IDI: Coses importants per l'examen de lab

1. Transformacions de model

1.1 Translate

```
TG_model = glm::translate(TG_model, glm::vec3 suma_valor);
```

Agafa tots els vèrtexs i els hi suma el vec3 suma_valor. Serveix per poder ficar el model a on tu vulguis. Si per exemple volem que el centre de la base del Patricio estigui al punt (0, 5, 0) i a l'inici el Patricio està a (1, 2, -3), llavors hem de fer:

```
TG_model = glm::translate(TG_model, glm::vec3(-1, 3, 3));
```

1.2 Scale

```
TG_model = glm::scale(TG_model, glm::vec3 multiplica_valor);
```

Agafa tots els vèrtexs i els multiplica pel vec3 multiplica_valor. Serveix perquè el teu model tingui la mida apropiada a l'hora de ser pintat. Si per exemple sabem que el model del Patricio fa alçada x, volem que faci alçada 4 i tenim una variable escala donada per BL2GLWidget (classe que no es pot modificar):

```
TG_model = glm::scale(TG_model, glm::vec3(4*escala));
```

Normalment fem escalats uniformes per no deformar el model, per tant, la major part de vegades el valor que multipliquem a les x és el mateix que multiplicarem a les y i a les z.

1.3 Rotate

```
TG_model = glm::rotate(TG_model, float valor_a_rotar, glm::vec3 eix_rotació);
```

Agafa tots els vèrtexs i els rota valor_a_rotar **radians** sobre l'eix eix_rotació de coordenades. Serveix per moure un model sobre un eix de coordenades. Hem de tenir en compte que el sentit de l'angle de rotació va d'acord amb la regla de la mà dreta (fet a física). Per exemple, si el Patricio està mirant cap a tu a la càmera i vols que miri cap a l'esquerra, hem de fer:

```
TG_model = glm::rotate(TG_model, radians(-90), glm::vec3(0,1,0));
```

La funció radians(float) transforma el float que li entres pel valor equivalent a radians.

2. Càmera

S'ha d'actualitzar el seu valor cridant les funcions al paintGL. Normalment, per escollir entre una o l'altra es fa servir un bool i comprovar abans de cridar les funcions corresponents quin tipus de càmera volem. Les combinacions són: primera persona+ortogonal, primera persona+prespectiva, tercera persona+perspectiva, no hi ha tercera persona +ortogonal.

2.1 View matrix

2.1.1 Càmera en primera persona

```
View = glm::lookAt(glm::vec3 obs, glm::vec3 vrp, glm::vec3 up);
```

2.1.2 Càmera en tercera persona: angles d'Euler

```
View = glm::translate(View, glm::vec3 (0.0, 0.0, -d));
View = glm::rotate(View, factorAngleY+theta, glm::vec3(1, 0, 0));
View = glm::rotate(View, -factorAngleX+psi, glm::vec3(0, 1, 0));
View = glm::translate(View, -vrp);
```

El factorAngleY i el factorAngleX normalment els modificarem al mouseMoveEvent.

2.2 Project matrix

2.2.1 Càmera ortogonal

```
Proj = glm::ortho(float left, float right, float bottom, float top, float zNear, float
zFar);
```

2.2.2 Càmera perspectiva

```
Proj = glm::perspective (fov, ra, znear, zfar);
```

2.3 Calcular paràmetres de càmera

2.3.1 Radi i centre de l'escena (vrp) en iniEscena()

El primer que hem de fer és anar a iniEscena() i calcular el radi i el centre de l'escena a partir del Pmin i el Pmax (els dos hardcodejats) de la capsa contenidora. Sempre et donen els paràmetres ja calculats al LL4GLWidget (no es pot modificar), però, normalment, estan malament calculats.

```
Pmin = glm::vec3(-(15/2.0), 0.0, -5.0); //valors segons l'enunciat
Pmax = glm::vec3(15/2.0, 3.0, 5.0); //valors segons l'enunciat
radiEscena = glm::distance(Pmin,Pmax)/2;
vrp = gl::vec3((Pmin.x+Pmax.x)/2, (Pmin.y+Pmax.y)/2, (Pmin.z+Pmax.z)/2);
```

2.3.2 Càmera en primera persona en iniCamera()

```
obs = glm::vec3(0, 2, 5);
vrp = glm::vec3(0, 1, 0);
up = glm::vec3(0, 1, 0);
```

Els valors són donats per l'enunciat.

2.3.3 Càmera en tercera persona en iniCamera()

Els angles theta i psi són els angles de la posició original de la càmera, mentre que factorAngleX i factorAngleY estan inicialitzats a 0.0 i es van actualitzant cada cop que es mou la càmera:

```
void MyGLWidget::mouseMoveEvent(QMouseEvent *event)
{
  makeCurrent();
  if (!ortho) {
      factorAngleX = -((event->x() - xClick)) * ((float)M_PI/180);
      factorAngleY = ((event->y() - yClick)) * ((float)M_PI/180);
  }
  update();
}
```

Si es vol traballar amb radiants també es pot fer així:

```
void ExamGLWidget::mouseMoveEvent(QMouseEvent *e)
{
  makeCurrent();
  if ((DoingInteractive == ROTATE) && !camPlanta)
  {
    // Fem la rotació
    angleY += (e->x() - xClick) * M_PI / ample;
    angleX += (yClick - e->y()) * M_PI / alt;
    viewTransform ();
  }
  xClick = e->x();
  yClick = e->y();
  update ();
}
```

Fem un update al final de tot per cridar a paintGL i actualitzar l'escena amb els nous valors de la càmera. Normalment a l'examen et donen un eix fet, i tu has de fer l'altre.

2.3.4 Càmera ortogonal en iniCamera()

Els valors són marcats segons l'enunciat i/o per el radiEscena obtingut en iniEscena(), pero normalment son així:

```
left = -radiEscena;
right = radiEscena;
bottom = -radiEscena;
top = radiEscena;
zNear = radiEscena; // A vegades també pot ser 3*radiEscena
zFar = 3*radiEscena; // A vegades també pot ser 5*radiEscena
```

En general el zNear i el zFar dependran del enunciat, pero **sempre** hi ha d'haver una diferencia de, almneys, 2*radiEscena entre ells.

2.3.5 Càmera perspectiva en iniCamera()

Els paràmetres d'aquesta funció són marcats pel valor radiEscena obtingut en iniEscena():

```
d = 4*radiEscena; //distància de vrp a obs, també pot ser 2*radiEscena
alfaInicial = asin(radiEscena/d);
fov = 2*alfaInicial;
zNear = d-radiEscena; //zNear = 3*radiEscena, també pot ser radiEscena
zFar = d+radiEscena; //zFar = 5*radiEscena, també pot ser 3*radiEscena
```

Si el fov l'hem de calcular nosaltres amb la formula `asin(radiEscena/d)`, utilitzarem els valors "grans", si ens donen el fov ja calculat podem utilitzar els valors en el comentari

2.4 Resize (que no es retalli ni deformi)

Resize per a camara perspetive + ortogonal

```
void MyGLWidget::resizeGL (int w, int h) {
  rav = (float)w/(float)h;
  if (perspective)
  {
    ra = rav;
    if (rav < 1)
      fov = 2*atan(tan(alfaInicial)/rav);
    else
      fov = alfaInicial*2.0;
  }
  else
  {
    if (rav > 1) {
      l = rav;
      r = rav;
      b = 1.0;
      t = 1.0;
    }
    else {
      l = 1.0;
      r = 1.0;
      b = 1/rav;
      t = 1/rav;
    }
    projectTransform();
  }
```

Després el project de la perspectiva NO fa falta modificar-lo, pero al del ortho harem de modificarlo per que quedi algo semblant a lo seguent:

```
Proj = glm::ortho(l*-radiEscena, r*radiEscena, b*-radiEscena, t*radiEscena, zNear,
zFar);
```

On l, r, b i t seran variables "globals" que modificarem al resize.

2.5 Zoom

> NOTA: no provat en codi, ni compilat ni executat.

2.5.1 Càmera ortogonal

El que hem de fer és modificar el window (left, right, bottom i top), però sempre mantenint la relació d'aspecte del window (raw) perquè no hi hagi deformacions. Augmentar els valors fa que s'allunyi i decrementar-los fa que s'apropi a l'escena.

2.5.2 Càmera perspectiva

Per fer zoom modificarem el fov a la funció keyPressEvent. Si augmenta el fov ens estem allunyant, si disminueix el fov ens estem apropant a l'escena.

```
void MyGLWidget::keyPressEvent(QKeyEvent* event) {
  makeCurrent();
  switch (event->key()) {
      case Qt::Key_Up: {
          fov -= 0.1;
          break;
      }
      case Qt::Key_Down: {
           fov += 0.1;
           break;
       }
  default: LL4GLWidget::keyPressEvent(event); break;
  }
  update();
}
```

Si fem zoom i després el resize, el zoom es perd. Per tant, hem de modificar l'alfa inicial abans de fer el resize:

```
alfaInicial = fov_zoom/2;
```

Primer s'executa el zoom i el fov canvia de valor, després modifiquem alfaInicial i així el resize al cridar-se agafarà el valor correcte de alfaInicial per poder calcular el nou fov.

Hi ha una altra manera de fer zoom però és una fumada encara més gran.

3. Qt Designer

3.1 Introducció

El designer funciona amb widgets, cada objecte que hi ha a la interfície és un widget. Hi ha classes ja definides i fetes amb les seves funcions com el Button, RadioButton, Label, LineEdit,

etc. però també en podem fer de personalitzats, anomenats custom widgets. El típic nom és ficar MyButton, MyRadioButton, MyLabel, MyLineEdit, etc.

Hi ha un .cpp i un .h per cada custom widget, i a l'examen sempre ens en donen un de fet, el MyGLWidget. L'estructura dels custom widgets és aquesta, fent servir d'exemple la classe Label (a l'examen no t'enrecordes ni de conya jajajaj):

```
//MyLineEdit.h
#include <QLineEdit>

class MyLineEdit: public QLineEdit {
  Q_OBJECT

  public:
      MyLineEdit(QWidget *parent);

  public slots:
      //accions que pot fer MyLineEdit quan rep una senyal
      void captureReturn();

  signals:
      //senyals que MyLineEdit pot enviar a altres widgets
      void sendText(const QString&);
};
```

```
//MyLineEdit.cpp
#include "MyLineEdit.h"

MyLineEdit::MyLineEdit(QWidget *parent): QLineEdit(parent) {
}

void MyLineEdit::captureReturn() {
    emit sendText(text()); //emetre una senyal
}
```

3.2 Funcionalitats útils

És important fer un update() al final dels slots perquè s'actualitzi la interfície, sobretot al MyGLWidget per poder actualitzar l'escena al moment.

3.2.1 Canviar de color el widget

Font de color negre i color del widget vermell:

```
void MyLabel::changeToRed() {
    this->setStyleSheet("background-color: rgb(255, 0, 0); color: rgb(0, 0, 0)");
}
```

3.2.2 Enviar senyal

Aquest slot envia una senyal anomenada sendText que envia en format QString el text que el LineEdit té escrit.

```
void MyLineEdit::captureReturn() {
    emit sendText(text()); //emetre una senyal
}
```

4. Il·luminació

4.1 Vertex shader

Al vertex shader li arriben els vèrtexs de tots els objectes a pintar, i per a cadascun ha de calcular la normalMatrix, la normal i el vertex em SCO (Sistema de Coordenades d'Observador):

```
void main()
{
    //això es sempre igual i ja està fet
    fmatamb = matamb;
    fmatdiff = matdiff;
    fmatspec = matspec;
    fmatshin = matshin;
    //la normal i la normalMatrix s'han de calcular bé
    //normalment no te les donen fetes
    mat3 normalMatrix = inverse(transpose(mat3 (view * TG)));
    normalSCO = vec3(normalMatrix * normal);
    //el vertex en SCO tampoc el donen fet, normalment
    vertexSCO = view * TG * vec4(vertex, 1);
    gl_Position = proj * view * TG * vec4 (vertex, 1.0);
}
```

La normal matrix es pot calcular aquí o al MyGLWidget per fer-ho una mica més eficient. En canvi vertexSCO i normalSCO sempre s'han de calcular aquí.

4.2 Fragment shader

Les funcions per a calcular l'ambient, la difusa i l'especular sempre te les donen fetes, i assumeixen que tots els vectors que se li passem estan normalitzats.

```
vec3 Ambient() {
  return llumAmbient*fmatamb;
}
```

```
vec3 Difus (vec3 NormSCO, vec3 L, vec3 colFocus) //Lambert
{
  // Fixeu-vos que SOLS es retorna el terme de Lambert!
  // S'assumeix que els vectors que es reben com a paràmetres estan normalitzats
  vec3 colRes = vec3(0);
  if (dot (L, NormSCO) > 0)
    colRes = colFocus * fmatdiff * dot (L, NormSCO);
  return (colRes);
}
```

```
vec3 Especular (vec3 NormSCO, vec3 L, vec3 vertSCO, vec3 colFocus) //Phong
{
  // Fixeu-vos que SOLS es retorna el terme especular!
  // Assumim que els vectors estan normalitzats
  vec3 colRes = vec3 (0);
  // Si la llum ve de darrera o el material és mate no fem res
  if ((dot(NormSCO,L) < 0) || (fmatshin == 0))
    return colRes;  // no hi ha component especular

  vec3 R = reflect(-L, NormSCO); // equival a: 2.0*dot(NormSCO,L)*NormSCO - L;
  vec3 V = normalize(-vertSCO); // perquè la càmera està a (0,0,0) en SCO

  if (dot(R, V) < 0)
    return colRes;  // no hi ha component especular

  float shine = pow(max(0.0, dot(R, V)), fmatshin);
  return (colRes + fmatspec * colFocus * shine);
}
```

El que hem de fer ara s'ha de fer per a cadascun dels focus que tenim a l'escena. Calculem el vector L per a cada focus fent la resta de posFocus-vertexSCO, i després normalitzar aquesta L per poder-li passar a les funcions de càlcul d'il·luminació. En aquest cas (Entrega 4: il·luminació 2122Q2), tenim 3 focus:

```
void main()
{
    //Calcular el valor de L per a cada focu
    //focus1 de càmara
    vec3 lSCO1 = posFocus1-vertexSCO.xyz;
    //focus2 d'escena
    vec3 lSCO2 = posFocus2-vertexSCO.xyz;
    //focus3 del Patricio
    vec3 lSCO3 = posFocus3-vertexSCO.xyz;

    //Normalitzar vectors: N, L
    lSCO1 = normalize(lSCO1);
    lSCO2 = normalize(lSCO2);
    lSCO3 = normalize(lSCO3);
    vec3 Fnormal = normalize(fnormal);
    FragColor = vec4(Ambient()+
        Especular(Fnormal, lSCO1, fvertex, colorFocus1)+Difus(Fnormal, lSCO1,
colorFocus1)+
        Especular(Fnormal, lSCO2, fvertex, colorFocus2)+Difus(Fnormal, lSCO2,
colorFocus2)+
        Especular(Fnormal, lSCO3, fvertex, colorFocus3)+Difus(Fnormal, lSCO3,
colorFocus3), 1.0);
}
```

> NOTA: Si tens mes d'un focus la ambient **només** s'ha de sumar **1** cop

4.3 MyGLWidget

4.3.1 Focus d'il·luminació

Els 3 focus són de tipus diferents, el primer és de càmera (està en la mateixa posició i direcció que la càmera), per tant no cal tocar-lo, amb el valor actual de posFocus està bé ja que aquest ja està en SCO. (No s'ha de multiplicar per View)

```
glm::vec3 posFocus1 = glm::vec3(0.0);           //Ja esta en SCO
glUniform3fv(posFocusLoc1, 1, &posFocus1[0]);
```

El segon focus és d'escena, per tant està en SCA (cooordenades d'aplicació). L'hem de passar a SCO, i per fer-ho el multipliquem per la View:

```
glm::vec3 posFocus2 = glm::vec3(10.0, 3.0, 5.0);
posFocus2 = glm::vec3(View*glm::vec4(posFocus2, 1.0));   //El passem a SCA
glUniform3fv(posFocusLoc2, 1, &posFocus2[0]);
```

El tercer focus és un focus de model (i l'ehm de passar a SCM), un focus que l'enunciat ens diu que segueix la punta del cap més alta del Patricio. Ara no només hem de multiplicar el focus per la View, sinó que també l'hem de multiplicar per la TG del Patricio, i ha d'estar actualitzada a cada cop que el Patricio es mou:

```
TGPatricio = glm::translate(glm::mat4(1.f), glm::vec3(5.0,0.0,5.0));
TGPatricio = glm::rotate(TG, gir*(float)M_PI/180, glm::vec3(0, 1, 0));
TGPatricio = glm::translate(TG, glm::vec3(0.0 ,0.0, -3.0));
TGPatricio = glm::scale(TG, glm::vec3(escala, escala, escala));
TGPatricio = glm::translate(TG, -centreBasePatr);

glm::vec3 posFocus3 = glm::vec3(centreBasePatr[0], maxYPat, centreBasePatr[2]);
posFocus3 = glm::vec3(View*TGPatricio*glm::vec4(posFocus3, 1.0)); //El passem a
SCModel
glUniform3fv(posFocusLoc3, 1, &posFocus3[0]);
```

I evidentment, els hi hem de donar un valor al seu color:

```
glm::vec3 colorFocus1 = glm::vec3(0.9, 0.0, 0.9); //camara
glUniform3fv(colorFocusLoc1, 1, &colorFocus1[0]);
glm::vec3 colorFocus2 = glm::vec3(0.9, 0.9, 0.9); //escena
glUniform3fv(colorFocusLoc2, 1, &colorFocus2[0]);
glm::vec3 colorFocus3 = glm::vec3(0.9, 0.9, 0.2); //Patricio
glUniform3fv(colorFocusLoc3, 1, &colorFocus3[0]);
```

Tant el posFocus com el colorFocus es passen als shaders declarant posFocusLoc i colorFocusLoc com a uniforms en el MyGLWidget i en el carregaShaders(). A vegades també poden estar harcodejats be per ells o per l'enunciat mateix.

4.3.2 Material dels objectes

El com es veu un objecte no només depèn dels focus sinó també del material de què està fet aquest. En algun examen et demana canviar el material, i el que hem de fer és anar a una funció que es digui iniMaterial() o algo semblant, i canviar-li els paràmetres:

```
void LL4GLWidget::iniMaterialTerra ()
{
  // Donem valors al material del terra
  amb = glm::vec3(0.1,0,0.2);
  diff = glm::vec3(0.4,0,0.8);
  spec = glm::vec3(0.8,0.8,0.8);
  shin = 100;
}
```

Normalment el color ambient no l'hem de tocar. Quan se'ns demana que l'objecte sigui d'un color en específic, hem de canviar la difusa (diff) pel color demanat. Quan se'ns demana que la taca especular sigui d'un color en concret, hem de canviar l'especular (spec). Perquè es vegi la taca especular i el material sigui brillant (que no sigui opac), llavors el valor de shininess (shin) >= 50. En l'exemple el terra és de color magenta i brillant.