Libreria Grafica OpenGL

www.opengl.org

Che cosa è OpenGL

OpenGL (Open Graphics Library) è una libreria per creare applicazioni interattive, per gestire oggetti geometrici 3D ed immagini, per ottenere grafica ad alte prestazioni.

OpenGL è una libreria grafica indipendente dal sistema operativo e dal Window System nel senso che permette il rendering, ma non specifica come gestire finestre e ricevere eventi dal sistema.

Ogni **Window System** che supporta OpenGL offre quindi chiamate aggiuntive per integrare OpenGL nella gestione di finestre, colormap ed altre caratteristiche.

Un po' di Storia

- Inizialmente fu sviluppata da **Silicon Graphics**
- Oggi OpenGL **ARB** (**A**rchitecture **R**eview **B**oard)
  - mantiene e aggiorna le *specifiche*
  - versione attuale: 4.5 (agosto 2014)
  - una compagnia, un voto
- Inoltre ci sono le *estensioni* private
  - Soprattutto ***ATI*** e ***nVIDIA***

ARB

sgi intel

hp invent INTERGRAPH

COMPAQ nVIDIA.

Microsoft IBM

ATI ...

API
(Application Programming Interface)

- Librerie per interfacciarsi con **OpenGL (GL)**:

– **GLX** per Xwindow System

– **CGL** per Apple Macintosh

– **WGL** per Microsoft Windows

– **EGL** per OpenGL ES su mobile ed embedded device

- **GLU** (OpenGL Utility Library)

– Inclusa nella **GL** definisce funzioni di utilità e altre primitive grafiche (curve e superfici NURBS)

- **GLEW** (OpenGL Extension Wrangler Library)

– semplifica l'accesso alle funzioni OpenGL ed il lavoro con le OpenGL extension

(continua)

API
(Application Programming Interface)

- **GLUT** (OpenGL Utility Toolkit) (più precisamente freeglut)

– Funzioni per creare semplici interfacce utente, gestire eventi (keyboard e mouse); (interfaccia con il Window System)

- **SDL** (Simple Directmedia Layer)

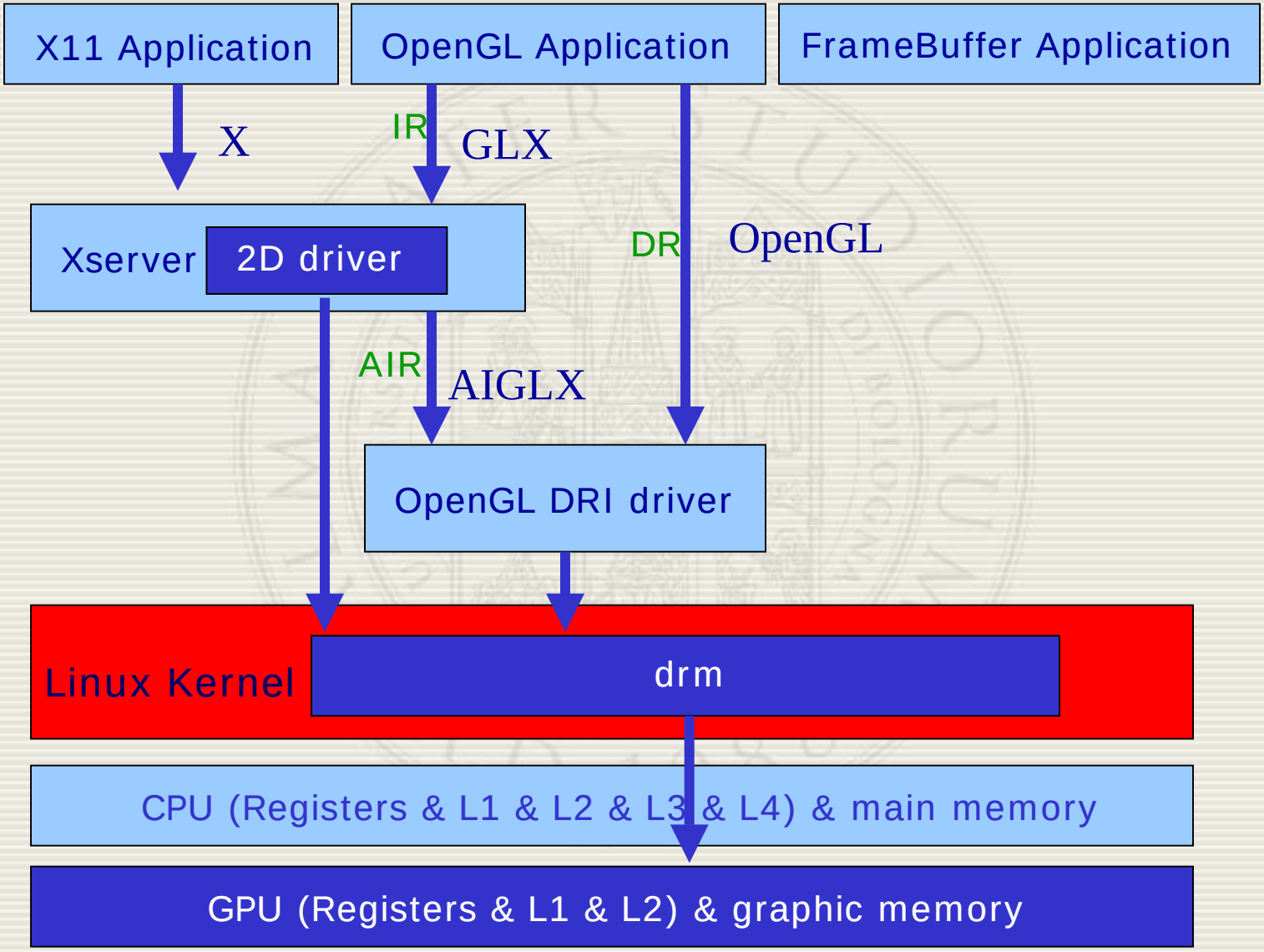
– Funzioni per gestire eventi (keyboard e mouse) e molto altro; (interfaccia con il Window System)

API
(Application Programming Interface)

- **OpenGL ES**
  - progettatta per embedded device
  - basata su OpenGL 3.1
  - basata su shaders

- **Web GL**
  - implementazione JavaScript di OpenGL ES 2.0
  - presente sui più recenti browser

Due parole sulla GLX



GLX

GLX (acronimo di "OpenGL Extension per X Window System") è il punto di connessione per OpenGL e X Window System: permette ai programmi che desiderano usare OpenGL, di farlo dentro una finestra fornita dall' X Window System. La GLX è costituita da tre parti:

1.Una API che fornisce funzioni OpenGL all'applicazione X Window System.

2.Un'estensione del protocollo X, che permette al client (l'applicazione OpenGL) di spedire comandi che fanno richiesta di rendering 3D al server X (il software responsabile della visualizzazione). Il client e il server possono essere eseguiti su computer differenti.

3.Un'estensione del server X che riceve i comandi di rendering dal client. Questa estensione o passa i comandi all'hardware accelerato 3D scheda video oppure li renderizza con un programma che sfrutta le librerie Mesa (metodo più lento).

GLX

Se il client e il server sono eseguiti sulla stessa macchina ed è disponibile una scheda grafica accelerata 3D con relativi driver, il client e il server possono essere bypassati mediante la Direct Rendering Infrastructure (DRI). In questo caso, il programma client può accedere direttamente all'hardware della scheda video.

Molte informazioni riguardo alla GLX, possono essere ottenute con il comando "glxinfo“ da shell (per esempio se il Direct Rendering è attivato o qual è la versione di OpenGL utilizzata e molte altre info).

La GLX è stata creata da Silicon Graphics ed è alla versione 1.4.

GLX, sia con DRI che con Mesa, è inclusa nelle versioni successive alla X11R6.7.0 di X Window System, e in XFree86 dalla versione 4.0.

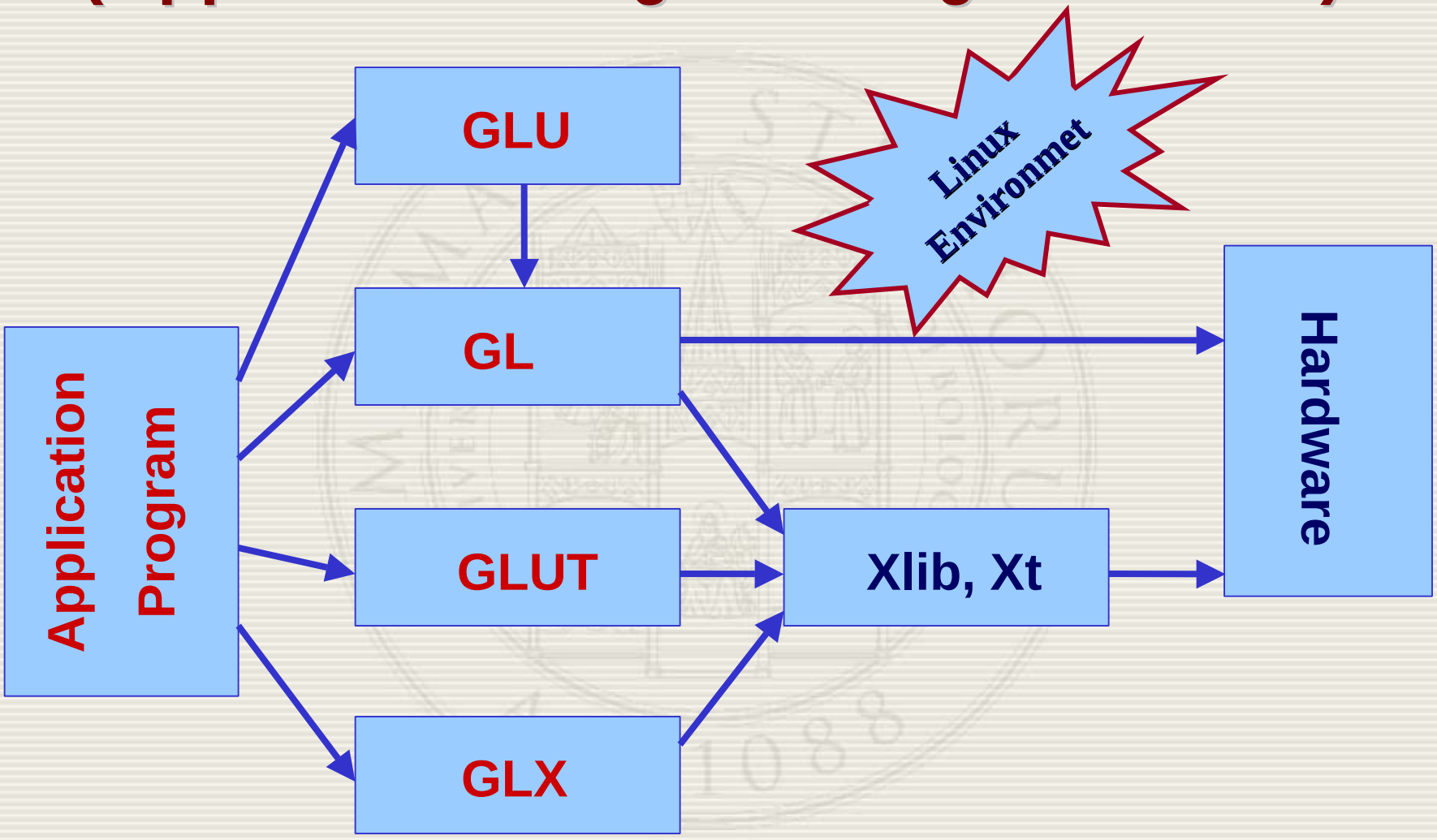
Due parole sulla AIGLX

Accelerated Indirect GLX (AIGLX) è un progetto open source fondato dalla X.Org Foundation e dalla comunità Fedora Core al fine di fornire capacità di rendering con accelerazione indiretta GLX a X.org e driver Direct Rendering Infrastructure (DRI).

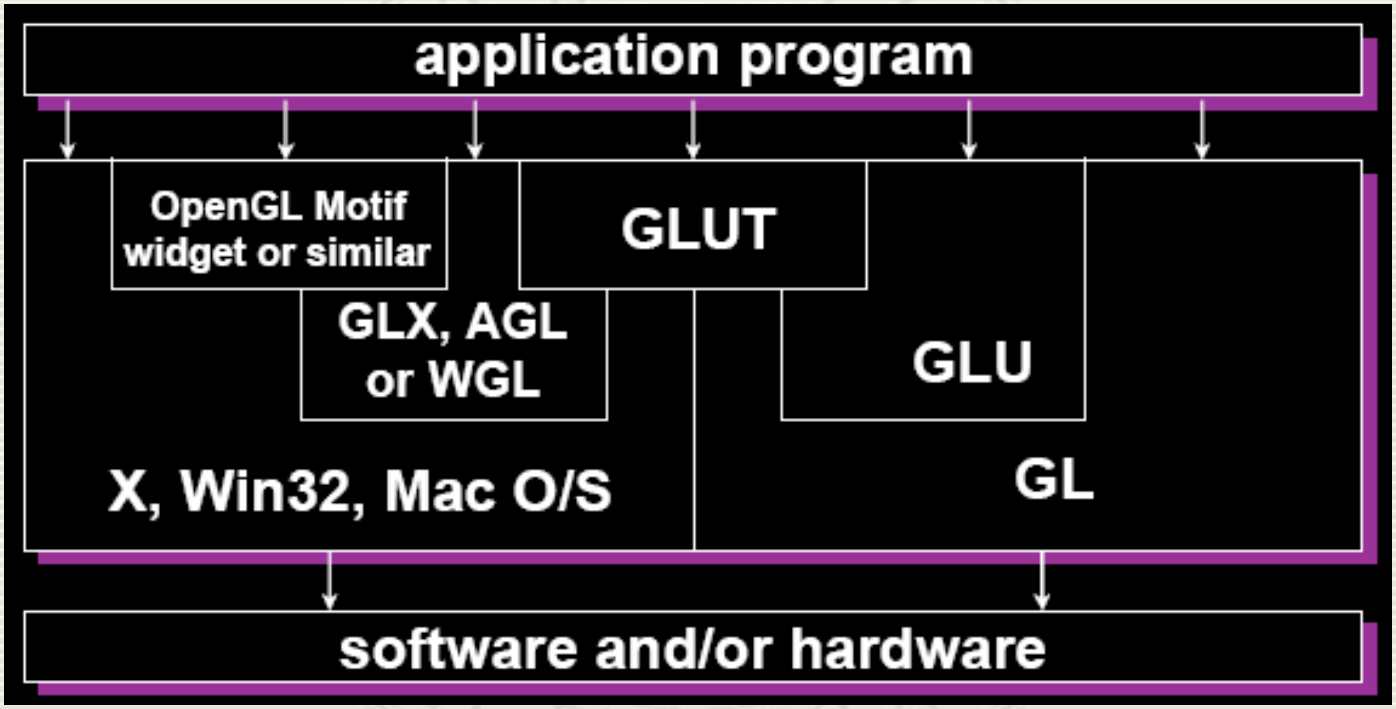
Questo permette a client X remoti di sfruttare a pieno le possibilità di accelerazione hardware attraverso il protocollo GLX.

Il modulo AIGLX è stato inserito di default in X.Org versione 7.1 e successive

API
(Application Programming Interface)



API
(Application Programming Interface)



OpenGL

- Programmable Pipeline

➢Dalla OpenGL 2.0, Pixel e Fragment shaders
➢OpenGL Shading Language (GLSL)

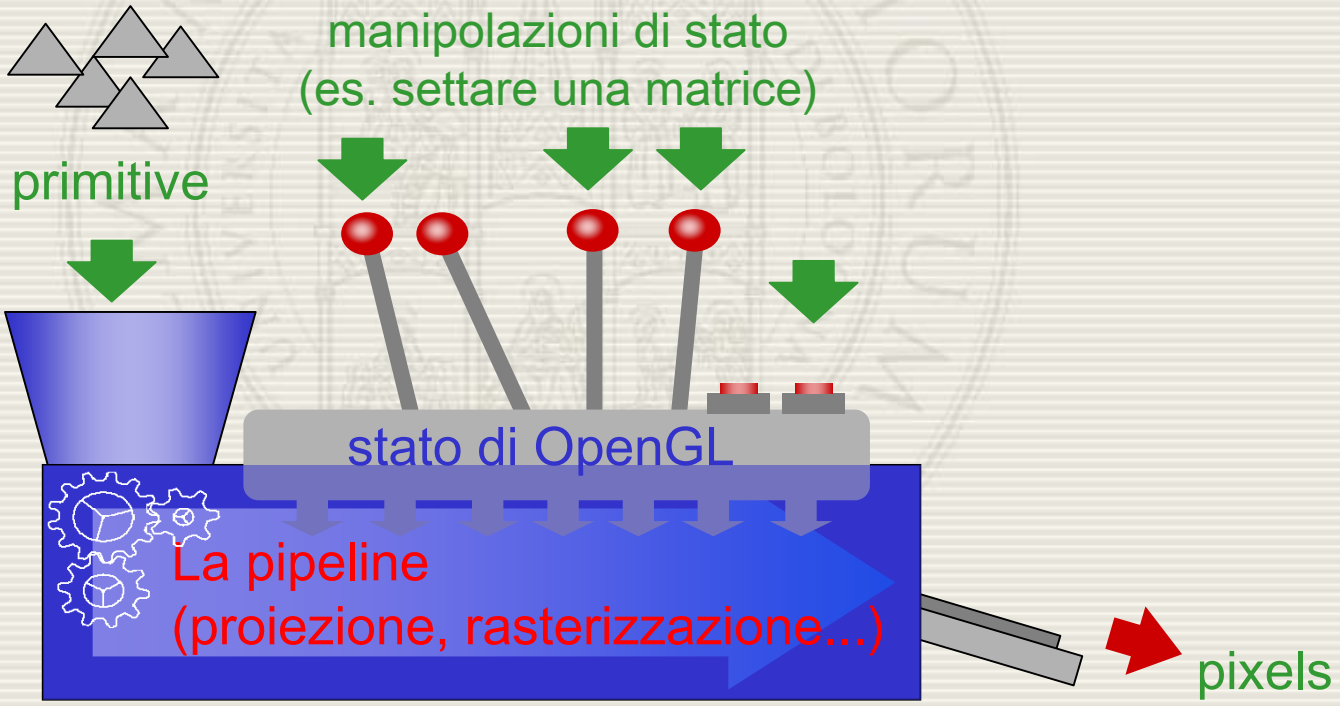
- Fixed-Function Pipeline

➢Fino alla OpenGL 1.5
➢Dalla OpenGL 2.0 alle 3.0 massima compatibilità fra i due modi, con le fixed function emulate da shaders
➢Dalla OpenGL 3.2 alcune fixed function non sono più emulate

Una macchina a Stati

OpenGL = macchina a STATI FINITI

l'input dall'applicazione altera gli stati e induce la macchina a produrre un output visibile.



Una macchina a Stati

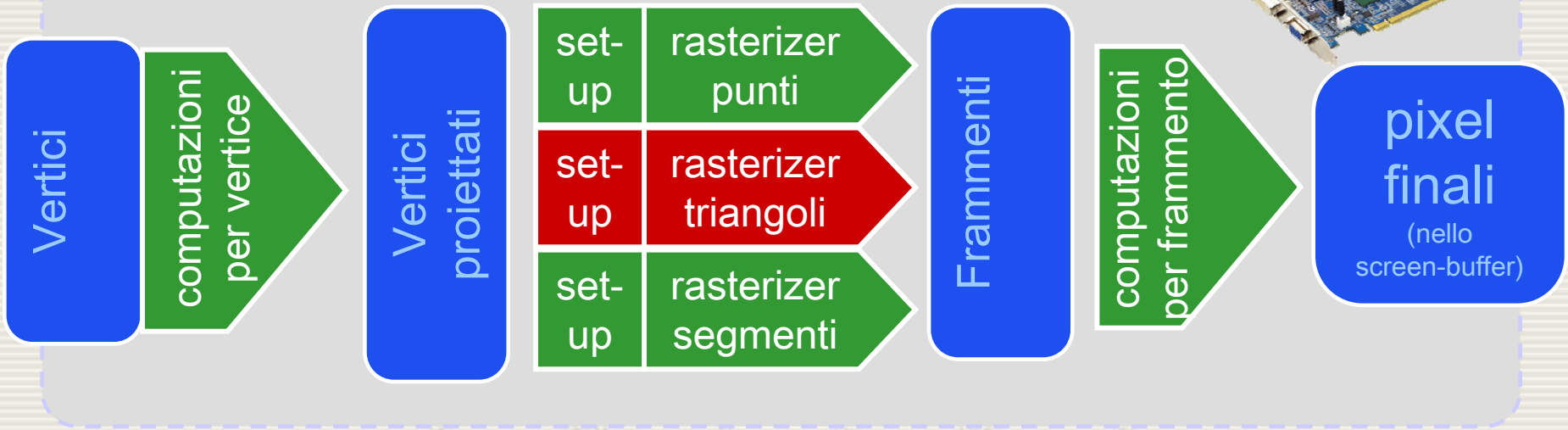
Tutti gli attributi di rendering sono gestiti da stati OpenGL:

Trasformazioni/Proiezioni
Stili di rendering
Texture mapping
Shading
Luci

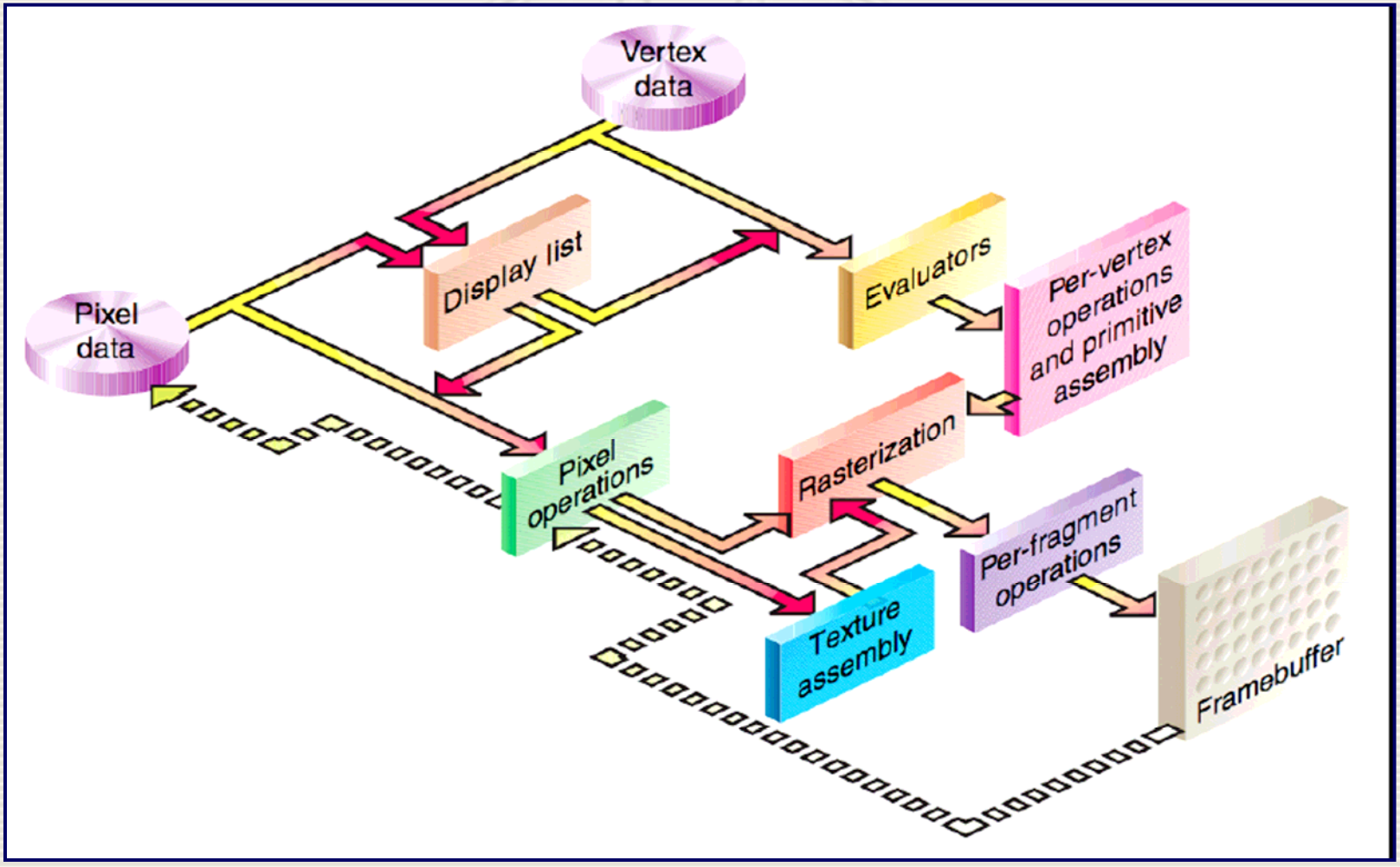
L'apparenza è controllata dallo stato corrente

```
for each primitive to render
{
        update OpenGL state
        render primitive
 }
```

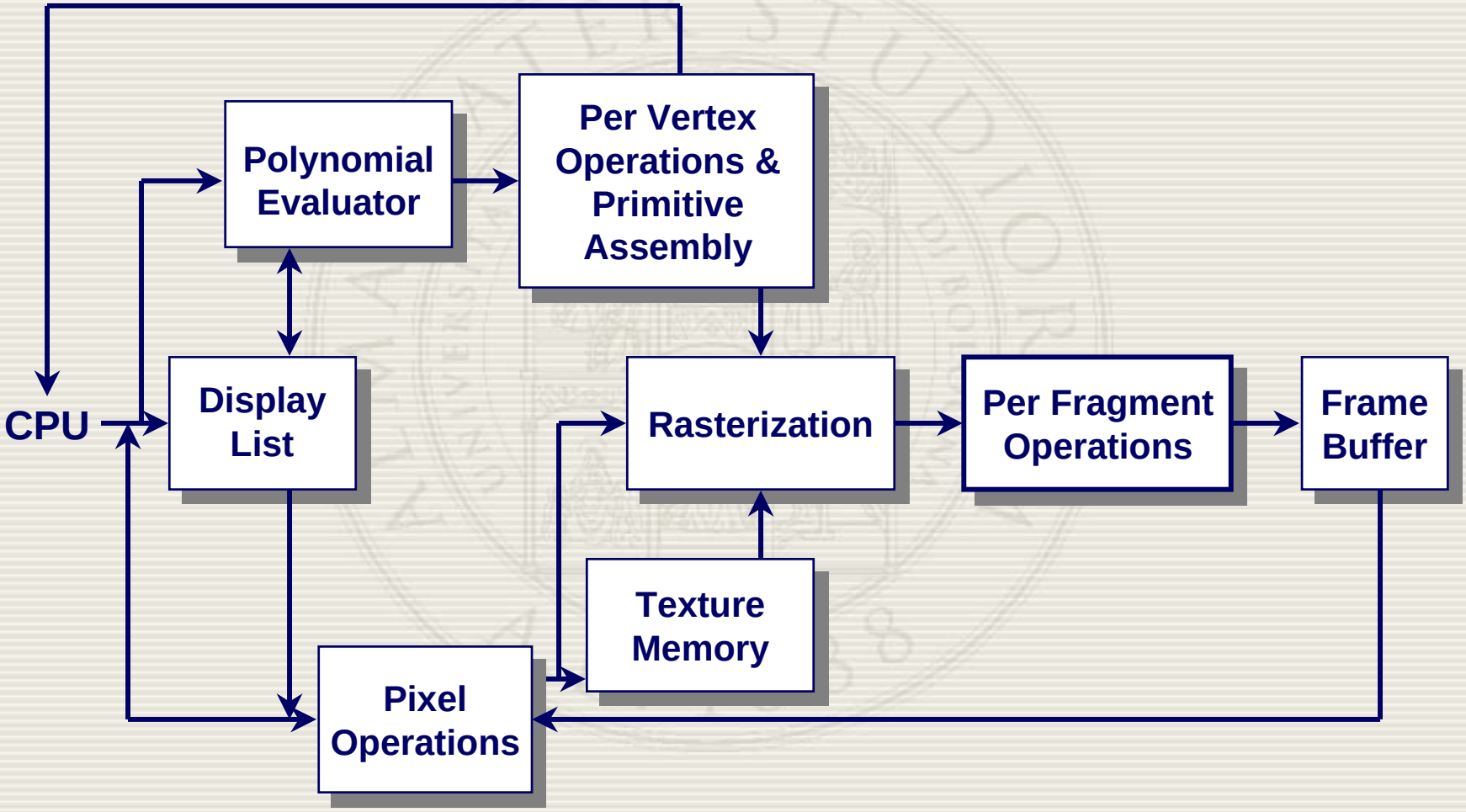
Pipeline Grafica



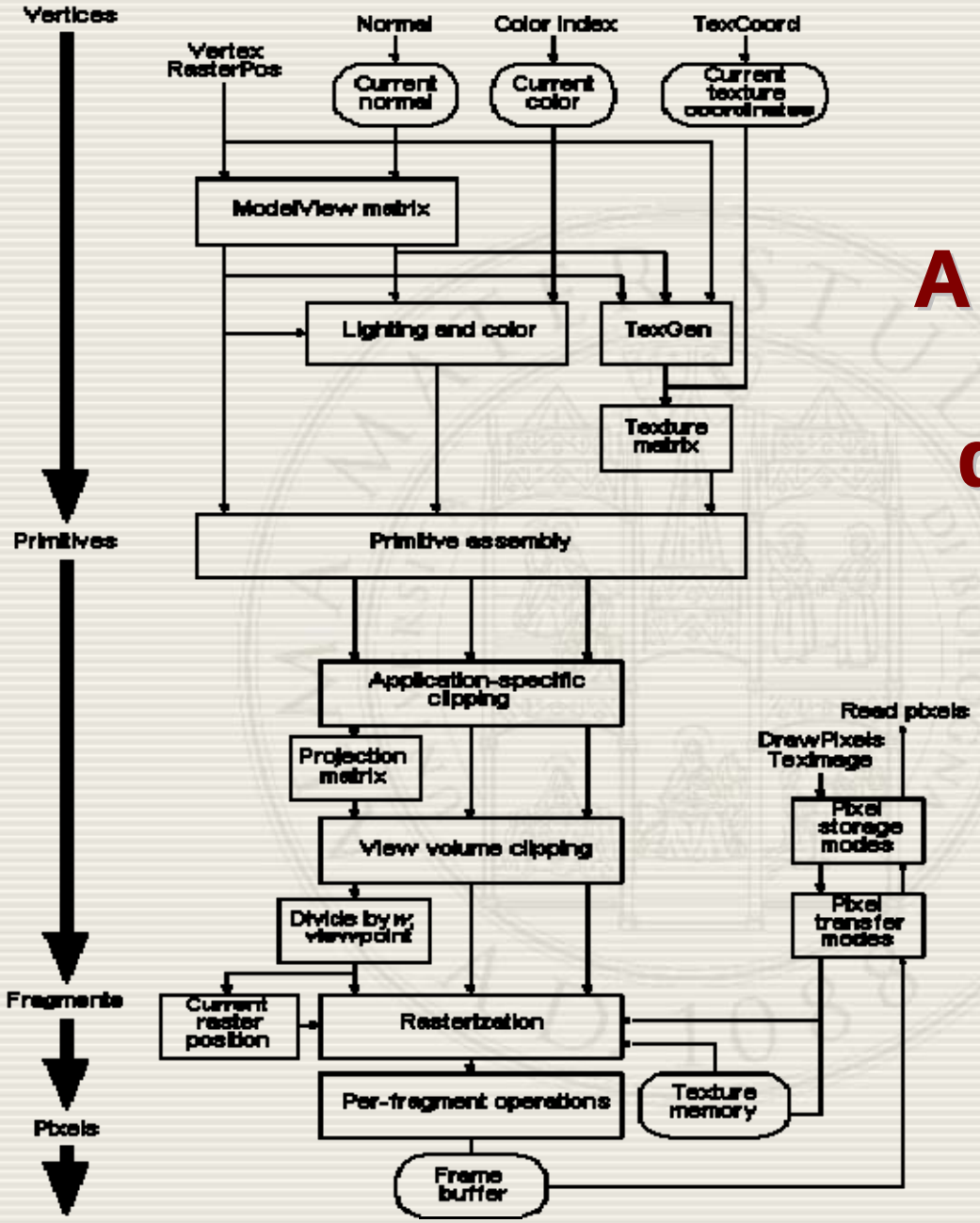
Architettura Pipeline di OpenGL



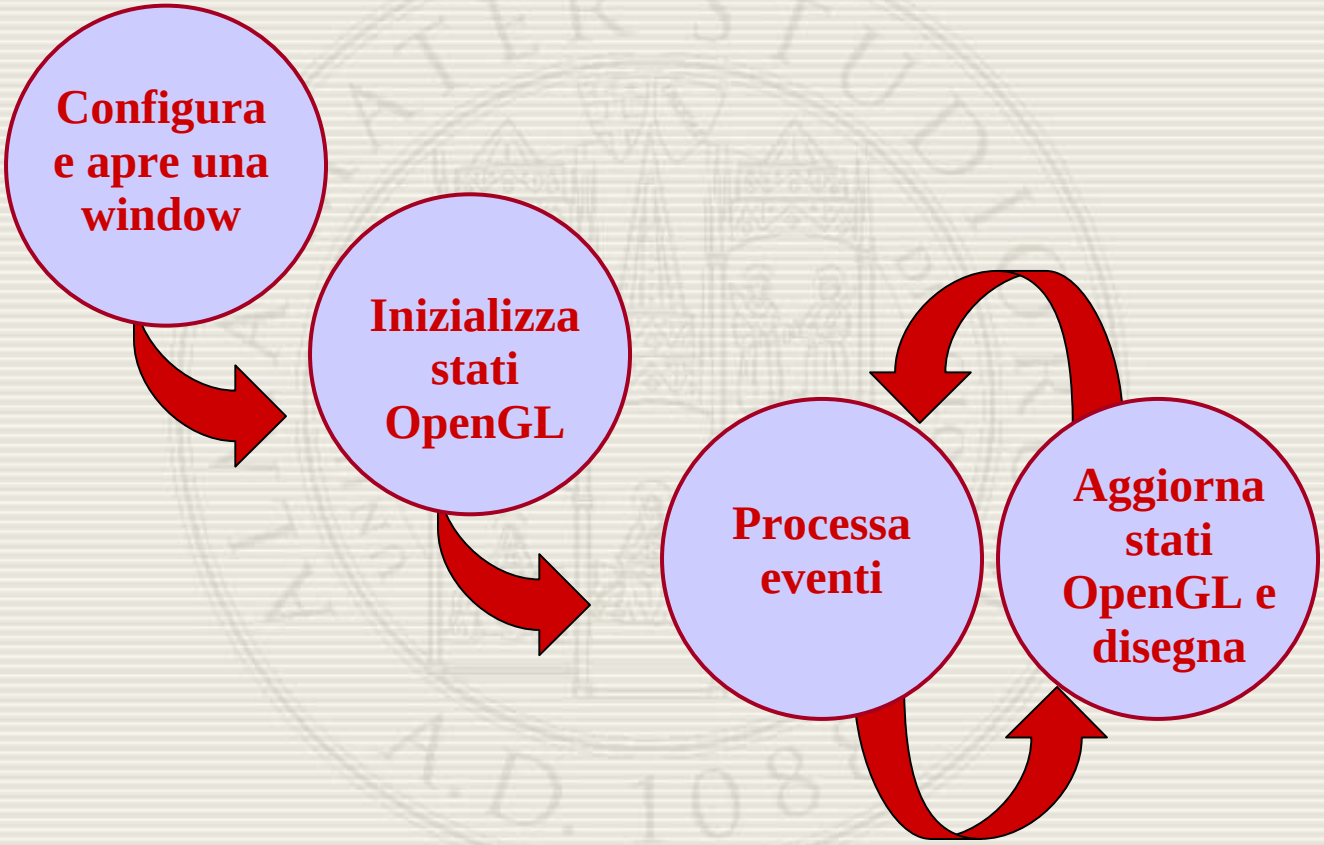
Architettura Pipeline di OpenGL



Architettura PipeLine di OpenGL



Struttura di un programma OpenGL



Introduzione alla programmazione in OpenGL

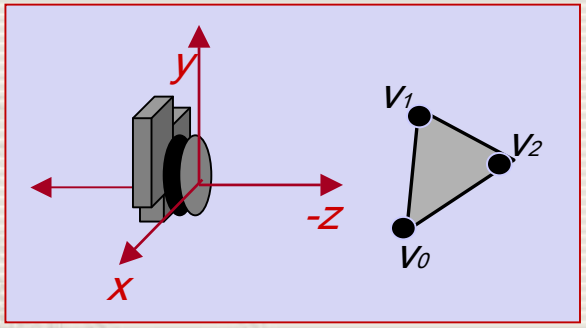
```
//#include <X11/Xlib.h>
 #include <GL/glx.h>
 #include <GL/gl.h>
 #include <GL/glu.h>

  void fun( void )
  {
  }

  void main( )
  {
  }
```

Header Files

Esempio di programma GL




```
Header Files
void main( )
{
   Inizializza_e_Apri_Window ( );
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
   glBegin(GL_TRIANGLES);
      glVertex3f(0.25, 0.25, -0.5);
      glVertex3f(0.75, 0.25, -0.75);
      glVertex3f(0.75, 0.75, -0.35);
   glEnd( );
   glFlush( );
   Aggiorna_Window_e_Controlla_Eventi ( );
}
```

Inizializza stati OpenGL: colore, proiezione, vista, ...

Definisce geometria 3D

Disegna

Esempio di Makefile

```
LIB = –lGL –L/usr/X11R6/lib  –lXext –LX11 -lm
CC = gcc


example: example.c
        $(CC) $(LIB) example.c  –o example
```

Librerie dipendenti dal sistema operativo:

libX11.a (UNIX)

libGL.so (UNIX) **(opengl32.lib per Microsoft Windows)**

Nota: nell'esempio si usano le GL e X11 via GLX

esempio glxsimple.c
(dir opengl/gl_start/glxsimple/)

La Sintassi OpenGL

- **Enumerated types:**

  OpenGL definisce per compatibilità tra piattaforme numerosi tipi per le variabili (**GLfloat, GLint,** ...)

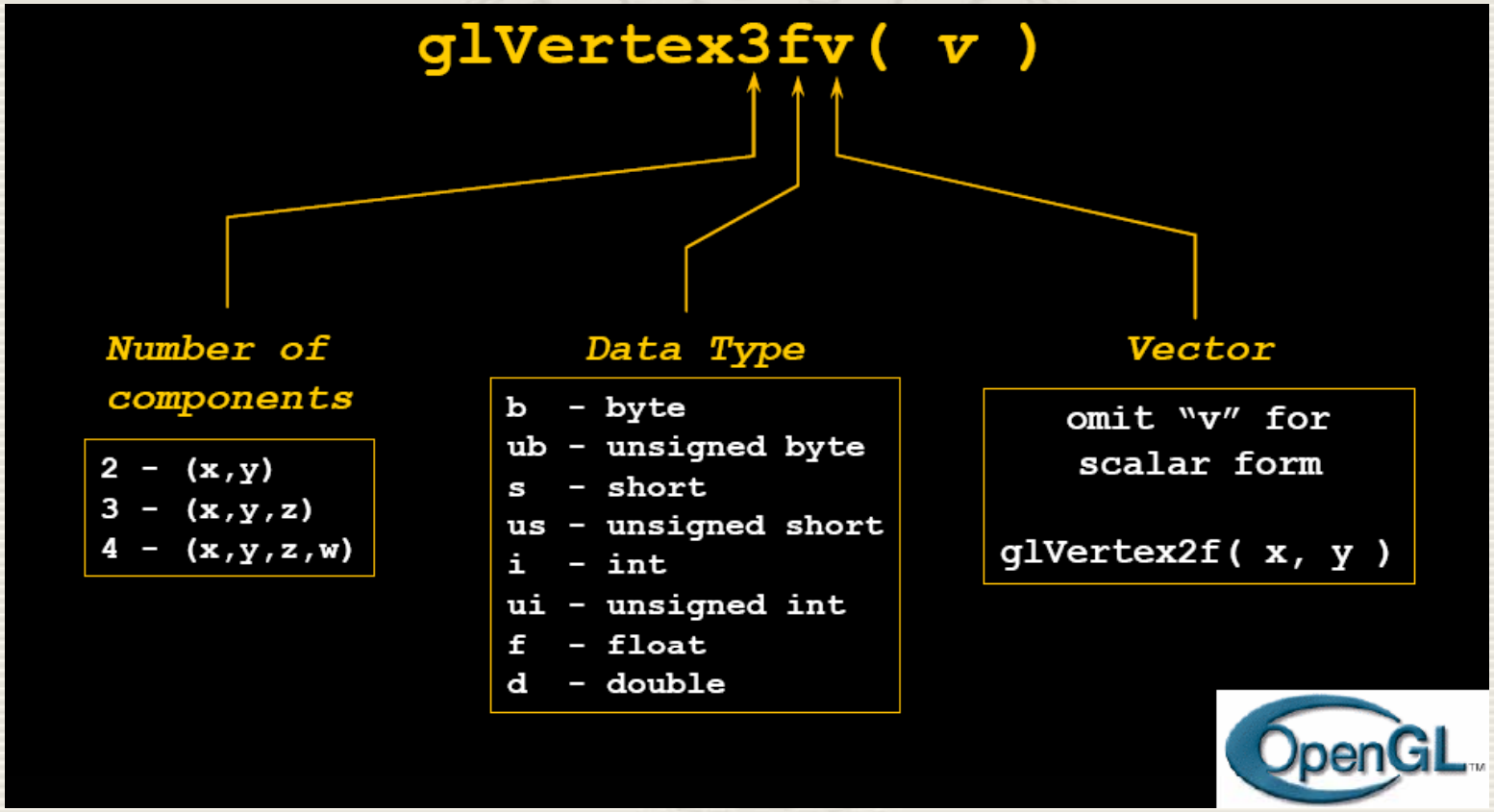
- Prefisso **gl** per i comandi;
- Prefisso **GL_** per le costanti predefinite;

.gl_____2(3)(4){sifd}[v] comandi per definizione dati, impostazione colori, ecc. (vedi esempi)

Esempio:

```
glColor3f(1.0, 1.0, 1.0);
glVertex3f(0.25, 0.75, 0.0);
glVertex3fv(p);
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

Formato dei Comandi OpenGL: esempio



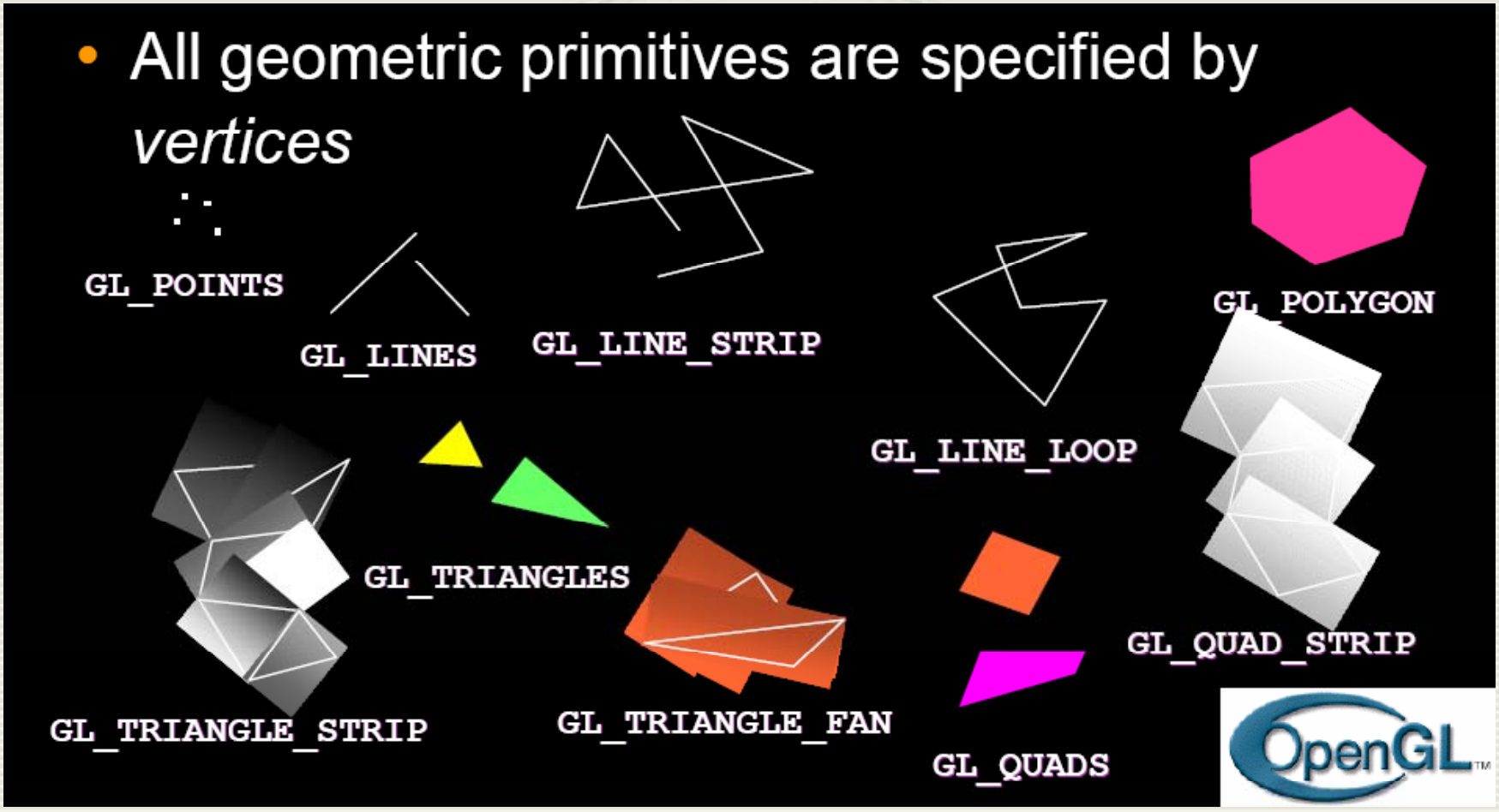
Specificare le Primitive Geometriche

```
glBegin( primType );
        glColor3fv( );
        glVertex3fv(  );
        ...
        ...
glEnd( );
```

primType: GL_POINTS, GL_LINES,
GL_TRIANGLES,
GL_QUADS, ecc.

Primitive Geometriche in OpenGL

- All geometric primitives are specified by *vertices*



Il Colore in OpenGL

RGB (True Color) **glColor…( );**

oppure

COLOR INDEX (Colormap) **glIndex…( );**

Nota:

le componenti colore RGB sono valori GLfloat in [0,1]

Specificare il colore

```
glBegin( primType );
        glColor3f(1.0, 0.0, 0.0 );
        glVertex3fv(  );
        …
        …
glEnd( );
```

Nota: ogni vertice ha un colore (**glColor3f( )**) e ogni punto interno ad un poligono ha un colore dato dall'interpolazione del colore dei vertici della primitiva.

Display LIST

- **Modalità grafica immediata**

– Ogni primitiva è inviata alla scheda grafica che la elabora in pipeline e la visualizza

– Per ridisegnare la stessa primitiva occorre inviarla nuovamente

- **Modalità display list**

– Le primitive vengono memorizzate in una display list sul server grafico (scheda grafica)

– Ad ogni display list viene associato un nome (identificatore numerico)

– Possono essere rivisualizzate con il contesto grafico corrente (default) o cambiarlo

Gestire il Display List

- **Creare un display list**

```
GLuint id;
void init( void )
{
  id = glGenLists( 1 );
  glNewList( id, GL_COMPILE );
// other OpenGL routines
   ……
  glEndList( );
}
```

- **Chiamare una lista creata**

```
void display( void )
{
  glCallList( id );
}
```

Effetto Display List

- L'effetto della chiamata al display list disegna il contenuto del display list utilizzando il contesto grafico corrente

-Oltre a **GL_COMPILE** si può usare

GL_COMPILE_AND_EXECUTE

per memorizzare e disegnare direttamente

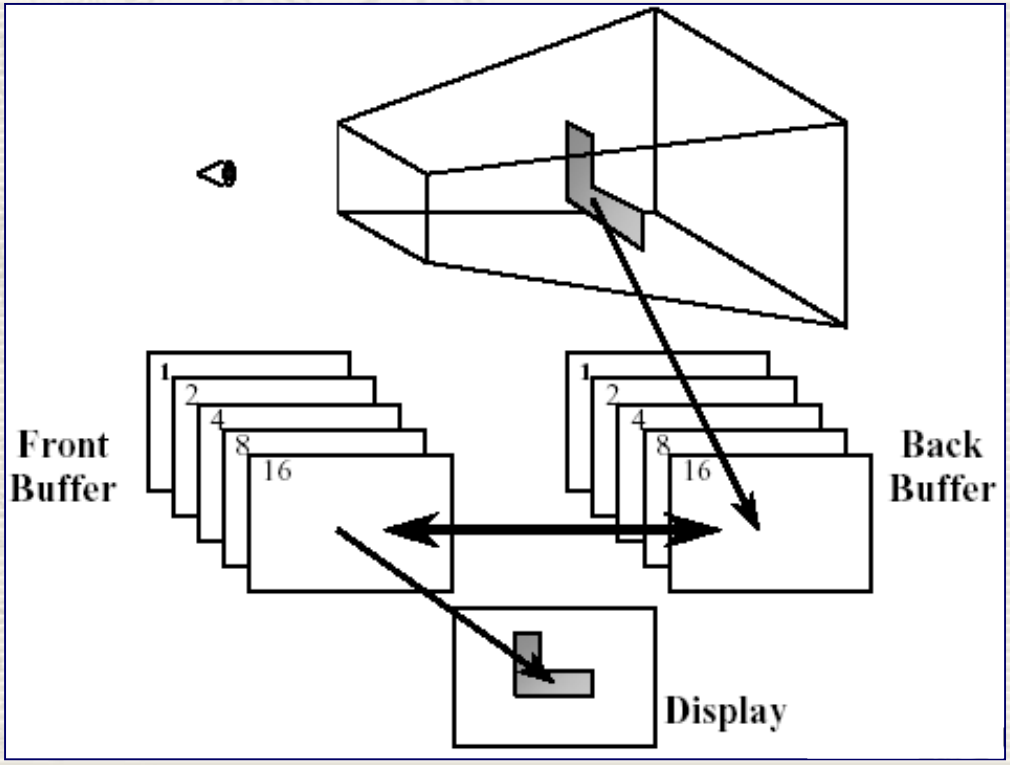
- Non tutte le function OpenGL possono essere memorizzate in un display list (per es. **glFlush,…**), vengono altrimenti disegnate in modalità immediata

Animazione con Double Buffering

Il color buffer (**Frame-Buffer**) viene diviso in due parti:
Front buffer e **Back buffer**

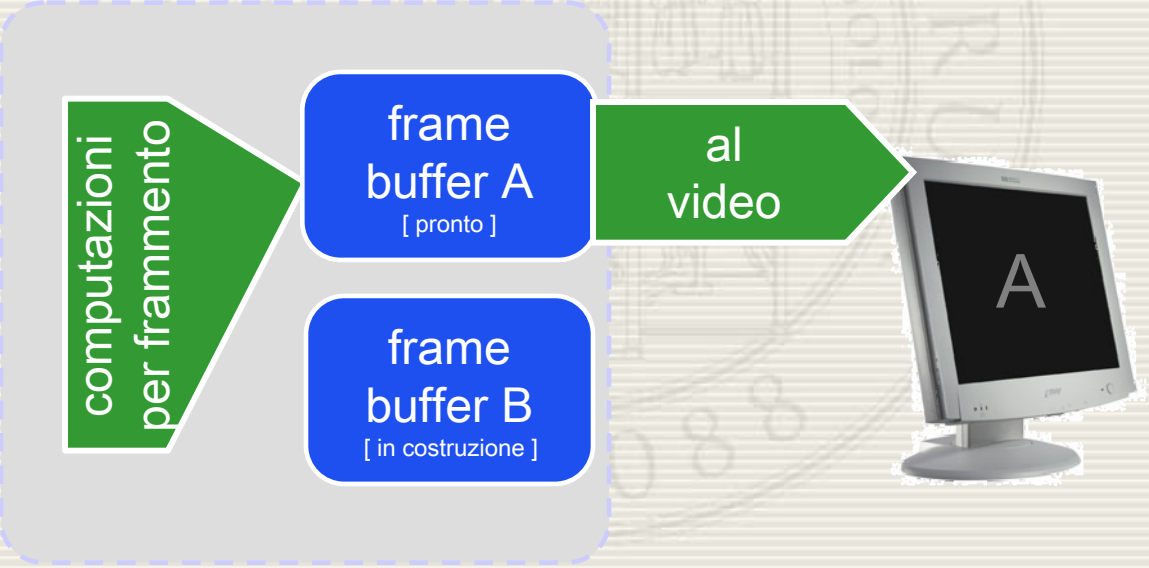
L'applicazione visualizza il contenuto del Front buffer mentre si disegna il prossimo frame nel Back buffer.

Quindi un opportuno segnale permette lo scambio dei ruoli dei due buffer.



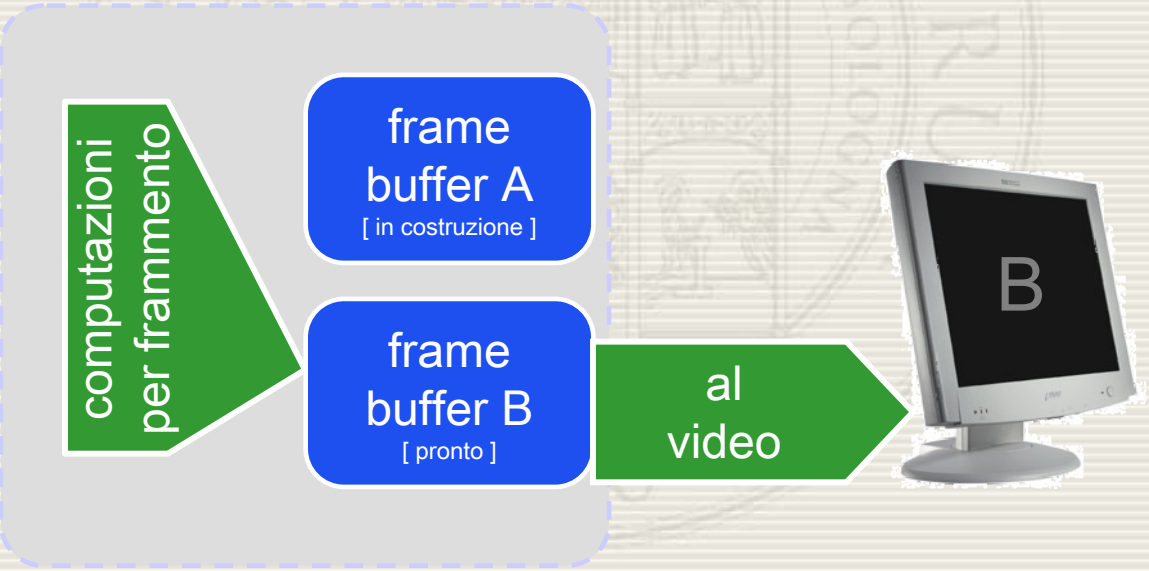
Animazione con Double Buffer

Tecnica per nascondere il frame buffer mentre viene riempito



Animazione con Double Buffer

Tecnica per nascondere il frame buffer mentre viene riempito



Gestire il Double Buffer con GLX

- **Controllare se il double buffer è presente**

```
vi = glXChooseVisual( dpy, DefaultScreen(dpy),
                      GLX_DOUBLEBUFFER );
```

- **Chiamare la funzione**

```
glXSwapBuffers( dpy, win );
```

demo glxsimple_dl.c
(dir opengl/gl_start/glxsimple)

La GLUT Library

Struttura di un'applicazione GL con GLUT:

- Aprire e configurare finestre
- Inizializzare lo stato OpenGL
- Registrare le callback function necessarie
  - render, resize, input keyboard, mouse,…
- Entrare nel ciclo degli eventi

Esempio di programma con GL e GLUT

```
void main(int argc, char **argv )
{

   glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE );
   glutCreateWindow( argv[0] );
   init( );
   glutDisplayFunc( display );
   glutReshapeFunc( resize );
   glutKeyboardFunc( keyb );
   glutMouseFunc( mouse );
   glutIdleFunc( idle );
   glutMainLoop( );
}
```

Vedi più avanti

Inizializzazione OpenGL

Inizializzazione degli **stati OpenGL** che verranno utilizzati dall'applicazione

```
void init ( void )
{
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );
  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_DEPTH_TEST );
}
```

Rendering Callback 1 / 4

```
void display ( void )
{
   glClear( GL_COLOR_BUFFER_BIT );
   glBegin( GL_TRIANGLE_STRIP );
      glVertex3fv( v[0] );
      glVertex3fv( v[1] );
      glVertex3fv( v[2] );
      glVertex3fv( v[3] );
   glEnd();
   glutSwapBuffers();
}
```

Reshape Callback 2 / 4

```
void resize( int w, int h )
{
  glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity( );
  gluPerspective( 65.0, (GLfloat) w / h,1.0, 100.0 );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity( );
  gluLookAt( 0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

Rendering Callback 3 / 4

Elabora l'input utente da dispositivi: per esempio tastiera

```
void keyb ( char key, int x, int y )
{
  switch( key )
   {
    case 'q' : case 'Q' :
            exit( EXIT_SUCCESS );
            break;
    case 'r' : case 'R' :
             rotate = GL_TRUE;
    break;
   }
}
```

Rendering Callback 4 / 4

Elabora l'input utente da dispositivi: per esempio mouse

```
void mouse ( int btn, int state, int x, int y )
{
  if (btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
   {
    .....
   }
  if (btn==GLUT_MIDDLE_BUTTON && state==GLUT_DOWN)
   {
    .....
   }
if (btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
   {
    .....
   }
}
```

Rendering Callback 5 / 4

Per gestire un'animazione

```
void idle ( void )
{
  t += dt;
  glutPostRedisplay( );
}
```

Esempio di Makefile

```
LIB = –lglut –lGLU –lGL  -lm

CC = gcc

triang: triang.c
        $(CC) $(LIB) triang.c –o triang
```

Animazione usando Double Buffer con GLUT

1. Richiesta di utilizzo di double buffer FB

   glutInitDisplayMode (GLUT_RGB | GLUT_DOUBLE);

2. Pulire il Frame Buffer

   glClear (GL_COLOR_BUFFER_BIT);

3. Resa della scena
4. Richiesta di swap dei buffer front e back

   glutSwapBuffers( );

5. Ripetere i passi 2 - 4 per l'animazione

Demo glutsimple_orig.c
(dir opengl/gl_start/)

Il Colore in OpenGL e GLUT

Abbiamo già detto:

RGBA (True Color) **glColor…( );**

oppure

COLOR INDEX (Colormap) **glIndex…( );**

Con **GLUT**, si usa **glutInitDisplayMode( )** per specificare:

una window RGBA (usando **GLUT_RGBA**),

una window color index (usando **GLUT_INDEX**).

RGBA

A (di **RGBA**) sta per **Alpha** ed è una quarta componente colore.

Misura l'opacità del pixel a cui è associato:

valori da 0 (trasparente) a 1 (completamente opaco)

– Simula gli oggetti traslucidi: vetro, acqua,…

– Composizione (sovrapposizione) di immagini

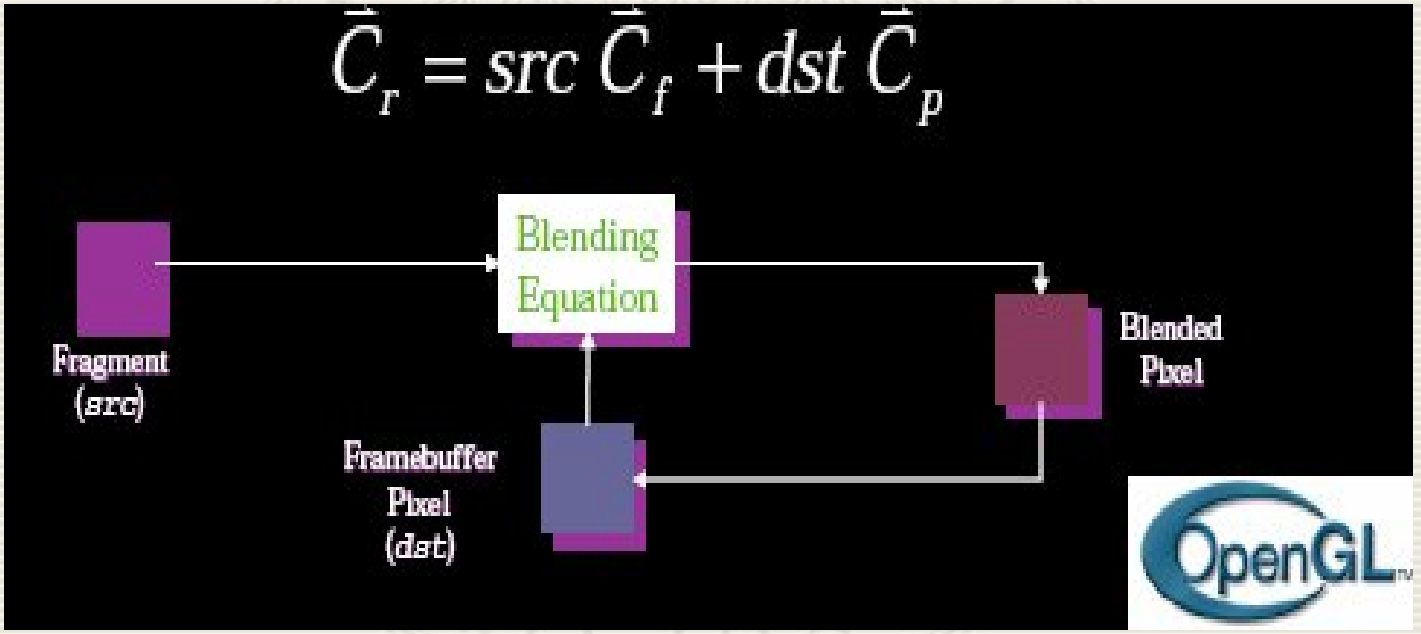
– Antialiasing di primitive geometriche

E' ignorato se non è abilitato il blending

glEnable(GL_BLEND)

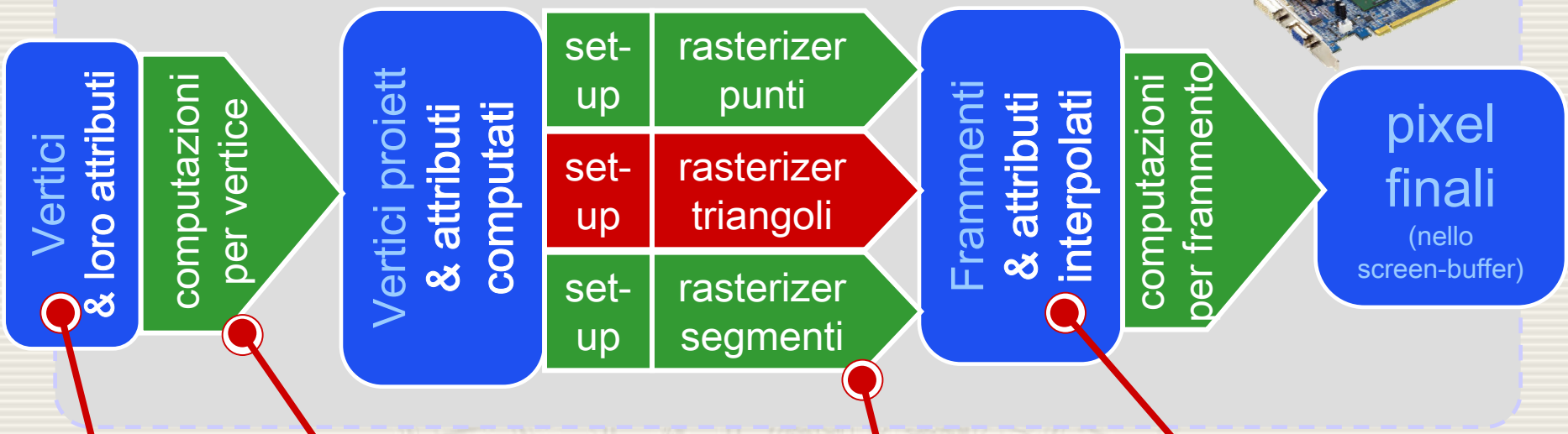
Blending

Combina i fragment con il valore dei pixel che sono già nel frame buffer secondo la seguente formula:



glBlendFunc(src, dst)

Attributi nella pipeline



associamo degli *attributi* ai vertici
es: colore RGB

gli attributi possono subire varie compuatzioni

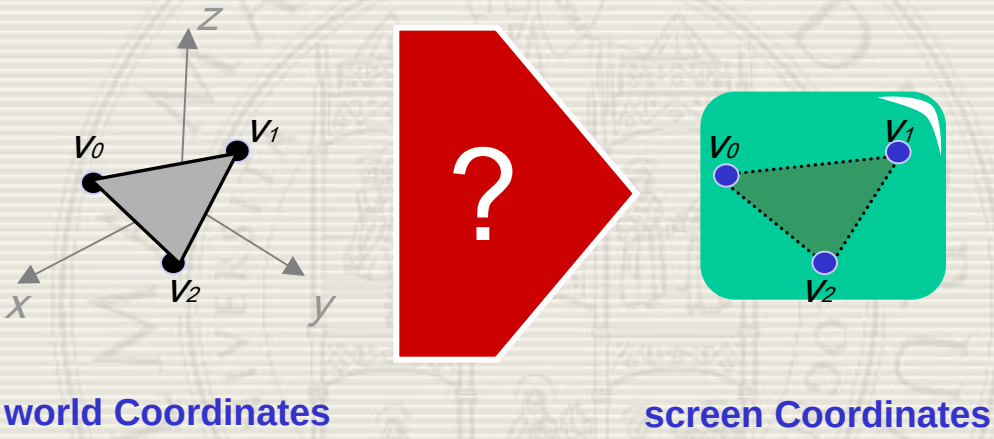
gli attributi vengono interpolati

ogni frammento avrà una valore interpolato degli attributi per vertice

Tutorial shapes.c
(gltutorials)

Transformazioni in OpenGL

- Per ogni vertice:



world Coordinates

screen Coordinates

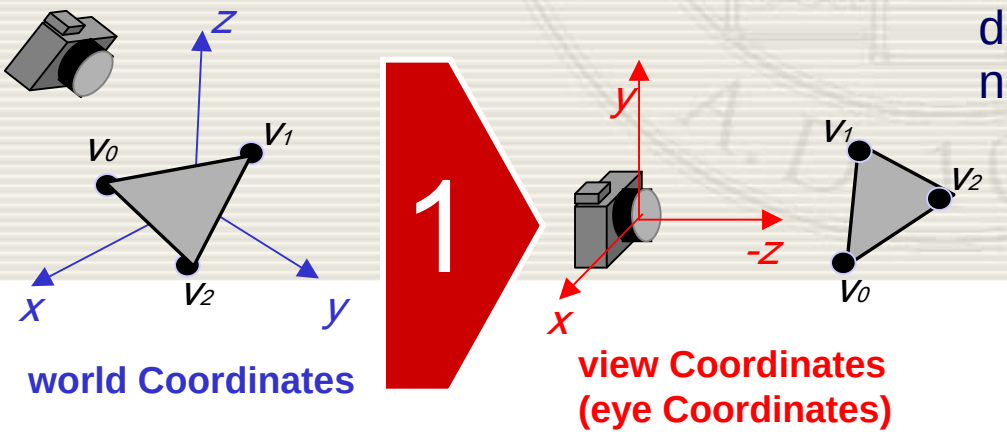
Transformazioni in OpenGL

1) "transformazione di vista": cambio di sistema di riferimento

ora la geometria è espressa in un sistema di coordianate in cui:

- l'origine è il centro di proiezione (obiettivo della camera)
- la camera guarda verso *-z*
- *y* è verso l'alto, e *x* è verso destra (rispetto all'osservatore)

Nota: OpenGL definisce il sistema di riferimento dell'Osservatore ad essere destrorso con direzione di vista l'asse z negativo.

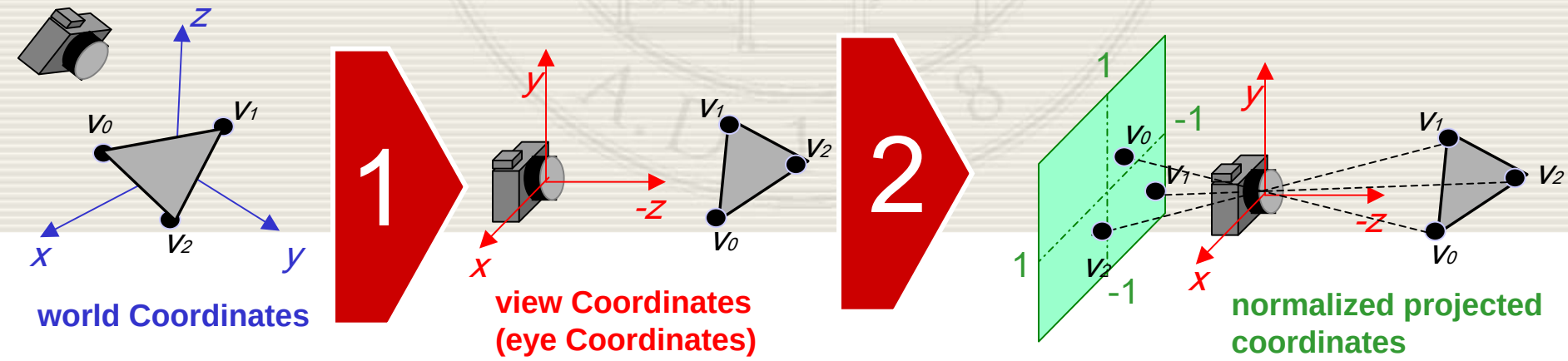


Transformazioni in OpenGL

1) "transformazione di vista":
cambio di sistema di riferimento

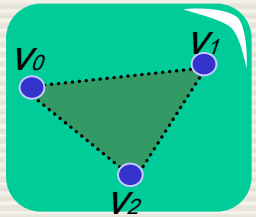
2) "transformazione di proiezione":
proietta la geometria sul piano
di proiezione

- è necessario sapere i parametri della "camera virtuale"
  - **in particolare, l'apertura angolare,**
  - **il view-up vector, ecc.**

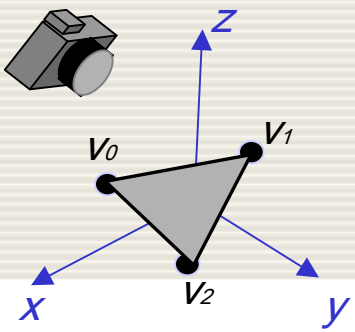


Transformazioni in OpenGL

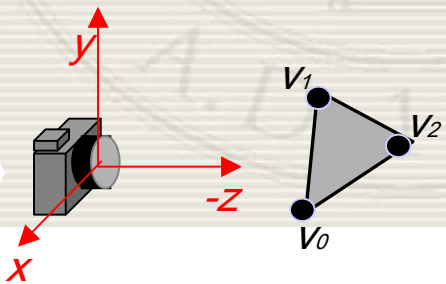
1) "transformazione di vista":
   cambio di sistema di riferimento
2) "transformazione di proiezione":
   proietta la geometria sul piano
   di proiezione
3) " transformazione viewport":
   da [-1,1] x[-1,1] a [0..res$_x$]x[0..res$_y$] (pixels)



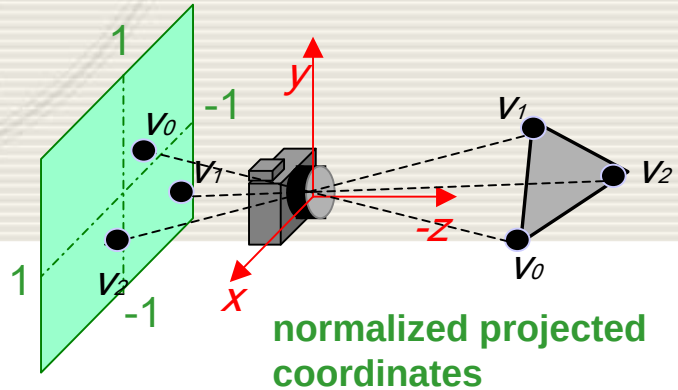
screen Space



world Coordinates

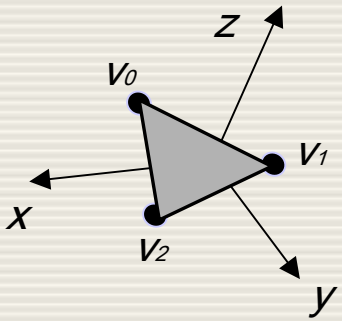


view Coordinates
(eye Coordinates)



normalized projected
coordinates

Transformazioni in OpenGL



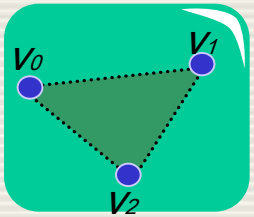
object Coordinates

0) transformazione di modellazione

1) transformazione di vista

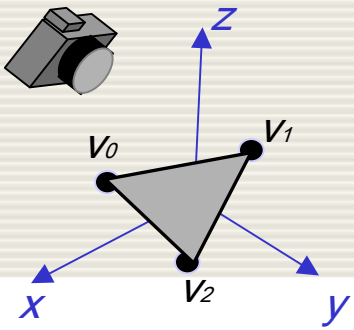
2) transformazione di proiezione

3) transformazione di viewport



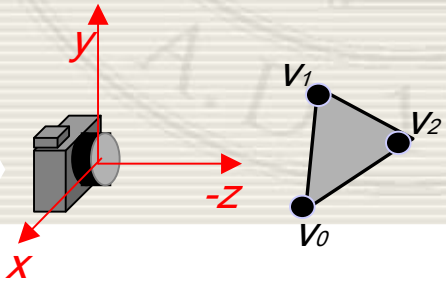
screen Space

0



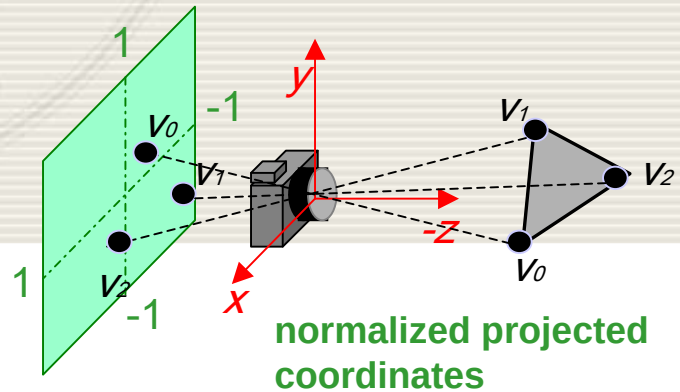
world Coordinates

1



view Coordinates
(eye Coordinates)

2



normalized projected
coordinates

3

Trasformazioni in OpenGL

- **Trasformazioni di modellazione**
  - **Muovere il modello**
- **Trasformazioni di vista**
  - **La posizione ed orientamento della telecamera definiscono il volume di vista nel mondo**
- **Trasformazioni di proiezione**
  - **Scatto della macchina fotografica**
- **Trasformazione Window-Viewport**
  - **Allargare o ridurre la fotografia fisica**

Trasformazioni in OpenGL

Ogni trasformazione può essere considerata come un cambio di rappresentazione di un vertice da un sistema di coordinate ad un altro.

- **Conversione da coord. Mondo a coord. Camera**

  ModelView matrix: Trasformazioni di modellazione e Trasformazioni di vista

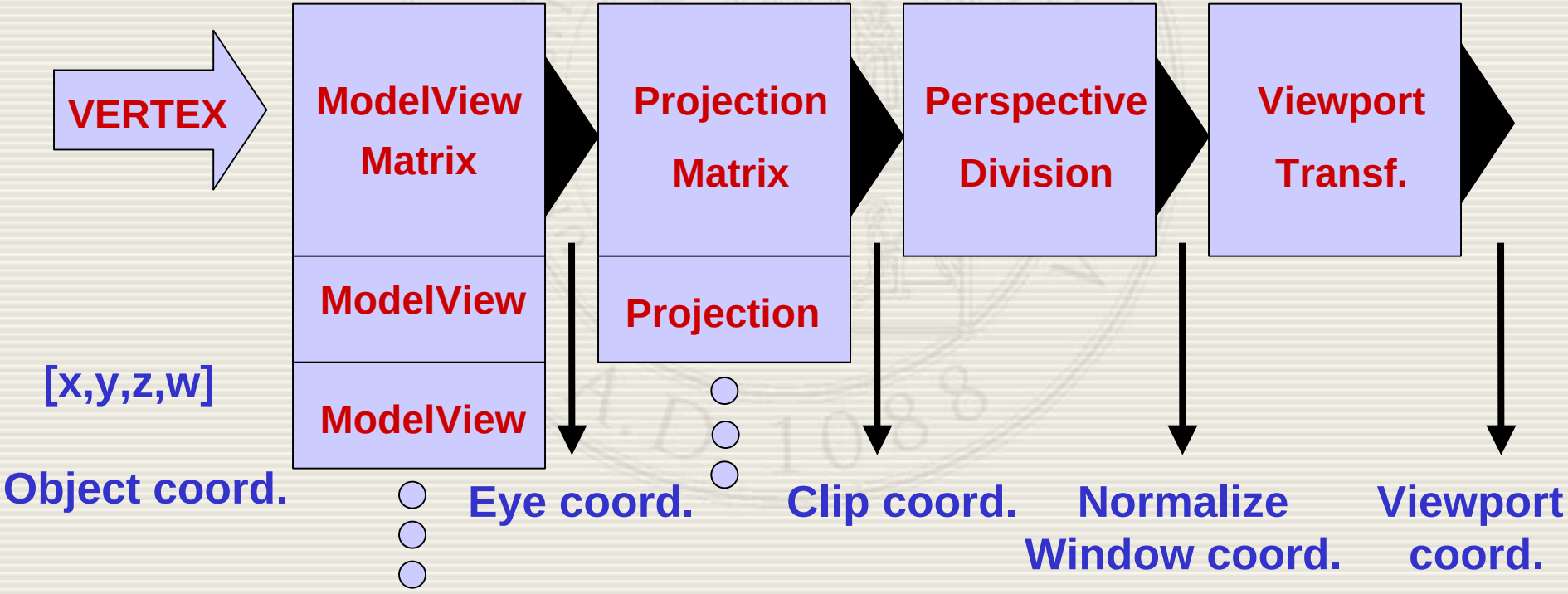
- **Conversione da coord. Camera a coord. Window**

  Projection matrix: Trasformazioni di proiezione

- **Conversione da coord. Window a coord. Viewport (coord. 2D espresse in pixel) viene realizzata automaticamente da OpenGL**

Trasformazioni in OpenGL

Le trasformazioni elementari sono rappresentate da matrici che vengono composte per moltiplicazione e memorizzate in un opportuno stack detto **CTM** (Current Transformation Matrix).



Gestione Trasformazioni

OpenGL fornisce uno **stack** di matrici per ogni tipo di matrice supportata (**ModelView** e **Projection**)

- Ci sono due modi per definire una trasformazione:

– Specificando le matrici (4x4)

glLoadMatrix(*pointer-to-matrix*)

glMultMatrix(*pointer-to-matrix*)

– Specificando l'operazione

glRotate, glTranslate, glScale, glOrtho, …

Trasformazioni in OpenGL

La definizione della trasformazione va eseguita **PRIMA** che l'oggetto venga visualizzato; in generale:

```
glMatrixMode(GL_MODELVIEW); //seleziona stack GL_MODELVIEW
glLoadIdentity( ); //definisce matrice identità I
.....                                                    M=I
glTranslatef( dx, dy, dz ); // matrice C;              M=C*M=C*I
glScalef( sx, sy, sz );  //matrice B;                  M=B*M=B*C*I
glRotatef( angle, rx,ry,rz );  //matrice A;            M=A*M=A*B*C*I
glBegin( );
  glVertex3fv( P );        // vertice P                 P*M
glEnd( );
```

$$P' = P*M = (((P*A)*B)*C)$$

Tutorial transformations.c

(gltutorials)

ModelView Matrix

- Le trasformazioni di modellazione e di vista sono controllate agendo sulla stessa matrice **Modelview**
- Si definiscono prima le trasformazioni di vista (che agiscono sull'intera scena) e poi quelle di modellazione (sono cioè date in ordine inverso)
- **Modelview** e **Projection** matrix vengono poi concatenate insieme per formare una matrice che viene applicata a tutti i vertici nella scena.

Trasformazione di Vista

Posizionare la telecamera nella scena specificando direttamente posizione ed orientamento.

Per non specificare una sequenza di rotazioni e traslazioni si consiglia la funzione della **GLU**:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity( );
gluLookAt( eyex, eyey, eyez, aimx, aimy, aimz, upx, upy, upz );
```

sposta la telecamera nel punto di osservazione della scena (**eyex,eyey,eyez**), considera (**aimx, aimy, aimz**) come centro della scena verso il quale è rivolto lo sguardo, e (**upx,upy,upz**) è il vettore che determina l'alto della camera.

glBegin( ); … glEnd( );

Trasformazioni di proiezione

glMatrixMode(GL_PROJECTION)

- Le successive definizioni di matrici o trasformazioni agiranno sulla matrice **Projection**,
- Proiezioni prospettiche

gluPerspective(fov, aspect, zNear, zFar);

glFrustum(left, right, bottom, top, zNear, zFar);

- Proiezioni ortografiche o parallele

glOrtho(left, right, bottom, top, zNear, zFar);

Nota: per default si ha proiezione ortogonale, ossia non obliqua

Altre Operazioni su Stack

glLoadIdentity( )

Pone la matrice identità in cima allo stack corrente;

glPushMatrix( )

glPopMatrix( )

Permettono rispettivamente di salvare la matrice in cima allo stack (portandola in seconda posizione) e di caricare la matrice, seconda nello stack, in prima posizione.

Lo stack **Modelview** ha almeno 32 posizioni, lo stack **Projection** almeno 2.

Tutorial projection.c

(gltutorialss)

demo

glutsimple_load.c

glutsimple_mult.c

(dir opengl/gl_start/glutsimple_mat/)

Clipping e Culling

- Occlusion culling

  non si vede... perché coperto da qualcos'altro
- View-frustum culling

  non si vede... perché è fuori dal frustum di vista
- Backface culling

  non si vede... perché è la parte interna di una superficie chiusa
- Importance culling

  (quasi) non si vede... perché la sua proiezione è troppo piccola rispetto alla scena

Notazione

- Viene chiamato **culling** se si scartano intere *primitive* o interi gruppi di primive
- Se una primitiva viene *spezzata* in una parte visibile e una no, viene chiamato **clipping**
- Se è un frammento ad essere scartato, si chiama **testing per frammento**

Tipi di Clipping e Culling

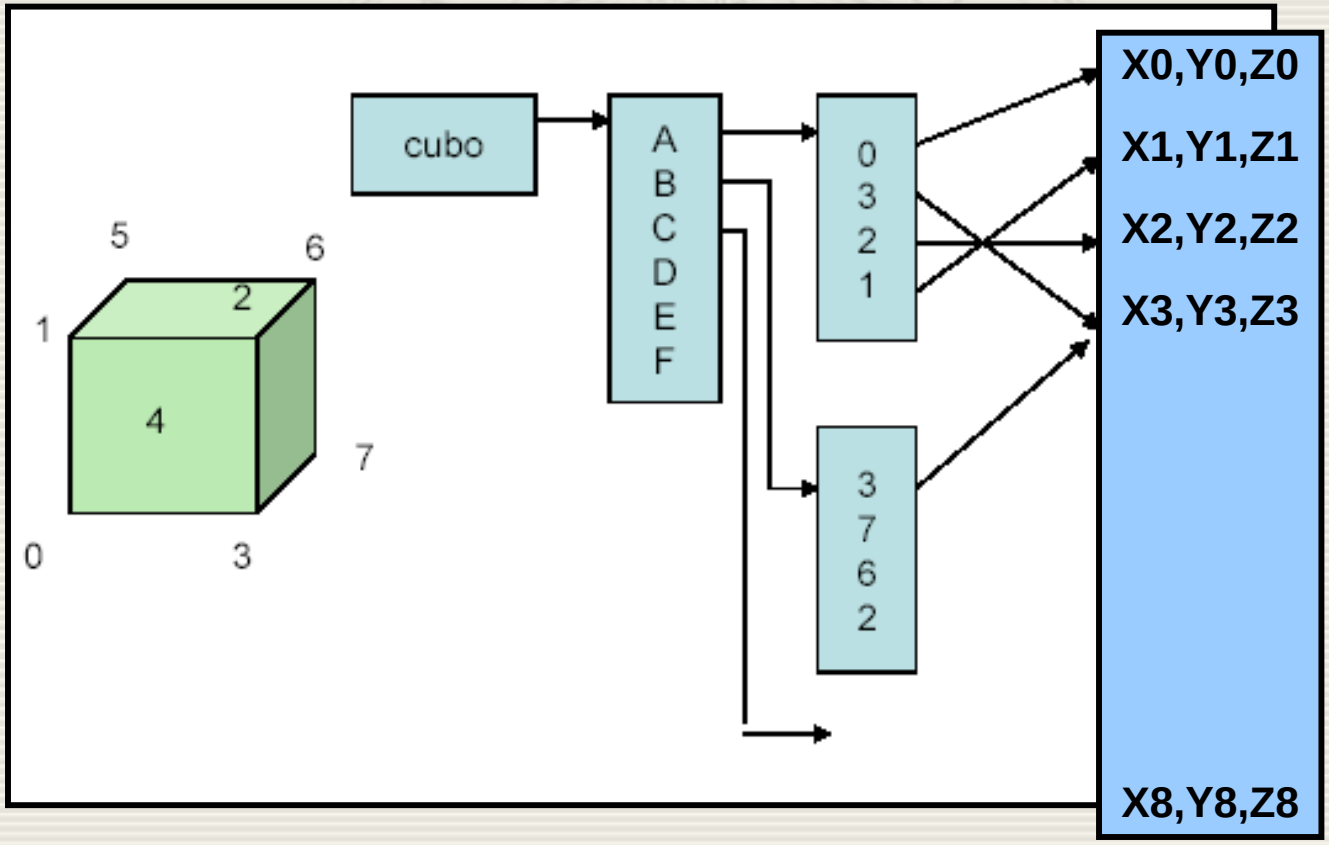
- Eseguiti dall'hardware
  - automatici
  - molto efficienti (overhead minuscolo)
  - ma scartano tardi, e una primitiva alla volta

- Eseguiti dall'applicazione (SW)
  - richiedono algoritmi e strutture dati
  - meno efficienti (overhead anche grande)
  - ma scartano presto, e a gruppi

Definiamo un Cubo 1 / 3

Cubo come oggetto a 6 facce e 8 vertici.

Distinguiamo tra geometria e topologia.



Definiamo un Cubo 2/3

Definiamo i vertici del cubo:

```
GLfloat vertices[ ][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
                         {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,1.0,1.0},
                         {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

Oppure

```
Typedef Glfloat point3[3];
point3 vertices[8] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
                      {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
                      {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

Definiamo un Cubo 3 / 3

Definiamo le facce del cubo, per esempio una è:

```
void polygon(int a, int b, int c, int d)
{
   glBegin(GL_POLYGON);
     glVertex3fv(vertices[a]);
     glVertex3fv(vertices[b]);
     glVertex3fv(vertices[c]);
     glVertex3fv(vertices[d]);
  glEnd( );
}
```

per es. chiamata:

polygon(0, 3, 2, 1)

demo cubeview.c
(dir opengl/opengl_prg/opengl_prg1)

FRONT e BACK di una faccia

Ogni poligono ha 2 facce; una **INTERNA** ed una **ESTERNA**: come le possiamo differenziare?

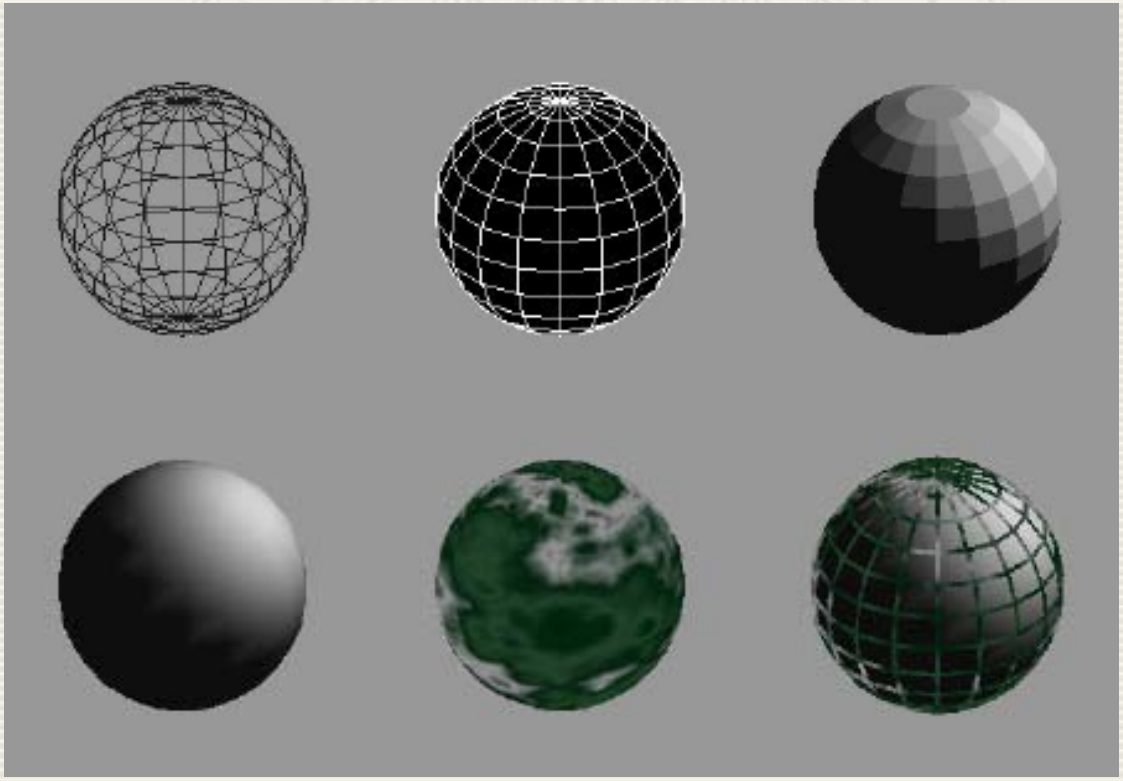
glPolygonMode(GLenum face, Glenum mode);

face: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK

mode: GL_POINT, GL_LINE, GL_FILL

Tecniche di Rendering

OpenGL è in grado di rendere un oggetto utilizzando un semplice **wireframe**, fino ad un sofisticato **texture mapping**.

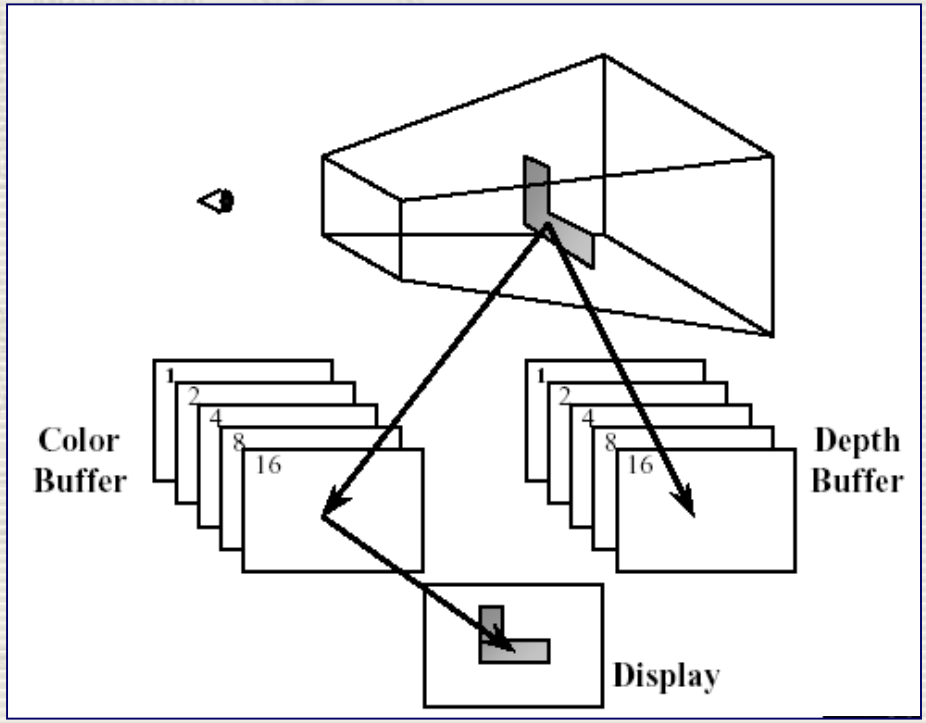


Hidden Surface Removal usando Depth Buffering (Z-buffer)

E' la tecnica usata per determinare quali primitive nella scena sono nascoste da altre e quindi non visibili da un certo punto di vista. Con il depth buffering abilitato, prima di disegnare un pixel sul Frame-Buffer lo si confronta con il depth value di quella locazione nel depth buffer.

Algoritmo di depth buffer (Z-buffer):

```
DepthBuffer(:,:)->z=far clipping plane;
if ( pixel->z < DepthBuffer(x,y)->z )
 {
   DepthBuffer(x,y)->z = pixel->z;
   ColorBuffer(x,y)->color = pixel->color;
 }
```



pixel->z è la distanza del vertice dal piano di vista. Si usa un depth buffer con valori in [0,1], dove 1 indica infinitamente lontano dal punto di vista.

Depth Buffering

- Richiesta di un depth buffer

  `glutInitDisplayMode( GLUT_RGB |GLUT_DOUBLE | GLUT_DEPTH );`

- Abilita il depth buffering

  `glEnable( GL_DEPTH_TEST );`

- Pulisce il depth buffer (lo inizializza a z-far-plane) e il color buffer

  `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );`

- Resa della scena

- Swap color buffers

Un programma di aggiornamento 1/3

```
void main( int argc, char** argv )
{
glutInit( &argc, argv );
glutInitDisplayMode( GLUT_RGB |GLUT_DOUBLE | GLUT_DEPTH );
glutCreateWindow( “Tetrahedron” );

init( );
glutIdleFunc( idle );
glutDisplayFunc( display );
glutMainLoop( );
}
```

Un programma di aggiornamento 2/3

```
void init( void )
{
/* Abilita il Depth-Buffer */
    glEnable(GL_DEPTH_TEST);
/* Background color per il color buffer: blu */
    glClearColor( 0.0, 0.0, 1.0, 1.0 );
}

void idle( void )
{
/* Richiesta di una nuova resa della scena */
    glutPostRedisplay( );
}
```

Un programma di aggiornamento 3/3

```
void display( void )
{
GLfloat vertices[ ] = { … };
GLfloat colors[ ] = { … };
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glBegin( GL_TRIANGLE_STRIP );
    /* chiamate a glColor*( ) e glVertex*( ) */
glEnd( );
glutSwapBuffers( );
}
```

Nota: si ricorda che ogni vertice ha un colore (glColor( )) e ogni punto interno ad un poligono ha un colore dato dall'interpolazione del colore dei vertici della primitiva.

Sitografia

Nella pagina Web del corso si consultino i siti OpenGL citati:

www.opengl.org (sito ufficiale)

OpenGL Links

Tutorials by Nate Robins

NEonHElium Productions

Ecc.