Laboratori VIG. Pràctica 1
Quadrimestre de Primavera 09-10

Professors de VIG

L'objectiu d'aquesta pràctica és que completeu el disseny i la implementació d'una aplicació que permeti visualitzar interactivament una escena que representa una carretera per la que ha de circular un vehicle, i que està envoltada d'arbres, cases, fanals i altres objectes decoratius del terreny. La descripció de l'escena està emmagatzemada en un fitxer XML que es llegeix quan es posa en marxa l'aplicació. En aquest procés s'omple l'estructura de l'escena amb els objectes que cal visualitzar, la seva posició dins de l'escena, els seus factors d'escalat i la seva orientació. Per a la programació de la interfície gràfica utilitzareu Qt i com llibreria gràfica OpenGL. Aquesta aplicació constitueix la segona pràctica de laboratori que serà objecte d'avaluació.

1 Introducció

Per al desenvolupament d'aquesta pràctica disposareu de tres sessions de laboratori. L'aplicació resultant ens l'haureu de lliurar seguint les instruccions que indiquem en l'apartat 4 **abans del 14 d'abril de 2010 a les 14:00h.** La seva avaluació es farà en base a l'assoliment de les funcionalitats bàsiques requerides.

Durant aquestes sessions de laboratori us introduirem o recordarem els conceptes bàsics necessaris per al disseny de les funcionalitats de l'aplicació, però **fonamentalment en aquestes sessions hauríeu de desenvolupar la pràctica**. Això sols serà possible si us **prepareu raonablement aquestes sessions de laboratori**. A tal efecte, podeu repassar els exemples que heu anat veient en les sessions anteriors de laboratori i hauríeu de fer, prèviament a cada sessió, el disseny de les diferents funcionalitats i/o de la interfície gràfica. La classe de laboratori ha de servir per a programar i depurar el que ja heu pensat.

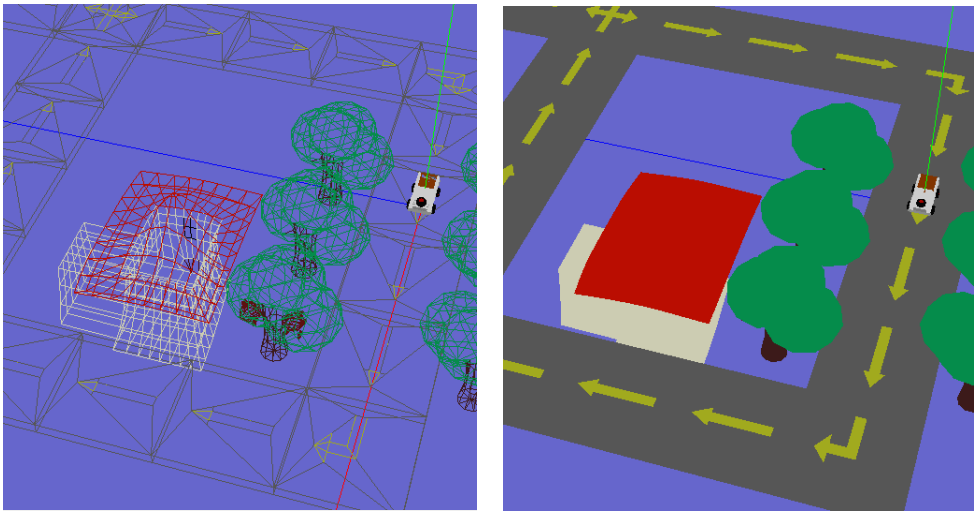
Al directori `/assig/vig/sessions/S2.1` trobareu els fitxers que heu de completar per a la realització de l'aplicació. De fet, tal i com estan, l'esquelet compila i executa perfectament, tot i que no es veu el que es vol per a l'aplicació perquè el codi està incomplet. Aquests fitxers són un esquelet de l'aplicació final que inicialitza un entorn gràfic, i que conté les crides i descripció d'algunes de les funcions que haureu de programar.

2 Funcionalitats de l'aplicació

Aquest document especifica les funcionalitats que la vostra pràctica ha d'incorporar. Per a poder obtenir un 10 de la pràctica, aquestes funcionalitats han de funcionar correctament i estar ben implementades. Es valorarà també la usabilitat de la interfície. A l'apartat 3 teniu

el guió detallat de cadascuna de les sessions de laboratori previstes per a aquesta pràctica i informació addicional per a la implementació de les funcionalitats requerides.

Globalment, la pràctica consisteix en el desenvolupament d'una aplicació que ha de permetre visualitzar en filferros una escena que està formada per una carretera, formada per trams que descriuen cadascun la direcció de moviment del vehicle en la carretera, i un conjunt d'*objectes* al voltant de la carretera (arbres, cases, fanals,...). A aquesta escena li podrem afegir un *vehicle*. En pràctiques subsegüents afegirem la possibilitat que aquest es mogui seguint la direcció marcada pels trams concrets de carretera en cada moment (veure figura 1).



(a) Imatge d'una possible escena en filferros (b) Imatge en omplert de polígons

Figura 1: Imatges de l'aplicació en funcionament amb una possible escena

L'escena es troba descrita en un fitxer `escena.xml` que es troba al subdirectori `data` de l'aplicació, on es troben també tots els fitxers OBJ que contenen els models de tots els objectes de l'escena, així com el model d'un cotxe que podeu usar com a vehicle. En el directori `/assig/vig/models`, trobareu també altres fitxers OBJ que contenen models que també podeu fer servir com a vehicle.

En les sessions del guió detallat i en els annexos trobareu més informació sobre: l'estructura dels fitxers, les estructures de dades i les classes que les implementen i l'esquelet de l'aplicació que us proporcionem i que heu de completar per aconseguir les funcionalitats requerides en aquesta pràctica.

Les funcionalitats de l'aplicació que implementareu en aquesta entrega són:

Visualitzar l'escena amb filferros. Caldrà completar l'esquelet de l'aplicació que us proporcionem a efectes de permetre visualitzar en filferros l'escena descrita en el fitxer `escena.xml` amb una càmera perspectiva. En el fitxer es descriuen els objectes que componen l'escena juntament amb la seva ubicació i orientació. Els paràmetres de la càmera per defecte han de permetre veure tota l'escena en pantalla ocupant el màxim de la vista i sense deformació.

Rotacions de la càmera, zoom i pan. Mitjançant el botó esquerre del ratolí, l'usuari podrà inspeccionar l'escena modificant, fent rotar, interactivament, la càmera al voltant de

l'escena. A aquest mode d'inspecció l'anomenarem *de tercera persona*. També es podrà realitzar un zoom (que es produirà quan s'arrossegui el ratolí amb el botó esquerre premut juntament amb la tecla Shift). Aquest zoom es farà acostant la càmera al centre de l'escena, i correspon per tant més a un *travelling* que a un zoom. A més s'ha de poder desplaçar la càmera en una direcció perpendicular a la de visió realitzant un PAN. El PAN es produirà quan s'arrossegui el ratolí amb el botó dret premut juntament amb la tecla Shift. L'efecte ha de ser que tota l'escena es mogui "acompanyant", aproximadament, al ratolí.

Carregar el model del vehicle. La interfície ha de donar la possibilitat de triar i obrir un fitxer OBJ que contindrà el model del vehicle. Un cop carregat el vehicle s'haurà de situar a l'escena convenientment escalat (de manera que la seva mida màxima sigui la meitat de l'amplada d'un tram) i en el centre del que és el primer tram de carretera. Suposarem que el *davant*, per defecte, de qualsevol vehicle està orientat en la direcció de l'eix X positiu del model. Fixeu-vos que aquesta hipòtesi pot produir que el que considerem *davant* no sigui el *davant natural* del model escollit (en el cas del model de cotxe del directori `data` sí que coincideixen).

Per a solucionar-ho, cal que implementeu una funcionalitat que *orienta* correctament el vehicle (cal que, interactivament, roteu el vehicle respecte un eix paral·lel a l'eix Y de l'aplicació que passa pel centre de la seva caixa englobant).

Restaurar la càmera. Cal afegir un botó a la interfície que permeti restaurar els valors inicials de la càmera de manera que es torni a veure l'escena completa, exactament igual que tot just després de carregar-la.

Redimensió de la finestra de l'aplicació. S'ha de permetre canviar la mida de la finestra de l'aplicació de manera que la interfície i la finestra OpenGL s'escalin convenientment.

3 Guió detallat

Per a la realització de l'aplicació us proposem una distribució orientativa en les diferents sessions de laboratori de les tasques a realitzar per a completar la pràctica. No cal que us ajusteu exactament a elles, però si veieu que hi ha una desviació important respecte d'aquest guió, cal que us esforceu en avançar més ràpid, o difícilment acabareu la pràctica per a la data d'entrega.

L'esquelet de l'aplicació el teniu al directori `/assig/vig/sessions/S2.1`. Aquest esquelet consta dels fitxers necessaris per a implementar tota l'estructura de dades de l'aplicació (escena, objectes, vehicle, etc.), així com de la classe `GLWidget` (en els fitxers `glwidget.h` i `glwidget.cpp`) que és la que implementa el widget d'OpenGL en què es visualitzarà la nostra escena. Aquest esquelet també inclou ja el fitxer `Principal.ui` que descriu la interfície en Qt de l'esquelet (generat amb el designer) i els fitxers `Principal.h, Principal.cpp, *.pro` i `main.cpp` necessaris per a construir l'aplicació.

Sessió 1.

L'objectiu d'aquesta sessió és carregar l'escena (codi ja implementat en l'esquelet de la pràctica) i visualitzar-la en filferros, fent servir els paràmetres inicials de la càmera que heu de calcular. L'escena visualitzada ha de mostrar tots els objectes correctament ubicats (la figura 2 mostra la visualització d'una possible escena).

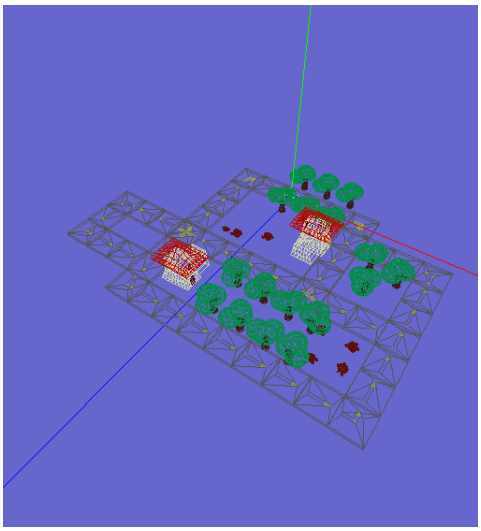


Figura 2: Imatge que es veuria en carregar una possible escena

- Estudieu **l'estructura de dades i l'esquelet de l'aplicació**. L'estructura de dades emmagatzemarà la informació dels trams de carretera, dels objectes al voltant i del vehicle en memòria. Al conjunt de tota la informació l'anomenarem *Scene*. Als annexos d'aquest document s'hi resumeixen els aspectes més bàsics de l'estrutura de dades. Analitzeu també les capçaleres de les classes que us proporcionem, i l'esquelet del codi que us proporcionem. **No comenceu a programar** fins que no us hagueu familiaritzat amb aquestes classes.

- Implementeu el codi necessari per a inicialitzar les matrius `MODELVIEW` i `PROJECTION` de manera que defineixin una càmera perspectiva que permeti veure tota l'escena ocupant el màxim de la vista i sense deformació. Fixeu-vos que teniu un mètode ja implementat de la classe `Scene` que us calcula l'esfera contenidora de l'escena.

- Implementeu els mètodes

  ```
  Scene::Render()
  Referencia::Render(std::vector<Object> &lobjects)
  Object::Render()
  ```

  que hauran de realitzar el pintat de tota l'escena. Fixeu-vos que per a visualitzar els objectes instanciats en la seva posició correcta, caldrà aplicar als objectes bàsics les transformacions de model indicades en cada instanciació (en la classe `Referencia`).

- Completeu el mètode `ResizeGL` per tal que en fer qualsevol redimensió de la finestra gràfica es continui veient com a mínim el mateix que es veia abans de començar el redimensionament i sense deformacions.

En aquest moment hauríeu de poder veure l'escena en filferros, encara que sense cap tipus d'interacció.

Sessió 2.

- Implementeu la modificació interactiva de l'*orientació de la càmera* composant adequadament la transformació de `MODELVIEW` amb un gir al voltant d'un eix perpendicular al moviment del ratolí en un pla paral·lel al del *window*, que passi pel centre de gir de l'escena. No cal actualitzar les distàncies del retallat anteroposterior al girar. La interacció per a aquestes rotacions ha de ser **mitjançant el botó esquerre del ratolí**.

  Recordeu que cada cop que volgueu repintar la vostra escena heu de fer una crida a `updateGL()`

- Implementeu el *zoom* apropant la càmera a l'escena. Heu d'actualitzar adequadament les distàncies als plans de retall anteroposterior, per a assegurar que no desapareixen parts de l'escena en fer-ho. Per a fer el zoom, l'usuari ha de prémer la tecla **"Shift" alhora que mou el ratolí sobre la finestra (en vertical) amb el botó esquerre premut**.

- Implementeu el *pan* de la càmera. S'ha de poder realitzar de forma interactiva quan l'usuari premi la tecla **"Shift" alhora que mou el ratolí sobre la finestra amb el botó dret premut**.

Sessió 3.

- Afegir un botó a la interfície per a carregar el model del vehicle. En el moment que es premi aquest botó, s'haurà d'obrir un `QFileDialog` que permetrà escollir el fitxer OBJ a obrir com a model del vehicle.

  El mètode `Vehicle::llegirModel(const char* filename)` ja fa les crides necessàries per a carregar el fitxer amb nom `filename`, actualitzar la seva capsa contenidora i inicialitzar la velocitat del vehicle a zero.

- Implementar el mètode `Vehicle::Render()` que faci el pintat del vehicle (recordeu que cal aplicar al model del vehicle les transformacions requerides per a que tingui la mida correcta i estigui ubicat a on toca dins l'escena). Inicialment el vehicle estarà situat sobre el tram inicial de la carretera i orientat en la direcció que aquest tram indica (el tram inicial no tindrà mai més d'una sortida).

- Afegir la possibilitat de reorientar del vehicle un cop es carrega per a poder modificar el seu *davant*.

- Afegir a la interfície un botó de "Reset" que permeti retornar la càmera als paràmetres inicials, com immediatament després de carregar l'escena.

4 Lliurament de la pràctica

Format de lliurament

Cal lliurar:

- **Tots** els fitxers font necessaris, de manera que un cop els copiem en un directori buit, quan fem "qmake" i "make" es generi l'executable de l'aplicació. Es recomana que proveu que aquest és el cas desempaquetant la vostra entrega en un directori buit i comprovant que es compila i funciona correctament al laboratori de la FIB. Els fitxers poden estar empaquetats en un .tar o .zip, que **no haurà** de contenir fitxers objecte ni executables ni Makefiles. Vigileu amb l'ús del tar. El que poseu després del flag 'f' s'obrirà per sortida. Si és un dels vostres fitxers, l'haureu perdut!
- **Un fitxer comentaris.txt**, si cal, amb aquells aclariments que creieu convenients fer sobre la vostra pràctica. Els comentaris que podeu introduir en la interficie de lliurament de la pràctica via el Racó, no els tenim accessibles en el moment de la correcció, per tant, no veurem els aclariments que no estiguin dins d'aquest fitxer `comentaris.txt`.

Mecanisme de lliurament

Via el Racó de la FIB

Termini

Fins al 14 d'abril a les 14:00 hores. A més a més, els professors us poden demanar una demostració de l'aplicació en hores de classe de laboratori.

Annex 1. Descripció de l'estructura de dades

La classe `Scene` emmagatzema tota la informació requerida per a la visualització de l'escena i del vehicle. S'ha triat una estructuració de la informació simple i orientada a les funcionalitats d'aquesta pràctica.

```
class Scene
{
 private:
   // Tindrem un vector amb els models geomètrics dels objectes de l'escena
   // i els altres dos amb instàncies seves (instàncies o referències a
   // objectes).
   // El vector circuit conté els trams de la carretera i el vector
   // lreferencies tota la resta d'instàncies d'objectes de l'escena
   // (arbres, cases, fanals, etc...)
   std::vector<Object> lobjects;
   std::vector<Referencia> lreferencies;
```

```
   std::vector<Tram> circuit;

   Vehicle veh;
};
```

A continuació es descriuen les propietats d'aquesta classe:

- *lobjects*: és un vector que conté el model geomètric dels objectes bàsics de què consta l'escena (els trams de carretera, les cases, els arbres, etc.). Noteu que la informació geomètrica està en coordenades de model, és a dir, ubicats on indiqui el fitxer OBJ.
- *lreferencies*: és un vector que conté instàncies dels objectes bàsics i representa el conjunt d'objectes de l'escena que no són trams de carretera. Cada instància d'un objecte bàsic (`Referencia`) consta d'un identificador (la posició dins del vector d'objectes de l'objecte bàsic que instancia), la seva posició a l'escena (que és la posició que ha de tenir el centre de la base de la seva capsa contenidora), la mida que volem que tingui en els tres eixos i l'orientació respecte l'eix Y.
- *circuit*: és un vector que conté instàncies dels trams de carretera. Una instància de tram de carretera (`Tram`) és una instància a un objecte bàsic (deriva de `Referencia`) i afegeix un vector de 4 identificadors indicant l'índex en `circuit` del següent tram de la carretera en les direccions X+, X-, Z+ i Z- respectivament. En aquelles direccions que no hi hagi següent tram, o en aquella que faria donar mitja volta al vehicle, l'identificador serà un -1.
- *veh*: pertany a la classe `Vehicle` i representa el vehicle que es mourà per la carretera.

Tant el vehicle com els objectes bàsics (i instanciats) tenen un mètode `Render` que és l'encarregat de pintar per pantalla la seva geometria.

Tant el vehicle com les instàncies a objectes tenen un atribut de tipus `Object` que 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:

```
class Object
{
     ...
   vector<Vertex> vertices;
   vector<Face>faces;
};
```

La classe `Vertex` que us proporcionem té un únic atribut amb les coordenades (x,y,z) del vèrtex:

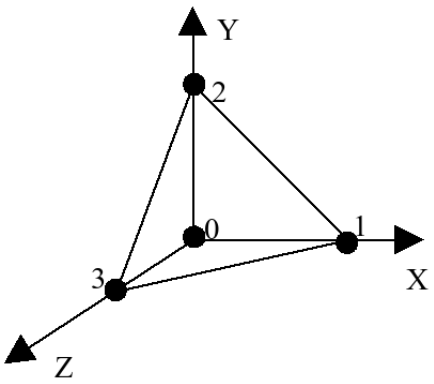
```
class Vertex
{
     ...
   Point coord;
};
```

La classe `Face` representa cadascuna de les cares (polígons) de l'objecte. Cada cara es descriu amb una seqüència ordenada d'índexs a vèrtexs. Aquests índexs fan referència a la posició del vèrtex dins el vector de vèrtexs de l'objecte. Fixeu-vos que no hem guardat el nombre de vèrtexs perquè la classe `vector` de STL ens permet saber aquest nombre mitjançant el mètode `size` (fariem `f.vertices.size()`):

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

Cada cara té també un índex al material que descriu les seves propietats òptiques.

Aquí teniu un exemple de creació d'un objecte senzill, concretament un tetraedre, 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

Els fitxers amb què tracta l'aplicació són els següents:

- Dins del directori *data* hi ha els fitxers *.obj* que contenen el model en format OBJ dels objectes que apareixen a l'escena. També hi ha els fitxers *.mtl* que contenen els materials de les cares dels objectes.
- *escena.xml*: També dins del directori `data` es troba la descripció de l'escena a visualitzar en un fitxer XML. Aquest fitxer conté informació de tots els objectes que formen l'escena i de la seva posició, mides i orientació en l'escena.

A més d'aquests fitxers, teniu uns altres que us proporcionen classes bàsiques sobre les que construir la vostra aplicació. Alguns dels 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);
```

vertex.h vertex.cpp
face.h face.cpp

Aquests fitxers proporcionen les classes `Vertex` i `Face` que ja hem comentat en l'annex anterior.

object.h object.cpp

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

```
void Object::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.

El mètode `Object::updateBoundingBox()` calcula la capsa englobant i actualitza l'atribut *boundingBox*.

```
void Object::updateBoundingBox();
```

El mètode `Object::Render()` serà l'encarregat de recórrer l'estructura de l'objecte per a pintar totes les seves cares. Fixeu-vos que al fitxer `object.cpp` que us donem **aquesta funció està buida i l'haureu d'implementar**:

```
void Object::Render();
```

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. El mètode `Box::update` permet expandir la capsa de forma que inclogui el punt donat:

```
void Box::update(const Point& p);
```

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 trobareu el color amb el que heu de pintar la cara en aquesta pràctica:

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

material_lib.h material_lib.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 `Object::readOBJ()` per cada fitxer MTL que s'hi referencia.

El mètode `MaterialLib::material` és el més important. Donat un índex d'un material (com el que teniu per cada cara d'un objecte 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;
```

tinystr.cpp, tinyxml.cpp, tinyxmlerror.cpp, tinyxmlparser.cpp, XML,h, XML.cpp

Aquests fitxers s'encarreguen de carregar i parsejar el fitxer XML que conté la descripció de l'escena. Un cop llegit l'XML, els vectors d'objectes i de referències de la classe `Scene` queden plens i ja preparats per a visualitzar l'escena.

Annex 3. Descripció de l'esquelet de l'aplicació

A continuació teniu una descripció de la resta de fitxers que us proporcionem que conformen l'esquelet de l'aplicació concreta que heu de desenvolupar.

glwidget.h glwidget.cpp

La classe `GLWidget` és la que proporciona la finestra OpenGL. Part del codi que desenvolupareu afectarà aquests dos fitxers. La versió que us donem l'haureu de completar de la forma indicada per assolir les funcionalitats de la pràctica:

```
class GLWidget : public QGLWidget
{
   Q_OBJECT

 public:
   GLWidget(QWidget * parent, const char * name);

 public slots:
   void help(void);// Ajuda per la terminal des de la que hem  engegat el programa.
      ...

 private:

   Scene scene;  // Escena a representar en el widget
};
```

Scene.h Scene.cpp

La classe `Scene` és la que conté i manega tota l'escena (la llista d'objectes bàsics, les instàncies a objectes (referencies i trams) i el vehicle).

```
class Scene
{
   friend class XML;
 private:

   // Tindrem un vector amb els models geomètrics dels objectes de l'escena
   // i els altres dos amb instàncies seves (instàncies o referències a
   // objectes).
   // El vector circuit conté els trams de la carretera i el vector
   // lreferencies tota la resta d'instàncies d'objectes de l'escena
   // (arbres, cases, fanals, etc...)
   std::vector<Object> lobjects;
   std::vector<Referencia> lreferencies;
   std::vector<Tram> circuit;
     ...
   Vehicle veh;

 public:
   Scene();

   void Init();
   void Render();
     ...
};
```

referencia.h referencia.cpp

La classe `Referencia` és la que s'encarrega d'instanciar els objectes a l'escena:

```
class Referencia
{
 private:
   int object;   // identificador de l'objecte (posició dins del vector d'objectes)
   Point pos;    // posició del centre de la base de la capsa contenidora
   Vector size;    // mida (sx, sy, sz) en unitats
   float orientation;  // orientació respecte Y

 public:
   Referencia(int idob, Point p, Vector sz, float ori);
   ~Referencia(void);

   void Render(std::vector<Object> &);

   int getObjectId();
     ...
};
```

tram.h tram.cpp

La classe `Tram` deriva de Referencia i representa un tram de carretera:

```
class Tram: public Referencia
{
 private:
   int seguents[4];   // vector d'indexos a trams connectats al tram donat

 public:
   Tram(int idob, Point p, Vector sz, float ori);
   ~Tram(void);
     ...
};
```

vehicle.h vehicle.cpp

La classe `Vehicle` s'encarrega de controlar i visualitzar el vehicle que es mou per l'escena.

```
class Vehicle
{
 private:
   Object obj;
   Point pos;          // posició del centre de la base del vehicle
   double escalat;     // escalat (homogeni) que cal aplicar-li en pintar-lo
   float orient;         // orientació (en graus) respecte l'eix Y

 public:
   Vehicle();

   // carrega l'objecte
   void llegirModel (const char* filename);

   void Render ();
     ...
};
```