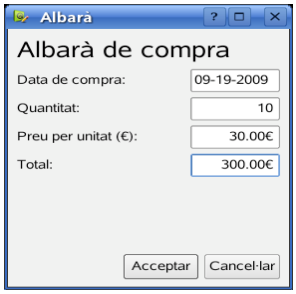
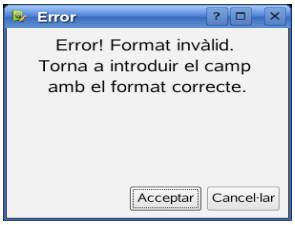
VIG – EXAMEN FINAL – QUADRIMESTRE PRIMAVERA CURS 08-09. **TEMPS: 2h**

==========

1. (1 punt) Un usuari està interactuant amb una aplicació, entra les dades que apareixen a l'esquerra de la figura i prem "Acceptar". Immediatament li surt la finestra d'error que apareix a la dreta de la figura. Analitza aquest missatge d'error, en funció de la finestra que l'ha originat, des d'un punt de vista de disseny d'interfícies i d'usabilitat. Si consideres que s'ha de millorar el disseny, proposa, raonadament, els canvis que faries.





El missatge que es dóna a la finestra d'error està mal dissenyat perquè no informa d'on s'ha produït l'error ni apunta possibles solucions. Això és especialment greu perquè la finestra que l'ha originat està dissenyada de tal forma que pot induir a error.

En funció del que es veu a la primera finestra, l'error pot estar ocasionat, tant a la introducció de les dades de data (no es sap exactament el format d'entrada perquè la finestra no ho indica i hauria de fer-ho) o del símbol d'euro que es posa a la quantitat econòmica. A banda d'això, en un missatge d'error no té sentit un botó d'acceptar junt amb el de cancel·lar, és un missatge informatiu, el de cancel·lar sobra.

Les millores a aplicar serien:

- A la primera finestra: Informació del format de la data i de si la quantitat ha de portar el símbol d'euro o no, també seria interessant saber si el separador ha de ser el punt o la coma.
- A la segona finestra: El missatge d'error ha d'especificar el camp al qual s'ha produït i eventualment proporcionar informació que ajudi a solucionar l'error, per exemple, si l'error està en un camp data, recordar que allò ha de ser una data vàlida i en el format (dia-mes-any o mes-any-dia) adequat. Treure el botó de cancel·lar.

2. (1 punt) Tenim un cub de costat 20 centrat a l'origen, i una càmera que té l'observador a (0,8,0), el vrp a (0,0,0) i el vector up (0,0,1) i amb un window de (-20, 20, -20, 20) i znear=0.1 i zfar=20. Si visualitzem aquesta escena amb el culling activat, el z-buffer activat i un focus de llum blanca situat a la posició de l'observador, veiem que la visualització no mostra res en la imatge al viewport. Per què?

Segons les dades de la càmera, l'observador es troba, dins del cub. D'acord amb el volum de visió definit, veuria la part posterior d'alguna de les cares del cub. Tanmateix, com tenim el culling activat, les cares del cub no seran potencialment visibles; per tant, seran eliminades, no seran rasteritzades i no es pintarà res en la pantalla.

3. (1 punt) Una aplicació requereix seleccionar objectes per a modificar la seva grandària. Decidim implementar la selecció utilitzant el mode de selecció d'OpenGL amb la gluPickMatrix(). Un estudiant dubte respecte l'ordre de les instruccions en la inicialització de la matriu de projecció (GL_PROJECTION). Li pots ajudar i explicar, raonadament, la solució?

L'ordre de les instruccions suposant que ja teniem el tipus de càmera definit d'una certa manera és:

```
// Obtenim la matriu de projecció activa
GLfloat projeccio[16];
glGetFloatv (GL_PROJECTION_MATRIX, projeccio);
// Obtenim la definició del viewport actiu
GLint viewport[4];
glGetIntegerv (GL_VIEWPORT, viewport);
// Redefinim la matriu de projecció afegint-li la transformació de selecció generada
// amb la crida gluPickMatrix
```

```
glMatrixMode (GL_PROJECTION);
glPushMatrix();     // farem el glPopMatrix() quan sortim del mode de selecció
glLoadIdentity ();
gluPickMatrix (xclick, yclick, ample, alt, viewport); // (xclick, yclick) = pixel pitjat
glMultMatrix (projeccio);                             // (ample, alt) = dimensions selecció
```

La TS (transformació de selecció) és una transformació d'escalat que produeix que el volum de visió normalitzat de la càmera actual quedi restringuit al volum que es projecta en l'area de selecció. Per tant, aquesta transformació s'aplicarà als vèrtexs després de la TP (transformació de projecció) i abans de la TMD (transformació món-dispositiu). Com que cal que s'apliqui al punts després de la TP, i OpenGL multiplica les matrius i els punts per la dreta, cal que multipliquem primer la matriu TS i després la TP per a què la multiplicació de matrius a aplicar al punt sigui: **TS***TP*TC. D'aquesta manera, només la geometria que es projecta en la zona de selecció quedarà dins del volum normalitzat. La TS s'acumula a la TP en la pila GL_PROJECTION.

4. (1 punt) Un cub amb constants de material Kd=(0.8,0,0.8) i Ks=(1,1,1) i N=100, és il·luminat amb un focus que emet llum de color (1,1,0). No hi ha llum ambient. La càmera (correctament definida) és axonomètrica i l'observador i el focus estan a una distància 10 d'una cara (i mirant cap a ella) sobre una recta que és perpendicular a la cara i que passa pel seu centre. Indica, raonant la resposta:
   a. quins colors observa l'observador en el cub si s'utilitza *FLAT shading* (colorat constant)? Indica els colors dels vèrtexs en RGB i HSB.
   b. quins colors observa l'observador en el cub si es pinta amb *SMOOTH shading* (colorat de Gouraud)?

Donat que no hi ha llum ambient, **la component ambient del càlcul d'il·luminació serà nul·la**.
Com que l'observador i el focus de llum estàn alineats tots dos amb el centre d'una cara del cub, **tampoc podrem observar especularitat en els seus vèrtexs,** perquè l'observador no podrà estar en cap cas en la direcció de la reflexió especular.
Per tant, el càlcul d'il·luminació en els vèrtexs de la cara del cub tindrà només component difusa. El càlcul serà:

$\mathbf{I(P_i) = Kd * I_{focus} * \cos(\alpha)}$ on α és l'angle que forma el vector $\mathbf{P_iFocus}$ i la normal de la cara

Com que la llum del focus no té component blava i la Kd no té component verda, el color resultant en els vèrtexs tindrà **només component vermella**, i serà el mateix color en tots 4 vèrtexs perquè l'angle format per la direcció P_iFocus i la normal a la cara és en tots 4 casos el mateix.

Així doncs el color en RGB serà: (0.8*cos(α), 0, 0),
Que en HSB serà (0, 1, 0.8*cos(α)) perquè H=0 correspon al vermell, és un color pur i per tant S=1 i la B és la intensitat màxima del color que és 0.8*cos(α).

I com que el color és el mateix en tots els vèrtexs de la cara, **el resultat serà també el mateix** tant si es fa *FLAT shading* com si es fa *SMOOTH shading*.
FLAT shading: calcula el color en un vèrtex de la cara i posa a tota la cara aquest mateix color.
SMOOTH shading: calcula el color en cada vèrtexs de la cara i interpola els colors en els fragments del mig

5. (1 punt) El curs vinent, un estudiant de VIG et comenta que ha de crear una escena que ha d'utilitzar moltes instàncies d'un mateix model. Li preguntes si utilitza "Display Lists" per a millorar l'eficiència del pintat. El company et pregunta: per a què serveix una DL? (és a dir, per què millora l'eficiència) , i què cal modificar del codi per a utilitzar-les? Contesta al teu company, raonadament, cadascuna de les preguntes.

Una Display List és un conjunt de comandes OpenGL que serveixen per dibuixar primitives gràfiques i que s'emmagatzemen (compilades) en el servidor (GPU). Un cop creada la DL pot ser utilitzada repetidament des de l'aplicació sense necessitat de transmetre totes les seves instruccions ni que aquestes siguin avaluades. Per tant, la seva utilització minimitza els temps de transmissió (i de procés) entre Client (CPU) i servidor (GPU). El principal inconvenient de les DL és que un cop creades no es poden modificar; per tant, no poden incloure instruccions que utilitzin variables que modifiquem a través del codi o de la interfície.

Per utilitzar DL en la visualització d'una escena que requereix pintar moltes instàncies d'un determinat model cal:

- crear els identificadors de les diferents DL que volguem utilitzar; tantes com objectes primitius.
- per a cada objecte primitiu, crear una DL (glNewList()... glEndList()) que inclogui les seves comandes de pintat que no s'haguin de modificar. Per exemple, el codi que fem servir per a pintar les seves cares.
- substituir, en el render de cada instància, la crida a pintar objecte primitiu per la execució de la display list (glCallList()) associada amb ell.

6. (1 punt) En les inicialitzacions prèvies al pintat d'una escena, suposa que tenim la següent seqüència d'instruccions OpenGL que defineix una càmera amb un *window* quadrat i un *viewport* també quadrat:

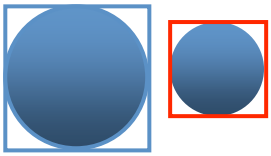
```
gluPerspective(myFovy, 1.0, myNear, myFar);
glViewport(0, 0, 400, 400);
```

Quina diferència s'observaria en la visualització de l'escena si les canviem per:

```
gluPerspective(myFovy, 2.0, myNear, myFar);
glViewport (0, 0, 400, 400);
```

Com s'hauria de redefinir el *viewport* per tal que les imatges obtingudes siguin similars en ambdues visualitzacions? Raona la teva resposta.

En el segon cas, s'ha modificat la relació d'aspecte del window a 2.0 (s'ha fet el doble d'ample), mentre que la del viewport continua essent 1 (quadrat). Per tant, ara l'escena és veurà deformada com si fos el doble d'alta que en la visualització anterior.
Per a no tenir aquest efecte, cal redefinir el viewport per a què també tingui una relació d'aspecte de 2. Per tant, si volem garantir que hi cap en la finestra gràfica, una opció és que la seva alçada sigui la meitat que en el primer codi: `glViewport (0, 0, 400, 200); rav=400/200 o sigui rav=2.`



Window i viewport inicials

Window i viewport amb el segon codi

7. (1 punt) Volem visualitzar un cilindre aproximat per cares planes. Tenim un vector amb les coordenades de tots els vèrtexs i altre vector que ens permet coneixer, per a cada cara, la seva normal i els índexs dels seus vèrtexs. Sabem que les dues primeres cares de la llista de cares són les tapes del cilindre. Volem pintar el cilindre suavitzant **només** les arestes entre les cares que aproximen la part corbada del cilindre. Les arestes de les tapes del cilindre no s'han de suavitzar. Indica l'algorisme requerit per a fer el pintat de la geometria. Si s'escau, pots modificar/afegir informació a l'estructura de dades tot indicant com es calcula. *Observació:* No cal definir la càmera.

Per a aconseguir la visualització requerida les tapes del cilindre s'han de pintar (càlcul del color en el seus vèrtexs) fent servir normal per Cara (la normal de la cara que tenim en el model); en canvi, les cares laterals s'han de pintar fent servir normal per vèrtex; és a dir, una normal que aproxima la normal del cilindre en el vèrtex. Com que només volem suavitzar les cares laterals, aquesta normal la podem calcular ponderant la normal de les dues cares larerals que comparteixen el vèrtex. Si, a més, fem servir Gouraud shading aconseguirem colors similars a les dues bandes de l'aresta que comparteixen dues cares laterals, i, per tant, disminuirem la percepció de l'aresta.

Per tant, caldria un nou camp per a cada vèrtex que contingui la normal promig de les dues cares laterals que arriben al vèrtex (noteu que no cal considerar la normal de la tapa).

```
inicialitzar camp de normal de cada vèrtex
per cada cara lateral
  per cada vèrtex
     sumar la normal de la cara a la normal del vèrtex
  fper
fper
normalitzar les normals
```

Algorisme de pintat

```
per les dues cares tapes
    glBegin (GL_POLYGON)
    glNormal (cara.normal)
    per cada vèrtex
        glVertex(x,y,z)
    pfer
    glEnd()
fper

per les cares laterals
    glBegin(GL_POLYGON)
    per cada vèrtex
         glNormal (vertex.normal)
         glvertex (x,y,z)
    fper
    glEnd()
fper
```

8. (1.5 Punts) El següent codi permet pintar un cub de costat 6 centrat en el punt (0,3,0) . El volem il·luminar amb dues llums. La llum 0 s'ha de moure amb la càmera. La llum 1 estarà sempre ubicada sobre el cub en la recta que passa pel seu centre en la direcció de l'eix Y de l'aplicació i a distància 3 del centre de la seva cara superior. Contesta raonadament les respostes.

- Indica les crides a `glLight` necessàries per a posicionar ambdues llums i el lloc del codi en que les posaries. *Observació:* els colors de les llums ja estan inicialitzats, només has de fer la crida per a indicar la posició.
- Imagineu que volem moure el cub en una direcció horitzontal respecte de la pantalla. Indica i justifica les modificacions requerides del codi.

```
1. setProjection(); //inicialitza la gluPerspective
2. glMatrixMode(GL_MODELVIEW);
3. glLoadIdentity();
4. glTranslatef(0,0,-dist);
5. glRotatef(-angleZ,0,0,1);
6. glRotatef(angleX,1,0,0);
7. glRotatef(-angleY,0,1,0);
8. glTranslatef(-VRP.x,-VRP.y,-VRP.z);
9. Pinta_Cub();
```

La llum 0, com que s'ha de moure amb la càmera, s'ha de posicionar respecte de l'observador. La matriu de ModelView activa en el moment del seu posicionament ha de ser la identitat; per tant, s'ha de definir després de la instrucció 3. La seva posició és la que volguem que quedi respecte de l'observador. Si està en l'observador serà (0,0,0). El codi a afegir:

```
float pos0[4]={0,0,0,1};
glLigtfv(GL_LIGHT0, GL_POSITION, pos0); //llum fix respecte la càmera
```

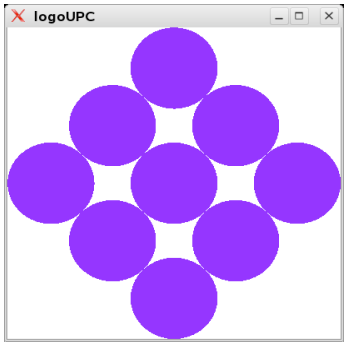
La llum 1, com ha de ser fixa respecte l'escena, es defineix en coordenades de l'aplicació i quan a la ModelView tinguem la matriu de càmera; per tant, després de la instrucció 8. La seva posició serà (0,9,0) d'acord amb el que diu l'enunciat. El codi a afegir:

```
float pos1[4]={0,9,0,1};
glLigtfv(GL_LIGHT1, GL_POSITION, pos1); //llum fix respecte l'escena
```

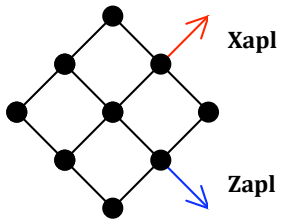
Si volem que el cub i el llum 1, es moguin en una direcció horitzontal respecte de la pantalla, cal afegir una translació, que afecti només al cub i al llum segons l'eix x de l'observador; per tant, s'ha d'ubicar després de la instrucció 8 i abans de definir la posició del llum,. El vector de translació el podem consultar en la Matriu de ModelView. El mòdul de la translació suposem que s'ha calculat en funció del moviment del ratolí. Cal afegir, després de la instrucció 8:

```
GLfloat mv[4][4];
glGetFloatv (GL_MODELVIEW_MATRIX, &mv[0][0]);
Vector xobs={mv[0][0],mv[0][1],mv[0][2])};
glTranslatev(xobs.x*d, xobs.y*d, xobs.z*d);
```

9. (1.5 punts) La Degana de la FIB ens ha demanat que fem un logo de la UPC per computador, però amb el conjunt d'esferes girat 45 graus (veure figura). Totes les esferes tenen el seu centre ubicat en el pla XZ (és a dir, la seva y=0). Disposem d'una acció `pinta_esfera(R)` que pinta una esfera de radi *R* centrada en l'origen de coordenades. El viewport és quadrat i ocupa tota la finestra gràfica i, per tant, es defineix amb glViewport(0, 0, w, h). Defineix TOTS els paràmetres d'una càmera axonomètrica que permet generar la figura adjunta i indica el codi requerit per a generar l'imatge (definició de la càmera i pintat de la geometria). Dóna el codi per a definir la ModelView tant amb transformacions geomètriques com amb gluLookAt.



Esquema centres esferes



Primer decidim com situem les esferes respecte del sistema de coordenades de l'aplicació. Sabem que han d'estar situades en el pla XZ. Situarem les esferes aliniades amb els eixos coordenats (X i Z). En aquest cas hem decidit (com mostra la figura) situar l'esfera central centrada a l'origen de coordenades, de manera que els centres de les altres 8 tenen coordenades (X, 0, Z) amb X i Z amb valor -2R, 0 o 2R. Com no ens diuen el R de les esferes, suposem que tenen R=1. Per tant, només, cal traslladar les esferes a la seva posició abans de pintar-les.

Donat que les esferes estan situades en el pla XZ, l'observador s'ho haurà de mirar des d'una posició elevada, és a dir des de l'eix de les Y (suposem la part positiva de l'eix de les Y). I mirant cap al centre de l'esfera que es troba al mig. (que està centrada a l'origen). El vector up haurà d'estar girat 45 graus, agafem per exemple el (1, 0, -1);
Com les esferens tenen R=1, el quadrat format pel centres de les 9 esferes (veure figura) fa 4 de costat, i per tant la seva diagonal farà 4*arrel(2). A aquesta dimensió li haurem de sumar dos cops el radi per a que càpiguen les esferes de les cantonades i aquesta serà la dimensió que necessitem per a l'amplada i alçada del window. Així doncs, la càmera serà:
OBS = (0, 5, 0); VRP = (0, 0, 0); up = (1, 0, -1);
Window = (-(2*arrel(2)+1), 2*arrel(2)+1, -(2*arrel(2)+1), 2*arrel(2)+1); Znear = 4; Zfar = 6;

El codi serà doncs el següent:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho (-(2*arrel(2)+1), 2*arrel(2)+1, -(2*arrel(2)+1), 2*arrel(2)+1, 4, 6);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
gluLookAt (0, 5, 0, 0, 0, 0, 1, 0, -1);
for (int i=-1; i<2; ++i)   // translació en Z
{
    for (int j=-1; j<2; ++j)  // translació en X
    {
        glPushMatrix ();
        glTranslatef (2*j, 0, 2*i);
        pinta_esfera (1);
        glPopMatrix ();
    }
}
```

I la matriu Modelview amb transformacions geomètriques es posaria:

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslate (0, 0, -5);
glRotatef (45, 0, 0, 1);
glRotatef (90, 1, 0, 0);
```