Laboratori IDI: OpenGL, bloc 3, 3 sessions

Professors d'IDI, 2013-14. Q1

7 d'octubre de 2013

Igual que els blocs 1 i 2, aquest està pensat per a que el feu experimentant amb l'efecte de les diferents crides i paràmetres i aneu avançant cap a l'aplicació final que haureu de lliurar. Per a això us suggerim una sèrie d'experiments marcats amb '►', 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. 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!

Hem intercalat el signe '►' per a senyalar punts específics en què es planteja un exercici fonamental o quelcom que requerireu per a l'aplicació final. No us estigueu de fer els altres experiments, naturalment!

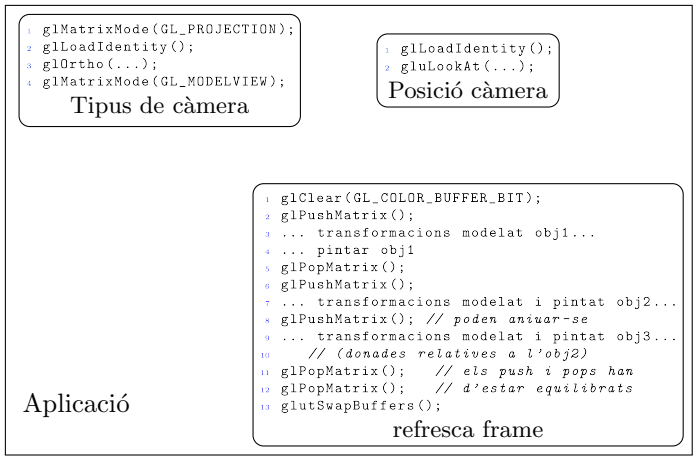
Com als blocs anteriors, cal que mireu de produir codi net, que més endavant pugueu aprofitar. Mireu de deixar una **aplicació "neta"** al final del bloc, en què es puguin activar les diferents funcionalitats (o almenys una versió de cadascuna) via tecles o botons del ratolí (amb modificadors). En la darrera secció d'aquest document, us indiquem les funcionalitats mínimes de l'aplicació que haureu de lliurar.

Aquest bloc us ocuparà **3 sessions** de laboratori. La planificació seria que en la primera sessió -som concients que serà la més carregada de feina però us serà útil de cara a l'exàmen- arriveu fins a la Secció 5; en la sessió 2 fins el primer 'walk' de la Secció 8.2 i en la sessió 3 completeu les funcionalitats de l'aplicació que heu de lliurar.

1 Introducció

L'objectiu d'aquest bloc és que entengueu les inicialitzacions de càmera en l'àmbit d'OpenGL i algunes tècniques bàsiques per a la seva manipulació. Us recomanem que repasseu els conceptes i problemes analitzats en classes de teoria.

Amb les modificacions que introduireu en aquesta pràctica, la vostra aplicació tindrà les parts que us indiquem en la figura (el codi inclòs és a títol d'exemple; no el preneu literalment!). Us recomanem fortament que, per a facilitar el manteniment del codi, cada bloc ho implementeu en un mètode independent.



Cada bloc de codi sols s'hauria d'executar quan sigui necessari: el de dalt a l'esquerra, quan canviem el tipus de càmera (per exemple, si passem d'ortogonal a perspectiva, o si es canvien les mides de la finestra gràfica); el de dalt a la dreta, quan es vol reposicionar la càmera (amb `gluLookAt` o amb transformacions geomètriques); el de sota, cada cop que cal refrescar l'escena. Això és important per a què després pugueu afegir nous aspectes i millores al vostre codi. També és important que entengueu quan convé fer cada una d'aquestes operacions. La inicialització dels paràmetres de la càmera inicial les podeu calcular (i declarar a OpenGL) en un bloc independent.

►Agafeu com a **aplicació de partida** per aquesta pràctica, l'aplicació final del bloc2 però netejant el codi de funcionalitats no requerides i reorganitzant-ho d'acord amb els blocs indicants.

2 Càmera axonomètrica o ortogonal

En el bloc 1 vàreu aprendre que en redimensionar la finestra calia adaptar el *viewport* per a evitar deformacions. Però tal com ho vàrem fer aleshores, amb les eines conceptuals de què disposàvem, es desaprofitava potencialment molta part de la finestra gràfica. A més, fins ara estàvem obligats a posar tota la geometria a dibuixar dins d'un volum determinat (l'anomenat volum de visió) fixat a $[-1,1]^3$.

OpenGL ofereix mecanismes per a solucionar aquesta darrera restricció, i pal·liar el problema de l'aprofitament de la finestra gràfica. La crida `glOrtho` permet definir un volum de visió hexaèdric però entre altres límits. Ara bé, per a poder-ho fer (el definir un altre volum de visió) caldrà que primer coneguem més en detall els mecanismes de transformacions d'OpenGL. En realitat, OpenGL emmagatzema més d'una transformació/matriu. Fins ara, les transformacions que hem fet servir (que bellugaven els models) modificaven la transformació que OpenGL guardava com a `ModelViewMatrix` (resultat de concatenar les transformacions de modelat i de "view" -com sabeu de teoria-). Per a modificar el volum de visualització, hem d' alterar la `ProjectionMatrix`. Per a fer-ho, abans de cridar a `glOrtho` caldrà indicar-ho:

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

Un cop modificada la matriu convenientment, la línia 4 d'aquest tros de codi ens retorna al mode en que estàvem. Això és convenient perquè és molt més freqüent que l'aplicació hagi de modificar la matriu de modelat o la de visualització que no pas la que defineix el volum de visió. D'altra banda és més fàcil desenvolupar un codi si adoptem la convenció que OpenGL està en un "`MatrixMode`" determinat i fix sempre que no es modifiqui localment.

►Agafant la vostra aplicació final del bloc 2 com a punt de partida, mireu d'incloure al *callback* de *reshape* crides a `glViewport` i a `glOrtho` de manera que s'aprofiti al màxim la finestra, no importa com es redimensioni, però sense tenir deformacions. No cal que la solució sigui un màxim —quant a l'aprofitament de l'espai— sinó un bon compromís entre aprofitar l'espai i calcular fàcilment els paràmetres de les crides. Per a fer-ho i que us serveixi per quan feu altres escenes:

- Recordeu que, de moment, teniu la posició i orientació de la càmera per defecte d'OpenGL.
- Utilitxeu el concepte *d'esfera contenidora de l'escena* que heu vist a classe.
- Assegureu que l'esfera és completament interior al volum de visió, d'aquesta manera, encara que gireu l'escena, no es sortirà d'ell. Recordeu que en la càmera orthogonal zN pot ser negatiu.
- Si heu de fer servir al màxim la finestra gràfica en pantalla, òbviament ara el viewport que declareu a OpenGL haurà de cobrir-ne la seva totalitat, i, per tant, haureu d'evitar la deformació donant la mateixa *relació d'aspecte* al volum de visió —bé, a la cara anterior del volum de visió, és a dir al *window*—.

3 Posicionament de la càmera amb transformacions geomètriques (angles d'Euler)

Fins a aquest moment, el nostre observador (o la nostra càmera) ha estat quiet a l'origen de coordenades. Sovint aquesta manera de pensar no és òptima, i en comptes d'això preferim poder col·locar-lo en qualsevol altra posició. Hi ha dues maneres principals d'indicar-ho a OpenGL: utilitzar `gluLookAt` o ubicar els objectes respecte la càmera de defecte (ubicada en l'origen de coordenades de l'aplicació, mirant en la direcció de eix Z negatiu i amb la part vertical de la càmera segons l'eix Y). Comencem per aquesta segona.

Podem posicionar els objectes (de fet, l'escena) respecte la càmera fent servir transformacions geomètriques ja que el que importa per a obtenir una vista concreta **és la posició relativa entre la càmera i l'escena**. Alternativament, les transfomacions geomètriques es poden interpretar com que estem **actuant (de la manera oposada) sobre la càmera** per a ubicar-la respecta l'escena.

Una tècnica concreta per a posicionar la càmera és indicant el seu punt d'enfoc (View Reference Point, VRP), i la posició de la càmera sobre una esfera de radi *dist* centrada en VRP donant els seus angles d'Euler. Noteu que *dist* és la distància entre VRP i l'observador. Si voleu veure tota l'escena, centrada en el volum de visió, una solució és que VRP sigui el centre de l'esfera contenidora de l'escena i *dist* un valor superior al seu radi.

►Inicialitzeu la càmera (matriu de ModelView) amb transformacions geomètriques. Per a fer-ho, podeu traslladar el VRP a l'origen, de forma que les rotacions que es facin a continuació girin al seu entorn. Aleshores s'apliquen tres rotacions al voltant dels eixos y, x, i z (en aquest ordre), que corresponen a la direcció (del pla x-z) des de la que es mira, l'elevació (o la inclinació de la càmera "cap avall"), i la rotació de la càmera al voltant del seu propi eix. Finalment, cal allunyar el VRP a la distància *dist* que li correspon de la càmera. **Important!**, noteu que ara teniu tota l'escena per davant de l'observador (origen de coordenades), per tant, heu de modificar els paràmetres de la crida a `glOrtho` per a que l'escena no quedi retallada.

Observació: cal que aquestes transformacions es defineixin fora de la funció de refresc, i que **a la funció de refresc ja no es cridi a** `glLoadIdentity`. En canvi, per a evitar l'acumulació de transformacions, parentitzeu les transformacions geomètriques que us calguin entre `glPushMatrix` i `glPopMatrix`. En altres paraules, cal que sols es cridi a las transformacions anteriors quan s'inicialitza la càmera o quan hom la mogui. Això pot semblar una complicació però permetrà després un millor control de l'aplicació, a més de ser més eficient car no cal definir repetidament unes variables tot donant-les-hi sempre el mateix valor...)

Observeu que ara tenim desacoblades dues transformacions: unes defineixen la posició de la càmera (de fet dels objectes respecte la càmera), i altres modifiquen els models (les crides que tingueu a `glScale, glRotate`, etc). En realitat totes estan composant-se en una única transformació que s'emmagatzema en la matriu de ModelView. Cal doncs que l'assignació de la posició de la càmera es defineixi primer, i les altres transformacions s'afegeixin després. Per aquest motiu cal que fem el `LoadIdentity` just abans de les crides a:

```
1 glTranslated(0., 0., -dist);
2 glRotated(-anglez, 0., 0., 1.);
3 glRotated(anglex, 1., 0., 0.);
4 glRotated(-angley, 0., 1., 0.);
5 glTranslated(-VRP.x, -VRP.y, -VRP.z);
```

►Experimenteu amb diferents angles i distàncies, i canviant el VRP. L'escena mai hauria de quedar retallada.

►Afegiu el codi necessari per a què es puguin canviar els angles d'Euler interactivament arrossegant el ratolí, associant el moviment horitzontal al gir al voltant de l'eix Y (canviat de signe), i el moviment vertical al gir al voltant de l'eix X. **Noteu** que el el bloc 2 ja feieu aquestes rotacions per a girar l'escena.

4 Càmeres perspectives

Per tal d'augmentar el realisme, afegint l'empetitiment perspectiu (és a dir el fet que les coses llunyanes es veuen més petites que les properes), podem fer servir una càmera perspectiva, en comptes d'una càmera axonomètrica.

Per a programar la càmera perspectiva caldrà substituir la crida a `glOrtho` per una crida a `gluPerspective(fovy, aspect, near, far)` (per tant amb les mateixes condicions de fer-se en "`MatrixMode`" `GL_PROJECTION`, com al tros de codi de la Secció 2). També caldrà que afegiu, en linux, l'*include* `GL/glu.h`, i que afegiu la llibreria corresponent a la instrucció de muntatge -lGLU.

Aquest model de càmera s'assembla més al d'una càmera real, i per tant les escenes es veuran més fidelment, especialment en quan a la seva profunditat. Tanmateix, heu de tenir algunes consideracions en compte:

- El paràmetre `fovy`, que mesura l'angle entre el pla de retall superior i el pla de retall inferior, s'ha de donar en graus sexagesimals. Tingueu-ho en compte si feu servir funcions trigonomètriques en el seu càlcul, car aquestes operen en radians.
- `aspect` és la relació d'aspecte del *window*. Es defineix com l'amplada dividida per l'alçada. Però alerta que quan feu la divisió es faci en coma flotant, no entre enters!
- Els paràmetres `near` i `far` han de ser estrictament més grans que zero. Representen distàncies a la càmera. Tot el que sigui a menys de `near` o a més de `far` en la direcció de les z es retallarà, i no apareixerà en pantalla.
- L'anterior implica que ara, l'observador no pot trobar-se dins del volum de visió. Per tant, quan tinguem la càmera posicionada amb angles d'Euler—secció anterior—, cal que la distància a la que es troba l'escena de l'observador sigui superior a `near` per tal que el retallat no l'elimini.
- Cal que aprofiteu l'espai el millor possible (és a dir que cal fer que l'escena es vegi tant gran com es pugui, sense retallar-ne cap part; tanmateix, si aquest òptim és costós de calcular exactament, és més raonable adoptar una bona aproximació que sigui fàcil de calcular. Però no és acceptable una que es basi en "factors de seguretat" i pugui deixar l'escena empetitida al mig del viewport...).
- Com en els altres casos, cal fer un tractament diferenciat quan `aspect` $>= 1$ i quan `aspect` < 1 per a evitar deformacions.

►Inicialitzeu una càmera perspectiva que permeti veure tota l'escena, aprofitant l'espai del viewport (que sempre serà tota la finestra gràfica) i assegurant que no hi ha deformacions en fer un `resize`.

►Programeu una tecla (per exemple la `'p'`) que passi a càmera perspectiva en la vostra aplicació, i una altra (per exemple la `'x'`) que torni a la càmera axonomètrica de la Secció 2. Quan tingueu programada la càmera perspectiva, mireu d'apreciar les diferències.

5 Exercici: Navegació per veure tota l'escena

A partir de l'aplicació final del bloc 2, feu una aplicació amb les següents funcionalitats (de fet són les que hauríeu de tenir implementades si heu seguit el guió):

- En iniciar l'aplicació es mostri l'escena des d'una posició arbitrària (sense retallar), amb una càmera perspectiva i aprofitant la grandària del viewport (definit com tota la finestra gràfica). La càmera inicial estarà definida amb angles d'Euler.
- Permeti commutar entre les càmeres perspectiva i axonomètrica amb les tecles 'p' i 'x'. En cap cas ha d'haver-hi deformació si es fa un "resize" de la finestra gràfica.
- Permeti inspeccionar l'escena modificant els angles d'Euler amb el ratolí.
- Permeti fer un "reset" per tornar a la visualització de l'escena des de la càmera inicial.
- Hi hagi un help que mostri com activar les diferents funcionalitats.

6 Zoom

A banda de les rotacions, les aplicacions sovint necessiten altres modificacions de les vistes. En aquest apartat aprendrem a implementar dues, el zoom i el pan, associades (en sentit laxe) a translacions.

El zoom —en càmeres perspectives— correspon a apropar-se a l'escena. Pot implementar-se de dues maneres diferents: com un canvi efectiu de la posició de la càmera, fent-la avançar (o retrocedir) al llarg de l'eix de visió, o bé com un "canvi en l'òptica de la càmera", que correspon més exactament amb l'operació de fer zoom amb una càmera real.

La primera opció **(OPTATIVA)**, acostar o allunyar la càmera, és fàcil d'implementar en totes les formes de posicionar la càmera. Si fem servir `gluLookAt` (veure Secció 8.2), mourem l'observador al llarg de l'eix OBS–VRP. ►Penseu com fer-ho si càmera està posicionada amb angles d'Euler.

►Proveu d'implementar el zoom canviant l'óptica de la càmera, és a dir, modificant el paràmetre `fovy` de la crida a `gluPerspective`, o els paràmetres de la `glOrtho` si esteu fent servir una càmera axonomètrica. Per fer-ho, tingueu en compte el desplaçament del ratolí en "Y" quan teniu el "shift" premut. ►Aquest, per cert, és l'únic mecanisme de zoom en cas de càmeres axonomètriques. Veieu per què?. ►Per les càmeres perspectives, veieu alguna diferència entre el zoom obtingut movent la càmera i l'obtingut modificant `fovy`? Sabeu explicar el perquè?.

7 Nova escena

Per a les funcionalitats següents que heu de programar i provar, us proposem modificar l'escena de la vostra aplicació. Us indicarem els propers dies l'especificació d'aquesta escena.

►Abans de continuar, verifiqueu que us funcionen totes les funcionalitats de l'exercici de la Secció 5. Molt probablement, haureu de recalcular l'esfera contenidora de l'escena i els paràmetres inicials de la càmera.

8 Navegació

Els mecanismes per a anar modificant la càmera interactivament sovint es denominen genèricament mecanismes de navegació per l'escena. N'hi ha de diferents, apropiats segons a la situació i al tipus d'aplicació, així com també per preferències personals dels usuaris. A continuació veurem alguns d'aquests mecanismes de navegació i la seva implementació. Trobareu d'altres – opcionals i avançants per aquest curs– en la Secció 10. **En la discussió següent suposem sempre una càmera perspectiva.**

8.1 Inspect

El mode "inspecció" més simple correspon (possiblement amb petites variants) al que heu estat implementant en els apartats anteriors modificant els angles d'Euler. L'escena es veu com un objecte d'interès que volem inspeccionar, mirant-lo des de direccions diferents, i acostant-nos als detalls rellevants (és com si el sostinguéssim a la mà i el moguéssim per a estudiar-lo). Aquest mode, per tant, ja l'hem implementat.

Una variant d'aquest mètode que permet una inspecció, a cops, més intuitiva, sobretot per a la inspecció d'un objecte, el trobareu a la Secció 10.2

8.2 Walk

En aquest cas, hem d'imaginar que la càmera la porta una persona caminant, o recorrent l'escena en un vehicle o, en general, un objecte que es desplaça per l'escena sempre a la mateixa alçada. El programa actualitza constantment la posició de la càmera (fent-la avançar en la direcció del moviment) i l'usuari controla a través del teclat i/o del ratolí la velocitat, així com el seu possible gir cap a la dreta o cap a l'esquerra. En la vostra aplicació, l'element mòbil serrà el `legoman`.

Per a dur a terme aquesta opció, probablement, us serà útil definir la càmera amb la crida a `gluLookAt`. ►Busqueu la plana del manual corresponent per a veure els seus paràmetres, que designen la posició de l'observador (la càmera), un punt on l'observador centra la seva mirada (anomenat VRP, acrònim de l'anglès "view reference point"), i un vector *up* que designa una direcció que l'observador veuria com vertical.

Igual que en el posicionament de la càmera amb angles d'Euler, aquesta cal definir-la (crida a `gluLookAt`) fora de la funció de refresc i cal que fem el `LoadIdentity` just abans de la crida a `gluLookAt` -tenim activa la pila de ModelView-:

```
1 glLoadIdentity();
2 gluLookAt(...);
```

De fet la crida a `gluLookAt` pot substituir les crides per a posicionar la càmera amb angles d'Euler. Podeu aprofitar per a entendre tots els paràmetres d'aquesta instrucció, en particular el vector `up`.

►Afegiu una tecla (per exemple la de la lletra `'w'`) que en prémer-la, faci que la càmera -perspectiva- es situi a sobre del Legoman. A partir d'ara, la càmera sempre ha de mirar en la direcció del moviment del Legoman. Aquesta direcció s'ha de poder modificar, tot girant el `legoman` cap a la seva dreta/esquerra quan es mou el ratolí cap a la dreta/esquerra. La tecla `'a'` ha de permetre avançar al legoman en la direcció de moviment. Inicialment aquesta direcció serà la $(0, 0, -1)$. Les tecles `'V'` i `'v'` han de permete, respectivament incrementar/decrementar la velocitat d'avançament.

►Podeu aconseguir una versió més avançada del "walk" si desacobleu la direcció de visió de la d'avançament (és a dir li deixem caminar mirant lleugerament sempre cap a un costat, per exemple). Aquest nou "walk" s'hauria d'activar quan premeu la tecla `'t'` de `tafaner`

9 Aplicació a lliurar

Cal que al començament de la primera sessió de laboratori del bloc 4, lliureu una aplicació amb les següents funcionalitats:

- En 'iniciar l'aplicació mostri una escena indicada en la Secció 7 des d'una posició arbitrària (sense retallar), amb una càmera perspectiva i aprofitant la grandària del viewport (definit com tota la finestra gràfica).
- Permeti communtar entre les càmeres perspectiva i axonomètrica. En cap cas ha d'haver-hi deformació si es fa un "resize".
- Permeti fer zoom modificant l'òptica de les càmeres.
- Permeti inspeccionar l'escena modificant els angles d'Euler.
- Permeti inspeccionar l'escena amb les dos tècniques de "walk".
- Permeti fer un "reset" per tornar a la visualització de l'escena des de la càmera inicial.
- Hi hagi un help que mostri com activar les diferents funcionalitats.

10 Per apredre més...tot és opcional

10.1 Pan

El "pan" correspon a una translació de la càmera en una direcció **perpendicular** a l'eix de visió. ►Això es pot implementar movent efectivament la càmera en aquella direcció.

Per a obtenir les direccions en què us hauríeu de moure en la vostra escena per a aconseguir un desplaçament horitzontal o vertical de la càmera (un desplaçament horitzontal o vertical de la vista), podeu recórrer a la funció `glGetDoublev(GL_MODELVIEW_MATRIX,...)`, i fer servir per a aquestes direccions les primeres dues files (ignorant la quarta component); alerta però, perquè OpenGL guarda les matrius per columnes i no per files.

10.2 Modificació interactiva en coordenades d'ull

Tanmateix, el girar l'escena a inspeccionar amb angles d'Euler no dóna girs intuïtius (en alguns casos). Vegem ara com millorar-ho.

El problema prové del fet que les crides OpenGL que estem fent servir per a composar transformacions geomètriques es multipliquen per la transformació actual, posant-les a la dreta, que és també el costat per on es multipliquen els punts (recordeu que el producte de matrius no és commutatiu). Per tant si afegim una rotació al voltant de, diguem, l'eix y, aquesta serà la primera cosa que el punt "vegi" en la seva transformació: és a dir, estarem girant al voltant de l'eix y de l'escena. Però l'usuari pot pensar que mou el ratolí pensant en les coordenades ja transformades que veu a pantalla. Per a poder fer que el moviment acompanyi realment els moviments del ratolí des del punt de vista de l'usuari, necessitaríem poder multiplicar la rotació en qüestió **per l'esquerra** de la transformació de càmera actual. Per a fer-ho, podem llegir l'estat de la transformació actual amb un `glGet`, i tornar a composar les transformacions:

```
1 GLdouble m[16];
2 glGetDoublev(GL_MODELVIEW_MATRIX, m);
3 glLoadIdentity();
4 glRotated(angle, 0.0, 1.0, 0.0);
5 glMultMatrixd(m);
```

La segona línia d'aquest codi llegeix la transformació actual en una variable local m, de manera que podem reinicialitzar la transformació a la identitat, composar-hi la rotació (en aquest exemple al voltant de l'eix y), i finalment composar-hi la transformació que teníem abans (a la línia 5) multiplicant-la per la dreta per la que hi ha (que és la rotació!). En total hem aconseguit que en el producte de transformacions la rotació estigués al davant.

►Prova-ho i compara amb l'efecte de simplement multiplicar la rotació per la dreta amb una crida a `glRotate` com fèiem abans. Us recomanem fer dues aplicacions: una amb el nou codi i altre posant la rotació per la dreta, i exercutar-les en paral·lel per a veure les diferències.

►També ara podeu integrar aquesta millora a un posicionament de la càmera efectuat per qualsevol dels mecanismes (gluLookAt() o angles d'Euler). Per això, posicioneu d'antuvi la càmera a una posició detetrminada, però a partir d'allí, modifiqueu la seva posició concatenant transformacions per l'esquerra (via `glGetDoublev...`). El mètode de posicionament "absolut" tan sols s'ha de tornar a cridar si l'usuari vol fer un *reset* de la càmera (cosa per la qual és bo proporcionar-li algun mecanisme. ►Afegiu alguna tecla que en ésser premuda retorni la càmera a una posició coneguda). Aquests canvis han de succeir en el bloc de codi de posicionament de càmera, no en el moment de refrescar un frame. D'aquesta manera preserveu el desacoblament de la gestió de la càmera i del model. A més, donat que el mètode de composar els girs per l'esquerra és incremental, es facilita tot plegat, ja que tan sols s'executa aquest codi quan es vol modificar la posició de la càmera.

►Tornant a la inspecció del model, feu servir aquesta tècnica per a proporcionar una millor rotació interactiva del model (quin eix de rotació cal fer servir quan l'usuari mou el ratolí de la posició (x_0, y_0) a la posició (x_1, y_1)? Pensa-ho primer i prova després d'implementar-ho).

10.3 Fly-through

En aquest cas, hem d'imaginar que estem a un avió que el sobrevola. El programa actualitza constantment la posició de l'avió (fent-lo avançar, tal com fèieu a un dels modes de zoom), i l'usuari controla a través del teclat i del ratolí la velocitat, així com els girs. Per donar realisme, els girs a dreta i esquerra s'haurien d'acompanyar d'un gir al voltant de l'eix de visió (els avions no giren perfectament horitzontals. Aquest gir al llarg de l'eix de visió s'anomena "roll"). També pot controlar l'alçada del vol, canviant la inclinació de l'avió (donada pel gir en x en angles d'Euler, anomenat en aquest context "pitch").

►Mireu d'implementar una navegació amb aquest paradigma per a la vostra aplicació. Aprofitareu el *callback* de `glutIdleFunc` per a cada *frame* actualitzar la posició de la càmera, en comptes (o a més) de modificar l'escena.

Fixeu-vos que el que veiem, és el que veuria una càmera fixada al nas del nostre avió imaginari, no el pilot. El pilot, en realitat, té més graus de llibertat, perquè pot triar no mirar recte cap endavant. ►Com implementaríeu un model més detallat que a més de moure's al llarg de la trajectòria de l'avió com abans, permeti al pilot "moure el cap"?