordonnées sphériques (http://www.siteduzero.com/tuto-3-8528-1-la-trigonometrie.html#ss_part_3) :



Mouvement horizontal de la souris (schéma vu du dessus)



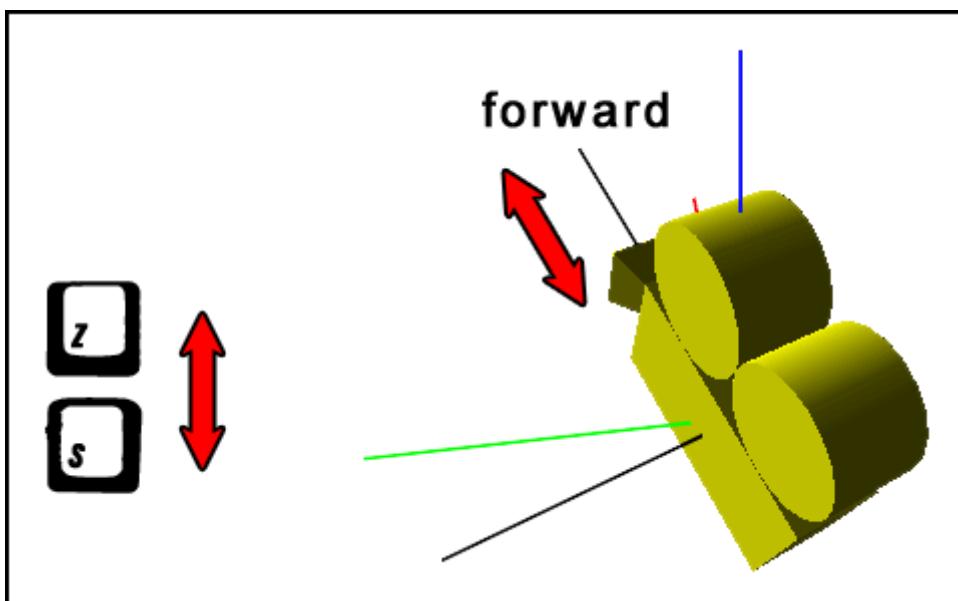
Mouvement vertical de la souris (schéma vu de côté)

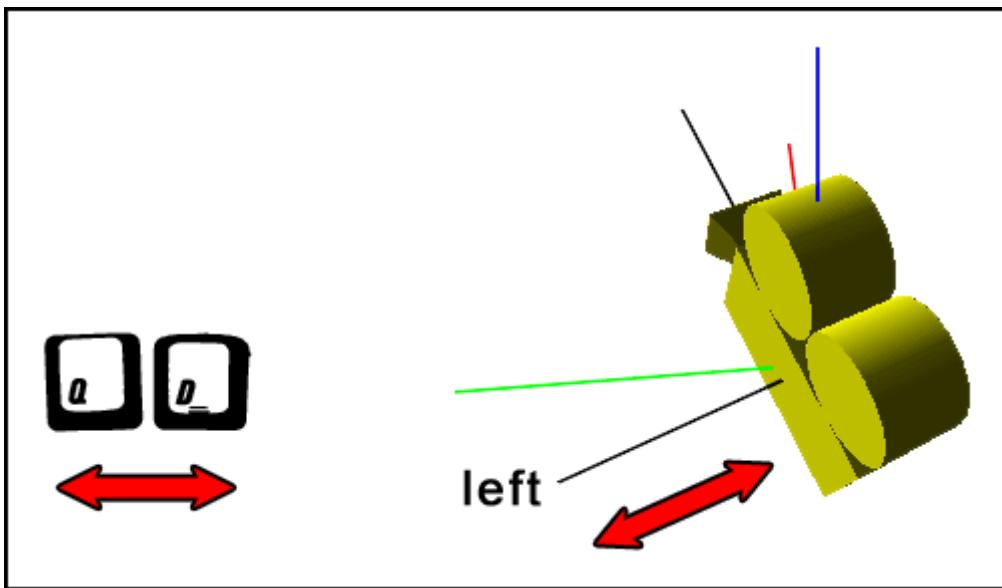
Gestion du déplacement au clavier

La souris nous permet d'orienter la caméra par rapport aux axes X, Y, Z locaux, le clavier quant à lui va nous permettre de déplacer la caméra selon l'orientation actuelle :

- 2 touches nous permettent de faire **avancer/reculer** la caméra ;
- 2 touches nous permettent de faire **strafe** (déplacement latéral) la caméra.

J'ai choisi d'utiliser les touches ZQSD, utilisées de façon presque standard pour ce genre de mouvement sur un clavier AZERTY :





Gestion fluide du mouvement

Pour l'instant en suivant mon tutoriel, nous avons utilisé les événements pour :

- faire bouger notre grue dans le chapitre sur les transformations (http://www.siteduzero.com/tuto-3-6304-1-les-transformations.html#ss_part_4) ;
- faire tourner la caméra dans le chapitre précédent (<http://www.siteduzero.com/tuto-3-9174-1-controle-avance-de-la-camera-partie-1-2.html>) sur la caméra TrackBall.

Nous utiliserons le même système de mouvement à la souris, donc l'utilisation des `SDL_MouseMotionEvent` pour orienter la caméra.

Pour notre grue, la ruse de l'animation était basée sur l'activation de la répétition des touches avec `SDL_EnableKeyRepeat`. La vitesse du mouvement était donc dépendante de la vitesse de répétition des touches, ce qui pose trois problèmes :

- si le taux de répétition des touches est trop faible le mouvement est saccadé ;
- utiliser l'événement d'appui sur une touche à son apparition empêche de pouvoir utiliser plusieurs touches en même temps ;
- le mouvement n'est pas réellement lié au temps écoulé.

Dans le cadre de notre petite grue sans prétention tout cela était tout à fait tolérable, ici nous rentrons dans un domaine différent : le contrôle de la vue 3D. Les personnes habituées aux FPS ne se rendent peut-être pas compte que la manière avec laquelle ils se déplacent peut vite donner la nausée à un spectateur non habitué. La **fluidité du mouvement** de la caméra est donc primordiale ici.

Les KeyStates

Je vous l'ai dit juste avant, utiliser l'événement clavier juste à son apparition n'est pas la bonne solution pour gérer un déplacement fluide.

Nous allons donc nous contenter de mettre à jour un **tableau** interne de l'**état des touches** qui nous intéressent.

Par exemple si la touche `SDLK_z` nous intéresse et que l'on reçoit un événement `SDL_KEYDOWN`, nous mettrons la case `SDLK_z` de notre tableau de booléens à `true`, pour indiquer qu'elle est actuellement enfoncee.

De même sur la réception d'un événement `SDL_KEYUP` avec la touche `SDLK_z`, nous mettrons la case correspondante du tableau à `false` pour indiquer qu'elle n'est pas/plus enfoncee.

Comment faire correspondre un nom de touche avec un indice dans un tableau ?

Nous allons pour cela utiliser ce que l'on appelle, comme en PHP, un tableau associatif. Au lieu de faire `mon_tableau[indice_numérique]` nous ferons `mon_tableau_associatif[clé]`. L'outil en C++ qui permet de faire cela est appelé une **map**. On la définit par les deux types qu'elle utilise, le type pour la clé (la clé est ce qui remplace l'indice numérique, bien qu'elle puisse être numérique aussi :)), et le type de la valeur à stocker.

Dans notre cas, nous souhaitons stocker des booléens (bool) indiquant si une touche (SDLKey) est actuellement enfoncee ou non.

Cela correspond donc à la map suivante :

```
typedef std::map<SDLKey,bool> KeyStates;
```

On peut l'utiliser ainsi :

```
Keystates keystates; //déclaration d'une variable de type KeyStates  
keystates[SDLK_Z] = true; //association bidon, eh oui je n'ai pas recu d'événement comment je peux s  
avoir ? :p )
```

Configuration du clavier

Dans l'absolu on se fiche pas mal que la touche pour faire avancer soit la touche SDLK_z. Ça pourrait être SDLK_i qu'on s'en ficherait tout autant. :-°

Ce que l'on veut savoir à n'importe quel moment c'est si la touche qui sert à faire avancer en avant (forward) est enfoncee ou non, quel que soit le choix du joueur/concepteur en ce qui concerne la configuration des touches.

Nous allons donc utiliser un deuxième tableau associatif qui associera une **action** et la **touche** utilisée pour effectuer cette action. L'action sera nommée textuellement, comme « forward » pour aller en avant, et la touche sera le code de la SDLKey comme SDLK_z (qui est en fait une valeur numérique mais on n'a pas besoin de le savoir).

Cela correspond donc à la map suivante :

```
typedef std::map<std::string,SDLKey> KeyConf;
```

On peut alors l'utiliser ainsi :

```
KeyConf keyconf; //déclaration d'une variable de type KeyConf.  
keyconf["forward"] = SDLK_z; //ici je choisis d'utiliser la touche z pour l'action forward.
```

On peut alors tout à fait imaginer charger la configuration des touches via un fichier de configuration ou même demander à l'utilisateur quelles touches utiliser. Ces deux techniques ne seront pas enseignées ici car dépassent le cadre du tuto OpenGL. Nous utiliserons juste une affectation action/touche comme dans l'exemple du dessus.

Grâce à la combinaison de KeyStates et de notre configuration des touches, il est alors facile de définir si la touche pour réaliser une action donnée est enfoncee :

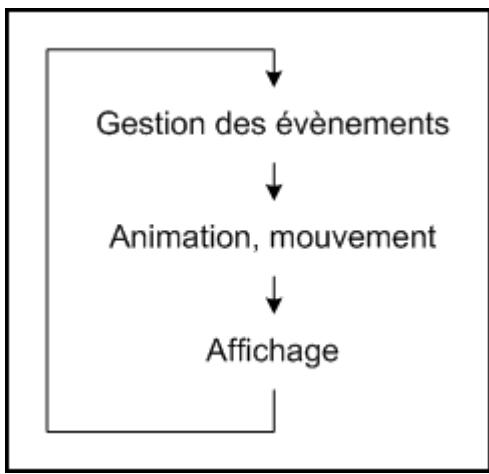
```
Keystates keystates;  
KeyConf keyconf;  
keyconf["forward"] = SDLK_z;  
keystates[keyconf["forward"]] = true;
```

La partie `keyconf["forward"]` vient récupérer la touche utilisée pour l'action « forward », cette touche sert de clé pour la map touche/booléen `keystates`.

Mouvement animé

Nous avons déjà réalisé un mouvement animé dès notre première scène 3D avec un cube.

La boucle principale de notre programme était la suivante :



Structure de la boucle principale

Ici nous ferons la même chose : un mouvement de la caméra qui dépend du **temps écoulé** mais aussi de l'**état des touches**.

Par exemple nous ne ferons avancer la caméra que si la touche correspondante est enfoncée, ce qui donne en code :

```

if (keystates[keyconf["forward"]])
    position += forward * (speed * timestep);

```

où timestep est le temps écoulé depuis la dernière image, speed est la vitesse de déplacement, forward est le vecteur d'orientation de la caméra (qui pointe là où elle regarde), et position est la position de la caméra.

Vector3D

Mathématiquement je sais ce qu'est un vecteur mais ça existe en C++ ?

À la base non. Mais rien ne nous empêche de le créer. D'ailleurs si vous avez déjà fait un peu de SDL, j'imagine que vous avez pu être amenés à créer des struct ayant comme attributs x et y les positions de vos sprites à blitter. Quoiqu'il en soit ici des attributs (X,Y et Z) ne nous suffisent pas. Comme on l'a vu dans le code juste au-dessus, il nous faut pouvoir multiplier un vecteur par un nombre, additionner deux vecteurs, et un peu plus encore. Pour cela on utilise en C++ **la surcharge d'opérateurs** qui permet de redéfinir les opérations élémentaires comme l'addition, la multiplication pour un type (ici notre classe vecteur) non élémentaire.

En définissant ainsi l'opérateur + comme méthode de notre classe, il est alors tout à fait possible dans le code, comme on le fait pour des nombres, de faire vecteur1 + vecteur2. Et mine de rien ça simplifiera beaucoup le code de la caméra.

Voici la déclaration de la classe Vector3D que nous allons utiliser. Je vous la donne pour vous montrer la déclaration des opérateurs. L'implémentation en elle-même est triviale, car purement mathématique et je laisse les curieux aller la lire dans l'archive finale. Pour l'utiliser nous n'avons pas besoin de connaître les détails de l'implémentation.

```

class Vector3D
{
public:
    double X;
    double Y;
    double Z;

    Vector3D();
    Vector3D(double x,double y,double z);
    Vector3D(const Vector3D & v);
    Vector3D(const Vector3D & from,const Vector3D & to);

    Vector3D & operator= (const Vector3D & v);

    Vector3D & operator+= (const Vector3D & v);
    Vector3D operator+ (const Vector3D & v) const;

    Vector3D & operator-= (const Vector3D & v);
    Vector3D operator- (const Vector3D & v) const;

    Vector3D & operator*= (const double a);
    Vector3D operator* (const double a) const;
    friend Vector3D operator* (const double a,const Vector3D & v);

    Vector3D & operator/= (const double a);
    Vector3D operator/ (const double a) const;

    Vector3D crossProduct(const Vector3D & v) const;
    double length() const;
    Vector3D & normalize();
};


```

Fluidité VS Molette

Il me reste une dernière chose à couvrir avant d'entrer dans l'implémentation de la caméra et cela concerne la molette. Vous l'avez vu dans la chapitre précédent, la molette se gère comme un bouton et on détecte que la molette a été montée si on reçoit un événement de type `SDL_MOUSEBUTTONDOWN` sur le bouton `SDL_BUTTON_WHEELUP`. Par conséquent si on bougeait sur cet événement on viendrait remettre en cause toute ma super théorie de fluidité quantique à convergence rétroactive ! Image utilisateur

Nous allons donc faire comme avec le clavier :

- sur détection d'un mouvement de la molette on met un booléen à true ;
- quand nous faisons bouger la caméra, si ce booléen vaut true nous déplaçons verticalement la caméra en fonction du temps écoulé.

Oui mais quand remet-on ce booléen à false ?

On ne peut pas le remettre à false quand on détecte que la molette ne bouge plus car les événements `SDL_MOUSEBUTTONDOWN` et `SDL_MOUSEBUTTONUP` pour la molette sont consécutifs, on n'aurait donc aucun mouvement. On utilise donc le **temps** !

- Sur détection d'un mouvement de la molette on met un booléen à true et une variable temps restant à 250 (ms, c'est un exemple).

- Quand nous faisons bouger la caméra, si ce booléen vaut true :
 - on décrémente le temps restant du temps écoulé ;
 - si le temps restant est < 0 alors on passe le booléen à false.

Nous allons bien sûr voir toutes ces techniques en pratique, appliquées à notre caméra FreeFly.

Implémentation de la caméra

Tout comme notre caméra TrackBall du chapitre précédent, nous allons implémenter la caméra FreeFly en C++ à l'aide d'une classe.

Nous devons ici aussi gérer trois types d'événements :

- le mouvement de la souris : pour changer l'orientation de la caméra ;
- le clavier : pour déplacer la caméra ;
- la molette (qui se gère comme un clic de bouton) : pour faire monter/descendre la caméra.

Nous aurons une méthode look dont l'appel viendra remplacer dans nos codes l'utilisation du gluLookAt (appelé en interne par la méthode look de toute façon).

Nous allons aussi créer trois méthodes pour paramétrier la caméra :

- setSpeed : pour définir la vitesse de déplacement de la caméra ;
- setSensitivity : pour déterminer la vitesse de rotation de la caméra en fonction du mouvement en pixels du curseur de la souris ;
- setPosition : pour placer précisément la caméra quand on le souhaite.

Nous introduirons aussi une dernière méthode, animate, qui viendra gérer le mouvement fluide de la caméra, comme expliqué dans la partie précédente.

Voici une traduction formelle de ce que je viens de dire rapidement :

UML simplifié

FreeFlyCamera	
#_speed : double	
#_sensitivity : double	
#_timeBeforeStoppingVerticalMotion : Uint32	
#_verticalMotionActive : bool	
#_verticalMotionDirection : int	
#_keystates : KeyStates	
#_keyconf : Keyconf	
#_position : Vector3D	
#_target : Vector3D	
#_forward : Vector3D	
#_left : Vector3D	
#_theta : double	
#_phi : double	
+OnMouseMove(SDL_MouseMotionEvent)	
+OnMouseButtonDown(SDL_MouseButtonEvent)	
+OnKeyboard(SDL_KeyboardEvent)	
+animate(Uint32)	
+setSpeed(double)	
+setSensitivity(double)	
+setPosition(Vector3D)	
+look()	
#VectorsFromAngles()	

FreeFlyCamera	
#_speed : double	
#_sensitivity : double	
#_timeBeforeStoppingVerticalMotion : Uint32	
#_verticalMotionActive : bool	
#_verticalMotionDirection : int	
#_keystates : KeyStates	
#_keyconf : Keyconf	
#_position : Vector3D	
#_target : Vector3D	
#_forward : Vector3D	
#_left : Vector3D	
#_theta : double	
#_phi : double	
+OnMouseMove(SDL_MouseMotionEvent)	
+OnMouseButtonDown(SDL_MouseButtonEvent)	
+OnKeyboard(SDL_KeyboardEvent)	
+animate(Uint32)	
+setSpeed(double)	
+setSensitivity(double)	
+setPosition(Vector3D)	
+look()	
#VectorsFromAngles()	

Déclaration C++

```
class FreeFlyCamera
{
public:
    FreeFlyCamera(const Vector3D & position = Vector3D(0,0,0));

    virtual void OnMouseMove(const SDL_MouseMotionEvent & event);
    virtual void OnMouseButtonDown(const SDL_MouseButtonEvent & event);
    virtual void OnKeyboard(const SDL_KeyboardEvent & event);

    virtual void animate(Uint32 timestep);
    virtual void setSpeed(double speed);
    virtual void setSensitivity(double sensitivity);

    virtual void setPosition(const Vector3D & position);

    virtual void look();

    virtual ~FreeFlyCamera();

protected:
    double _speed;
    double _sensitivity;

    Uint32 _timeBeforeStoppingVerticalMotion;
    bool _verticalMotionActive;
    int _verticalMotionDirection;

    typedef std::map<SDLKey, bool> KeyStates;
    KeyStates _keystates;
    typedef std::map<std::string, SDLK> Keyconf;
    Keyconf _keyconf;

    Vector3D _position;
    Vector3D _target;
    Vector3D _forward;
    Vector3D _left;
    double _theta;
    double _phi
```

UML simplifié

Déclaration C++

```
void VectorsFromAngles();  
};
```

Décryptons ensemble tous les attributs dont nous allons avoir besoin :

- **double _speed** : vitesse de déplacement de la caméra ;
- **double _sensitivity** : sensibilité de la caméra aux mouvements de la souris ;
- **Uint32 _timeBeforeStoppingVerticalMotion** : si un mouvement vertical (à la molette) est en cours, combien de temps reste-t-il avant de l'arrêter ?
- **bool _verticalMotionActive** : y a-t-il un mouvement vertical en cours ?
- **int _verticalMotionDirection** : sens du mouvement vertical quand il a été déclenché à la molette (+1 vers la haut, -1 vers le bas) ;
- **KeyStates _keystates** : tableau donnant l'**état** des touches **utilisées** ;
- **Keyconf _keyconf** : tableau donnant les **touches** à utiliser pour chaque **action** ;
- **Vector3D _position** : position de la caméra dans l'absolu ;
- **Vector3D _target** : point regardé par la caméra dans l'absolu ;
- **Vector3D _forward** : vecteur donnant la direction du regard (et donc du déplacement vers l'avant) ;
- **Vector3D _left** : vecteur perpendiculaire au regard pour le déplacement latéral ;
- **double _theta** : angle de rotation horizontale de la caméra (autour de la verticale) ;
- **double _phi** : angle de rotation verticale de la caméra.

On remarque aussi, dans le schéma UML comme dans la déclaration de la classe, une méthode protégée (non publique) : **VectorsFromAngles**. Cette méthode viendra calculer les vecteurs **_forward** et **_left** en fonction des nouvelles valeurs de **_theta** et **_phi**.

Constructeur

Vous l'avez vu dans l'implémentation, le constructeur possède un paramètre facultatif : la position initiale de la caméra. Si on ne la spécifie pas, elle sera en (0,0,0) au départ.

Le code n'est pas passionnant, juste de l'initialisation des attributs :

```

FreeFlyCamera::FreeFlyCamera(const Vector3D & position)
{
    _position = position; //si aucune position n'est définie on reçoit quand même (0,0,0) en paramètre
    _phi = 0;
    _theta = 0;
    VectorsFromAngles(); //décrit un peu plus loin

    _speed = 0.01;
    _sensitivity = 0.2;
    _verticalMotionActive = false;
    //Initialisation de la configuration des touches
    _keyconf["forward"] = SDLK_z;
    _keyconf["backward"] = SDLK_s;
    _keyconf["strafe_left"] = SDLK_q;
    _keyconf["strafe_right"] = SDLK_d;
    _keyconf["boost"] = SDLK_LSHIFT;
    //Initialisation des KeyStates
    _keystates[_keyconf["forward"]] = false;
    _keystates[_keyconf["backward"]] = false;
    _keystates[_keyconf["strafe_left"]] = false;
    _keystates[_keyconf["strafe_right"]] = false;
    _keystates[_keyconf["boost"]] = false;

    SDL_WM_GrabInput(SDL_GRAB_ON);
    SDL_ShowCursor(SDL_DISABLE);
}

```

Pourquoi ces deux dernières lignes ?

Le but de notre caméra est de bouger le regard avec la souris en permanence. Nous utilisons donc l'information de **déplacement relatif** de la souris pour faire tourner d'autant la caméra. Que se passe-t-il si la souris quitte la fenêtre et re-rentre par un autre endroit ?

1. Tout le temps où la souris sera en dehors de la fenêtre la caméra sera indirigeable, ça peut être pénible quand on est pas loin du bord.
2. Si la caméra revient dans la fenêtre par un autre côté, le déplacement relatif peut être énorme d'un coup et faire trop tourner la caméra, on ne comprendra plus ce que l'on regarde (changement trop brusque).

Par la combinaison des deux lignes :

```

SDL_WM_GrabInput(SDL_GRAB_ON);
SDL_ShowCursor(SDL_DISABLE);

```

on vient demander à SDL d'interdire à la souris de quitter la fenêtre, mais en plus, quand le curseur est juste au bord de la fenêtre, de générer quand même des informations de déplacement relatif de la souris.

Ces deux commandes appelées par le constructeur de notre caméra nécessitent que la SDL soit initialisée. Il ne faut donc pas construire un objet FreeFlyCamera avant. Dans notre scène de test comme nous utiliserons la caméra en variable globale pour l'instant, nous utiliserons l'allocation dynamique (avec un new) pour pallier ce problème.

On Mouse Motion

Comme notre caméra est assurée de recevoir des événements de mouvement de la souris corrects, la méthode OnMouseMotion est très simple :

```
void FreeFlyCamera::OnMouseMove(const SDL_MouseMotionEvent & event)
{
    _theta -= event.xrel*_sensitivity;
    _phi -= event.yrel*_sensitivity;
    VectorsFromAngles();
}
```

Pourquoi des signes « - » ?

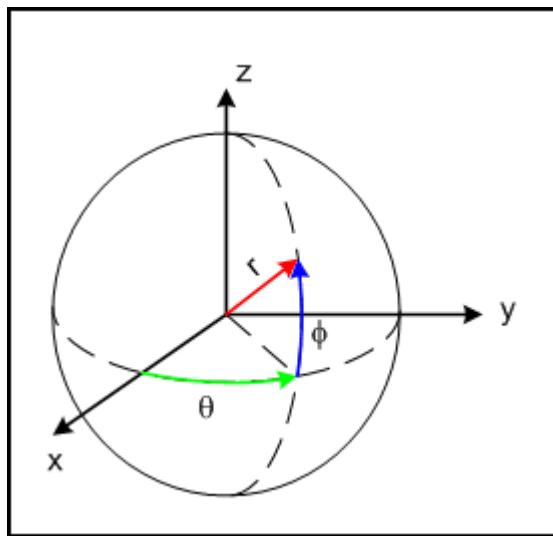
Les angles s'expriment dans le sens trigonométrique (sens inverse des aiguilles d'une montre). Un changement positif de `_theta` correspond donc à un mouvement « vers la gauche » de la souris, soit un déplacement négatif sur l'axe des X. Soustraire un nombre négatif revient à additionner son opposé donc nous retombons sur nos pattes. De même pour `_phi`, un changement positif correspond à un mouvement « vers le haut » de la souris, soit un déplacement négatif sur l'axe des Y (pour SDL).

Vectors From Angles

À chaque fois que les angles sont changés, il faut recalculer le vecteur d'orientation de la caméra, `_forward`, ainsi que le vecteur latéral `_left`.

Le vecteur `_forward` permet de définir à la fois **où regarder**, `_target`, et dans quelle direction **avancer**. Le vecteur `_left` sert pour le mouvement latéral.

Comme ce sont des vecteurs, ils donnent des directions et ne sont donc pas localisés à un endroit précis de l'espace. On peut donc réfléchir comme si nous étions en (0,0,0). De deux angles on veut passer aux coordonnées 3D d'un vecteur, hum... :-° mais comme c'est bizarre... ça me fait penser étrangement aux coordonnées sphériques de l'annexe de trigonométrie (http://www.siteduzero.com/tuto-3-8528-1-la-trigonometrie.html#ss_part_3). Comme c'est bizarre et pas du tout intentionnel... :soleil:



Rappel de l'utilisation des angles en coordonnées sphériques

Nous appliquerons donc le calcul bête et méchant expliqué en annexe pour calculer le vecteur `_forward` (équivalent du vecteur rouge sur le dessin du dessus). Ici nous réfléchissons sur des vecteurs unitaires (de longueur 1) donc le rayon de la « sphère » vaut 1.

```

void FreeFlyCamera::VectorsFromAngles()
{
    static const Vector3D up(0,0,1); //une constante, le vecteur vertical du monde, utilisé dans les calculs

    //On limite les valeurs de _phi, on vole certes, mais on en fait pas de loopings :
    if (_phi > 89)
        _phi = 89;
    else if (_phi < -89)
        _phi = -89;

    //passage des coordonnées sphériques aux coordonnées cartésiennes
    double r_temp = cos(_phi*M_PI/180);
    _forward.Z = sin(_phi*M_PI/180);
    _forward.X = r_temp*cos(_theta*M_PI/180);
    _forward.Y = r_temp*sin(_theta*M_PI/180);

    //diantre mais que fait ce passage ?
    _left = up.crossProduct(_forward);
    _left.normalize();

    //avec la position de la caméra et la direction du regard, on calcule facilement ce que regarde la caméra (la cible)
    _target = _position + _forward;
}

```

Le seul point flou qui peut subsister ici est le calcul du vecteur `_left` pour le mouvement latéral.

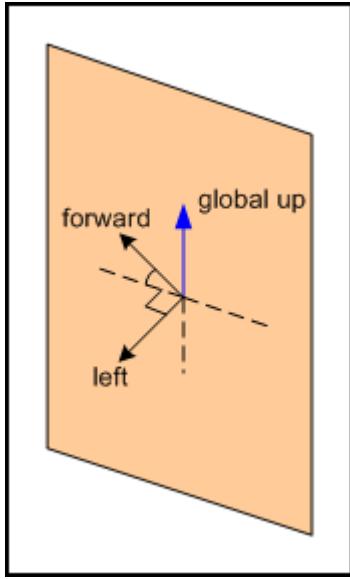
On utilise ici le produit vectoriel (méthode `crossProduct` de la classe `Vector3D`) pour calculer le vecteur `_left`, **perpendiculaire** au plan formé par le vecteur `_forward` avec la verticale. Ce vecteur doit ensuite être normalisé (on lui donne une longueur de 1) pour pouvoir être utilisé dans nos mouvements.

Tout cela justifie donc les deux lignes :

```

_left = up.crossProduct(_forward);
_left.normalize();

```



`_left` est perpendiculaire au plan formé par `_forward` avec la verticale

On Mouse Button

La gestion de la molette est assez simple. Comme il a été vu dans la partie « Gestion fluide du mouvement », nous ne bougerons pas la caméra directement mais signalerons juste qu'il faut bouger.

J'ai choisi de faire monter/descendre la caméra pendant 250 ms. Ce temps n'est **pas** à changer en fonction de la vitesse de la caméra, c'est le mouvement lui-même qui utilisera l'information de vitesse.

```

void FreeFlyCamera::OnMouseButton(const SDL_MouseButtonEvent & event)
{
    if ((event.button == SDL_BUTTON_WHEELUP)&&(event.type == SDL_MOUSEBUTTONDOWN)) //coup de molette
vers le haut
    {
        _verticalMotionActive = true; //on demande à activer le mouvement vertical
        _timeBeforeStoppingVerticalMotion = 250; //pendant 250 ms
        _verticalMotionDirection = 1; //et vers le haut

    }
    else if ((event.button == SDL_BUTTON_WHEELDOWN)&&(event.type == SDL_MOUSEBUTTONDOWN)) //coup de
molette vers le bas
    {
        _verticalMotionActive = true; //on demande à activer le mouvement vertical
        _timeBeforeStoppingVerticalMotion = 250; //pendant 250 ms
        _verticalMotionDirection = -1; //et vers le bas
    }
}

```

On Keyboard

Dans la méthode OnKeyboard, on va utiliser les KeyStates découverts plus haut et initialisés dans le constructeur. On pourrait utiliser des if, ou un switch...case, mais il y a plus simple :

```

void FreeFlyCamera::OnKeyboard(const SDL_KeyboardEvent & event)
{
//on parcourt tous les keystates actuels
    for (KeyStates::iterator it = _keystates.begin();it != _keystates.end();
         it++)
    {
        if (event.keysym.sym == it->first) //est-ce que la touche responsable de l'événement est celle du keystate ?
        {
            it->second = (event.type == SDL_KEYDOWN); //true si enfoncé, false si relâché
            break; //la touche responsable de l'événement a été utilisée, on quitte le for
        }
    }
}

```

La seule difficulté ici est le parcours des KeyStates, le tableau associatif qui donne l'état des touches utilisées. L'avantage de cette technique : code très court, simple, et je n'ai pas codé en dur quelles touches tester (pas d'informations redondantes).

Animate

Jusqu'à présent nous avons vu des méthodes qui étaient appelées sur des événements souris/clavier. Ces méthodes (à part le MouseMotion) ne venaient que définir des booléens pour une gestion fluide du mouvement. Mouvement qui va réellement s'effectuer ici, dans la méthode Animate, appelée à chaque boucle du programme. Le code n'est qu'une simple application des principes vus dans la partie « Gestion fluide du mouvement » :

```

void FreeFlyCamera::animate(Uint32 timestep)
{
//la vitesse réelle du déplacement est soit la vitesse de croisière, soit 10*la vitesse, en fonction
//de l'état enfoncé ou non de la touche correspondant à l'action "boost"
    double realspeed = (_keystates[_keyconf["boost"]])?10*_speed:_speed;
    if (_keystates[_keyconf["forward"]])
        _position += _forward * (realspeed * timestep); //on avance
    if (_keystates[_keyconf["backward"]])
        _position -= _forward * (realspeed * timestep); //on recule
    if (_keystates[_keyconf["strafe_left"]])
        _position += _left * (realspeed * timestep); //on se déplace sur la gauche
    if (_keystates[_keyconf["strafe_right"]])
        _position -= _left * (realspeed * timestep); //on se déplace sur la droite
    if (_verticalMotionActive)
    {
        if (timestep > _timeBeforeStoppingVerticalMotion)
            _verticalMotionActive = false;
        else
            _timeBeforeStoppingVerticalMotion -= timestep;
        _position += Vector3D(0,0,_verticalMotionDirection*realspeed*timestep); //on monte ou on descend, en fonction de la valeur de _verticalMotionDirection
    }
    _target = _position + _forward; //comme on a bougé, on recalcule la cible fixée par la caméra
}

```

On remarque que dans certains cas, il est possible lors du même appel à Animate d'avancer et reculer en même temps (si les touches « forward » et « backward » sont **pressées simultanément**). De manière tout à fait logique les deux mouvements **s'annulent** et la caméra ne bougera pas (en avant ou en arrière).

Look

La dernière méthode intéressante est bien sûr la méthode Look, appelée à chaque image, avant de dessiner la scène. Rien de compliqué ici, les autres méthodes ont calculé pour nous la position de la caméra et la cible qu'elle regarde. Il suffit de faire un appel propre à gluLookAt et tout va pour le mieux dans le meilleur des mondes.

```

void FreeFlyCamera::look()
{
    gluLookAt(_position.X,_position.Y,_position.Z,
              _target.X,_target.Y,_target.Z,
              0,0,1);
}

```

Destructeur

On finit néanmoins avec le destructeur pour « remettre les choses comme on les a trouvées ». Si l'utilisateur pense qu'il n'a plus besoin de sa caméra, il serait de bon ton quand même de lui rendre son curseur que l'on a masqué pour nos besoins personnels :

```

FreeFlyCamera::~FreeFlyCamera()
{
    SDL_WM_GrabInput(SDL_GRAB_OFF);
    SDL_ShowCursor(SDL_ENABLE);
}

```

Vous trouverez bien sûr l'implémentation complète de la caméra dans l'archive en fin de chapitre.

Scène de test

Voilà, la caméra est prête mais une fois encore, comme pour la caméra TrackBall, il faut évidemment s'assurer de bien la créer et de lui envoyer tous les événements dont elle a besoin.

Un dilemme cependant se présentait à moi à la préparation de ce chapitre. Quelle scène minable allais-je encore créer pour montrer l'intérêt d'une caméra FreeFly ? :euh:

Pour vous épargner une réapparition de la scène du chapitre des textures (sol + cube + pyramide) j'ai lancé un appel sur les forums.

Deux zéros, qui ont appris l'OpenGL grâce à ce cours, ont répondu présents et nous ont donc préparé une petite scène sympathique qui nous servira même dans les chapitres à venir. Merci donc à TheDead Master (<http://www.siteduzero.com/membres-294-1659.html>) et à 42 (<http://www.siteduzero.com/membres-294-1498.html>) !

Leur scène est incluse dans un fichier à part, scene.cpp, dont les fonctions seront bien sûr appelées à partir de main.cpp.

Leurs fonctions que l'on doit appeler sont :

```
void chargerTextures(); //pour initialiser les textures que la scène utilise  
void dessinerScene(); //pour dessiner leur scène
```

Nous pouvons donc nous concentrer sur la création de la caméra.

En variable globale nous aurons donc :

```
FreeFlyCamera * camera;
```

Encore une fois, sous forme de pointeur pour ne pas la créer avant que la SDL ne soit initialisée. Nous allouons la caméra dynamiquement après la création de la fenêtre :

```
int main(int argc, char *argv[])
{
//...
    atexit(stop);
//...
    SDL_SetVideoMode(width, height, 32, SDL_OPENGL);
//...
    chargerTextures(); //nécessaire pour que les textures de leur scène soient chargées

    camera = new FreeFlyCamera(Vector3D(0,0,2)); //pour les besoins de la scène on surélève la caméra dès le départ
//...
}
```

La caméra sera automatiquement détruite à la fin du programme grâce à la fonction stop :

```
void stop()
{
    delete camera;
    SDL_Quit();
}
```

Une fois la caméra créée, dans notre boucle principale il ne faut pas oublier :

- d'envoyer les événements que la caméra attend ;

- d'appeler la méthode FreeFlyCamera::animate pour que la caméra bouge.

Cela se traduit donc par le code suivant :

```

while(SDL_PollEvent(&event))
{
    switch(event.type)
    {
        case SDL_QUIT:
            exit(0);
            break;
        case SDL_KEYDOWN:
            switch (event.key.keysym.sym)
            {
                case SDLK_p:
                    takeScreenshot("test.bmp");
                    break;
                case SDLK_ESCAPE:
                    exit(0);
                    break;
                default : //on a utilisé la touche P et la touche ECHAP, le reste (en keydown) est donné à la caméra
                    camera->OnKeyboard(event.key);
            }
            break;
        case SDL_KEYUP: //on n'utilise pas de keyup, on donne donc tout à la caméra
            camera->OnKeyboard(event.key);
            break;
        case SDL_MOUSEMOTION: //la souris est bougée, ça n'intéresse que la caméra
            camera->OnMouseMotion(event.motion);
            break;
        case SDL_MOUSEBUTTONUP:
        case SDL_MOUSEBUTTONDOWN: //tous les événements boutons (up ou down) sont donnés à la caméra
            camera->OnMouseButton(event.button);
            break;
    }
}

current_time = SDL_GetTicks();
elapsed_time = current_time - last_time; //on calcule le temps écoulé depuis la dernière image
last_time = current_time;

camera->animate(elapsed_time); //et on fait bouger la caméra

DrawGL();

```

Le détail de la fonction d'affichage ne nous intéresse pas vraiment ici. En effet la majeure partie se fait dans le scene.cpp pour utiliser la scène de The Dead Master et 42. Par contre, il ne faut pas oublier d'appeler la méthode look() de notre caméra :

```

void DrawGL()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

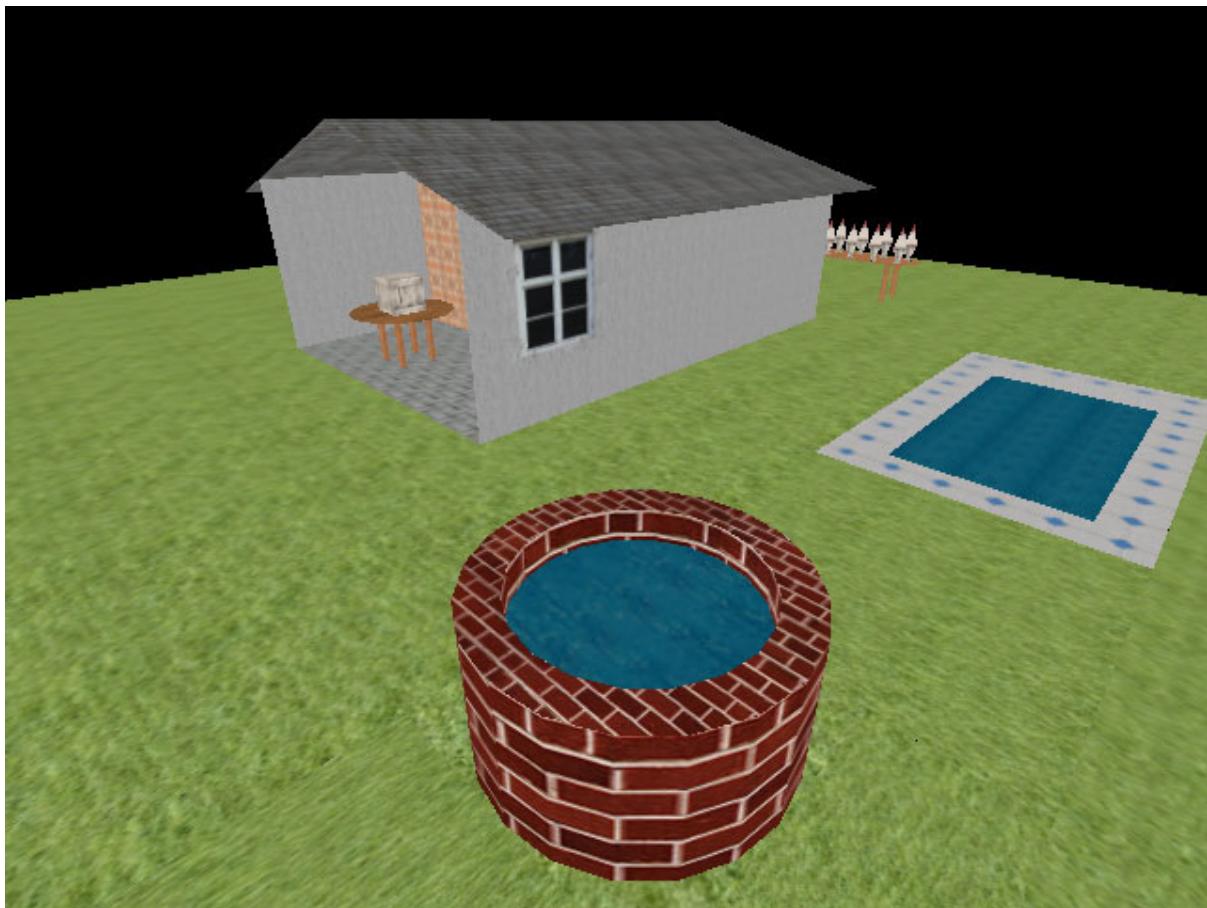
    camera->look(); //remplace un appel manuel à gluLookAt

    dessinerScene();

    glFlush();
    SDL_GL_SwapBuffers();
}

```

Et voilà le travail ! :soleil:



Téléchargez la vidéo au format avi/Xvid (2.11 Mo) (<http://62.4.17.167/uploads/fr/ftp/kayl/freefly.avi>)

Téléchargez le projet Code::Blocks, l'exécutable Windows et le Makefile Unix (1.23 Mo) (http://sdz.tdct.org/sdz/médias/www.siteduzero.com_uploads_fr_ftp_kayl_sdlgl_09_freefly.zip)

On arrive enfin au bout de ce chapitre ! :pirate:

Rappelez-vous que si vous n'avez pas tout compris des détails de l'implémentation de la caméra, comprendre son principe (mouvement) et comment l'utiliser peut suffire.

Maintenant grâce à notre caméra **FreeFly** nous pouvons enfin nous balader librement dans nos scènes et les contempler sous vraiment tous les angles.

Nous garderons cette scène pour le prochain chapitre, **la transparence**, dans lequel nous verrons comment rendre l'eau et les fenêtres transparentes. :magicien: Tout un programme !

La trigonométrie

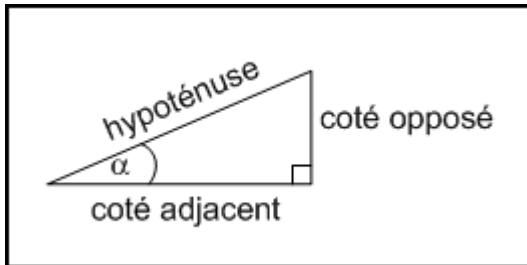
Dans cette annexe mathématique nous verrons le concept mathématique de trigonométrie, généralement rencontré dès la classe de 3e dans le cursus français.

Nous nous contenterons de parcourir le **minimum utile** pour faire de la **3D** et nous verrons que des outils mathématiques faciles d'utilisation peuvent nous être grandement utiles.

Trigo dans un triangle rectangle

La trigonométrie (littéralement « mesures » dans le « triangle »), est une branche des maths qui vient donner des relations entre les longueurs des côtés d'un triangle et ses angles.

La trigonométrie se rencontre généralement dans le cas d'un triangle rectangle :



En considérant l'angle alpha, on définit deux fonctions « cosinus » et « sinus » permettant de donner une relation entre la valeur de cet angle, et les longueurs des côtés du triangle :

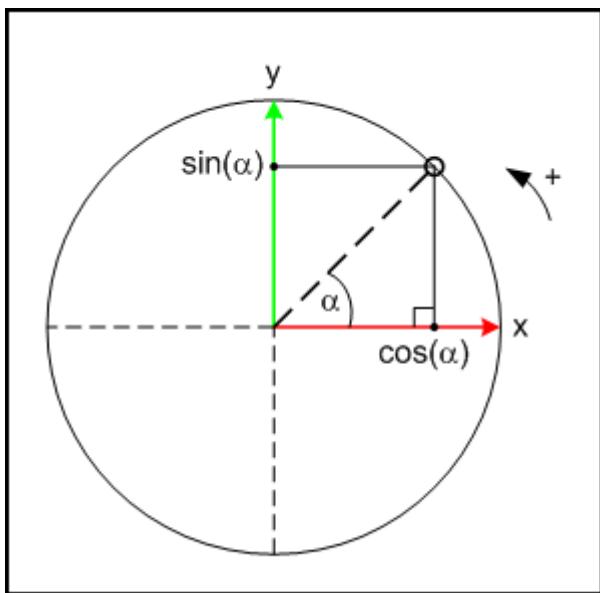
$$\cos(\alpha) = \frac{\text{coté adjacent}}{\text{hypoténuse}}$$

$$\sin(\alpha) = \frac{\text{coté opposé}}{\text{hypoténuse}}$$

En utilisant l'inverse de ces fonctions on peut alors trouver la valeur de l'angle en fonction des longueurs des côtés. On peut tout aussi bien, à partir de la valeur de l'angle et de la longueur d'un seul côté du triangle, trouver les longueurs des autres côtés.

Le cercle trigonométrique

Le triangle rectangle est la première approche que l'on a de la trigonométrie. On introduit généralement un autre objet mathématique : le cercle trigonométrique. Ce cercle mesure « 1 » (pas d'unité requise) de rayon et nous permet de donner tout leur sens au cosinus et sinus que nous venons de rencontrer :



Dans la figure ci-dessus on retrouve un triangle rectangle et donc nos formules vues plus haut sont toujours applicables :

$$\begin{aligned} \text{coté adjacent} &= \cos(\alpha) \times \text{hypoténuse} \\ x &= \cos(\alpha) \end{aligned}$$

$$\begin{aligned} \text{coté opposé} &= \sin(\alpha) \times \text{hypoténuse} \\ y &= \sin(\alpha) \end{aligned}$$

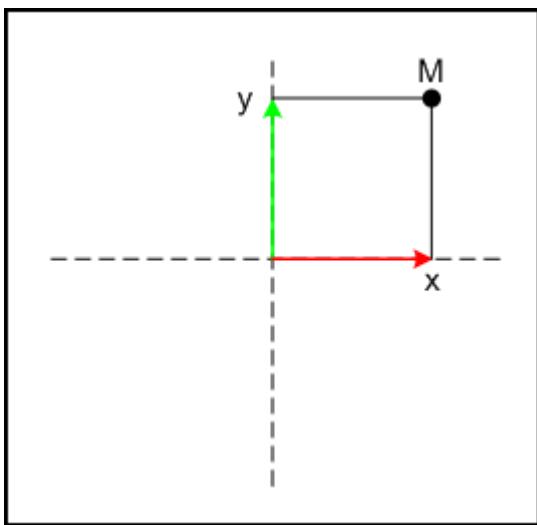
Et là tout s'éclaire ! Alors qu'auparavant nous pensions que cos et sin étaient de simples mesures abstraites qui ne servaient pas à grand chose, ici on se rend compte grâce au cercle trigonométrique qu'elles permettent de donner l'abscisse (sur X) et l'ordonnée (sur Y) d'un point quelconque du cercle trigonométrique se trouvant à un angle alpha donné par rapport à l'origine.

On note au passage que les angles se mesurent ici dans le sens trigonométrique (c'est-à-dire le sens inverse des aiguilles d'une montre).

Systèmes de coordonnées

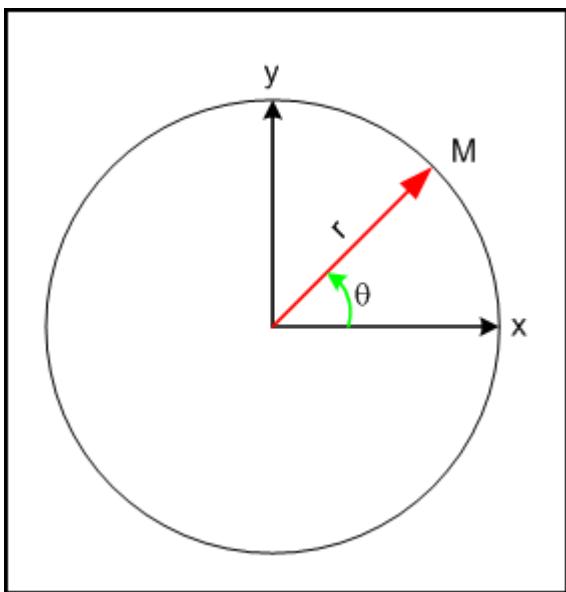
Nous avons vu rapidement dans le cas du cercle trigonométrique que nous pouvions passer d'un angle à une coordonnée (X,Y). Il est donc temps de faire un petit rappel non exhaustif sur les systèmes de coordonnées qui nous intéressent avec OpenGL.

Coordonnées cartésiennes



Les coordonnées cartésiennes sont les plus courantes. C'est d'ailleurs dans ce système que l'on exprime les positions de nos vertices avec OpenGL : `glVertex3d(X, Y, Z);`

Coordonnées polaires



Les coordonnées polaires sont un autre système de coordonnées que nous n'utiliserons pas trop mais qui nous sert d'intermédiaire pour bien comprendre la suite.

Un point est repéré par un **angle** (θ) et un **rayon** (r). On y voit une généralisation du cercle trigonométrique (où r valait 1) et donc nous pouvons très facilement en déduire comment passer des coordonnées polaires aux coordonnées cartésiennes :

$$x = r \times \cos(\theta)$$

$$y = r \times \sin(\theta)$$

Exemple en OpenGL

Nous venons de voir qu'il est possible d'exprimer un même point dans des systèmes de coordonnées différents. Nous pouvons donc envisager deux méthodes identiques en OpenGL pour faire tourner un point autour de l'origine :

Coordonnées polaires

```
glRotated(theta,0,0,1);
glBegin(GL_POINTS);
glVertex2i(r,0);
glEnd();
```

Coordonnées cartésiennes*

```
#include <math.h>

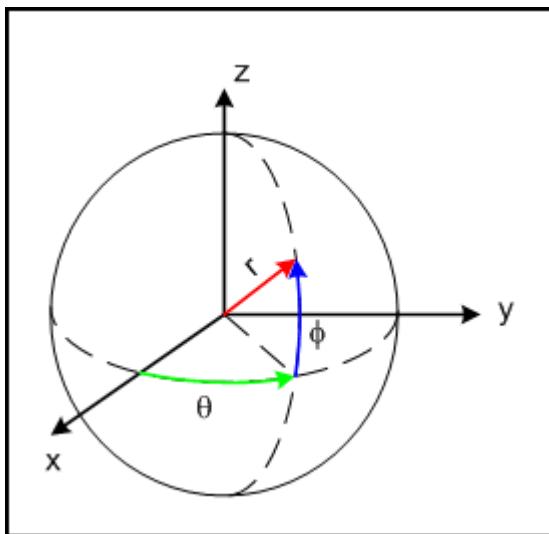
x=r*cos(theta*M_PI/180);
y=r*sin(theta*M_PI/180);
glBegin(GL_POINTS);
glVertex2i(x,y);
glEnd();
```

* les angles en OpenGL sont exprimés en degrés, cependant cos et sin, les fonctions définies dans `<math.h>`, attendent des radians. Il faut donc les convertir en multipliant par `M_PI/180`.

Coordonnées sphériques

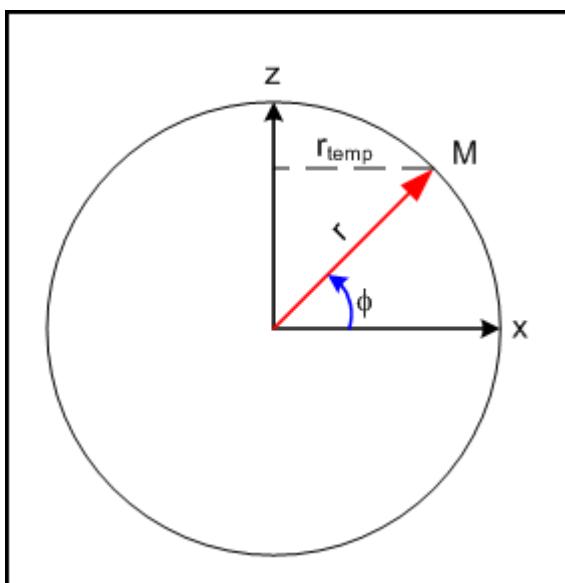
Ce système de coordonnées peut être vu comme une généralisation des coordonnées polaires en 3D. Il est très utile et nous servira notamment à contrôler l'orientation de notre caméra à l'aide de la souris.

En coordonnées sphériques, un point est représenté par **un rayon r**, et **deux angles** thêta et phi :



Pour passer des coordonnées sphériques aux coordonnées cartésiennes qui nous sont si chères, il faut y aller en deux étapes et se ramener à des cas 2D très simples.

Tout d'abord considérons seulement le rayon r et l'angle phi. Nous nous ramenons à un cas de coordonnées polaires sur le plan XZ :

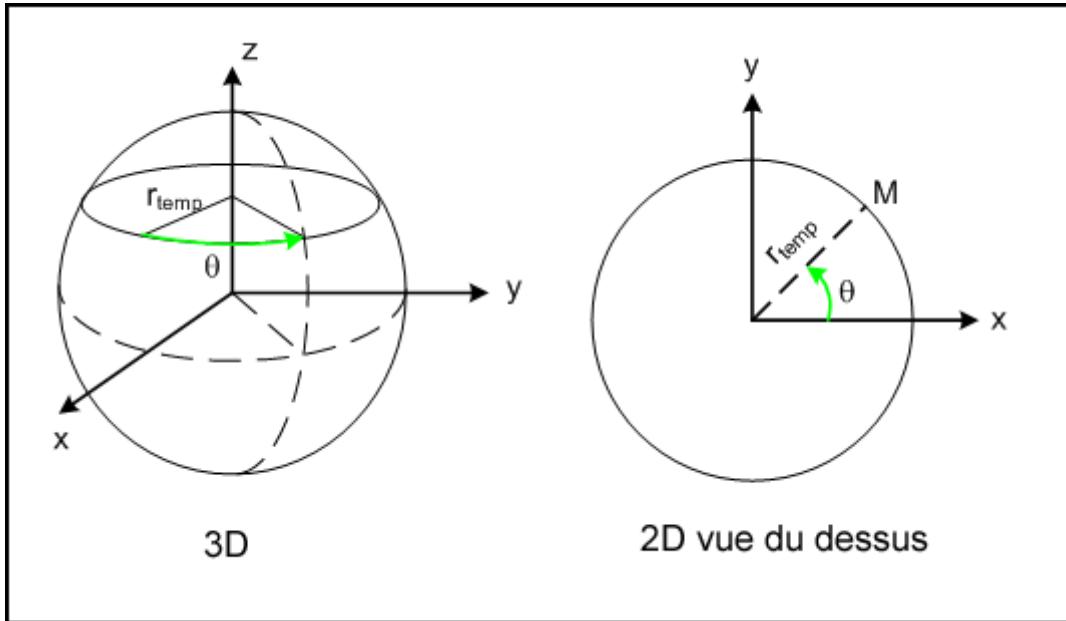


Nous pouvons en déduire directement la coordonnée Z de notre point avec le sinus, et le cosinus nous permet de définir une variable intermédiaire rtemp.

$$z = r \times \sin(\phi)$$

$$r_{temp} = r \times \cos(\phi)$$

rtemp est le rayon d'un cercle intermédiaire placé à l'altitude Z de notre point :



En regardant notre sphère du dessus et considérant le cercle de rayon rtemp avec le dernier angle thêta, nous sommes à nouveau dans un simple problème de coordonnées polaires. Il est donc aisé de déterminer les coordonnées X et Y finales de notre point :

$$x = r_{temp} \times \cos(\theta) = r \times \cos(\phi) \times \cos(\theta)$$

$$y = r_{temp} \times \sin(\theta) = r \times \cos(\phi) \times \sin(\theta)$$

En résumé nous avons donc tout simplement :

$$x = r \times \cos(\phi) \times \cos(\theta)$$

$$y = r \times \cos(\phi) \times \sin(\theta)$$

$$z = r \times \sin(\phi)$$

Comme je l'ai dit plus haut, ce système de coordonnées nous sera très utile car nous pourrons ainsi contrôler l'orientation de la caméra à la souris : l'angle thêta contrôle l'orientation horizontale du regard (comme quand on fait « non » de la tête) et l'angle phi contrôle l'orientation verticale du regard (comme quand on fait « oui » de la tête).

Ce chapitre, d'un niveau mathématique assez facile (collège), viens trouver tout son sens dans le chapitre : le contrôle avancé de la caméra, où nous contrôlons une caméra de type *free-fly* avec la souris et le clavier.

Les matrices

Cette annexe mathématique est un passage obligé pour tous ceux qui veulent comprendre ce qui se passe réellement derrière les calculs qu'OpenGL fait pour nous.

Bien que l'*objet mathématique* matrice ne soit enseigné à l'école que dans le supérieur il n'a rien de mystique

et une utilisation basique avec peu de connaissances en est tout à fait possible.

Je ne détaillerai ici que ce que nous avons besoin de savoir pour le cours d'OpenGL et vous verrez que ça n'a vraiment rien de sorcier.

Les novices y découvriront donc un nouvel objet mathématique très pratique, ceux qui les connaissent déjà dans d'autres contextes (trouver le Kernel, le vecteur propre, oui oui je suis passé par là aussi ;) quant à eux vont enfin voir une application pratique et très simple des matrices.

L'outil matrice

Représentation

Une matrice (*matrix* en anglais) se représente comme un tableau de nombres composé de lignes et de colonnes.

$$\begin{vmatrix} 4 & 15 & 23 \\ 8 & 16 & 42 \end{vmatrix}$$

Un exemple de matrice

On désigne généralement un élément par son numéro de ligne suivi de son numéro de colonne. Par exemple pour la matrice ci-dessus, l'élément en (1,2) est 15.

Les matrices peuvent être de toutes tailles mais dans le cas d'OpenGL les matrices que nous utiliserons, implicitement ou non, seront des matrices 4x4 (4 lignes, 4 colonnes). Cette taille n'est pas anodine et est liée à l'utilisation géométrique qui en est faite en 3D (nous le verrons juste après).

Tout comme avec les nombres ou les vecteurs, nous pouvons effectuer certaines opérations sur les matrices. La seule qui nous intéresse ici est la multiplication.

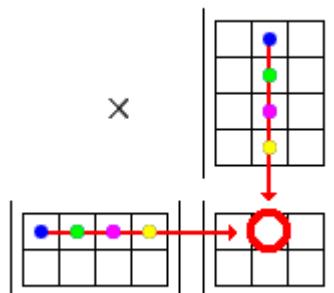
Multiplication de matrices

Multiplier deux matrices est une gymnastique facile et rigolote. Mais les erreurs quand on le fait à la main sont assez fréquentes (tous ces chiffres qui se croisent, pauvres de nous ! ^^).

Le principe est simple, pour calculer l'élément (i,j) de la matrice C, produit de deux matrices A et B, on multiplie la ligne i de A par la colonne j de B comme formalisé par :

$$C_{i,j} = \sum_{k=0}^n A_{i,k} \cdot B_{k,j}$$

Cette formule un peu barbare se résume très simplement par le schéma suivant :



Principe de la multiplication matrice x matrice

Et pour bien comprendre ce que l'on fait de chaque élément, voici un calcul étape par étape pour 2 matrices simples :

$$1. \quad \begin{array}{c|cc} & \begin{array}{cc} e & f \\ g & h \end{array} \\ \hline \begin{array}{c|cc} a & b \\ c & d \end{array} & ae \end{array}$$

$$2. \quad \begin{array}{c|cc} & \begin{array}{cc} e & f \\ g & h \end{array} \\ \hline \begin{array}{c|cc} a & b \\ c & d \end{array} & ae + bg \end{array}$$

$$3. \quad \begin{array}{c|cc} & \begin{array}{cc} e & f \\ g & h \end{array} \\ \hline \begin{array}{c|cc} a & b \\ c & d \end{array} & \begin{array}{cc} ae + bg & af + bh \\ ce + dg & cf + dh \end{array} \end{array}$$

Détail d'une multiplication matrice x matrice

Comme vous le voyez pour multiplier une ligne par une colonne, on fait la somme de chaque sous-produit entre éléments de la ligne de la matrice de gauche et de la colonne de la matrice de droite. La meilleure façon pour bien assimiler la technique est la pratique avec un papier et un crayon (ça va pour les matrices simples seulement, quand ça devient grand on s'embrouille vite :lol:).

Pour une raison évidente, comme nous multiplions des lignes par des colonnes, on ne peut multiplier deux matrices A et B que si le nombre de colonnes de A est égal au nombre de lignes de B. Mais rassurez-vous, vous n'avez pas à vous en préoccuper car dans notre cas nous utilisons des matrices carrées (autant de lignes que de colonnes) et donc cette condition est toujours vérifiée.

Contrairement aux nombres, la multiplication entre matrices n'est **pas commutative** ($A \times B \neq B \times A$). L'ordre des multiplications est donc important (comme nous le verrons lors de la *combinaison de transformations*).

Multiplication matrice x vecteur

Un vecteur n'est qu'un cas particulier de matrice avec une seule colonne. Il est donc tout à fait possible de multiplier une matrice par un vecteur (si la condition sur les tailles énoncée plus haut est remplie).

Exemple :

$$\begin{array}{c|cc} a & b \\ c & d \end{array} \times \begin{array}{c} x \\ y \end{array} = \begin{array}{c} ax + by \\ cx + dy \end{array}$$

Multiplication matrice x vecteur (le résultat est un vecteur)

Matrice identité

La matrice identité est une matrice particulière, souvent notée I, dont tous les éléments de la diagonale sont à 1 (exemple en 4x4) :

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

C'est ce que l'on appelle un élément neutre, en effet un vecteur multiplié par la matrice identité en ressort inchangé :

$$I \times \vec{v} = \vec{v}$$

Inverse

La dernière propriété qui nous intéresse sur les matrices est la notion d'inverse. Si par exemple nous avons une matrice M qui vient coder d'une certaine façon une translation de $(1,1,1)$, alors son inverse M' sera une translation de $(-1,-1,-1)$. Donc si l'on applique la matrice et son inverse à la suite à un vecteur, il reviendra donc comme il était au début. Mathématiquement on écrit ça comme ceci :

$$M' \times M = I$$

Le produit d'une matrice avec son inverse donne l'identité. L'exemple qui était pris (translation) était très simple. Dans le cas général calculer l'inverse d'une matrice est assez barbare mais je vous en donnerai l'implémentation quand nous en aurons besoin.

Transformations

Maintenant que nous comprenons l'outil mathématique matrice, je vais vous en montrer une utilisation bien pratique : les transformations géométriques.

Nous l'avons vu plus haut, multiplier une matrice par un vecteur donne un autre vecteur. Ce vecteur résultat n'est autre que la transformée du vecteur initial par la transformation « contenue » dans la matrice.

On se sert donc d'une simple multiplication matrice de transformation x vecteur de coordonnées pour obtenir les coordonnées transformées d'un point :

$$\begin{vmatrix} \text{Coordonnées} \\ \text{absolues} \\ (\text{Vecteur}) \end{vmatrix} = \begin{vmatrix} \text{Transformation} \\ \text{repère} \\ (\text{Matrice}) \end{vmatrix} \times \begin{vmatrix} \text{Coordonnées} \\ \text{locales} \\ (\text{Vecteur}) \end{vmatrix}$$

Comme il est possible en utilisant une matrice de transformer un vecteur par un autre, nous allons utiliser cette propriété pour stocker dans la matrice des éléments pour exécuter chaque transformation élémentaire dont nous avons besoin en 3D : la rotation, la translation, et le changement d'échelle :

$$\begin{array}{c|c|c|c} & \text{rotation} & \text{scale} & \text{translation} \\ \left[\begin{array}{c} x_{absolu(hom)} \\ y_{absolu(hom)} \\ z_{absolu(hom)} \\ w_{absolu(hom)} \end{array} \right] & = & \left[\begin{array}{cccc} M_{1,1} & M_{1,2} & M_{1,3} & M_{1,4} \\ M_{2,1} & M_{2,2} & M_{2,3} & M_{2,4} \\ M_{3,1} & M_{3,2} & M_{3,3} & M_{3,4} \\ M_{4,1} & M_{4,2} & M_{4,3} & M_{4,4} \end{array} \right] & \times \left[\begin{array}{c} x_{local} \\ y_{local} \\ z_{local} \\ 1 \end{array} \right] \end{array}$$

Vous trouverez l'explication numérique de chaque transformation un peu après.

À quoi servent donc les éléments du bas qui n'ont d'utilité pour aucune transformation apparemment ?

Le fait que la matrice fasse 4 colonnes de large est fixé pour nos besoins (rotation, scale, translation). Mais pour pouvoir faciliter les choses et plus tard savoir, à partir d'une matrice, trouver son inverse, il **faut** que celle-ci soit **carrée** (autant de lignes que de colonnes). C'est pourquoi la matrice de transformation possède aussi 4 lignes au lieu de 3. Il s'avère qu'une partie de la 4e ligne est utilisée par OpenGL pour la projection mais nous n'avons pas besoin d'en savoir plus sur le sujet.

Coordonnées homogènes

Ok pour la taille de la matrice mais nos vecteurs sont en 3D hein, c'est quoi ce 4e élément ? C'est le temps ?

Non non, ici nous ne travaillons pas en 4D :) mais bien en 3D. Vous le savez maintenant pour pouvoir multiplier une matrice et un vecteur il faut une condition particulière sur les dimensions : la taille du vecteur doit être égale au nombre de colonnes de la matrice, ici 4. Nous introduisons donc **temporairement** une 4e coordonnée (appelée w) pour exprimer et former ce que l'on appelle une coordonnée **homogène**.

Pour passer d'un vecteur 3D (x,y,z) à son équivalent en coordonnées homogènes il suffit de rajouter un 1 soit (x,y,z,1). Par contre pour passer d'un vecteur en coordonnées homogènes (x,y,z,w) à sa version 3D normale il faut diviser les 3 premières coordonnées par la dernière soit : (x/w,y/w,z/w).

Dans notre cas (les transformations), la 4e coordonnée n'est **jamais** modifiée et reste donc toujours à 1. Nous n'avons donc pas à nous en soucier.

Voyons donc maintenant ce qui est mis réellement dans la matrice pour coder les transformations.

Identité

$$\left| \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right| \times \left| \begin{array}{c} a \\ b \\ c \\ 1 \end{array} \right| = \left| \begin{array}{c} a \\ b \\ c \\ 1 \end{array} \right|$$

Nous l'avons vu plus tôt cette matrice ne fait rien. Ça peut paraître inutile mais c'est ce que nous utiliserons pour réinitialiser la matrice à un état **connu** dont nous sommes sûrs qu'il n'affecte pas les coordonnées des vertices.

Translation

Une translation de (x,y,z) s'écrit matriciellement :

$$\begin{vmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} a \\ b \\ c \\ 1 \end{vmatrix} = \begin{vmatrix} x+a \\ y+b \\ z+c \\ w \end{vmatrix}$$

Et en effet si on effectue la multiplication d'un vecteur quelconque (a,b,c) par cette matrice de translation, on obtient un vecteur résultat qui n'est ni plus ni moins que l'image du premier vecteur par cette translation.

Changement d'échelle (Scale)

Un changement d'échelle s'écrit matriciellement :

$$\begin{vmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} a \\ b \\ c \\ 1 \end{vmatrix} = \begin{vmatrix} x \times a \\ y \times b \\ z \times c \\ 1 \end{vmatrix}$$

En effectuant la multiplication de la matrice de changement d'échelle par un vecteur quelconque on voit bien que ses composantes sont multipliées par chacune des composantes du changement d'échelle.

Rotation

Attention accrochez-vous, une rotation d'angle θ , autour d'un axe quelconque (x,y,z) s'écrit matriciellement :

$$\begin{vmatrix} xx(1 - \cos(\theta)) + \cos(\theta) & xy(1 - \cos(\theta)) - z \sin(\theta) & xz(1 - \cos(\theta)) + y \sin(\theta) & 0 \\ yx(1 - \cos(\theta)) + z \sin(\theta) & yy(1 - \cos(\theta)) + \cos(\theta) & yz(1 - \cos(\theta)) - x \sin(\theta) & 0 \\ xz(1 - \cos(\theta)) - y \sin(\theta) & yz(1 - \cos(\theta)) + x \sin(\theta) & zz(1 - \cos(\theta)) + \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

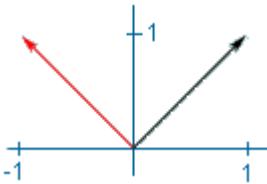
Ça fait peur hein ! :diable: Ici c'est le cas le plus général qui soit. Simplifions vite cela en prenant le cas particulier de la rotation d'angle θ (et oui toujours) autour de l'axe Z (0,0,1). Tout ce qui concerne x et y s'en va et on se retrouve avec une matrice qui fait nettement moins peur :

$$\begin{vmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Pour vérifier que cela marche bien comme on l'entend, prenons encore un cas plus particulier : celui de la rotation de 90° autour de l'axe Z et appliquons-la au vecteur (1,1) :

$$\begin{vmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 1 \\ 1 \\ 0 \\ 1 \end{vmatrix} = \begin{vmatrix} -1 \\ 1 \\ 0 \\ 1 \end{vmatrix}$$

Par le calcul on obtient donc le vecteur (-1,1) en rouge qui est en effet la rotation de 90° du vecteur (1,1) en noir.



Combinaison de transformations

Pour l'instant nous avons vu comment écrire chaque transformation de base matriciellement. Mais qu'en est-il quand nous souhaitons en faire plusieurs à la suite ?

Eh bien il suffit de les multiplier ! On fait ce qu'on appelle des *multiplications à droite*. Si nous voulons d'abord faire une translation puis une rotation, la matrice représentant la transformation totale sera : Translation x Rotation.

Exemple :

Translation de (1,0,0) suivie d'une rotation de 90° autour de l'axe Z :

$$\begin{vmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

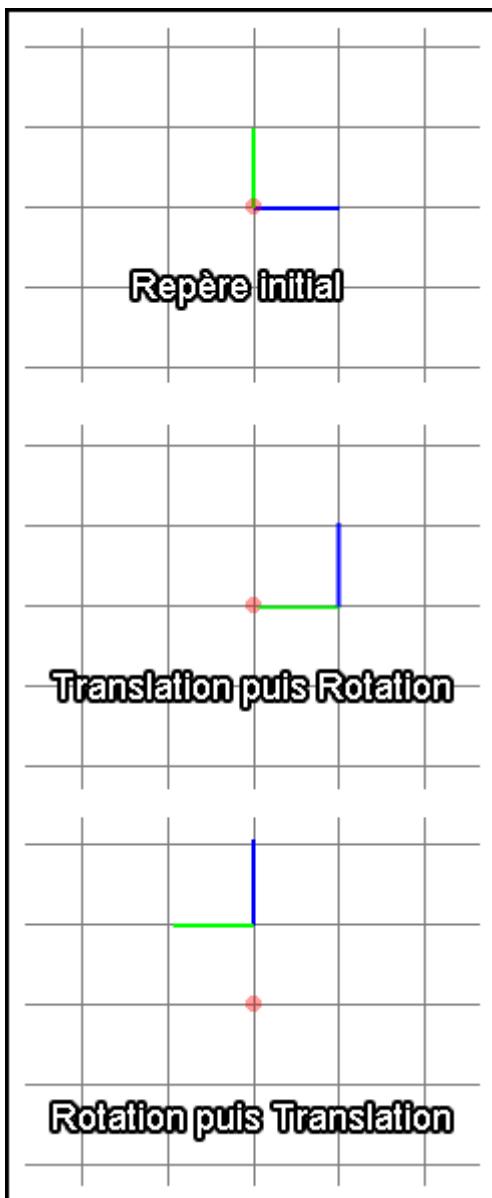
Ordre des transformations

L'ordre dans lequel sont faites les transformations est important. En effet faire une translation suivie d'une rotation n'a pas le même effet que faire une rotation suivie d'une translation.

Reprendons l'exemple de la translation de (1,0,0) mais cette fois-ci précédée d'une rotation de 90° autour de l'axe Z :

$$\begin{vmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

La matrice finale n'est pas la même que précédemment. Et en effet si on compare graphiquement, l'effet est totalement différent :



Importance de l'ordre des transformations

Ce chapitre assez mathématique peut nécessiter plusieurs lectures et je vous conseille de le relire conjointement avec le chapitre sur les transformations (<http://www.siteduzero.com/tuto-3-6304-1-les-transformations.html>) pour avoir des exemples appliqués à OpenGL.

Si vous voulez en savoir un peu plus sur l'outil matrice je vous conseille d'aller faire un tour sur Wikipédia ([http://fr.wikipedia.org/wiki/Matrice_\(math%C3%A9matiques\)](http://fr.wikipedia.org/wiki/Matrice_(math%C3%A9matiques))).

Et enfin si vous demandez d'où je tire toutes ces informations barbares sur les représentations matricielles des transformations, je vous conseille, si ce n'est déjà fait, de jeter un coup d'œil dans l'**excellent** documentation d'OpenGL (http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/), notamment les pages de :

- `glLoadIdentity`
(http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/loadidentity.html) ;
- `glTranslate` (http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/translate.html) ;
- `glRotate` (http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/rotate.html) ;
- `glScale` (http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/scale.html) ;
- `glMultMatrix` (http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/multmatrix.html)
(pour la partie sur la combinaison de transformations).

Toutes ces informations combinées nous permettent de mieux comprendre comment fonctionne OpenGL et surtout d'être capables d'implémenter nous-mêmes les transformations (nous serons amenés à les utiliser).

Créer une vidéo de votre programme

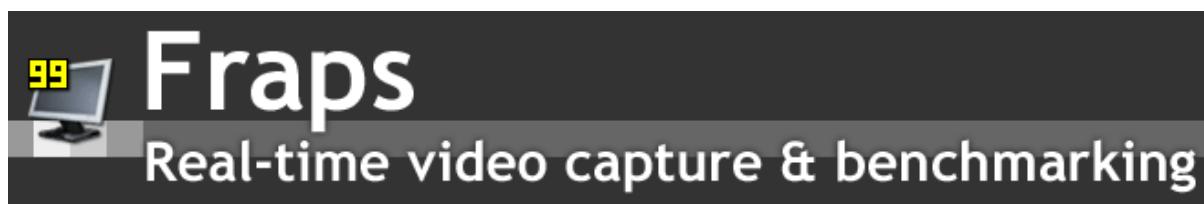
Vous avez peut-être été surpris de voir apparaître dans mon tutoriel des vidéos montrant les résultats animés des chapitres.

Cette annexe a pour but de vous expliquer comment procéder pour créer des vidéos montrant vos jeux/animations 3D sous Windows.

Si vous n'avez pas Windows, n'ayez crainte ! Vos programmes OpenGL sont censés être portables si vous suivez correctement le tutoriel jusqu'à présent. Il vous suffit d'utiliser l'ordi d'un ami (on a tous un ami sous Windows :p) en lui emmenant vos sources pour les compiler sous Windows avec Code::Blocks par exemple.

Enregistrer la vidéo

La première étape est donc l'enregistrement de la vidéo. Pour ce faire nous allons utiliser un logiciel bien pratique : Fraps.



Télécharger Fraps

Ce logiciel est disponible gratuitement à l'adresse suivante : <http://www.fraps.com/download.php> (<http://www.fraps.com/download.php>)

La version gratuite possède cependant certaines limites :

- les vidéos seront limitées à 30 secondes ;
- un petit texte « www.fraps.com (<http://www.fraps.com/>) » apparaîtra en haut des vidéos.

Le texte n'est pas trop handicapant, car discret et rien ne vous empêche de faire un montage de plusieurs vidéos de 30 secondes (une vidéo de démo d'un jeu ne s'attarde jamais bien longtemps sur un même endroit).

Configurer Fraps

Après avoir installé et lancé Fraps, rendez-vous dans l'onglet « Movies » pour configurer les options de capture de vidéo :



1. Le répertoire où seront stockées les vidéos.

Pour des raisons d'optimisation de performances, les vidéos en sortie de Fraps seront très peu compressées. Pensez donc à spécifier un répertoire sur un disque où il y a de la place.

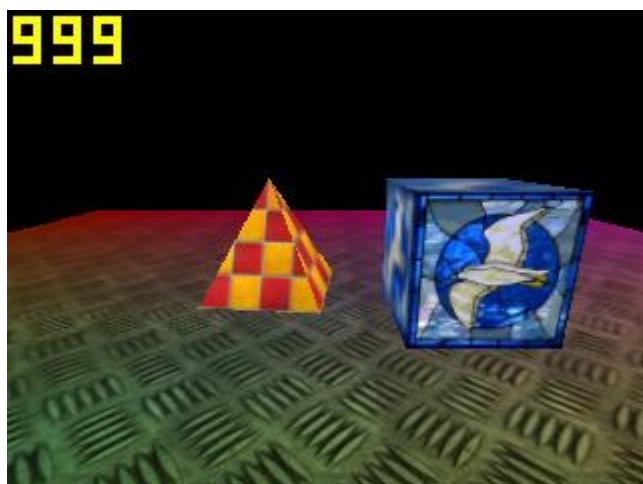
2. Utilisez de préférence *Full-Size* et changez la taille de votre fenêtre dans votre code. Cela évitera à Fraps d'avoir à redimensionner les images à la volée. En full Size avec une fenêtre de 640x480, la vidéo sera de taille 640x480.
3. Choisissez le nombre d'images par seconde désirées pour votre vidéo. **30** est un très bon compromis entre fluidité et taille de la vidéo.

Pour respecter le framerate choisi, Fraps viendra limiter lui-même les FPS de votre application. Je vous conseille donc fortement de désactiver la limitation des FPS dans votre code avant d'enregistrer votre vidéo (il suffit simplement de commenter l'appel à `SDL_Delay`).

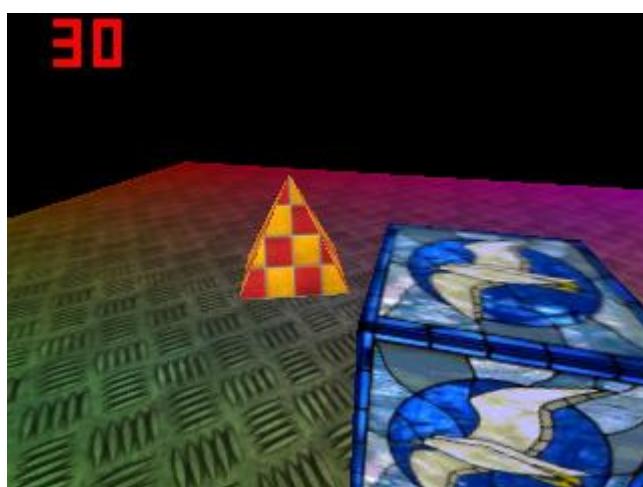
4. Choisissez une touche qui déclenchera, une fois votre application lancée, l'enregistrement de la vidéo. Vous pouvez garder la touche par défaut, ou changer si cette touche est déjà utilisée dans votre programme.
5. Enfin minimisez Fraps (ça ne sert à rien de laisser la fenêtre visible) dans le systray.

Enregistrement de la vidéo

En lançant votre application avec Fraps actif, vous verrez apparaître un nombre sur votre fenêtre. C'est le nombre d'images par seconde. En effet Fraps sert à faire des vidéos mais peut aussi vous aider à tester les performances de votre application.



Quand vous désirez commencer l'enregistrement, appuyez sur la touche choisie. Le nombre devient alors rouge pour vous signaler que Fraps a pris le contrôle du nombre d'images par seconde. Au bout de 30 secondes (pour la version gratuite), ou si vous appuyez à nouveau sur la touche, l'enregistrement s'arrête. Vous pouvez recommencer autant de fois que vous voulez sans quitter votre application tant que vous avez de la place sur votre disque dur. :p



En se rendant dans le répertoire des vidéos, on voit un nouveau fichier apparaître avec le nom de votre exécutable ainsi que la date. Ainsi il n'y pas de risque d'écrasement entre deux vidéos consécutives.

E:\Videos\Fraps	Nom	Taille	Type
Tâches vidéo	sdlglapp 2006-05-11 09-57-34-89....	54 289 Ko	VLC media file

Cependant comme je vous l'ai dit plus haut, la vidéo n'est pas compressée et prend donc pas mal de place. De plus, elle n'est lisible pour l'instant que par vous (ou toute autre personne ayant installé Fraps). Il va donc falloir l'encoder dans un format courant : **Xvid** (alternative non propriétaire au divX).

Encoder la vidéo

Pour encoder la vidéo nous allons utiliser un logiciel libre : **VirtualDub**.

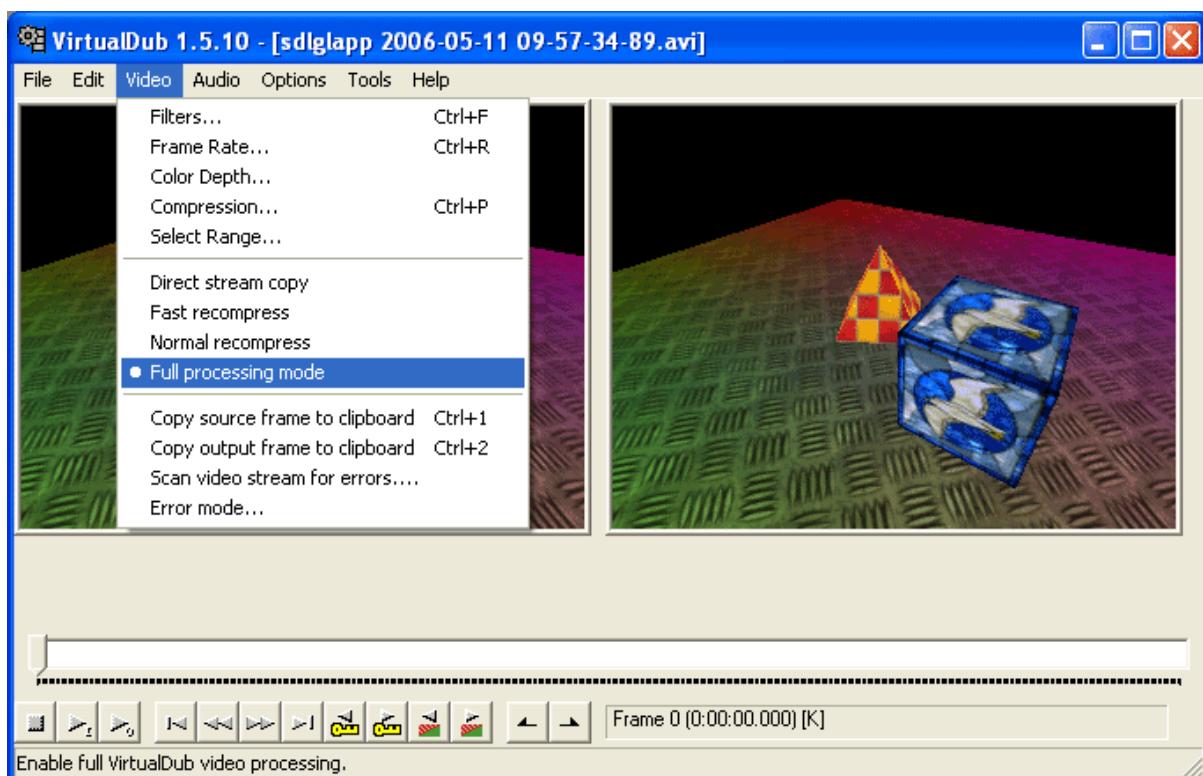
Vous pouvez le télécharger à l'adresse suivante : <http://virtualdub.sourceforge.net/> (<http://virtualdub.sourceforge.net/>).

Comme nous allons utiliser le format avi/Xvid, il nous faut les codecs appropriés que vous pouvez télécharger ici : XviD codec vX.X.X for Windows(by Koepi) (<http://www.xvidmovies.com/codec/>).

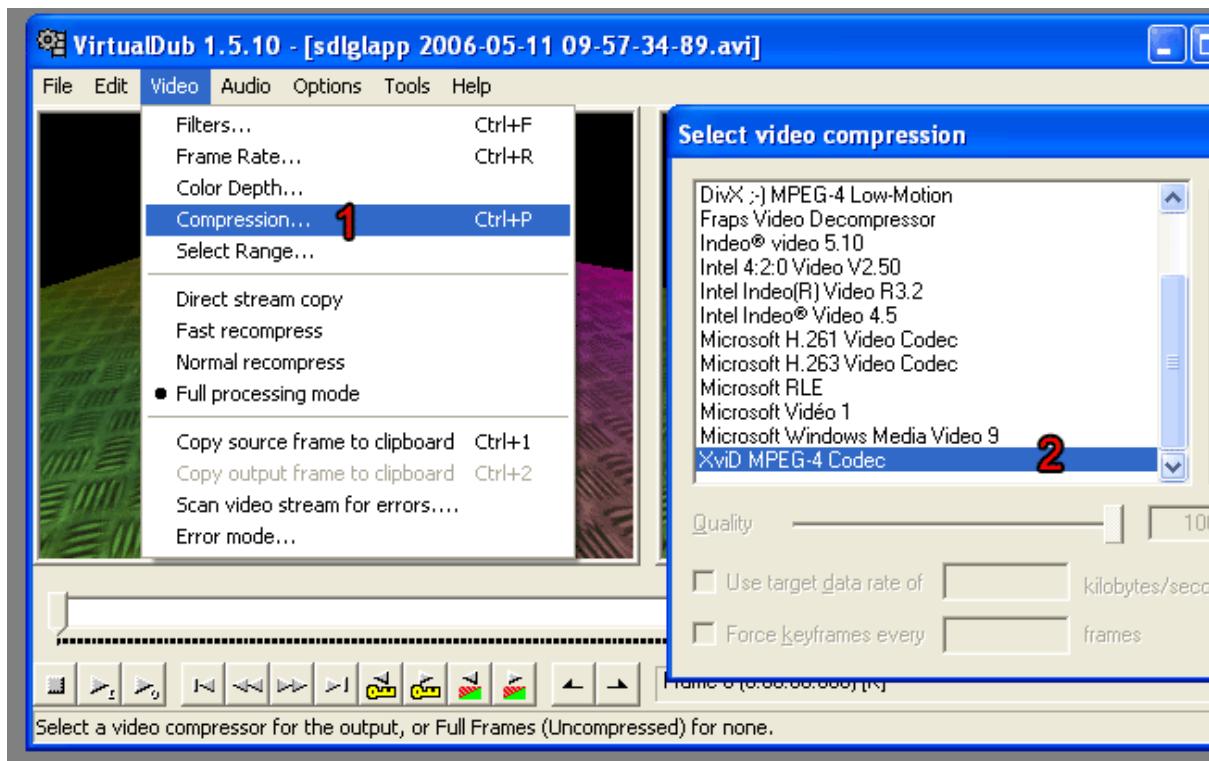
Même si vous utilisez comme moi le très bon lecteur VLC (<http://www.videolan.org/vlc/>) qui ne nécessite pas de codecs tiers, il est nécessaire de posséder les codecs XviD pour pouvoir **encoder** la vidéo.

Configuration de VirtualDub

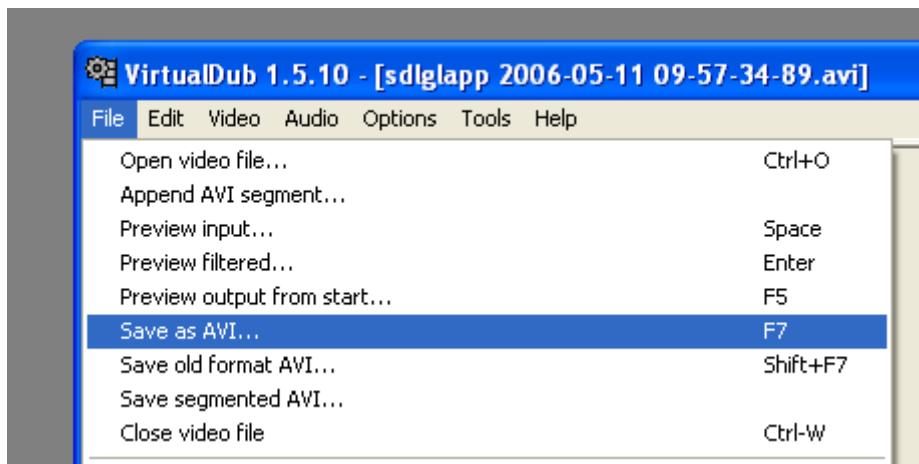
Après avoir installé VirtualDub et les codecs, lancez VirtualDub et ouvrez votre vidéo. Dans le menu « *Video* » vérifiez que « *Full processing mode* » est bien sélectionné :



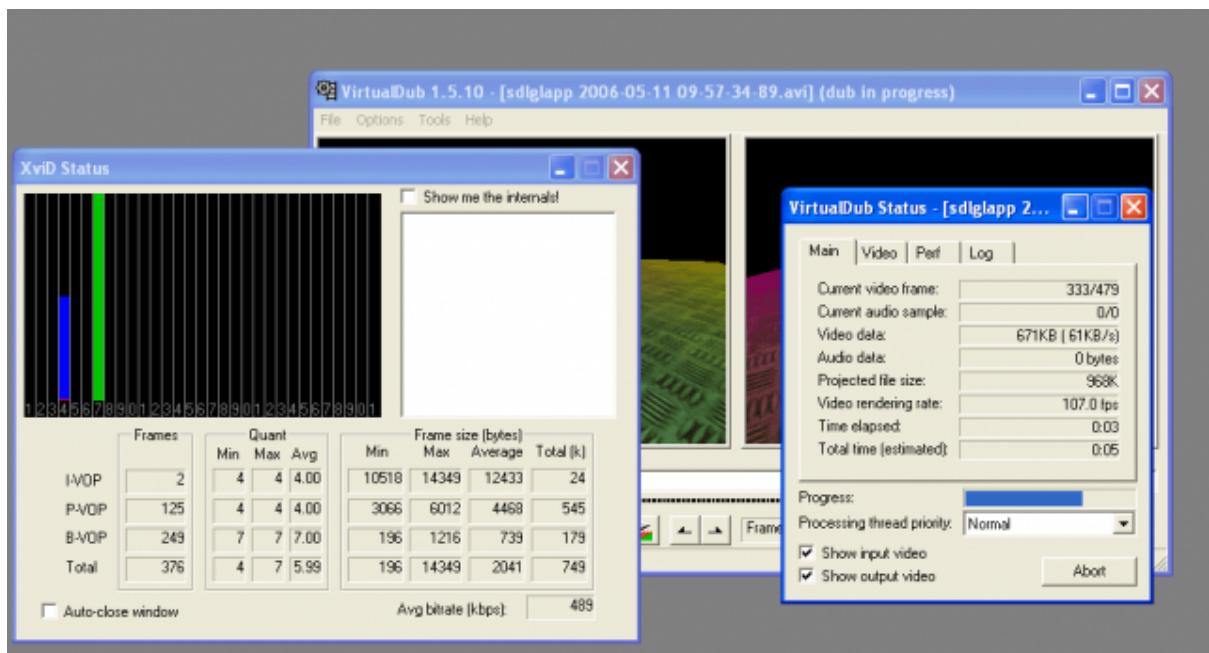
Toujours dans ce même menu « *Video* », allez dans « *Compression...* » et sélectionnez le codec de compression désiré : ici **XviD**. (Si vous ne choisissez pas de compression, la vidéo finale sera très volumineuse).



Il ne reste plus qu'à lancer l'encodage en allant dans le menu « *File* » > « *Save as AVI...* » :



VirtualDub vous affiche des informations le temps de l'encodage (vraiment rapide pour une vidéo de 30 secondes) :



Enfin, en regardant le répertoire où la vidéo a été enregistrée, on constate que la taille a énormément diminué :

E:\Videos\Fraps		
Tâches vidéo	Nom	Taille
	ma_video.avi	1 018 Ko
	sdlglapp 2006-05-11 09-57-34-89....	54 289 Ko

Vous pouvez maintenant distribuer vos vidéos librement pour montrer vos prouesses en OpenGL !

Comme vous pouvez le voir l'utilisation combinée de deux logiciels puissants, Fraps et VirtualDub, permet très facilement d'enregistrer des vidéos de vos applications 3D.

Pour vous faire saliver un peu sur la suite du tutoriel voici justement une vidéo que j'ai réalisée avec la technique expliquée dans ce chapitre :



Téléchargez une vidéo annonçant un chapitre futur de ce tuto (7.8 Mo) (<http://illament.free.fr/sdz/engine2.avi>)