Laboratori VIG. Pràctica 1

Professors de VIG

Curs 2010-2011. Quadrimestre Tardor

Resum

L'objectiu d'aquesta pràctica és que completeu el disseny i la programació d'una aplicació que ens permetrà crear escenes -simples- a partir d'instàncies de models geomètrics de sòlids i visualitzar-la en filferros. Els models geomètrics els llegireu de fitxers en format *Alias|Wavefront OBJ*. Per a la programació de la interfície gràfica utilitzareu Qt4 i com llibreria gràfica OpenGL. Les funcionalitats d'aquesta pràctica s'aprofitaran en el projecte final que haureu de desenvolupar.

1. Introducció

La pràctica es desenvoluparà durant 2.5 sessions. L'avaluació dels conneixements adquirits s'efectuarà mitjançant una prova que es realitzarà en el laboratori en les dates anunciades (mireu avís del racó). La prova consistirà en modificar alguna de les funcionalitats requerides, o bé en afegir alguna nova funcionalitat.

Hem dividit l'enunciat per sessions per a orientar-vos en el desenvolupament de les funcionalitats. En les sessions de laboratori us introduirem -si s'escau- els conceptes d'*OpenGL* que us calguin per la pràctica; però, fonamentalment, en aquestes sessions hauríeu de desenvolupar-la.

Al directori */assig/vig/sessions/S2.1* trobareu fitxers amb *l'esquelet* d'una aplicació gràfica que, simplement, inicialitza un entorn gràfic, i conté les crides i la descripció d'*algunes* de les funcions que haureu de programar i d'altres que us hem programat a mode d'exemple. De fet, tal i com està, l'aplicació només mostra els eixos de coordenades.

Concretament, aquesta versió inicial de la pràctica consta de les classes bàsiques (incompletes) necessàries per a guardar els models geomètrics de l'escena així com de la classe `GLWidget` que és la que implementa el *widget* d'OpenGL en què es visualitzarà la nostra escena. També s'inclou el fitxer `principal.ui` que descriu la seva interfície en Qt i el fitxer `main.cpp` necessari per a executar l'aplicació. Aquests fitxers els proporcionem per a ajudar-vos a començar, i perquè no hagueu d'invertir massa temps en aspectes allunyats dels continguts de l'assignatura. Però no hi ha res en ells que hagueu de respectar literalment, si no voleu. Podeu canviar tot el que vulgueu si penseu que us facilita el desenvolupament o que us simplifica el disseny. De fet, haureu d'afegir nous atributs, mètodes i classes al llarg de les pràctiques i en el projecte. Recordeu que avaluarem l'eficiència gràfica de les aplicacions. Us aconsellem que **LLEGIU** les funcionalitats requerides (secció 2), les sessions de desenvolupament que us proposem (secció 3), i analitzeu l'esquelet que us proporcionem **ABANS** de començar la programació.

L'escena estarà formada per un conjunt d'objectes (instàncies) que poden compartir el seu model geomètric (objectes primitius). Analitzeu l'Annex 1, 2 i 3 i l'esquelet. Fixeu-vos que en el codi que us lliurem, només engegar l'aplicació, es crea l'objecte *polígon base* (model i objecte) sobre el que s'hauran d'ubicar la resta d'objectes de l'escena; es tracta d'un polígon de 10x10 centrat en l'origen de coordenades i ubicat en el pla x-z.

Noteu que no existeix cap càmera explícitament declarada, en aquest cas OpenGL utilitza la seva càmera de defecte (mireu en el manual d'OpenGL els paràmetres d'aquesta càmera per entendre el resultat de la visualització).

Al directori */assig/vig/models* trobareu alguns models geomètrics amb què provar les funcionalitats d'aquesta pràctica (per exemple, *dolphin.obj, legoman.obj, homer.obj*). Estudieu els annexos 1 i 2 en què es descriuen l'estructura de dades i les classes bàsiques.

2. Funcionalitats

A continuació us indiquem les funcionalitats que heu d'implementar per poder passar amb èxit la prova de la pràctica i que requerireu en el projecte que us proposarem.

- *Càmera en tercera persona*.

  Aquesta càmera perspectiva ha de permetre a l'usuari inspeccionar l'escena interactivament. Inicialment (al posar l'aplicació en funcionament), serà la càmera activa i ha de permetre visualitzar el *polígon base* sense deformacions i el més gran possible en el *viewport*; els seus angles inicials d'orientació seran -15º segons l'eix x, +30º segons eix y, i 0º segons l'eix z (angles d'orientació positius en sentit contrari a les agulles del rellotge). L'escena constarà només d'un objecte: el *polígon base* que crea automàticament l'esquelet que us proporcionem (mireu l'esquelet).

  L'usuari, haurà de poder modificar interactivament l'orientació d'aquesta càmera.

  Ha d'haver un *widget* que permeti tornar la càmera a la seva orientació inicial, en aquest cas, si hi ha més d'un objecte en l'escena, s'hauran de veure tots, sense deformació i ocupant el màxim d'espai del *viewport*.

- *Afegir objectes a l'escena*

  Via la interfície, l'usuari ha d'escollir l'objecte (model geomètric) que vol afegir, concretament, ha de poder seleccionar un fitxer *OBJ*, assignar-li un nom i ubicar-ho en l'escena.

  Nota: Si el model del nou objecte no existeix en l'escena, s'ha de crear, assignar-li nom i carregar del fitxer *OBJ* el model (mireu el mètode que us oferim). Si existeix, només s'haurà d'instanciar/referenciar el model. Per tant, en qualsevol cas, s'haurà de crear un nou objecte que instanciarà (referenciarà) al seu model.

  Per a permetre a l'usuari ubicar el nou objecte en l'escena, es visualitzarà el seu model amb el centre de la base de la seva capsa mínima contenidora a l'origen de coordenades, orientat tal i com es troba en el model geomètric i escalat uniformement de manera que el costat més llarg de la base de la seva capsa mínima contenidora mesuri 1. L'usuari, aleshores, podrà desplaçar l'objecte segons els eixos x i z, prement, per exemple, x/X i z/Z per desplaçar l'objecte en la direcció de les X,Z positives o x,z negatives. Per a indicar que l'objecte està posicionat en la posició desitjada, l'usuari haurà de premer el boto dret ratolí. Un cop s'ha posicionat l'objecte no es podrà bellugar -en aquesta primera pràctica-.

  L'ubicació dels objectes, s'ha de poder fer sigui quina sigui la càmera activa.

- *Visió en planta i alçat.*

  S'ha de poder seleccionar (via la interfície) una visió de l'escena que permeti veure-la, simultàniament, en planta (projectada sobre el pla x-z) i alçat (projectada sobre el pla x-y). Les càmeres a implementar hauran de ser axonomètriques. Aquestes càmeres no es podran modificar interactivament.

- *Visió amb filferros o amb parts ocultes.*

  Via la interfície (un *widget*) s'ha de poder decidir si es visualitza l'escena en filferros o amb sòlid i eliminació de parts amagades.

3. Sessions

Per a la realització de l'aplicació us proposem una distribució orientativa del seu desenvolupament en les diferents sessions de laboratori.

Sessió 1 i part de la 2.

L'objectiu d'aquesta sessió és visualitzar en filferros l'escena inicial que estarà formada només pel *polígon base*, utilitzant la càmera en tercera personal amb els seus valors inicials.

- Estudieu les capçaleres de les classes i l'esquelet que us proporcionem. No comenceu a programar fins que no us hagueu familiaritzat amb aquestes classes i hagueu fet un primer disseny de l'aplicació. L'estructura de dades emmagatzemarà tant la informació del *polígon base* com la dels objectes que afegirem. Al conjunt de tota la informació l'anomenem *Scene*. Als annexos 1 i 2 d'aquest document s'hi resumeixen els aspectes més bàsics de l'estructura de dades. Noteu que els objectes seran instàncies d'objectes primitius (models geomètrics) dels que guardem la informació geomètrica en coordenades de model (tal i com la llegim del fitxer). Us proporcionem un mètode per a llegir els fitxers *OBJ.*

- L'objecte del *polígon base* i el seu model es creen automàticament al iniciar l'aplicació. Analitzeu com es fa en l'esquelet.

- Inicialitzeu els paràmetres de la càmera en tercera persona que permet veure l'escena inicial tal i com s'indica en la secció 2. Verifiqueu que funciona sempre, encara que feu un *resize* de la finestra.

- Cal que implementeu la visualitzacio de l'escena (comporta implementar el *render* de l'escena, dels objectes i dels models geomètrics -aquest darrer el teniu implementat-). En aquest punt, hauríeu de poder veure l'escena inicial.

Sessió 2 i part de la 3

L'objectiu d'aquesta sessió és implementar la funcionalitat de posar objectes en l'escena i fer una visualització en filferros interactiva de l'escena amb la *càmera en 3ra persona*.

- Completeu el *widget* que permet carregar els objectes.

  L'usuari podrà escollir el fitxer *OBJ* que conté el model i podrà ubicar-ho sobre el *polígon base tal* i com s'indica en les funcionalitats. Si es tracta d'un nou model (no existeix en l'escena) és crearà un `nou Model`, s'inicialitzaran els atributs i es carregarà el model (mireu els mètodes que us proporcionem). Sempre, per a cada objecte afegit es crearà un `Objecte` (una instància al model) i s'inicialitzaran els seus atributs (alguns depenen del posicionament final de l'objecte). Fixeu-vos que, en l'esquelet que us proporcionem, els atributs que indiquen la ubicació dels `Objecte` són: la posició del centre de la base de la seva capsa mínima contenidora, escalat (per a garantir la mida requerida de la base de la seva capsa mínima contenidora) i orientació (gir respecte un eix paral·lel a l'eix-y que passa pel seu centre que no fareu servir en aquesta pràctica, és a dir, inicialitzeu a 0º). Si preferiu tenir altres paràmetres per ubicar els objectes, per exemple, una matriu de transformació geomètrica ho podeu fer.

  A més, cal calcular la capsa mínima contenidora de l'objecte.

  Per a la programació del codi necessari per a la creació d'objectes i models geomètrics us podeu inspirar en el següent trosset de codi d'exemple que haureu d'adaptar al vostre disseny com toqui.

- per a crear un `Model`,

```
Model o("xxx");
o.readObj("../models/xxx.obj", Scene::matlib);
scene.AddModel(o);
```

- per a crear un `Objecte`,

```
Objecte oref(idnom, idobj, posicio, grausGir, Escala);
scene.AddObjecte(oref);
```

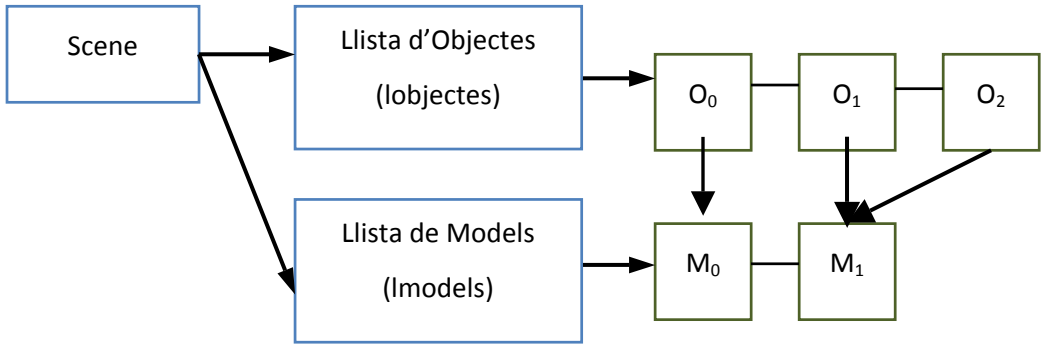
- Programeu la interacció de la càmera per poder modificar els angles d'orientació de la càmera X e Y, desplaçant el ratolí amunt/avall i a dreta/esquerra, respectivament.
- Implementeu un widget que permeti tornar als valors inicials d'orientació de la càmera PERÒ garantint que es veu tota l'escena sense deformacions.
- Si es fà un *resize*, no s'han de produir MAI deformacions en l'escena visualitzada.

Sessió 3.

L'objectiu d'aquesta sessió és incorporar la visualització simultània de l'escena en planta i alçat (dues càmeres axonomètriques) i la visualització en sòlid i sense parts amagades.

Annex 1. Descripció de l'estructura de dades

La classe **Scene** emmagatzema tota la informació requerida per a la visualització de l'escena S'ha triat una estructuració de la informació simple i orientada a les funcionalitats d'aquesta pràctica. El següent esquema us ajudarà a entendre l'estructura de dades:




```
class Scene
{
 private:
 //Tindrem un vector amb els models geomètrics dels objectes modelats
 //i un altre amb els objectes de l'escena que seran instàncies
 //de models).
 vector<Model> lmodels;
 vector<Objecte> lobjectes;
}
```

A continuació es descriuen les propietats bàsiques d'aquesta classe:

lmodels: és un vector que conté el model geomètric dels objectes primitius en coordenades de model, és a dir, ubicats on indiqui el fitxer OBJ corresponent.
lobjectes: és un vector que conté les instàncies als objectes primitius i representa el conjunt des objectes de l'escena. El primer objecte és el polígon base. Cada instància consta d'un conjunt d'atributs: identificador de de l'objecte, model geomètric que el representa (índex al vector de models), posició en el tauler, escalat, orientació.

La classe **Model** permet representar objectes 3D formats per una malla de polígons. La classe Object té com a atributs bàsics un vector de cares i un vector de vèrtexs (a més de la seva capsa mínima contenidora)

```
class Model
{
        …
        vector<Vertex> vertices;
        vector<Face> faces;
};
```

Noteu que us proporcionem els mètode per a llegir un OBJ i omplir l'estructura dels Object, tot calculant la seva capsa mínima contenidora i normals per cara. També us proporcionem el seu mètode *render* que us pot servir "d'inspiració" per implementar d'altres.

La classe **Vertex** que us proporcionem té com atributs les coordenades (x,y,z) del vèrtex i la normal en cada vèrtex (la fareu servir en la segona pràctica)

```
class Vertex
{
	…
	Point coord;
	Vecor normal;
};
```

La classe **Face** representa cadascuna de les cares (polígons) d'un model. Cada cara es descriu amb una seqüència ordenada -de 3 o 4- *índexs* a vèrtexs. Aquests índexs fan referència a la posició del vèrtex dins el vector de vèrtexs de l'objecte:

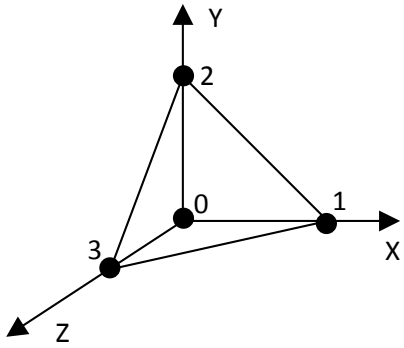
```
class Face
{
	...
	int material;
	vector<int> vertices;
	 vector normal;
};
```

Cada cara té també un índex al material que descriu les seves propietats òptiques i la seva normal.

Aquí teniu un exemple de creació d'un objecte senzill, concretament un tetraedre de costat 10, així com la numeració dels quatre vèrtexs del tetraedre que hem utilitzat:

```
Object tetra;
tetra.vertices.push_back(Vertex(Point( 0, 0, 0)));
tetra.vertices.push_back(Vertex(Point(10, 0, 0)));
tetra.vertices.push_back(Vertex(Point( 0,10, 0)));
tetra.vertices.push_back(Vertex(Point( 0, 0,10)));

tetra.faces.push_back(Face(1, 2, 3)); // front
tetra.faces.push_back(Face(0, 2, 1)); // right
tetra.faces.push_back(Face(0, 3, 2)); // left
tetra.faces.push_back(Face(0, 1, 3)); // bottom
```



Annex 2. Descripció dels fitxers de suport

Aquí teniu una descripció dels fitxers que us proporcionen classes bàsiques sobre les quals construir la vostra aplicació. Gairebé tots els atributs s'han definit com a públics per simplicitat. Tots aquests fitxers els podeu modificar/enriquir amb els mètodes/atributs que considereu oportuns.

Point.h, Point.cpp

Aquests fitxers proporcionen les classes *Point* i *Vector* que permeten representar respectivament punts i vectors a l'espai. Teniu definides les operacions bàsiques entre punts i vectors (sumes, restes...), que podeu consultar a la capçalera *point.h.* Les coordenades del punt i les components del vector són els atributs x, y, z.

Aquí teniu un exemple d'ús:

```
Vector v(1,1,1);
cout << v << " té longitud  " << v.length() << endl;
v.normalize();
cout << v << " té longitud  " << v.length() << endl;

Point min(0,0,0) , max(10,10,10);
Vector diag = (max - min);
Point centre = (max + min) / 2.f;
cout << "La coord x es : " << centre.x << endl;
```

Vertex.h, Vertex.cpp

Aquests fitxers proporcionen la classe *Vertex* que ja hem comentat. La versió que us proporcionem només té un únic atribut *coord* amb les coordenades del vèrtex.

Face.h, Face.cpp

Proporcionen la classe *Face* que també hem comentat. Una cara pot contenir un número arbitrari de vèrtexs. Hi ha un constructor per construir cares triangulars i quadrilàters. Podeu afegir més constructors, o afegir els (índexs a) vèrtexs amb les operacions de vector< >. OpenGL requereix que els polígons que es dibuixen amb *glBegin()-glEnd()* siguin convexes.

Model.h Model.cpp

Proporcionen la classe Model. El mètode readObj permet llegir un fitxer en format OBJ:

```
void Model::readObj(const char* filename, MaterialLib& matlib);
```

El segon paràmetre representa una biblioteca de materials on el mètode afegirà els materials de les llibreries de materials referenciades al fitxer OBJ.

Analitzeu tots els mètodes que us proporcionem, en especial, el *Render* i el càlcul de la capsa mínima contenidora (en coordenades del model)

Box.h, Box.cpp

Aquests dos fitxers proporcionen la classe *Box*, que permet representar capses orientades segons els plans cartesians. La capsa té únicament dos atributs amb els punts mínim i màxim de la capsa.

Color.h, Color.cpp

Defineixen la classe *Color*, que es representa amb les components *r*, *g*, *b*, on cada component té un valor entre 0.0 y 1.0. També hi ha una component d'opacitat *a*.

Material.h, Material.cpp

Proporcionen la classe *Material*. Aquesta classe s'explicarà amb més detall a la següent pràctica. De moment, només haureu de fer servir l'atribut *kd* on hauríeu de guardar el color amb què heu de pintar cada objecte d'aquesta pràctica:

```
class Material
{
public:
      ...
   string name;
   Color ka, kd, ks;
   float shininess;
};
```

MaterialLib.h, MaterialLib.cpp

Proporcionen una biblioteca de materials:

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

   void readMtl(const char* filename);
         const Material& material(int index) const;
   };
```

El constructor crea dos materials per defecte. El mètode *readMtl()* llegeix els materials d'un fitxer MTL, afegint-los a la biblioteca. Normalment aquest mètode no l'haureu de cridar directament, ja que es crida a la funció *Model::readOBJ()*.

El mètode *material* és el més important. Donat un índex d'un material (com el que teniu per cada cara d'un model 3D), retorna una referència al material. Per tant, si volem consultar la component de vermell d'una cara, el codi podria ser semblant a això:

```
const Face& face = obj.faces[i];
const Material& mat = matlib.material(face.material);
cout << "Component vermell color: " << mat.kd.r << endl;
```

Annex 3. Descripció d'alguns dels fitxers amb l'esquelet de l'aplicació

Aquí teniu una descripció d'alguns dels fitxers que us proporcionem com a esquelet de l'aplicació concreta que heu de desenvolupar.

GLWidget.h, GLWidget.cpp

La classe *GLWidget* és la que proporciona la finestra OpenGL. La major part del codi que desenvolupareu afectarà aquests dos fitxers. La versió que us donem l'haureu de completar amb tots els mètodes i atributs per assolir les funcionalitats de la pràctica: