Introduzione a OpenGL e alle GLUT

S

OpenGL (Open Graphics Library) è una libreria per fare computer grafica. Il programmatore, per poter visualizzare sullo schermo quanto elaborato con openGL, necessita di usare un buffer video. A questo scopo introdurremo le GLUT (openGL Utilities Toolkit) che forniscono un'interfaccia tra openGL ed il windows manager *del sistema operativo.*

Premessa

- Esistono diverse librerie avanzate per fare grafica 3D interattiva
- Essenzialmente ne sono emerse due: ***Direct3D*** e ***OpenGL***.

Direct3D	***OpenGL***
`www.microsoft.com/windows/directx`	`www.openGL.org`
Appartenenti alle *DirectX* di Microsoft	Sviluppate da Silicon Graphics, Inc.
Chiuse	Semi-aperte
Object Oriented	Scritte in Linguaggio C
Solo per piattaforma Windows	Multipiattaforma (per Linux esiste *Mesa 3D Graphics Library* `www.mesa3d.org`)
Efficienti	Disastro in modalità software
Compatibili con Hardware	Qualche problema di compatibilità
Video ludico	Grafica professionale

GLUT

- Un'applicazione grafica deve poter accedere ad un buffer su video, sia in finestra che a schermo intero
- Tale accesso ***dipende*** dalla piattaforma; le OpenGL non provvedono funzionalità per gestire una finestra su cui disegnare, lasciano il compito al sistema operativo
- Sotto Windows si usano, per esempio, le ***Win32 API***, mentre sotto ambiente Unix (Linux in particolare) si usano, in genere, le librerie ***X-Windows***
- *Che vantaggio c'è ad usare una libreria multipiattaforma se poi il codice deve comunque contenere chiamate a librerie specifiche?*
- Ecco il perché delle ***GLUT***
- Oltre alla gestione delle finestre, offrono dell'altro. Le ***funzionalità*** fornite dalle GLUT possono essere riassunte come segue:
  1. gestione delle finestre
  2. gestione dei menù
  3. la gestione degli eventi (della finestra, del timer, dei dispositivi di input)
  4. gestione di varie primitive geometriche solide e in *wireframe*

- Noi useremo le GLUT non essendo interessati a problemi di prestazioni e discuteremo solo le principali chiamate di funzione
- Per ulteriori informazione seguite il link ***http://www.opengl.org/resources/libraries/glut.html***
- Nella pagina del corso trovate tale link e il manuale (GLUT 3 Specifications) dove poter approfondire quello che noi a lezione accenneremo appena

Come funzionano le GLUT

- GLUT lavora con il paradigma degli ***eventi***
- Ogni volta che succede qualcosa (l'utente preme un tasto sulla tastiera, muove il mouse, sposta una finestra, ecc.) avviene un ***evento***
- Il sistema operativo manda quindi un ***messaggio*** all'applicazione con il tipo di evento
- La gestione dei messaggi tra sistema operativo e applicazione può essere complessa: GLUT intercetta tali messaggi e li gestisce in maniera semplice
- Ad ogni messaggio GLUT fa corrispondere la chiamata ad una funzione ed è compito del programmatore costruire delle funzioni opportune (di rendering, di interazione I/O, ecc.) e ***registrarle*** con chiamate alla API GLUT
- Riassumendo: quando il sistema operativo si accorge di un evento invia un opportuno messaggio all'applicazione, GLUT determina il tipo di evento che lo ha generato e chiama quindi l'opportuna funzione designata dal programmatore

Programmare con le GLUT passo per passo

- Tutte le funzioni della libreria sono della forma `glutSomething(something)`
- Per prima cosa è necessario includere il file header della libreria

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

- In fase di compilazione bisognerà anche eseguire il link del programma con la libreria ***(specifico del compilatore)***. Per esempio, sulla pagina del corso trovate un Makefile di prova con le opzioni `-lglut -lGL -lGLU` per il compilatore `gcc`.
- Quindi si dovranno definire varie funzioni (vedi dopo) per il rendering, l'interazione con i device di input, la gestione dei timer, ecc.
- Tali funzioni o vengono definite in ordine, oppure basta usare i ***prototipi*** all'inizio del file e dopo le dichiarazioni di apertura del preprocessore (meglio ancora mettere i prototipi in un file header esterno)
- Il main del programma quindi richiamerà alcune funzioni interne delle GLUT che ***registrano*** queste funzioni definite dall'utente e le associano ai vari ruoli (rendering, timer, I/O, ecc) .. sarà più chiaro tra un attimo

Inizializzazione

- Iniziamo a vedere le funzioni di inizializzazione

```
/* Main */
int main(int argc, char** argv)
{

  /* Inizializzazione GLUT */
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

- `glutInit(int *argcp, char **argv)` richiede al sistema operativo le risorse per aprire una finestra su cui disegnare. L'argomento è dato dagli argomenti del main e può essere usato per passare delle flag al programma in fase di esecuzione (dipende dal sistema operativo, non ne faremo mai uso)

- `glutInitDisplayMode(unsigned int mode)` setta le caratteristiche di tale finestra tramite OR di maschere di bit; in particolare
  - `GLUT_DOUBLE` inizializza una "double buffered window". Si usa per ottenere animazioni più fluide; l'applicazione disegna l'intera scena su un buffer secondario (back buffer) mentre la scena al passo precedente è mostrata nel buffer principale, e solo quando ha finito i due buffer vengono scambiati. Ovviamente richiede il doppio di memoria rispetto ad una applicazione single buffered.
  - `GLUT_RGB` si chiede una finestra che supporti il formato RGB dei colori (non indicizzato quindi); in realtà è RGBA essendoci anche un valore per l'alpha-blending (vedremo quando parleremo di shading)
  - `GLUT_DEPTH` la finestra deve possedere un ***depth-buffer***, ovvero uno z-buffer (capiremo seguendo la *"Lecture 06: Depth Buffering"*). Il sistema operativo deve quindi fornire la memoria necessaria.
- Ci sono molte altre opzioni settabili con tale funzione, ma per ora ci accontentiamo di queste.

Apertura di Finestre

- Quindi possiamo inizializzare ed aprire la finestra vera e propria

```
/* Creazione della finestra */
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow ("Orpo che Programma");
```

- `glutInitWindowSize(int h, int w)` fissa le dimensioni a `h * w`
- `glutInitWindowPosition(int x, int y)` fissa la posizione del vertice superiore sinistro della finestra a `(x,y)`
- `glutCreateWindow(char *name)` crea la finestra e pone il titolo pari a `name`
- Dalla versione (3.7), GLUT permette anche applicazioni a schermo intero
- Permette inoltre di aprire più finestre e fornisce varie funzioni per la gestione di programmi multi-finestra.

- È bene avere una funzione definita dall'utente in cui si inizializza l'applicazione (ad esempio si settano i parametri della telecamera, si caricano i modelli da rappresentare, ecc.)

```
/* Inizializzazione applicazione */
initgfx ();
```

- ***Nota bene:*** del comportamento di `initgfx()` è completatemente responsabile il programmatore; da non confondersi con la funzione per l'inizializzazione di GLUT `glutInit(...)` vista precedentemente.

Registrazione delle funzioni utente

- A questo punto si possono ***registrare*** le varie funzioni definite dall'utente responsabili del rendering, dell'I/O, del timing ecc.

```
/* Specifica la funzione di rendering */
glutDisplayFunc(drawScene);

/* Specifica la funzione di input da tastiera */
glutKeyboardFunc(keyboard);

/* Specifica la funzione di idle */
glutIdleFunc(animate);
```

- Nell'esempio abbiamo registrato la funzione `drawScene()` come funzione di rendering che viene chiamata ogni volta che l'applicazione deve disegnare sulla finestra, la funzione `keyboard()` come funzione di I/O da tastiera che viene chiamata ogni volta che l'utente usa la tastiera e la funzione `animate()` come funzione che viene chiamata quando non succede nulla (utile per le animazioni, come vedremo nella *"Lecture 07: Animation"*)

Funzioni di registrazione (callback)

- Vediamo alcune principali funzioni di registrazione, altresì dette di ***callback***
- Sono tutte della forma `glutSomethingFunc`
- `glutDisplayFunc(void (*func)(void))`

  registra la funzione che viene richiamata ogni qualvolta GLUT deve disegnare o ridisegnare la finestra; all'interno della funzione registrata andranno quindi, come vedremo, le chiamate alle funzioni OpenGL
- `glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))`

  registra la funzione che gestisce la tastiera ***normale*** (ovvero esclusi i tasti speciali, quali i tasti funzione, il tastierino numerico, le frecce, ecc.). La variabile `key` contiene il carattere ASCII che è stato digitato e le due variabili `x` e `y` la posizione del mouse ***(rispetto alla finestra)***
- `glutSpecialFunc(void (*func)(unsigned char key, int x, int y))`

  registra la funzione che gestisce i ***caratteri speciali*** della tastiera che non vengono gestiti dalla precedente. La variabile `key` contiene il codice relativo al tasto digitato e le due variabili `x` e `y` la posizione del mouse

- `glutMouseFunc(void (*func)(int button, int state, int x, int y))`

  registra la funzione che gestisce i bottoni del mouse; l'intero `button` rappresenta il bottone e può essere `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` o `GLUT_RIGHT_BUTTON`; la variabile `state` da lo stato del bottone e può essere `GLUT_UP` o `GLUT_DOWN`. La posizione del mouse è data da `x` e `y`

- `glutMotionFunc(void (*func)(int x, int y))`

  registra la funzione che gestisce i movimenti del mouse; viene chiamata quando il mouse si muove e gli argomenti danno la posizione in `x` ed in `y` del puntatore.

- `glutIdleFunc(void (*func)(void))`

  registra la funzione di idle, ovvero la funzione che viene chiamata quando non ci sono input da parte dell'utente.

- Dopo aver registrato le funzioni definite dal programmatore con le callback di GLUT si deve cominciare il ***loop principale*** dell'applicazione

```
  /* Entra nel loop principale dell'applicazione */
  glutMainLoop();
  return 0;
}
```

- Segue lo scheletro del main di un'applicazione realizzata con GLUT...

```
/* Main */
int main(int argc, char** argv)
{
  /* Inizializzazione GLUT */
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  /* Creazione della finestra */
  glutInitWindowSize (500, 500);
  glutCreateWindow (argv[0]);
  /* Inizializzazione applicazione */
  initgfx ();
  /* Specifica la funzione di rendering */
  glutDisplayFunc(drawScene);
  /* Specifica la funzione di input da tastiera */
  glutKeyboardFunc(keyboard);
  /* Specifica la funzione di idle */
  glutIdleFunc(animate);
  /* Entra nel loop principale dell'applicazione */
  glutMainLoop();
  return 0;
}
```

- Un esempio banale di funzione `keyboard()`

```
/* Gestione dell'input da tastiera */
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27: /* codice ASCII del tasto ESC */
         exit(0);
         break;
   }
}
```

- Un esempio banale di funzione `animate()`

```
/* Funzione di idle */
void animate(void)
{
  glutPostRedisplay();
}
```

- `glutPostRedisplay()` forza un evento di ridisegno della finestra; al successivo passo del loop la funzione di rendering, definita con `glutDisplayFunc`, viene richiamata.

- Questo conclude (per ora) la breve introduzione alle GLUT
- *Cosa mettere in* `drawScene()`, `initgfx()` *e in versioni più complesse di* `keyboard()` *e di* `animate()`*?*
- Dipende dall'applicazione; comunque, in generale, in quelle funzioni vanno messi comandi ***OpenGL***
- Iniziamo dunque (finalmente) ad introdurre questa libreria

OpenGL

- Per tutorial, esempi e manuali, ***http://www.opengl.org***
- In OpenGL i tipi di dati vengono definiti con nomi aventi il prefisso ***GL***. Ad esempio, anche i tipi base seguono questo criterio
  1. GLint
  2. GLfloat
- Le funzioni OpenGL sono tutte della forma `glSomething(something)`
- Dal momento che le openGL sono una libreria del linguaggio C, privo del meccanismo di ***overloading*** (detto anche *ad-hoc polymorphism*), la stessa funzione, in base al tipo ed al numero di argomenti, viene dichiarata con lo stesso prefisso ma con suffissi diversi
- Ad esempio:
  1. `glVertex3f(GLfloat x, GLfloat y, GLfloat z)`
     definisce un vertice (punto) con tre coordinate float
  2. `glVertex2i(GLint x, GLint y)`
     definisce un vertice (punto) con due coordinate intere

- Le OpenGL sono basate (molto schematicamente) sui seguenti punti:
  1. ***Stati:*** si possono settare vari stati della pipeline grafica (ad esempio abilitare lo z-buffer) tramite comandi del tipo `glEnable(somehting)` e `glDisable(something)`. Tutto quello che viene disegnato dopo un `glEnable(something)` avrà quella particolare feature attivata
  2. ***Stack di matrici:*** OpenGL ne prevede tre, uno stack per la matrice di ***proiezione***, uno per la matrice di ***modeling*** ed uno per la matrice di ***texturing***
  3. ***Primitive:*** tutte le primitive sono richiamate dando un `glBegin(GL_PRIMITIVA)`, seguito da una serie di vertici (vedremo poi nel dettaglio) e concludendo con un `glEnd()`.
- Approfondiamo...

Stati GL

- Gli stati GL sono attivati con la funzione `glEnable(NOME)` e disattivati con `glDisable(NOME)`
- Vediamo due stati che useremo spesso (quando faremo la parte delle OpenGL dedicata allo shading ne vedremo un po' di più):
  1. `GL_DEPTH_TEST:` attiva lo z-buffer (la finestra deve essere stata predisposta dalle GLUT utilizzando la maschera `GLUT_DEPTH` con `glutInitDisplayMode`). Se disattivato gli oggetti vengono disegnati nell'ordine in cui vengono mandati alla pipe-line grafica
  2. `GL_CULL_FACE:` attiva la rimozione delle facce posteriori; necessita che le normali siano definite (lo vedremo nel seguito)
- È bene, per motivi di prestazioni, fare meno cambiamenti di stato possibile
- Il colore dell'oggetto non è un vero e proprio stato, ma si comporta come se lo fosse
- Chiamando la funzione `glColor3f(r, g, b)`, tutti i vertici (e le facce per interpolazione) definiti dopo tale chiamata saranno colorati con (r,g,b).

Stack di Matrici

- Le OpenGL introducono tre stack di matrici
  1. ***Projection:*** questa matrice viene usata per proiettare la geometria sul piano immagine.
  2. ***Model-View:*** questa matrice serve per portare gli oggetti nel riferimento della camera.
  3. ***Texture:*** questa matrice verrà usata nella parte dedicata allo shading
- Gli stack si possono manipolare uno alla volta; per fissare quello su cui si vuole lavorare, che chiameremo stack attuale, si usa la funzione

  ```
  void glMatrixMode(enum mode)
  ```

  dove `mode` può essere `MODELVIEW`, `TEXTURE` o `PROJECTION`

- È possibile manipolare gli stack con le funzioni `glPushMatrix(void)` e `glPopMatrix(void)`. Nella *"Lecture 03: Basic Transformations"* avremo modo di capire meglio l'utilizzo degli stack
  Nota bene: l'operazione di push non ha parametri in quanto *duplica* l'elemento di testa della pila.
- Come modificare la testa dello stack?
  - Per rimpiazzare la testa dello stack attuale con la matrice desiderata si usa la funzione `glLoadMatrix(GLfloat *m)`
  - In particolare, si può caricare la matrice identità nello stack attuale con la funzione `glLoadIdentity(void)`
  - Si può moltiplicare ***a destra*** la matrice attuale dello stack per una matrice m con la funzione `glMultMatrix(GLfloat *m)`. Il risultato verrà posto in testa allo stack
    Nota bene: dopo la moltiplicazione si perde l'informazione in testa allo stack perché sostituito con il risultato della moltiplicazione

- Vi sono poi varie funzioni per poter moltiplicare, sempre a destra, varie matrici utili; eccone alcune
  1. `glRotatef(t, x, y, z):` moltiplica l'attuale matrice per la matrice di rotazione (senso antiorario) di angolo (gradi) t lungo il vettore (x,y,z)
  2. `glTranslatef(x, y, z):` moltiplica l'attuale matrice per la matrice di traslazione lungo il vettore (x,y,z)
  3. `glScalef(x, y, z):` moltiplica l'attuale matrice per la matrice di scalatura con vettore di scala pari a (x,y,z)
- Per chiarimenti si esegua l'esempio del SIGGRAPH 2001 con nome `transformation`, mentre nella *"Lecture 03: Basic Transformations"* proveremo a fare qualche riga di codice

Cosa accade nella pratica

- Ogni volta che si disegna un oggetto specificato in ***coordinate oggetto*** $(x_o, y_o, z_o, 1)$ è necessario passare alle coordinate sul piano immagine (x_p, y_p, z_p, w_p):
  1. Viene prima applicata la matrice M in cima allo stack modelview, ottenendo le ***coordinate vista*** $(x_v, y_v, z_v, 1)$

$$\begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} = M \begin{pmatrix} x_o \\ y_o \\ z_o \\ 1 \end{pmatrix}$$

  2. Quindi viene applicata la matrice P in cima allo stack projection

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ w_p \end{pmatrix} = P \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix}$$

- Il vettore ottenuto viene “normalizzato”

$$\begin{pmatrix} x_n \\ y_n \\ z_n \\ 1 \end{pmatrix} = \begin{pmatrix} x_p/w_p \\ y_p/w_p \\ z_p/w_p \\ w_p/w_p \end{pmatrix}$$

- ***Nota bene:*** M può contenere di fatto due termini: il primo è la matrice di trasformazione della telecamera (o meglio la sua inversa) che permette di passare dallo spazio mondo al sistema solidale alla telecamera, il secondo è la matrice che trasforma l’oggetto dallo spazio oggetto allo spazio mondo.

- Facciamo un esempio concreto, supponendo di aver già fissato la matrice di proiezione

```
/* Lavoro sulla matrice modelview */
glMatrixMode(MODELVIEW);
/* Carico nello stack la matrice della camera mc --> m = mc */
glLoadMatrix(mc);
/* La duplico e, quindi, la ‘‘spingo’’ nello stack */
glPushMatrix();
/* Calcolo la matrice oggetto-mondo di 1 --> m = mc m1 */
glMultMatrix(m1);
Disegna_Oggetto1();
/* Recupero la matrice della camera --> m = mc */
glPopMatrix();
/* La riduplico e, quindi, la ri-‘‘spingo’’ nello stack */
glPushMatrix();
/* Calcolo la matrice oggetto-mondo di 2 --> m = mc m2*/
glMultMatrix(m2);
Disegna_Oggetto2();
/* Recupero la matrice della camera --> m = mc */
glPopMatrix();
/* ecc. ecc. */
```

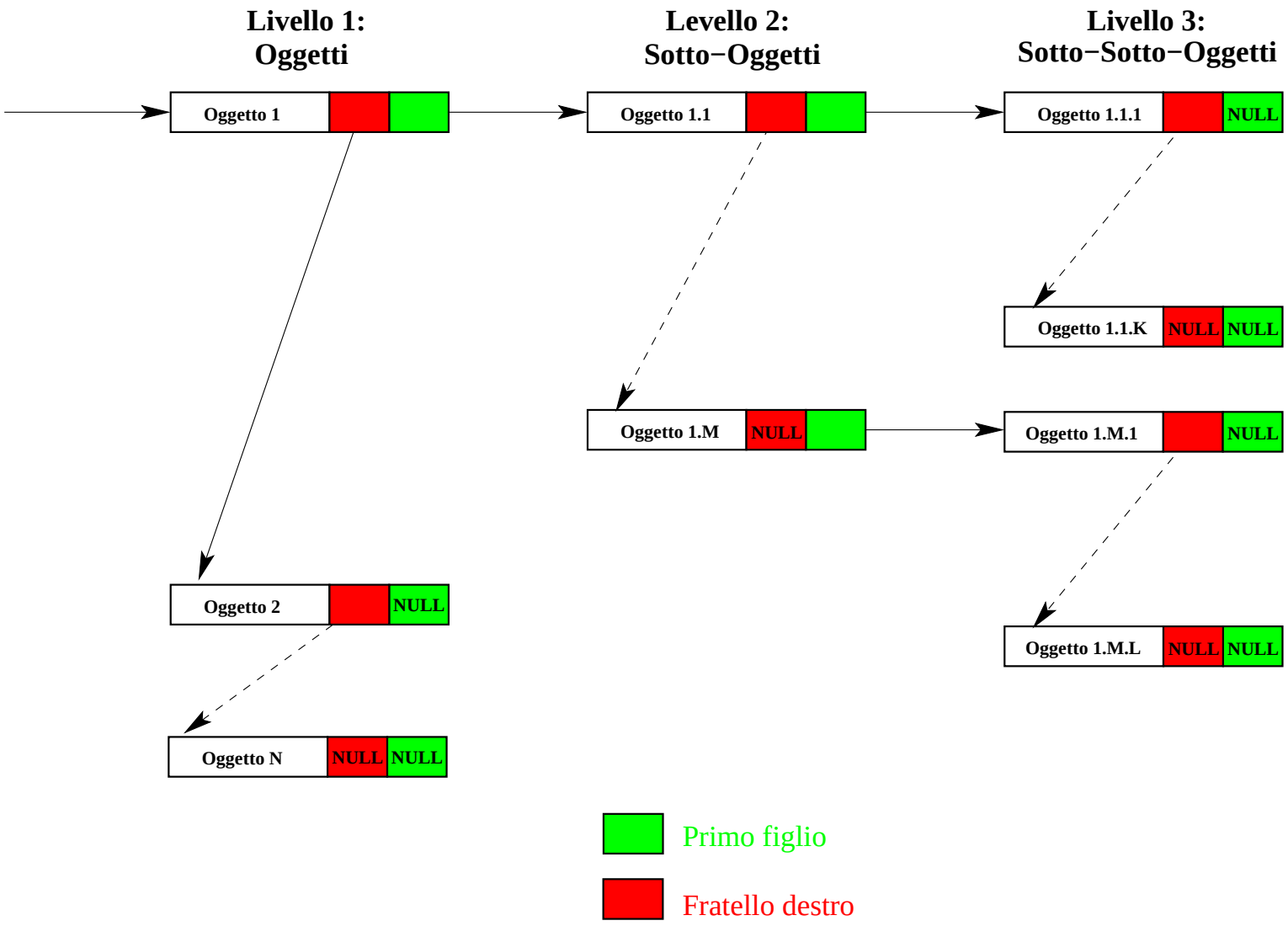
Gerarchie

- Per rappresentare oggetti articolati è necessario definire una relazione gerarchica tra le componenti
- Un primo modo usa direttamente lo stack `MODELVIEW`

```
glMatrixMode(MODELVIEW);
glLoadMatrix(mc); /* carico matrice per un certo oggetto --> m = mc */

glPushMatrix();   /* duplico la testa dello stack --> m = mc */
Disegna_Oggetto();
glTranslatef(...); glRotate3f(...); /* m = mc * T1 * R1 */
Disegna_SottoOgg1();
glPopMatrix();    /* recupero la matrice dell'oggetto --> m = mc */

glPushMatrix();   /* duplico la testa dello stack --> m = mc */
glTranslatef(...); glRotate3f(...); /* m = mc * T2 * R2 */
Disegna_SottoOgg2();
glTranslatef(...); glRotate3f(...); /* m = m * T21 * R2 */
Disegna_SottoSottoOgg21();          /*   = mc * T2 * R2 * T21 * R21 */
/* ecc. ecc. */
```

- Un altro modo, più intelligente, è di definire una struttura dati ad albero:

```
typedef struct{
  float M[16]; /* matrice che lega il sotto-oggetto all'oggetto */
  void (*disegna)(); /* funzione che fa il rendering di tale sotto-oggetto */
  struct treenode *primo_figlio; /* puntatore al primo sotto-sotto-oggetto */
  struct treenode *fratello_destro; /* puntatore al prossimo sotto-oggetto */
} treenode;

void Attraversa(treenode *radice) {
  if (radice == NULL) return;
  glPushMatrix();
  glMultMatrix(radice->M);
  radice->disegna();
  if (radice->primo_figlio != NULL)
     Attraversa(radice->primo_figlio);
  glPopMatrix();
  if (radice->fratello_destro != NULL)
     Attraversa(radice->fratello_destro);
}
```


Livello 1:
Oggetti
Levello 2:
Sotto−Oggetti
Livello 3:
Sotto−Sotto−Oggetti
Oggetto 1
Oggetto 1.1
Oggetto 1.1.1
NULL
Oggetto 1.1.K
NULL
NULL
Oggetto 1.M
NULL
Oggetto 1.M.1
NULL
Oggetto 2
NULL
Oggetto 1.M.L
NULL
NULL
Oggetto N
NULL
NULL
Primo figlio
Fratello destro

Come osservare una scena

- Per osservare una scena si deve posizionare la "videocamera" e fissare il campo di vista, ovvero si devono definire la matrice di proiezione e quella della camera.
- Un modo banale è specificare il volume (parallelepipedo) da osservare e proiettare ortogonalmente su di una sua faccia (piano immagine). Per fare ciò si usa:
  `glOrtho(left, right, bottom, top, zNear, zFar)`
  dove `left` e `right` definiscono le coordinate x, rispettivamente, dei vertici sinistri e destri, `bottom` e `top` definiscono le coordinate y, rispettivamente, dei vertice in basso e in alto, mentre `zNear` e `zFar` definiscono le coordinate z, rispettivamente, della faccia su cui proiettare e dell' "orizzonte"
- Capiremo meglio seguendo la *"Lecture 02: Rendering"* del tutorial online

- Più elegante è l'utilizzo del modello *pin-hole*. A tal fine si useranno delle funzioni ausiliarie fornite dalla libreria ***GLU***; usando le GLUT gli header della GLU e delle openGL sono automaticamente caricati (bisogna ricordarsi di linkarli)

- Le funzioni sono le seguenti

  1. `gluLookAt(px, py, pz, vx, vy, vz, ux, uy, uz)`: informalmente diremo che permette di posizionare "videocamera"; formalmente, moltiplica l'attuale matrice (si usa con la `MODELVIEW`) con la matrice di roto-traslazione corrispondente ad un punto di vista in `(px, py, pz)` che guarda `(vx,vy,vz)` e con vettore up pari a `(ux,uy,uz)`.
  2. `gluPerspective(a, ar, n, f)`: informalmente definisce il volume (frustum) da osservare; formalmente, moltiplica l'attuale matrice (si usa con la `PROJECTION`) per la matrice di proiezione con `a` angolo di apertura orizzontale, `ar` aspect-ratio, `n` near clipping plane e `f` far clipping plane

- Per capire il funzionamento di queste due funzioni si esegua l'esempio del SIGGRAPH 2001 con nome `projection` e nella *"Lecture 03: Basic Transformations"* avremo modo di esercitarci

Primitive grafiche

- Le primitive grafiche in OpenGL sono tutte specificate da una lista di vertici
- Cominciano con la funzione `glBegin(GL_PRIMITIVA)` e terminano con `glEnd()`.
- In mezzo vi è una lista di vertici (il cui significato varia a seconda del valore di `GL_PRIMITIVA`), con possibilmente anche la specifica dei colori (e, vedremo in seguito, di altre quantità legate allo shading). Il colore viene interpolato sulle facce con la tecnica bi-lineare già vista.

```
glBegin(GL_PRIMITIVA);

glColor3f(1.0, 0.0, 0.0);
glVertex3d(1.0, 1.0, 0.0);

glColor3f(1.0, 1.0, 0.0);
glVertex3d(0.0, 1.0, 0.0);

/* ecc. ecc. */
glEnd();
```

- Vediamo le principali primitive
  1. `POINTS:` ogni vertice passato rappresenta un punto nello spazio; il colore è specificato dall'ultima chiamata a `glColor3f`
  2. `LINES:` ogni coppia di vertici consecutivi definisce un segmento passato alla pipeline grafica. Se i vertici sono dispari, l'ultimo non viene considerato
  3. `LINE_STRIP:` è come il punto precedente, ma i segmenti si considerano collegati, quindi ogni vertice costituisce un segmento con il vertice precedente (ovviamente fa eccezione il primo). Non c'è problema per un numero dispari di vertici
  4. `LINE_LOOP:` come il precedente, ma l'ultimo vertice ed il primo sono connessi e formano un segmento.
  5. `TRIANGLES:` questa primitiva costruisce un triangolo per ogni terna di vertici consecutivi che contiene. Se il numero di vertici non è multiplo di tre gli ultimi vengono ignorati.
  6. `TRIANGLE_STRIP:` come la precedente, ma i primi tre vertici definiscono il primo triangolo, ogni vertice successivo definisce un triangolo con i due vertici precedenti
  7. `TRIANGLE_FAN:` come la precedente, ma con un vertice a comune a tutti i triangoli
- Per capire meglio si esegua l'esempio del SIGGRAPH 2001 con nome `shapes` e per esercitarsi fare riferimento al modulo *"Lecture 02: Rendering"* del tutorial online