

Corso di
Informatica Grafica

Introduzione
a
OpenGL

parte 3

Proiezioni

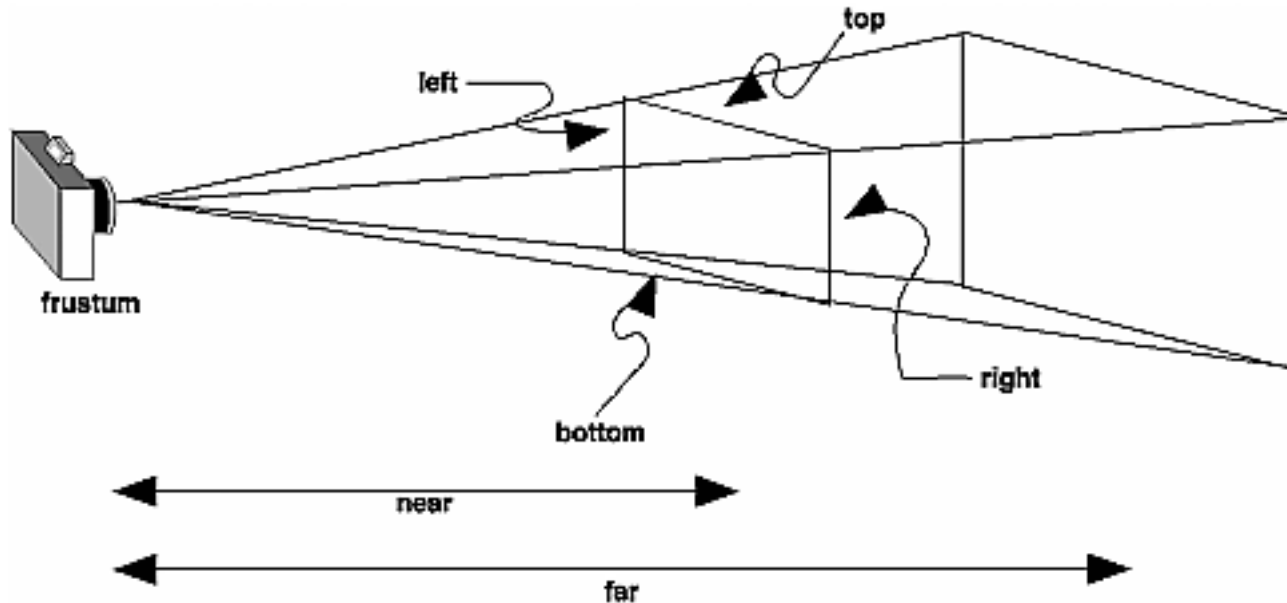
Una volta che è stato definito il contenuto della scena, OpenGL deve rappresentare questo contenuto sullo schermo (superficie bidimensionale). Per questa operazione sono disponibili due modalità:

- Proiezione prospettica
- Proiezione ortogonale (detta anche parallela)

La prima produce un risultato realistico caratterizzato dall'effetto dell'apparente diminuzione delle dimensioni con l'aumentare della distanza dall'osservatore, mentre la seconda si adatta meglio alla rappresentazione di oggetti per i quali non serve o non è gradito l'effetto di rimpicciolimento con la distanza, come nel caso della grafica 2D e/o molte applicazioni CAD in cui è importante confrontare le misure.

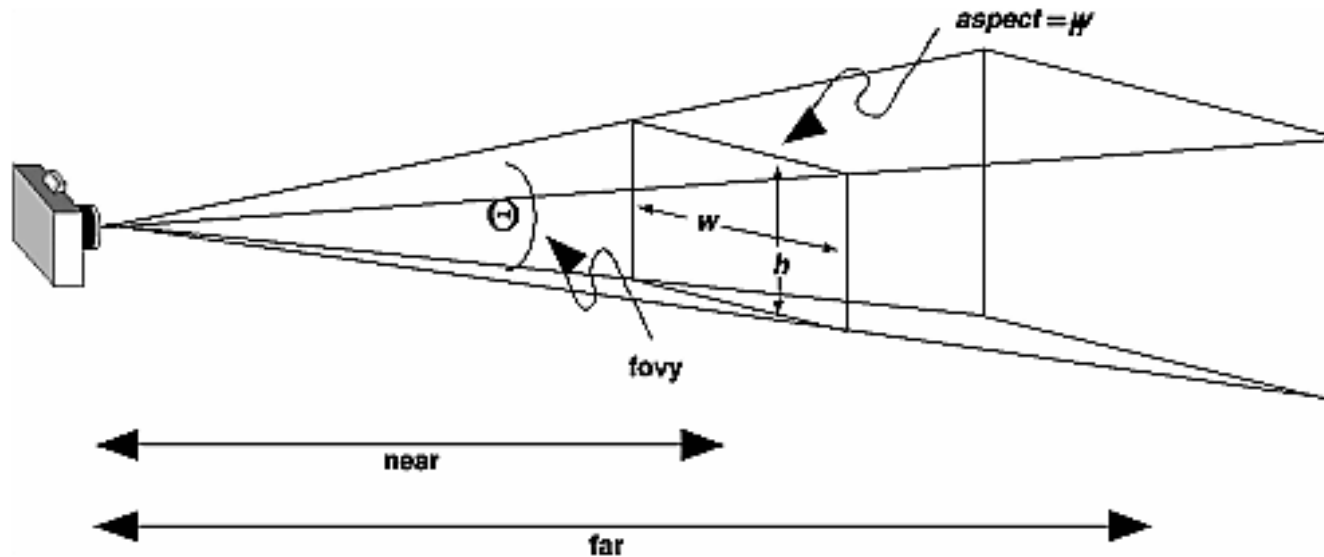
Il viewing volume nel primo caso è un tronco di piramide, mentre nel secondo è un parallelepipedo

Proiezione prospettica



La proiezione di tipo prospettico si abilita con
`void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,`
`GLdouble top, GLdouble near, GLdouble far)`
i parametri `near` e `far` indicano la distanza dei due clipping planes dal punto di osservazione e devono essere sempre valori positivi. E' possibile creare frustum asimmetrici.

Proiezione prospettica (2)

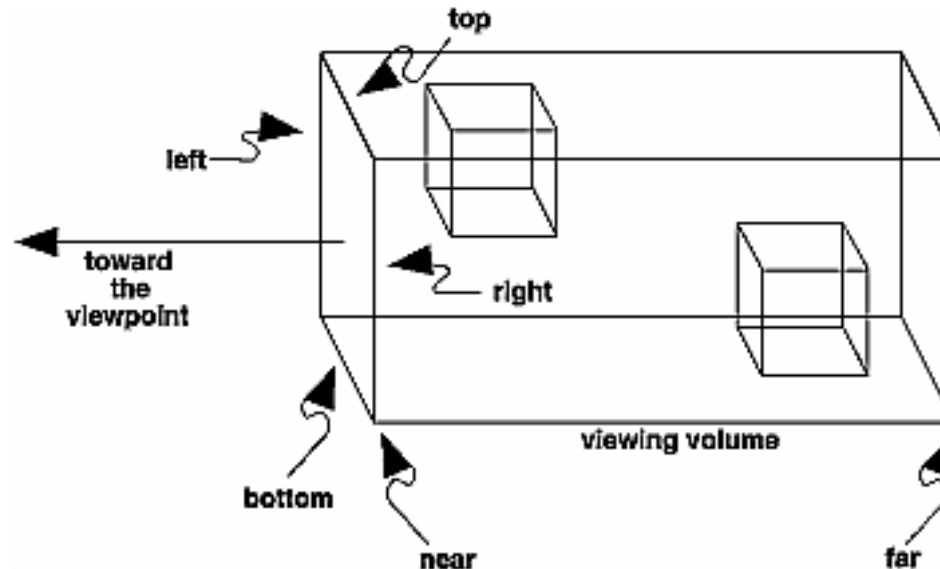


È possibile definire una proiezione prospettica in modo differente, con una funzione messa a disposizione da glu, maggiormente intuitiva da utilizzare rispetto a glFrustum, ma che ottiene gli stessi risultati:

```
void gluPerspective(GLdouble fovy, GLdouble aspect,  
                    GLdouble near, GLdouble far)
```

in cui fovy è l'angolo sotto il quale viene vista la scena nel piano xz ed è compreso tra 0.0 e 180.0 gradi, mentre aspect è il rapporto tra larghezza e altezza (w/h) del tronco di piramide

Proiezione ortogonale



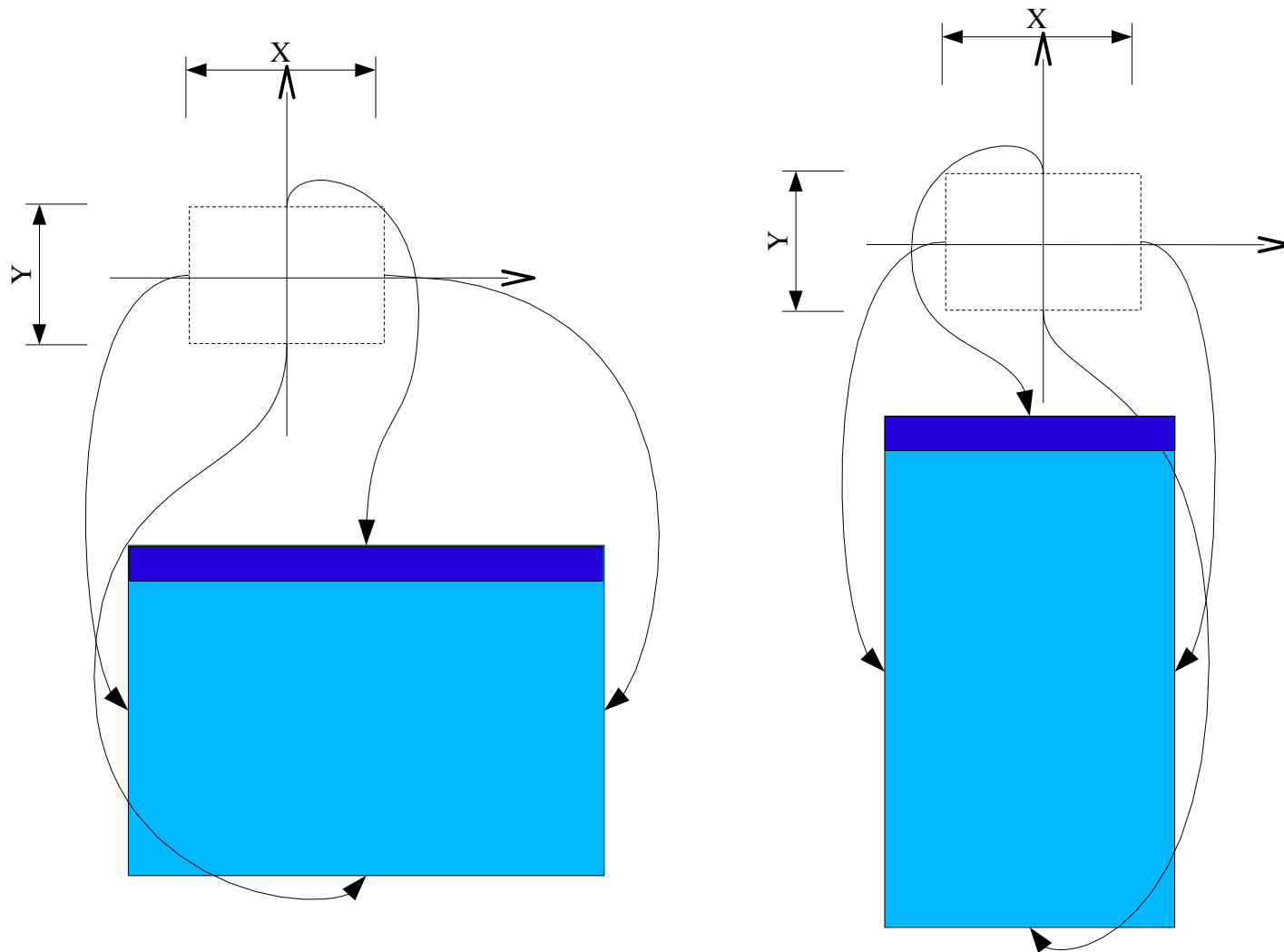
Una proiezione ortogonale è specificata con

void **glOrtho**(*GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far*)

in cui *left* e *right* sono la minima e massima coordinata *x*, *bottom* e *top* la minima e massima coordinata *y* e *near* e *far* la minima e massima coordinata *z*. In questo caso *near* e *far* possono essere positivi o negativi. Esiste la variante

void **gluOrtho2D**(*GLdouble left, GLdouble right, GLdouble bottom, GLdouble top*)
equivalente a **glOrtho** con *near* = 0 and *far* = 1.

Distorsione



Evitare la distorsione

Con le proiezioni nasce un problema: se specifichiamo un viewing volume costante e poi ridimensioniamo la finestra, l'immagine appare distorta: questo perchè lo stesso viewing volume, cioè le stesse coordinate, vengono mappate su regioni di schermo differenti; in altre parole quello che prima era un quadrato potrebbe diventare un rettangolo.

Soluzione possibile (**ma non l'unica**):

If ($w \leq h$)

```
glOrtho (-nRange, nRange, -nRange*h/w, nRange*h/w, -nRange*2.0f,  
nRange*2.0f);
```

else

```
glOrtho (-nRange*w/h, nRange*w/h, -nRange, nRange, -nRange*2.0f,  
nRange*2.0f);
```

Evitare la distorsione

come evitare la distorsione in proiezione prospettica:

si calcola l'aspect-ratio dai parametri restituiti dalla funzione di reshape e si inserisce il nuovo aspect-ratio calcolato nella funzione gluPerspective

```
fAspect = (float)w/(float)h;
```

```
gluPerspective(fovy, fAspect, near, far );
```

Trasformazioni della Viewport

La fase finale delle operazioni di rendering sta nel mappare il risultato della proiezione (ortogonale o prospettica) su una parte dello schermo come specificato con la funzione

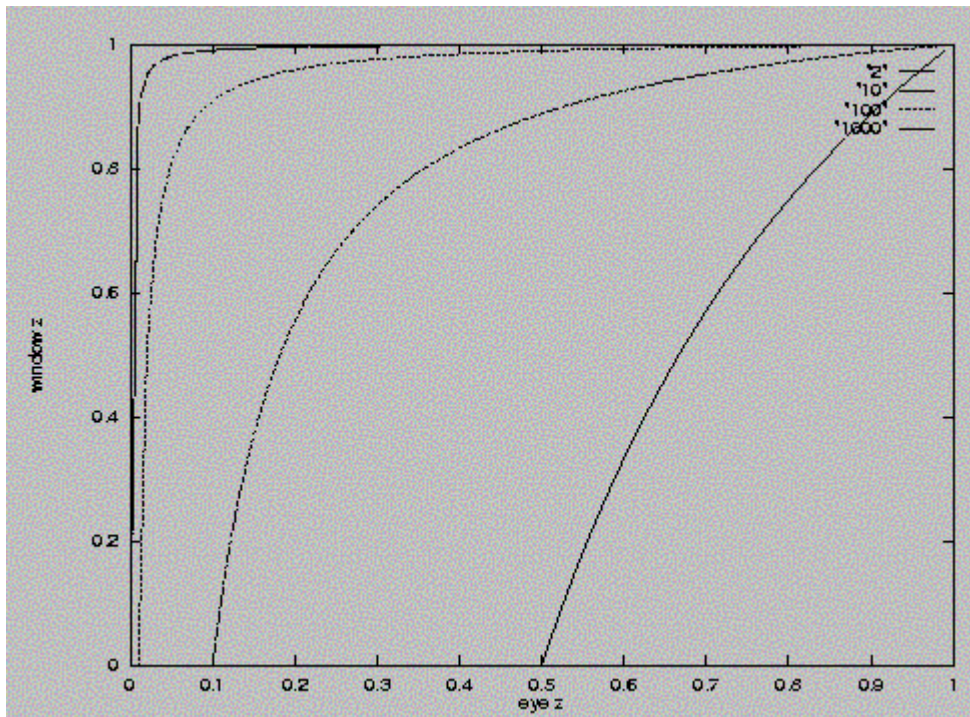
void **glViewport**(*GLint x, GLint y, GLsizei, width, GLsizei height*)

in cui *x* e *y* sono in window coordinates e rappresentano l'angolo superiore sinistro del rettangolo di pixel su cui mappare l'immagine, mentre *width* e *height* sono le dimensioni di tale rettangolo. Solitamente l'aspect ratio (*w/h*) della viewport è uguale all'aspect ratio del viewing volume, altrimenti l'immagine appare distorta.

E' possibile limitare a particolari zone dello schermo (o di una finestra) il rendering con OpenGL, anche se solitamente le dimensioni della viewport coincidono con quelle della finestra in cui si disegna

La coordinata Z e la proiezione prospettica

Come le coordinate x e y , anche la z viene mappata in window coordinates, ma con valori compresi tra 0 e 1, in cui i frammenti con coordinata 0 sono i più vicini al near clipping plane mentre quelli con coordinata 1 i più lontani. Il problema nasce dal fatto che la relazione tra distanza dall'osservatore e coordinata z non è lineare in una proiezione prospettica



Può accadere che con valori del rapporto far/near elevati, diminuisca la precisione del depth buffer

possibile soluzione:

diminuire il rapporto allontanando dal punto di vista il near clipping plane

Depth buffer

Finora le primitive sono apparse una sull'altra a seconda dell'ordine con cui venivano disegnate. Questo sarebbe accaduto anche se avessero avuto diversi valori di posizione sull'asse z. Per tenere conto dell'effetto di profondità occorre attivare il depth buffer. Esso contiene un valore per ogni punto sullo schermo compreso tra 0 e 1 che rappresenta la distanza tra l'oggetto e l'osservatore. Quando due valori di profondità vengono comparati, il più basso apparirà sullo schermo.

Abilitazione: **glutInitDisplayMode(...| GLUT_DEPTH |...);**

 glEnable(GL_DEPTH_TEST);

Per inizializzare tutti i valori nel buffer si la funzione `glClearDepth` es: **glClearDepth(0.5f);**
il valore di default e' 1.0

Come per il color buffer, si utilizza la `glClear` per inizializzare il depth buffer al valore passato alla `glClearDepth`: **glClear(...| GL_DEPTH_BUFFER_BIT);**

Depth buffer

La funzione **glDepthFunc** imposta il criterio in base al quale viene utilizzato il depth buffer

FLAG	DESCRIZIONE
GL_NEVER	non passa mai il test.
GL_LESS	passa il test se il valore di profondità in arrivo è minore del valore memorizzato (DEFAULT).
GL_EQUAL	passa il test se il valore di profondità in arrivo è uguale al valore memorizzato.
GL_LEQUAL	passa il test se il valore di profondità in arrivo è minore o uguale al valore memorizzato.
GL_GREATER	passa il test se il valore di profondità in arrivo è maggiore del valore memorizzato.
GL_NOTEQUAL	passa il test se il valore di profondità in arrivo è diverso dal valore memorizzato.
GL_GEQUAL	passa il test se il valore di profondità in arrivo è maggiore o uguale del valore memorizzato.
GL_ALWAYS	passa sempre il test.

ESEMPIO:

```
glDepthFunc(GL_LEQUAL);
```

ESERCIZIO: Abilitare il DEPTH BUFFER nell'esercizio del cubo inscritto nella sfera

Esempio 1: nucleo di un atomo e tre elettroni

Nell'esempio seguente viene mostrato un esempio di modellazione di una scena che comprende il nucleo di un atomo (fisso) e tre elettroni che gli ruotano attorno su orbite circolari.

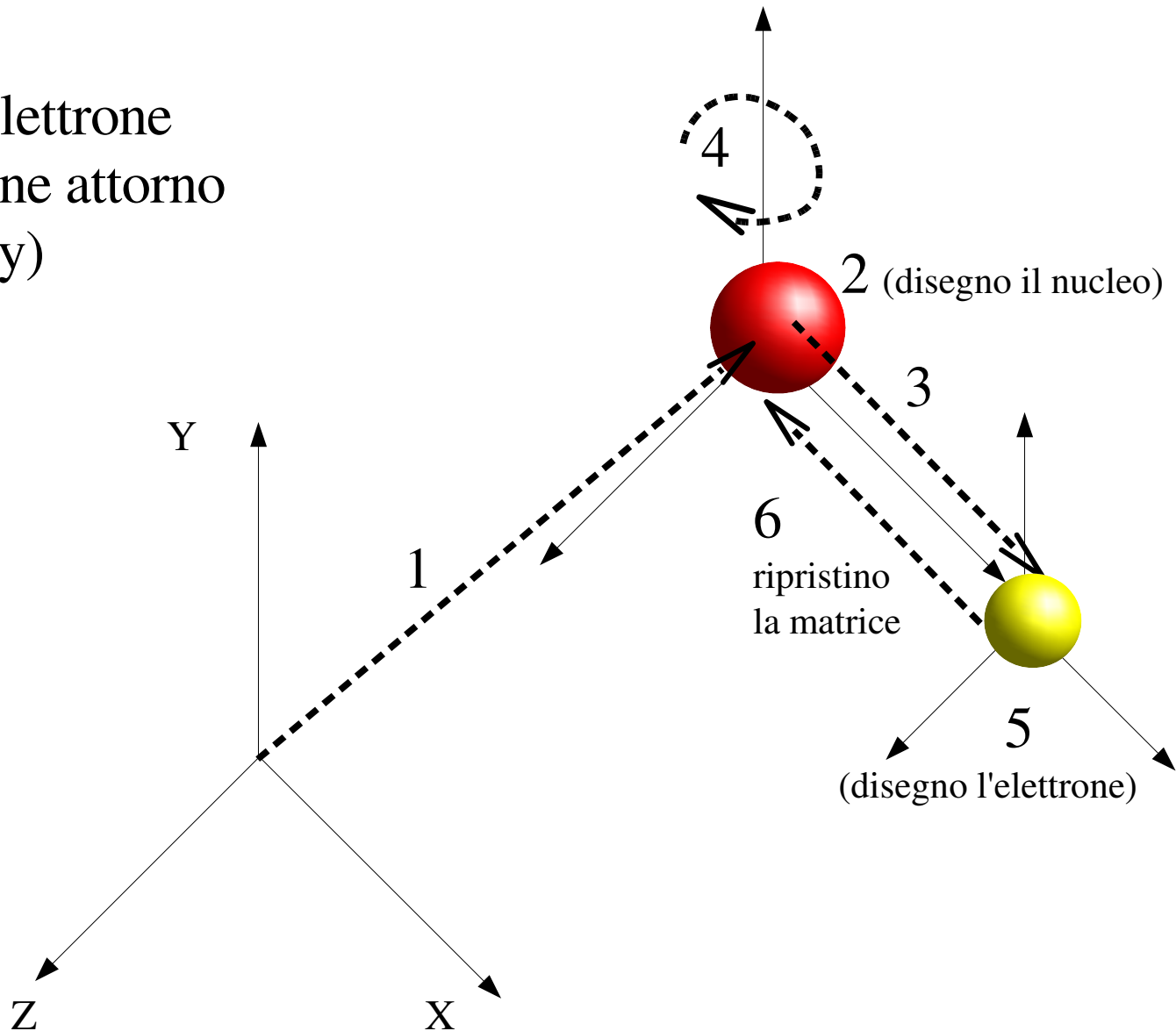
Le orbite hanno inclinazioni diverse rispetto all'asse z, dal quale l'osservatore vede la scena (direzione di default, che non viene modificata)

IMPORTANTE:

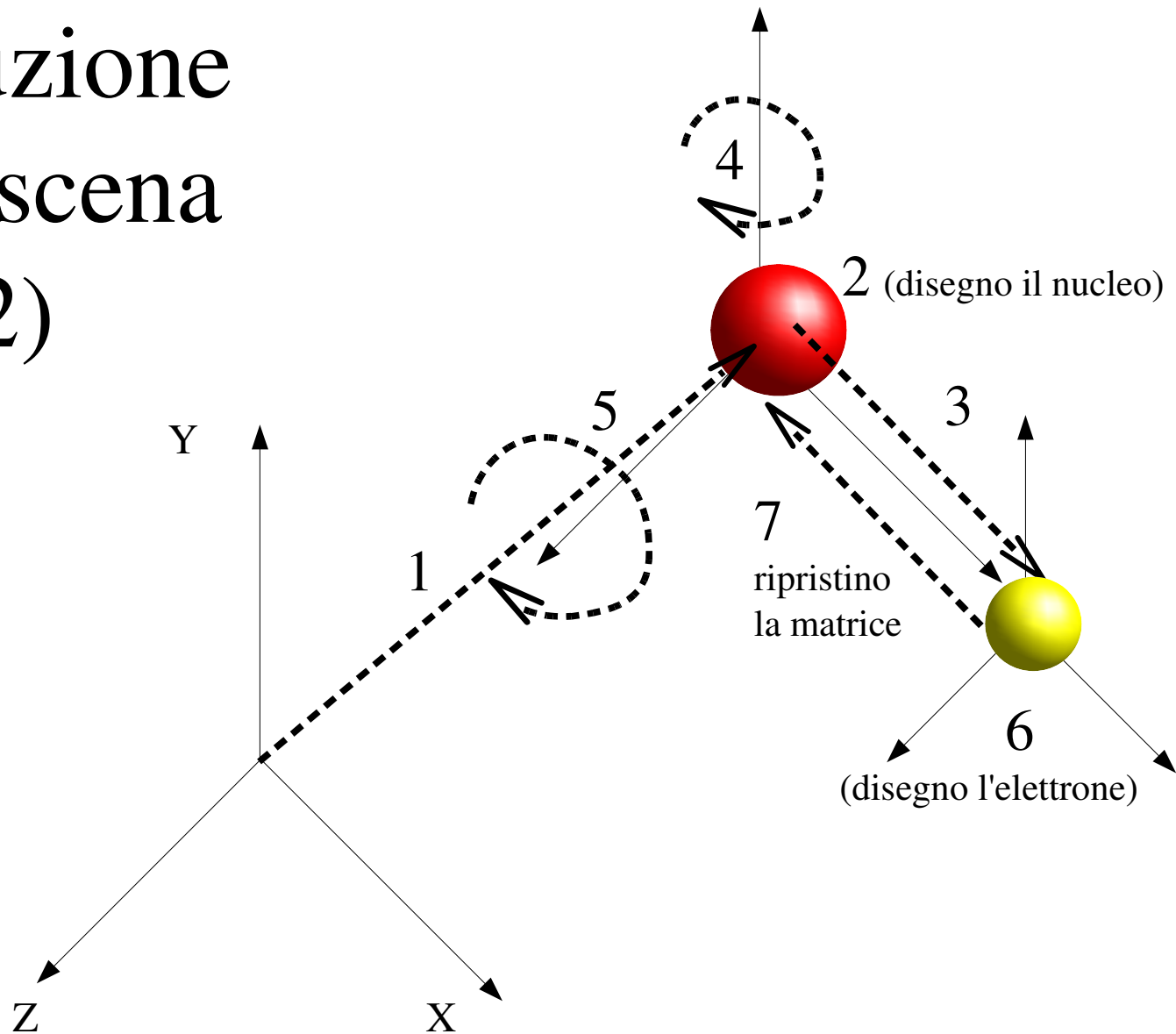
osservare come la funzione `RenderScene` segua lo schema indicato nell'esercitazione precedente per il rendering tridimensionale, in particolare come vengano utilizzate le chiamate a `glPushMatrix` e `glPopMatrix` per creare dei “punti di ripristino” e tornare ad essi

Esempio 1: costruzione della scena (1)

Primo elettrone
(rotazione attorno
all'asse y)



Esempio 1: costruzione della scena (2)



Esempio 1

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

#include <GL/glu.h>

#include <GL/gl.h>

void RenderScene(void){

// Angle of revolution around the nucleus

static GLfloat fElect1 = 0.0f;

glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);

glMatrixMode(GL_MODELVIEW);

glLoadIdentity();

// Translate the whole scene out and into view

glTranslatef(0.0f, 0.0f, -100.0f);

// Red Nucleus

glColor3ub(255, 0, 0); //GLubyte

glutSolidSphere(10.0f, 15, 15);

// Yellow Electrons

glColor3ub(255,255,0);

// First Electron Orbit

// Save viewing transformation

glPushMatrix();

// Rotate by angle of revolution

glRotatef(fElect1, 0.0f, 1.0f, 0.0f);

// Translate out from origin to orbit distance

glTranslatef(90.0f, 0.0f, 0.0f);
```

Esempio 1 (2)

// Draw the electron

```
glutSolidSphere(6.0f, 15, 15);
```

// Restore the viewing transformation

```
glPopMatrix();
```

// Second Electron Orbit

```
glPushMatrix();
```

```
glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
```

```
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
```

```
glTranslatef(-70.0f, 0.0f, 0.0f);
```

```
glutSolidSphere(6.0f, 15, 15);
```

```
glPopMatrix();
```

// Third Electron Orbit

```
glPushMatrix();
```

```
glRotatef(360.0f-45.0f,0.0f, 0.0f, 1.0f);
```

```
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
```

```
glTranslatef(0.0f, 0.0f, 60.0f);
```

```
glutSolidSphere(6.0f, 15, 15);
```

```
glPopMatrix();
```

// Increment the angle of revolution

```
fElect1 += 10.0f;
```

```
if(fElect1 > 360.0f) fElect1 = 0.0f;
```

```
glutSwapBuffers();
```

```
}
```

Esempio 1 (3)

```
void SetupRC() {  
    glEnable(GL_DEPTH_TEST);  
  
    // Hidden surface removal  
  
    glEnable(GL_CULL_FACE);  
  
    // Do not calculate inside  
  
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );  
  
}  
  
void SpecialKeys(int key, int x, int y){  
  
    if(key == GLUT_KEY_UP)  
        glutPostRedisplay();  
  
}
```

```
void ChangeSize(int w, int h) {  
  
    GLfloat nRange = 100.0f;  
  
    // Prevent a divide by zero  
  
    if(h == 0)  
        h = 1;  
  
    // Set Viewport to window dimensions  
  
    glViewport(0, 0, w, h);  
  
  
    // Reset coordinate system  
  
    glMatrixMode(GL_PROJECTION);  
  
    glLoadIdentity();
```

Esempio 1 (4)

```
// Establish clipping volume (left, right, bottom,  
    top, near, far)
```

```
if (w <= h)
```

```
    glOrtho (-nRange, nRange, -nRange*h/w,  
            nRange*h/w, -nRange*2.0f, nRange*2.0f);
```

```
else
```

```
    glOrtho (-nRange*w/h, nRange*w/h,  
            -nRange, nRange, -nRange*2.0f,  
            nRange*2.0f);
```

```
glMatrixMode(GL_MODELVIEW);
```

```
glLoadIdentity();
```

```
}
```

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

```
    glutInitDisplayMode(GLUT_DOUBLE |  
                        GLUT_RGB | GLUT_DEPTH);
```

```
    glutInitWindowSize(800, 600);
```

```
    glutCreateWindow("OpenGL Atom");
```

```
    glutReshapeFunc(ChangeSize);
```

```
    glutSpecialFunc(SpecialKeys);
```

```
    glutDisplayFunc(RenderScene);
```

```
    SetupRC();
```

```
    glutMainLoop();
```

```
    return 0;
```

```
}
```

Esempio 2

Creare un nuovo progetto, copiare il file .c dell'esempio precedente nella nuova directory e aggiungerlo al nuovo progetto (aggiungi elemento esistente) quindi sostituire ChangeSize

```
void ChangeSize(int w, int h) {  
    GLfloat fAspect;  
  
    // Prevent a divide by zero  
  
    if(h == 0)  
        h = 1;  
  
    // Set Viewport to window dimensions  
  
    glViewport(0, 0, w, h);
```

// Reset coordinate system

```
glMatrixMode(GL_PROJECTION);  
  
glLoadIdentity();  
  
fAspect = (float)w/(float)h;  
  
gluPerspective(45.0, fAspect, 1.0, 500.0);  
  
glMatrixMode(GL_MODELVIEW);  
  
glLoadIdentity();  
  
glTranslatef(0.0f, 0.0f, -250.0f);  
}
```

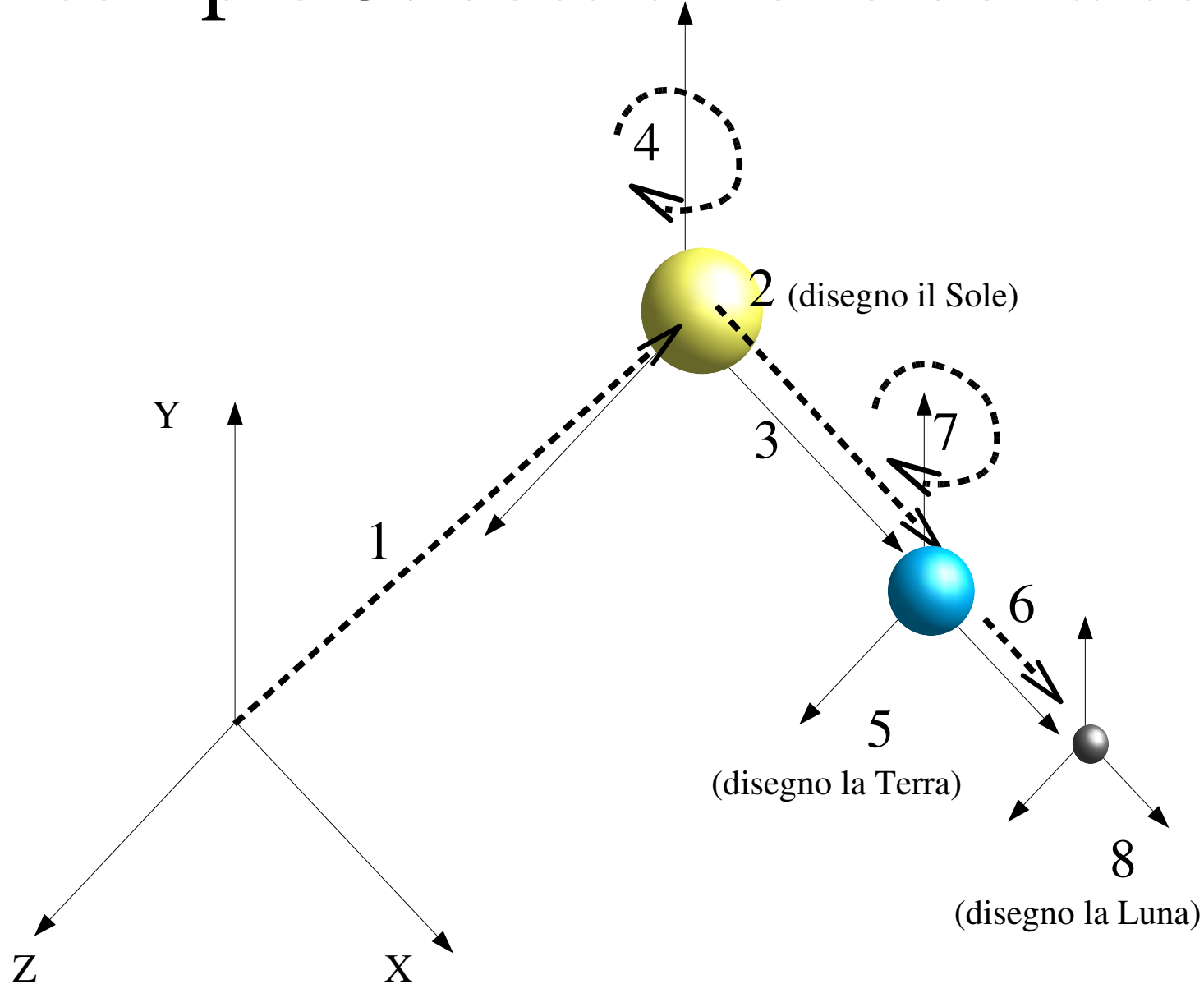
Esempio 3

Nel prossimo esempio viene presentato un altro caso di utilizzo di trasformazioni composte: è rappresentato un “sistema solare” formato da Sole, Terra e Luna. L'esempio 3 presenta inoltre un'altra particolarità: l'animazione.

Tale effetto è ottenuto settando un timer con `glutTimerFunc`, in modo tale che dopo un certo tempo venga chiamata una funzione che produca un evento di “expose” e quindi il ridisegno della scena. Ovviamente da una chiamata all'altra della funzione che disegna la scena varieranno uno o più parametri che modificheranno l'aspetto della scena.

```
void TimerFunc(int value) {  
    glutPostRedisplay();  
    glutTimerFunc(100, TimerFunc, 1); //100 milli-secondi  
}  
...  
glutTimerFunc(250, TimerFunc, 1);
```

Esempio 3: costruzione della scena



Esempio 3 (2)

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

```
#include<GL/glu.h>
```

```
#include<GL/gl.h>
```

```
void RenderScene(void){
```

```
// Earth and Moon angle of revolution
```

```
static float fMoonRot = 0.0f;
```

```
static float fEarthRot = 0.0f;
```

```
// Clear the window with current clearing color
```

```
glClear(GL_COLOR_BUFFER_BIT |  
        GL_DEPTH_BUFFER_BIT);
```

```
// Save the matrix state and do the rotations
```

```
glMatrixMode(GL_MODELVIEW);
```

```
glPushMatrix();
```

```
// Translate the whole scene out and into view
```

```
glTranslatef(0.0f, 0.0f, -300.0f);
```

```
// Sun
```

```
glColor3ub(255, 255, 0);
```

```
glutSolidSphere(15.0f, 30, 17);
```

Esempio 3 (3)

// Rotate coordinate system

glRotatef(fEarthRot, 0.0f, 1.0f, 0.0f);

// Draw the Earth

glColor3ub(0,0,255);

glTranslatef(105.0f,0.0f,0.0f);

glutSolidSphere(15.0f, 30, 17);

// Rotate from Earth based coordinates and draw
Moon

glColor3ub(200,200,200);

glRotatef(fMoonRot,0.0f, 1.0f, 0.0f);

glTranslatef(30.0f, 0.0f, 0.0f);

fMoonRot+= 15.0f;

if(fMoonRot > 360.0f)

fMoonRot = 0.0f;

glutSolidSphere(6.0f, 30, 17);

// Restore the matrix state

glPopMatrix();// Modelview matrix

// Step earth orbit 5 degrees

fEarthRot += 5.0f;

if(fEarthRot > 360.0f)

fEarthRot = 0.0f;

glutSwapBuffers();

}

Esempio 3 (4)

```
void SetupRC() {  
    glEnable(GL_DEPTH_TEST);  
  
    // Hidden surface removal  
  
    glFrontFace(GL_CCW);  
  
    // Counter clock-wise polygons face out  
  
    glEnable(GL_CULL_FACE);  
  
    // Do not calculate inside  
  
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );  
  
    // Black blue background  
  
}
```

```
void TimerFunc(int value) {  
    glutPostRedisplay();  
  
    glutTimerFunc(100, TimerFunc, 1);  
}  
  
void ChangeSize(int w, int h) {  
    GLfloat fAspect;  
  
    // Prevent a divide by zero  
  
    if(h == 0) h = 1;  
  
    // Set Viewport to window dimensions  
  
    glViewport(0, 0, w, h);
```

Esempio 3 (4)

```
// Calculate aspect ratio of the window

fAspect = (GLfloat)w/(GLfloat)h;

// Set the perspective coordinate system

glMatrixMode(GL_PROJECTION);

glLoadIdentity();

// field of view of 45 degrees, near and far
// planes 1.0 and 425

gluPerspective(45.0f, fAspect, 1.0, 425.0);

// ModelView matrix reset

glMatrixMode(GL_MODELVIEW);

glLoadIdentity();

}
```

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

    glutInitDisplayMode(GLUT_DOUBLE |
        GLUT_RGB | GLUT_DEPTH);

    glutInitWindowSize(800, 600);

    glutCreateWindow("Earth/Moon/Sun System");

    glutReshapeFunc(ChangeSize);

    glutDisplayFunc(RenderScene);

    glutTimerFunc(250, TimerFunc, 1);

    SetupRC();

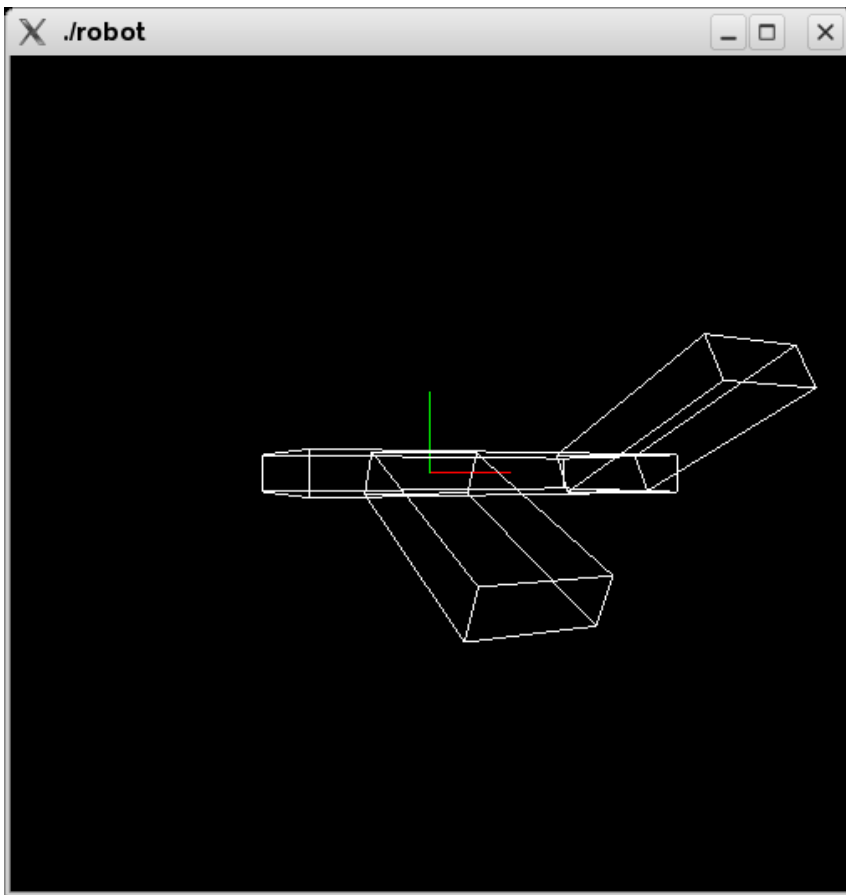
    glutMainLoop();

    return 0;

}
```

Esercizio

Costruire una rappresentazione gerarchica di un “polso” e di due “dita”. Rappresentare il polso e le dita con dei parallelepipedi in wireframe. Le dita possono essere ruotate (alla pressione di tasti) di angoli positivi e negativi rispetto ad un asse solidale al polso. Tutta la scena puo' essere ruotata rispetto agli assi z e y con la pressione appositi tasti. La scena comprende anche un sistema di riferimento (x,y,z) fisso rappresentato mediante tre segmenti colorati di rosso, verde e blu.



Il polso è inizialmente allineato con l'asse x . Le dita sono inizialmente allineate con la base “destra” del polso.

- 1) Disegnare i parallelepipedi nel modo standard, specificando i singoli vertici
- 2) Risolvere l'esercizio utilizzando la funzione `glutWireCube`
- 3) Aggiungere altre falangi/dita al polso

Animazione

Per comprendere meglio il meccanismo dell'animazione vediamo ora un altro esempio. In questo caso un quadrato continuerà a rimbalzare per lo schermo, cambiando direzione quando incontra un bordo della finestra.

In questo caso l'attenzione si pone sulla funzione `TimerFunc` che gestisce la posizione del quadrato e la direzione di movimento.

Lo stesso esempio verrà quindi riproposto abilitando una caratteristica di OpenGL chiamata “stenciling”, che permette di disegnare evitando aree di forma arbitraria, come ad esempio nei casi in cui l'osservatore vede la scena dall'interno di un'automobile o dalla cabina di pilotaggio di un aereo . Come tutte le caratteristiche di OpenGL, lo stenciling si abilita con `glEnable(GL_STENCIL_TEST)` e il buffer coinvolto è lo stencil buffer.

Esempio 4

```
#include<GL/glut.h>
#include<GL/glu.h>
#include<GL/gl.h>

// posizione e dimensione iniziale del quadrato
GLfloat x = 0.0f;
GLfloat y = 0.0f;
GLfloat rsize = 25;

// passo nelle direzioni x e y
// (numero di pixel di cui muovere ogni volta

// tiene traccia delle dimensioni della finestra
GLfloat windowWidth;
GLfloat windowHeight;

// disegna la scena
void RenderScene(void){

// pulisce la finestra con il colore di clearing
glClear(GL_COLOR_BUFFER_BIT);

glColor3f(1.0f, 1.0f, 1.0f);

// disegna un quadrato pieno con il colore corrente
glRectf(x, y, x + rsize, y - rsize);

glutSwapBuffers();

}
```

Esempio 4 (2)

```
void TimerFunction(int value) {  
  
    // inverte la direzione quando arriva al bordo  
    sinistro o destro  
  
    if(x > windowWidth-ysize || x <  
        -windowWidth)  
  
        xstep = -xstep;  
  
    // inverte la direzione quando arriva al bordo  
    superiore o inferiore  
  
    if(y > windowHeight || y < -windowHeight +  
        rsize)  
  
        ystep = -ystep;  
  
    x += xstep;  
  
    y += ystep;
```

// controlla la posizione del quadrato nel caso in cui si ridimensioni la finestra mentre sta rimbalzando

```
    if(x > (windowWidth-ysize + xstep))  
        x = windowWidth-ysize-1;  
  
    else if(x < -(windowWidth + xstep))  
        x = -windowWidth -1;  
  
    if(y > (windowHeight + ystep))  
        y = windowHeight-1;  
  
    else  
  
        if(y < -(windowHeight - rsize + ystep))  
            y = -windowHeight + rsize - 1;
```

Esempio 4 (3)

// ridisegna la scena con le nuove coordinate

```
glutPostRedisplay();
```

```
glutTimerFunc(33,TimerFunction, 1);
```

```
}
```

// inizializza il contesto di rendering

```
void SetupRC(void)
```

```
{
```

// setta il colore di clearing a nero

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

```
}
```

```
void ChangeSize(int w, int h) {
```

```
    GLfloat aspectRatio;
```

// previene una divisione per zero

```
    if(h == 0)
```

```
        h = 1;
```

// setta la viewport alle dimensioni della finestra

```
    glViewport(0, 0, w, h);
```

// resetta il sistema di coordinate

```
    glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();
```

Esempio 4 (4)

```
// stabilisce il volume di visuale (left, right,  
    bottom, top, near, far)
```

```
// aspetto della finestra
```

```
aspectRatio = (GLfloat)w / (GLfloat)h;
```

```
if (w <= h)
```

```
{
```

```
    windowWidth = 100;
```

```
    windowHeight = 100 / aspectRatio;
```

```
    glOrtho (-100.0, 100.0, -windowHeight,  
            windowHeight, 1.0, -1.0);
```

```
}
```

```
else
```

```
{
```

```
    windowWidth = 100 * aspectRatio;
```

```
    windowHeight = 100;
```

```
    glOrtho (-windowWidth, windowWidth,  
            -100.0, 100.0, 1.0, -1.0);
```

```
}
```

```
glMatrixMode(GL_MODELVIEW);
```

```
glLoadIdentity();
```

```
}
```

Esempio 4 (5)

```
int main(int argc, char* argv[])  
{  
    SetupRC();  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_DOUBLE |  
        GLUT_RGB);  
    glutInitWindowSize(800,600);  
    glutCreateWindow("Bounce");  
    glutDisplayFunc(RenderScene);  
    glutReshapeFunc(ChangeSize);  
    glutTimerFunc(33, TimerFunction, 1);  
    glutMainLoop();  
    return 0;  
}
```

Stenciling

Metodo utilizzato per modificare un'immagine da renderizzare utilizzandone un'altra come filtro (applicazioni: mascherare zone dello schermo, ombre dinamiche). Come nella realtà è possibile mascherare con una sagoma di cartone una parte della superficie su cui si dipinge, per ottenere la stessa sagoma dipinta, così in OpenGL è disponibile un buffer, chiamato stencil buffer, che contiene la sagoma che maschera una certa area di disegno. Tecnicamente funziona come il Depth Buffer.

Per abilitare lo stenciling è necessario chiamare

```
glutInitDisplayMode(GLUT_RGB | ... | GLUT_STENCIL)
```

e

```
glEnable(GL_STENCIL_TEST)
```

Una volta abilitato, il disegno della scena avviene solo nei punti che hanno superato lo stencil test, definito con

```
void glStencilFunc(GLenum func, Glint ref, Gluint mask)
```

Stenciling (2)

Func può valere: GL_ALWAYS, GL_NEVER, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, GL_NOTEQUAL.

Questi valori dicono a OpenGL come confrontare il valore nello stencil buffer con quello passato con *ref*. Ad es:

GL_LESS : il test passa se il valore di ref è minore di quello nel buffer

GL_NOTEQUAL : il test passa se il valore di ref è diverso da quello nel buffer

L'ultimo parametro, mask, è una maschera di bit che è posta in AND bit a bit con ref ed il valore nel buffer prima di confrontarli (se la si vuole ignorare, basta passare 0x1 come valore).

Per prima cosa è necessario settare il valore di clearing, ad esempio **glClearStencil(0.0f)**, quindi il buffer va resettato con

glClear(...| GL_STENCIL_BUFFER_BIT| ...)

Stenciling (3)

Prima di poter utilizzare lo stenciling, dobbiamo “disegnare” nello stencil buffer la “sagoma” con cui mascherare l'area di disegno. In questo caso si usa la funzione

void glStencilOp(*GLenum fail, GLenum zfail, GLenum zpass*)

I parametri di glStencilOp indicano come modificare il valore nello stencil buffer in tre casi:

- Quando lo stencil test fallisce (*fail*)
- Quando lo stencil test ha successo e il depth test fallisce (*zfail*)
- Quando lo stencil test ha successo il depth test ha successo (*zpass*)

Valori validi per gli argomenti sono : GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR... (GL_REPLACE sostituisce il valore nel buffer con *ref*)

Stenciling (4)

Riepilogo:

- 1) Abilitare lo Stencil Buffer (**glutInitDisplayMode, glEnable**)
- 2) Inizializzare lo Stencil Buffer (**glClearStencil, glClear**)
- 3) Definire come avviene lo Stencil test (**glStencilFunc**)
- 4) Definire come modificare lo Stencil Buffer dopo il test (**glStencilOp**)
- 5) Il disegno della scena avviene solo nei punti che hanno superato lo stencil test

Stenciling (5)

Nell'esempio che segue è riproposto il cubo che rimbalza nella finestra, ma questa volta quando attraverserà una regione rettangolare al centro dello schermo non verrà disegnato nulla.

Il trucco per definire questo stencil consiste nell' inizializzare lo stencil buffer, abilitare lo stencil test, chiamare `glStencilFunc` in modo tale che il test fallisca sempre, e chiamare `glStencilOp` in modo tale che modifichi opportunamente il valore nel buffer al fallimento del test (cioè sempre).

Una volta terminata questa operazione si passa alla fase di rendering vera e propria, in cui si disegna il quadrato con lo stencil test abilitato, ovviamente dopo che si è chiamata ancora una volta `glStencilFunc` per far fallire il test ogni volta che il valore nel buffer è uguale a quello da noi inserito al passo precedente.

Esempio 5

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

```
#include<GL/glu.h>
```

```
#include<GL/gl.h>
```

```
GLfloat x = 0.0f, y = 0.0f;
```

```
GLfloat rsize = 40;
```

```
// passo lungo x e lungo y
```

```
GLfloat xstep = 1.0f;
```

```
GLfloat ystep = 1.0f;
```

```
// tiene traccia delle dimensioni della finestra
```

```
GLfloat windowHeight;
```

```
GLfloat windowHeight;
```

```
void RenderScene(void) {
```

```
    // colore di clear della finestra nero
```

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

```
    // Usa 0.0 per inizializzare lo stencil buffer
```

```
    glClearStencil(0.0f);
```

```
    glEnable(GL_STENCIL_TEST);
```

```
    // inizializza color e stencil buffer
```

```
    glClear(GL_COLOR_BUFFER_BIT |  
            GL_STENCIL_BUFFER_BIT);
```

```
    glStencilFunc(GL_NEVER, 0x0, 0x0);
```

```
    glStencilOp(GL_INCR, GL_INCR, GL_INCR);
```

Esempio 5 (2)

```
glBegin(GL_TRIANGLES);  
glVertex2f(-20.0f, 0.0f);  
glVertex2f(20.0f, 0.0f);  
glVertex2f(0.0f, 40.0f);  
  
glVertex2f(-20.0f,0.0f);  
glVertex2f(-60.0f,-20.0f);  
glVertex2f(-20.0f,-40.0f);  
  
glVertex2f(-20.0f,-40.0f);  
glVertex2f(0.0f, -80.0f);  
  
glVertex2f(20.0f, -40.0f);  
glVertex2f(60.0f, -20.0f);  
glVertex2f(20.0f, 0.0f);  
glEnd();  
  
glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);  
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);  
glColor3f(1.0f, 1.0f, 1.0f);  
glRectf(x, y, x + rsize, y - rsize);  
glutSwapBuffers();  
}
```

Esempio 5 (3)

void ChangeSize (int w, int h)

e

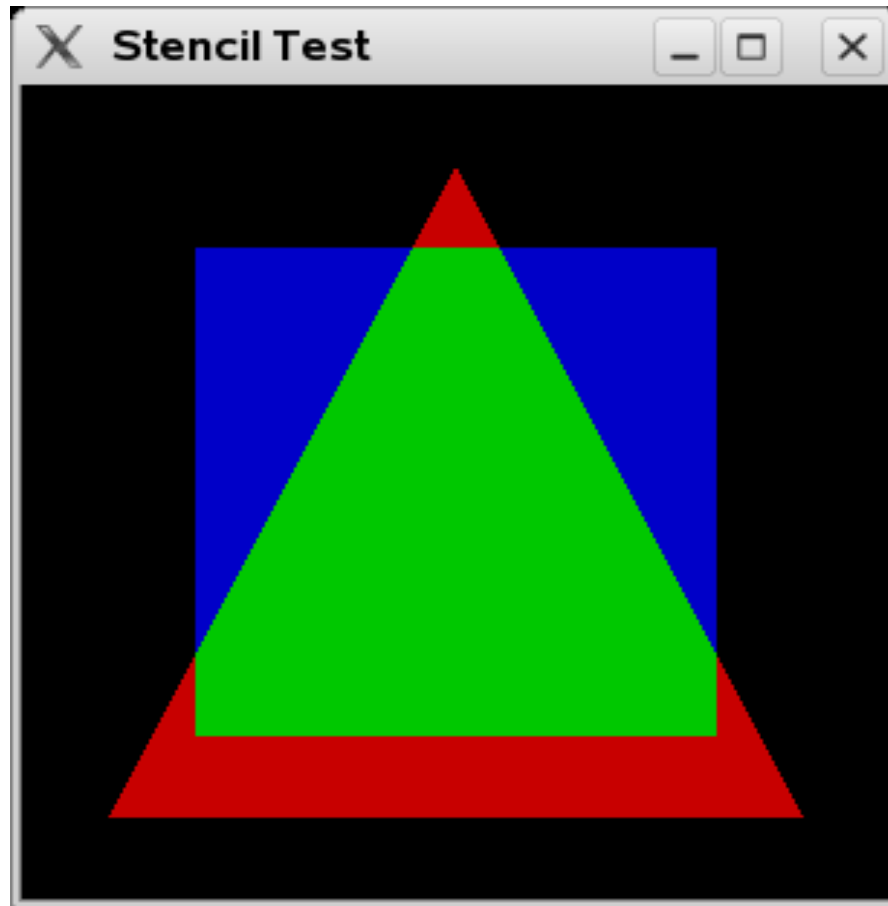
void TimerFunc(int value)

sono uguali a quelle dell'esempio precedente, quindi possono essere copiate e incollate dal sorgente dell'esempio 4.

```
int main(int argc, char* argv[]){  
    glutInit(&argc, argv);  
  
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE  
        | GLUT_STENCIL);  
  
    glutInitWindowSize(800,600);  
  
    glutCreateWindow("OpenGL Stencil Test");  
  
    glutReshapeFunc(ChangeSize);  
  
    glutDisplayFunc(RenderScene);  
  
    glutTimerFunc(33, TimerFunction, 1);  
  
    glutMainLoop();  
  
    return 0;  
}
```

Esercizio

Rappresentare questa scena utilizzando le stencil buffer



Esempio 6 : mouse move

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

```
#include<GL/glu.h>
```

```
#include<GL/gl.h>
```

```
#include<stdio.h>
```

```
GLdouble spherex=0,spherey=0;
```

```
void RenderScene(){
```

```
glMatrixMode(GL_MODELVIEW);
```

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

```
glClear(GL_COLOR_BUFFER_BIT);
```

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

```
glPushMatrix();
```

```
glTranslated(spherex,spherey,0);
```

```
glutSolidSphere(0.2,20,20);
```

```
glPopMatrix();
```

```
glFlush();
```

```
}
```

Esempio 6 :mouse move (2)

```
void MouseMove(GLint x, GLint y){  
    GLdouble spherez;  
    GLdouble model[16];  
    GLdouble project[16];  
    GLint viewport[4];  
  
    glGetDoublev(GL_MODELVIEW_MATRIX,model);  
    glGetDoublev(GL_PROJECTION_MATRIX,project);  
    glGetIntegerv(GL_VIEWPORT,viewport);  
  
    gluUnProject(x,viewport[3]-  
        y,1.0f,model,project,viewport,  
        &spherex,&spherey,&spherez);  
  
    printf("%f %f %f\n",spherex,spherey,spherez);  
    printf("%i %i\n",x,y);  
    printf("_____ \n");  
    glutPostRedisplay();  
}  
  
void main(int argc, char** argv){  
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);  
    glutInitWindowSize(640,640);  
    glutCreateWindow("mouse move");  
    glutDisplayFunc(RenderScene);  
    glutPassiveMotionFunc(MouseMove);  
    glutMainLoop();  
}
```

Schermo -> Scena

Negli esempi precedenti abbiamo utilizzato le chiamate `glTranslate` e `glRotate` per costruire una matrice `modelview` che permettesse di descrivere la posizione e l'orientazione di tutti gli oggetti sulla scena, per poi proiettare sullo schermo una “fotografia” di tale scena: è possibile effettuare l'operazione inversa con

GLboolean `gluUnProject`(*GLdouble winx, GLdouble winy, GLdouble winz, const GLdouble modelMatrix[], const GLdouble projMatrix[], const GLint viewport[], GLdouble *objx, GLdouble *objy, GLdouble *objz*)

`winx, winy, winz` : coordinate del punto sulla finestra (origine angolo alto-sinistra). Il valore di `winz` è espresso in riferimento ai limiti specificati con `glDepthRange` o, per default, tra 0.0 e 1.0)

`modelMatrix` : vettore di 16 elementi ottenuto con `glGetDoublev(GL_MODELVIEW_MATRIX, nome_array)`

`projMatrix` : vettore di 16 elementi ottenuto con `glGetDoublev(GL_PROJECTION_MATRIX, nome_array)`

Scena -> Schermo (2)

viewport : vettore di 4 elementi ottenuto con
`glGetDoublev(GL_VIEWPORT, nome_array)`

objx, objy, objz : puntatori a `Gldouble` utilizzati dal `gluUnProject` per memorizzare i risultati ottenuti e la cui memoria associata è stata allocata precedentemente dall'utente. Il parametro objz potrà assumere un valore in base a quello di winz

E' possibile compiere l'operazione opposta, cioè passare dalle coordinate della scena a quelle dello schermo, con **gluProject**, che accetta gli stessi argomenti di `gluUnProject`, ma con i parametri objx, objy, obj scambiati con winx, winy, winz. In questo caso l'operazione è “esatta”, cioè per ogni punto sulla scena la sua proiezione sullo schermo è determinata univocamente.