Sujets de TP - M4101c - Modélisation géométrique

R. Raffin
IUT Aix-Marseille
Département Informatique, Arles
romain.raffin[at]univ-amu.fr

v2016

Table des matières

1 TP1 - premiers pas

1.1 La guerre de génération

On l'a vu en cours, 2 versions majeures de sources OpenGL co-existent encore :
— le mode immédiat (*depecated*, à ne plus utiliser) à l'aide de :

```
glBegin(gl_TRIANGLES);
 glVertex3f(0.0, 0.0, 0.0);
 glVertex3f(1.0, 1.0, 1.0);
 glVertex3f(2.0, 2.0, 2.0);
glEnd();
```

— le passage par tableaux des géométries(*vertex arrays*)

Pour pouvoir gérer plus facilement les extensions et les versions d'OpenGL on s'appuiera sur la librairie open source GLEW.

1.1.1 Compilation

On utilise la librairie « OpenGL » pour l'affichage 2D/3D. C'est une API, les fonctions commencent par `glXXX` dont les spécifications sont faite par le groupe Khronos(`www.opengl.org`). On lui ajoute la librairie GLU pour des fonctions supplémentaires (fonctions commençant par `gluXXX`). Pour gérer une fenêtre sur votre système nous utiliserons GLUT, qui à les avantages d'être portable sur de nombreux OS et léger, mais est un peu ancien (gestion des surfaces tactiles ou du multipoints). On prendra une version OSS maintenue constamment : *freeglut*. Cela permet de définir un canevas dans lequel tracer (selon les systèmes d'exploitation c'est fastidieux (et non portable)).

Enfin pour pouvoir utiliser facilement les nouvelles versions d'OpenGL (extensions), on utilisera la librairie GLEW. La compilation se fait donc par :

```
g++ -Wall TP1points.cpp -lGL -lGLU -lglut -lGLEW -o TP1points
```

Il peut arriver que l'on utilise une version locale (que l'on compile soit même) de GLEW. Par exemple si on a mis les libs dans un répertoire `lib/` et le fichier d'include `glew.h` dans `include/`, la compilation devient :

```
g++ -Wall Tp1points.cpp -I./include/ -lGL -lGLU -lglut -L./libs/ -lGLEW
```

1.2 Premier exemple

1.3 Affichage de points

Nous allons afficher des points et d'autres primitives dans ce TP de prise en main, qui vous servira également à mettre en place la conception de vos objets géométriques.

1.3.1 Comment cela fonctionne ?

Le programme de base (« *TP1points.cpp* ») contient 4 fonctions :

1. `RenderScene()`
2. `InitializeGlutCallbacks()`
3. `CreateVertexBuffer()`
4. et `main()`

Et dans le détail :

1. `RenderScene()` contient les ordres d'affichage :
   (a) `glClear()` qui efface le canevas.
   (b) `glEnableVertexAttribArray(0);`
   `glBindBuffer(GL_ARRAY_BUFFER, leVBO);`
   `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);`
   demandent l'utilisation d'un buffer de données géométriques déclarées dans `CreateVertexBuffer()`.
   (c) `glDrawArrays(GL_POINTS, 0, 3);` demande l'affichage des données géométriques sous forme de points
   (d) `glDisableVertexAttribArray(0);` désactive les données courantes (s'il y en avait d'autre)
   (e) `glutSwapBuffers();` qui demande le rafraîchissement de l'écran
2. `CreateVertexBuffer()`. Comme on utilise un affichage asynchrone (et pas les vieux `glBegin()`/`glEnd()` marqués *deprecated*), on prépare les données une fois avant de demander l'affichage. On crée donc un tableau (dans l'exemple statique mais à changer le plus vite possible...) qui contient les coordonnées des sommets. Les positions sont X0, Y0, Z0, X1, Y1, Z1, ... On va envoyer ces données à OpenGL :
   (a) on génère un buffer
   `glGenBuffers(1, &leVBO);`
   (b) on donne un type OpenGL aux futures données
   `glBindBuffer(GL_ARRAY_BUFFER, leVBO);`
   (c) on met des données dedans
   `glBufferData(GL_ARRAY_BUFFER, sizeof(float)*9, vertices, GL_STATIC_DRAW);`
3. `InitializeGlutCallbacks()`. Sert à mettre en place la boucle de rendu via la fonction `RenderScene()` : `glutDisplayFunc(RenderScene);`. On y mettra aussi les interactions (*callbacks* de clavier, souris, *timers*).
4. `main()`
   (a) les fonctions de GLUT pour le système de fenêtrage :

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA); // utilisation de couleurs+transparence
//et 2 mémoire de rendu affichées alternativement
glutInitWindowSize(500, 500); // taille de la fenêtre d'affichage
glutInitWindowPosition(100, 100);
glutCreateWindow("TP1 : quelques points");
```

   (b) `InitializeGlutCallbacks();`
   (c) l'initialisation des extensions OpenGL (GLEW)

```
//toujours après l'initialisation de GLUT
GLenum res = glewInit();

if (res != GLEW_OK) {
cerr << "Error: " << glewGetErrorString(res) << endl;
return (EXIT_FAILURE);
```

```
}
//cout << "Using GLEW Version: " << glewGetString(GLEW_VERSION);

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

(d) la création des données géométriques : `CreateVertexBuffer();`

(e) la boucle de rendu : `glutMainLoop();`

1.4 À faire

Dans ces exemples les coordonnées possibles dans le canevas 3D sont $[-1, 1]$ en x et y. Nous verrons dans un prochain TP comment modifier cela (par la gestion des projections).

1. compiler l'exemple, l'exécuter
2. d'autres primitives sont possibles, modifier pour cela ligne 25 `GL_POINTS` par `GL_TRIANGLES` ou `GL_LINE_LOOP` par exemple. la liste des primitives se trouvent sur le *redBook* en ligne (`http://www.glprogramming.com/red/chapter02.html#name14`).
3. trouver la figure décrite par les 27 sommets (modifier le tableau, les valeurs des fonctions OpenGL, le nombre d'indices...)
4. afficher un tableau de points dont les coordonnées sont prises au hasard (entre -1 et 1 pour x et y),
5. afficher des points d'un cercle de centre $(0, 0)$ et de rayon 1, tester `GL_LINES` et `GL_LINE_LOOP`. On prend un pas constant d'angles dans l'équation paramétrique du cercle. Attention, il faut dès maintenant différencier la vue du modèle. Créez des classes pour vos objets géométriques, avec une fonction de sortie en tableaux de points (une classe générique « objetGeometrique » avec héritage serait encore mieux). Utilisez des fonctions pour les calculs des positions des sommets.
6. afficher des points d'un disque de mêmes paramètres, tester `GL_POLYGON` et `GL_TRIANGLE_FAN`,
7. faire du cercle précédent un objet suffisamment générique (via le constructeur ou les modificateurs) pour pouvoir placer des cercles de centres et de rayons aléatoires (positions des centres entre -1 et 1 en x et y, rayons entre 0 et 2 en réel) .
8. tracer une spirale (nouvel « objetGeometrique ») avec les mêmes paramètres au départ (centre $(0, 0)$, rayon 1). Le rayon décroît en même temps que l'angle croît.
9. tracer un premier carré (nouvel « objetGeometrique ») centré en $(0, 0)$, puis dans celui-ci des carrés obtenu par rotation autour de $(0, 0)$. La longueur de l'arête des carrés décroît en fonction de l'angle.
10. couleurs : pour donner une couleur à toute une géométrie on peut utiliser `glColor3f(R, G, B)` avec R, G et B compris dans l'intervalle $[0, 1]$, qui représentent les quantités de Rouge, Vert et Bleu. Cela change l'état de la machine OpenGL et oblige tous les éléments suivants à être de cette couleur (jusqu'au prochain `glColor()`).

1.5 Liens et documentations du TP

— le site d'OpenGL : `www.opengl.org`
— `http://www.lighthouse3d.com/`
— `http://www.opengl-tutorial.org/`
— `https://en.wikibooks.org/wiki/OpenGL_Programming`
— OpenGL mode immediat `https://en.wikibooks.org/wiki/OpenGL_Programming/GLStart/Tut3`
— `http://glew.sourceforge.net/`

1.5.1 Prototypes de fonctions

— void glVertexAttribIPointer(GLuint index, GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);
index Specifies the index of the generic vertex attribute to be modified.

size Specifies the number of components per generic vertex attribute. Must be 1, 2, 3, 4.
Additionally, the symbolic constant GL_BGRA is accepted by glVertexAttribPointer. The initial value is 4

type Specifies the data type of each component in the array. The symbolic constants
GL_BYTE,
GL_UNSIGNED_BYTE,
GL_SHORT,
GL_UNSIGNED_SHORT,
GL_INT, and
GL_UNSIGNED_INT are accepted by both functions. Additionally
GL_HALF_FLOAT,
GL_FLOAT,
GL_DOUBLE,
GL_INT_2_10_10_10_REV, and
GL_UNSIGNED_INT_2_10_10_10_REV are accepted by glVertexAttribPointer.
The initial value is GL_FLOAT.
normalized For glVertexAttribPointer, specifies whether fixed-point data values should be normalized (GL_TRUE)
or converted directly as fixed-point values (GL_FALSE) when they are accessed.

stride Specifies the byte offset between consecutive generic vertex attributes.
If stride is 0, the generic vertex attributes are understood to be tightly packed in the array.
The initial value is 0.

pointer Specifies a offset of the first component of the first generic vertex attribute
in the array in the data store of the buffer currently bound to the GL_ARRAY_BUFFER
target. The initial value is 0.

— void glDrawArrays(GLenum mode, GLint first, GLsizei count);
mode Specifies what kind of primitives to render.
Symbolic constants
GL_POINTS,
GL_LINE_STRIP,
GL_LINE_LOOP,
GL_LINES,
GL_LINE_STRIP_ADJACENCY,
GL_LINES_ADJACENCY,
GL_TRIANGLE_STRIP,
GL_TRIANGLE_FAN,
GL_TRIANGLES,
GL_TRIANGLE_STRIP_ADJACENCY and
GL_TRIANGLES_ADJACENCY
are accepted.

first Specifies the starting index in the enabled arrays.

count Specifies the number of indices to be rendered.

— void glBufferData(GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage);
target Specifies the target buffer object.
The symbolic constant must be GL_ARRAY_BUFFER,
GL_COPY_READ_BUFFER,
GL_COPY_WRITE_BUFFER,
GL_ELEMENT_ARRAY_BUFFER,
GL_PIXEL_PACK_BUFFER,
GL_PIXEL_UNPACK_BUFFER,
GL_TEXTURE_BUFFER,
GL_TRANSFORM_FEEDBACK_BUFFER, or
GL_UNIFORM_BUFFER.

size Specifies the size in bytes of the buffer object's new data store.

data Specifies a pointer to data that will be copied into the data store for initialization,
or NULL if no data is to be copied.

usage Specifies the expected usage pattern of the data store. The symbolic constant must be
GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY,
GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY,
GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, or GL_DYNAMIC_COPY.

2 TP2 - couleurs, caméras, transformations

2.1 Mise en couleur

Pour pouvoir mettre des couleurs à un objet on utilise `gColor3f(GLfloat red, GLfloat green, GLfloat`
Cela applique une couleur à la totalité d'un objet ou des objet d'une scène (OpenGL est une machine à état, ne l'oubliez pas). Par contre si on veut changer les couleurs des sommets, il faut attribuer à chaque vertex une couleur. Pour cela on va passer par les *Buffer Object* comme pour les sommets vus précédemment. Si on reprend le principe de la description des sommets du TP1 :

```
static void CreateColorBuffer()
{
 float colors[9]; // 3 couleurs à 3 valeurs R, G, B par point

//couleur du 1er sommet
 colors[0] = 1.0f;
 colors[1] = 0.0f;
 colors[2] = 0.0f;

//couleur du 2ème sommet
 colors[3] = 0.0f;
 colors[4] = 1.0f;
 colors[5] = 0.0f;

//couleur du 3ème sommet
 colors[6] = 0.0f;
 colors[7] = 0.0f;
 colors[8] = 1.0f;

//génération d'une référence de buffer object
 glGenBuffers(1, &leBufferCouleur);

//liaison du buffer avec un type tableau de données
 glBindBuffer(GL_ARRAY_BUFFER, leBufferCouleur);

//création et initialisation du container de données (3 couleurs -> 9 float)
 glBufferData(GL_ARRAY_BUFFER, sizeof(float)*9, colors, GL_STATIC_DRAW);

}
```

Les opérations :

1. `glGenBuffers(1, &leBufferCouleur)`,
2. `glBindBuffer(GL_ARRAY_BUFFER, leBufferCouleur);`,
3. `glBufferData(GL_ARRAY_BUFFER, sizeof(float)*9, colors, GL_STATIC_DRAW);`.

sont exactement les mêmes (aux paramètres près).

L'appel à ce buffer pour le rendu [1] se fera de la même façon :

```
//Liaison avec le buffer de vertex
glEnableClientState(GL_VERTEX_ARRAY);
```

1. On utilise dans ce TP les *Vertex Buffer Object* plutôt que les *Vertex Attrib Buffer*, qui sont difficiles à manipuler sans faire les shaders correspondant. Il y a donc quelques changements par rapport au TP1, avec les `glEnableClientState();`

```
glBindBuffer(GL_ARRAY_BUFFER, leVBO);
glVertexPointer(3, GL_FLOAT, 0, 0); //description des données pointées

//Liaison avec le buffer de couleur
glEnableClientState(GL_COLOR_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, leBufferCouleur);
glColorPointer(3, GL_FLOAT, 0, 0); //description des données pointées

glDrawArrays(GL_TRIANGLES, 0, 3); //3 éléments à utiliser pour le dessin

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

2.1.1 À faire dans la partie Couleur

1. vérifier avec « TP2.cpp » que le rendu d'une couleur par sommet fonctionne (utiliser « RenderSceneCouleurs » comme *DisplayFunc*),
2. construire un object tétraèdre régulier (`http://www.mathcurve.com/polyedres/tetraedre/tetraedre.shtml`), composé de 4 faces triangulaires équilatérales, une couleur par sommet, centré en (0.0, 0.0, 0.0), de taille d'arête 0.5 (le mettre en variable du constructeur).

2.2 Transformations

Pour effectuer une transformation de type rotation, translation ou mise à l'échelle d'un objet OpenGL (une géométrie ou une caméra, cf. ci-dessous), on utilise les fonctions :

- `glRotatef(angle, axeX, axeY, axeZ);`. C'est à placer avant l'appel à la primitive et les paramètres sont l'*angle* en degrés, les *axe ? ?* sont la définition des axe autour desquels effectuer la rotation. Par exemple : `glRotatef(13.2, 0.0, 1. 0, 0.0)` effectue une rotation d'angle 13.2 autour d'y. On peut combiner les axes. Le centre par défaut est $(0, 0, 0)$.
- `glTranslatef( Tx, Ty, Tz)`. Cela effectue la translation de vecteur $\overrightarrow{T}(T_x, T_y, T_z)$.
- `glScalef( Sx, Sy, Sz)`. Cela effectue la mise à l'échelle (homothétie) de coefficients (S_x, S_y, S_z). Le centre par défaut est $(0, 0, 0)$.

Attention l'ordre des transformations est important, souvenez-vous du TD (ordre d'inverse).

2.2.1 À faire dans la partie Transformations

1. vérifier avec « TP2.cpp » que la rotation de la face triangulaire par défaut autour de y fonctionne (utiliser « RenderSceneRotation » comme *DisplayFunc*, et la *glutIdleFunc*),
2. faire tourner le tétraèdre régulier construit au premier exercice autour des 3 axes.

Remarque : On utilise la fonction `glutIdleFunc`, lié à la fonction `static void IdleFunc()` du code, qui est lancé à chaque fois que le GPU ne sait pas quoi faire (a fini de tracer) pour modifier continuellement la valeur de l'angle ($angle = angle + 1.0f$ par exemple, en degrés). Comme on a mis un `glLoadIdentity` dès le début de l'usage de la matrice `GL_MODELVIEW`, on repart de la position initiale et on fait une rotation de plus en plus grande autour de l'axe de y. Le `glutPostRedisplay()` de cette fonction permet de redemander le rendu d'une nouvelle image.

2.3 Caméras

Il y a dans OpenGL 2 types de projections (vues en cours) :

1. la projection orthographique, suivant une direction, sans centre (plan de projection), qui conserve les distances et les angles. Elle est utilisée dans les vues de la CAO notamment.
2. la projection perspective, suivant une direction et par un centre de projection. Plus un objet est lointain plus il apparaîtra petit (et vice-versa). Elle ne conserve ni distance ni angle mais est identique à la perception humaine.

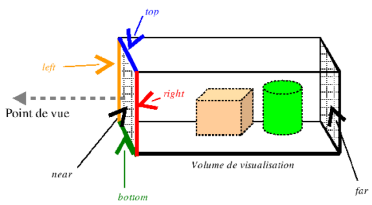
Vous avez des exemples de projection dans le cours et dans le redbook OpenGL (`http://glprogramming.com/red/chapter03.html#name3`). Pour définir une projection il faut utiliser la matrice... de projection. On écrit donc :

```
//Modification de la matrice de projection
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity(); //remise à 0 (identité)

 //projection orthogonale sur z=0, avec un volume de vue : xmin=-10, xmax=10, ymin=-10, ymax=10, ...
 glOrtho(-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);

 //Modification de la matrice de modélisation de la scène
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
...
```

Dans l'exemple ci-dessus on effectue une projection orthographique. Le prototype de la fonction est :

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)
```

Les paramètres correspondent à la taille du volume de vue, centré en 0.



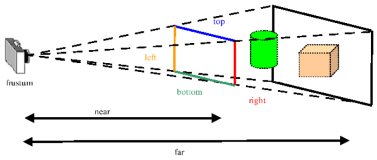
Pour effectuer une projection perspective il faut 2 définitions :

1. la caméra (ouverture, plans *near* et *far*, ration Largeur/Hauteur),
2. la position de la caméra et son orientation.

La première fonction est :

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);
```

Les paramètres donnent l'angle d'ouverture (env. 90 degrés pour un oeil, 180 au maximum pour les 2), le ration $\frac{L}{H}$ et les plans avant et arrière du volume de vue.



La deuxième fonction est :

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery,
Gldouble centerz, GLdouble upx, GLdouble upy, GLdouble upz)
```

Avec :

— $(GLdoubleeyex, GLdoubleeyey, GLdoubleeyez)$ les coordonnées de l'œil de l'observateur ou de la caméra,

— $(GLdoublecenterx, GLdoublecentery, Gldoublecenterz)$ les coordonnées du point visé par la caméra (ce n'est pas une direction mais bien un position),

— $(GLdoubleupx, GLdoubleupy, GLdoubleupz)$ le vecteur qui représente la normale (l'orientation) de la caméra autour de sa direction de visée.

Attention, le `gluPerspective()` se fait dans la matrice de projection, le `gluLookAt()` dans celle de modélisation. On aura donc :

```
//Modification de la matrice de projection
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity(); //remise à 0 (identité)
//définition d'une perspective (angle d'ouverture 90 degrés,rapport L/H=1.0, near=0.1, far=100)
 gluPerspective(90.0, 1.0, 0.1, 100);

//Modification de la matrice de modélisation de la scène
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();

//Définition de la position de l'observateur
//paramètres position(0.0, 0.0, 5.0), point visé(0.0, 0.0, 0.0, upVector - verticale (0.0, 1.0, 0.0)
 gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

... Tracé des objets ...
```

Attention, pour ces 2 projections, les éléments hors du volume de vue ne sont pas affichés. En cas de problème, vérifiez vos valeurs de plans *near* et *far* ou les min/max de `glOrtho`.

2.3.1 À faire pour la partie Caméras

1. vérifier avec « TP2.cpp » que les rendus par projection orthographique et perspective sont comprises (utiliser « RenderSceneCameraOrthographique » et « RenderSceneCameraPerspective » comme *DisplayFunc*),
2. utiliser la projection orthographique pour visualiser la vue de côté, de face, de dessus d'une `glutWireTeapot`. Pour cel aajouter un `glRotatef()` dans la partie `glMatrixMode(GL_PROJECTION)`.

2.4 Liens et documentations du TP

— *Vertex Buffer Object* `http://www.songho.ca/opengl/gl_vbo.html`
— `https://www.opengl.org/sdk/docs/man2/xhtml/glEnableClientState.xml`

2.4.1 Prototypes de fonctions

— `glEnableClientState - enable or disable client-side capability: void glEnableClientState(GLenum cap);`

```
Parameter
 cap Specifies the capability to enable.
  GL_COLOR_ARRAY,
  GL_EDGE_FLAG_ARRAY,
  GL_FOG_COORD_ARRAY,
  GL_INDEX_ARRAY,
  GL_NORMAL_ARRAY,
  GL_SECONDARY_COLOR_ARRAY,
  GL_TEXTURE_COORD_ARRAY, and
  GL_VERTEX_ARRAY

void glDisableClientState(GLenum cap);
Parameter
 scap Specifies the capability to disable.
```

3 TP3 - événements, hiérarchie

3.1 Événements

Les événements possibles sur le canvas OpenGL et la fenêtre sont les suivants : clavier touches classiques et spéciales (F1, LEFT...), souris, *timer*, manipulation de la fenêtre. Ces événements sont gérés par le système de fenêtrage (MsWindows, X11) et renvoyés à l'application. Comme nous utilisons la librairie GLUT, des fonctions sont à mettre en place pour traiter les *callbacks* de GLUT. En voilà quelques exemples (tirés du « TP3.cpp ») :

— le rendu ?
`glutDisplayFunc(RenderScene);`
— quand le GPU est en attente ?
`glutIdleFunc(callback_Idle);`
— les événements du clavier (classique) ?
`glutKeyboardFunc(callback_Keyboard);`
— les événements souris ?
`glutMotionFunc(callback_Mouse);`
— les événements sur la fenêtre ?
`glutReshapeFunc(callback_Window);`
— les touches spéciales du clavier ?
`glutSpecialFunc(&callback_SpecialKeys);`

Il y en a d'autres, dans la documentation de GLUT ou freeglut, ou dans la spécification OpenGL (par exemple : `glDebugMessageCallback()`, `glFinish()` ou `glWaitSync()`).

3.1.1 À faire événements

Utiliser les sources du « TP3.cpp ». Par défaut la touche « ESC » et le mouvement de la souris (`glutMotionFunc()`) sont traités.

1. utiliser les flèches `UP` et `DOWN` pour faire varier le point de vue. Lorsque l'utilisateur appuie sur `UP` on rapproche la caméra (cf `gluLookAt()`) sur le même axe (au départ la caméra est en $(5.0, 5.0, 5.0)$), en pointant vers $(0.0, 0.0, 0.0)$. A contrario reculer la caméra sur cet axe si l'utilisateur appuie sur `DOWN`. Remarquer les effets sur la perspective.
2. utiliser les flèches gauche et droite pour faire tourner la caméra autour de l'axe Oy

3.2 `Push` & `Pop`, hiérarchie de transformations

Comme nous l'avons vu en TD, OpenGL permet de faire des hiérarchies d'objets géométriques, en utilisant `glPush()` pour stocker la matrice de transformation et `glPop()` pour la dépiler. Les sources du TP3 présentent déjà un système complexe :

```
//rotation due aux mouvements de la souris
glRotatef(mouseAngleX, 1.0, 0.0, 0.0);
glRotatef(mouseAngleY, 0.0, 1.0, 0.0);

//dessin des axes du repère
drawAxis();
```

```
//Dessin de la théière centrée en (0, 0, 0)
//Tournant autour de Oy. Ajout de la sphère rouge tournant autour de la théière
glPushMatrix();
 //théière
 glColor3f(0.2, 0.5, 0.3);
 glRotatef(angle, 0.0, 1.0, 0.0);
 glutSolidTeapot(2.0);

  //sphère rouge
  glColor3f(1.0, 0.0, 0.0);
  glRotatef(1.2*angle, 0.0, 1.0, 0.0);
  glTranslatef(3.0, 0.0, 0.0);
  glutWireSphere(0.5, 10, 10);
glPopMatrix();
```

```
//Dessin du tore tournant autour de Ox.
//Ajout de la sphère bleue tournant autour du tore
//tore
glColor3f(0.4, 0.4, 0.4);
glRotatef(0.8*angle, 1.0, 0.0, 0.0);
glTranslatef(0.0,0.0, 7.0);
glutSolidTorus(1.0, 2.0, 10, 5);

 //sphère bleue
 glColor3f(0.0, 0.0, 1.0);
 glRotatef(1.5*angle, 0.0, 1.0, 0.0);
 glTranslatef(3.0, 0.0, 0.0);
 glutWireSphere(0.5, 10, 10);
```

Les 2 premiers `glRotatef` servent à modifier les axes du rendu selon les actions de la souris. Comme ils interviennent dès le début du rendu ils modifieront toute la scène (sauf la position de la caméra, le `gluLookAt()` intervient avant).

Le premier `glPushMatrix()` empile (sauvegarde) la matrice de `MODELVIEW` avant de dessiner une théière (tournant autour de Oy), puis une sphère rouge décalée (par translation $(3.0, 0.0, 0.0)$) qui tourne autour de Oy elle aussi. On a donc ici un mouvement relatif, de la sphère autour de la théière. Il est à noter que, comme le mouvement de la théière n'est pas isolé, si on accélère la vitesse de rotation de la sphère cela accélérera la rotation de la sphère (ajout des 2 angles pour la rotation autour de Oy).

Le `glPop()` qui suit ramène l'ancienne matrice de `MODELVIEW` comme matrice courante. Cela annule donc les transformations faites pour la théière et la sphère rouge.

Les 2 objets tore et sphère bleue ont donc leur mouvement propre. Comme précédemment, le mouvement de la sphère bleue est dépendant du mouvement du tore. Si on rajoutait un troisième groupe d'objets il faudrait mettre des `glPush()` et `glPop()` supplémentaires.

3.2.1 À faire `Push` & `Pop`

Autant faire un système solaire :

1. une sphère jaune de taille 4 est centrée en (0, 0, 0), elle représente le Soleil.
2. autour d'elle, à une vitesse $V_{rotTerre}$ et à une distance de 10, tourne une sphère bleue de taille 2 : la Terre.
3. autour de cette terre, et à une distance 4, tourne une sphère grise, de taille 1 et à la vitesse $V_{rotLune} = 2 \times V_{rotTerre}$: la Lune.
4. attention : le soleil tourne autour de lui-même à la vitesse V_S, la terre tourne autour d'elle même à la vitesse $V_T = 3 \times V_S$, la lune tourne autour d'elle même à la vitesse $1,5 \times V_S$.

5. on rajoute une 3e planète, rouge (mars), de taille 1,8, qui tourne autour du soleil à une distance 18 avec une vitesse $V_{rotMars} = V_{rotTerre}$, et qui tourne autour d'elle même à la même vitesse que le soleil.

Note : pour mettre en évidence la rotation des sphères autour d'elles-mêmes, mettre peu de segments dans le glutSolidSphere.

3.3 Liens et documentations du TP

— *callbacks* de GLUT `https://www.opengl.org/resources/libraries/glut/spec3/node45.html`
— documentation de freeglut `http://freeglut.sourceforge.net/docs/api.php`
— documentation d'openGL `https://www.opengl.org/sdk/docs/man4/`
— touches spéciales traitées par GLUT `https://www.opengl.org/resources/libraries/glut/spec3/node54.html`

3.3.1 Annexes

Par rapport aux TP précédents, la fonction `InitializeGL` a été ajouté au démarrage de l'application (`main()`). Cela sert à créer une lumière, la positionner et permettre sa gestion par OpenGL (`glEnable(GL_LIGHTING);`). On demande aussi la prise en compte des profondeurs pour le rendu (`GL_DEPTH_TEST`). C'est pour cela qu'au début de la boucle de rendu on effectue :

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

pour réinitialiser le tampon de profondeur (et des pixels de rendu).

```
static void InitializeGL() {
GLfloat lightPosition [] = { 0.0f, 10.0f, 0.0f, 0.0 };

//Crée une source de lumière directionnelle
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

//Définit un modèle d'ombrage
glShadeModel(GL_SMOOTH);

//Z Buffer pour la suppression des parties cachées
glEnable(GL_DEPTH_TEST);

//La variable d'état de couleur GL_AMBIENT_AND_DIFFUSE
//est définie par des appels à glColor
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

}
```

4

TP4 - courbes de Bézier

L'objectif de ce TP est de manipuler des courbes paramétriques, en prenant les courbes de Bézier vues en cours. Construisez des classes : courbes paramétriques, courbe de Bézier, intervalle. Cette dernière classe permet de définir l'intervalle du paramètre d'une courbe paramétrique, nous avons vu en CM/TD que l'intervalle paramétrique pouvait être assez compliqué ($-\infty$, avec des trous...), on n'utilisera qu'un intervalle de réels $[0, 1]$. Comme vu dans les TP précédents, n'utilisez aucune fonction OpenGL dans ces classes d'objets géométriques. Ils ne savent que sortir un tableau de float qui contiennent des points de la courbe (ou les points du polygone de contrôle).

Les classes points et vecteurs des sources peuvent vous aider...

4.1 Rappels

Les polynômes de Bernstein $\mathcal{B}_{i,n}(t)$, de degré n sont définis pour $i = 0, ..., n$ par la formule :

$$\mathcal{B}_{(i,n)}(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} \text{ , pour } t \in [0, 1] \tag{4.1}$$

Si $P_0, ..., P_{n-1}$ sont les n points de contrôle d'une courbe de Bézier $Q(t)$, avec $t \in [0, 1]$, la courbe de Bézier $Q(t)$ de degré $(n-1)$ s'exprime par :

$$Q(t) = \sum_{i=0}^{n-1} P_i \mathcal{B}_{i,n}(t) \text{ , pour } t \in [0, 1] \tag{4.2}$$

4.2 Exercice 1

On veut créer une courbe de Bézier cubique en utilisant les polynômes de Bernstein (éq. 4.1). Ses points de contrôles sont :

$$P_0(-2, -2, 0) P_1 = (-1, 1, 0) P_2 = (1, 1, 0) P_3 = (2, -2, 0)$$

1. afficher la polyligne qui relie les 4 points de contrôles ;
2. afficher une approximation de la courbe de Bézier, à l'aide segments ;
3. on veut modifier les positions des points de contrôles :
   — un carré bleu (initialement sur le premier point de contrôle) permet de montrer quel est le point sélectionné.
   — les touches + et − permettent de choisir les points de contrôles en déplaçant ce carré bleu,
   — les flèches du clavier ↑, →, ←, ↓ permettent déplacer le point de contrôle sélectionné vers le haut, la droite, la gauche ou vers le bas respectivement,

4.3 Exercice 2

On veut raccorder deux courbes de Bézier cubiques en utilisant les polynômes de Bernstein :

1. assurez la continuité de position entre les 2 courbes. Le dernier point de contrôle d'une courbe correspond en position au premier point de contrôle de l'autre courbe,
2. posez des pré-requis afin que les tangentes soient colinéaires au point de contact des 2 courbes (G^1-continuité). On doit pouvoir modifier les points de contrôle de la $1^{\text{ère}}$ courbe et voir la modification induite sur la 2^{e} courbe.
3. assurez la C^1-continuité.

4.4 Exercice 3

Les courbes de Bézier peuvent être obtenues avec l'algorithme de De Casteljau (voir cours). Implémentez cette solution pour un nombre de points de contrôles paramétrable. Ré-écrivez seulement une fonction de la classe « courbe de Bézier » pour cette nouvelle sortie de points sur la courbe.

4.5 Exercice 4 (option)

Utilisez l'exercice 2 pour implémenter une courbe B-Spline uniforme. Réutilisez la classe « courbe paramétrique » pour implémenter cette nouvelle courbe.

4.6 Liens et documentations du TP

Utilisez le cours et au besoin les liens suivants :
— `http://www.math.u-psud.fr/~pansu/web_maitrise/Geometrie_maitrise.html`
— `http://fr.wikipedia.org/wiki/B-spline`