Master IMMGV 2005

Mini projet Programmation OPENGL

Généralités

Le système fourni permet de charger des objets dans le format .obj avec des matériaux (ambient, specular, diffuse) mais pas de textures et affiche la scène vue depuis un point de vue fixe.

Nous donnons l'exécutable complet pour pouvoir comparer au fur et à mesure de l'avancement du miniprojet.

run.bat data/owl.obj

charge un objet avec matériaux mais pas de textures

run.bat data/building.obj

charge un objet simple avec texture.

Pour plus d'informations sur le format .obj voir le fichier de spécification fourni sur le site web du cours : **obj_spec.txt**
Note: Seuls les modèles polygonaux triangulés sont actuellement supportés dans cette application.

Tache 1 : Un viewer 3D

La classe **RvExaminerViewer** permet de gérer le point de vue avec la souris. Actuellement, les fonctions permettant la modification des paramètres de vue sont vides, et la vue est initialisée avec des paramètres fixes par défaut.

Les fonctionnalités doivent être les suivantes :

Bouton droit + déplacement : rotation
Bouton gauche + déplacement : translations 2D dans le plan de vue
SHIFT + Bouton droit + déplacement (haut/bas) : zoom

Actuellement tout est géré par (RvCamera.cpp) :

1) **RvViewer::processMouseEvents(button,button_state,modifiers,x,y,type);**

Appelée depuis le callback "bouton souris" de GLUT :

```
void myGlutMouse(int button, int button_state, int x, int y )
{
      viewer->processMouseEvents(button,button_state,glutGetModifiers(),x,y);
}
```

et une fonction de mise à jour :

2) **void RvViewer::update(int mousex,int mousey);**

appelée depuis le callback "mouvement souris" de GLUT

```
void myGlutMotion(int x, int y )
{
  viewer->update(x,y);
}
```

Il faudra completer les méthodes de RvExaminerViewer (dans RvCamera.h) :

- begin/end rotate
- begin/end translate
- begin/end zoom

ainsi que

- rotate
- translate
- zoom

permettant de gérer la camera. Les endroits à changer dans le code sont indiqués par le symbole :

```
//....
```

Il faudra modifier/remplir des fonctions dans RvCamera.h, RvCamera.cpp

Typiquement les fonctions begin.../end... devront être appelées lorsque le bouton de la souris est appuyé et relaché, c'est à dire dans la fonction **processMouseEvents**(...). Les fonctions rotate/translate/zoom seront appelées durant le mouvement de la souris, c'est à dire dans la fonction **update**(...).
Il faudra chercher et traiter les événements GLUT (voir doc GLUT http://www.opengl.org/documentation/specs/glut/glut-3.spec.pdf).

Les rotations sont gérées grâce à une trackball qui donne une interface "naturelle" pour les rotations 3D. Ces fonctionnalités sont implémentées dans la classe **RvTrackBall**. Le champ **track** de la class **RvExaminerViewer** est une trackball.

La trackball utilise des quaternions (classe **RvQuat**) pour représenter des rotations 3D de manière stable et continue.
Pour plus d'information sur les quaternions, vous pouvez consulter les documents powerpoint et pdf mis en ligne sur le site du cours. Egalement le lien suivant:
http://www.gamasutra.com/features/19980703/quaternions_01.htm

La classe **RvQuat** propose des fonctions pour créer et manipuler des quaternions puis récupérer la matrice de rotation correspondante pour l'envoyer directement à OpenGL.

a/ Pour créer un quaternion il suffit d'utiliser une des fonctions suivantes:

Rvquat q = RvQuat(1,0,0,0); ou le quadruplet correspond aux 4 composantes du quaternion

Alternativement on peut également utiliser:

static RvQuat RvQuat::fromBallPoints(const RvVec &from, const RvVec &to)

qui donne le quaternion permettant de passer de la direction <from> a la direction <to>.

static RvQuat RvQuat::fromAxisAngle(const RvVec &v,float a)

qui donne le quaternion correspondant a la rotation d'un angle <a> (en radians) autour de l'axe de direction <v>.

b/ Pour combiner deux rotations en multipliant les quaternions :

RvQuat RvQuat::compose(const RvQuat &q1,const RvQuat &q2)

donne la rotation correspondant à la rotation q2 **puis** la rotation q1.
Exemple: RvQuat dest = compose(q1,q2);

Attention: cette composition n'est évidemment pas reflexive:
compose(q1,q2) n'est pas égal à compose(q2,q1) !

Alternativement on peut utiliser

RvQuat::applyQuat(RvQuat &q)

Par exemple : q1.applyTo(q2);
applique la rotation définie par le quaternion q2 au quaternion q1.

c/ Pour récuperer la matrice *openGL* correspondant à un quaternion :

RvQuat::getRotationMatrix(float *m)

d/ Pour faire tourner un vecteur il suffit de lui appliquer le quaternion :

RvQuat::applyTo(RvVec &V)

e/ La rotation dans le "sens inverse" peut être obtenue à l'aide du conjugué du quaternion:

RvQuat::getConj()

Les valeurs d'orientation de la trackball seront utilisées pour mettre à jour la position et l'orientation de l'objet **RvCamera** contenu dans le viewer.

Dans begin/end rotate il faudra utiliser les fonctions de la trackball pour tracker les modifications d'orientation de la vue **: RvTrackball::beginDrag / RvTrackball::endDrag** dans **RvExaminerViewer::beginRotate/endRotate** et **RvTrackball::update** pour mettre à jour l'orientation de la trackball en fonction de la position de la souris dans **RvExaminerViewer::rotate**.
Le quaternion correspondant à l'état actuel de la trackball est obtenu par **RvTrackball::getQuaternion**.

Dans la classe **RvCamera**:

Il faudra compléter la fonction **loadGL** pour appliquer les modifications de translation et orientation en OpenGL.

Bonus: on peut également réaliser un "walkViewer" avec les fonctionnalités suivantes :

bouton gauche enfoncé : l'utilisateur se déplace en ligne droite vers l'avant
bouton droit enfoncé : : l'utilisateur se déplace en ligne droite vers l'arrière
lorsque la souris s'éloigne du centre de l'écran vers la droite où la gauche, l'utilisateur tourne dans cette direction.

Note: l'utilisateur se déplace dans le plan (comme si il marchait d'où le nom du viewer…)

Tache 2 : chargement et affichage de textures

Actuellement le système charge les matériaux et initialise les maillages.

Il faudra :

1) Ajouter des champs dans **RvDisplayMaterial** pour le nom de la texture, les données image de la texture et ses paramètres

2) Initialiser le nom de la texture dans **RvModel ::loadObjMtl**

3) Completer les méthodes de **RvDisplayMaterial:: setTexture**, **setTiffTexture**, qui charge les fichiers image (tif) de texture et qui initialise les structured de données correspondantes de RvDisplayMaterial

4) Completer les méthodes **RvDisplayMaterial:: initGL**, **load**, **loadGL**,
   La fonction **loadGL()** active le materiau dans openGL, et effectue l'initialisation nécessaires pour l'affichage des textures.

   Il faudra initialiser les paramètres de texturing.

   La fonction **initGL()** charge les textures des materiaux dans openGL

   La fonction **load()** est la fonction responsable du chargement de l'image depuis le disque.

   Typiquement le pipeline est le suivant:

   setTexture(nom_de_fichier) → load() → setTiffTexture()

   puis

   initGL() qui permet de créer l'identificateur de texture openGL avec la fonction **glGenTexture** et charger la texture dans la mémoire graphique avec **glTexImage2D.**

   Enfin au moment de l'affichage d'un objet, il faudra activer le matériau avec **loadGL().**

Bonus : implementer un cache de textures

Le cache de texture vérifie si une image à déjà été chargée par un matériau et partage son identificateur de texture openGL

Tache 3 : selection et animation d'objet

Dans la classe **RvModel**

1) Implementer la méthode **getRay(int mousex, int mousey)** dans **RvViewer** qui crée un RvRay (défini dans RvGeom.h) qui trouve la direction du rayon à partir de la position de la souris.

2) Implémenter (dans RvSpace.cpp) la méthode
RvPrimitiveList ::intersect(RvRay &r, RvVec *intersection_point, RvVec* normal, float* distance)

qui renvoie un pointeur sur l'objet qui est l'intersection de la liste des objets de **RvPrimitiveList** avec un rayon **RvRay** ainsi que les coordonnées 3D du point d'intersection, de la normale au point et la distance sur le rayon.

3) Dans la classe **RvViewer**:

Completer la **méthode RvModel *RvViewer ::getPrimitive(int mousex,int mousey,**
RvVec *intersection_point, RvVec* normal, float* distance)

qui renvoie un pointeur sur l'objet intersecté par un rayon qui passe par le pixel ou se trouve la souris ainsi que les coordonnées 3D du point d'intersection, de la normale au point et la distance sur le rayon. Si il n'y a pas d'intersection, la fonction renvoie un pointeur NULL.

Actuellement la fonction d'intersection de RvModel renvoie l'intersection avec la boite englobante. Pour verifier si ca marche, afficher la boite englobante de l'objet intersecte dans la boucle centrale de glut.

5) Implementer l'intersection des triangles avec le rayon. Il faut remplacer l'intersection (**RvModel ::intersect**) avec la boite englobante avec une intersection avec les triangles contenus.

La structure d'un RvModel est un RvMesh, qui est une façon compacte de coder un maillage de triangles. Pour chaque RvMesh il y un tableau de sommets (Vertex* vertex) et un primitivegroup (actuellement 1 par mesh) qui a un tableau d'indices (unsigned short * indices, et leur nombre numIndices). A partir de ces deux tableaux il faut parcourir l'ensemble de triangles (c'est-à-dire les triplets de sommets définis par les indices), et utiliser une fonction d'intersection de rayon avec un triangle.

6) Implémenter les fonctions:

setPosition(RvVec pos)
setOrientation(RvQuat orient)

qui permettent de placer et orienter l'objet dans l'espace (par defaut l'objet est en (0,0,0), orientation = identité)

7) Utiliser cette fonction pour appliquer une translation et une rotation à l'objet sélectionné (par exemple en utilisant une **RvTrackBall** pour la rotation).

Bonus

Grille Implementer la classe RvGridSpace qui est une grille. Il faut completer les méthodes buildSpace et intersect. Ensuite remplacer votre WORLD dans BasicViewer.cpp par un RvGridSpace *WORLD.

Si le hardware est disponible, modifier **RvMaterial::loadGL** pour utiliser un "pixel shader" implémentant de l'ombrage de Phong par pixel.