

Informatica Grafica

Introduzione

a

OpenGL

parte 2

Gestione degli eventi

Come accennato in precedenza, OpenGL in senso stretto non possiede un meccanismo di gestione degli eventi, dato che tale compito è solitamente svolto dal sistema operativo per il quale viene sviluppata l'applicazione ed ogni sistema operativo ha il proprio modo di gestire gli eventi.

OpenGL possiede però una libreria che fornisce un minimo livello di astrazione rispetto al sistema operativo in modo tale che si possano gestire gli eventi più comuni => glut

PRO:

- Essendo parte di OpenGL, è indipendente dal s.o.
- E' semplice da utilizzare
- Richiede la stesura (relativamente) di poco codice

CONTRO:

- La gestione degli eventi non è completa come quella del s.o
- Non è abbastanza completo per costruire una GUI (graphical user interface) avanzata

Gestione degli eventi (2)

Eventi gestiti da glut:

- Eventi generati da window system
- Eventi generati da mouse
- Eventi generati da tastiera
- Eventi generati da altri dispositivi

Gestione tramite **callback**

- Si definisce una funzione che gestisca l'evento
- Si associa la funzione all'evento con una chiamata a glut
- Ogni volta che si verifica l'evento la funzione viene eseguita

Reshape & Expose

Reshape

è l'evento che si verifica ogni volta che la finestra viene ridimensionata (viene generato anche quando la finestra viene creata per la prima volta)

Solitamente la funzione associata ha il compito di settare i parametri di viewing (viewing volume, viewport...)

`void glutReshapeFunc(void(f*)(int,int))`

i due argomenti della funzione puntata da *f* sono le dimensioni della finestra

Expose

Permette di gestire l'evento “finestra scoperta”, che si verifica ogni volta che la finestra diventa visibile. Tale evento è generato anche se la finestra viene ridimensionata.

La funzione associata di solito ha il compito di ridisegnare la scena

`void glutDisplayFunc(void(*f)(void))`

Reshape & Expose (2)

Esempio 1:

```
#include <stdio.h>
#include <GL/glut.h>

void reshape(int w,int h){
printf( "event: reshaped to %d x
      %d\n", w, h );
}

void expose(void){
printf( "event: exposed\n" );
}

void main( int argc, char**
          argv ){
glutInitDisplayMode(
    GLUT_RGB);
glutInitWindowPosition(300,
    100 );
glutInitWindowSize(150,150);
glutCreateWindow(argv[0]);
glutReshapeFunc(reshape);
glutDisplayFunc(expose);
glutMainLoop();
}
```

Keyboard

Gestione caratteri ASCII

`void glutKeyboardFunc( void(*f)`

`(unsigned char k, int x, int y) )`

- viene chiamata alla pressione del tasto k di codice ASCII k (ricordare che char è un tipo intero...)
- x, y sono le coordinate del mouse al momento del verificarsi dell'evento (relativo all'angolo in alto a sinistra)

Gestione di caratteri “speciali”

`void glutSpecialFunc( void(*f)(int k, int x, int y) )`

come nel caso precedente, ma k può valere:

GLUT_KEY_F1, ..., GLUT_KEY_F12

GLUT_KEY_LEFT, GLUT_KEY_UP

GLUT_KEY_RIGHT, GLUT_KEY_DOWN

GLUT_KEY_HOME, GLUT_KEY_PAGE_UP

GLUT_KEY_END

GLUT_KEY_PAGE_DOWN

GLUT_KEY_INSERT

Keyboard (esempio)

```
void processNormalKeys(unsigned char
    key, int x, int y)
{
    if (key == 27)
        exit(0);
}

void main(int argc, char **argv) {
    ...
    glutKeyboardFunc(processNormalKeys);
    ...
    glutMainLoop();
}
```

Mouse

Gestione del mouse:

```
void glutMouseFunc(void(*f)(int b, int
    s, int x, int y) )
```

valori possibili per b :

GLUT_LEFT_BUTTON

GLUT_MIDDLE_BUTTON

GLUT_RIGHT_BUTTON

valori possibili per s :

GLUT_DOWN GLUT_UP

x e y sono le coordinate del mouse al verificarsi dell'evento

Esempio 2

```
void mouse( int but, int
            state, int x, int y ){
char *b, *left="left",
        *middle="middle",
        *right="right";

char *s, *press="pressed",
        *rel="released";

switch( but ) {

case GLUT_LEFT_BUTTON :
    b=left; break;

case GLUT_MIDDLE_BUTTON:
    b=middle; break;

case GLUT_RIGHT_BUTTON :
    b=right; break; }
```

```
switch( state ) {

case GLUT_DOWN : s=press;
    break;

case GLUT_UP : s=rel;
    break; }

printf( "%s mouse button %s
        in %d, %d\n",
        b, s, x, y );
}
```

Evento “mouse move”

La libreria glut permette di gestire l'evento “spostamento del mouse”, associando ad esso una funzione del tipo

`void mouse(int x, int y)`

in cui *x* e *y* sono le coordinate del mouse (in coordinate finestra).

Le chiamate glut che permettono di associare tale funzione all'evento “spostamento del mouse” sono le seguenti:

`void glutPassiveMotionFunc( void (*f)(int x, int y))` per movimento “passivo” che si verifica quando il mouse viene mosso sulla finestra senza pressione di pulsanti.

`void glutMotionFunc(void (*func) (int x,int y))` per movimento “attivo” che si verifica quando il mouse viene mosso sulla finestra con un pulsante premuto.

Nota:

gli argomenti *x* e *y* corrispondono alla posizione del mouse rispetto all' “origine” della finestra, che è posta nell'angolo in alto a sinistra

Idle & Timer

Idle:

permette di fare del “background processing”

void **glutIdleFunc**(*void(*f)(void)*) utile per generare delle animazioni

Timer:

viene eseguita allo scadere di un tempo prefissato (“lower bound”, intervallo minimo)

void **glutTimerFunc**(*int ms*,
*void(*f)(int value)*, *int value*)

ms= millisecondi value= valore definito dall'utente e passato a f

Se una callback deve ridisegnare la scena (es. idle callback per animazione) lo deve fare chiamando la funzione **glutPostRedisplay()**

Menu

glut permette inoltre di predisporre dei menu di pop up alla pressione di un tasto del mouse:

int glutCreateMenu(*void (*f)(int)*) crea un nuovo menu, il menu creato diventa il menu “corrente”. Un menu puo' contenere “entry” elementari oppure dei “sotto-menu”. *f* è la funzione di callback che viene chiamata quando viene selezionata una entry del menu

void glutAddSubMenu(*char**, *int m*) Aggiunge un sotto-menu al menu “corrente”, il primo parametro è la stringa che compare nel sotto-menu, *m* e' l'identificatore del sottomenu (valore di ritorno di **glutCreateMenu**)

void glutAttachMenu(*GLenum button*) associa la pressione di un tasto del mouse alla visualizzazione del menu corrente

button può valere ad es.

GLUT_RIGHT_BUTTON

GLUT_LEFT_BUTTON

void glutAddMenuEntry(*char *name*, *int value*); *name* e' la stringa che compare nel menu; *value* e' il valore ritornato alla callback quando il menu viene scelto

Esempio 3 (pop up menu)

```
#include <math.h>
```

```
// Define a constant for the value of PI
```

```
#define GL_PI 3.1415f
```

```
// Rotation amounts
```

```
static GLfloat xRot = 0.0f;
```

```
static GLfloat yRot = 0.0f;
```

```
// Flags for effects
```

```
#define MODE_SOLID 0
```

```
#define MODE_LINE 1
```

```
#define MODE_POINT 2
```

```
int iMode = MODE_SOLID;
```

```
Glboolean bEdgeFlag = true;
```

```
void ProcessMenu(int value){
```

```
switch(value){
```

```
case 1:
```

```
iMode = MODE_SOLID;
```

```
break;
```

```
case 2:
```

```
iMode = MODE_LINE;
```

```
break;
```

```
case 3:
```

```
iMode = MODE_POINT;
```

```
break;
```

Esempio 3 (continua)

```
case 4:  
    bEdgeFlag = true;  
    break;  
case 5:  
    default:  
        bEdgeFlag = false;  
        break;  
}  
glutPostRedisplay();  
}
```

```
void RenderScene(void){  
    glClear(GL_COLOR_BUFFER_BIT);  
    if(iMode == MODE_LINE)  
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  
    if(iMode == MODE_POINT)  
        glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);  
    if(iMode == MODE_SOLID)  
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);  
    glPushMatrix();  
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);  
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);  
    glBegin(GL_TRIANGLES);
```

Esempio 3 (continua)

```
glEdgeFlag(bEdgeFlag);
```

```
glVertex2f(-0.2f, 0.0f);
```

```
glEdgeFlag(true);
```

```
glVertex2f(0.2f, 0.0f);
```

```
glVertex2f(0.0f, 0.4f);
```

```
glVertex2f(-0.2f,0.0f);
```

```
glVertex2f(-0.6f,-0.2f);
```

```
glEdgeFlag(bEdgeFlag);
```

```
glVertex2f(-0.2f,-0.4f);
```

```
glEdgeFlag(true);
```

```
glVertex2f(-0.2f,-0.4f);
```

```
glVertex2f(0.0f, -0.8f);
```

```
glEdgeFlag(bEdgeFlag);
```

```
glVertex2f(0.2f, -0.4f);
```

```
glEdgeFlag(true);
```

```
glVertex2f(0.2f, -0.4f);
```

```
glVertex2f(0.6f, -0.2f);
```

```
glEdgeFlag(bEdgeFlag);
```

```
glVertex2f(0.2f, 0.0f);
```

```
glEdgeFlag(true);
```

Esempio 3 (continua)

```
glEdgeFlag(bEdgeFlag);
```

```
glVertex2f(-0.2f, 0.0f);
```

```
glVertex2f(-0.2f,-0.4f);
```

```
glVertex2f(0.2f, 0.0f);
```

```
glVertex2f(-0.2f,-0.4f);
```

```
glVertex2f(0.2f, -0.4f);
```

```
glVertex2f(0.2f, 0.0f);
```

```
glEdgeFlag(true);
```

```
glEnd();
```

```
glPopMatrix();
```

```
glFlush();}
```

```
void SetupRC(){
```

```
glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
```

```
glColor3f(0.0f, 1.0f, 0.0f);
```

```
}
```

```
void SpecialKeys(int key, int x, int y)
```

```
{
```

```
if(key == GLUT_KEY_UP)
```

```
xRot-= 5.0f;
```

```
if(key == GLUT_KEY_DOWN)
```

```
xRot += 5.0f;
```

```
if(key == GLUT_KEY_RIGHT)
```

```
yRot += 5.0f;
```

Esempio 3 (continua)

```
If(key == GLUT_KEY_LEFT)
```

```
yRot -= 5.0f;
```

```
if(key > 356.0f)
```

```
xRot = 0.0f;
```

```
if(key < -1.0f)
```

```
xRot = 355.0f;
```

```
if(key > 356.0f)
```

```
yRot = 0.0f;
```

```
if(key < -1.0f)
```

```
yRot = 355.0f;
```

```
glutPostRedisplay();
```

```
}
```

```
int main(int argc, char* argv[]){
```

```
int nModeMenu,nEdgeMenu,nMainMenu;
```

```
glutInit(&argc, argv);
```

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
```

```
glutCreateWindow("Solid and Outlined Star");
```

```
nModeMenu = glutCreateMenu(ProcessMenu);
```

```
glutAddMenuEntry("Solid",1);
```

```
glutAddMenuEntry("Outline",2);
```

```
glutAddMenuEntry("Points",3);
```

```
nEdgeMenu = glutCreateMenu(ProcessMenu);
```

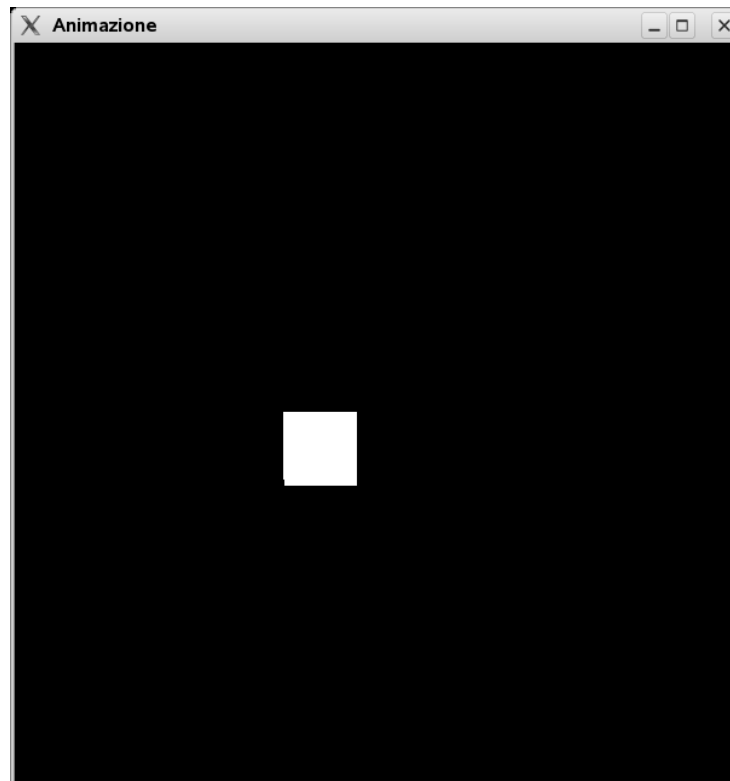
```
GlutAddMenuEntry("On",4);
```

Esempio 3 (continua)

```
GlutAddMenuEntry("Off",5);  
  
nMainMenu = glutCreateMenu(ProcessMenu);  
  
glutAddSubMenu("Mode", nModeMenu);  
  
glutAddSubMenu("Edges", nEdgeMenu);  
  
glutAttachMenu(GLUT_RIGHT_BUTTON);  
  
glutSpecialFunc(SpecialKeys);  
  
glutDisplayFunc(RenderScene);  
  
SetupRC();  
  
glutMainLoop();  
  
return 0;  
  
}
```

Esercizio: quadrato che trasla

Realizzare una animazione di un quadrato che oscilla lungo l'asse x, utilizzando la funzione `glutIdleFunc(void(*f)(void))` e `glRectf()`



Double buffering

Caratteristica di OpenGL che “nasconde” la fase di rendering vera e propria e mostra il risultato completo del FrameBuffer. Prevede che il disegno effettivo avvenga in un “back buffer”, mentre il “front buffer” contiene il risultato finito. Quando è terminato il disegno nel back buffer, i due buffer vengono scambiati, evitando il tipico sfarfallio delle immagini che vengono ridisegnate di frequente.

Per abilitarlo bisogna chiamare

glutInitDisplayMode(... | GLUT_DOUBLE | ...)

e una volta disegnata l'intera scena chiamare

void glutSwapBuffers()

al posto di glFlush() o glFinish();

L'utilizzo del double buffering è richiesto soprattutto nelle animazioni e in genere quando si necessita di ridisegnare la scena molto di frequente

Disegnare elementi complessi

- Solitamente per ciascuna parte che compone l'oggetto nel suo insieme si distingue la GEOMETRIA dalla CONNETTIVITA', caratteristiche che insieme formano l'oggetto nel modo in cui lo abbiamo modellato.
- GEOMETRIA = posizione dei vertici che formano i poligoni dell'oggetto
- CONNETTIVITA' = composizione dei poligoni in base ai vertici specificati separatamente

VANTAGGIO

è possibile variare dinamicamente la geometria dell'oggetto in modo efficiente,
senza cambiare il codice che disegna effettivamente l'oggetto

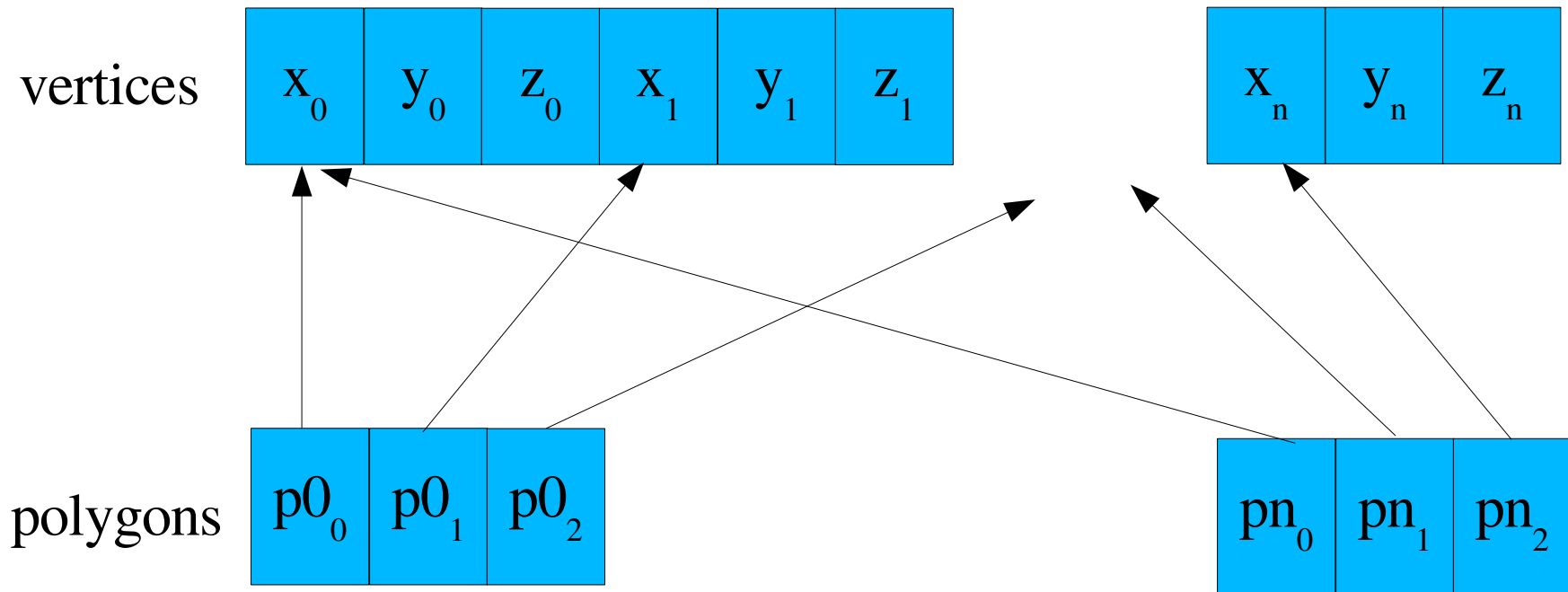
Esempio di struttura topologica complessa

```
typedef struct _act{  
    /*dimensione (dinamica) dell'array  
       di vertici*/  
    int vertexnum;  
    /*dimensione (dinamica) dell'array  
       di poligoni*/  
    int polynum;  
    GLfloat *vertices;  
    GLint *polygons;  
    float matrix[16];  
} actor;
```

Tipica funzione di inizializzazione:

```
void init(int vrtx,int poly,actor act){  
    act.vertices= new GLfloat[3*vrtx];  
    act.polygons=new GLint[3*poly];  
    /*in questo caso si dà per scontato che  
       i vertici siano 3D e che i poligoni  
       siano solo triangoli, ma non è  
       necessariamente sempre così*/  
}
```

Geometria + Connettività



Disegnare strutture topologiche

Possibile funzione per il disegno del singolo elemento:

```
void DrawActor(actor act){  
    for(int i=0; i < act.polynum; i+=3){  
        glBegin(GL_TRIANGLES);  
        glVertex3f(act.vertices[act.polygons[i]],act.vertices[act.polygons[i]+1],act.vertices[act.polygons[i]+2]);  
        glVertex3f(act.vertices[act.polygons[i+1]],act.vertices[act.polygons[i+1]+1],act.vertices[act.polygons[i+1]+2]);  
        glVertex3f(act.vertices[act.polygons[i+2]],act.vertices[act.polygons[i+2]+1],act.vertices[act.polygons[i+2]+2]);  
        glEnd();  
    }  
}
```

Viewing

Nella descrizione della scena occorre specificare:

- Primitive
- Parametri di viewing
- Proprietà dei materiali (in relazione alla luce)
- Modello di illuminazione

I parametri di viewing sono quelli che permettono di posizionare gli oggetti nella scena e di scegliere un punto di osservazione dal quale visualizzare la scena stessa. Prima di iniziare a parlare di viewing è necessario definire alcuni termini:

object coordinates, eye coordinates, normalized device coordinates, window coordinates (coordinate finestra, dopo trasformazione window to viewport).

Coordinate

- Object coordinates:

sistema di riferimento solidale con l'oggetto e utilizzato principalmente per la sua costruzione

- Eye coordinates:

sistema di riferimento “virtuale” e fisso, in riferimento al quale vengono descritte posizione e orientazione di tutti gli oggetti. Si chiama così perchè è solidale con l'osservatore

- 3D Normalized device coordinates:

coordinate normalizzate (volume di vista canonico)

- 3D window coordinates:

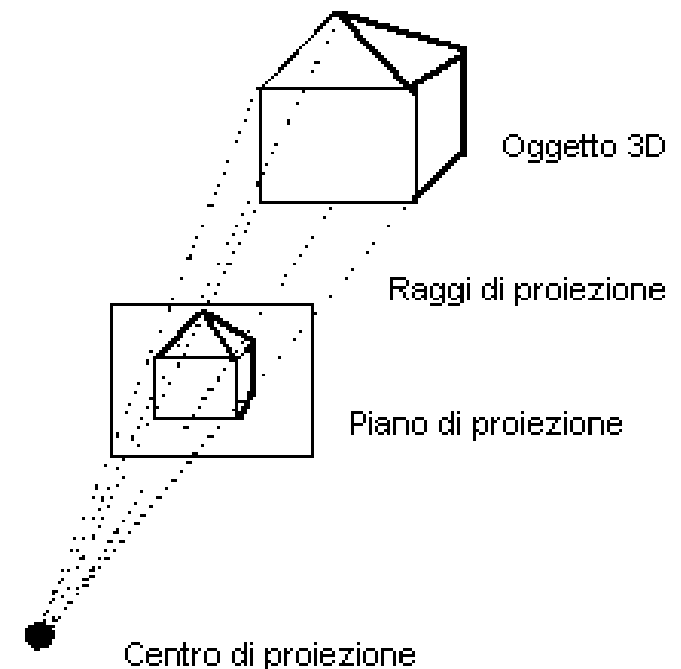
coordinate relative alla finestra

Viewing

I fondamenti concettuali della computer graphics sono basati su modelli di sistemi ottici. Il processo di formazione delle immagini generate dal computer viene assimilato al processo di formazione di una immagine da parte di un sistema ottico, quale ad esempio una macchina fotografica.

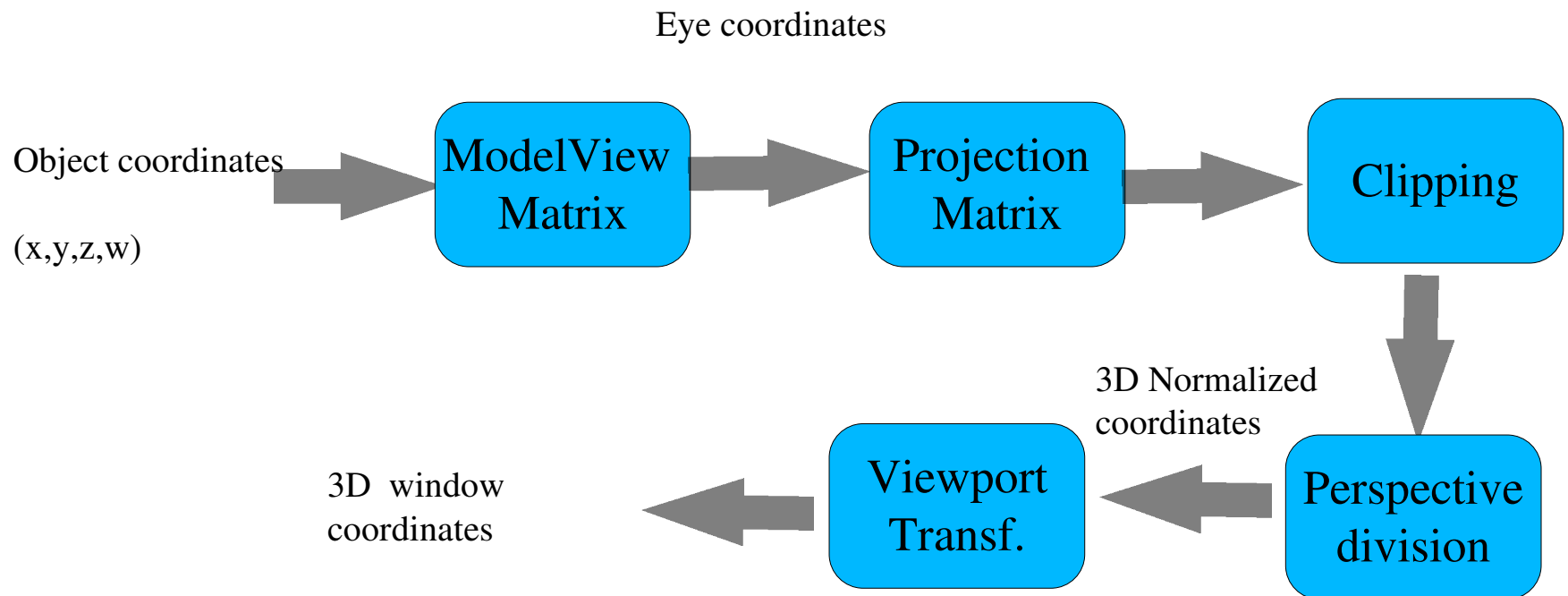
Passi nella formazione di una foto:

- 1) disporre gli oggetti nella scena da fotografare
(modeling transformation)
- 2) disporre la macchina e puntarla verso la scena
(viewing transformation)
- 3) scegliere una lente o aggiustare lo zoom
(projection transformation)
- 4) scegliere la dimensione della foto
(viewport transformation)



Viewing (2)

Da quando si specificano i vertici delle primitive a quando si vede l'immagine su schermo vengono effettuati alcuni passaggi intermedi:



ModelView Matrix

E' la matrice che trasforma le object coordinates in eye coordinates e viene modificata ad ogni trasformazione geometrica della scena. E' usata inoltre per definire il punto di vista, dal momento che le trasformazioni geometriche che determinano il punto di vista sono le medesime usate per modellare la scena, cioè, per esempio, spostare l'osservatore nel verso negativo dell'asse z di una certa quantità è lo stesso che spostare tutta la scena in della stessa quantità nel verso positivo dell'asse z.

OpenGL possiede tre tipi di matrici, organizzate in “stack”:

- MODELVIEW (modellazione scena e posizione osservatore)
- PROJECTION (definizione viewing volume)
- TEXTURE (operazione di modellazione su texture, analogamente alla modelview)

Ogni matrice in cima allo stack può essere manipolata direttamente oppure attraverso alcune funzioni di alto livello come **glRotate**, **glTranslate**, **glScale**

ModelView Matrix (2)

OpenGL memorizza le matrici per colonna (mentre il C le memorizza per riga), quindi quando si manipolano direttamente le matrici di OpenGL si ha a che fare con un array di 16 float (matrici 4x4) organizzato nel seguente modo:

$$\begin{matrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{matrix}$$

Prima di poter modificare una matrice, occorre selezionare il suo stack di appartenenza con la funzione

void **glMatrixMode**(*GLenum s*), in cui *s* può valere:

- GL_MODELVIEW
- GL_PROJECTION
- GL_TEXTURE

Operazioni su matrici

Push

`void glPushMatrix(void)`

- una copia della matrice corrente viene posta sulla cima dello stack
- la nuova matrice diviene la matrice corrente



Pop

`void glPopMatrix(void)`

- la matrice corrente viene eliminata dalla cima dello stack
- la matrice sottostante diviene la nuova matrice corrente



Operazioni su matrici (2)

Inizializzazione

`void glLoadIdentity(void)`

sostituisce la matrice corrente con la matrice identità

Caricamento di una matrice

`void glLoadMatrixf(const GLfloat *f)`

la matrice corrente viene sostituita da f, che punta ad un'area di memoria contenente 16 float

Moltiplicare, ruotare, traslare, scalare

`void glMultMatrix(const GLfloat *m)`

la matrice corrente C viene premoltiplicata per m e il risultato salvato nella matrice corrente = Cm

`void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)`

la matrice corrente viene premoltiplicata per una matrice che rappresenta una rotazione di *angle* gradi attorno all'asse specificato dal vettore $[x, y, z]^T$

Operazioni su matrici (3)

`void glTranslatef(GLfloat x, GLfloat y, GLfloat z)`

premultiplica la matrice corrente per una che rappresenta una traslazione delle quantità x, y, z lungo i tre assi e memorizza il risultato nella matrice corrente

`void glScalef(GLfloat x, GLfloat y, GLfloat z)`

come sopra ma in questo caso l'operazione è una scalatura. Per manipolare direttamente le matrici occorre tenere presente la struttura delle matrici che si usano con OpenGL.

Una rotazione in R^3 è descritta da una matrice 3x3, in questo modo:

$$\begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{Rot}(z)$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix} \quad \text{Rot}(x)$$

$$\begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix} \quad \text{Rot}(y)$$

Operazioni su matrici (4)

Tutte le trasformazioni di viewing e modeling sono rappresentate da matrici 4x4. I comandi di trasformazione (**glMultMatrix**, **glRotatef**, **glTranslatef**, **glScalef**) moltiplicano una nuova matrice M per l'attuale matrice in cima allo stack C. I vertici delle primitive vengono moltiplicati dalla trasformazione risultante.

Esempio:

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glMultMatrixf(N); /* apply transformation N */  
glMultMatrixf(M); /* apply transformation M */  
glMultMatrixf(L); /* apply transformation L */  
glBegin(GL_POINTS);  
glVertex3f(v); /* draw transformed vertex v */  
glEnd();
```

La trasformazione complessiva è data dalla relazione $N(M(Lv))$

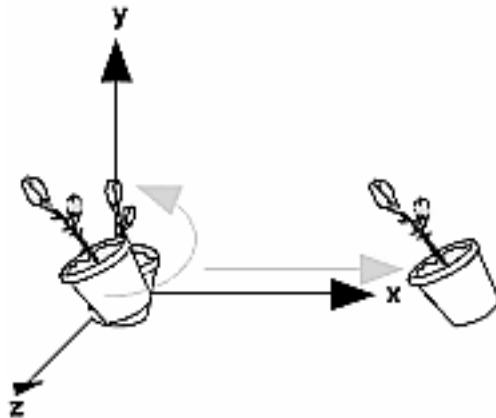
Trasformazioni composte

In generale:

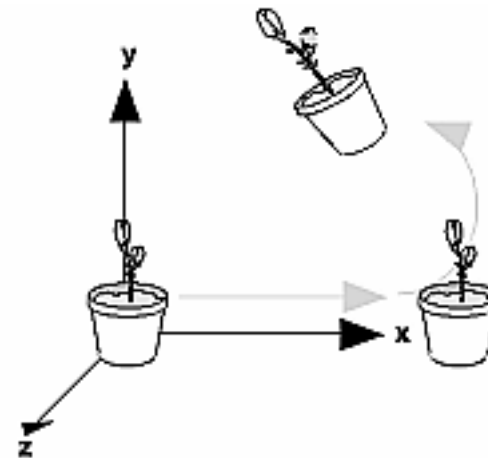
se si vuole una trasformazione di tipo “assoluto”, cioè relativa ad un unico sistema di riferimento fisso iniziale, è necessario specificare le moltiplicazioni in ordine inverso.

se invece si desiderano trasformazioni di tipo “relativo”, cioè in riferimento al sistema ottenuto dopo l'ultima trasformazione, è necessario specificare le moltiplicazioni nel loro ordine.

```
glMultMatrixf(T);  
glMultMatrixf(R);
```



Rotate then Translate



```
glMultMatrixf(R);  
glMultMatrixf(T);
```

Translate then Rotate

ESERCIZIO: Studiare nell'esempio 3 come avviene la rotazione della scena.

Orientazione degli oggetti

La trasformazione omogenea descritta dalla matrice 4x4 permette in altre parole di conoscere la posizione e l'orientazione di un qualsiasi oggetto. Memorizzare (o ricordarsi) 16 valori potrebbe in alcuni casi essere poco pratico, perciò si possono utilizzare altri metodi per rappresentare posizione e orientazione di un oggetto:

Roll Pitch Yaw

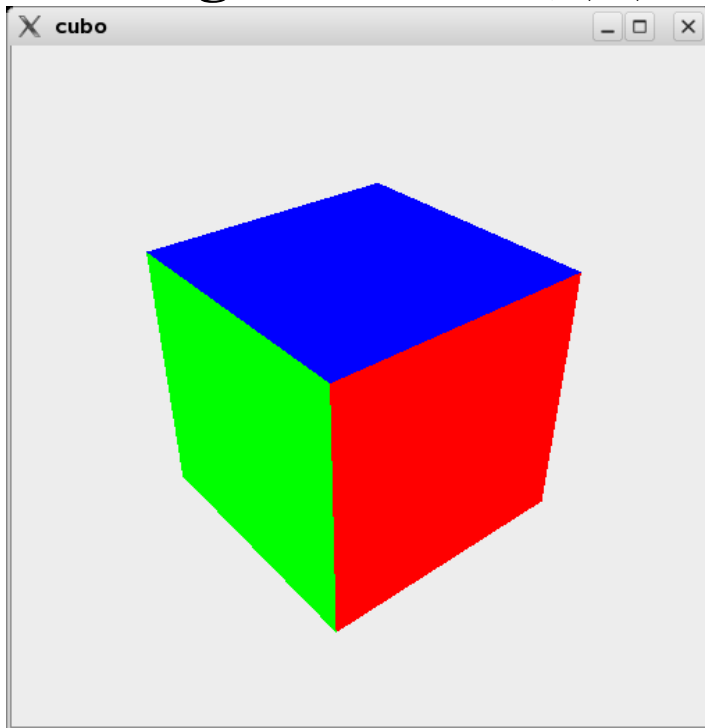
```
struct RPY{  
    GLfloat location[3];  
    GLfloat Roll; /* asse x */  
    GLfloat Pitch; /* asse y */  
    GLfloat Yaw; /* asse z*/  
}
```

si memorizzano tre angoli di cui ruotare l'oggetto attorno ai tre assi del sistema di riferimento **fisso**. La sequenza con cui moltiplicare le matrici è $\text{rot}(z, \text{Yaw}) * \text{rot}(y, \text{Pitch}) * \text{rot}(x, \text{Roll})$

Esercizio: cubo che ruota

Realizzare una animazione di un cubo (lato $l=1$) centrato nell'origine le cui facce hanno colori diversi. Quando il tasto sinistro del mouse e' premuto il cubo ruota. Lo spostamento del mouse (con il tasto premuto) lungo l'asse x causa la rotazione del cubo rispetto all'asse y, viceversa lo spostamento del mouse lungo l'asse y causa la rotazione del cubo rispetto all'asse x.

Osservazione: oltre alla rotazione è necessario inserire una traslazione del cubo (ovvero dell'osservatore) di una certa quantità rispetto all'asse z. Es: `glTranslatef(0,0,-3.0f);`



utilizzare la seguente funzione di reshape dello schermo

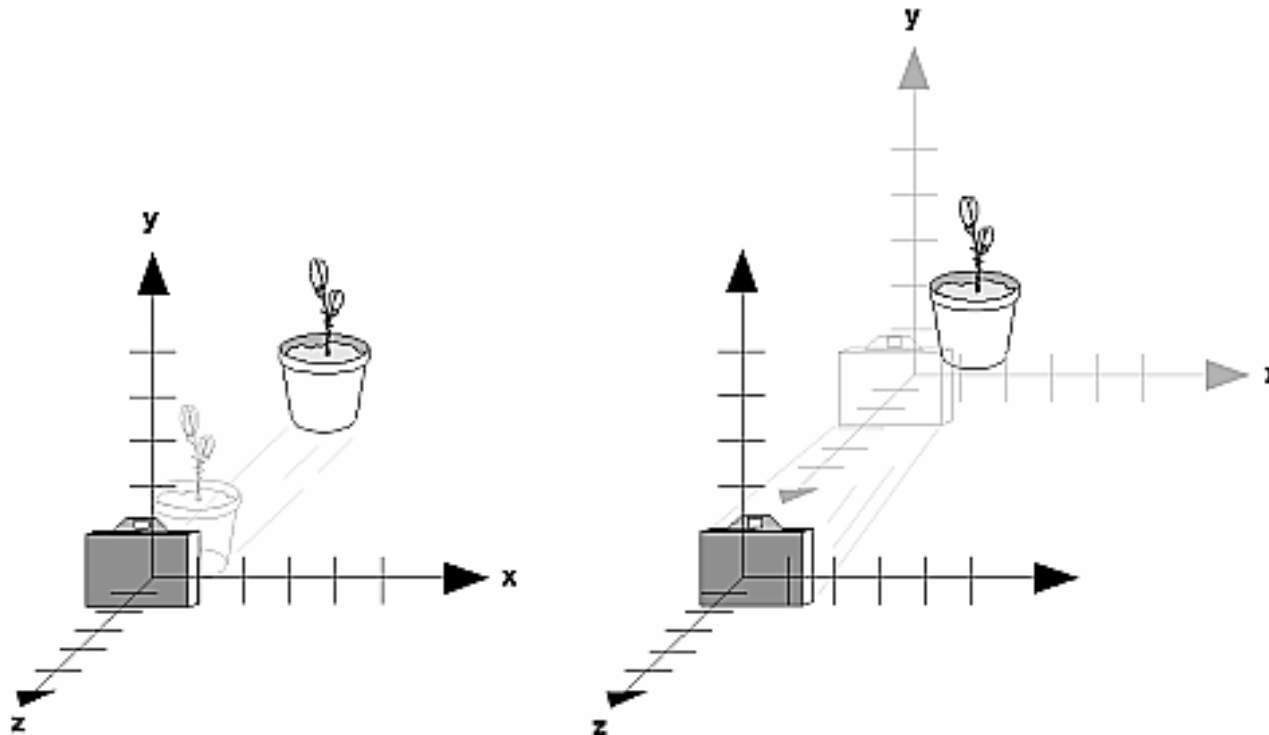
```
void resize(int w, int h)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    glViewport(0, 0, w, h);

    gluPerspective(45.0f, 1.0f * w / h, 1.0f, 100.0f);
}
```

Punto di vista

Le trasformazioni di modeling viste fino ad ora possono essere utilizzate anche per il posizionamento del punto di vista dell'osservatore, dato che muovere il punto di vista lungo una direzione equivale a muovere tutta la scena nella direzione opposta. Per default l'osservatore è posto nell'origine ed è rivolto nella direzione negativa dell'asse z (guarda all'interno dello schermo, per intenderci).



Punto di vista (2)

OpenGL (glu per la precisione) mette a disposizione una funzione che accetta una descrizione della posizione e dell'orientamento dell'osservatore e la traduce in una matrice con cui moltiplicare la modelview:

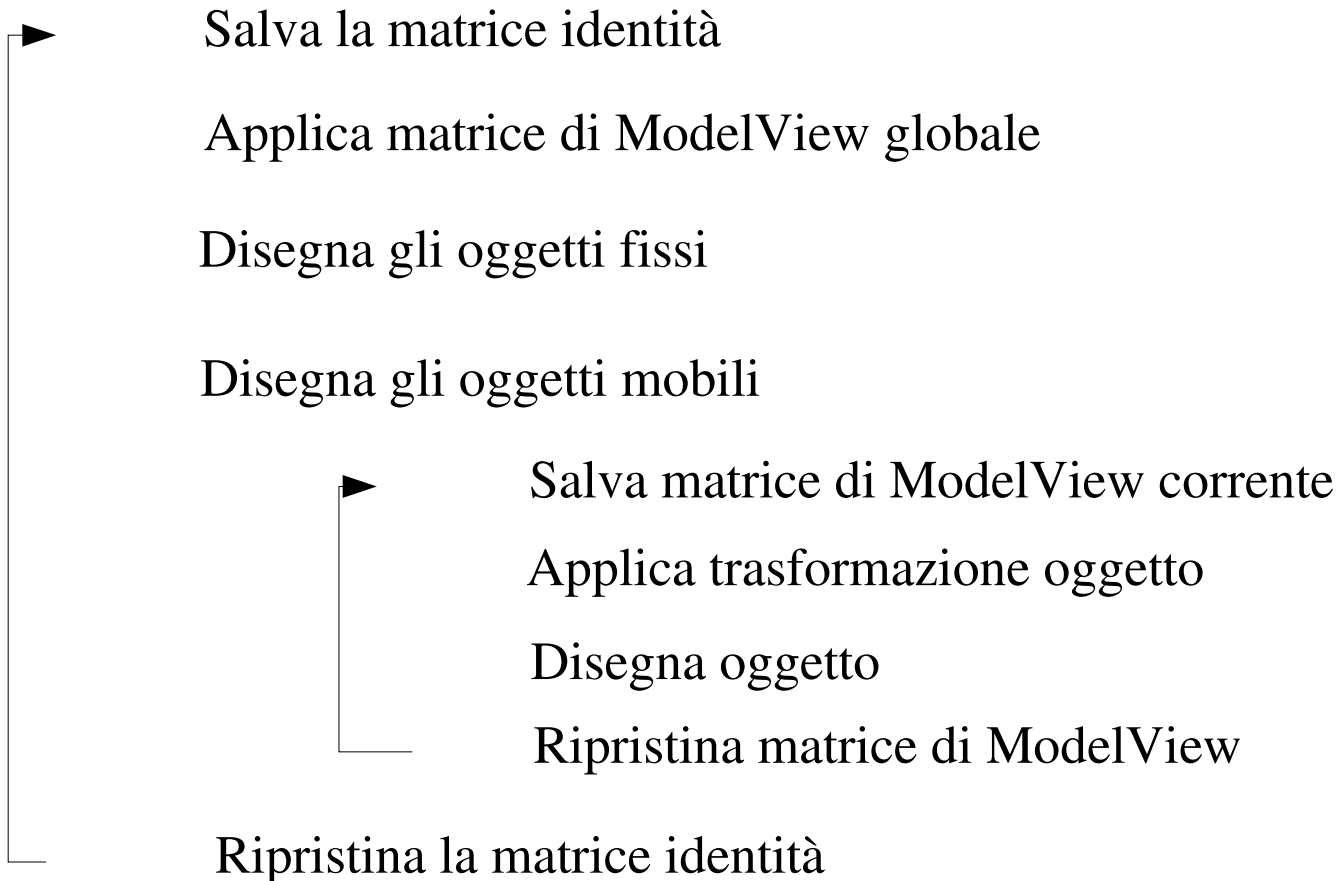
```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble  
    centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy,  
    GLdouble upz);
```

in cui eye[x|y|z] rappresentano la posizione dell'osservatore, center[x|y|z] la posizione del punto dove guardare, up[x|y|z] la direzione “alto”, cioè l'inclinazione dell'osservatore rispetto al sistema i riferimento.

Dal momento che OpenGL premoltiplica soltanto e che il posizionamento del punto di vista deve avvenire per ultimo (non voglio più avere trasformazioni successive perchè lo modificherebbero), la chiamata a gluLookAt deve avvenire prima delle trasformazioni di modelling

Rendering loop in 3D

Una tipica sequenza di operazioni da effettuare per il rendering di un ambiente tridimensionale potrebbe dunque essere il seguente:



come è possibile rappresentare ambienti complessi descritti da uno scene graph?

Finestre Multiple

OpenGL consente di gestire finestre multiple di rendering. Ad ogni finestra è associato un contesto OpenGL che deve essere inizializzato immediatamente dopo la creazione di una finestra (funzioni di callback associate ad ogni finestra). L'ultima finestra creata è impostata come “finestra corrente”.

`void glutSetWindow(int win)`

imposta win come “finestra corrente”

`int glutGetWindow(void):`

utilizzata per reperire la “finestra corrente” in uso.

`void glutDestroyWindow(int win):`

distrugge la finestra indentificata da win e il suo contesto OpenGL.

`void glutPostRedisplay(void):`

genera un evento che permette di ridisegnare il contenuto della “finestra corrente”.

(ESEMPIO 4)