VIG · Q2 (2009-10)
Guió de la pràctica 2

Índex

· Enunciat de la pràctica

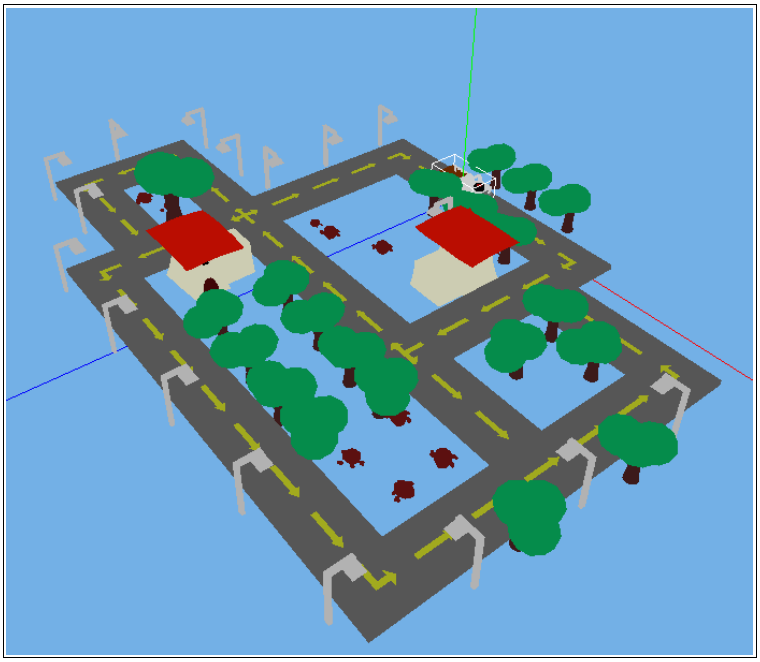
En aquesta pràctica demanen programar una escena de carrers amb un cotxe, faroles, arbres i cases, fent que el cotxe es mogui per els carrers, seguint les fletxes de cada tram de carrer. També s'ha de poder tenir control dels llums del cotxe, de les faroles, del suavitzat dels objectes, etc.

· OpenGL és una màquina d'estats

Abans de començar a programar, s'ha de tenir clar que OpenGL és una màquina d'estats. Això implica un canvi en la manera de programar habitual i un canvi d'idea del flux dels programes que estem acostumats: tenim un objecte que anomenarem “opengl”, i que té una sèrie de propietats tals com objectes, paràmetres, estructures, etc. I cada un d'aquestes propietats es troba en un estat determinat en un moment determinat.

Cada X temps, es llegeix l'estat d'“opengl”, i en funció de com està configurat en aquell moment, ens dona un resultat o un altre: es dibuixa una o altra cosa a la pantalla.

Per tant, nosaltres jugarem amb tots els paràmetres que necessitem tocar en cada moment. Si rebem una interrupció de teclat modificarem una variable, si fem un clic al ratolí modificarem un angle de visió, si activem llums modificarem l'estat de certs elements, etc. I cada X temps es rellegirà l'estat del sistema i es faran efectius els canvis.



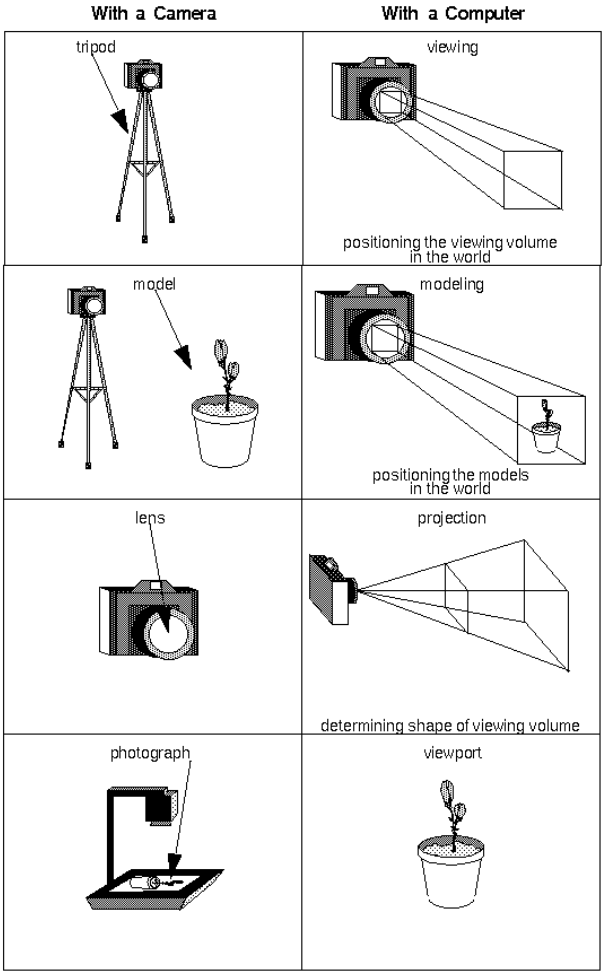
· Sessió 1

Analogia de la càmera

Per poder realitzar amb èxit aquesta sessió, necessitem conèixer i entendre l'analogia de la càmera:

The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera. As shown in figure, the steps with a camera (or a computer) might be the following:

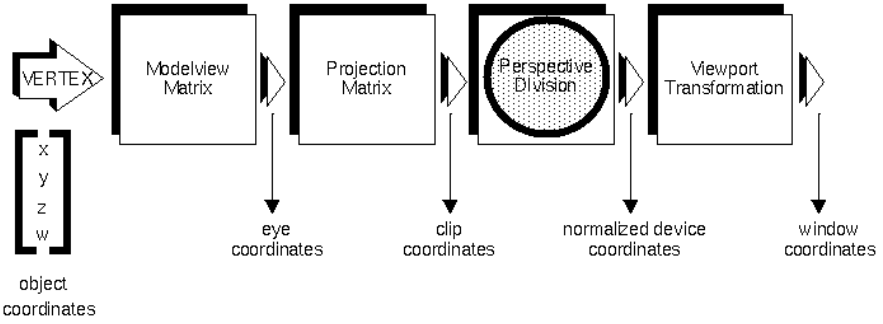
1. Setting up your tripod and pointing the camera at the scene (viewing transformation).
2. Arranging the scene to be photographed into the desired composition (modeling transformation).
3. Choosing a camera lens or adjusting the zoom (projection transformation).
4. Determining how large you want the final photograph to be - for example, you might want it enlarged (viewport transformation).



After these steps are performed, the picture can be snapped, or the scene can be drawn.

(*http://fly.srk.fer.hr/~unreal/theredbook/chapter03.html)*

Note that these steps correspond to the order in which you specify the desired transformations in your program, not necessarily the order in which the relevant mathematical operations are performed on an object's vertices. The viewing transformations must precede the modeling transformations in your code, but you can specify the projection and viewport transformations at any point before drawing occurs. Figure 3-2 shows the order in which these operations occur on your computer.

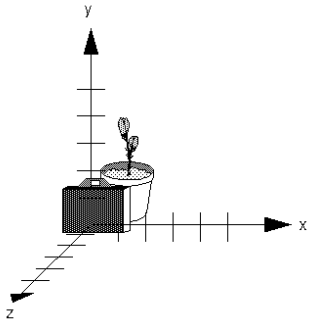


Programarem els següents mètodes, permetent la visualització del model 3D en filferros.

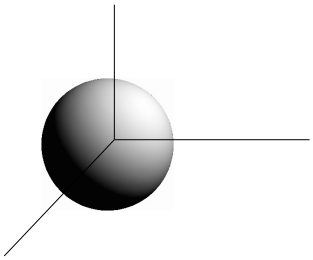
void GLWidget::updateModelView()

Aquesta funció l'hem d'escriure de 0 al fitxer glwidget.cpp. Està definida al .h però no al .cpp.

ModelView es fa servir per posicionar el model. Inicialment ens trobem amb la càmara i l'escena de la següent forma:



El que farem serà, un cop tenim calculada l'esfera contenidora de l'escena, moure l'escena per tal de que el centre (VRP) quedi a l'origen de coordenades de l'aplicació (SCA):



Una vegada feta la translació, rotem l'escena respecte l'eix de coordenades, el nombre de graus que haguem definit a angleX, angleY i angleZ, perquè quedi posicionada com nosaltres vulguem (la volem veure una mica de dalt).

Finalment movem enrere l'escena una distància igual al doble del radi, més o menys, perquè la poguem visualitzar tota. La càmera no s'ha mogut de posició, però l'escena si i s'ha inclinat com hem volgut i l'hem col·locat a distància i més enrere de la càmera.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0,0,-distancia);
glRotatef(angleZ,0,0,1);
glRotatef(angleX,1,0,0);
glRotatef(angleY,0,1,0);
glTranslatef(-VRP.x,-VRP.y,-VRP.z);
```

glMatrixMode especifica quina pila de matrius és l'objectiu per aplicar les operacions que es faran a continuació.

glRotate*(*angle*, *xAxis*, *yAxis*, *zAxis*): (xAxis, yAxis, zAxis) is a unit vector *defining the axis of rotation*. It represents a unit vector. The angle parameter is the rotation angle in degree.

En aquest cas hem optat per moure el model en lloc de moure la càmera. S'ha de saber que hi ha les dues opcions, l'altra opció és realitza mitjançant gluLookAt(..) i l'explicarem en altres apartats.

void GLWidget::updateProjection()

Un cop tenim el model posicionat una mica lluny de la càmera, és hora d'especificar els paràmetres "d'obertura de la lent", de "ratio d'aspecte", etc.

Això es fa modificant la matriu Projection. Els paràmetres importants s'explicaràn a continuació a la funció "computeDefaultCamera()".

Inicialitzem la matriu Projection per exemple així:

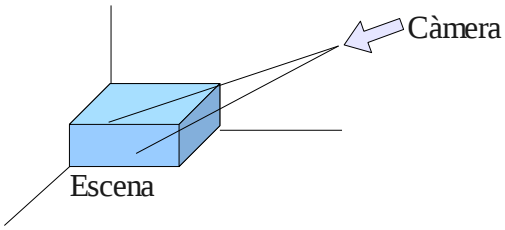
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fovy,ratio, near, far);
```

Els paràmetres els expliquem a continuació, s'inicialitzen a initGL() amb la funció computeDefaultCamera().

void GLWidget::computeDefaultCamera()

Aquest mètode **inicialitza els paràmetres de la càmera**. Val la pena explicar tot el que es fa al mètode, ja que són conceptes bàsics. Aquest paràmetres seran utilitzats després en les funcions de updateModelView i updateProjection.

Tenim l'escena, que té unes certes dimensions, i la volem visualitzar tota. Abans ja l'hem col·locat lluny de la càmera, però ens falta obrir l'objectiu, i definir "les lents":



Començarem calculant l'esfera que conté la nostra escena que ens ajudarà a realitzar els següents càlculs:

1. Calcularem a quina distància de la càmera s'ha de situar l'escena (o model) ja que inicialment la tenim aferrada a la càmera.
2. Calcularem un angle de rotació perquè l'escena es presenti davant nosaltres com vulguem.
3. Obrirem l'objectiu de la càmera tant com necessitem per visualitzar l'escena al complet, i l'obrirem tant d'altura com d'amplada.
4. Definirem com a que de lluny volem que la càmera hi veigi, i com a que d'aprop.

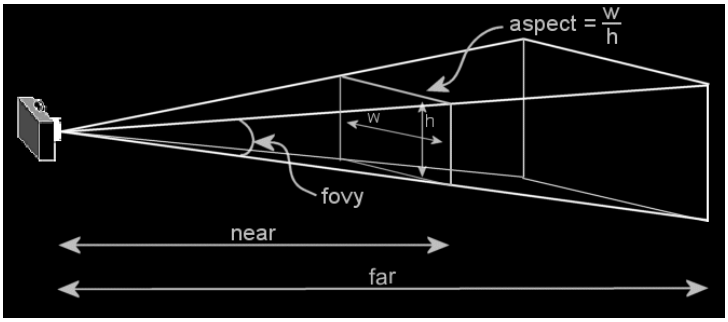
Necessitem els següents paràmetres, els definim a glwidget.h:

```
GLfloat fovy,dynamic_fovy;
Point VRP;
double distancia,radi,near,far;
float angleX,angleY,angleZ,ratio;
```

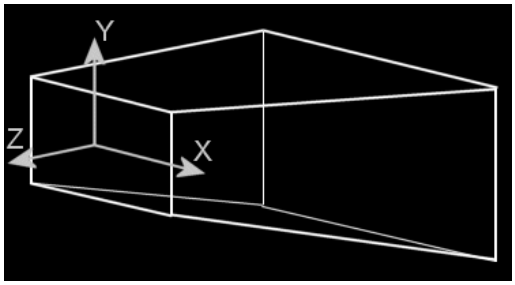
Explicació dels paràmetres i del tipus de càmera:

VRP: View Reference Point, és el centre de l'esfera i per tant centre de la nostra escena
distancia: serà la distància a la que mourem l'escena respecte l'origen de coordenades del SCA
radi: radi de l'esfera
angle_ : angles en que volem moure l'escena perquè quedi agradable a la vista de la càmera
fovy: Obertura de l'objectiu de la càmera, en l'eix Y.
dynamic_fovy: Utilitzat en apartats posteriors. Per no perdre el fovy inicial.
ratio: Obertura de l'objectiu de la càmera, en l'eix X. És típicament w/h.
*near:*Distancia mínima a la que pot veure-hi la càmera. Ha de ser sempre positiva.
*far:*Distancia màxima a la que pot veure-hi la càmera. Ha de ser sempre positiva.

Aquests paràmetres els utilitzem perquè volem una càmera de tipus “perspectiva”, que permet al contrari que les ortogonals, que els objectes tornin grans o petits segons la distància de la càmera.



A les càmeres de tipus perspectiva, la forma del volum de visualització és el d'una piràmide tallada que no te punta. La càmera inicialment enfoca a aquell plànol. A les cares de la piràmide se li diuen “clipping planes”, i té 6 cares, top, bottom, left, right, front, back. Aquestes cares es generen durant la funció “gluPerspective(fovy,aspect,near,far)”.
Un cop cridada aquesta funció el model de visualització, tot el que podrem veure, queda així:



El model de visualització ha de ser coherent amb el ViewPort: quan modifiquem la mida del glWidget, el ratio s'ha d'adaptar en funció de la nova mida de la finestra.

Per tenir una imatge sense deformació, la mida del view port (o glwidget) width/height, ha de coincidir amb el ratio d'aspecte definit.

Quan cridem a gluPerspective, el sistema de coordenades es modifica quedant centrat al pla frontal, amb l'eix X i Y paral·lels a aquest, i Z perpendicular i orientat cap a la càmera. Llavors la funció glLoadIdentity restaura el sistema de coordenades com estava abans.

Ens ha de quedar clar que tot el que estigui fora dels “clipping planes”, estarà amagat i no apareixerà a la pantalla.

El motiu de definir un model de visió limitat és que l'espai 3D és infinit, i amb el viewing volume el limitem. Per saber més d'aquest procés mirar el pipeline de OpenGL:
http://fly.srk.fer.hr/~unreal/theredbook/chapter03.html

Codi de la funció:

Un cop tenim els paràmetres, començarem a omplir la funció amb el següent codi:

· Aquesta funció modifica paràmetres per referència radi i VRP.

```
      scene.CalculaEsfera(radi,VRP);
```

Inicialitzem altres paràmetres:

```
      distancia = 2*radi;     //Distància a la que voldrem el centre de l'escena
                              //de l'origen de coordenades.
      near = radi;            //A partir d'un radi de distància, visualitzarem
      far = 3*radi;           //3 radis lluny, tindrem un back clipping plane
      angleX=45.0;            //Aquesta combinació d'angles per moure l'escena
      angleY=340.0;           //queda atractiva a la vista. (345=-45)
      angleZ=0.0;
```

El fovy és l'obertura de l'objectiu de la càmera en l'eix de les Y. Per calcular-lo adequadament, ens interessarà fer un càlcul trigonomètric.

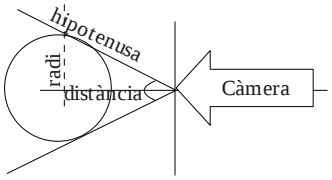
Tenint en compte que:

Radi és perpendicular a la tangent que passa per l'esfera mínima.

La tangent d'un angle de la càmera, és:

tan α = *catet oposat (radi) / distància*

Llavors, com que necessitem l'angle i no la tangent farem arc tangent, i multiplicarem per 2 per obtenir l'angle total.




```
    fovy = (float) 2*atanf(radi/distancia)*RAD2DEG;
```

RAD2DEG és una macro definida a glwidget.h. Aquesta utilitza el nombre PI, que és 3.1415.. Ho podem definir al principi del fitxer de la següent manera:

```
    #define PI 3.141592654
    #define RAD2DEG 180/PI
```

I finalment calculem el ratio de la pantalla:

```
    ratio = (float) width()/height();
```

En propers apartats haurem de conservar el fovy calculat aquí, i modificar-lo. Per això crearem un atribut anomenat dynamic_fov. Així que el crearem a glwidget.h i li donarem valor inicial en aquesta funció:

```
dynamic_fovy=fovy;
```

void GLWidget::paintGL()

Només ens queda que algú cridi aquestes funcions. Serà a la funció paintGL.

Ho farem a paintGL perquè és una funció que es crida cada vegada que s'ha de refrescar la finestra. si en algun moment es modifica per exemple la posició de l'escena, el canvi s'ha de fer efectiu.

Afegim abans del glBegin(GL_LINES):

```
updateModelView();
updateProjection();
```

· *Entenent les estructures de dades*

Inicialització de les estructures de dades:

Per continuar implementant els següents mètodes necessitem saber com estan implementades les estructures de dades.

Seguint el flux d'execució del codi tenim:

1. Es crida a glWidget::initializeGL()

   Aquesta funció crida a scene.Init();.

2. Scene té tres vectors:
   1. std::vector<Object> lobjects;

      Conté models d'objectes. Arbres, faroles, cases, etc.

   2. std::vector<Referencia> lreferencies;

      Conté instàncies a objectes, serà el vector que conté realment l'escena.

   3. std::vector<Tram> circuit;

      Conté trams de carretera, es diferència d'una referència perquè té una propietat “destí”.

Llavors, la funció scene.Init(), omple aquests vectors. Ho fa cridant primer a

xmlloader.LoadSceneDescription(this, "../data/escena.xml");

funció que es troba dins la clase XML.

3. XML::LoadSceneDescription(Scene,string)

Aquesta funció obre i itera el fitxer xml passat a l'string, i per cada definició de object que troba, crida a XML::ProcessObject(Scene,TiXmlNode). Aquesta funció crea un nou Object i crida a la funció Object::readObj(filename,..), i finalment quan ha creat l'Object a partir del filename extret del XML, crida a:

scene->AddObject(o);

afegint l'objecte o al vector d'objectes de Scene.

4. LoadSceneDescription(..) continua llegint el fitxer fins que es troba Trams, fa el mateix procediment que per els objectes i acaba afegint el tram al vector de trams.

5. Finalment LoadSceneDescription(..) acaba trobant al fitxer XML el nombre d'elements que hi ha i la seva posició i orientació. Crea la referència i els afegeix igual que abans:

scene->AddObject(ref);

Podem veure l'estructura del fitxer XML:

```
<scene>
	<objects> //Comença el processat d'objectes
		<object file="fletxadreta.obj" name="DRETA"></object>  //Objecte individual
		.........
	</objects>
	<description> //Comença el processat de referències I trams
		<objectref object="RECTA" id="0"> // Un tram, fixem-nos com té destination
			<position x="0" y="0" z="0"></position>
			<sizex>1</sizex>
			<sizey>0.1</sizey>
			<sizez>1</sizez>
			<orientation>180</orientation>
			<destination>1</destination>
		</objectref>
		..
		<objectref object="PEDRES"> //Una referència, fixemnos com no te destination
			<position x="-3.8" y="0" z="5.3"></position>
			<sizex>0.4</sizex>
			<sizey>0.2</sizey>
			<sizez>0.3</sizez>
			<orientation>120</orientation>
		</objectref>
	</description>
</scene>
```

Descripció de les estructures de dades:

Object:

vector<Vertex> vertices; //Tots els vèrtexs de l'objecte, un vèrtex té un Point (x,y,z).
vector<Face> faces; //Representen cares de l'objecte. Cada cara té un conjunt d'índexs del
//vector vertices. També té un identificador de material.

Referència:

int object; //Identificador d'objecte al que apunta la referència
Point pos; //Posició sobre el terra
Vector size; //Mida en unitats
float orientation; //Orientació respecte Y

Tram:

Atenció, Tram hereta de Referència, per tant té tot el que té Referència i a més el següent:

private:
int seguents[4]; //Indexos a trams connectats a aquest

Aquest vector seguents s'indexa de 0 a 3, i la informació que guarda són indexos del vector "circuit" que tenim a Scene.h.

#define XPOS 0 //index del seguent tram en les x positives
#define XNEG 1 //index del seguent tram en les x negatives
#define ZPOS 2 // index del seguent tram en les z positives
#define ZNEG 3 //index del seguent tram en les z negatives

Aquests indexos venen definits al fitxer XML, amb el paràmetre "id". Cada tram té un "id" i l'identifica dins Scene::circuits[id]. El paràmetre "destination" del fitxer XML és el valor que tindrà el vector seguents[]. Per exemple, si mirem el fitxer XML, el tram amb id="7", es col·locarà dins circuits[7] i tindrà degut a la seva orientació de 90º i de ser "DOBLE":

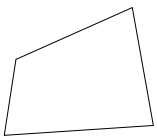
<destination>8</destination>
<destination>18</destination>

seguents[0] = 8
seguents[1] = 18
seguents[2] i seguents[3] sense definir.

Altres:

Face:

int material;
vector<int> vertices;



El vector de vertices de Face només és un vector d'enters que utilitza la classe Object. Aquests enters s'utilitzen per indexar: Object::vertices<Vertex>

Vertex:

Point coord; *

Point:

int x,y,z; (x,y,z)

Material:

string name; //Nom del material
Color ka,kd,ks; //Cada un té float r,g,b.
float shininess; //Brillantor

void Scene::Render()

Un cop entesa tota l'estructura de dades, inicialitzades les càmeres, projeccions i models de visió, anem a dibuixar l'escena. Com hem comentat en paràgrafs anteriors, la funció GLWidget::paintGL() es el callback cridat a cada interrupció de rellotge. Llavors aquí és on fem el renderitzat continu de la nostra escena amb la crida a:

scene.Render();

Només ens cal que scene:Render() recorri tot el vector de referències (el que realment defineix la nostra escena), i que recorri també el vector de trams cirtuit[], que recordem hi ha emmagatzemats separadament els trams, que hereten de referència.

```
int num = lreferencies.size();
for (int i=0; i<num;i++) lreferencies[i].Render(lobjects);
num = circuit.size();
for (int i=0; i<num; i++) circuit[i].Render(lobjects);
```

Per cada element dels vectors, cridarem a la funció Render, passant-li el vector d'objectes perquè la funció Render pugui instanciarla (és privada de scene.h, i aquesta manera de passar dades viola el principi de canviabilitat i crea acoblaments (veure assignatura ES2)).

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

Aquesta funció consisteix en el següent:

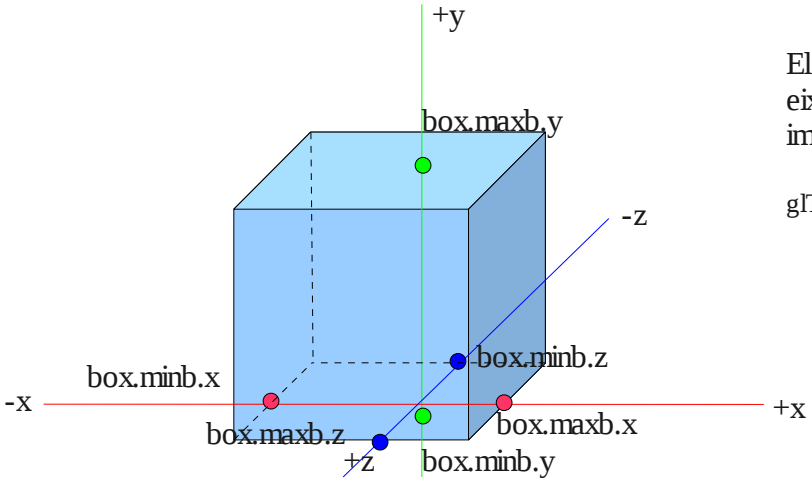
1. Agafar l'objecte i centrar-lo a l'origen
2. Escalar la mida de l'objecte
3. Orientar l'objecte
4. Traslladar l'objecte a la seva posició pertinent
5. Pintar l'objecte

1. Agafar l'objecte i centrar-lo a l'origen, però fent que la base estigui a altura y=0:

Ens ajudarà calcular la caixa contenidora de l'objecte:

```
Box box=o.boundingBox();
```

Llavors sabem que la caixa contenidora (i l'objecte) no es troba situat a l'origen[1]. Mitjançant el següent esquema podem fer el càlcul fàcilment per saber on col·locar-lo:



El càlcul del nombre d'unitats per cada eix per moure la figura a l'origen, és immediat:

```
glTranslatef(
        -(box.maxb.x+box.minb.x)/2.0
        ,-box.minb.y
        ,-(box.maxb.z+box.minb.z)/2.0
        );
```

2. Escalar la mida de l'origen

Obtindrem la mida de la capsa en el nostre sistema.

```
      float sx, sy, sz;
      sx = box.maxb.x – box.minb.x;
      sy = box.maxb.y – box.minb.y;
      sz = box.maxb.z – box.minb.z;
```

Llavors hem de tenir en compte que les mides que se li han passat a l'objecte, obtingudes del fitxer XML, són relatives a aquestes: NO tenen unitats.

Per tant dividim la mida que ha de tenir l'objecte per la que te ara:

```
      glScalef(size.x/sx,size.y/sy,size.z/sz);
```

3. Orientar l'objecte

Els arbres, faroles, casa, etc. Tenen un paràmetre “orientació” que defineix el nombre de graus de rotació respecte y. Està definit a l'XML. Aplicarem aquesta rotació tenint en compte que encara estem a l'origen.

```
      glRotatef(orientation,0.0,1.0,0.0);
```

4. Traslladar l'objecte a la seva posició

També tenim emmagatzemada la posició de l'objecte al fitxer XML. Només hem de traslladar-lo on toca:

```
      glTranslatef(pos.x,pos.y,pos.z)
```

5. Pintar l'objecte

```
      o.Render();
```

El codi complet:

```
  float sx, sy, sz;
  Object o=lobjects[this->object];
  Box box=o.boundingBox();
  sx = box.maxb.x - box.minb.x;
  sy = box.maxb.y - box.minb.y;
  sz = box.maxb.z - box.minb.z;
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glTranslatef(pos.x,pos.y,pos.z);
  glRotatef(orientation,0.0,1.0,0.0);
  glScalef(size.x/sx,size.y/sy,size.z/sz);
  glTranslatef(-(box.maxb.x+box.minb.x)/2.0,-box.minb.y,-
               (box.maxb.z+box.minb.z)/2.0);
  o.Render();
  glPopMatrix();
```

[1] - Quan diem que l'objecte no es troba situat a l'origen, és relatiu. El lloc on es defineix la posició inicial de l'objecte és el fitxer .obj que s'ha carregat quan hem inicialitzat els objectes. Si obrim el fitxer veurem com una línia defineix un punt amb 3 coordenades, x,y,z. Només hem de trobar quins d'aquests són els vèrtex situats “més als extrems”. Això és el que fa la funció que calcula la caixa i podem accedir-hi amb els atributs maxb._ i minb._.

void Object::Render()

Per aquesta part hem de tenir ben clara l'estructura de dades de Object, Faces, Vèrtexs i Points, Material i Color. Veure la pàgina 10.

El codi és senzill:

1. Per cada cara de l'objecte (conjunt de n vèrtexs)
2. Obtenim el color del material de la cara i l'activem a OpenGL
3. Dibuixem els vèrtexs de la cara

```
Material m;
//Recorrem totes les cares de l'objecte
for (uint i=0;i<faces.size();i++)
  {
    //Obtenim l'ID del material de la cara actual
    faces[i].material;

    /*Utilitzem l'atribut matlib de Scene, i cridem al
    mètode material(int), que retorna el material associat
    a l'index passat per paràmetre*/
    m = Scene::matlib.material(faces[i].material);

    //Amb el material obtenim els colors i l'activem a la màquina d'estats,
    //l'estructura s'explica en pràctiques posteriors i aquí només usem kd.
    glColor3f(m.kd.r,m.kd.g,m.kd.b);

    /*Definim que treballem amb poligon pq hi
    pot haver cares de 3 o 4 vertexs, per exemple les finestres
    de la casa*/
    glBegin(GL_POLYGON);

    //Per cada vertex de la cara actual
    for (uint j=0;j<faces[i].vertices.size();j++)
    //Pintar-lo on toca, accedint en ordre als
    //vèrtexs de Object::vertices<Vertex>.
      glVertex3f(
             vertices[faces[i].vertices[j]].coord.x,
             vertices[faces[i].vertices[j]].coord.y,
             vertices[faces[i].vertices[j]].coord.z);

    glEnd();
  }
```

void GLWidget::resizeGL()

Com hem vist en la funció de computeDefaultCamera(), hem calculat el ratio d'aspecte i el fovy que ens establien l'obertura horizontal i vertical respectivament de les lents de la càmera.

resizeGL() es crida cada vegada que la mida del ViewPort es modifica, i com hem dit abans hem d'adaptar la càmera al nou viewport. Només ens caldrà ajustar de nou el ratio d'aspecte i el fovy:

El ratio s'actualitza fent el càlcul de width / height.
Per el fovy haurem de crear un nou atribut, per exemple dynamic_fovy. Serà un atribut privat de glwidget.h de tipus Glfloat.

Llavors afegirem el següent codi a updateProjection(), eliminant l'anterior crida a gluPerspective:

```
  if (ratio < 1)
      gluPerspective(dynamic_fovy,ratio,near,far);
    else
      gluPerspective(fovy,ratio,near,far);
```

Això modificarà la perspectiva de la càmera si el ratio ha variat essent w és menor que h.

Finalment només hem de escriure el codi de resizeGL:

(Hem canviat els noms dels paràmetres perquè es deien igual que les funcions).

```
void GLWidget::resizeGL (int w, int h)
{
  glViewport (0, 0, w, h);
  //Recalculem objectiu de la càmera
  ratio = (float) w/h;
  if (ratio < 1) //Si w < h
      dynamic_fovy=atan(tan(fovy*DEG2RAD/2)/ratio)*RAD2DEG*2;
  updateGL();
}
```

· Sessió 2

*void GLWidget::mouseMoveEvent(QMouseEvent *e)*

Per implementar els mètodes ROTATE, ZOOM i PAN, en funció del moviment del ratolí, utilitzarem aquesta funció.

Què és un ROTATE?: Rotació de l'escena en X i Y, tal com feiem al updateInitModelView. Per tant només ens caldrà modificar l'angleX i l'angleY, atributs que haviem definit a glwidget.h.

Què és un ZOOM?: Modificació de l'obertura de la càmera en Y. És a dir, canviarem el fovy en funció del fovy actual (dynamic_fovy) i del moviment del ratolí.

Què és un PAN?: Com diuen els apunts de l'assignatura, un PAN només consisteix en traslladar el model o l'escena en funció del moviment del ratolí, i del sistema de coordenades de la càmera. Com que la nostra càmera està a l'origen com en vist en apartats anteriors, només haurem de moure el VRP, que és el centre de l'escena que haviem calculat.

```
  if (DoingInteractive == ROTATE)
 {
    // Fem la rotació
      angleX= angleX + (e->y()-yClick)/2;   //La divisió ens dona més suavitat
      angleY= angleY + (e->x()-xClick)/2;   //i precisió al moure el ratolí
  }
  else if (DoingInteractive == ZOOM)
  {
    // Fem el zoom
    if(dynamic_fovy+(e->y()-yClick)/2 >0
       && dynamic_fovy+(e->y()-yClick)/2< 180)  //Per no sortir-nos de l'escena
      {
      dynamic_fovy = dynamic_fovy+(e->y()-yClick)/2;
      fovy=dynamic_fovy;
      }
  }
else if (DoingInteractive==PAN)
  {
      // Fem el pan
      float m[4][4];
      glGetFloatv(GL_MODELVIEW_MATRIX,&m[0][0]); //Obté la matriu ModelView
      Point x_obs = Point(m[0][0],m[1][0],m[2][0]) * (xClick - e->x());
      Point y_obs = Point(m[0][1],m[1][1],m[2][1]) * (e->y() - yClick);
      VRP += (x_obs + y_obs) * 0.05; //Multipliquem per obtenir suavitat
  }
```

El paràmetre “e” és l'event QMouseEvent que rebem. Aques té diversos paràmetres, a nosaltres ens interessa la nova posició x i y, i per obtenir-la fem e->x() o e->y().

xClick o yClick són l'última posició del ratolí, ja està definida al final de la funció:

```
  xClick = e->x();
  yClick = e->y();
```

També necessitem actualitzar la visualització:

```
  updateGL();
```

*void GLWidget::mousePressEvent(QMouseEvent *e)*

Ens demanen que les accions anteriors es faixin mitjançant una sèrie de tecles en combinació amb el ratolí. Això es toca en aquesta funció, i veiem que per fer el PAN hem de pitjar CTRL quan ens demanen pitjar Shift. Ho canviem. També canviarem el LeftButton per RightButton.

```
  else if (e->button()&Qt::RightButton &&  e->modifiers() &Qt::ShiftModifier)
  {
    DoingInteractive = PAN;
  }
```

*void GLWidget::wheelEvent(QWheelEvent *e)*

Posats a fer implementarem aquest mètode que ens permetrà fer zoom amb la roda del ratolí, a l'estil google maps. És tan fàcil com afegir la següent línia a l'apartat protected de glwidget.h:

```
 //Implementem zoom amb la roda del ratolí
  virtual void wheelEvent(QWheelEvent *e);
```

I llavors implementar el mètode a glwidget.cpp. Com que és apartat opcional no l'explicarem amb profunditat. Per més informació veure el manual de Qt.

```
void GLWidget::wheelEvent(QWheelEvent *e)
{

  //Descomentar si volem que s'hagi de pitjar shift per fer zoom
  //amb la roda.
  //  if (e->modifiers()&Qt::ShiftModifier)
    {
      //apropa_allunya és un enter + si hem d'apropar, i - si hem
      //d'allunyar
      int apropa_allunya = -1*e->delta()/15/8;

      //Factor de zoom de 3.2
      float factor_zoom = apropa_allunya*3.2;

      if(dynamic_fovy+factor_zoom>0 && dynamic_fovy+factor_zoom<180)
      {
        dynamic_fovy = dynamic_fovy+factor_zoom;
        fovy=dynamic_fovy;
      }
    }

  e->accept();

  xClick = e->x();
  yClick = e->y();
  updateGL();
}
```

· Sessió 3

void GLWidget::resetCamera()

En aquest apartat introduirem un botó al a interfície per tal de que cridi a un slot de resetCamera per tornar a veure l'escena com quan havíem començat a executar el programa.

1. Declarar l'slot a glwidget.h

```
public slots:
...
void resetCamera();
```

2. Implementar el mètode a glwidget.cpp:

```
void GLWidget::resetCamera()
{
  computeDefaultCamera();
  updateGL();
}
```

3. Obrir el designer, col·locar un botó, donar-li de nom resetButton i posar-li el text “Reset càmera”. Llavors passar a mode edició de signals/slots i fer una fletxa del botó al glwidget. Finalment, enllaçar el signal “clicked()” amb l'slot “resetCamera()”. Si no apareix el resetCamera(), pitjar “Edit...” a l'apartat de gLWidget1, i crear un nou slot amb el nom exacte resetCamera().

· *Carregant el model del vehicle*

Crearem un botó a la interfície anomenat carregaVeh amb text “Carregar vehicle”. Aprofitarem per fer més interessant els botons usant uns icones que col·locarem dins data/icons/ en format gif i de 100x100, i que es definiran a la propietat Icon del botó. Els icones al botó seran de 24x24.

Un cop fet això enllaçarem el botó com hem fet al punt 3 de l'apartat anterior, amb un slot que anomenarem “carregaVehicle”.

Declararem l'slot a glwidget.h, i implementarem el mètode a glwidget.cpp:

```
void GLWidget::carregaVehicle()
{
  QString fitxer = QFileDialog::getOpenFileName(this, tr("Carregar
vehicle"),"../data",tr("Objectes (*.obj)"));
  const char *veh = (fitxer.toStdString()).c_str();
  scene.carregaVehicle(veh);
  updateGL();
}
```

Haurem d'implementar també scene.carregaVehicle(veh). Un cop declarat a scene.h el codi queda així:

```
void Scene::carregaVehicle(const char* filename)
{
  veh.llegirModel(filename);
}
```

Podem fer la prova de que ha anat bé posant un cout a Vehicle::llegirModel(). **Alerta amb les rutes de fitxers llargues i amb espais!!!**

També ens caldrà inicialitzar els atributs del vehicle i proporcionar un mètode per saber si el vehicle està carregat o no (per funcions posteriors):

```
Vehicle::Vehicle():obj("VEHICLE"), orient(0)
{
  veh_carregat = FALSE;      //Aquest és el nou paràmetre.
}

void Vehicle::llegirModel (const char* filename)
{
   obj.readObj(filename, Scene::matlib);
   obj.updateBoundingBox();
   Box box=obj.boundingBox();

   // Aqui cal que inicialitzeu correctament la resta d'atributs del vehicle
   Point pos(0,0,0);   //El vehicle, si no es diu el contrari, està a 0,0,0.
   escalat = 1.0;       //Mida del vehicle
   orient = 0.0;        //Orientació respecte Y
   veh_carregat = TRUE;  //Hem carregat el vehicle!
}
```

Declararem l'atribut privat veh_carregat com a booleà, i la funció Vehicle::enabled() que retornarà el booleà. Les declaracions a vehicle.h, i la implementació d'enabled a vehicle.cpp.

· Fer el Render del vehicle

Per implementar aquesta funció, aprofitarem el codi de Referencia::Render(). L'explicació és la mateixa que en l'altre apartat de Render que hem explicat. Només canvien els paràmetres de rotació, escalat i posició, que són els del vehicle:

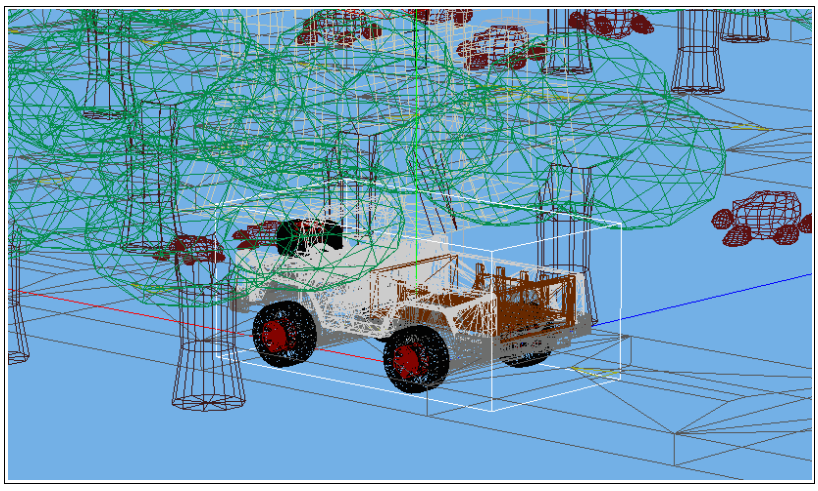
```
Box box=obj.boundingBox();

glMatrixMode(GL_MODELVIEW);
glPushMatrix();

//Traslladar l'objecte a on volem
glTranslatef(pos.x, pos.y, pos.z);


//Orientar l'objecte respecte y
glRotatef(orient,0.0,1.0,0.0);
```

```
//Escalar l'objecte
glScalef(escalat,escalat,escalat);

//Portar l'objecte a l'origen de coordenades
glTranslatef(
		-(box.maxb.x+box.minb.x)/2.0,
		-box.minb.y,
		-(box.maxb.z+box.minb.z)/2.0);
obj.Render();   //Pintar l'objecte
box.Render();  //Opció de debugging
glPopMatrix();
```

Ara ens falta que algú cridi aquesta funció amb els paràmetres adequats a l'enunciat. Modificarem Scene::carregaVehicle(..) de la següent manera:

```
//Llegim el model
  veh.llegirModel(filename);

  //Obtenim la posició i orientació del tram amb Id 0, com diu l'enunciat.
  Point pos = circuit[0].getPosition();
  float ori = cirtuir[0].getOrientation();
  //Movem el vehicle a la posició del tram 0 i el pugem a nivell del terra
  pos.y += 0.1;
  veh.setPos(pos);
  veh.setOrientation(180-ori);
```

El vehicle està inicialment a Y = 0, com el terra, per això incrementem pos.y en 0.1. Si fem un cout << circuit[0].getSize() << endl; comprovarem que ens dona: 0, 0.1, 0. Y val 0.1 que és l'altura del terra.



Finalment, només ens queda modificar el render de l'Scene per dir si ha de pintar o no la part de l'escena corresponent al vehicle. Com que hem proporcionat un mètode per saber si s'ha de pintar, simplement modifiquem Scene::Render() afegint al final:

```
  if (veh.enabled()) veh.Render();
```

· Re-orientant el vehicle

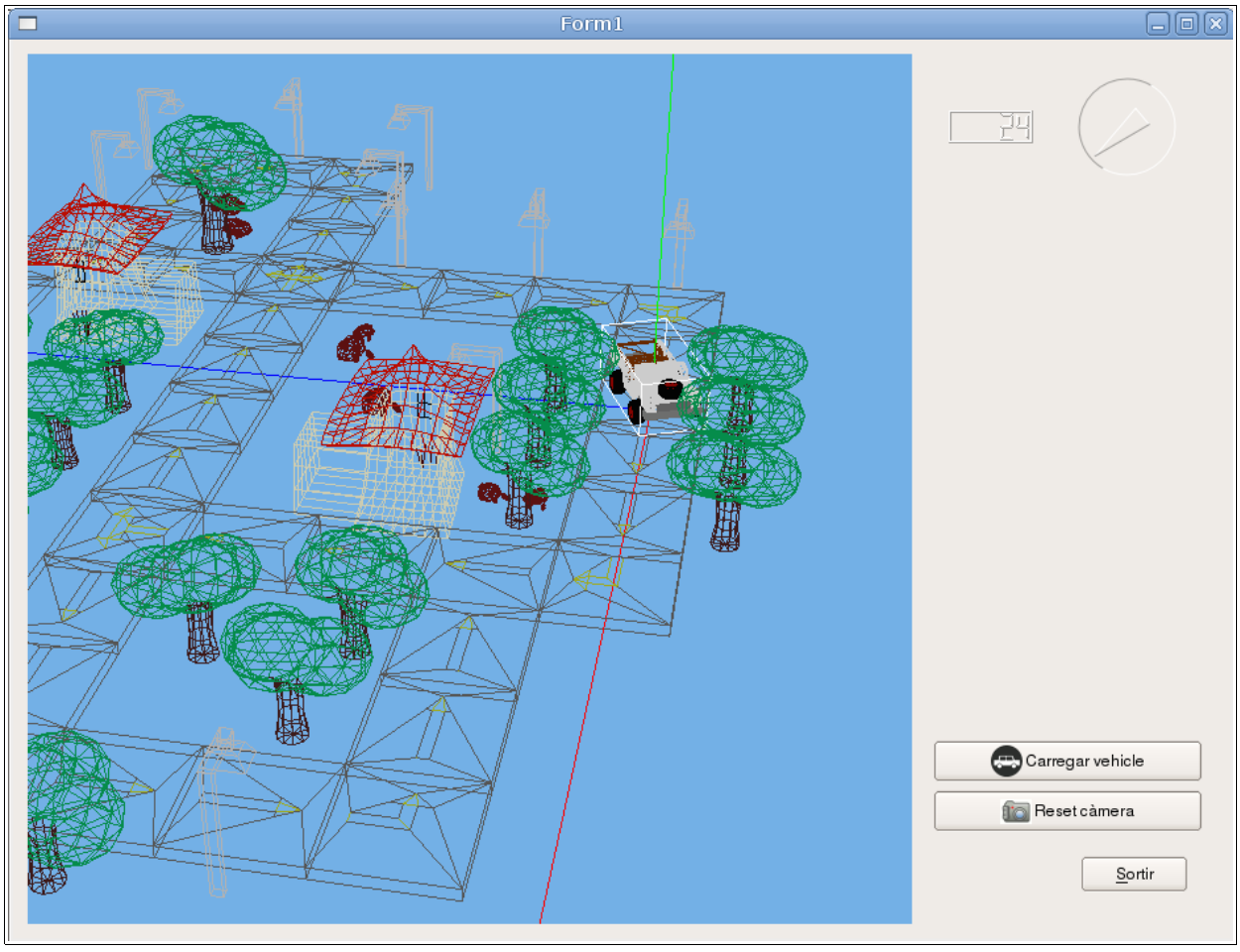
Per permetre la re-orientació del vehicle, afegirem a la interfície un “dial” circular que ens permetrà moure el vehicle de 0 a 360º, amb precisió d'1º. El connectarem a un display LCD que mostrarà els graus de rotació actuals. La connexió serà “sliderMoved(int)”->display(int).

També crearem una connexió de sliderMoved(int)->GLWidget::orientaVehicle(int). Aquest slot el declararem a glwidget.h i la implementació simplement cridarà a Scene::orientaVehicle(int). Llavors farà un updateGL().

Scene::orientaVehicle(int graus) farà un cast de l'int a float, i cridarà a: veh.setOrientation(graus).

Resultat final

Hem aconseguit fer tots els apartats de la pràctica i a més amb la funcionalitat de zoom per roda de ratolí, que millora molt la interfície amb l'esser humà.



· Bibliografia

http://jerome.jouvie.free.fr/OpenGl/Lessons/

http://www.opengl.org/sdk/docs/man/

http://fly.srk.fer.hr/~unreal/theredbook/

http://doc.trolltech.com/4.3/qwheelevent.html

Felip Moll Marquès

Abril 2010

Visualització i Interacció Gràfica