Premiers pas avec OpenGL et SDL 2

Objectifs : le but de ce TP est de se familiariser avec les commandes de base des deux bibliothèques particulières utilisées lors de ce projet :
- *OpenGL* pour la création d'une scène 3D et sa projection sur un plan 2D. *OpenGL* ne possède pas de sortie graphique, ce qui peut paraître curieux au prime abord.
- *SDL2* : pour la création d'une fenêtre de visualisation de la projection de la scène 3D *OpenGL,* mais aussi pour la gestion des événements clavier.

Premier projet SDL2 / OpenGL

Première compilation

Sous *CodeBlocks*, la génération d'une application passe par la création d'un projet qui regroupe l'ensemble des sources et des bibliothèques utiles. Ces opérations demandant une bonne connaissance des bibliothèques et du processus de compilation, elles ont été réalisées pour vous. Un projet *CodeBlocks* autonome et fonctionnel est disponible sur *Dokeos*.

Travail à effectuer :

- Récupérer le projet correspondant à ce TP (sous forme d'une archive zip) sur *Dokeos*.
- Décompresser l'archive sur un **dossier local** sur votre machine de travail (pas sur le réseau). Le chemin d'accès à ce dossier ne doit contenir **ni espace**, **ni caractères spéciaux** (accents…).
- Ouvrir le projet depuis l'explorateur *Windows* (fichier « *TP_OpenGL_SDL2.cbp* »). Le projet contient, entre autres, un fichier source nommé « *first_prog.cpp* ». C'est ce fichier qui contient notamment la fonction *main* du programme. Il devra être complété, ainsi que d'autres, dans la suite du TP.
- Vérifier que la compilation de votre projet est fonctionnelle en **re**construisant l'exécutable : menu « *Build* → *Rebuild* ». Si tout se passe bien, le message suivant devrait s'afficher dans la console de *Code::Blocks* :

```
Process terminated with status 0 (0 minutes, 0 seconds)
0 errors, 0 warnings (0 minutes, 0 seconds)
```

Le but est maintenant d'analyser le code source minimal fourni, qui dessine sur fond noir un carré multicolore à l'intérieur d'une fenêtre. Pour cela :

- Exécuter le programme précédemment compilé (raccourci *F9*)
- Etudier le code source afin d'essayer de comprendre le fonctionnement du programme et des bibliothèques *OpenGL* et *SDL2* (notamment le rôle des différentes fonctions utilisées).

Un cube multicolore…

En s'inspirant du code source précédent, **ajouter les instructions nécessaires** à l'affichage d'un cube aux sommets multicolores, centré sur le point de coordonnées *(0, 0, 0)*.

Transformations : rotations, translations et mise à l'échelle

Statiques

- Observer le résultat. Les faces du cube qui devraient être cachées le sont elles réellement ? Pour avoir un rendu correct, il faut activer le tampon de profondeur (« *Z-Buffer* »). Pour cela, ajouter à la fin de la fonction *initGL* :

  glEnable(GL_DEPTH_TEST);

  et modifier la fonction *render* en remplaçant :

  glClear(GL_COLOR_BUFFER_BIT);

  par :

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

- Diviser par 2 l'ensemble des dimensions du cube à l'aide de la fonction *glScaled*.
- Déplacer le centre de gravité du cube au point de coordonnées (0, 2, 4).
- Vérifier le résultat obtenu lorsque le tampon de profondeur est activé.

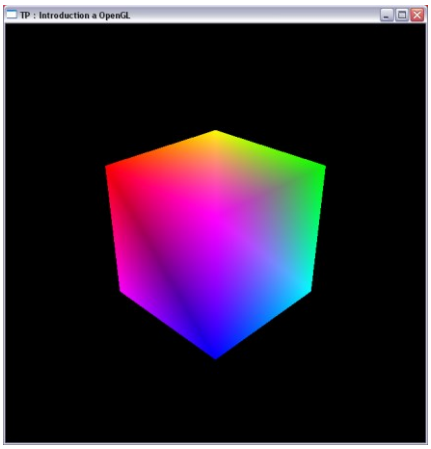

Figure 1

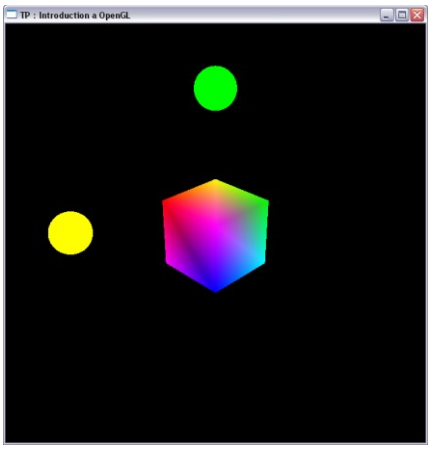

Figure 2

Animation…

L'objectif de cette partie est d'ajouter deux « satellites » (des sphères), tournant autour du cube précédent dans les plans *Oxy* et *Oxz* respectivement. On repositionnera ici le centre de gravité du cube en *(0, 0, 0)* (cf. Figure 3).

Afin de gérer l'affichage et les animations de n'importe quel type de forme ultérieurement (pour le projet par exemple), il a été nécessaire de définir un certain nombre de classes et de méthodes associées. On propose le modèle objet représenté ci-après (cf. diagramme de classes Figure 3). C'est un des modèles possibles, qui pourra être enrichi et modifié par la suite.

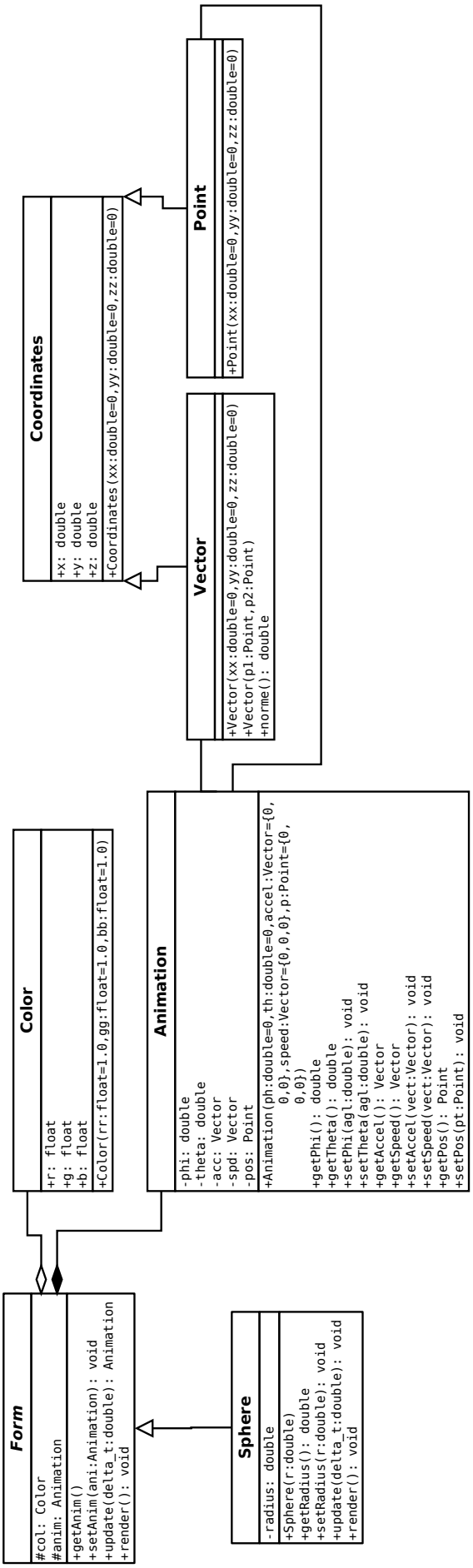

Figure 3 - Diagramme de classes proposé

Afin de représenter les objets (formes géométriques simples pour l'instant) à rendre dans l'espace, de manière générique, une classe mère abstraite *Form* est déclarée. Cette classe contient un champ donnant la couleur de la forme et un objet de type *Animation* (servant aux animations, que l'on détaillera plus tard). Elle dispose d'une méthode *render*, qui, lorsqu'elle est appelée, effectue le rendu *openGL* de la forme dans la fenêtre. Le rendu d'une forme n'étant possible que si l'on connaît le type de la forme (sphère, parallélépipède, plan…), la méthode *render* est donc une méthode virtuelle. On propose ensuite la modélisation d'une classe fille *Sphere*, permettant de représenter et d'effectuer le rendu 3D d'une sphère. Cette sphère est définie par son rayon. Sa position (centre), comme celle de toutes les autres formes, étant stockée dans la classe Animation (cf. ci-dessous). La classe *Sphere* dispose d'un constructeur et des habituels accesseurs, ainsi que des méthodes *update* et *render*.

La classe *Animation* est dédiée à la gestion des animations (translations de formes au cours du temps, rotations multiples, déplacements à l'aide de commandes utilisateurs…). Elle pourra être complétée en fonction des besoins de l'application (modélisations spécifiques). La classe *Animation* contient les champs suivants : la position du repère local dans lequel est tracée la forme associée ; les angles *phi* et *theta* qui définissent l'orientation du repère local par rapport au repère global ; les vecteurs vitesse et accélération du repère local de la forme. Elle dispose d'un constructeur et de divers accesseurs. La méthode *update* de chaque forme doit être appelée à intervalles de temps réguliers (cette action est réalisée par la fonction *update* générale, décrite ci-après). Elle actualise (via un modèle physique dédié, ou la gestion des entrées clavier ou souris…) tout ou partie des champs de l'animation. Si cette actualisation est réalisée régulièrement et que la scène *openGL* est rafraîchie à plus de 25 images par seconde, on obtiendra un mouvement régulier des formes qui composent la scène.

Afin d'automatiser l'animation et le rendu systématique d'un nombre quelconque de formes, une liste de pointeurs de formes (*forms_list*) est créée. La fonction de rendu général *render* prend comme paramètre cette liste (ainsi que la position de la caméra) et appelle la méthode *render* de chacune des formes de la liste. Cette liste est également utilisée par la fonction de mise à jour générale *update*. Le principe est le suivant : si un intervalle de temps fixé entre deux *update* (ici 10 ms) est dépassé, on relance la méthode *update* de chaque forme.

Travail à effectuer :

- Créer 2 objets de type *Sphere* centrés en *(2, 0, 0)* et *(0, 2, 0)* (cf. Figure 2) respectivement. Choisir et fixer les vecteurs directeurs adaptés pour la rotation des sphères autour du cube. Ajouter les adresses de ces deux sphères à la liste de pointeurs de formes *forms_list*.
- Compléter la méthode *render* de la *Sphere*. On utilisera, pour le rendu des objets, des quadriques (*GLUquadric*) qui seront manipulés par la fonction *gluSphere*. Prévoir une translation (*glTranslated*) et une rotation (*glRotated*) avant le rendu de la sphère afin de la positionner correctement dans la scène. On utilisera les données membres de la classe *Sphere* (et *Animation*). Vérifier le bon affichage des sphères en exécutant le programme.
- Analyser et comprendre la portion de code (fonction *main*, « *// Update the scene* ») qui permet d'actualiser la position des objets de la scène en vue de créer les animations.
- Ajouter une ligne permettant d'incrémenter (ou décrémenter) l'angle de rotation dans la méthode *Update* de la classe *Animation* et vérifier que les sphères tournent effectivement autour du cube.

De l'importance **de l'ordre des transformations** (translations, rotations…) :

- Modifier la fonction affichage et les coordonnées des sphères afin que le cube et les deux satellites soient centrés en (0, 2, 4).

Pour aller plus loin : déplacements de la caméra

La fonction ci-dessous permet d'indiquer :

- la position de la caméra (observateur) dans le repère global de la scène (*eye*) ;
- la direction (point de coordonnées center) vers laquelle l'observateur regarde
- la direction de l'axe vertical (up).

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

On souhaite pouvoir « tourner autour du cube » (i.e. la caméra peut décrire un cercle de centre le cube dans le plan *Oxz*) en appuyant sur les touches *'o'* et *'p'*. On souhaite également que l'appui sur les touches *'z'* et *'s'* permette d'incrémenter ou décrémenter la position de la caméra suivant l'axe vertical.

Travail à effectuer :

- Repérer dans la fonction *main* la variable permettant de fixer la position de la caméra. Vérifier que cette variable est bien utilisée dans le code pour fixer la position de la caméra via *gluLookAt*.
- Modifier les variables existantes et/ou en créer de nouvelles pour pouvoir gérer l'angle de rotation de la caméra ainsi que sa hauteur.
- Modifier la fonction affichage (notamment *gluLookAt*) afin de prendre en compte ces nouvelles variables.
- Modifier la portion de code gérant les événements claviers afin que l'appui sur les touches *'o'*, *'p'*, *'z'* et *'s'* déplace la caméra comme indiqué.

Indications : commencer à gérer le mouvement vertical, plus simple à réaliser. En ce qui concerne la rotation de la caméra autour du cube, l'utilisation de la bibliothèque standard c*math* pourrait s'avérer utile (cercle trigonométrique)…

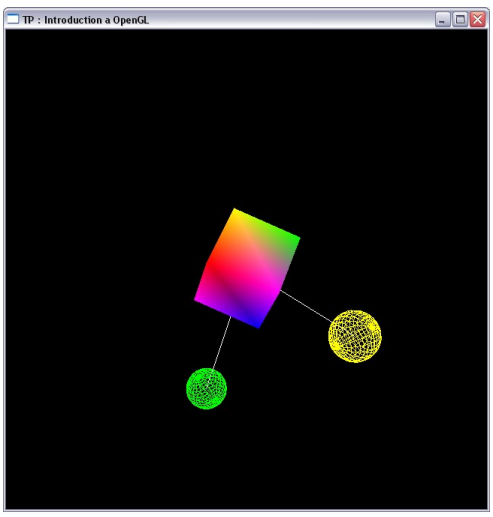

Figure 4 – Au final, avec les satellites en mouvement et la caméra mobile…

Références :

- Pages de manuel OpenGL : http://www.opengl.org/sdk/docs/man2/
- Cours d'infographie (notamment OpenGL) de Nicolas JANEY : http://raphaello.univ-fcomte.fr/IG/Default.htm
- Tutoriaux OpenGL NeHe : http://nehe.gamedev.net/