TD 1 - Découvrir OpenGL

Xavier Décoret

Résumé

Le but de ce TD est de découvrir les bases d'OpenGL
- décrire une scène 3D ;
- obtenir une image de cette scène ;
- découvrir les fonctions de bases OpenGL

1 Préparation pour les TD

Les instructions suivantes permettent de configurer votre environnement pour simplifier les TDs (utiliser des données sur le compte du professeur, etc.). Vous n'avez pas besoin de comprendre à quoi ca sert (mais vous pouvez demander des explications si vous êtes curieux !), et ca ne sera à faire qu'une seule fois.

Par contre, on suppose un minimum d'habitude avec linux, la navigation des répertoires, l'utilisation d'un éditeur de texte (au choix kate, gedit, emacs, vi pour les puristes), l'utilisation des *man pages*. Si vous n'êtes pas à l'aise, demandez de l'aide.

1. Éditer (ou créer) le fichier `.login` et rajouter
   `setenv QMAKEFEATURES /profs/xdecoret/qmakefeatures`
2. Éditer (ou créer) le fichier `.tcshrc` et rajouter
   `source /profs/xdecoret/.tcshrc-common`
3. Vous déconnecter et vous reconnecter

2 Se préparer pour OpenGL

2.1 Introduction

OpenGL est une API C qui permet d'envoyer des ordres à la carte graphique pour décrire une scène 3D et obtenir une image 2D de cette scène.

Pour cela, OpenGL fonctionne comme une machine à état. Il y a par exemple une couleur courante, une matrice de transformation courante, etc. Certains de ces états correspondent directement à un état de la carte graphique ; d'autres

sont des méta-combinaisons qui permettent d'abstraire différents types de cartes graphiques. OpenGL a donc besoin d'espace mémoire pour stocker ces états.

D'autre part, OpenGL gère le lien entre les fonctionnalités 3D de la carte graphique et la fenêtre (au sens *Windows Manager*) dans laquelle le résultat des calculs 3D sont affichés. Pour cela, OpenGL a aussi besoin d'espace mémoire ou stocker les "images" produites.

Tout cela est encapsulé dans ce qu'on appele un *contexte* OpenGL. En gros, c'est l'objet qu'il faut créer pour pouvoir "faire" de la 3D avec OpenGL. La création de contexte est assez technique, et dans 99% des cas, il n'est pas nécessaire de savoir comment cela marche : on utilise une librairie de haut-niveau qui gère cela pour nous. Pour ce cours, nous allons utiliser Qt4, qui est une librairies extrêmement puissante, dont nous n'utiliserons pas tout, mais qui va nous rendre de grands services, et notamment gérer les contextes OpenGL pour nous.

2.2 Première application

Voici le code à créer pour une application de base qui ouvre une fenêtre avec un contexte OpenGL associé.

```
1  #include <QtOpenGL>
2
3  class Viewer : public QGLWidget
4  {
5  public:
6    Viewer()
7      : QGLWidget()
8      {
9      }
10 protected:
11   virtual void initializeGL()
12     {
13       // OpenGL commands to setup the state machine
14     }
15   virtual void paintGL()
16     {
17       // OpenGL commands to "render" a 3D view in the widget
18     }
19   virtual void keyPressEvent(QKeyEvent* e)
20     {
21       switch (e->key())
22       {
23       case Qt::Key_Escape:
24         close();
25         break;
26       }
27     }
28 };
29
30 int main(int argc,char** argv)
31 {
```

```
32    QApplication application(argc,argv);
33
34    Viewer v;
35    v.show();
36
37    return application.exec();
38  }
```

2.3 Comment compiler ?

Qt4 est une librairie multi-platforme (linux, window, mac, ...). Comme les environnements de développement diffèrent d'une plateforme à l'autre (et même entre plateformes !), Trolltech a "abstrait" ce système avec un outil nommé `qmake`. Cet outil prend en entrée un fichier de description de projet (avec l'extension `.pro`) et produit en sortie ce qu'il faut pour compiler le projet sur une plateforme cible (e.g. un `Makefile` sous linux, un projet Visual sous Windows, etc.).

Le fichier projet permet plein de choses compliquées, mais la plupart du temps, il est très simple ! En voici un exemple qui parle de lui-même.

```
TEMPLATE = app

CONFIG *= qt warn_on debug
QT     *= opengl

# Ceci est un commentaire

SOURCES *= \
    main.cpp
```

Le cycle de développement est le suivant :

1. Se mettre dans le répertoire où il y a le `.pro`
2. Lancer `qmake myfile.pro` pour générer un `Makefile`
3. Lancer `make` pour compiler
4. Si la compilation marche, lancer le programme (il s'appelle `myfile`)
5. Modifier votre code ou le `.pro` (si vous rajoutez des fichiers au projet)
6. Recommencer à l'étape 3 !

Quelques remarques en vrac :

– Il n'y a (normalement) besoin de lancer `qmake` qu'une seule fois. En effet, le `Makefile` fabriqué est assez intelligent. Quand on lance `make`, il vérifie si le `.pro` qui a servi à le générer a changé. Si oui, il commence par relancer `qmake` pour se regénérer !

– Cependant, si vous changez les directives `#include` dans votre code, il vaut mieux relancer `qmake`.
– Pour changer le nom de l'exécutable, utiliser dans le `.pro`
`TARGET = my_exe_name`
– Si il n'y a qu'un seul `.pro` dans le répertoire (c'est généralement le cas !), il suffit de lancer `qmake` sans passer le nom du `.pro` en paramètre.
– L'habitude veut qu'on nomme le `.pro` comme le répertoire (c'est pratique pour des compilations récursives de gros projets).

2.4 Pour aller plus loin

Qt4 vient avec une aide intégrée très bien faite. Lancer `assistant` (ou `assistant-qt4` si vous avez Qt3 et Qt4 installé) pour en savoir plus. Vous pouvez notamment faire la section "Tutorials".

3 Premier rendu 3D

L'application compilée précédemment affiche... un beau fond noir ! Normal, nous n'avons rien mis dans `paintGL()`. Essayons nos premières commandes :

```
15  virtual void paintGL()
16    {
17      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
18
19      glViewport(0,0,width(),height());
20      glMatrixMode(GL_PROJECTION);
21      glLoadIdentity();
22      gluPerspective(40.0,1.0,0.1,10.0);
23      glMatrixMode(GL_MODELVIEW);
24      glLoadIdentity();
25      glTranslatef(0.0,0.0,-4.0);
26
27      glBegin(GL_QUADS);
28      glVertex3f(-0.5,-0.5,0.0);
29      glVertex3f( 0.5,-0.5,0.0);
30      glVertex3f( 0.5, 0.5,0.0);
31      glVertex3f(-0.5, 0.5,0.0);
32      glEnd();
33    }
```

Compiler ce code, lancer l'application. Que se passe-t-il quand vous redimensionnez la fenêtre ? Pouvez-vous expliquer pourquoi (*hint* : pensez à l'*aspect ratio*). En regardant la documentation de `gluPerspective` et en utilisant les fonctions `width()` et `height()` qui retourne la dimension en pixels de la fenêtre, corrigez le problème.

Modifier le viewport pour dessiner dans le cadre supérieur droit de la fenêtre. Utiliser une boucle et des viewport changeant pour dessiner quatre vues de la scène dans les quatre quadrants de la fenêtre.

Utiliser `glClearColor()` pour afficher un fond gris moyen plutôt que noir.

3.1 Pour aller plus loin

OpenGL étant une machine à état, il n'est pas forcément nécessaire d'appeller `glClearColor()` (ou même `gluPerspective()`) à chaque ré-affichage (c'est à dire chaque appel de `paintGL()`. On peut faire l'appel une bonne fois pour toute dans `initializeGL()` par exemple.

Il faut aussi faire attention à ce que l'état est conservé entre la fin d'un appel à `paintGL()` et le début de l'appel suivant (prochain réaffichage, par exemple parce que la fenêtre est dimensionné). Un symptôme classique est d'avoir un rendu initial, et un rendu différent quand on redimensionne la fenêtre. C'est parce que l'état lors du premier passage dans `paintGL()` n'est pas le même que dans les passages suivants ! Une bonne solution pour cela est d'entourer le code dans `paintGL()` par des `glPushAttrib()` et `glPopAttrib()`. Consultez la documentation de ces fonctions pour en savoir plus.

4 Découverte

Recherchez dans la documentation à quoi servent les fonctions suivantes :
- `gluLookAt()`
- `gColor3f()`
- `glPolygonMode()`

À l'aide de ces fonctions, modifier le programme pour qu'il affiche un cube :
- dont les 6 faces sont de couleurs différentes,
- que l'on peut voir en "fil de fer" en appuyant sur `Key_W`
- autour duquel on puisse "tourner" en appuyant sur les flèches gauche (`Key_Left`) et droite (`Key_Right`).

4.1 Pour aller plus loin

Les fonctions OpenGL existent souvent en plusieurs version qui prennent des paramètres de types différents. On trouve ainsi

```
glColor{1|2|3}{f|d|i|ui}
glColor{1|2|3}{f|d|i|ui}v
```

On peut par ainsi spécifier des sommets du plan (x, y) à coordonnées entières stockés dans des tableaux C avec du code comme suit :

```
int a = {-1,-1};
int b = {+1,-1};
int c = { 0,+1};
glVertex2iv(a);
glVertex2iv(b);
glVertex2iv(c); // same effect than glVertex2i(c[0],c[1])
```

5 Exercices obligatoires

Afficher maintenant un cylindre plutôt qu'un cube (garder un rendu fil de fer : les faces "remplies" seront l'objet du prochain TD). Si vous y arrivez facilement, essayer d'afficher une sphère. Écrire du code de manière à ce que la *tesselation*, c'est à dire la finesse avec laquelle vous approximez une forme lisse (cylindre, sphère) par une forme polygonale, soit paramétrable.

Encapsuler le code pour le cube, le cylindre et la sphère dans trois fonctions

Utiliser les fonctions précédentes pour *instancier* des primitives géométriques et représenter un bonhomme simplifié (une tête, un tronc, des hanches et des épaules, et quatre membres). Pour cela :
- modifier la matrice `MODELVIEW` avec des fonctions comme `glTranslatef()`, `glRotatef()`, `glScalef()`
- utiliser `glPushMatrix()` et `glPopMatrix()`

6 Exercices avancés

Utiliser maintenant le code précédent pour instancier $n \times m$ bonhomme positionnés sur une grille 2D dans le plan (x, y). Essayer des valeurs de plus en plus grandes de n et m et regarder comment l'affichage 3D se met à ramer (regarder le temps qu'il faut pour rafraîchir la fenêtre après un redimensionnement). Si vous voulez faire des mesures, regarder la classe `QTime` de Qt, et la fonction `glFinish()`.

Essayer d'optimiser le rendu en utilisant :
- du *back face culling* (regarder la doc. de `glCullFace()` ;
- des *display lists* (regarder la doc. de `glCallList()` ;
- des *indexed face sets* ;
- des *vertex array elements* pour les plus avancées (technique !).