Laboratori IDI: OpenGL, bloc 2

Professors d'IDI, 2012-13.Q2

16 de setembre de 2013

Aquest bloc està pensat per a què el feu en **2 sessions de laboratori**, experimentant amb l'efecte de les diferents crides i paràmetres. Per a això, us suggerim una sèrie d'experiments, però a més esperem de vosaltres que afegiu els propis, fins a convèncer-vos que enteneu per què passen les coses que passen en el vostre programa. Hem intercalat el signe '►' per assenyalar punts específics en què es planteja un exercici fonamental o una nova funcionalitat que, probablement, reutilitzareu al llarg d'aquest bloc o d'altres, i el signe '►' per marcar quelcom que necessita d'experimentació de paràmetres o instruccions. En cas de dubtes, aprofiteu el professor de laboratori, que us donarà pistes de què mirar o on buscar. Podeu consultar internet per cercar informació, documentació,... Però NO cerqueu codi i retalleu i enganxeu als vostres programes. Fent-lo no aprendreu gaire, i durant les proves de laboratori no hi ha accés a internet. Vigileu, perquè aquest bloc pot semblar superficialment senzill, però és un dels que acostumen, al capdavall, a causar més dificultats als estudiants. Recordeu d'anar guardant les aplicacions dels exercicis que feu.

1 Usant geometria més complexa

És hora que fem les nostres escenes un pèl més interessants. `glut` incorpora vàries rutines que generen geometria en una posició estàndard (elles mateixes s'encarreguen de fer les crides necessàries a `glBegin()--glEnd()`, i d'enviar les coordenades dels vèrtexs corresponents). Concretament disposeu de `glutWireCube`, `glutWireCylinder`, `glutWireSphere`, `glutWireCone`, `glutWireTeapot`, `glutWireTorus`, `glutWireTetrahedron`, `glutWireOctahedron`, `glutWireDodecahedron`, `glutWireIcosahedron`, `glutWireRhombicDodecahedron` i `glutWireSierpinskiSponge`.

►Proveu de substituir les instruccions que pintaven el triangle de l'aplicació del bloc 1 per crides a aquestes funcions, per a generar escenes més riques... Recordeu que estem utilitzant el volum de visió de defecte i, per tant, vigileu els valors dels paràmetres. ►Podeu, per exemple, provar de pintar la tetera (pista: utilitzeu 0.5 de grandària)

2 Transformacions geomètriques

En aquest apartat farem servir transformacions geomètriques per a moure els objectes que dibuixem (per a ubicar-los en l'escena o per animar-los), sense haver de calcular nosaltres les coordenades dels vèrtex transformats. Us aconsellem fer servir com objecte a pintar el triangle del bloc 1 (amb un color diferent per a cada vèrtex) o la tetera; en qualsevol cas, cal que tingueu implementat el *callback* de `glutReshapeFunc()` abans de començar. És molt important que entengueu l'ordre en què s'indiquen les transformacions, els seus paràmetres, com es concatenen i com s'apliquen.

2.1 Rotacions

Observació: Abans de fer res, poseu el codi que us indiquem just després de crear la finestra gràfica, altrament, potser tindreu problemes per entendre el sentit dels girs.

```
1 glMatrixMode (GL_PROJECTION);
2 glLoadIdentity();
3 glOrtho (-1,1,-1,1, -1, 1);
4 glMatrixMode(GL_MODELVIEW);
```

►Per tal de veure com funciona aquest mecanisme, afegiu una crida a `glRotated()` a la vostra funció `refresh`. Aquesta funció rep per paràmetres quatre `double`s, el primer dels quals indica l'angle a girar (en graus sexagesimals), i els altres tres són les components x, y i z del vector director de l'eix de rotació (l'eix passa per l'origen de coordenades). Poseu-hi valors adequats, i mireu què succeeix.

►Ara proveu de redimensionar lentament la finestra, o minimitzar-la i maximitzar-la diverses vegades. Què passa? Per què? Recordeu que OpenGL és una màquina d'estats, i que les crides que fem modifiquen l'estat (el context gràfic).

Concretament, les transformacions geomètriques es concatenen amb les que ja hi hagi emmagatzemades en aquest context gràfic. Per a solucionar-lo, incloeu una crida a `glLoadIdentity()` al començament de `refresh()`. Proveu què passa ara, ►i proveu també col·locant el `glRotated()` en diferents llocs de la funció. Mireu d'entendre realment què succeeix, i proveu amb diferents valors dels paràmetres. Proveu, en particular, diferents valors per les components del vector director de l'eix de gir, i deduïu com es troben ubicats els eixos de coordenades de l'escena respecte de la vista que es mostra a la finestra. ►Si voleu podeu pintar els eixos de coordenades però vigileu! que no els heu de rotar.

►Afegiu una segona rotació, al voltant d'un eix linealment independent del de la primera (per exemple, una respecte l'eix X i altre respecte l'eix Y). ►Proveu què passa quan les permuteu. Estudieu un altre cop la diferència amb altres valors dels angles de gir.

►Podeu aprofitar el que ja sabeu sobre *callbacks* per a fer un programa en què es pugui veure el resultat d'incrementar/decrementar el valor dels angles en picar una certa tecla o en funció del desplaçament del ratolí en x o en y, usant `glutMotionFunc()`. ►Proveu també el resultat d'una i altra ordenació de les crides a les rotacions (tot prement una determinada tecla) i/o diferents valors dels paràmetres de les crides sense haver de sortir–editar–recompilar...

Recordeu que per a fer que la finestra es redibuixi després de modificar el que calgui al *callback*, cal cridar a `glutPostRedisplay()`, que marcarà la finestra com necessitada d'un refresc, de forma que la següent iteració pel bucle d'esdeveniments cridarà a `refresh()`.

2.2 Translacions

OpenGL també inclou la funció `glTranslated()` que composa una translació amb la transformació actual. Ara ja sabem on ens interessa incloure aquesta operació: ►Substituïu la rotació al vostre `refresh()` per una translació, i feu que els seus paràmetres (bé, dos dels seus paràmetres, corresponents —per exemple— a les direccions coordenades x i y) es puguin modificar interactivament amb el ratolí (usant `glutMotionFunc()`). Si voleu, podeu tenir una tecla que modifiqui l'estat de l'aplicació per a realitzar translacions o girs. ►També podeu provar que les direccions de translació siguin altres.

►Aquest exercici és prou interesant. Fixeu ara uns valors de translació, i afegiu-hi una rotació. Programeu una tecla per a intercanviar l'ordre en què es fan aquestes dues transformacions, i mireu d'entendre què passa. ►Proveu amb diferentes posicions relatives del vector director de l'eix de gir respecte del vector de translació.

2.3 Escalats

Finalment (en quan a transformacions), OpenGL inclou la funció `glScaled()`. Com en el cas de les translacions, ►feu un programa on s'apliqui un escalat al triangle (o objecte) que dibuixeu, que es pugui modificar interactivament (de forma que l'escalat pugui ser diferent en els diferents eixos). També igual que abans, ►mireu d'experimentar intercanviant un escalat i una rotació, i també intercanviant un escalat amb una translació. Què passa?

2.4 Diversos objectes

Naturalment, en general voldrem poder representar geometria més complexa que un únic triangle o objecte. ►Modifiqueu la vostra aplicació per a què dibuixi... **dos** triangles (o altres dos objectes però, en qualsevol cas, ubicats en posicions conegudes per vosaltres dins del volum de visió). Un cop feta aquesta extensió, què passa quan fem transformacions com als apartats anteriors? Podeu provar, per exemple, amb una rotació.

En general no és això el que voldríem, sinó que voldríem poder aplicar transformacions diferents i específiques a cada objecte de la nostra aplicació. Per a això, OpenGL no emmagatzema en realitat sols una transformació actual, sinó que les transformacions les desa en una estructura de dades de tipus pila, de tal manera que la transformació actual en cada instant és la que està al cim de la pila. Per això disposem de les primitives `glPushMatrix()` (que empila una còpia de la matriu que hi hagi actualment al cim, és a dir de la transformació actual), i `glPopMatrix()` que desempila (però no ens retorna) la matriu del cim, de forma que passa a ser la transformació actual la que ho era quan es va fer el darrer `glPushMatrix()`.

►Feu que quan arrosseguem el ratolí horitzontalment, els dos triangles (o objectes) girin sobre sí mateixos, al voltant d'un eix paral·lel a $(0,1,0)$ que passi pel baricentre del triangle (o objecte) corresponent, però en sentits oposats —recordeu que el baricentre d'un triangle té per coordenades les mitjanes aritmètiques de les coordenades corresponents dels vèrtexs: $\bar{x} = \left(\frac{1}{3}(x_0+x_1+x_2), \frac{1}{3}(y_0+y_1+y_2), \frac{1}{3}(z_0+z_1+z_2)\right)$—. Cal que ho feu cridant `glLoadIdentity()` com a molt una vegada per frame.

3 Un primer exercici

Es vol crear un ninot de neu mitjançant dues esferes i un con que farà de nas. L'esfera inferior ha de tenir el seu centre al (0,0,0) i un radi de 0.4, i la superior, que simula el cap, cal posar-la amb el seu centre a (0,0.6,0) i amb un radi de 0.2.

El con té un radi de 0.1, llargada 0.2, el centre de la seva base a (0.1,0.6,0) i està orientat segons l'eix X. Utilitzeu les funcions `glutWireSphere` i `glutWireCone`.

►Implementeu la possibilitat de poder girar i escalar -uniformemnt- el ninot, tot interactivament a través de tecles per a triar l'acció i el ratolí per a manipular el model. En funció de l'estat de l'aplicació (tecla premuda), el moviment del ratolí ha d'afectar a les rotacions, escalat o a cap de les dues transformacions.

4 Càrrega de models

És hora que puguem fer servir quelcom més interessant que esferes i teteres en les nostres escenes tot afegint el codi necessari per a poder llegir models geomètrics en un format estàndard.

Per a això, trobareu a `/assig/idi/Model` el codi d'una classe `model` que emmagatzema el model geomètric d'un objecte en una estructura de dades senzilla. Per a fer-ho servir tant sols necessitareu veure l'arxiu `model.h`; veureu que la classe `Model` disposa d'un mètode `Model::load(std::string filename)`, que inicialitza aquestes estructures de dades a partir d'un model en format OBJ-Wavefront en disc. Podeu trobar uns quants models en aquest format a `/assig/vig/models`, i també podeu trobar molts per la xarxa. Si els copieu a un altre directori (p. ex. a un portàtil vostre), per cada model `.obj` copieu també (si existeix) l'arxiu del mateix nom amb extensió `mtl`, que conté les definicions dels materials corresponents.

Veureu també que el model està emmagatzemat simplement en tres vectors *stl* un de coordenades (`_vertices`), un altre de components de normals (`_normals`) i un altre de cares (`_faces`). També hi ha mètodes consultors que tornen referències a aquests vectors. Vigileu que retornen referències a vectors **constants**; el codi en què les feu servir haurà de ser "const-correcte".

Tant les coordenades dels vèrtexs com les components de les normals estan emmagatzemades com `double`s contigus, de forma que puguem fer-los servir dient per exemple

```
1 glNormal3dv(&m.normals()[f.n[i]]);
```

En principi no us hauria de caldre consultar el codi a `model.cpp` per a fer-ho servir (tot i que ho podeu fer si voleu, és clar). Tingueu en compte però alguns aspectes:

- El mètode `Model::load()` carrega sols triangles. Malgrat que l'estructura de dades permetria altres cares de major nombre de costats, si n'hi han al model, es triangulen en el moment de llegir-lo. Podeu per tant donar per segur que tots els vectors de vèrtexs del model resultant són de tres components.
- Per contra, el vector de normals pot ser buit, si el model original no contenia normals per vèrtex. En canvi, sempre podreu fer servir el vector `Face::normalC` que la crida a `Model::load()` haurà inicialitzat amb un vector unitari perpendicular al triangle.

►A partir de l'aplicació de l'exercici anterior, creeu una nova aplicació que carregui l'objecte "HomerProves.obj" (aquest model està escalat de manera que cabrà en el volum de visió i sortirà centrat en ell). Podeu passar els models a carregar per línia de comandes, o habilitar una tecla que faci que es llegeixi el nom d'un arxiu pel terminal. Haureu de programar la funció de refresc de manera que es recorrin les cares del model i es dibuixin. Tal com hem dit més amunt, podeu fer servir la primitiva `GL_TRIANGLES` per a totes les cares. Per a la fase de muntatge de la vostra aplicació, feu servir una comanda com `g++ -o vostreBinari vostrePrograma.o model.o -lGLU -lGL -lglut`.

►Modifiqueu el codi anterior per a carregar qualsevol model OBJ i ubicar-lo en l'escena centrat a l'origen de coordenades, sense retallar i sense deformació. Haureu de programar les transformacions geomètriques per a ubicar l'objecte i, per a calcular-les, us recomanem que calculeu la capsa mínima contenidora del model. Utilitzeu les rotacions interactives per a, un cop ubicat i escalat el model, girar-lo per a poder veure'l des de diferents orientacions.

►A continuació us expliquem com fer per a veure l'escena amb parts amagades. És molt simple de programar i us facilitarà entendre la visualització; però podeu deixar-ho per a una propera sessió .... Fins ara, com fèiem servir les rutines `glutWire*`, no ens hem hagut de preocupar. ►Ara cal que sapigueu que OpenGL dibuixarà les cares amb filferros o plenes segons li indiquem fent servir la funció `glPolygonMode`, que té dos paràmetres. El primer selecciona les cares a les quals ens referim (fes servir aquí `GL_FRONT_AND_BACK`[1]). El segon paràmetre és el mode de dibuix, que pot ser `GL_POINT` (sols es dibuixen els vèrtexs), `GL_LINE` (es dibuixa el contorn de la cara) i `GL_FILL` (es dibuixa el triangle omplert). Afegiu una tecla que commuti entre el mode `GL_LINE` i el mode `GL_FILL`. Quan pinteu en mode `GL_FILL`, naturalment voldreu veure les cares que siguin davant, no les darreres que s'hagin enviat a dibuixar. Per a que OpenGL faci servir l'algorisme de *z-buffer* vist a classe, haurà d'estar activat el *flag* `GL_DEPTH_TEST` (que també es controla amb `glEnable()`/`glDisable()`) i per a que aquest mecanisme funcioni, cal que en el moment de refrescar cada frame, esborreu també el buffer de profunditats

[1]Els altres modes són `GL_FRONT` i `GL_BACK`. Aquest paràmetre indica que l'assignació que estem a punt de fer afectarà a les cares quan se les pinti del costat de davant, del de darrera, o —usant el mode que us proposem al text— per qualsevol dels dos costats. Podeu, si voleu, fer que els triangles es vegin omplerts només quan som fora de l'objecte, però si fiquem la càmera dins l'objecte, farà que els que quedin d'esquena a la càmera es vegin sols en filferros, per exemple.

(afegint el flag `GL_DEPTH_BUFFER_BIT` a la crida a `glClear()`). ▶afegiu a la interfície una tecla que commuti entre els estats actiu i inactiu d'aquest *flag*, i comproveu les diferències en el que veieu en un cas i l'altre al llarg dels diferents exercicis d'aquest bloc. Assegureu-vos d'entendre perquè són les diferències (o perquè no n'hi han) en cada cas.

5 Segon Exercici: aplicació a lliurar

Ara podem fer servir tot el que tot just hem après per a resoldre un exercici que posarà a prova la nostra comprensió de les transformacions geomètriques. L'aplicació resultant serà la que heu de lliurar en la primera classe de laboratori del bloc següent: Bloc 3.

Feu una aplicació que inicialment:

- Pinta un quadrat (que simularà un terra) centrat en $(0, -0.4, 0)$, paral·lel al pla ZX i de longitud d'aresta 0.75.
- Pinta un ninot recolçat sobre el terra en el punt $(0, -0.4, 0)$.
- Pinta el `homer.obj` en una cantonada del terra.

Tota l'escena, en l'instant inicial, ha d'estar dins del volum $[-1, 1]^3$ per a no ser retallada per OpenGL.
A més a més, l'aplicació ha de tenir les següents funcionalitats:

- S'ha de poder girar interactivament l'escena amb el ratolí respecte els eixos X i Y.
- En picar la tecla 'c', l'estat de l'aplicació es modifica i al moure el ratolí, el homer és desplaçarà sobre el terra, en la direcció indicada pel ratolí; és a dir, en la direcció de l'eix X si es mou en horitzontal i de l'eix Y si es mou en vertical.